Deep Learning in Agriculture: Exploring Food Production & Classifying Wild Edible Plants

UNIVERSITY OF
LINCOLN

Ryan Partridge
PAR18686962

18686962@students.lincoln.ac.uk

School of Computer Science

College of Science

University of Lincoln

Submitted in partial fulfilment of the requirements for the
Degree of BSc(Hons) Computer Science

Supervisor Dr. Vassilis Cutsuridis

April 2021

Acknowledgements

I would like to thank my dissertation supervisor **Dr. Vassilis Cutsuridis**, whose guidance and feedback has been invaluable throughout this project. Furthermore, I would also like to thank my personal tutor **Dr. James Brown** for being a great source of support throughout the year.

Abstract

Maintaining a steady flow of food is becoming increasingly difficult. With the global population predicted to reach 9.6 billion by 2050, new food sources are required. The research conducted in this paper shows that Deep Learning (DL) applications can improve the quantity and quality of harvests produced when using disease identification and plant recognition techniques. However, they have not yet been used to identify natural vegetation as a potential food source. This study aims to understand the role DL plays in agricultural food production, expand DLs use within horticulture, and potentially identify new food sources within natural environments for daily consumption. Throughout this paper, various toolsets, machine environments, and research methods are discussed, assisting in determining the best methodology for creating an artefact that identifies wild edible plants. While the artefact focuses on three state-of-the-art Convolutional Neural Networks (CNNs), additional information found within this report includes the components of CNNs, accompanied by the design and development of the artefact itself. The three architectures, GoogLeNet, MobileNet v2, and ResNet-34, were built using the open-source deep learning framework PyTorch, where 36 variants of these models were created and tested using 12 different parameters. The model's performance was evaluated based on six performance metrics to classify 35 classes of wild edible plants, each trained and tested on a dataset containing 16,535 images. Overall, achieving classification accuracies of 74.29%, 82.85%, and 80.35%, for GoogLeNet, MobileNet v2, and ResNet-34, respectively.

Keywords: Agriculture; Convolutional Neural Networks; Plant Classification; Deep Learning.

Table of Contents

1 Introduction **1**

1.1 Aims and Objectives 3

2 Background **5**

2.1 Challenges of Increased Food Demand 5

2.2 Deep Learning in Agriculture 6

2.2.1 Disease Identification 7

2.2.2 Plant Recognition 8

3 Methodology **11**

3.1 Project Management 11

3.2 Software Development 12

3.3 Toolsets and Machine Environments 13

3.4 Research Methods 16

3.4.1 Convolutional Neural Networks 16

A. Convolutional Layer 17

B. Pooling Layer 18

C. Fully-connected Layer 19

D. Activation Function 21

E. Chosen Architectures 22

3.4.2 Artefact Pipeline 24

3.4.3 Performance Metrics 25

4 Design, Development and Evaluation **27**

4.1 Software & Hardware Requirements 27

4.2 Design 29

4.3 Development 31

4.3.1 Initial Implementation 32

4.3.2 Parameter Tuning 36

4.4 Testing 38

4.5 Results 39

5 Conclusions **45**

6 Reflective Analysis **47**

References **49**

A CNN Architectures **59**

List of Figures

1.1 35 Classes Of Wild Edible Plants 3

3.1 Gantt Chart 11
3.2 Trello Board 12
3.3 Feedforward Neural Network (FNN) 16
3.4 Convolutional Operation 18
3.5 Max Pooling Operation 19
3.6 MobileNet v2 Architecture Block Components 23
3.7 Machine Learning Pipeline 24

4.1 Class & Notebook Topology 30
4.2 Example Classifier Structure 34
4.3 Best Model's Learning Curves 38
4.4 Best Model's ROC Curves 42
4.5 Best Model's Confusion Matrices 44

A.1 ResNet-34 Architecture 59
A.2 GoogLeNet Architecture 60

List of Tables

4.1 Python Libraries 28
4.2 GoogLeNet Architecture Results 39
4.3 MobileNet v2 Architecture Results 40
4.4 ResNet-34 Architecture Results 41
4.5 Best Model's Results 41

Chapter 1

Introduction

Agriculture plays a critical role in the global economy. Unfortunately, it is currently pressured by a growing population that requires immediate attention. When technology was first integrated into agriculture more than one century ago, with the first tractor in 1913 (Santos et al., 2020), there was a large increase in food production. Nowadays, mechanical technology has dramatically impacted agriculture, improving its efficiency (Schmitz and Moss, 2015). However, it may not be enough to sustain the increasing global population predicted to reach 9.6 billion by 2050, increasing by approximately 1.8 billion people. This massive increase will require an immense demand for new food (Gikunda and Jouandeau, 2019), making further agricultural advances essential.

Several studies on a concept known as smart farming use a combination of technologies, such as the Internet of Things (IoT), Big Data, and robotics, and plays a crucial step in increasing the sustainability and reliability of food production while simultaneously reducing its environmental impact (Walter et al., 2017). This modern approach assists in tackling the key issues facing agriculture and expands future agricultural advances. It focuses on DL tools, specifically CNNs, for identifying patterns in images, which have now become fundamental in agriculture (Liakos et al., 2018), aiding in the identification of plants and diseases through image recognition.

However, the applications of smart farming are limited. It solely focuses on existing farms and currently consumed crops, despite the existence of over 50,000 edible plants worldwide, with only 15 of them providing 90% of the world's food energy intake (National Geographic Headquarters, Undated). Examining other edible flora and

considering them as a resource for daily consumption can prove beneficial, assisting in the limitations of future food provisions. Furthermore, with the continued decline in the number of botany students (Drea, 2011; Lauer, 2015), the world's understanding of plants may diminish, demanding the need for new methods to understand the plants around us.

This paper presents a method to efficiently identify edible plants in the wild, using CNN architectures, compensating for smart farmings limitations. Experimenting with three CNN architectures, the GoogLeNet, MobileNet v2, and ResNet-34, their results are compared based on the ability to correctly classify unique wild edible plants, given a plant image. The purpose of this project is to expand the use of DL within horticulture, enabling inexperienced individuals the ability to swiftly recognise wild edible plants within the vast expanse of flora in the world. Opening the opportunity for scientists and individuals interested in horticulture to extract, experiment, and reproduce new types of edible vegetation. Additionally, encouraging future DL research that concentrates on highlighting and obtaining a deeper understanding of plants.

Classifying plants can be challenging due to the wide variety of similarities between them. What distinguishes plants from one another are their features, such as colour, shape, style, and unique properties, e.g. ground ivy has whiskers, and coneflowers have cones on top of their petals. For botany experts, the difficulty with classifying plants lies in finding their distinct features. Unfortunately, this is the same for CNNs, except they identify the features through numeric pixel values.

The set of data used in this paper is a Wild Edible Plants dataset (Partridge, 2021), created specifically for this problem, consisting of Flickr images. The CNN models were trained to recognise 35 types of plants, where each class contains 400 to 500 plant images. Overall, the dataset contains 16,535 images. The plant classes within the dataset are shown in Figure 1.1.

Subsequently, the research questions to be addressed in this paper are as follows:

1. What challenges are faced when overcoming an increase in food demand?
2. What role does Deep Learning play in agriculture?
3. Which Convolutional Neural Network performs the best?

In the remainder of this study, an analysis of related literature is in chapter 2 the methodology used for research in chapter 3; the algorithms design, development, and results in chapter 4; the project's conclusions and summary in chapter 5; and lastly, a reflective analysis of the project in chapter 6.

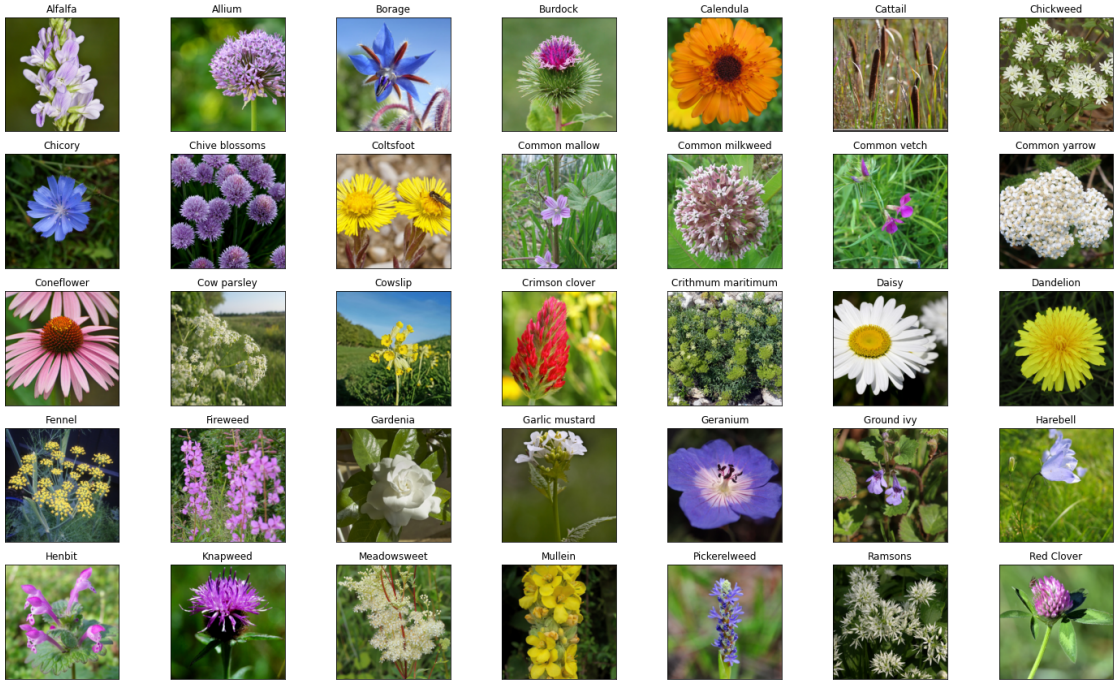

Figure 1.1: The 35 classes of plants used within the Wild Edible Plants dataset.

1.1 Aims and Objectives

With the world population continuing to increase, additional food sources are required. This project aimed to successfully classify different classes of wild edible plants and understand each of their distinct features, allowing easy identification within a large expanse of vegetation. Three CNN architectures were implemented to investigate the unique elements of each plant. The process to achieve this can be divided into multiple objectives.

Objective 1: maximise each model's classification accuracy by creating a dataset containing 35 classes of wild edible plants, where each category has a minimum of 400 images per one. Additionally, employ data augmentation techniques onto the images, increasing the diversity of the dataset.

Objective 2: examine, select, and perform classification with three different CNN architectures and apply transfer learning to each one, bolstering the classification performance to achieve an accuracy of over 80%.

Chapter 2

Background

2.1 Challenges of Increased Food Demand

The global population continues to increase daily, causing an expected increase in food demand, where an estimate of 70% more food needs to be produced by 2050 (Conijn et al., 2011). Unfortunately, the resolution of this food increase requires more than farming additional land for increased fresh produce. With the challenges of resource scarcity and climate change, addressing this food demand becomes extremely difficult.

Place et al. (2013) explains that resource scarcity requires the efficient and sustainable use of natural resources to ensure food security. They explain that successfully mitigating resource scarcity requires available resources to be optimized through its availability, accessibility, utilization, and stability. Additionally, it requires a need for new technologies that assist in increasing physical production and accounting for the sustainable use of resources. Furthermore, Mancosu et al. (2015) details that water is the primary resource affected by scarcity in agriculture When food demand increases, it directly affects agricultural water usage. Following Place et al., they express the need to optimize and streamline the efficiency of water consumption. Otherwise, the effectiveness of irrigating future crops will diminish drastically.

In a report by Adams et al. (1998), they write about the effects of global climate change on agriculture and explain that climate is a critical determinant of agricultural productivity, causing it to affect food production by influencing crops and livestock. Some of the core factors produced by climate change include temperature changes,

precipitation, droughts, floods, windstorms, crop and livestock pests, and soil erosion, varying in frequency and severity. Moreover, Aydinalp and Cresser (2008) express the cause of climate change to be greenhouse gases that get released into the atmosphere, where today's agricultural facilities contribute to approximately 20% of the annual increase in greenhouse gas emissions through carbon dioxide (CO2), methane (CH4) and nitrous oxide (N2O).

Methods to overcome these challenges have been an active interest to researchers for years. While both have their complications, technology has proven to be a valuable asset in combating them. With recent developments in DL, agriculture will continue to thrive and alleviate a global food crisis. Some of these developments are explored in section 2.2.

2.2 Deep Learning in Agriculture

In recent years, the field of DL has taken the world by storm and has been implemented into various industries, predominantly scientific fields. Some of these fields include bioinformatics (Li et al., 2019; Min et al., 2017), biochemistry (Cova and Pais, 2019; Richardson et al., 2016), medicine (Ching et al., 2018; Esteva et al., 2019), food security (Mohanty et al., 2016; Ramcharan et al., 2017) and robotics (Pierson and Gashler, 2017; Sünderhauf et al., 2018). DL is a sub-field of Machine Learning (ML) that focuses on the creation of computer models (neural networks) that can learn without being strictly programmed (LeCun et al., 2015).

When considering the factors in food production, it is no surprise that agriculture requires multiple disciplines to be working in tandem to create an effective yield of crops. In a report written by Santos et al. (2020), they review 43 papers, where they discover a minimum of 14 domains that use DL in agriculture. Some of the most popular are disease identification, plant recognition, and land cover.

2.2.1 Disease Identification

Disease identification is specific to food security and is estimated, by the Food and Agriculture Organization of the United Nations, that 20 - 40% of global crop yields are lost each year to pests and diseases, despite the application of around two-million tonnes of pesticide (King, 2017). These large crop yield losses have required innovative approaches to increase the effectiveness of global crop yields.

One example is a report from Zhang et al. (2018) that presents a solution for accurately detecting maize leaf diseases using CNN architectures. They explain that maize disease species have increased over recent years, along with the degree of harm they can cause, and express the difficulty of diagnosing diseased leaves, especially by inexperienced farmers. This problem encouraged a need for an automatic system designed to identify maize plant leaf diseases by only looking at its appearance. Using a small dataset of 500 images, split into nine categories (8 types of maize diseases and a set of healthy ones), they applied data augmentation techniques to increase the dataset size to 3,060 images. This dataset was divided into a training set (2,248 images) and test set (612 images) and was applied to the GoogLeNet and Cifar10 CNN architectures, achieving identification accuracies of 98.9% and 98.8%, respectively. Comparatively Ren et al. (2019) used a Support Vector Machine (SVM), a type of ML model, on a similar disease identification problem. Unfortunately, the approach is limited to three types of leaf diseases and 48 images, reducing the models' capabilities, achieving an accuracy of 89.38%. Zhang et al. only uses a dataset with 10x the quantity (before data augmentation) and 3x more classes but ascertains a higher accuracy of 9.42%. Whilst DL approaches require more data, SVMs are limited in their classification abilities, providing an opportunity for DL to be extremely beneficial in the field of disease identification.

Another study written by Rahman et al. (2020) discusses a two-stage light-weight CNN architecture called Simple CNN that uses only 0.8 million parameters. This architecture can accurately identify rice diseases using a mobile device, which can be difficult due to the large number of parameters used within state-of-the-art architectures. The importance of Rahman et al.'s research involves the effectiveness

of rice production as it provides food security for over half the world's population. In countries such as Bangladesh, 93% of the cereal production is strictly used for rice, making this research extremely beneficial. The paper details an automatic rice disease detection app that helps reduce the average rice yield lost annually, which is between 10 - 15%. Rahman et al. successfully achieves a classification accuracy of 93.3% on nine classes consisting of five diseases, three pests, and one healthy set of images. The dataset used contained 1,426 images of pests and rice diseases in paddy fields from the Bangladesh Rice Research Institute (BRRI). Interestingly, the model has successfully outperformed similar memory-efficient CNN architectures, such as MobileNet (2.3 million parameters) and SqueezeNet (0.8 million parameters). With this discovery, Simple CNN helps to make a giant impact on the effectiveness of disease identification CNN models.

This section has shown the importance and benefits of CNN models within disease identification. Creating systems that use DL can increase food productivity by reducing the loss of crop yields and provide an increase in produce annually. Alas, disease identification alone cannot mitigate a global food crisis and requires assistance from other techniques such as plant recognition.

2.2.2 Plant Recognition

Plant recognition is a tool that supports agriculture automation and environmental conservation. It can be used for education, biodiversity monitoring, and plant observation, helping to bridge the taxonomic knowledge gap (Lasseck, 2017). Additionally, it is critical in robotics for developing robotic harvesting technologies (Ampatzidis et al., 2017), helping to increase food productivity and food security.

Initial approaches to address plant recognition problems uses Computer Vision (CV) methods. One example, written in a report by Jin et al. (2015), employs an image processing chain of image binarization to separate the background and leaf, detect contours and contour corners, and applies geometrical derivations to leaf tooth features. The approach evaluated eight leaf species consisting of 700 leaf images, split in half to create the training and test datasets. Overall, the method achieved low

identification accuracies between 72.8% and 79.3%. Unfortunately, image processing relies heavily on the contrast of images and requires the images features to be initially separated to achieve effective results. Comparatively, Lee et al. (2015) creates a CNN model to address a similar leaf recognition problem. The model is pre-trained on the ILSVRC2012 dataset and uses transfer learning, producing superior results. The model's final fully-connected layer is adjusted to accommodate the MalayaKew Leaf dataset that consists of two 44 class datasets: one smaller one with 2,288 training and 528 test images; and a larger one with 34,672 training and 8,800 test images. After running both datasets on a separate copy of the pre-trained model, the resulting identification accuracies achieved were 97.7% and 99.5%, respectively. The difference in performance shows how effective DL can be in image classification tasks by successfully obtaining a massive increase of 24.9% over Jin et al.'s approach. However, it does have its limitations.

Mitrović and Milošević (2019) implement three CNN architectures, LeNet, AlexNet, and a modified LeNet, classifying three types of flowers. Both datasets (training and testing) consisted of Flickr, Google and Yandex images, totalling 4,242 images, where the images were resized to 100x100 pixels with three colour channels. The modified LeNet performed the best with only 73.41% accuracy. Mitrović and Milošević trained each model from scratch, which massively reduced the classification accuracy caused by the limited training data. Normally, DL models require large quantities of data to generalize effectively. Fortunately, a technique called transfer learning can accommodate datasets with small quantities of data. Tan et al. (2018) found that the scale of a DL model and the size of the required amount of data have an almost linear relationship, where the expressive space of the model needs to be large enough to discover patterns within the data. While it is common practice for training data to be independent and identically distributed (i.i.d.) with the test data, transfer learning doesn't require this. Instead, transfer learning provides an opportunity to address the problem of insufficient training data and avoid newly created models needing to be trained from scratch, significantly reducing the demand for training data and training time. A report written by Gogul and Kumar (2017) describes an implementation of flower classification that uses three CNN architec-

tures combined with transfer learning. They use two datasets, one with 28 classes (2,240 images) and another with 102 classes (8,189 images). The architectures used included Inception-v3, Xception and OverFeat, where the achieved accuracies are 92.41%, 90.18%, 85.71% (28 classes) and 93.41%, 90.60%, 73.05% (103 classes), respectively. Interestingly, the dataset is half the size for the smaller number of classes and double the size for the larger classes but achieved an accuracy 19% higher than Mitrović and Milošević's approach.

Additional studies by Kaya et al. (2019), Sun et al. (2017), and Yalcin and Razavi (2016) present further success of DL models for plant identification, where they achieve accuracies of over 91%. The effectiveness of this field in agriculture has encouraged the creation of applications such as Pl@ntNet, a large-scale participatory platform and information system dedicated to the production of botanical data through image-based plant identification (Affouard et al., 2017); and LeafNet, a CNN-based plant identification system (Barré et al., 2017).

When combining both disease identification and plant recognition techniques, increasing food productivity becomes a little easier. Over recent years, disease identification and plant recognition have been introduced into smart farming to assist in the automation of crop management, making farming more manageable and cost-effective (Ünal, 2020). As smart farming is very new, it can take several years to become fully adopted worldwide, reducing the time available before a potential food crisis. Additionally, there is the possibility that smart farming alone will be unable to accommodate the increase in food demand, making it crucial to consider alternative solutions. The artefact presented within this paper focuses on plant recognition of wild edible plants using state-of-the-art CNN architectures, enabling the consideration of a category of plants for daily consumption. Thus, providing a secondary solution to the food crisis, increasing the consumable food within the world through natural means.

Chapter 3

Methodology

3.1 Project Management

The artefact focuses on using three state-of-the-art CNN architectures to successfully surpass experts at accurately classifying wild edible plants, given a plant image. To achieve this common image classification characteristics were considered. These include: selecting effective training samples, opting for a suitable method to assess the accuracy and to choose the appropriate classification models (Dey et al., 2014).

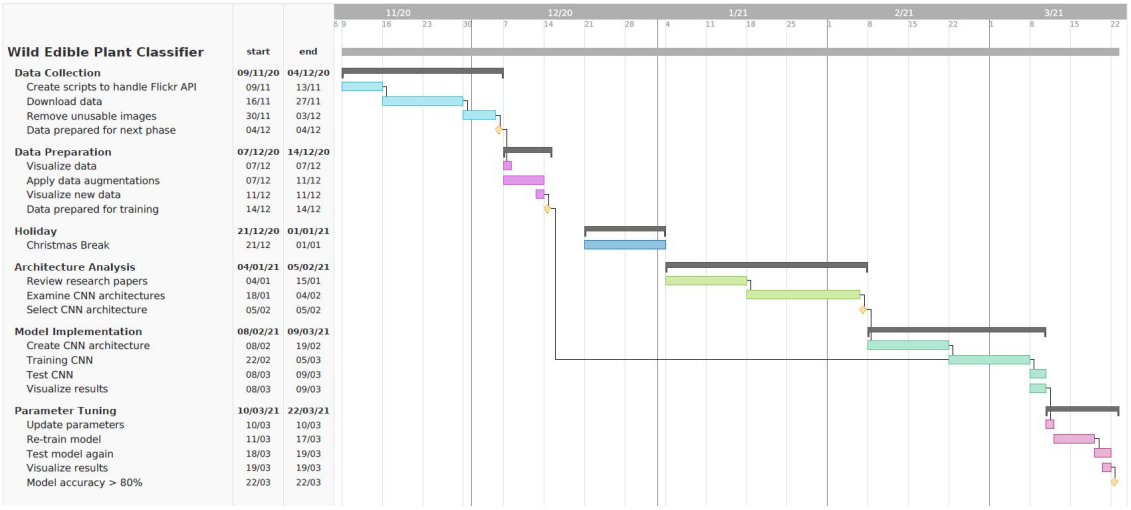

Figure 3.1: The Gantt chart used for the project, demonstrating the Waterfall methodology.

These characteristics required a project management methodology that follows a linear process. Suitable methods that were examined include Waterfall, Kanban, Agile and Scrum. Unfortunately, Scrum and Agile both require a team of people, presenting unfavourably for this project. However, Waterfall and Kanban both provide

unique benefits and were used within the creation of the artefact. The Waterfall method takes a linear-sequential approach where every stage starts only after the previous one has completed. It begins with a research stage before planning the artefact, designing and developing it (Despa, 2014). Similarly, the Kanban method can be employed through a sequential approach but drives a visualization of a given workflow, using card list-style applications like Trello, and is used to limit the quantity of work-in-progress (WIP) at each stage (Ahmad et al., 2013).

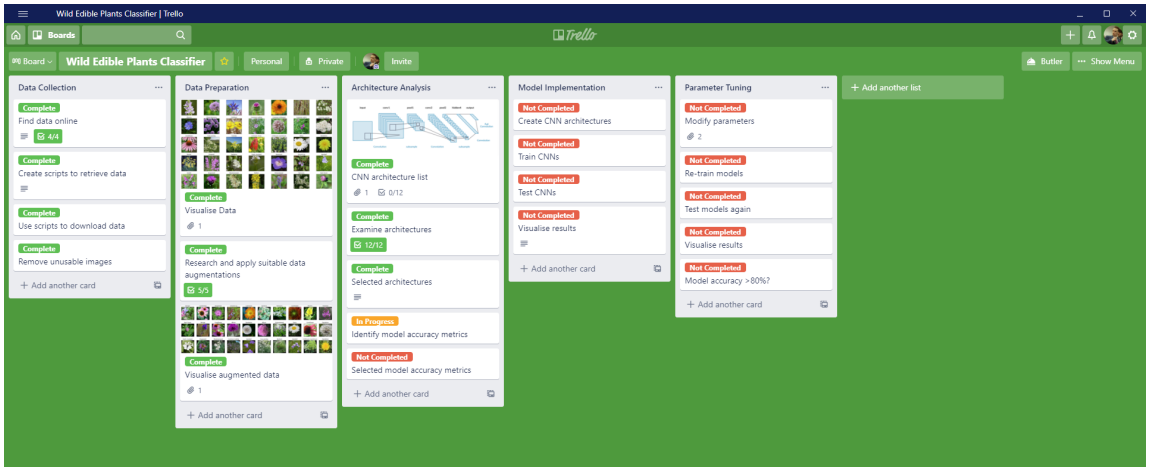

Figure 3.2: The Trello board used for the project, representing the Kanban methodology.

The Waterfall method provided an overview of the key tasks implemented with timeframes, represented within a Gantt chart. Kanban focused on a Kanban board to provide the opportunity to split tasks into smaller components, enabling the ability to store information associated with each one. Combining both methodologies helped to achieve the required deadlines and provided a coherent context for each task.

3.2 Software Development

As mentioned in section 3.1, the software development methodologies implemented within the project were Waterfall and Kanban. Analysing the functionality of both methodologies further, Waterfall has six distinct components: analysis, design, development, testing, implementation and maintenance (Eason, 2016). The Kanban board used within the project was divided into groups, and the groups were subdivided into tasks (Johnson, 2017). For example, data collection, data preparation

and architecture analysis encompass the analysis section of the Waterfall methodology. A design section wasn't required as the architectures that were used already have a specific design. Whereas the model implementation section relates to the development, testing and implementation, and lastly, the parameter tuning section focuses on the maintenance component.

The methodologies enforced the artefact to be created through a traditional approach using object-oriented programming (OOP). That included the pre-processing of the data, accessing the CNN models using transfer learning, training and testing the models on the pre-processed data, and visualising the results. An exception involved the collection of the image data from Flickr's API, which required a scripting programming language to access files on the given computer. Python 3 presented the optimal programming language to accommodate both requirements due to its capability to be a scripting language and an OOP language. Furthermore, the project focuses on a research approach to identifying wild edible plants rather than an application optimised for use within a natural environment, making the artefact suitable for an office or home environment.

3.3 Toolsets and Machine Environments

Selecting the right toolsets and machine environments can be crucial in creating effectual software applications (Bulajic et al., 2013). Integrated development environments (IDEs) are a coding environment that enables the ability to write, test, and debug code easier, with elements such as code completion, syntax highlighting, resource management, and debugging tools (Vasconcellos, 2018). With their extensive utility, IDEs are a valuable asset in creating software applications efficiently. Three popular cross-platform IDEs were examined for use within the creation of the artefact. These were: Spyder, PyCharm and JupyterLab.

Spyder provides the traditional elements of IDEs, along with a built-in IPython Console that enables running code in selected blocks. Additionally, it provides a viewable area for stored variables that can be interacted with and modified on the fly and has plugin support for additional functionality (Spyder, Undated). Moreover, Spyder

focuses on Python programming in conjunction with Data Science, ML and DL, and is included within the Anaconda package (Anaconda, Undated). In comparison, PyCharm uses similar functionality to Spyder but has a larger focus on all aspects of Python programming, including full-stack web development, with a wider variety of extensions made by community developers and has the addition of a visual debugger (JetBrains, Undated).

JupyterLab presents a unique approach to IDEs and is a web-based interactive development environment that incorporates Jupyter Notebooks. These are files that allow for creating small blocks of code that can be executed for immediate results. Furthermore, JupyterLab provides the ability to use text editors, terminals, data file viewers, and other custom components side-by-side with notebooks in a tabbed working environment (Project Jupyter, 2018). Although each IDE has its benefits, JupyterLab was selected as the IDE of choice for this project. DL requires efficient visualisation of multiple components that are commonly created through small segments of code, making Jupyter Notebooks a powerful tool for DL. Equally, having the convenience of writing code into a separate file and viewing a dataset or plot alongside the Notebook prevents the need for remembering information.

Relative to project management tools, there are multiple Kanban software applications available. These were created using three core concepts: a board, lists, and cards. The broad represents a workplace to manage workflows and tasks. Lists can be found within the board and are classified as columns for the types of duties to complete. Cards are presented within lists and are the individual tasks for each group needed to complete the project (Zapier, Undated). Three Kanban applications were considered for the project: Trello, MeisterTask and KanbanFlow.

KanbanFlow uses a lean workflow approach to Kanban boards. Some of its core features include WIP limits per column, checklists inside tasks for sub tasking, splitting a board into teams, task filtering, setting recurring tasks, and the opportunity to add relationships between tasks (KanbanFlow, Undated). MeisterTask uses similar functionality to KanbanFlow with a few additions, such as custom fields and card attachments (MeisterTask, Undated). Trello extends the features from both Kan-

banFlow and MeisterTask further, enabling the creation of templates, application extensions and providing an automation feature, Butler, that allows users to set rule-based triggers (Stone, 2020). Overall, Trello was selected as the Kanban board application of choice due to its extensive features, clean design, and user-friendly interface. Over recent years, DL's popularity has sparked the creation of multiple frameworks that offer building blocks for designing, training, and validating deep neural networks (NVIDIA, Undated), presenting the opportunity to efficiently create models within a real-world setting. Three popular open-source DL frameworks were considered when creating the artefact: TensorFlow, PyTorch and Keras.

TensorFlow provides support for multiple programming languages, such as JavaScript, Python and C++, and comes equipped with a variety of tools and community resources to provide a simple process for training and deploying DL models. Its core tool focuses on the building and deploying of models within browsers. However, TensorFlow Lite enables model deployment on mobile and embedded devices (Goyal, 2021). TensorFlow can be challenging for DL beginners due to some of its syntax for complex models. To combat this, a second DL framework, Keras, can be run on top of TensorFlow. Keras is written in Python and enables fast experimentation when creating DL models. It automatically takes care of core tasks and generates an output while minimising user actions and makes it easy to understand models (Sharma, 2019; Keras, Undated).

PyTorch is based on the Torch library that aims to accelerate the path from research prototyping to production deployment. Like TensorFlow, it has a rich ecosystem of tools and libraries and enables flexibility when creating simple to complex models (PyTorch, Undateda). Additionally, PyTorch has a programming style that feels more 'pythonic', comparative to other frameworks, making code cleaner and understandable for developers. Although, another core difference focuses on the graph definitions between PyTorch and TensorFlow. TensorFlow's graph is defined statically, which requires developers to outline the entire structure (layers and connections) of the model before running it. While PyTorch is defined dynamically, enabling the graph and its input to be modified during runtime. The dynamic definition offers developers more accessibility to the inner-workings of models created, providing a

considerably easier time debugging errors (Welch, 2020). Working with complex architectures can be time-consuming in training, testing, and debugging. As PyTorch helps to mitigate this, it was chosen as the framework for creating the CNN models.

3.4 Research Methods

Quantitative and qualitative are the two fundamental research methods used in projects today. Quantitative research focuses on numbers and statistics, while qualitative research analysing words and meaning (Streefkerk, 2020). When reviewing relevant literature within the field (chapter 2), it became clear that quantitative data is beneficial to CNNs.

3.4.1 Convolutional Neural Networks

CNNs are an extension of Artificial Neural Networks (ANNs), which are computational processing systems inspired by how biological nervous systems work within the brain. ANNs comprise of a high number of interconnected computational nodes (neurons). They work in an entwined distributed fashion to collectively learn from its provided input and optimise its final output (O'Shea and Nash, 2015). Figure 3.3 models a common ANN architecture that takes in a matrix of inputs. Which then performs computations of those inputs through the hidden layer and returns an output via the output layer.

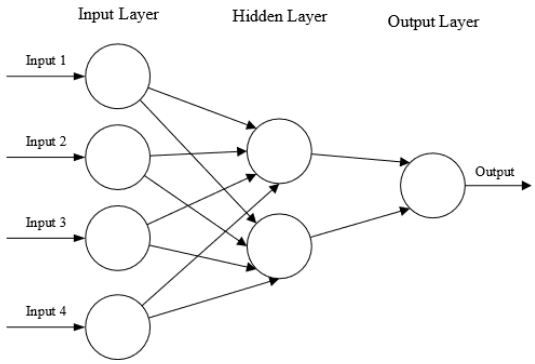

Figure 3.3: A simple three-layered Feedforward Neural Network (FNN) (ibid.).

The entwined neurons each have a connection (weight value) and a bias value assigned

to them, allowing ANNs and CNNs to learn patterns within data. Weights represent coefficients that reflect the importance of an input value. The bias is a constant value added to the dot product of the inputs and weights, reducing the variance between nodes in the network and increasing its generalization (Malik, 2019).

Each neuron within a hidden layer is known as a perceptron, which performs mathematical computations to achieve an output. Equation 3.1 highlights the formula for a single perceptron that contains: a non-linear activation function (g), a vector of matrix values (x), a matrix of weight values (w), and a bias term (b). The dot product of the input values and weight values are added to the bias term; before being passed through a non-linear activation function to achieve the predicted output ($\hat{y}$).

$$\hat{y} = g(b + x^T w) \tag{3.1}$$

As images are a matrix of pixels, why use a CNN over an ANN for image classification? CNNs capture the spatial and temporal dependencies in images using filters, increasing classification accuracy and reducing the number of parameters required for training (Saha, 2018). Additionally, CNNs can effectively manage shifts and distortion in images, such as changes in shape due to camera lens, different lighting conditions, and horizontal and vertical shifts (Hijazi et al., Undated). These benefits make it extremely beneficial in the field of image classification. CNNs have four components: convolutional layers, pooling layers, fully-connected layers and activation functions.

A. Convolutional Layer

A convolutional layer is one of the fundamental components of a CNN architecture. It takes a specified sized filter (kernel) to perform feature extraction, separating the distinct features from the image into feature maps. This kernel is commonly between the sizes 2x2 and 4x4, represented as a tensor, or matrix, of numbers with 1s and 0s. The layer calculates an element-wise product between each number within the

kernel and the pixels it overlaps. Next, it moves a specified stride over the image, repeating the same procedure until the whole image is processed. These new values are summed together to create a new smaller image known as a feature map, which is output by the convolutional layer (Yamashita et al., 2018). Figure 3.4 depicts a visual representation of a convolutional operation.

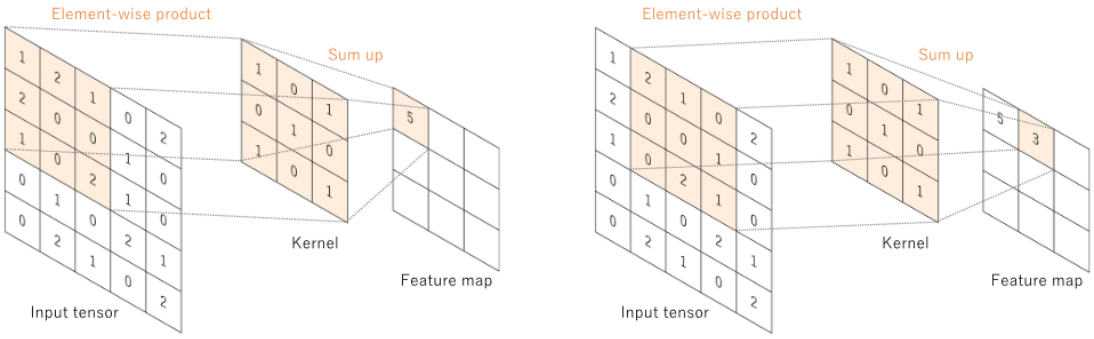

Figure 3.4: An example of a convolution operation with a kernel size of 3x3 that has no padding and uses a stride of 1. The kernel is iterated over the input tensor (image), where the element-wise product is calculated between each element of the kernel and the highlighted input tensor. These values are summed together to achieve the result of 5 and then 3 (ibid.).

Equation 3.2 presents the mathematical formula for a convolutional layer. The operation is a function G that accepts the row (m) and column (n) indices of the top left corner of the kernel, on top of the input image. Subsequently, the input image (f) and kernel (h) are calculated using an element-wise product and are summed together (Skalski, 2019).

$$G[m,n] = (f * h)[m,n] = \sum_{j} \sum_{k} h[j,k] f[m-j, n-k] \tag{3.2}$$

B. Pooling Layer

Pooling layers downsample feature maps, which reduces the dimensionality, number of parameters, and computational complexity of the model (O'Shea and Nash, 2015). Interestingly, a report written by Suárez-Paniagua and Segura-Bedmar (2018) identifies that selecting the appropriate pooling operation can improve the classification accuracy of a CNN model. Overall, this helps to reduce overfitting and provide the

ability to extract the most representative features of an image. While there are multiple types of pooling operations available, this report focuses on some of the most famous ones: max pooling and global average pooling. Like convolutional layers, pooling layers use a filter size and stride as hyperparameters to perform pooling operations.

Max pooling is the most popular pooling operation used in image classification tasks. It takes a feature map as input and uses a kernel, of a specified size, over the top of the feature map to extract the highest values within parts of the image. Given a specified stride, the kernel moves several pixels across the image until it is fully processed, resulting in a smaller sized image with only the most important features. An example of this operation is presented in Figure 3.5 (Cook et al., Undated).

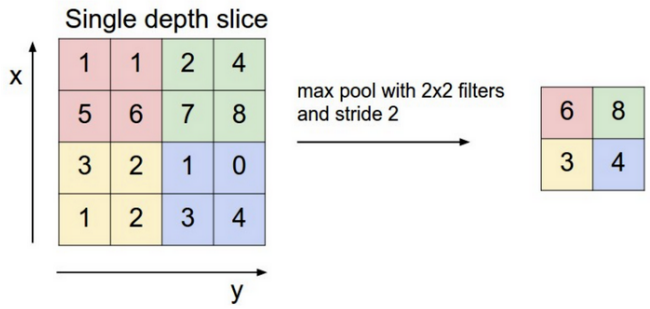

Figure 3.5: Max pooling operation with a 2x2 filter and stride of 2 (ibid.).

Global average pooling is a more extreme type of dimensionality reduction. It doesn't require a specified kernel or stride and replaces fully-connected layers in CNNs, making it useful in the realm of natural language processing. Like max pooling, it takes a given feature map as input but, instead of taking the maximum value, it computes the average of all the pixel values and returns only one value as the output (ibid.).

C. Fully-connected Layer

Fully-connected (FC) layers are commonly used as the final set of layers in a CNN architecture. These layers perform just like FNNs, where they provide non-linear combinations to classify an image's high-level features into a label. Before the FC

layers can operate, the output of the convolutional or pooling layer, connected to the first FC layer, must be flattened into a one-dimensional vector. This vector is then fed into the FNN and is backpropagated at every training iteration. Over time, the model is able to distinguish between image features and label them with a given probability using a Softmax activation function (Saha, 2018).

Backpropagation (BP) is a critical component of FNNs and is the essence of the network's training. It is the practice of fine-tuning the NNs weights based on the error rate (cost/loss function) obtained within the previous iteration (epoch). The lower the networks error rate, the more reliable the model is at generalizing (Al-Masri, 2019). BP uses the chain rule (Equation 3.3 and 3.4) to calculate the gradients of each input that update the weights and biases for each layer. There are two equations for calculating the gradients, with one equation for each component.

$$\frac{\partial C}{\partial w^{(l)}} = \frac{\partial C}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial w^{(l)}} \tag{3.3}$$

$$\frac{\partial C}{\partial b^{(l)}} = \frac{\partial C}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}} \tag{3.4}$$

Equation 3.3 outlines the gradient calculation for the weights and Equation 3.4 for the biases. Each component of the equations represents a change of a particular element within the network. C is the cost function, l is the current layer of the network, w are the weights of the given layer, a is the activations (input values), b is the biases of the given layer, and z represents the output of a forward pass of the network (Equation 3.1).

$$w^{(l)} = w^{(l)} - \alpha \times \frac{\partial C}{\partial w^{(l)}} \tag{3.5}$$

$$b^{(l)} = b^{(l)} - \alpha \times \frac{\partial C}{\partial b^{(l)}} \tag{3.6}$$

Equation 3.5 and 3.6 depicts the formulas for both the weights and biases. The formulas use the current weight and bias values, subtracting them from a given learning rate α (e.g. a small value of 0.001) that is multiplied by the calculated gradient (Hansen, 2019).

D. Activation Function

Activation functions are responsible for transforming linear inputs into nonlinear outputs, aiding the network to learn complex patterns. Additionally, they restrict output values to a limit, commonly between 0 and 1, which enables a reduction in the computational complexity of the model and simplifies image classification tasks (Jain, 2019). There are two common activation functions used in CNNs when handling image classification problems: Rectified Linear Unit (ReLU) and Softmax.

The ReLU activation function provides the ability to deactivate neurons by setting any negatively valued neurons to zero, increasing the model's computational efficiency and providing a linear dropout effect for CNN architectures, reducing model overfitting. Additionally, it eliminates the vanishing gradient problem. An issue that happens when there is successive multiplication of derivatives, resulting in values becoming closer and closer to zero, which occurs during gradient descent (Nwankpa et al., 2018). The mathematical formula for ReLU is presented in Equation 3.7, where the input value x is set to its initial value or zero (if the value is negative).

$$f(x) = max(0, x) = \begin{cases} x, & if\ x \geq 0 \\ 0, & if\ x < 0 \end{cases} \tag{3.7}$$

The Softmax function is designed for multi-class classification problems, as it computes a probability distribution from a vector of real numbers. This function produces an output of numbers between the range of 0 and 1, where the target class has the highest probability value, and all summed values must equal 1. This function returns a vector of probabilities for each image class and is used within the output layer of CNN architectures (ibid.).

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{3.8}$$

The Softmax function's mathematical formula is depicted in Equation 3.8. This function accepts an input vector as a parameter and has the standard exponential function applied to each element within the input vector. Subsequently, the inputs are normalized by dividing the individual exponential element by the sum of all the exponentials within the input vector.

E. Chosen Architectures

The CNNs chosen were selected based on two factors: to vary in size/complexity and to have a similar top-5 error rate of approximately 10%. PyTorch (Undatedb) provides a useful table of statistics for various state-of-the-art models with top-1 error and top-5 error rates. Using this table, CNNs were selected to achieve the selection requirements. Overall, this approach considers whether the size of the architecture truly matters when classifying wild edible plants. Schematics of the GoogLeNet and ResNet-34 architectures are depicted in Appendix A, while MobileNet V2's components are highlighted in Figure 3.6.

MobileNet v2 is the largest architecture chosen in this paper and achieved a top-5 error rate of 9.71%. Sandler et al. (2019) describes the design to be effective specifically in mobile and resource-constrained environments. Compared to other models, such as VGG-16, the architecture significantly decreases the number of operations and memory needed for training and testing; while retaining a high classification accuracy. The secret to its efficiency stems from using depth wise separable and pointwise convolutions over traditional convolutional layers. It uses 53 convolutional layers with no pooling layers but manages to reduce the total number of parameters used and is composed of multiple blocks of layers, represented in Figure 3.6.

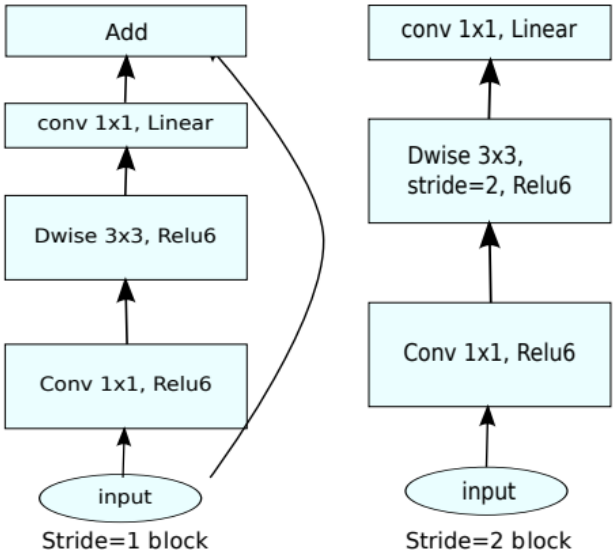

Figure 3.6: MobileNet v2 architecture block components. Both represent bottleneck-residual blocks that are the core component of the architecture. The first one (left) depicts an inverted residual block and the second one (right) outlines a standard residual block (ibid.).

Moving onto the middle-sized architecture, ResNet-34, He et al. (2015) explains the difficulties of training deep neural networks and how the ResNet framework helps to mitigate this. While it contains only 34 layers (with no pooling layers), the architecture uses a unique functionality known as shortcut connections. These shortcut connections address a degradation problem that deep neural networks can experience. This problem causes a reduction in accuracy when increasing the depth of a network after reaching a maximum. Identity shortcuts are primarily used in ResNet architectures and focus on inputs and outputs that use the same dimensions. However, when the image dimensions increase, an additional type of shortcut is considered. As before, an identity mapping can be performed but uses extra zero padding to standardise an image's dimensions. The alternative includes using a projection shortcut to match the new dimensions. In contrast to MobileNet V2, ResNet-34 achieved a top 5-error rate of 8.58%, the lowest of the three architectures.

The final and smallest architecture selected in this paper is GoogLeNet, which achieved a top-5 error rate of 10.47%. Szegedy et al. (2014) writes specifically about the requirements of this architecture and details its structure. They explain

that the models' design limited the computational budget to 1.5 billion multiply-accumulation operations at inference time and extended the usability to real-world scenarios on large datasets, rather than purely for academic purposes. In total, it contains a total of 22 layers (27 including pooling layers).

3.4.2 Artefact Pipeline

The artefact created within this paper focuses on a 7 step ML pipeline, depicted in Figure 3.7. Revisiting the problem definition, the goal of the artefact is to successfully classify wild edible plants with a minimum of 80% classification accuracy.

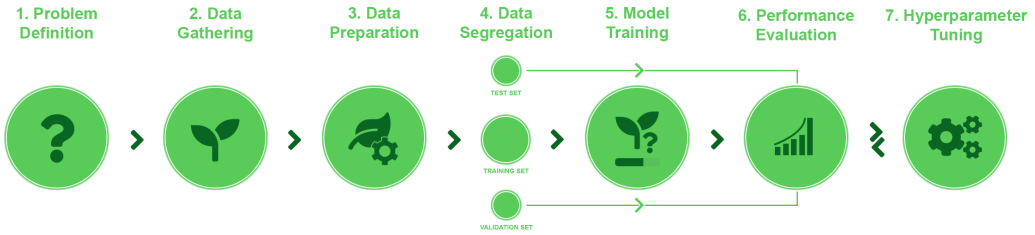

Figure 3.7: ML pipeline.

The first challenge to achieving this problem definition required functional plant images. While the idea of gathering plant images sounds simple, the difficulty stemmed from finding a large enough quantity of data for specific wild edible plants. Although some edible plant datasets are available on websites such as Kaggle and GitHub, the number of data and quality of images was limited. Fortunately, Flickr provides a wide variety of plants, including wild edible plants. Using Flickr's API, 35 classes of useful plant images (16,526 total) was easily obtainable. The dataset was examined and replaced with new data, where appropriate, to ensure the validity of the dataset.

Once the images were collected, Python scripts assisted in preparing them for data augmentation and segregation. The images were resized to a width of 600 pixels and renamed to their corresponding label name accompanied by an index value (e.g. alfalfa0 and alfalfa1). Enabling a consistent format across all images reduces computational complexity, improving the efficiency of the algorithm.

The data segregation step focuses on augmenting the images and splitting them into

three datasets: test, training, and validation. These followed a 15/70/15 percentage split ratio, respectively. Fortunately, the PyTorch DL framework enables an efficient method for pre-processing data that focus on projects utilising quantitative data. Its torchvision module enables swift augmentation (transformation), splitting, and loading data into batches for training.

The last three components of the pipeline, model training, performance evaluation, and hyperparameter tuning, correlate heavily. Model training focuses on creating the CNN architectures and the initial training implementation, which uses the training set with a static set of parameters. Performance evaluation focuses on the performance of each model via the selected performance metrics discussed in subsection 3.4.3. Hyperparameter tuning involves implementing the three models using different batch sizes and hidden layers, determining each models' optimal parameters. These parameters were evaluated using the performance metrics and saved for each model. A final set of training was performed using the training set and evaluated on the test set. The test set served as a 'new' unseen dataset making it perfect for the final performance evaluation.

3.4.3 Performance Metrics

Selecting the appropriate performance metrics is essential in accurately determining the best model for classifying wild edible plants. This paper utilises some of the most popular classification metrics: classification accuracy, top-1 error rate, top-5 error rate, precision, recall, and F1-score.

Accuracy is a metric that is effective at providing a baseline for each model's performance. While the top-1 error rate refers to the standard error rate of the model (1 – accuracy), and the top-5 error rate relates to the correct class being present within the top 5 predictions. Unfortunately, these three metrics alone cannot provide a detailed understanding of how the models are performing. Precision and recall are two class-specific performance metrics that are beneficial when a class distribution is imbalanced. Precision focuses on the proportion of true positives that were correct against false positives, and recall is the proportion of true positives against false

negatives. When precision is 1, there are no false positives in a model's predictions. Similarly, when the recall is 1, there are no false negatives in a model's predictions (Google, Undated).

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \tag{3.9}$$

Precision and recall are calculated using a confusion matrix, which contains the true positive (TP), false positive (FP), false negative (FN), and true negatives (TNs) of a model. Represented in the top left, top right, bottom left, and bottom right of the matrix. F1-score (Equation 3.10) is used to find a balance between precision and recall, which is calculated using the harmonic mean of these two metrics. The harmonic mean punishes extreme values, providing a better measure of incorrectly classified predictions. Like precision and recall, this metric is a number between 0 and 1, where 1 has a perfect balance between the metrics (Koehrsen, 2018).

$$F_1 = 2 * \frac{precision * recall}{precision + recall} \tag{3.10}$$

These metrics are presented within a statistics table in section 4.5, comparing each model's performance. Additionally, confusion matrices and receiver operating characteristic (ROC) curves visually evaluate the difference between the models.

Chapter 4

Design, Development and Evaluation

4.1 Software & Hardware Requirements

The artefact created utilises the Python programming language, specifically version 3.9.2, and has its functionality extended with the assistance of 8 packages (libraries). While the latest Python version is recommended, it is possible to run the artefact with a minimum of version 3.7. However, it is critical to ensure that the libraries installed utilise the same versions described in this section. The 8 libraries used (Table 4.1) include Matplotlib, NumPy, Torch, Torchvision, SciPy, Pandas, Scikit-learn, and Scikit-plot.

Another core piece of software used for creating the artefact is virtual environments. When developing software, it is common for developers to have packages and libraries already set up with the latest package versions. Unfortunately, it can be time-consuming and troublesome to switch between package versions when working on multiple projects. However, virtual environments help to mitigate this issue by preventing conflicting package versions. Anaconda provides a platform to manage Python environments and enables accessibility through a Python kernel, integrated seamlessly with Jupyter Notebooks. To create a conda environment for this project, please refer to the following GitHub repository: https://github.com/Achronus/wep-classifier.

Name	Description	Version
Matplotlib	A package designed for graphic representations of data and results. Required for visualising the data and the CNN models results.	3.3.4
NumPy	A package used for scientific computing, providing multidimensional array objects and the ability to perform mathematical computations on them.	1.20.1
Torch	A tensor library for DL using graphics processing units (GPUs) and central processing units (CPUs). The heart of the creation, training and testing of the CNN architectures used within this project.	1.7.1
Torchvision	An extension to the torch library, providing dataset management, data augmentation, and transfer learning functionality.	0.8.2
SciPy	Another package for scientific computations, extensively used within the Scikit-learn package.	1.6.0
Pandas	A package dedicated to data analysis that uses dataframe objects to view and manipulate data within a tabular format. Required for presenting the model results in a tabular format.	1.2.2
Scikit-learn	A package built on top of NumPy, SciPy, and Matplotlib that provides functionality for efficiently creating ML models and calculating their performance. Required for creating confusion matrices.	0.2.4
Scikit-plot	An extension of Scikit-learn specific to plotting performance metrics. Required for creating the confusion matrix and ROC curve plots.	0.3.7

Table 4.1: Python libraries used within the project.

Furthermore, the code within the artefact requires a platform that accesses ipynb files (Jupyter Notebooks). Applications such as Google Colab, Jupyter Notebooks, Jupyter Lab, and Microsoft Azure Notebooks provide the available functionality to achieve this.

CNNs can take a long time to train and tune when only using a CPU. To increase this, NIVIDA has a parallel computing toolkit that allows the transfer of data onto GPUs, called CUDA. Fortunately, PyTorch has CUDA functionality built into its library that automatically keeps track of the available GPUs on a system and the tensors allocated to them. When experimenting with different CUDA versions, CUDA toolkit version 10.1 proved fundamental and is required to run the artefact within this project to minimise the training, tuning and evaluation computation speed.

The system hardware used to create the artefact is as follows:

- Intel Core i7-4790k CPU 4.00GHz
- 16GB of RAM
- NVIDIA GeForce GTX 970 (4GB) GPU
- Windows 10 Operating System

It is recommended, but not required, to have similar hardware specifications to run the artefact to maximise its efficiency. If a GPU is not available, it is possible to run the artefact on a typical CPU. However, the tuning time taken to complete the 36 model variants on a GPU took approximately two days. Using a CPU, the time taken will increase by a minimum of 4x as long.

4.2 Design

The artefacts structure is split into three notebooks and four classes, where it follows a linear process to complete the ML pipeline discussed in subsection 3.4.2. The first notebook provides insight into the functionality of the core components. This functionality includes pre-processing and augmenting the data, building, training and testing the CNN architectures, and evaluating their performance on static parameters. This notebook covers steps 2 to 6 of the ML pipeline (Figure 3.7) and uses functions from three of the four classes.

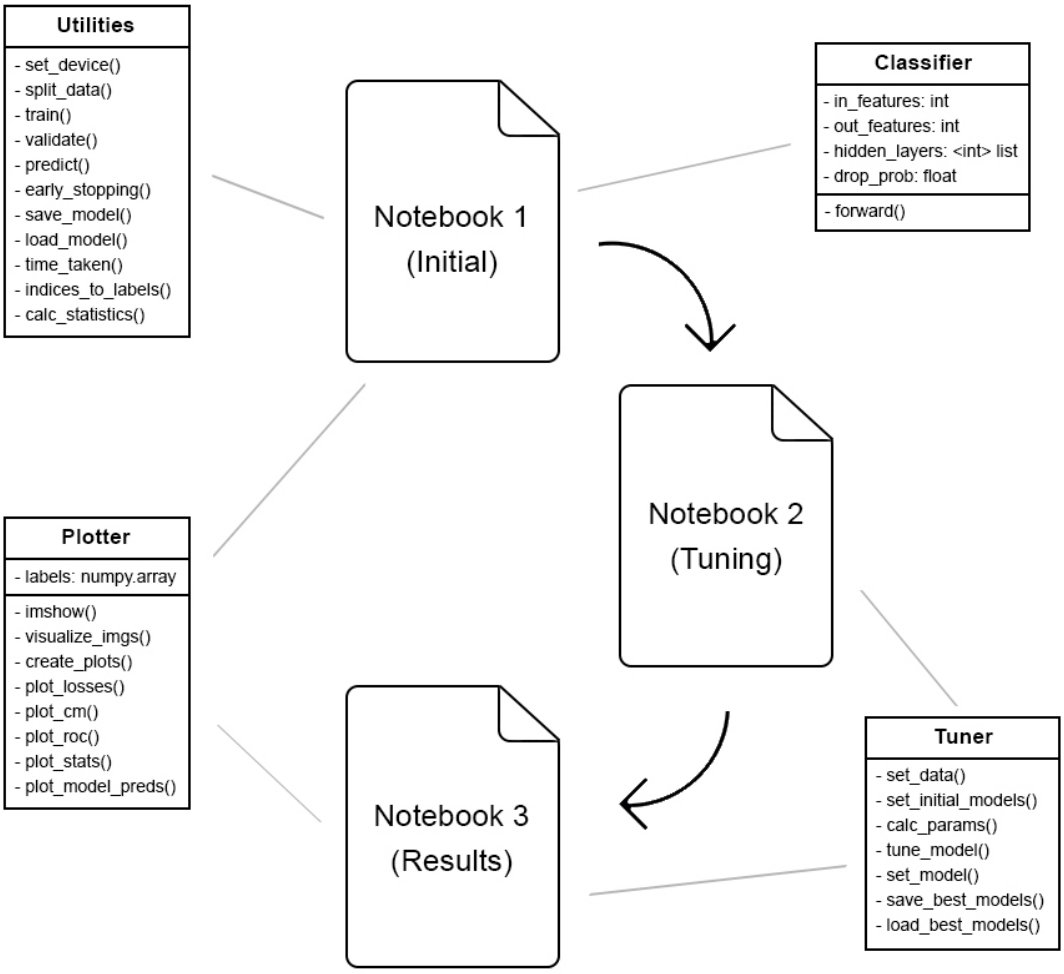

Figure 4.1: A topology of the notebook and class structures, where each class has a pointer indicating which notebook is associated with it. Some items (classifier and plotter) contain three components: a name, input parameters and functions, while others (utilities and tuner) only include two: a name and functions, respectively.

The second notebook uses similar functionality to the first one. However, it focuses strictly on tuning and saving the model's parameters for all 36 trained variants. This notebook covers step 7 of the ML pipeline and uses only one class of functionality. The third notebook focuses on evaluating and visualising the results of all 36 model variants, with a core focus on the best-performing ones. This notebook covers step 6 for step 7 of the ML pipeline and uses two classes for its functionality.

The artefacts design provides a clear and concise approach to understanding its functionally, enabling users and other developers the opportunity to easily decipher its components and utilise them within applications of their own.

4.3 Development

The development phase of the project focuses on the programming of the artefact. This phase is one of the longest within software methodologies and commonly requires the project to be divided into manageable subtasks. The subtasks followed are shown within the Gantt chart and Trello board outlined in section 3.1, covering most of the project's development process, where they are explained in detail within this section.

The first stage in creating the artefact began with identifying suitable wild edible plants for classification. A total of 50 arbitrary plant types were selected, obtained through the assistance of websites such as Original Outdoors and Wild Food UK (Original Outdoors, Undated; Wild Food UK, Undated), and analysed against Flickr's website, confirming the number of images accessible. Unfortunately, a portion of the plant classes was removed due to limited quantities of data available, reducing the total number of plant classes used for classification. Using Flickr's API, images of the remaining plant classes were obtained and stored locally. Each set of images were manually reviewed and were removed based on one of four conditions. These conditions include: if it was a painting, the wrong type of plant, it was unidentifiable, or it contained human entities.

These restrictions reduced the number of available data further but provided a more accurate reflection of natural environments. The remaining 35 classes of wild edible plants accommodated a total of 16,535 images. Furthermore, they were resized to a width of 400 pixels and renamed to a uniform format across all classes (e.g. alfalfa0, alfalfa1). These optimisations reduced the file size of each image and provided accessibility to PyTorch's data loaders.

4.3.1 Initial Implementation

With the data prepared, an analyse of suitable Python libraries was conducted. Having had previous experience with ML techniques and creating CNN architectures before, identifying suitable packages was simple. The list of libraries used, detailed in section 4.1, were installed onto a conda virtual environment with Jupyter Lab enabled. The first notebook starts by importing the libraries (packages) and initialising the hyperparameters used throughout the document. The hyperparameters chosen were based on previous knowledge of CNN architectures, setting a good benchmark for later tuning. These parameters include: 20 epochs (training iterations), a learning rate of 0.001, a batch size of 64, a validation and test dataset split size of 15% (70% training data), and two hidden layers for the new classifier with node sizes of 512 and 256.

The notebook continues with data augmentation techniques, where the images are resized and centrally cropped to 224x224 pixels. This size is required as input for each CNN and has become a standard for many state-of-the-art architectures. Additional augmentations are applied, including a 50% chance to flip an image randomly horizontally and randomly rotating them by 20%, providing diversity to the dataset. The image data is then converted into a tensor and normalized using a mean and standard deviation of 0.5, which scales the data between the range of -1 and 1, helping to improve each model's accuracy. The augmented data is passed into a function that splits the data into three separate data loaders (based on the split size): training, validation, and testing. It starts by selecting images from each class, allocating them to one of the data loaders at random. This approach increases the loaders effectiveness and prevents the models from learning sequential patterns based on how the image data is ordered. These loaders are then converted into iterators that return data in batches based on the given batch size. Using groups of data helps to reduce the computational complexity during training and can increase classification accuracy. Different batch sizes are explored during the tuning of the models in subsection 4.3.2.

The models are created and pre-trained on the ImageNet dataset using PyTorch's torchvision module, enabling transfer learning. Transfer learning is a ML technique that reuses an existing trained network, developed on one task, and used as a foundation for another network on a similar one. There are four different cases available when considering a transfer learning approach:

- Case 1: using a new small dataset that is similar to the pre-trained data.
- Case 2: using a new small dataset that is different from the pre-trained data.
- Case 3: using a new large dataset that is similar to the pre-trained data.
- Case 4: using a new large dataset that is different from the pre-trained data.

Each case requires different modifications to the pre-trained network. Cases 3 and 4, commonly known as fine-tuning, require the model to be fully re-trained on the new data. While cases 1 and 2 only require minor adjustments to the network (Cook et al., Undated). As the dataset used is relatively small, and the ImageNet dataset contains plant images, case 1 presented itself as the best method for use. This approach requires replacing the end of the network with a new fully-connected layer, where its output nodes match the number of classes within the dataset. When training the model on the new dataset, the pre-trained weights are frozen, preventing updates. Allowing the model to use the best pre-trained weights while learning features from the new dataset.

Initially, the three models only used one fully-connected layer at the end of the network. However, this was increased to three layers, adding non-linearity to the features to improve the model's generalisation capabilities. These layers are called a classifier and determine the models predictions when classifying input images. The first layer accepts the previous convolutional layer, or pooling layers, output as input and outputs the first hidden node size (512). The second layer takes those 512 nodes and outputs 256 values (as per the hyperparameters). These two layers values are passed through a ReLU activation function, followed by a dropout layer to prevent overfitting, before being accepted into the final layer and passed through a log Softmax function, where probabilities of the 35 classes are calculated and returned.

```
GoogLeNet Classifier(
  (hidden_layers): ModuleList(
    (0): Linear(in_features=1024, out_features=512, bias=True)
    (1): Linear(in_features=512, out_features=256, bias=True)
  )
  (out): Linear(in_features=256, out_features=35, bias=True)
  (dropout): Dropout(p=0.3, inplace=False)
)
```

Figure 4.2: An example of GoogLeNet's classifier structure with a 30% dropout rate.

The classifier is stored within a class object that inherits PyTorch's base neural network module, containing two functions: init and forward. The init function creates instances of the hidden layers and dropout layers. The hidden layers are generated using a dynamic approach by simply adding or removing an integer value to a list of hidden layer node sizes. This flexible approach provides scalability when needing to increase (or reduce) the number of layers within the classifier. The forward function performs a forward pass on the network, where the hidden layers are iterated over, passing the images through each layer and their respective activation functions.

With the models created and modified, training on the dataset of edible wild plants could begin. Firstly, the model is moved to GPU (if available) using cuda. Next, both the Adam optimizer and negative log-likelihood (NLL) loss function is initialised. NLL loss combines perfectly with the log Softmax activation function as both are components of cross-entropy loss, which effectively measures the performance of models that require classification of multiple classes, specifically when the output is of probability values. The Adam optimizer is a technique for stochastic optimization, an extension to gradient descent, that computes individual adaptive learning rates for updating network weights. Given a learning rate, Adam automatically adapts the value for each network weight while the training is in progress, reducing the training complexity and improving the model's convergence time (Kingma and Ba, 2017). The learning rate is a small value between 0 and 1 that updates the network weights during training. Model training aims to find the best optimal weight values that provide the highest classification accuracy, commonly found at a local minimum, where the learning rate is crucial at making this happen. Having the right-sized

learning rate helps prevent the network from converging too quickly or getting stuck at one value.

A custom training function is used to train the models individually on the training data and evaluate them against the validation data. The training continues for a maximum of 20 epochs or until an early stopping condition is reached. Early stopping is a technique that prevents a model from overfitting a dataset by stopping after several iterations, known as a patience value. The patience value is set to 5 and will only increment when the validation loss during training is higher than the previous one. This functions goal is to reduce the validation loss to its smallest value without becoming larger than the training loss. If the training loss becomes higher than the validation loss, the model is overfitting the data. During training, each batch of images and labels are passed onto the GPU using cuda, and the optimizers gradients are set to 0 to avoid accumulating them on subsequent backpropagation passes. The model performs a forward pass on the images, and the predictions are compared against the labels using the criterion to calculate the loss value. The loss value is backpropagated through the network, and the optimizer updates the network weights. After a given number of batches (half the total number per epoch), the model is evaluated against the validation data, which provides its loss values and accuracy. If the validation loss is lower than the lowest validation loss, the model's weights (parameters) and list of train and validation losses are stored in a saved models folder, and the best validation loss value is updated. This process continues until training concludes.

The final section of the notebook involves graphics of the trained model's performance. These are represented as loss comparison graphs, ROC curves, and confusion matrices. Before plotting the diagrams, the model's predictions (0s or 1s), labels, and predication probabilities are calculated using a custom predict function. The function iterates over each batch of test data and performs a forward pass through the network using the data, which returns the predictions in a normalized format. Next, these predictions are exponentially increased to create their probabilities, unnormalizing the data, where lastly, the maximum values are taken and returned by the function. Using the Scikit plot library, both the confusion matrices and ROC curves

are plotted. The ROC curve requires the model probabilities and labels, while the confusion matrix requires the model predictions and labels. For this section, these plots have been selectively ignored from this report and are within the respective notebook. However, similar graphs of the best performing models are evaluated in section 4.5.

4.3.2 Parameter Tuning

The second notebook focuses on tuning the models using seven different parameters, creating 12 variants of each model (36 total). The notebook combines functions from the utilities and classifier classes used from the previous notebook into a new tuner class, condensing the notebook and focusing on the core elements of tuning the models. Similar to the initial implementation, the hyperparameters used are set first before moving onto the model tuning. The parameters include: 1000 epochs, a learning rate of 0.001, three batch sizes of 32, 64, and 128, a split size of 15%, and four sets of hidden node sizes at [256, 256], [512, 256], [512, 512], and [1024, 516]. Additionally, the models train using the same criterion and optimizer as the previous notebook (NLL loss and Adam).

Next, the wild edible plant dataset is stored within an image folder object from the torchvision library that uses the same transforms as the previous notebook, created from the tuner classes set data function. The notebook continues by iterating over the three batch sizes and divides the dataset into their respective data loaders (training, validation, and testing). Per batch size iteration, each hidden layer size is iterated over and used to train the three model architectures on the training data, which gets evaluated against the validation data. For example, the GoogLeNet architecture is trained on 32 batches and hidden node sizes [256, 256]. Once training concludes for that model, the MobileNet v2 architecture trains on the same parameters (32 batches and [256, 256] hidden node sizes), and then the same parameters are used to train the ResNet-34 architecture. The tuning process continues for the remaining parameters on all three models. During this process, all parameter variants best model version (lowest validation loss) is saved for each of the models, totalling 36 saved models for performance evaluation.

The final notebook combines functions from the tuner and plotter classes, providing visualisations of the best performing models. The results are discussed in section 4.5, while the remainder of this section describes the process to achieve those results. The process is almost identical to the second notebook, except each models' predictions and statistics are calculated instead of tuning them. As before, the data is split into its respective data loaders when iterating through the three different batch sizes. Next, using a function from the tuner class, the hidden node sizes are looped over, and a copy of the pre-trained models are set with the respective size. Iterating over the models, one at a time, the saved version is loaded based on the components provided (model name, batch size and hidden node sizes) and makes predictions on the test dataset. The predictions are then used to calculate the classification accuracy, top-1 error rate, top-5 error rate, precision, recall, and F1-score. Storing these statistics within a dictionary and concatenating them into a list containing every set of statistics, they are converted into a Pandas dataframe (master) for simple data manipulation and tabling. Lastly, the dataframe is split into model groups, as seen in Table 4.2, 4.3, and 4.4, and condensed into a final table of the best performing models (Table 4.5).

The best models are saved using a tuner class function called save best models, which extracts the models with the highest accuracy from the master dataframe, storing the model's statistics and parameters, in addition to other information about each model. A complete list of the details saved includes a list of train and valid losses, the batch size used, hidden layer sizes used, statistics, weight values (parameters), predictions, prediction labels, and prediction probabilities. The models are loaded back into the notebook, using the tuner class function called load best models, to include the additional information to the model object in preparation for visualising the results. After loading the models, the notebook concludes with graphics of the training and validation losses against the number of epochs, ROC curves, confusion matrices, and model predictions vs true labels for each model. These graphics are discussed further in section 4.5.

4.4 Testing

Throughout the artefact's development, constant testing of its components followed a unit testing approach. Unit testing is a type of software testing used to validate individual units, one coding block at a time, while the software is developed (Software Testing Fundamentals, 2020). Fortunately, notebooks provide coding cells. These cells are equivalent to units in unit testing and were fundamental in creating the artefact. During development, class functions were constructed and tested within a coding cell. If an error occurred, the code would be debugged immediately and validated within the same coding cell. This approach provided a consistent and efficient method for creating the artefact, drastically reducing its development time.

While this approach ensures that the code works correctly, it cannot identify if a CNN model performs to the best of its ability. To accommodate this, training and validation losses were calculated during the models training process. These metrics reduce the chance of model overfitting, as mentioned in previous sections, and also aids in monitoring if the model is training properly.

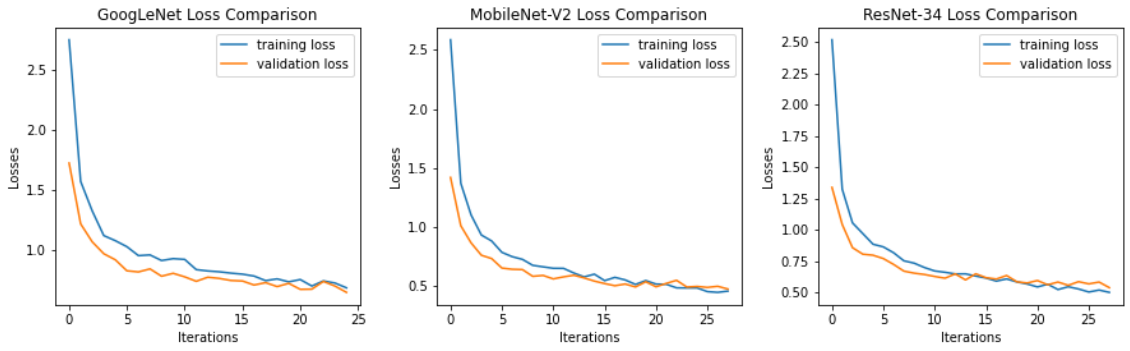

Figure 4.3: The best performing model architecture's learning curves, comparing the training and validation losses against each other over the total number of training epochs.

Figure 4.3 provides an example of learning curve diagrams of the best performing models. When examining these diagrams, it is clear that there is a sharp decline in both the training and validation losses. The graphs represent a balance between an underfit and overfit model, showing that the models have trained correctly and achieve great results.

4.5 Results

The artefact contains 36 model variants, consisting of a combination of seven hyperparameters (three batch sizes and four sets of hidden node sizes), trained, tuned, and evaluated against six performance metrics. The model's statistics are split into four tables, three relating to each unique architecture (12 models per table) and one that contains the best performing model of each of those tables (3 total). This section first discusses the results of each model separately before moving onto the best performing models.

Batch	Hidden sizes	Accuracy	Top-1 error	Top-5 error	Precision	Recall	F1-score
32	256, 256	67.47%	32.53%	8.27%	53.57%	54.22%	53.89%
32	512, 256	68.89%	31.11%	7.75%	57.33%	75.44%	65.15%
32	512, 512	68.08%	31.92%	7.91%	66.13%	48.24%	55.78%
32	1024, 516	71.07%	28.93%	7.18%	77.55%	46.34%	58.02%
64	256, 256	70.58%	29.42%	7.47%	58.97%	64.79%	61.75%
64	512, 256	71.43%	28.57%	7.18%	80.00%	41.67%	54.79%
64	512, 512	71.23%	28.77%	6.17%	72.41%	43.75%	54.55%
64	1024, 516	72.03%	27.97%	6.26%	70.77%	74.19%	72.44%
128	256, 256	71.59%	28.41%	6.13%	78.95%	48.91%	60.40%
128	512, 256	74.29%	25.71%	6.05%	72.13%	73.33%	72.73%
128	512, 512	72.72%	27.28%	6.05%	76.27%	42.86%	54.88%
128	1024, 516	74.21%	25.79%	5.65%	61.45%	65.38%	63.35%

Table 4.2: All 12 performance results for the GoogLeNet architecture variants.

Table 4.2 identifies the 12 model variants for the GoogLeNet architecture. When analysing the table, it is clear that as the model's batch size and hidden node sizes increase, the classification accuracy also increases. Interestingly, this is somewhat similar for precision and recall. The two metrics achieve higher values as the hidden sizes increases. However, these values are slightly arbitrary, where the highest values are 72.13% and 73.33%, respectively, with a batch size of 128 and hidden layer sizes of [512, 256]. Additionally, the results show that the top-5 error rate only decreases

between approximately 1 to 3%, which is dependent on the size of the hidden nodes, where the larger sizes give larger decreases.

Batch	Hidden sizes	Accuracy	Top-1 error	Top-5 error	Precision	Recall	F1-score
32	256, 256	76.31%	23.69%	4.96%	71.43%	76.27%	73.77%
32	512, 256	78.13%	21.87%	4.72%	75.81%	53.41%	62.67%
32	512, 512	77.44%	22.56%	4.64%	57.14%	48.28%	52.34%
32	1024, 516	75.59%	24.41%	4.48%	64.79%	65.71%	65.25%
64	256, 256	79.42%	20.58%	4.16%	78.18%	51.81%	62.32%
64	512, 256	80.10%	19.90%	3.79%	52.68%	73.75%	61.46%
64	512, 512	81.36%	18.64%	3.67%	64.77%	67.06%	65.90%
64	1024, 516	80.75%	19.25%	3.43%	79.37%	56.82%	66.23%
128	256, 256	80.43%	19.57%	4.12%	61.80%	82.09%	70.51%
128	512, 256	82.85%	17.15%	3.03%	73.68%	73.68%	73.68%
128	512, 512	82.32%	17.68%	3.23%	80.30%	61.63%	69.74%
128	1024, 516	81.11%	18.89%	3.23%	70.51%	59.78%	64.71%

Table 4.3: All 12 performance results for the MobileNet v2 architecture variants.

Table 4.3 focuses on the 12 model variants for the MobileNet v2 architecture. Complementary to GoogLeNet, the classification accuracy increases as the model's batch size and hidden node sizes increase. However, the middle ground hidden node sizes are more favourable, achieving higher accuracies for each batch size. The architecture achieves higher precision results than recall, across most of the batch and hidden node sizes, with 128 and [512, 256] achieving the same value of 73.68% for both metrics. As with GoogLeNet, these parameters are the most optimal for the architecture, providing some of the best overall results. Furthermore, the top-5 error rate decreases in the same manner as GoogLeNet, with a balance across all sizes at approximately 2%.

Batch	Hidden sizes	Accuracy	Top-1 error	Top-5 error	Precision	Recall	F1-score
32	256, 256	75.22%	24.78%	5.49%	76.79%	60.56%	67.72%
32	512, 256	74.29%	25.71%	5.04%	73.33%	72.13%	72.73%
32	512, 512	75.71%	24.29%	4.36%	72.06%	66.22%	69.01%
32	1024, 516	74.13%	25.87%	5.25%	73.02%	51.69%	60.53%
64	256, 256	76.31%	23.69%	4.60%	67.07%	57.29%	61.80%
64	512, 256	78.81%	21.19%	3.91%	77.94%	71.62%	74.65%
64	512, 512	76.27%	23.73%	4.96%	68.35%	49.09%	57.14%
64	1024, 516	78.01%	21.99%	4.04%	75.76%	61.73%	68.03%
128	256, 256	76.63%	23.37%	4.20%	75.00%	60.00%	66.67%
128	512, 256	79.38%	20.62%	3.75%	71.79%	65.12%	68.29%
128	512, 512	78.85%	21.15%	3.67%	77.63%	72.84%	75.16%
128	1024, 516	80.35%	19.65%	3.39%	82.35%	86.15%	84.21%

Table 4.4: All 12 performance results for the ResNet-34 architecture variants.

Table 4.4 represents the 12 model variants for the final architecture, ResNet-34. Comparatively to GoogLeNet and MobileNet v2, the classification accuracy increases as both hyperparameters sizes increase. The top-5 error rate decreases by approximately 2%, achieving its best results with the highest batch and hidden node sizes. When examining the precision and recall statistics, ResNet provides a better balance between both metrics, comparative to the other two architectures. Achieving the highest precision and recall values required, a batch size of 128 and [1024, 516] hidden node sizes, obtaining 82.35% and 86.15%, respectively.

Name	Hidden sizes	Accuracy	Top-1 error	Top-5 error	Precision	Recall	F1-score
GoogLeNet	512, 256	74.29%	25.71%	6.05%	72.13%	73.33%	72.73%
MobileNet v2	512, 256	82.85%	17.15%	3.03%	73.68%	73.68%	73.68%
ResNet-34	1024, 516	80.35%	19.65%	3.39%	82.35%	86.15%	84.21%

Table 4.5: The performance results for the 3 best model variants. These models used a batch size of 128.

The final table (4.5) highlights the best performing models for each of the architectures. These models used the largest batch size tested (128) and varied in hidden node sizes. Interestingly, MobileNet v2 achieves the highest classification accuracy of 82.85% with node sizes [512, 256]. The remaining two architectures, GoogLeNet and ResNet-34, GoogLeNet used identical hidden node sizes as MobileNet v2, and ResNet-34 used larger node sizes of [1024, 516], achieving accuracies of 74.29% and 80.35%, respectively. Unfortunately, GoogLeNet was unable to meet the accuracy requirements, falling short by 5.71%, using the tested parameters. Furthermore, ResNet-34 achieved the highest values for precision and recall, beating MobileNet v2 by a massive 10.53% when comparing the F1-score.

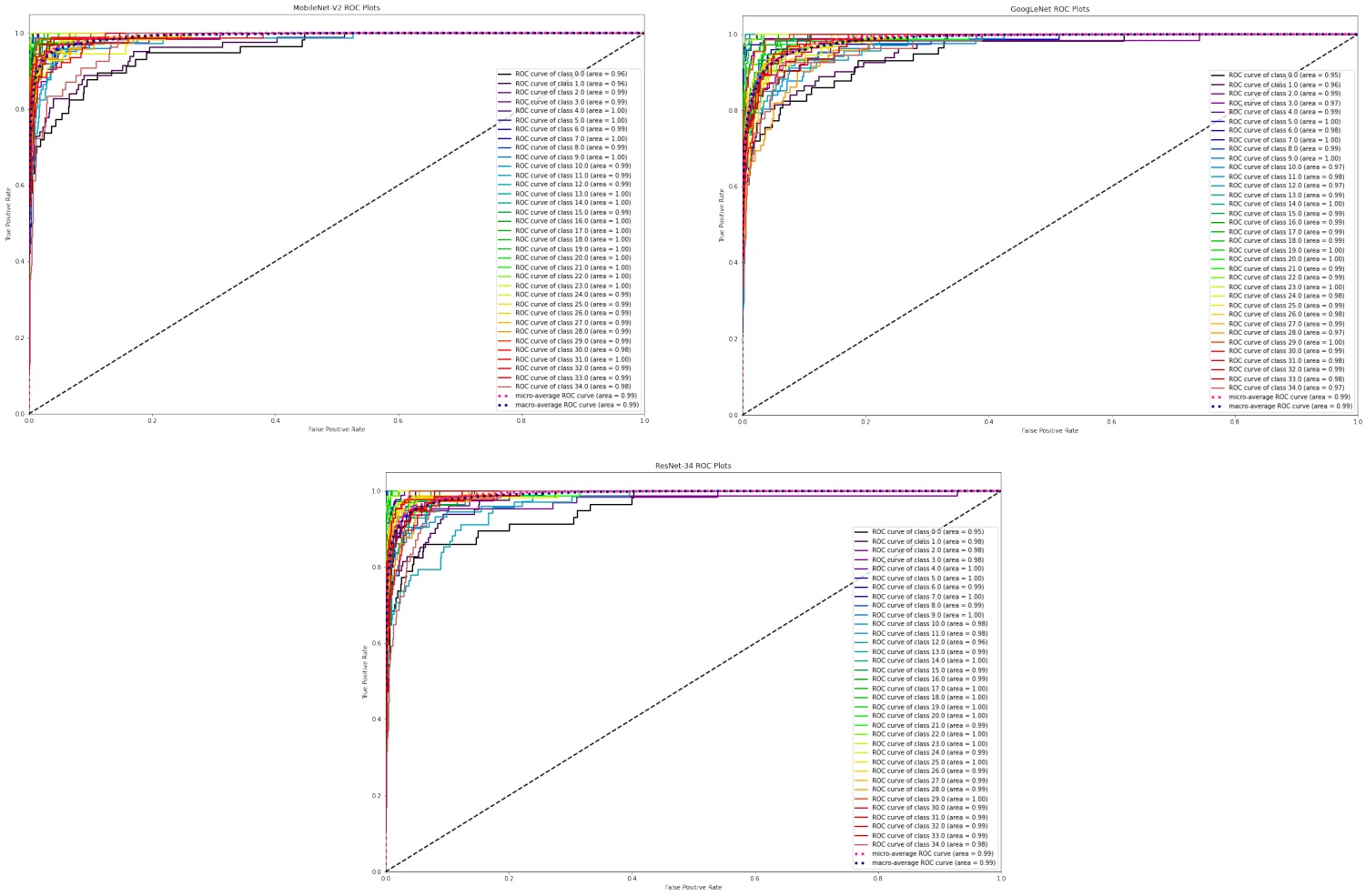

Figure 4.4: ROC curves of the three best performing models.

Analysing the model's further, Figure 4.4 provides a unique insight into each plant class's performance. ROC curves consist of two metrics: a false positive rate (FPR) and a true positive rate (TPR). These graphs describe the model's capability when distinguishing between plant classes. The closer the curve is to 1 (top left corner), the better the model predicts a plant as its correct type. It is clear that MobileNet v2 performances the best across all classes, where only a handful of classes are performing poorly. While ResNet-34 and GoogLeNet have a similar number of poor performing classes. Interestingly, even though GoogLeNet has the lowest classification accuracy, it still performs very well and is slightly better at predicting similar plant classes correctly when compared against ResNet-34.

Additionally, confusion matrices help identify where the model needs to improve when classifying each plant type correctly. Figure 4.5 highlights the best performing model's confusion matrices, covering all 35 plant classes. Examining the matrices, it is clear that MobileNet v2 performs fantastically with some minor difficulty classifying a few plant classes, such as cow parsley against meadowsweet, fennel against cowslip, and geranium against common mallow. ResNet-34 performs nearly as equally effective but struggles to classifier more classes correctly, such as calendula against dandelions and chickweed against garlic mustard. However, GoogLeNet, unfortunately, performs worse than the others and struggles to classify various classes, specifically meadowsweet against cow parsley and common mallow against geranium. Overall, all models have similar difficulties distinguishing meadowsweet against cow parsley and common mallow against geranium.

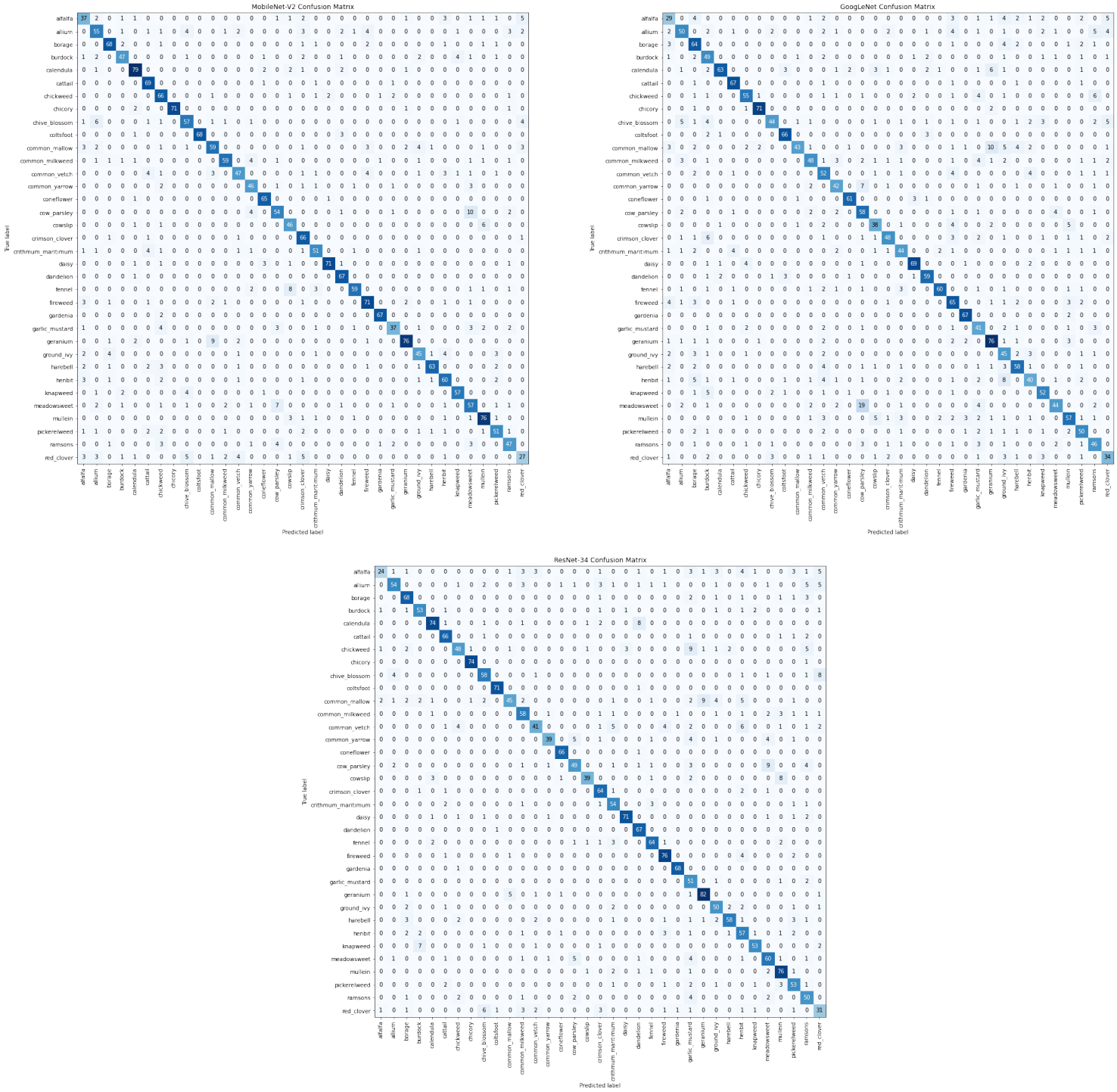

Figure 4.5: The three best performing model's confusion matrices.

Chapter 5

Conclusions

The research within this paper aimed to explore DL techniques to address the food limitations in 2050. Some of the challenges identified include resource scarcity and climate change that are mitigated using disease identification and plant recognition techniques. However, research shows that these techniques alone cannot fully resolve these issues, requiring consideration of alternative methods. One of these methods, explored in this paper, involves identifying wild edible plants in natural environments, providing an opportunity for scientists and researchers the ability to retrieve and examine them as a potential food source.

Using a combination of the Waterfall and Kanban methodologies, and the Jupyter Lab IDE, an artefact was created that involved transfer learning to train and tune three state-of-the-art CNN architectures, GoogLeNet, MobileNet v2, and ResNet-34, on a dataset containing 16,535 images, divided into 35 classes of wild edible plants. When tuning the architectures, seven hyperparameters, consisting of three batch sizes and four sets of hidden node sizes, were used to create 36 models evaluated on the dataset.

The three best models, one for each architecture, achieved classification accuracies of 74.29%, 82.85%, and 80.35%, for GoogLeNet, MobileNet v2, and ResNet-34, respectively. Although MobileNet performed the best for classification accuracy, results showed that the ResNet architecture outperformed the models when evaluating precision and recall, achieving the highest F1-score of 84.21%. Furthermore, the obtained results used the largest batch size tested (128), where MobileNet and GoogLeNet used half the size (512, 256) of the hidden node sizes that ResNet used (1024, 516).

Further evaluation revealed that only a small number of plant classes were incorrectly classified, varying per architecture. Nevertheless, the models struggled to correctly classify two main sets of plant types, meadowsweet against cow parsley and common mallow against geranium.

Overall, determining the best model for classifying wild edible plants depends on the users' requirements. If the aim is to achieve a high classification accuracy and reduce computational complexity, MobileNet is the best architecture to use. Alternatively, if computational complexity doesn't matter, and there is a need for higher precision and recall across multiple class types, ResNet is superior.

Chapter 6

Reflective Analysis

When reflecting on the project, its creation went quite smoothly. Having familiarity with the Python programming language and the PyTorch library helped create a satisfying artefact. Unfortunately, it did encounter some challenging errors to resolve that related to cuda.

The first error prevented accessibility to MobileNet v2's architecture. Removing this architecture would have required a whole project rework, requiring the selection of a new model. While this is simple in practice, it would have required an additional two weeks to a month of work. To avoid this, I spent approximately three days searching forums before finding that the error relates to the latest version of the torchvision module. Initially, I tried to install an older version of the module, but Anaconda and pip failed to find an older version of the package, requiring a more creative solution. A forum post explained that the error originated from the torch hub module and that it was incorrectly caching the required information on my computer. After reading through the documentation about the torch hub module, I found a function that lists all of the usable architectures from a given GitHub page. When passing a link of PyTorch visions master repository to the function, it provided a new error explaining that MobileNet failed accessibility. I managed to find the cached files on my computer and began examining their contents. I found no issues with the code and could not understand why the error was happening. After another day of forum searching, I found that it was possible to forcefully download the required cached files using an older version of the torchvision module. This approach worked like a dream and fixed the issue. To effectively mitigate this issue in future, each notebook

has an optional command to download an older cached version of the torchvision module when required.

The second cuda error involved limited memory. Unfortunately, this error was unavoidable due to the limitations of the GPU memory available. The simplest solution to resolve this required using smaller batch sizes when training, tuning and testing the models. In hindsight, this impacted the model's performance.

Although the project also provided opportunities to extend my programming knowledge, enabling creative solutions to unexpected problems during development. Admittedly, while the project did have a plan, some additional functionality had to be created during the project's development. Some of this functionality included a helper class that converted normalized data back to a readable format, allowing visualisation of the image predictions. Another involved converting the class indices to labels, providing a suitable name for each class within the confusion matrices.

If the project had more time available, the model's goal would change from achieving classification accuracies above 80% to 90%. One approach to accomplish this would be to extend the early stopping patience value from 5 to 10 (or 20). While this increases the chance of overfitting, it also provides the models with a greater likelihood of finding the best local minimum, which can be extremely difficult to find without large quantities of training time. Another approach would be to extend the GPU memory available, allowing consideration of larger batch sizes. Additionally, there would be consideration of increasing the quantity of available training data. However, based on the model's results, additional data may not be required, and instead, further parameter tuning would suffice.

References

Adams, R., Hurd, B., Lenhart, S. and Leary, N. (1998) Effects of global climate change on agriculture: an interpretative review. *Climate Research*, 11(1) 19–30. Available from https://www.int-res.com/abstracts/cr/v11/n1/p19-30/ [accessed 19 January 2021].

Affouard, A., Goeau, H., Bonnet, P., Lombardo, J-C. and Joly, A. (2017) *Pl@ntNet app in the era of deep learning.* OpenReview. Available from https://openreview.net/forum?id=HJVJpENFg [accessed 22 January 2021].

Ahmad, M.O., Markkula, J. and Oivo, M. (2013) Kanban in Software Development: A Systematic Literature Review. In: *Software Engineering and Advanced Applications (SEAA)*, Santander, Spain, 4-6 September. IEEE Computer Society, 9–16. Available from https://www.researchgate.net/publication/260739586_Kanban_in_Software_Development_A_Systematic_Literature_Review [accessed 28 January 2021].

Al-Masri, A. (2019) *How Does Back-Propagation in Artificial Neural Networks Work?* Available from https://towardsdatascience.com/how-does-back-propagation-in-artificial-neural-networks-work-c7cad873ea7 [accessed 2 March 2021].

Ampatzidis, Y., De, L. and Luvisi, A. (2017) iPathology: Robotic Applications and Management of Plants and Plant Diseases. *Sustainability*, 9(6) 1010. Available from https://www.mdpi.com/2071-1050/9/6/1010 [accessed 23 January 2021].

Anaconda (Undated) *Individual Edition - Your data science toolkit.* Anaconda. Available from https://www.anaconda.com/products/individual [accessed 3 February 2021].

Aydinalp, C. and Cresser, M. (2008) The Effects of Global Climate Change on Agriculture. *American-Eurasian J. Agric. Environ. Sci.*, 3. Available from https://www.researchgate.net/publication/238091112_The_Effects_of_Global_Climate_Change_on_Agriculture [accessed 19 January 2021].

Barré, P., Stöver, B., Müller, K. and Steinhage, V. (2017) LeafNet: A computer vision system for automatic plant species identification. *Ecological Informatics*, 40 50–56. Available from http://www.sciencedirect.com/science/article/pii/S1574954116302515 [accessed 23 January 2021].

Bulajic, A., Sambasivam, S. and Stojic, R. (2013) An Effective Development Environment Setup for System and Application Software. *Issues in Informing Science and Information Technology*, 10 037–066. Available from https://www.informingscience.org/Publications/1795 [accessed 2 February 2021].

Ching, T., Himmelstein, D., Beaulieu-Jones, B., Kalinin, A., Do, B.T., Way, G., Ferrero, E., Agapow, P.-M., Zietz, M., Hoffman, M., Xie, W., Rosen, G., Lengerich, B., Israeli, J., Lanchantin, J., Woloszynek, S., Carpenter, A., Shrikumar, A., Xu, J., Cofer, E., Lavender, C., Turaga, S., Alexandari, A., Lu, Z., Harris, D., DeCaprio, D., Qi, Y., Kundaje, A., Peng, Y., Wiley, L., Segler, M., Boca, S., Swamidass, S., Huang, A., Gitter, A. and Greene, C. (2018) Opportunities and obstacles for deep learning in biology and medicine. *Journal of The Royal Society Interface*, 15(141) 20170387. Available from https://royalsocietypublishing.org/doi/full/10.1098/rsif.2017.0387 [accessed 16 January 2021].

Conijn, J., Querner, E., Rau, M.-L., Hengsdijk, H., Kuhlman, T., Meijerink, G., Rutgers, B. and Bindraban, P.S. (2011) *Agricultural resource scarcity and distribution: a case study of crop production in Africa.* ResearchGate. Available from https://www.researchgate.net/publication/254834493_Agricultural_resource_scarcity_and_distribution_a_case_study_of_crop_production_in_Africa [accessed 18 January 2021].

Cook, A., Chakraborty, A., Leonard, M., Serrano, L., Camacho, C., Sheahan, D., Yadav, C., Delgado, J. and Morales, M. (Undated) *Deep Reinforcement Learning Nanodegree Program* [MOOC]. Udacity. Available from https://www.udacity.com/course/deep-reinforcement-learning-nanodegree--nd893 [accessed 08 February 2021].

Cova, T. and Pais, A. (2019) Deep Learning for Deep Chemistry: Optimizing the Prediction of Chemical Patterns. *Frontiers in Chemistry*, 7. Available from https://www.frontiersin.org/articles/10.3389/fchem.2019.00809/full [accessed 16 January 2021].

Despa, M. (2014) *Comparative study on software development methodologies.* Available from https://dbjournal.ro/archive/17/17_4.pdf [accessed 28 January 2021].

Dey, N., Mishra, G., Kar, J., Chakraborty, S. and Nath, S. (2014) A survey of image classification methods and techniques. In: *2014 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*. Kanyakumari District, India, 10-11 July 2014. ResearchGate: IEEE Computer Society, 554-557. Available from https://www.researchgate.net/publication/269984702_A_survey_of_image_classification_methods_and_techniques [accessed 27 January 2021].

Drea, S. (2011) The End of the Botany Degree in the UK. *Bioscience Education*, 17(1) 1–7. Available from https://www.tandfonline.com/doi/full/10.3108/beej.17.2 [accessed 13 January 2021].

Eason, O. (2016) *Information Systems Development Methodologies Transitions: An Analysis of Waterfall to Agile Methodology.* Main Campus: University of New Hampshire. Available from https://scholars.unh.edu/cgi/viewcontent.cgi?article=1288&context=honors [accessed 29 January 2021].

Esteva, A., Robicquet, A., Ramsundar, B., Kuleshov, V., DePristo, M., Chou, K., Cui, C., Corrado, G., Thrun, S. and Dean, J. (2019) A guide to deep learning in healthcare. *Nature Medicine*, 25(1) 24–29. Available from http://www.nature.com/articles/s41591-018-0316-z [accessed 16 January 2021].

Gikunda, P. and Jouandeau, N. (2019) Modern CNNs for IoT Based Farms. *arXiv:1907.07772 [cs, stat]*, Available from http://arxiv.org/abs/1907.07772 [accessed 24 November 2020].

Gogul, I. and Kumar, V. (2017) Flower species recognition system using convolution neural networks and transfer learning. In: *2017 Fourth International Conference on Signal Processing, Communication and Networking (ICSCN)*. March 2017 IEEE Computer Society, 1–6. Available from https://ieeexplore.ieee.org/abstract/document/8085675 [accessed 24 January 2021].

Google (Undated) *Classification: Precision and Recall | Machine Learning Crash Course.* Available from https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall [accessed 14 February 2021].

Goyal, K. (2021) *Top 10 Deep Learning Frameworks in 2021 You Can't Ignore.* Available from https://www.upgrad.com/blog/top-deep-learning-frameworks/ [accessed 5 February 2021].

Hansen, C. (2019) *Neural Networks: Feedforward and Backpropagation Explained & Optimization.* Available from https://mlfromscratch.com/neural-networks-explained/ [accessed 2 March 2021].

He, K., Zhang, X., Ren, S. and Sun, J. (2015) Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]*, Available from http://arxiv.org/abs/1512.03385 [accessed 13 January 2021].

Hijazi, S., Kumar, R. and Rowen, C. (Undated) *Using Convolutional Neural Networks for Image Recognition.* 12. Available from https://ip.cadence.com/uploads/901/cnn_wp-pdf [accessed 7 February 2021].

Jain, V. (2019) *Everything you need to know about "Activation Functions" in Deep learning models.* Towards Data Science. Available from https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253 [accessed 9 February 2021].

JetBrains (Undated) *PyCharm Features.* Available from https://www.jetbrains.com/pycharm/features/ [accessed 3 February 2021].

Jin, T., Hou, X., Li, P. and Zhou, F. (2015) A Novel Method of Automatic Plant Species Identification Using Sparse Representation of Leaf Tooth Features. *PLOS ONE*, 10(10) e0139482. Available from https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0139482 [accessed 23 January 2021].

Johnson, H. (2017) Trello. *Journal of the Medical Library Association*, 105. Available from https://www.researchgate.net/publication/315796559_Trello [accessed 30 January 2021].

KanbanFlow (Undated) *KanbanFlow - Kanban Board Features.* Available from https://kanbanflow.com/features [accessed 3 February 2021].

Kaya, A., Keceli, A., Catal, C., Yalic, H., Temucin, H. and Tekinerdogan, B. (2019) Analysis of transfer learning for deep neural network-based plant classification models. *Computers and Electronics in Agriculture*, 158 20–29. Available from http://www.sciencedirect.com/science/article/pii/S0168169918315308 [accessed 25 January 2021].

Keras (Undated) *Keras documentation: About Keras.* Available from https://keras.io/about/ [accessed 5 February 2021].

King, A. (2017) Technology: The Future of Agriculture. *Nature*, 544(7651) S21–S23. Available from https://www.nature.com/articles/544S21a [accessed 17 January 2021].

Kingma, D. and Ba, J. (2017) Adam: A Method for Stochastic Optimization. *arXiv:1412.6980 [cs]*, Available from http://arxiv.org/abs/1412.6980 [accessed 12 March 2021].

Koehrsen, W. (2018) *Beyond Accuracy: Precision and Recall.* Available from https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c [accessed 14 February 2021].

Lasseck, M. (2017) *Image-based Plant Species Identification with Deep Convolutional Neural Networks.* 11. Available from http://ceur-ws.org/Vol-1866/paper_174.pdf [accessed 13 January 2021].

Lauer, C. (2015) *Fewer students study botany, more plant collections closing.* Available from https://phys.org/news/2015-05-students-botany.html [accessed 13 January 2021].

LeCun, Y., Bengio, Y. and Hinton, G. (2015) Deep learning. *Nature*, 521(7553) 436–444. Available from https://www.nature.com/articles/nature14539 [accessed 16 January 2021].

Lee, S., Chan, C., Wilkin, P. and Remagnino, P. (2015) Deep-Plant: Plant Identification with convolutional neural networks. *arXiv:1506.08425 [cs]*, Available from http://arxiv.org/abs/1506.08425 [accessed 23 January 2021].

Li, Y., Huang, C., Ding, L., Li, Z., Pan, Y. and Gao, X. (2019) Deep learning in bioinformatics: Introduction, application, and perspective in the big data era. *Methods*, 166 4–21. Available from http://www.sciencedirect.com/science/article/pii/S1046202318303256 [accessed 16 January 2021].

Liakos, K., Busato, P., Moshou, D., Pearson, S. and Bochtis, D. (2018) Machine Learning in Agriculture: A Review. *Sensors*, 18(8) 2674. Available from https://www.mdpi.com/1424-8220/18/8/2674 [accessed 24 November 2020].

Malik, F. (2019) *Neural Networks Bias And Weights - Understanding The Two Most Important Components.* Available from https://medium.com/fintechexplained/neural-networks-bias-and-weights-10b53e6285da [accessed 7 February 2021].

Mancosu, N., Snyder, R., Kyriakakis, G. and Spano, D. (2015) Water Scarcity and Future Challenges for Food Production. *Water 2015*, 7(3) 975–992. Available from https://www.mdpi.com/2073-4441/7/3/975 [accessed 18 January 2021].

MeisterTask (Undated) *MeisterTask Features.* Available from https://www.meistertask.com/pages/features/ [accessed 3 February 2021].

Min, S., Lee, B. and Yoon, S. (2017) Deep learning in bioinformatics. *Briefings in Bioinformatics*, 18(5) 851–869. Available from https://doi.org/10.1093/bib/bbw068 [accessed 16 January 2021].

Mitrović, K. and Milošević, D. (2019) Flower Classification with Convolutional Neural Networks. In: *2019 23rd International Conference on System Theory, Control and Computing (ICSTCC)*. October 2019 IEEE Computer Society, 845–850. Available from https://ieeexplore.ieee.org/document/8885580 [accessed 23 January 2021].

Mohanty, S., Hughes, D. and Salathé, M. (2016) Using Deep Learning for Image-Based Plant Disease Detection. *Frontiers in Plant Science*, 7. Available from https://www.frontiersin.org/articles/10.3389/fpls.2016.01419/full [accessed 16 January 2021].

National Geographic Headquarters (Undated) *Food Staple.* Available from http://www.nationalgeographic.org/encyclopedia/food-staple/ [accessed 24 November 2020].

NVIDIA (Undated) *Deep Learning Frameworks.* Available from https://developer.nvidia.com/deep-learning-frameworks [accessed 5 February 2021].

Nwankpa, C., Ijomah, W., Gachagan, A. and Marshall, S. (2018) Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv:1811.03378 [cs]*, Available from http://arxiv.org/abs/1811.03378 [accessed 10 February 2021].

Original Outdoors (Undated) *The Wild Food Directory - foraging guides for the edible plants of the UK.* Available from https://originaloutdoors.co.uk/wild-food-directory/ [accessed 4 November 2020].

O'Shea, K. and Nash, R. (2015) An Introduction to Convolutional Neural Networks. *arXiv:1511.08458 [cs]*, Available from http://arxiv.org/abs/1511.08458 [accessed 7 February 2021].

Partridge, R. (2021) *Wild Edible Plants Dataset.* Available from https://kaggle.com/ryanpartridge01/wild-edible-plants [accessed 22 February 2021].

Pierson, H. and Gashler, M. (2017) Deep learning in robotics: a review of recent research. *Advanced Robotics*, 31(16) 821–835. Available from https://doi.org/10.1080/01691864.2017.1365009 [accessed 16 January 2021].

Place, F., Meybeck, A., Colette, L., Young, C., Gitz, V., Dulloo, M., Hall, S., Müller, E., Nasi, R., Noble, A., Spielman, D., Steduto, P. and Wiebe, K. (2013) Food security and sustainable resource use - what are the resource challenges to food security? In: *Food Security Futures Conference*, Dublin, Ireland, 11 April 2013. Available from https://www.researchgate.net/publication/342052908_Food_security_and_sustainable_resource_use_-_what_are_the_resource_challenges_to_food_security [accessed 17 January 2021].

Project Jupyter (2018) *JupyterLab is Ready for Users.* Available from https://blog.jupyter.org/jupyterlab-is-ready-for-users-5a6f039b8906 [accessed 3 February 2021].

PyTorch (Undateda) *PyTorch - From Research to Production.* Available from https://www.pytorch.org [accessed 5 February 2021].

PyTorch (Undatedb) *torchvision.models — Torchvision 0.8.1 documentation.* Available from https://pytorch.org/vision/stable/models.html [accessed 12 February 2021].

Rahman, C., Arko, P., Ali, M., Khan, M., Apon, S., Nowrin, F. and Wasif, A. (2020) Identification and Recognition of Rice Diseases and Pests Using Convolutional Neural Networks. *arXiv:1812.01043 [cs]*, Available from http://arxiv.org/abs/1812.01043 [accessed 21 January 2021].

Ramcharan, A., Baranowski, K., McCloskey, P., Ahmed, B., Legg, J. and Hughes, D. (2017) Deep Learning for Image-Based Cassava Disease Detection. *Frontiers in Plant Science*, 8. Available from https://www.frontiersin.org/articles/10.3389/fpls.2017.01852/full?report=reader [accessed 16 January 2021].

Ren, T., Zhang, Y. and Wang, C. (2019) Identification of Corn Leaf Disease Based on Image Processing. In: *2019 2nd International Conference on Information Systems and Computer Aided Education (ICISCAE)*. September 2019 165–168. Available from https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9075623 [accessed 20 January 2021].

Richardson, A., Signor, B., Lidbury, B. and Badrick, T. (2016) Clinical chemistry in higher dimensions: Machine-learning and enhanced prediction from routine clinical chemistry data. *Clinical Biochemistry*, 49(16) 1213–1220. Available from http://www.sciencedirect.com/science/article/pii/S0009912016301709 [accessed 16 January 2021].

Saha, S. (2018) *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* Available from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 [accessed 7 February 2021].

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. (2019) MobileNetV2: Inverted Residuals and Linear Bottlenecks. *arXiv:1801.04381 [cs]*, Available from http://arxiv.org/abs/1801.04381 [accessed 18 February 2021].

Santos, L., Santos, F., Oliveira, P. and Shinde, P. (2020) *Deep Learning Applications in Agriculture: A Short Review.* 139–151. Available from https://www.researchgate.net/publication/337400899_Deep_Learning_Applications_in_Agriculture_A_Short_Review [accessed 11 January 2021].

Schmitz, A. and Moss, C. (2015) *Mechanized agriculture: machine adoption, farm size, and labor displacement.* Available from https://mospace.umsystem.edu/xmlui/handle/10355/48143 [accessed 13 January 2021].

Sharma, P. (2019) *5 Amazing Deep Learning Frameworks Every Data Scientist Must Know!* Available from https://www.analyticsvidhya.com/blog/2019/03/deep-learning-frameworks-comparison/ [accessed 5 February 2021].

Skalski, P. (2019) *Gentle Dive into Math Behind Convolutional Neural Networks - Mysteries of Neural Networks Part V.* Available from https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9 [accessed 7 February 2021].

Software Testing Fundamentals (2020) *Unit Testing.* Available from https://softwaretestingfundamentals.com/unit-testing/ [accessed 15 March 2021].

Spyder (Undated) *Spyder IDE.* Available from https://www.spyder-ide.org/ [accessed 3 February 2021].

Stone, K. (2020) *Top 10 Free Kanban Boards For Productive Teams in 2020.* Available from https://saaslist.com/blog/free-kanban-boards/ [accessed 3 February 2021].

Streefkerk, R. (2020) *Qualitative vs. Quantitative Research.* Available from https://www.scribbr.com/methodology/qualitative-quantitative-research/ [accessed 6 February 2021].

Suárez-Paniagua, V. and Segura-Bedmar, I. (2018) Evaluation of pooling operations in convolutional architectures for drug-drug interaction extraction. *BMC Bioinformatics*, 19(8) 209. Available from https://doi.org/10.1186/s12859-018-2195-1 [accessed 8 February 2021].

Sun, Y., Liu, Y., Wang, G. and Zhang, H. (2017) *Deep Learning for Plant Identification in Natural Environment.* Available from https://www.hindawi.com/journals/cin/2017/7361042/ [accessed 25 January 2021].

Sünderhauf, N., Brock, O., Scheirer, W., Hadsell, R., Fox, D., Leitner, J., Upcroft, B., Abbeel, P., Burgard, W., Milford, M. and Corke, P. (2018) The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, 37(4–5) 405–420. Available from https://doi.org/10.1177/0278364918770733 [accessed 16 January 2021].

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A. (2014) Going Deeper with Convolutions. *arXiv:1409.4842 [cs]*, Available from http://arxiv.org/abs/1409.4842 [accessed 13 January 2021].

Tan, C., Sun, F., Kong, T., Zhang, W., Yang, C. and Liu, C. (2018) A Survey on Deep Transfer Learning. *arXiv:1808.01974 [cs, stat]*, Available from http://arxiv.org/abs/1808.01974 [accessed 24 January 2021].

Ünal, Z. (2020) Smart Farming Becomes Even Smarter With Deep Learning—A Bibliographical Analysis. *IEEE Access*, 8 105587–105609. Available from https://ieeexplore.ieee.org/abstract/document/9108212 [accessed 25 January 2021].

Vasconcellos, P. (2018) *Top 5 Python IDEs For Data Science.* Available from https://www.datacamp.com/community/tutorials/data-science-python-ide [accessed 2 February 2021].

Walter, A., Finger, R., Huber, R. and Buchmann, N. (2017) Smart farming is key to developing sustainable agriculture. *Opinion*, 1–3. Available from https://www.researchgate.net/publication/317573838_Opinion_Smart_farming_is_key_to_developing_sustainable_agriculture [accessed 13 January 2021].

Welch, S. (2020) *Pytorch vs. TensorFlow: What You Need to Know.* Available from https://blog.udacity.com/2020/05/pytorch-vs-tensorflow-what-you-need-to-know.html [accessed 5 February 2021].

Wild Food UK (Undated) *Wild Plants in the UK: British Hedgerow Food & Foraging Guide.* Available from https://www.wildfooduk.com/wild-plant-guide/ [accessed 4 November 2020].

Yalcin, H. and Razavi, S. (2016) Plant classification using convolutional neural networks. In: *2016 Fifth International Conference on Agro-Geoinformatics (Agro-Geoinformatics).* July 2016 1–5. Available from https://ieeexplore.ieee.org/document/7577698 [accessed 13 January 2021].

Yamashita, R., Nishio, M., Do, R.K.G. and Togashi, K. (2018) Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4) 611–629. Available from https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9 [accessed 7 February 2021].

Zapier (Undated) *The 11 Best Kanban Apps to Build Your Own Productivity Workflow.* Available from https://zapier.com/blog/best-kanban-apps/ [accessed 3 February 2021].

Zhang, X., Qiao, Y., Meng, F., Fan, C. and Zhang, M. (2018) Identification of Maize Leaf Diseases Using Improved Deep Convolutional Neural Networks. *IEEE Access*, 6 30370–30377. Available from https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8374024 [accessed 20 January 2021].

Total document word count (excluding appendices): 14,961

Appendix A

CNN Architectures

Figure A.1: ResNet-34 Architecture

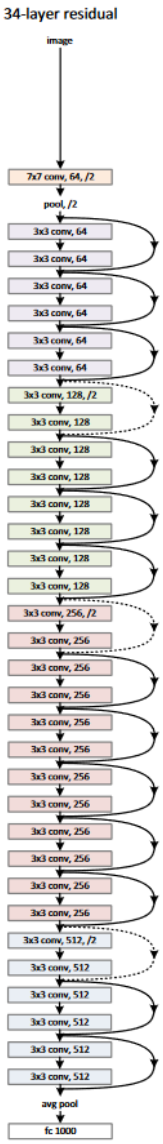

Figure A.2: GoogLeNet Architecture