UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE BIOLOGIA ANIMAL

Plant species identification through leaf venation extraction and CNNs

Jorge Miguel Viana da Cruz

Mestrado em Bioinformática e Biologia Computacional

Dissertação orientada por:

Professor Doutor Luís Miguel Parreira e Correia

Doutora Maria Cristina Reis de Lima Duarte

2020

In memory of David Risca.

Rest in peace, my friend.

Acknowledgements

First and foremost, I would like to thank my supervisors, Professor Luís Correia and Doutora Maria Cristina Duarte. Not only was their guidance and support invaluable, but without them this project simply would not have been possible. To my parents, Damião and Carolina, that although far away provided an unending stream of encouragement and love. To my brother Pedro for providing valuable moments of decompression and entertainment. To my girlfriend Mafalda for being an absolute pillar in my life for years. Big thanks to the friends I made along this journey: André Neves, Sofia Conceição, Fábio Neves, Pedro Santos, João Colaço, Nuno Lopes and José Ferrão. We shared many great moments of comradery. To my volleyball team, and especially to our Captain, Jorge Martins, for the crucial physical and psychological benefits of smashing a ball onto the floor. Finally, big thanks to my cat, Alaska, for biting my hand as I try to type this.

Abstract

The decline in the number of plant taxonomy experts is a known issue. Delegating part of the identification work of taxonomists to machine learning models would help reduce the workload on the dwindling number of available personnel. This project aims to test the concept of classifying four different species of plants solely by the venation network of its leaves. Specifically, we create a convolutional deep learning neural network that attempts to learn how to distinguish the distinct species based on the available augmented dataset of cleared leaves. Cleared leaves are leaves in which the venation network is rendered visible, by specific chemical processes and/or by other methods such as X-ray. We use an augmented dataset because of the scarcity of images of cleared leaves from each class.

Tests were run with different parameters to test the model's ability to predict the correct class with accuracy, and with other metrics. The model was tested on previously unseen images to ensure that it was not memorizing the training images. The results obtained were positive for the parameters selected through trial and error testing: an average testing accuracy of around 79.3% for the final set of parameters.

These results further suggest, as other studies before it, that it might be possible to rely on the venation network as an identifying characteristic for plants, although more large scale studies with more classes and significantly more data are necessary to obtain better support for the hypothesis.

Keywords: Plant Identification, Leaf Venation, Deep Learning, Image Processing

Resumo

O declínio no número de especialistas em taxonomia de plantas é um problema conhecido. Delegar parte do trabalho de identificação dos taxonomistas a modelos de aprendizagem automática ajudaria a reduzir o trabalho do número, cada vez menor, de profissionais disponível. Este projeto tem como objetivo testar a possibilidade de identificar quatro espécies de plantas diferentes exclusivamente pela nervação das suas folhas. Especificamente, criamos uma rede neuronal convolucional de aprendizagem profunda que tenta aprender como distinguir diferentes espécies com base no conjunto de dados aumentado de folhas diafanizadas disponíveis. Folhas diafanizadas são folhas que foram submetidas a processos específicos (como métodos químicos e/ou raio-X) para permitir visualizar não só as nervuras principais, mas também nervuras menores. Devido à escassez de imagens originais de folhas diafanizadas de cada classe, usamos um conjunto de dados aumentado.

Os testes foram executados com diferentes parâmetros para testar a capacidade do modelo de prever a classe correta com precisão, e com outras métricas. O modelo foi testado em imagens não utilizadas anteriormente para se assegurar que as imagens de treino não estavam a ser memorizadas. Os resultados obtidos foram positivos para os parâmetros selecionados nos testes de tentativa e erro: uma precisão média de teste de cerca de 79,3% para o conjunto final de parâmetros.

Estes resultados sugerem, como aliás outros estudos já o vêm apontando, que pode ser possível utilizar o padrão de nervação como uma característica para a identificação de plantas, embora mais estudos em larga escala, com mais classes e significativamente mais dados, sejam necessários para obter uma resposta mais confiante para a hipótese.

Palavras-chave: Identificação de Plantas, Nervação da Folha, Aprendizagem Profunda, Processamento de Imagem

Resumo alargado

Segundo estimativas recentes, existem cerca de 380 000 espécies de plantas vasculares, com novas espécies a serem descobertas regularmente. A identificação rigorosa de espécies é um procedimento indispensável a um elevado número de atividades, como ações de conservação, estudos de impacto ambiental, avaliação do uso de plantas para fins económicos (e.g. alimentares e medicinais), controle de espécies infestantes e invasoras, controle do comércio de espécies ameaçadas, não esquecendo, obviamente, os estudos científicos de cariz taxonómico.

O processo de identificação de plantas recai sobre um número cada vez mais escasso de taxonomistas, o que, a par da enorme diversidade vegetal, torna do maior interesse o desenvolvimento de métodos automáticos que possam contribuir e apoiar esta tarefa. Neste sentido, várias linhas de investigação têm sido desenvolvidas nos últimos anos. De entre elas, o recurso a modelos de aprendizagem automática tem obtido resultados promissores. No entanto, é reconhecido, há, ainda, um longo caminho a percorrer.

Os métodos usuais para a identificação de plantas baseiam-se nas características dos órgãos reprodutores (i.e., flores e frutos). Contudo, outras estruturas podem ser utilizadas. É o caso das folhas, estruturas vegetativas com um papel fundamental em processos fisiológicos, cujas características são usadas como complemento no processo de identificação das espécies. Devido a este interesse, elas têm sido utilizadas em modelos de aprendizagem automática. A maioria destes modelos tem-se focado no contorno destas estruturas, no entanto, esta característica pode variar de forma significativa numa mesma espécie, dependendo, por exemplo, da idade da planta ou do posicionamento da folha; acresce que, em certos casos, o contorno da folha pode estar comprometido por danos causados nas margens da folha por pequenos predadores. São poucos os projetos que exploram a utilização de modelos de aprendizagem automática para a classificação de plantas exclusivamente através da rede de nervação (sem informação de contorno, cor, entre outras características) e nenhum deles usa um modelo de aprendizagem profunda.

Neste projeto implementámos um *script* em *Python* que decorre automaticamente em duas fases: uma fase de processamento de imagem e outra de treino e teste de um modelo de aprendizagem profunda que tenta classificar plantas através de uma parte central da rede de nervação, extraída pela fase anterior. É vantajoso ter um *script* automático para reduzir as tarefas do utilizador, o que diminui o erro humano no processo. O *script* é também facilmente adaptável para alguém com competências mínimas na área de aprendizagem automática.

A base de dados original, deste projeto, consistia em folhas obtidas localmente e, posteriormente, secas de acordo com os procedimentos usados na herborização de espécimes vegetais. Infelizmente, não foi possível extrair as nervuras destas folhas com qualidade: era possível obter as nervuras principais, mas o detalhe das nervuras secundárias e, sobretudo, das terciárias e quaternárias ficava, muitas vezes, perdido. Em consequência, optámos por usar um repositório de imagens de folhas, *ClearedLeavesDB*, que contém folhas que foram submetidas a um tratamento químico denominado diafanização e, posteriormente, radiografadas. A partir destas imagens é possível extrair o padrão de nervação com

elevada qualidade. Deste repositório foram retiradas 48 imagens de quatro espécies (12 imagens de cada): *Betula pendula*, *Bridelia scleroneura*, *Cassia sieberiana* e *Quercus ilex*. Uma dessas imagens por espécie foi colocada de parte para efeitos de teste.

O *script* começa então por localizar as pastas de imagens no sistema de ficheiros, identifica o número de espécies que existem e o número de imagens por espécie. Cada uma dessas imagens passa por uma função que vai extrair um quadrado central da rede de nervação e guarda a imagem binária desse quadrado extraído numa nova pasta dedicada a cada espécie. A função passa por vários métodos de processamento de imagem. Primeiramente, aplicamos *Gaussian blur* e *adaptive thresholding* para criar uma imagem binária com as nervuras realçadas. De seguida, é necessário localizar a folha na imagem para saber em que zona retirar o quadrado. Idealmente pretende-se retirar de uma zona central para evitar incluir a borda da folha e assim prevenir o fornecimento de informação relativa ao contorno da folha. Para esse efeito, reforçamos a continuidade do contorno da folha com *Canny edge detection*, dilatação e erosão, e de seguida detetamos o maior contorno identificável (a folha completa) e contemos a folha dentro de um retângulo de área mínima. Finalmente, a partir deste retângulo, podemos localizar aproximadamente o centro da folha e criar um quadrado de 300x300 pixéis com o mesmo centro e alinhamento.

Após este método, obtemos 48 imagens binárias de zonas centrais de nervação, 12 para cada espécie. Na medida em que uma base de dados contendo 48 imagens é demasiado reduzida para treinar um modelo de aprendizagem profunda, é necessário aplicar técnicas de aumento de dados. Cada imagem vai então passar várias vezes pelo método de aumento de dados, sofrendo perturbações e, consequentemente, criando uma imagem alterada e tecnicamente diferente. As diferentes perturbações introduzidas incluem: inversão vertical, *zoom in* ou *out* até 10% e rotações até 5 graus. Os valores para as perturbações são relativamente comedidos para evitar deixar uma grande percentagem da imagem com falta de informação. Essa falta de informação é preenchida automaticamente pelo método, mas não é útil de todo, mas apenas uma repetição do pixel "real" mais próximo. Geramos através deste método um total de 5452 imagens adicionais, para um total de 5500 imagens de treino e validação, com 1375 para cada espécie, e 200 imagens de teste, com 50 para cada espécie (49 das quais também resultado da técnica de aumento de dados). A quantidade de imagens geradas por este método pode ser alterada, mas é recomendado manter o equilíbrio entre classes.

Com a base de dados aumentada, o *script* passa para a criação do modelo de aprendizagem profunda. Um modelo de aprendizagem automática é considerado um modelo de aprendizagem profunda quando tem várias camadas intermédias (também chamadas camadas "escondidas"). O modelo sofreu várias alterações ao longo do projeto, mas terminou com a seguinte estrutura: 5 camadas convolucionais alternando com 5 camadas de *max pooling*, uma camada de achatamento e duas camadas densas (totalmente conectadas). As camadas convolucionais criam mapas de ativação, que são representações mais abstratas da informação que lhes é fornecida. A presença de padrões semelhantes causa ativações que o modelo tenta aprender e associar a características. As camadas de *max pooling* reduzem as dimensões dos dados e a sensibilidade dos mapas criados pelas camadas convolucionais à localização exata das características. *Max pooling* combina o *ouput* de grupos de neurónios em um só neurónio, sintetizando a presença dessas características. A camada de achatamento transforma o *output* da camada anterior em uma matriz de uma dimensão, que é necessária para ser o *input* da camada totalmente conectada que se segue. Camadas totalmente conectadas significa que cada neurónio de *input* está ligado

a cada neurónio de *output*. A última camada tem apenas 4 neurónios de *output*, um para cada classe/espécie que temos na nossa base de dados.

O modelo de aprendizagem profunda é um modelo supervisionado, o que significa que cada imagem vai passar pelas camadas do modelo e no fim desse processo o modelo vai tentar prever a que classe corresponde à imagem que observou. Dependendo do sucesso da previsão, o modelo vai tentar alterar os pesos das ligações de neurónios no sentindo de os reforçar ou corrigir através do processo de retropropagação.

Cada vez que executamos o processo de aprendizagem total do modelo, este pode passar por até 50 épocas, que são ciclos de treino e validação. O grupo de treino corresponde a 80% do conjunto de 5500 imagens de treino, portanto 4400 imagens (1100 imagens por classe/espécie) escolhidas aleatoriamente e estratificadas. Os restantes 20% fazem parte do grupo de validação (275 por classe/espécie). A cada época, o modelo é exposto a todas as imagens do grupo de treino, uma por uma, alterando-se através de retropropagação. No fim da fase de treino, o modelo passa por todas as imagens nunca vistas do grupo de validação e é avaliado. Não existe retropropagação no grupo de validação, logo não existe aprendizagem, o que significa que mesmo usando sempre as mesmas imagens de validação, para o modelo serão sempre imagens nunca vistas. O grupo de validação é então exclusivamente para avaliação do desempenho do modelo ao fim de cada época de treino. Uma funcionalidade implementada que usa os resultados da validação é a de paragem prematura que, ao fim de 6 épocas sem melhoria nos resultados de validação, termina o processo de aprendizagem. Isto pode acontecer porque o modelo já encontrou a solução ótima há 6 épocas, ou porque o modelo aparenta estar preso numa solução subótima.

Ao fim de 50 épocas, ou menos no caso de paragem prematura, o *script* produz dois gráficos: um que mostra a evolução da exatidão do modelo nos grupos de treino e validação ao longo das épocas, e outro que mostra a evolução da perda (total da penalização por previsões erradas) do modelo nos grupos de treino e de validação ao longo das épocas.

Finalmente, com a versão final do modelo, testamos com o grupo de teste, um grupo de 200 imagens (50 por classe/espécie) realmente nunca vistas e aumentadas a partir de uma imagem original sem versões perturbadas nos grupos de teste ou de validação. O *script* produz um relatório de avaliação do modelo com base nos resultados deste teste, calculando a precisão e a revocação (também conhecida como sensibilidade). É produzido também um gráfico de matriz de confusão sob forma de *heatmap*.

O *script* foi então lançado múltiplas vezes, procurando, por tentativa e erro, obter valores superiores dos parâmetros e camadas do modelo para uma aprendizagem de maior sucesso. Depois, com a versão final, o *script* foi lançado 25 vezes. Desses 25 lançamentos, apenas 20 foram considerados nos resultados, tendo em conta que os outros 5 lançamentos ficaram detidos muito cedo em soluções significativamente subótimas e foram considerados discrepantes. Conseguimos obter uma exatidão geral de perto de 100% nos grupos de treino e validação, e uma exatidão geral de aproximadamente 79.3% no grupo de teste que, como referido, contém imagens completamente novas e nunca vistas pelo

modelo. Este é um resultado bastante positivo. No entanto, o modelo sofre de uma certa instabilidade e falta de robustez. Nos gráficos de exatidão produzidos pelo *script*, esta métrica apresenta saltos impetuosos seguidos por períodos estáticos. Possivelmente, devido à complexidade deste problema de classificação, estes saltos correspondem aos momentos em que o modelo aprende a distinguir uma nova classe/espécie, seguido de momentos de bloqueio temporários, ou no pior dos casos extensos ou finais. A instabilidade e falta de robustez poderá ser motivada pela falta de uma quantidade razoável de imagens originais, já que a base de dados de 48 imagens originais (4 delas isoladas para teste) é, presumivelmente, demasiado diminuta como ponto de partida para conseguir um modelo de aprendizagem profunda robusto, tendo em conta a complexidade deste problema de classificação.

Table of Contents

List of figures

Background

According to a 2016 study, there are 383,671 accepted species of vascular plants [1]. Identifying an unknown plant, i.e., assign it a species or infraspecific name, becomes a challenge due to the sheer number of species and their diversity. There are many features one can use to attempt to visually identify a species of plant, and most rely on reproductive structures and require the experience of a taxonomist. However, recently, complementary methodologies, such as machine learning models, are being developed using flowers and leaves or even location and time of observation [2][3].

In the present work we explore the potentialities of using convolutional deep learning neural networks and we will be focusing on the leaves, specifically on the venation patterns, that seem to be a reliable feature to support plant identification.

The main issue identified is that most of the machine learning models we have encountered during our research (those that are trained based on the plant's leaf) focus on the raw leaf image or on the leaf shape (or contour) [4][5][6].

The issue with using the leaf contour is that it can vary extremely depending on some factors. Leaves from a young plant can differ in shape compared to leaves of an older plant of the same species. The same can be said when varying the position of the leaf (at the base of the plant or at the top) or even between sun and shade leaves. Examples of this issue will be shown in the next section. Furthermore, there can be some damage on the edges of the leaf, caused by insects or diseases, which will alter the contour of the leaf.

We decided to focus on the venation network of leaves and explore the possibility that the features of the venation network are reliable to support species identification.

Objectives

The main objective for this specific project is to create a fully automated pipeline written in Python, going through the different stages of the process. First, it would perform the pre-processing of the image database and data augmentation to increase our dataset, followed by the initialization and training of the neural network, and finally the testing of the trained network. A secondary objective is that the model performs well in terms of accuracy. In other words, that the model can consistently correctly identify the plant species during the testing phase (i.e. on previously unseen images).

Furthermore, if this research, exclusively focused on leaf venation, sees positive indications, it would be interesting to explore the possibility of using the leaf's network of veins as a general "fingerprint" of sorts for every plant species.

Related Work

Concerning previous research on plant species identification models, we can find different approaches focused on different features of the plant.

There are several studies that used images of plant leaves and trained machine learning models to identify the species of plant. A 2013 study by Hati and Sajeeval used Artificial Neural Networks (ANNs) to classify unprocessed, raw images of complete plant leaves scanned on a white background, achieving an accuracy of 92% [7]. Another example of such study, by Wick and Puppe (2017), used different types of Convolutional Neural Networks (CNNs), deep learning machine vision models for classification of the images. The images were also not processed, so the network was learning from raw images of leaves, however the results were extremely positive [8]. The authors achieved accuracy levels of above 99%, although this could be due to the fact that the leaves they used (the Flavia dataset) vary quite markedly in shape.

In 2015, Lee et al. used manually cropped inner sections of raw images of leaves to train CNN models [9]. Data augmentation was performed to increase the dataset. The results were extremely positive, with 97-99% accuracy, although that might have to do with not having unique images segregated for the test set (there were augmentations of the same original image in both the training and test sets), and therefore it is difficult to rule out the possibility of some overfitting (memorization of images seen in training).

Looking for studies more closely related to our project, we find a few that focus on a combination of leaf shape and veins by using different methods to extract veins and margins. One study by Gu et al. (2005) used k-Nearest Neighbours (kNN) method and Radial Basis Probabilistic Neural Networks (RBPNNs) [10]. It achieved an average correct recognition rate of approximately 93.17% for 1-Nearest Neighbour models, 85.47% for 5-Nearest Neighbours models and 91.18% for Radial Basis Probabilistic Neural Network models.

A 2016 study by Wilf et al. also focused on a combination of leaf venation and leaf shape to classify leaves not by species, but by family, achieving results of around 72% for the 19 classes/families with the highest representation in the dataset [11] (classes/families with the highest number of examples).

Larese et al. (2014) [12] focused exclusively on the venation network by excluding the shape of the leaf. However, this study also used other machine learning models that are not considered deep learning or convolutional neural networks (Support Vector Machine, Random Forest, and Penalized Discriminant Analysis). The results obtained were varied, with several models reaching an average accuracy in the range of 55-60%, while others falling to a range between 40-45% in average accuracy.

The reason we selected a convolutional neural network model for this project is because CNNs are high performance networks for image identification and machine vision [13], and are expected to outperform classic machine learning models.

Data

The Leaf

The usual methods for plant identification by taxonomists are based on the characteristics of the reproductive organs (i.e.: fruits and flowers). However, other structures can be successfully used for that purpose. That is the case for leaves, which are vegetative structures that perform fundamental roles in physiological processes of the plant, whose characteristics are often complementary used in the process of plant species identification. Due to this interest, the viability of using the characteristics of leaves in machine learning models for classification has been the subject of recent research.

Our research focuses solely on the leaf venation network itself. Veins are important structures which perform crucial tasks for vascular plants. The vein xylem transports water from the roots to the leaves through the petiole, and the vein phloem transports sugars from the leaves (and other photosynthetic organs) to the rest of the plant [14].

The venation network itself is divided into different structures (Figures 1, 2):

- Petiole ➔ A narrow stalk that connects the leaf to the stem and supports it.
- Midrib ➔ The primary vein of the leaf, running along the midline of a leaf through the entire leaf blade.
- Secondary veins ➔ Veins branching from the primary vein. Intersecondary veins are more slender veins that run parallel to the adjacent secondary veins and that tend to ramify towards the middle of the blade.
- Tertiary and Quaternary veins ➔ lower-order veins forming an extensive and intricate structure.
- Areole ➔ Small area of leaf tissue surrounded by veins.

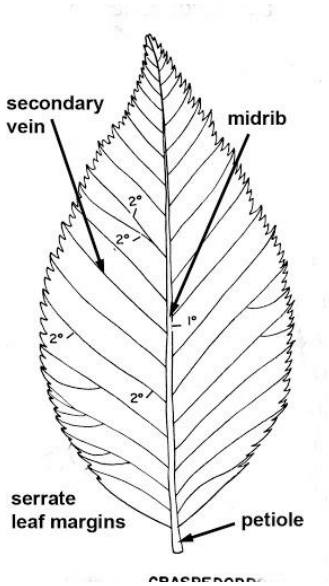

Figure 1: Major veins of a leaf [15]

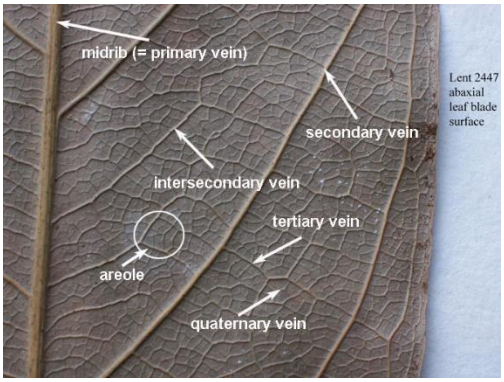

Figure 2: Leaf venation of a species of Eschweilera *(major and minor veins)[16]*

Dataset

Originally, our dataset consisted of leaves of four species of *Quercus* (*Q. pyrenaica*, *Q. faginea*, *Q. robur*, *Q. suber*) gathered in the field and subsequently dried according to the procedures used in the herborization of plant specimens. For each species, we had 15 different leaves that we both photographed and scanned. Unfortunately, over time we realized that we could not reliably and satisfactorily extract the venation network from the images. The main issues were that the lighting was not uniform and could not easily be corrected. Some of the leaves had an abundance of trichomes (hairs), which occulted its features or spots and dirt which impeded the use of thresholding to isolate the veins (Figure 3).

Figure 3: Example leaf of a Quercus faginea *leaf from the initial dataset. Only some of the primary vein and secondary veins are visible, which is not enough for the scope of our project. Additionally, notice the amount of noise and the lack of information in some areas.*

To overcome this issue, we used the Cleared Leaf Image Database [17] that contains cleared leaves images (by chemical processes and/or others, including subsequent X-ray imaging) [18]. These images are extremely useful, as they greatly enhance the visibility of both major and minor leaf veins. Currently, many repositories of cleared leaves are being created due to the rise of biological interest of botanists in the veins of vascular plants. ClearedLeavesDB, the origin of our samples, groups many of these repositories under one platform.

As mentioned, the shape of different leaves of the same species can vary significantly based on age and relative position in the plant, and examples of those differences can also be found in our dataset (Figures 4-6).

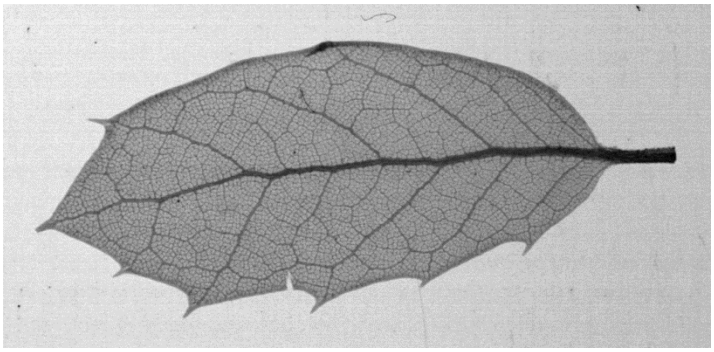

Figure 4: First example of a cleared leaf of the species Quercus ilex*, from the current dataset.*

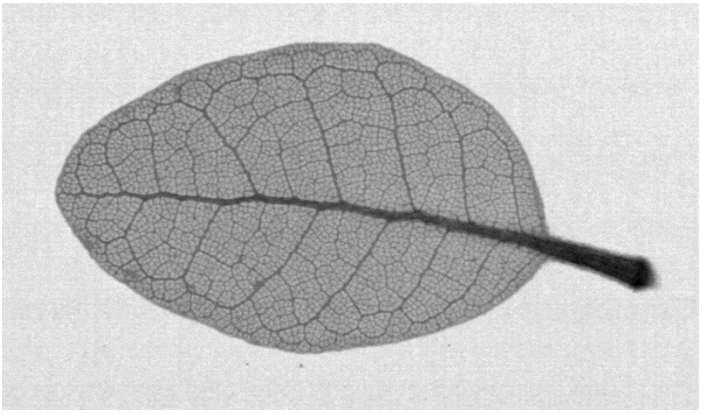

Figure 5: Second example of a cleared leaf of the species Quercus ilex*, from the current dataset.*

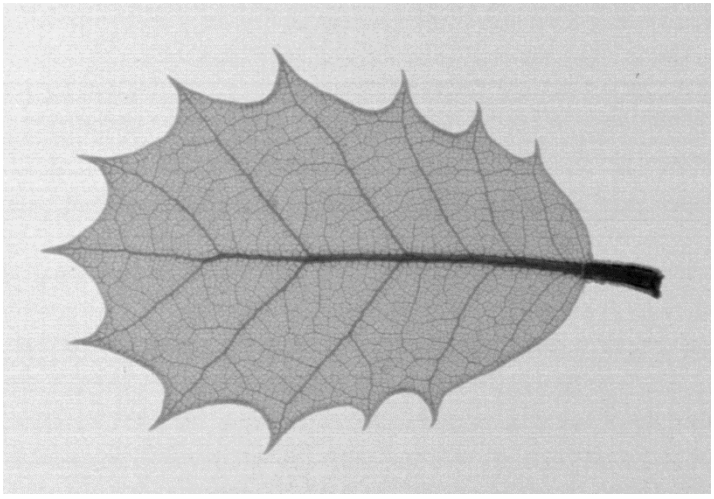

Figure 6: Third example of a cleared leaf of the species Quercus ilex*, from the current dataset. These images exemplify the significant differences in shape between leaves of the same species.*

From the database we select four species:

Betula pendula Roth (family *Betulaceae*) – silver birch, a tree occurring in Europe, Asia, N America [19].

Bridelia scleroneura Müll.Arg. (family *Phyllanthaceae*) – shrub or tree, with a native range in Tropical Africa and Yemen [19].

Cassia sieberiana DC. (family *Fabaceae*) – drumstick tree, from W. Tropical Africa to NW. Uganda [19].

Quercus ilex L. (family *Fagaceae*) – holm oak, a tree occurring from S. Central & S. Europe to Turkey [19].

We proceeded with the following 48-image dataset:

- 12 images from the species *Betula pendula*
- 12 images from the species *Bridelia scleroneura*
- 12 images from the species *Cassia sieberiana*
- 12 images from the species *Quercus ilex*

Design and Methods

As previously mentioned, the main goal is to create a fully automatic pipeline where all the user has to do is move the raw cleared images of leaves to the appropriate folders and run the Python script. Python was selected as the programming language for this project because of the availability of libraries, such as Tensorflow/Keras and Scikit-learn, which support the creation of machine learning models and related projects, and OpenCV, which provides a plethora of useful image processing functions.

There are two main technical and logistic challenges for this Python script:

1. Pre-processing of the images – feature extraction (final code on Appendix A)
2. Implementation and fine tuning of the neural network (final code on Appendix B)

Image Pre-processing

Essentially, the challenge is not only to extract the venation network from the images of leaves, but to also eliminate the influence of the contour of the leaf on the learning phase of the model.

As we have seen in the Data section, extracting the leaf's veins is extremely challenging due to several factors. Even with our cleared images of leaves, it can be tricky to obtain the complete network clearly and consistently in an automated way, for all the images. Therefore, we used two different tools to assist us in programming that task:

1. LeafGUI:

   LeafGUI (Fig. 4) is a manual tool for the processing of leaf images. Users can use several functions to modify images, e.g.: thresholding, erosion, binarization, skeletonization, etc. in order to try extracting the contour and the veins of the leaf [20].

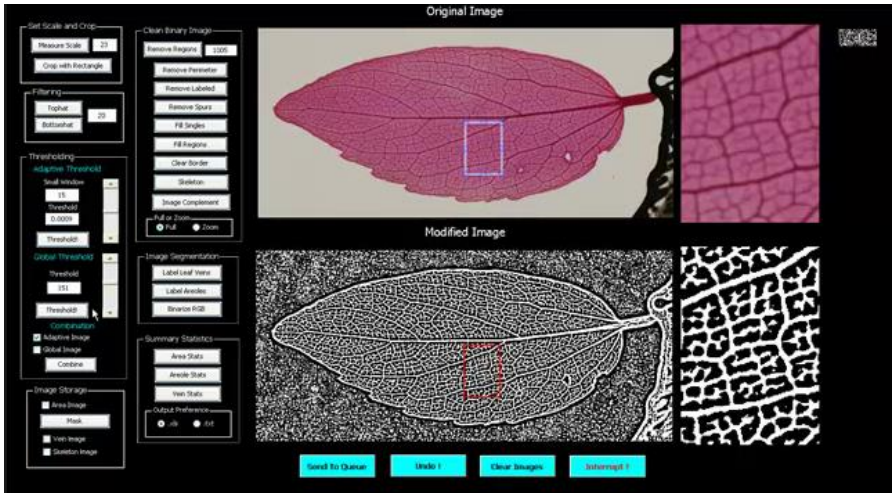

Figure 4: Overview of the LeafGUI platform. These images are from the LeafGUI instructional videos, not from our datasets.

2. NEFI

NEFI is a more versatile tool, in that it is geared towards the extraction of networks in general, not necessarily related to plants (for example, crack patterns on a wall). Interestingly, it does have a premade pipeline specifically geared towards leaf venation networks, but the user can also easily create new custom pipelines [2121]. All pipelines are automatic, which is one advantage NEFI has over LeafGUI, another advantage being its much faster performance. Using the appropriate pipelines, the venation extraction obtained from cleared leaves is, generally speaking, very detailed (Fig. 5).

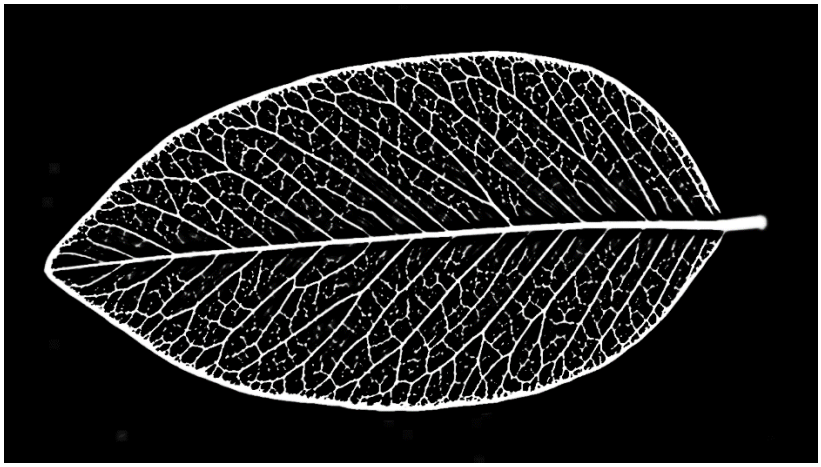

Figure 5: Result of running a NEFI pipeline on a cleared leaf image from our dataset (species: Cassia sieberiana*).*

These tools are extremely useful, however they have not been incorporated into the final pipeline, since that would make the pipeline only semi-automatic, parts of it being manual. Instead, their purpose is to help determine not only the optimal set of pre-processing steps that we can take to extract the venation network from the images, but also the optimal parameters for each of those steps.

Experimenting with these tools gave us information which helped us create a Python script that uses the OpenCV library to replicate what was researched using NEFI and LeafGUI. This library contains image processing algorithms commonly used for machine learning and computer vision tasks [22].

As we mentioned, after the extraction of the venation network, we still need to remove any influence the contour can cause on the learning phase of the model. This can be achieved by cropping out a central square of veins from each image. This square should be of the same size for every image. The most significant challenge in this step of the process is achieving this automatically. Essentially, the idea is that the Python script will identify the leaf object, calculate its aspect ratio, box the object, and calculate its centre. These steps will help determine where to crop the square from, without catching the borders of the leaf, where the vein endings might give a sense of the leaf's contour.

The image processing phase proceeds the following way (Figures 6 to 14):

1. An image is loaded

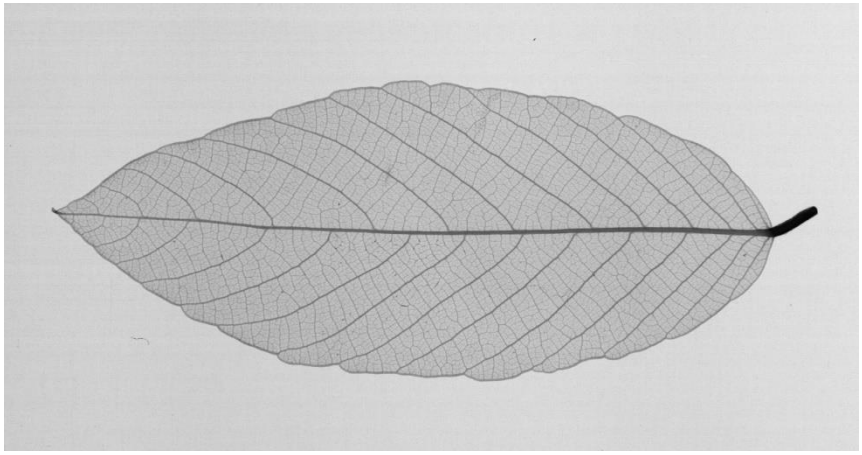

Figure 6: Original cleared leaf from the species Bridelia scleroneura.

2. A Gaussian blur is performed ➔ slightly blurring the image reduces the amount of noise and improves the edge detection step of the process. The kernel size parameter was set at (7,7).

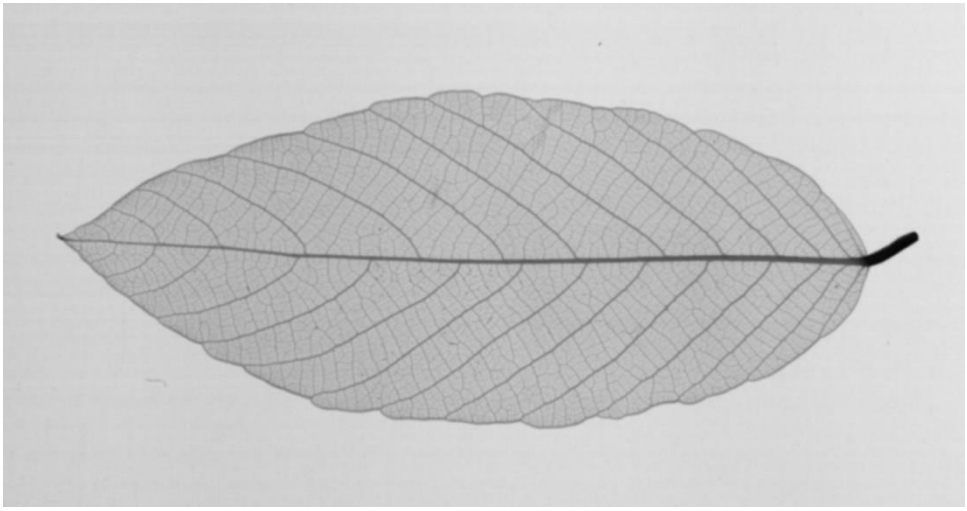

Figure 7: Bridelia *image after applying Gaussian blur.*

3. The image goes through adaptive thresholding ➔ thresholding converts a grayscale image to a binary image, based on a T value. Anything above the T value becomes a black pixel, anything below becomes a white pixel. In this case, we use adaptive thresholding, which means that the value for T, now determined by a hardcoded OpenCV formula, is different depending on the region. Adaptive thresholding performed the task of creating a quality binary image much more successfully than global thresholding. Furthermore, global thresholding would have been more cumbersome to optimize, as the T parameter has to be defined by the developer and would still be less effective. This binary image is saved to be used on step #8.

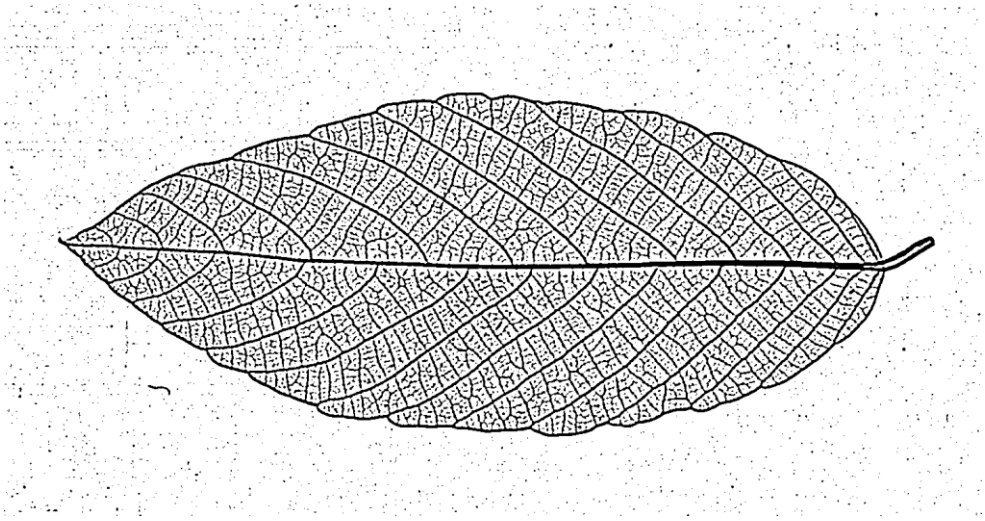

Figure 8: Bridelia *image after applying adaptive thresholding.*

These three steps are enough to enhance the venation network itself on a cleared leaf image. However, in order to know where to extract the 300x300 square from the image, and avoid clipping any outside edge of the leaf itself, we need to go through a few more steps:

4. We apply Canny edge detection on a copy of the image ➔ Canny edge is an algorithm used to extract structural information on the image. This algorithm goes through a few image processing steps with the purpose of extracting different edges from the image. It will enhance the contour of the leaf, which we are not interested in terms of training and classification, but which will be useful to detect the approximate placement and area of the leaf in the image.

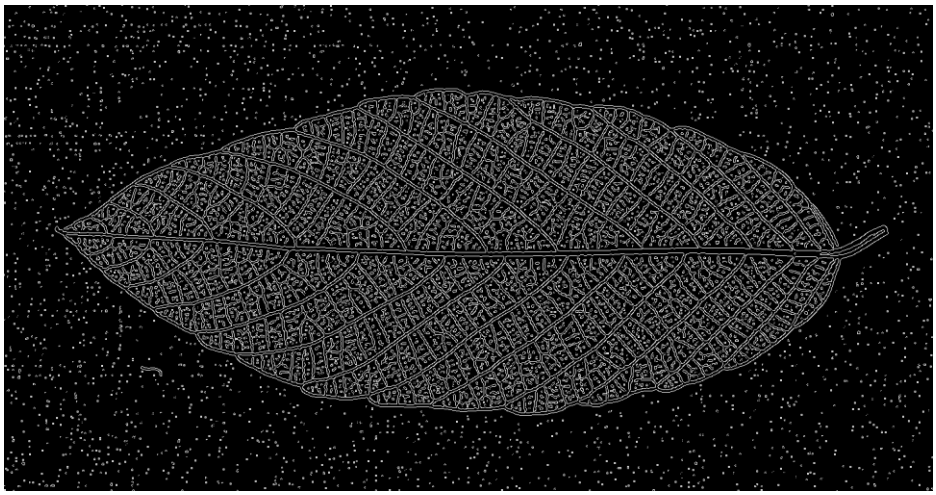

Figure 9: Bridelia *image after applying Canny edge detection.*

5. We proceed with a dilation and erosion step on the copy ➔ the dilation step is to ensure that the edge of the leaf contour that might have small gaps from the previous steps closes back into a full edge. However, it is necessary to perform an erosion next. This is to help remove pre-existent noise and spurs, or ones left out by the edge detection or dilation process. These two steps increase the clarity of the leaf contour enhanced by the Canny edge detection algorithm, which is a crucial step to ensure the following step does not fail. Those two operations together, in that order (dilation followed by erosion), are also referred to as a morphological "close" operation.

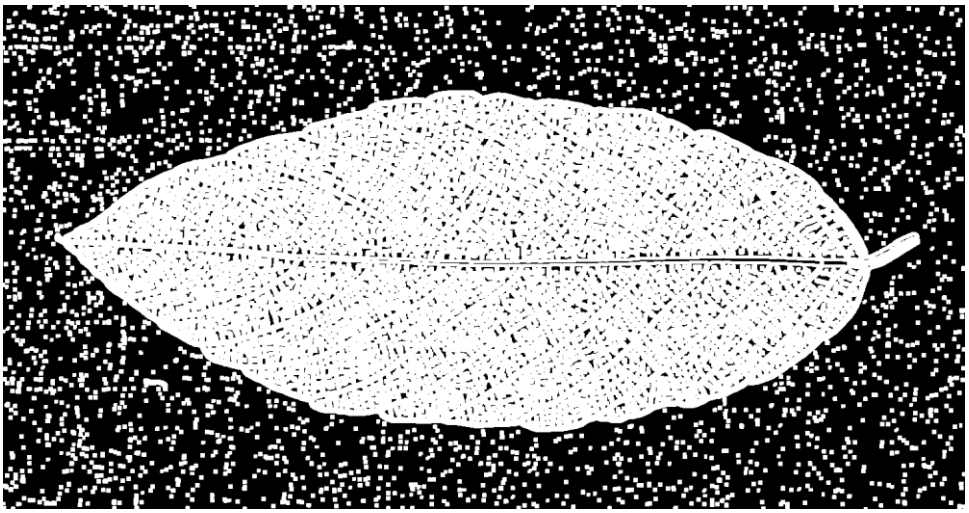

Figure 10: Bridelia *image after dilation.*

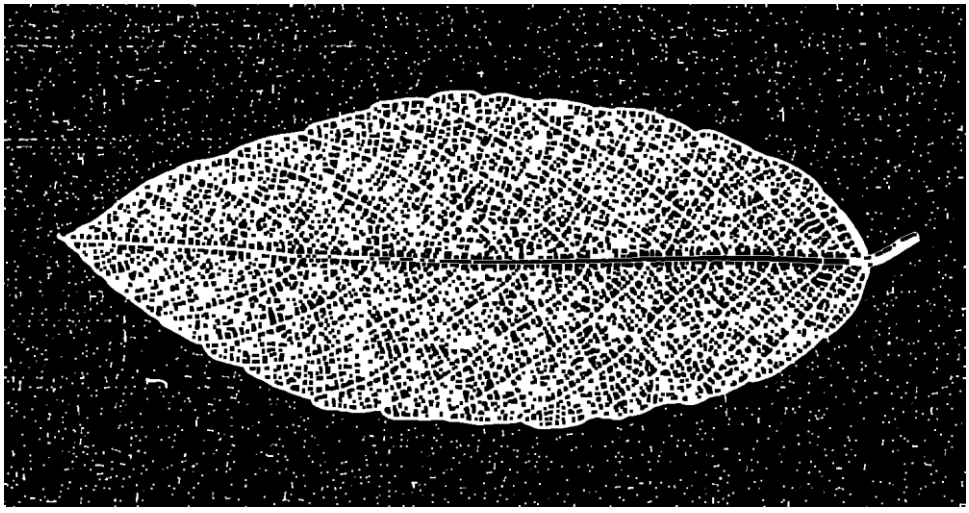

Figure 11: Bridelia *image after erosion.*

6. We apply the OpenCV *findCountours* method on the copy to detect contours and find the one with the largest area, which should be the contour of the leaf itself ➔ A contour is defined as a curve of continuous points that have the same colour or intensity. Sorting the returned contours by area and selecting the largest gives us the leaf itself because it is the largest object on the picture. This also ensures that we do not accidentally select any leftover dirt or noise that might be identified as a minor contour in the returned set.

7. A minimum area box is fitted to the contour and we determine the centre of the rectangle, which is essentially the centre of the leaf ➔ This function calculates the rectangle with minimum possible area that encompasses the contour of the leaf. From there we can simply calculate the approximated centre of the leaf by taking the midpoints of the edges of the rectangle and intersecting the lines, perpendicular to the rectangle's edges, that cross the midpoints and finding their intersection. This box should be approximately aligned with the orientation of the leaf itself.

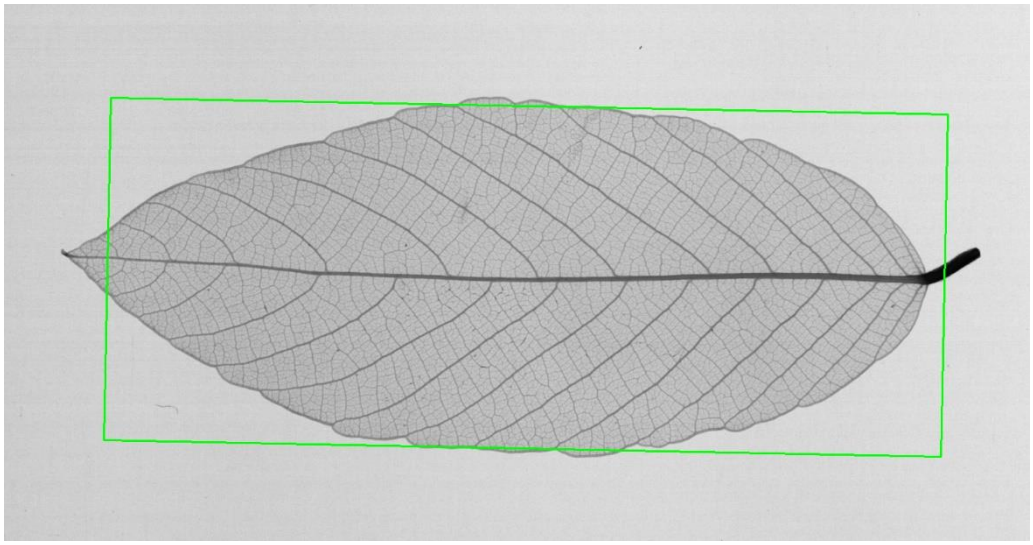

Figure 12: Bridelia *leaf located on the image, bound by the minimum area rectangle.*

8. Creating a 300x300 pixel square on the image we saved from step #3 around the calculated centre of the leaf, with the same alignment as the previous minimum area box ➔ Knowing the centre of the leaf and the rotation of the original box (obtained on the copy in step #7), we can now place four corner points 150 pixels away from the centre, rotating them around the origin, to match the original box rotation. Thus, we obtain a 300x300 pixel square around the centre of the leaf, which will be aligned according to the leaf contour in the image, in order to minimize the chances of the extracted square clipping a margin of the leaf. The extracted square is then reoriented so we can save it as an image in the filesystem.

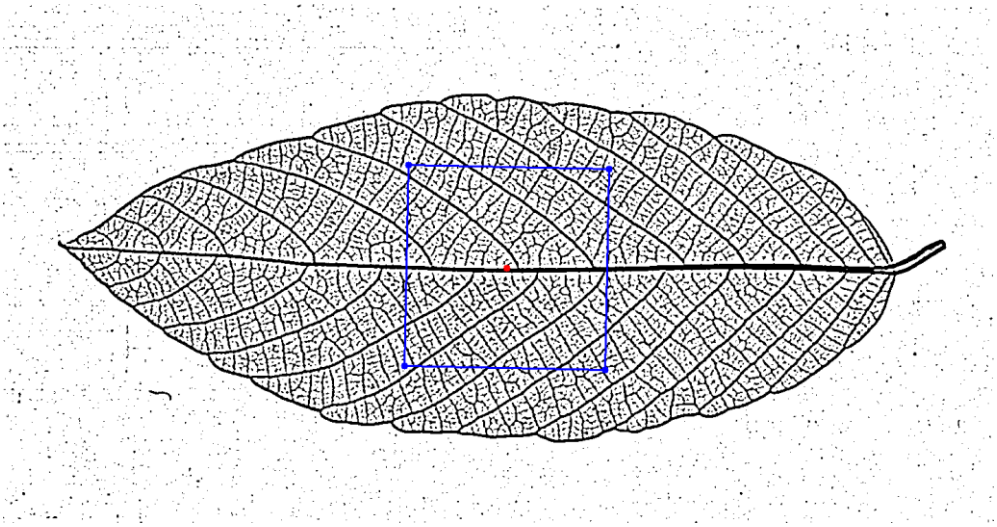

Figure 13: Square of veins to be extracted mapped from the centre of the leaf.

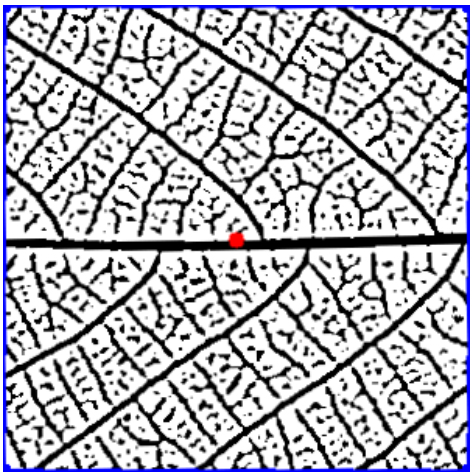

Figure 14: Extracted square of veins from the Bridelia *leaf.*

Once all the cropped squares have been extracted, the pipeline will move on to the second main phase: training the network.

Implementation of the Neural Network

Firstly, a short description of what a Convolutional Neural Network is (CNN):

CNNs are hierarchical neural networks, biologically inspired by the organization of the animal visual cortex, composed of several layers: an input layer (which are the images), alternating convolutional and subsampling/pooling layers, a flatten layer, and finally a fully-connected/dense output layer [23].

The convolutional layers act like a moving window kernel to filter the input and the subsampling layers (also called pooling layers) reduce the spatial sizes of the features, decreasing the necessary computational power required to process the large quantities of data that images contain, reducing noise and extracting dominant features without those features being specifically singled out manually. Most of the network consists of alternating convolutional and pooling layers. The flatten layer will flatten the output of the final subsampling layer into a vertical vector and feed it to the fully-connected layer. Fully-connected means that every single input node (every output node from the flatten layer) is connected to every single final output node of the network. The values of the output nodes of the final fully-connected layer will determine which class (species) that image is predicted to be. The learning process used to obtain the connection weights and biases in these layers correct these weights and biases through backpropagation.

Using CNNs is extremely advantageous since it processes the actual image, meaning there is no need to go through several steps of feature extraction and transformation to obtain dozens of numerical features that attempt to describe the image [24].

In order to implement the CNN, we used TensorFlow. TensorFlow is an open-source platform for machine learning written in Python [25]. Advantages of TensorFlow include high performance, scalability, Keras integration and good community support.

The first step in the implementation of the network is to fetch the locations of the original unaltered images from the file system. These locations are fed one by one to the image processing function that highlights the venation network of the leaf and extracts a central square for the model. The image squares returned by the function are saved on separate folders and we build a dictionary for each step of the learning process, containing the file locations of the images.

Now that we have our images for the learning process, we need to go through a data augmentation step. This step is necessary because of the shortage of real examples in our dataset. Typically, for a deep learning model, we would need thousands of unique examples for a learning process to produce viable results. Since our data does not contain a significant number of examples by itself, we need to find a way to increase our dataset.

The data augmentation step begins by loading the images one by one and passing each one through an image generator. This image generator creates a modified image, with one or more disturbances compared to the original, and stores it in the same folder. These disturbances include:

- Vertical flip ➔ some images will be mirrored on the X axis to mimic what would be the other side of the leaf
- Zoom in or out ➔ set at up to 10%, either closer or further away from the image. Zooming out of the extracted square causes an issue of missing data around the boundaries, which is filled in by the generator based on the nearest available pixel. The maximum value was set at 10% to

avoid extreme changes to the images. Zooming in too close could mean large areas of the images without information (white areas where the areoles are located). Zooming out too far would mean a large outer layer of generated "filler" surrounding the true information to learn.

- Slight rotations ➔ set at up to 5 degrees of rotation. Any gap in information caused by the rotation is filled in by the generator in the same way as previously stated. We again avoid large rotations because of the amount of "filler" created by the generator necessary to complete the image, which would make it very unlike a plausible section of a leaf.

This step will be repeated until it reaches the desired number of training and testing examples. These values can be modified at will (examples of modified images are presented in Fig.15). The balance of classes will be kept, which means that if a particular class has more original examples than the others, that imbalance would be maintained in the same proportion. Maintaining class balance is recommended for better performance.

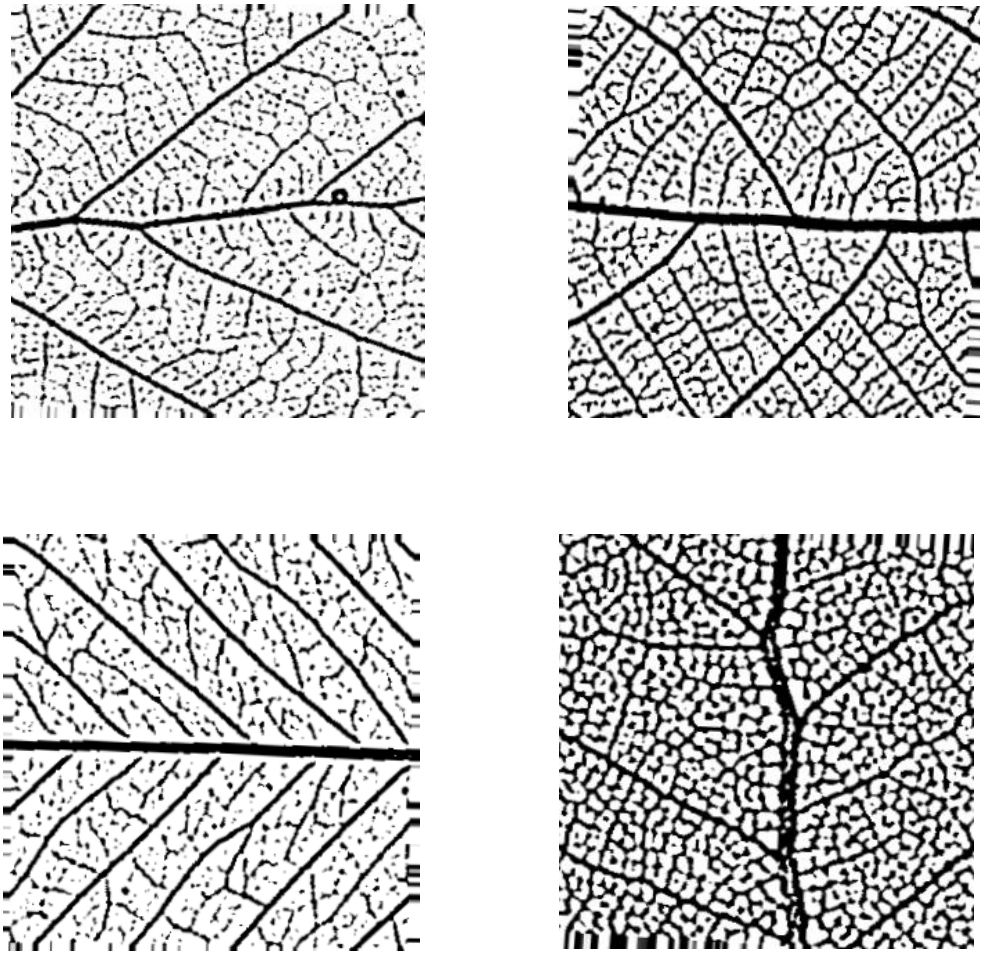

Figure 15: One randomly selected augmented example per class (top left: Betula*; top right:* Bridelia*; bottom left:* Cassia*; bottom right:* Quercus*).*

At this point, we have our complete training and testing dataset prepared for the learning phase, divided by class. In our main dataset we have a total of 5500 images, divided equally amongst our 4 classes (1375 per class), corresponding to the four analysed species. Out of these 5500 images, only 44 are

original images (11 from each class), the other 5456 are created by the generator, they are a result of the data augmentation step of our script. We use 4400 of them (1100 per class), randomly chosen and stratified, for the training set and the other 1100 (275 per class) for our validation set. In our test set, we have 200 images divided equally amongst our 4 classes (50 per class), of which only 4 are original images, one from each class.

The structure of the implemented Convolutional Neural Network is as follows:

1. Convolutional layer (Conv2D) ➔ 16 filters and kernel size of 5. These 5x5 filters are convolved with the input by a dot product operation and produce an activation map, which is a more abstract representation of the input. The network learns these filters that produce an activation signal when in the presence of a specific feature.
2. Max Pooling layer (MaxPooling2D) ➔ Pool size of 2. Pooling reduces the dimensions of the data and the sensitivity of the feature maps to the specific location of the features. It does so by combining the outputs of groups of neurons into a single neuron, therefore summarizing the presence of those features. We are using max pooling instead of average pooling, which means that in our 2x2 pools, we take the maximum value of the cluster to its corresponding neuron on the next layer, instead of the average value of the cluster. Max pooling is generally considered to be more effective at highlighting important features (like edges) [26].
3. Convolutional layer (Conv2D) ➔ 16 filters and kernel size of 5
4. Max Pooling layer (MaxPooling2D) ➔ Pool size of 2
5. Convolutional layer (Conv2D) ➔ 32 filters and kernel size of 3
6. Max Pooling layer (MaxPooling2D) ➔ Pool size of 2
7. Convolutional layer (Conv2D) ➔ 32 filters and kernel size of 3
8. Max Pooling layer (MaxPooling2D) ➔ Pool size of 2
9. Convolutional layer (Conv2D) ➔ 32 filters and kernel size of 3
10. Max Pooling layer (MaxPooling2D) ➔ Pool size of 2
11. Convolutional layer (Conv2D) ➔ 32 filters and kernel size of 3
12. Max Pooling layer (MaxPooling2D) ➔ Pool size of 2
13. Flattening layer ➔ This layer is necessary in order to transform the matrix output of the previous layer into a one-dimensional array. This array will then become the input for the fully-connected layer that follows it.
14. Dense layer (fully-connected) ➔ 64 output nodes. These next two layers are fully-connected, which means that every input node is connected to every output node.
15. Dense layer (fully-connected) ➔ 4 final output nodes, corresponding to the 4 classes in our dataset. Unlike the other Conv2D and Dense layers, which use a *relu* activation function, this final layer uses *softmax*. We use *softmax* for the final layer because it returns a probability distribution, with one value for each class. It describes how confident the model is of the predicted class, and the perceived likelihood of the image belonging to a different class than was predicted.

The number of layers, nodes, and parameters of the convolutional and pooling layers were selected partly through trial and error, and partly based on the available bibliography, to provide a good compromise between solid performance and a more acceptable number of learnable parameters of the model, which are the connections weights and biases learned by the network. The final total number of parameters for this network is 47732, as output by the script at the end of each run.

Through trial and error, the final learning rate set for the model was 0.000013. The learning rate is a parameter that dictates how quickly the model is trying to adapt to the problem. A high learning rate will change the weights dramatically after each iteration, but it is more likely to overshoot minima. A low learning rate will change the weights only slightly after each iteration, but it could potentially get stuck in a local, suboptimal minimum.

Optimizers are algorithms designed to tweak specific model parameters, to improve performance and minimize loss. The optimizer chosen for this model was *Nadam* [27], with its default parameters. *Nadam* is similar to another optimizer called *Adam* [28], but with integrated Nesterov's accelerated gradient [29]. *Adam* has a classical momentum [30] component, but there is strong indication that Nesterov's momentum performs better than classical momentum [31]. In the machine learning field, momentum is relevant to avoid getting stuck in local minima, because the loss function can be very complex and have multiple minima. Getting stuck in a local minimum means settling for a non-optimal solution, because it is not the global minimum sought (i.e.: the best possible solution). Having a momentum component helps avoid this problem by taking large steps to try to jump local minima, while at the same time trying not to overshoot the global minimum. We selected *Nadam* as our optimizer because it performs well in machine vision problems and may outperform most other optimizers [32].

The model goes through up to 50 epochs. In each epoch it sees each generated example once, going through all images in the dataset. The first phase consists of attempting to classify the images in the training section one by one, adjusting the values of the weights in the node connections based on the accuracy of the guess. That is the process through which the model learns, called backpropagation. Then, at the end of the training phase of each epoch, the model goes through the "previously unseen" images in the validation section and attempts to classify them. The model does not learn during this step, the weights of its nodes are not adjusted in the event of a wrong classification. Therefore, even though the validation images are always the same (hence the quotation marks on "previously unseen"), the model cannot memorize the specific images and improve on the validation result in that manner throughout the epochs, unless there is actual generalized learning. Having this validation step is crucial in finding if a model overfitted its training data or if it is steadily learning to solve the problem at hand. Overfitting means that the model is mostly memorizing the training examples instead, and when presented with previously unseen images, it has not learned enough general features about each class to be able to classify them as accurately as when it is presented with an image that has been memorized.

Throughout each epoch we see the accuracy and the loss of the model in the training phase as it goes through each example, output in real time. At the end of each epoch, we are provided with the accuracy and loss of the model tested on the validation data. Comparing this information will give us an idea if the model is overfitting the training data or not. If the training accuracy is significantly higher than the validation accuracy, that is an indication that the model could be overfitting its training data, essentially memorizing the training images instead of generalizing characteristics of the class. A model suffering from overfitting would score very highly on the memorized training images, but fail to understand what distinguishes each class, and therefore struggle to recognize identifiable class characteristics on unseen validation images, thus the pronounced lower validation performance.

Additionally, we used a callback method called Early Stopping. This method monitors the validation loss of the model at the end of each epoch and stop the learning process whenever it seems to have stabilized. Whether a model has stabilized is not always clear, so there are a couple of parameters that help a user determine when to stop the process: patience and delta. Patience is the number of epochs to wait for a significant improvement. After some trial-and-error, and due to the fact that we do not want to stop the process too early because of the problem's complexity, we set the patience parameter with a value of 6. This means that it will only stop the learning process after 6 epochs without significant improvement. Delta essentially defines "significant improvement". By default it is set to zero (meaning any improvement, no matter how small, resets the patience), but we found that the model would simply continue the learning process until the end of the 50 epochs on the basis of minuscule decreases in validation loss, defeating the purpose of Early Stopping. We set the threshold slightly higher, at 0.005. This allowed for a sizeable reduction of unnecessary training time.

After the learning process is complete, the script displays two graphs: the first one shows how the model accuracy and loss evolved over the epochs on the training examples. The second one shows how the model's accuracy and loss evolved on the validation examples.

Finally, there is a testing phase on truly unseen images. The results of this test are used to print a report (example of a report on Fig. 16) analysing the model's performance. The report provides several metrics, by class:

1. Precision: how often the model is correct when it predicts "class X"

$$Precision = \frac{true\ positives\ (TP)}{number\ of\ predicted\ positives\ (TP + False\ Positives)}$$

Equation 1: Precision

2. Recall: how often does the model correctly predict class X's images

$$Recall = \frac{true\ positives\ (TP)}{number\ of\ actual\ positives\ (TP + False\ Negatives)}$$

Equation 2: Recall

3. F1-score: the harmonic mean of the precision and the recall

$$F1 = \frac{2}{Precision^{-1} + Recall^{-1}}$$

Equation 3: F1-score

4. Support: how many testing images for class X

	precision	recall	f1-score	support
betula	0.91	0.20	0.33	50
bridelia	0.60	0.80	0.68	50
cassia	0.69	1.00	0.82	50
quercus	1.00	1.00	1.00	50
accuracy			0.75	200
macro avg	0.80	0.75	0.71	200
weighted avg	0.80	0.75	0.71	200

Figure 16: Example of metrics report.

Additionally, another graph called a confusion matrix in the form of a heatmap is displayed. Also known as an error matrix, it shows the results of the testing phase in a visually informative manner. There are four rows and four columns, one for each class, in the same order (Fig. 17). The columns represent the predicted class, and the rows represent the true class.

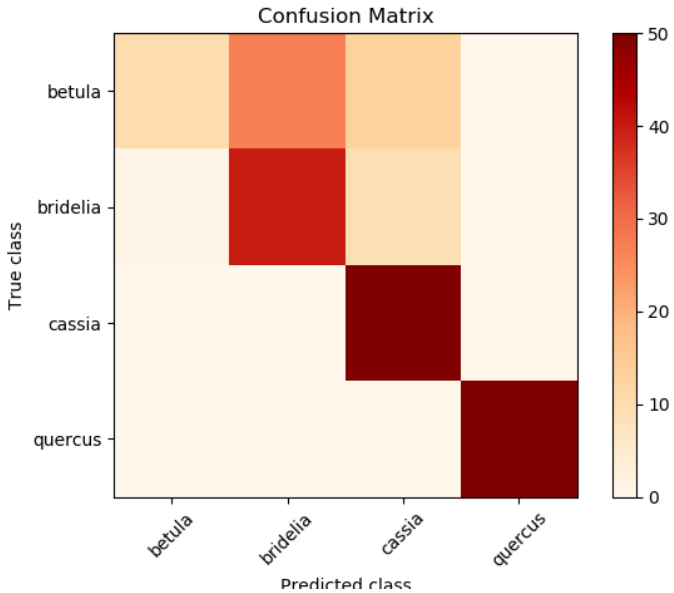

Figure 17: Example of confusion matrix.

In this example matrix, we the model was successful at recalling 3 of the 4 classes. However, when faced with unseen images of the *Betula* class, it often predicted them to be images of the *Bridelia* class. The rest of the predictions when presented with unseen images of the *Bridelia* class were either accurate or predicted to be of the *Cassia* class. We can also see that the model flawlessly recalled both the *Quercus* and the *Cassia* classes, when presented with unseen images of those classes.

Results

Training and testing was a long process, as each run with 4400 training images, 1100 validation images and 200 testing images took around 10 hours to complete.

Initially, while going through trial and error to find the optimal value for the learning rate, results were poor. There was constantly a rapid convergence to a fixed 0.2609 accuracy score for training and 0.2727 accuracy score for validation, no matter the value chosen, and we could not find a reason for why that was occurring. The breakthrough came when the learning rate was dropped to very small levels, more specifically 0.0001 and lower. In the 0.0001 range we saw an improvement that doubled the accuracy of the model, to around 0.46 in validation accuracy. Decreasing the learning rate by one more order of magnitude, to the 0.00001 range brought about another increase in accuracy, with performances averaging around 0.74 in validation accuracy. The final breakthrough was found by setting the learning rate to 0.000013, where the validation accuracy reached 0.98 and higher relatively consistently. In terms of training accuracy, most runs with that learning rate would also reach perfect or near-perfect accuracy levels of 0.98 or higher.

Inspecting the different runs and especially the graph of each run, showing the evolution of training and validation accuracy and loss throughout the epochs, we notice sudden, significantly large jumps in accuracy during training (Fig.18).

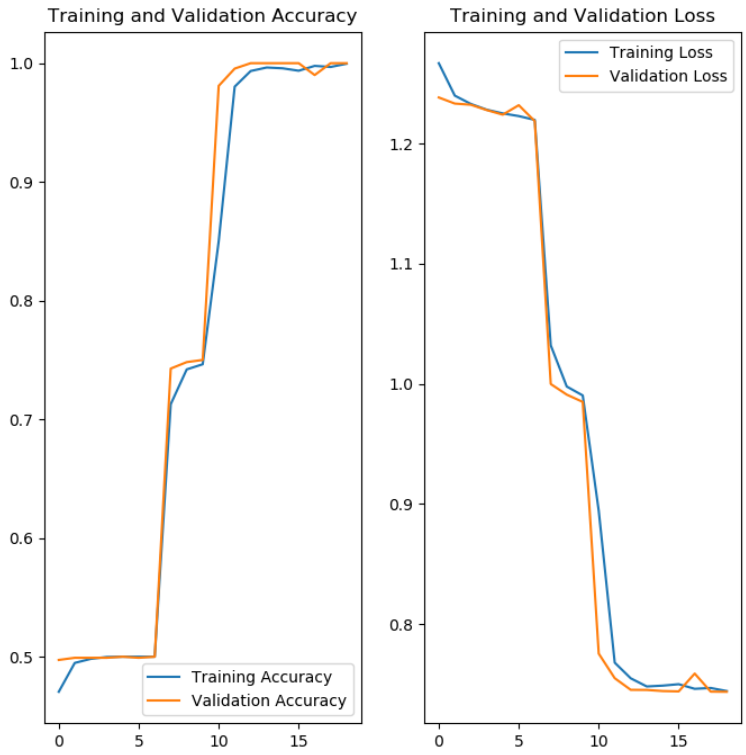

Figure 18: Example graph showing the sudden large jumps in accuracy and loss.

After completing the adjustment of the learning rate, the number of layers, the number of filters, among other parameters, we launched 25 complete runs, which incuded the test step. Out of the 25 runs, 20 were sucessful, while 5 outlier runs remained completely stuck at a poor local minimum early in the process. The outlier runs have not been considered in the following results:

	Betula	*Bridelia*	*Cassia*	*Quercus*
Average precision	**0.814**±0.2	**0.784**±0.201	**0.79**±0.176	**0.973**±0.075
Average recall	**0.682**±0.324	**0.804**±0.368	**0.932**±0.064	**0.748**±0.368

Table 1: Precision and recall scores of the 20 successful test runs, by species (mean and standard deviation)

In general, the results of these test runs (Table 1) are relatively positive. Overall, considering all classes, the precision averaged **0.84** and the recall averaged **0.792**, while the average accuracy for the model in these runs was **0.793**. Despite the positives, the standard deviation is quite high in most of the metrics observed, which indicates that each run is often vastly different from the previous. This is indicative of the complexity of the problem at hand and the lack of sufficient data.

Looking at the information in the reports and the confusion matrices generated by the script, we notice that often some classes have high scores for recall, while others do very poorly, barely above expected for randomness, although the classes affected vary between runs. Having a high recall on class X means that whenever an image of class X appears, it is correctly predicted. The reason why the precision for class X would be lower than the recall is that some images of other classes are being incorrectly predicted as being part of class X.

	precision	recall	f1-score
betula	0.77	0.99	0.86
bridelia	0.81	0.93	0.86
cassia	0.95	0.30	0.46
quercus	0.93	0.94	0.94
accuracy			0.83
macro avg	0.86	0.79	0.78
weighted avg	0.86	0.83	0.81

Figure 19: Report showing an example precision-recall disparity.

This example report (Fig. 19) shows that 3 species have high recall, with values 0.93 and above. This would indicate that the model can strongly recognize an image of this class when it is being presented with it. However, the precision for two of those classes is not as high, while the *Quercus* essentially maintains a similar score. We notice a large disparity in class *Betula* (0.77 in precision, 0.99 in recall). Not only that, but *Cassia*, the only class with low recall happens to score a 0.95 in precision, which presents another great disparity (0.95 in precision, 0.30 in recall). This indicates that whenever an image

of class *Cassia* is presented, the model classifies it incorrectly 70% of the time. Those incorrect guesses are what is going to affect the precision of the other classes. As mentioned above, the *Cassia* class also scores 0.95 in precision, which is surprising considering the poor recall score. It indicates that whenever the model classifies an image as *Cassia*, it is correct 95% of the time. We suppose that the model issuing *Cassia* as a prediction happens only very sparingly and only on the few images where the model has strong confidence that is the right class. Any other images are predicted to be one of the other three classes, with a preference for *Betula* and to some extent *Bridelia*, judging by how precision was affected in those classes.

In the graph shown, this issue happens exclusively with the *Cassia* class. However, in earlier runs, where we were trying to find the optimal value for the learning rate and obtained lower scores for accuracy, we observed the same phenomenon with two and even three classes having lower precision or recall. In fact, in our final test runs (Table 1), the *Cassia* class had a similar precision score as to the *Betula* class and was the class with highest recall.

We believe the complexity of the problem the model is attempting to classify means that the minima are difficult to find. However, when they are found it causes a very significant decrease in loss: the model has made a momentous breakthrough in a particular class. Having the learning rate set too high caused the model to jump over these fine minima and be stuck in that constant low score for accuracy and high loss. In addition, the considerable total number of trainable parameters, 47732, further conveys the complexity of the problem.

Conclusions

This project aimed to create a CNN model to classify cleared leaves from four different species, by means of an automatic script that extracts part of the venation network of the leaf and trains the model based on an augmented dataset. Based on the analysis of the statistics and graphs of our results, we can conclude that CNN models are capable of distinguishing species based exclusively on the visual information provided by the venation network, with relatively high precision and recall.

However, the lack of unique original images meant that all the data augmentation for training model had to come from a very small pool of cleared leaves, in addition to the issue brought about by the appreciable level of complexity of the problem to solve. We believe that it is one of the main reasons why the model's performance was unstable, lacked robustness and consistency.

Based on this, and because we believe the script itself is easily adaptable, future research could build on this project when in possession of a heavily enriched dataset, with higher orders of magnitude of original images per class, and perchance more classes as well.

Sources

1. Nic Lughadha, E., et al. (2016). Counting counts: revised estimates of numbers of accepted species of flowering plants, seed plants, vascular plants and land plants with a review of other recent estimates. *Phytotaxa* 272 (1): p.82–88

2. Wäldchen, J., Rzanny, M., Seeland, M., & Mäder, P. (2018). Automated plant species identification-Trends and future directions. *PLoS computational biology*, 14(4), e1005993. https://doi.org/10.1371/journal.pcbi.1005993

3. Seeland, M., Rzanny, M., Boho, D. et al. Image-based classification of plant genus and family for trained and untrained plant species. *BMC Bioinformatics* 20, 4 (2019). https://doi.org/10.1186/s12859-018-2474-x

4. Neto, J. C., et al. (2006) Plant Species Identification Using Elliptic Fourier Leaf Shape Analysis. *Computers and Electronics in Agriculture*, vol. 50, no. 2, pp. 121–34.

5. Bruno, O. M., et al. (2008) Fractal Dimension Applied to Plant Identification. *Information Sciences*, vol. 178, no. 12, pp. 2722–33.

6. Yanikoglu, B., et al. (2014) Automatic Plant Identification from Photographs. *Machine Vision and Applications*, vol. 25, no. 6, pp. 1369–83.

7. Hati, S. & Sajeevan, G. (2013). Plant Recognition from Leaf Image through Artificial Neural Network. International Journal of Computer Applications. 62. 15-18

8. Wick, C. and Puppe, F. (2017). Leaf Identification Using a Deep Convolutional Neural Network. p. arXiv:1712.00967

9. Lee, S. H., et al. (2015) Deep-Plant: Plant Identification with Convolutional Neural Networks. 2015 IEEE International Conference on Image Processing (ICIP), pp. 452–56

10. Gu, X., Du, J.X., Wang, X.F. (2005). Leaf Recognition Based on the Combination of Wavelet Transform and Gaussian Interpolation. In: Huang, D.S., Zhang, X.P., Huang, G.B. (eds). Advances in Intelligent Computing. Lecture Notes in Computer Science, vol. 3644.

11. Wilf, P., et al. (2016) Computer Vision Cracks the Leaf Code. Proceedings of the National Academy of Sciences, vol. 113, no. 12, pp. 3305–10

12. Larese, M.G., Namías, R., Craviotto, R.M., Arango, M.R., Gallo, C., Granitto, P.M. (2014). Automatic classification of legumes using leaf vein image features. Pattern Recognition. 47 (1), p.158–168.

13. Voulodimos, A., Doulamis, N., Doulamis, A., & Protopapadakis, E. (2018). Deep Learning for Computer Vision: A Brief Review. Computational intelligence and neuroscience, 2018, 7068349. https://doi.org/10.1155/2018/7068349

14. Chaffey N. (2014). Raven biology of plants, 8th edn. *Annals of Botany*, 113(7), vii. https://doi.org/10.1093/aob/mcu090

15. Hickey, L.J. (1973) Features of a simple leaf [Online Image]. New York Botanical Garden (NYBG), Steere Herbarium. http://sweetgum.nybg.org/science/glossary/glossary-details/?irn=1800

16. Mori, S.A. Leaf venation of a species of *Eschweilera* [Online Image]. New York Botanical Garden, Steere Herbarium. http://sweetgum.nybg.org/science/glossary/glossary-details/?irn=156

17. Das, A., Bucksch, A., Price, C.A. et al. (2014). ClearedLeavesDB: an online database of cleared plant leaf images. Plant Methods 10, 8. doi:10.1186/1746-4811-10-8

18. Bates, J.C. (1931). A Method for Clearing Leaves. *The American Naturalist* Volume 65, Number 698: p.288

19. POWO (2019). *Plants of the World Online*. Facilitated by the Royal Botanic Gardens, Kew. http://www.plantsoftheworldonline.org/ (Accessed November 18, 2020).

20. Price, C.A., Symonova, O., Mileyko, Y., Hilleyand, T., Weitz, J.S. (2011). Leaf Extraction and Analysis Framework Graphical User Interface: Segmenting and Analyzing the Structure of Leaf Veins and Areoles. *Plant Physiology*, 155 (236-245). http://www.leafgui.biology.gatech.edu/ (currently down) https://www.quantitative-plant.org/software/leaf-gui

21. Dirnberger, M., Kehl, T., Neumann, A. (2015). NEFI: Network Extraction From Images. *Scientific Reports* 5, 15669. http://nefi.mpi-inf.mpg.de

22. Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. https://opencv.org

23. Le Cun, Y., et al. (1990). Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems* 2: p. 396–404

24. Hassaballah M., Abdelmgeid A.A., Alshazly H.A. (2016) Image Features Detection, Description and Matching. In: Awad A., Hassaballah M. (eds) Image Feature Detectors and Descriptors. Studies in Computational Intelligence, vol 630. Springer, Cham. https://doi.org/10.1007/978-3-319-28854-3_2

25. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation* (*OSDI* 16): p. 265–283. https://www.tensorflow.org

26. Suárez-Paniagua, V., Segura-Bedmar, I. (2018). Evaluation of pooling operations in convolutional architectures for drug-drug interaction extraction. *BMC Bioinformatics* 19, 209. doi:10.1186/s12859-018-2195-1

27. Dozat, T. (2016). Incorporating Nesterov Momentum into Adam.

28. Kingma, D. P., & Ba, J. A. (2019). A method for stochastic optimization.

29. Nesterov, Y. (1983). A method for solving the convex programming problem with convergence rate O(1/k^2). *Proceedings of the USSR Academy of Sciences*, 269, 543-547.

30. Polyak, B. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4, 1-17.

31. Sutskever, I., Martens, J., Dahl, G. & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. *Proceedings of the 30th International Conference on Machine Learning*, in *PMLR* 28(3):1139-1147

32. Hoang, N. (2020, June 14). Full review on optimizing neural network training with Optimizer. https://towardsdatascience.com/full-review-on-optimizing-neural-network-training-with-optimizer-9c1acc4dbe78

Appendix A

Image_processing.py

```
import math
import os
import numpy as np
from imutils import contours
from imutils import perspective
import imutils
import cv2


def save_image(img, name):
    cv2.imwrite(name, img)


def show_image(img):
    cv2.imshow("Image", img)
    cv2.waitKey()
    return


def midpoint(ptA, ptB):
    return (ptA[0] + ptB[0]) * 0.5, (ptA[1] + ptB[1]) * 0.5


def point_rotate_around_origin(origin, point, angle):
    """
    radians.
    """
    ox, oy = origin
    px, py = point

    qx = ox + math.cos(angle) * (px - ox) - math.sin(angle) * (py - oy)
    qy = oy + math.sin(angle) * (px - ox) + math.cos(angle) * (py - oy)
    return int(qx), int(qy)


def extract_square_from_image(img_src, dest):
    image = cv2.imread(img_src, 0)
    # cv2.imshow("Image", image)
    # cv2.waitKey(0)

    gray = cv2.GaussianBlur(image.copy(), (7, 7), 0)
    # cv2.imshow("Image", gray)
    # cv2.waitKey(0)

    thresh = cv2.adaptiveThreshold(gray, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 7, 2)
    # cv2.imshow("Image", thresh)
    # cv2.waitKey(0)

    edged = cv2.Canny(thresh, 50, 100)
    # cv2.imshow("Image", edged)
    # cv2.waitKey(0)
```

```
    edged = cv2.dilate(edged, None, iterations=2)
    # cv2.imshow("Image", edged)
    # cv2.waitKey(0)

    edged = cv2.erode(edged, None, iterations=2)
    # cv2.imshow("Image", edged)
    # cv2.waitKey(0)

    cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
                            cv2.CHAIN_APPROX_SIMPLE)
    cnts = imutils.grab_contours(cnts)

    (cnts, _) = contours.sort_contours(cnts)

    contour_sizes = [(cv2.contourArea(contour), contour) for contour
in cnts]
    contour = max(contour_sizes, key=lambda x: x[0])[1]

    # compute the rotated bounding box of the contour
    orig = image.copy()
    orig = cv2.cvtColor(orig, cv2.COLOR_GRAY2BGR)

    largest_contour = cv2.polylines(orig.copy(), contour, 1, (0, 255,
255))
    # cv2.imshow("Image", largest_contour)
    # cv2.waitKey(0)

    rect = cv2.fitEllipse(contour)
    box = cv2.boxPoints(rect)
    box = np.array(box, dtype="int")

    box = perspective.order_points(box)

    angle = math.radians(rect[-1])

    cv2.line(orig, (int(box[0][0]), int(box[0][1])), (int(box[1][0]),
int(box[1][1])),
             (0, 255, 0), 2)
    cv2.line(orig, (int(box[0][0]), int(box[0][1])), (int(box[3][0]),
int(box[3][1])),
             (0, 255, 0), 2)
    cv2.line(orig, (int(box[1][0]), int(box[1][1])), (int(box[2][0]),
int(box[2][1])),
             (0, 255, 0), 2)
    cv2.line(orig, (int(box[2][0]), int(box[2][1])), (int(box[3][0]),
int(box[3][1])),
             (0, 255, 0), 2)

    center_x = (int(box[0][0]) + int(box[2][0])) / 2
    center_y = (int(box[0][1]) + int(box[2][1])) / 2
```

```
    gray2 = thresh.copy()
    # gray2 = cv2.cvtColor(gray2, cv2.COLOR_GRAY2BGR)

    # cv2.circle(gray2, (int(center_x), int(center_y)), 5, (0, 0, 255),
-1)

    point_1_rect = point_rotate_around_origin((center_x, center_y),
(int(center_x) - 150, int(center_y) + 150), angle)
    point_2_rect = point_rotate_around_origin((center_x, center_y),
(int(center_x) + 150, int(center_y) + 150), angle)
    point_3_rect = point_rotate_around_origin((center_x, center_y),
(int(center_x) + 150, int(center_y) - 150), angle)
    point_4_rect = point_rotate_around_origin((center_x, center_y),
(int(center_x) - 150, int(center_y) - 150), angle)

    # cv2.circle(gray2, point_1_rect, 5, (255, 0, 0), -1)
    # cv2.circle(gray2, point_2_rect, 5, (255, 0, 0), -1)
    # cv2.circle(gray2, point_3_rect, 5, (255, 0, 0), -1)
    # cv2.circle(gray2, point_4_rect, 5, (255, 0, 0), -1)
    #
    # cv2.line(gray2, point_1_rect, point_2_rect,(255, 0, 0), 2)
    # cv2.line(gray2, point_1_rect, point_4_rect,(255, 0, 0), 2)
    # cv2.line(gray2, point_2_rect, point_3_rect,(255, 0, 0), 2)
    # cv2.line(gray2, point_3_rect, point_4_rect,(255, 0, 0), 2)

    # cv2.imshow("Image", orig)
    # cv2.waitKey(0)

    # cv2.imshow("Image", gray2)
    # cv2.waitKey(0)

    min_area_rect = cv2.minAreaRect(np.array([point_1_rect,
point_2_rect, point_3_rect, point_4_rect]))

    box_rect = cv2.boxPoints(min_area_rect)
    box_rect = np.array(box_rect, dtype="int")

    width = int(min_area_rect[1][0])
    height = int(min_area_rect[1][1])

    src_pts = box_rect.astype("float32")

    dst_pts = np.array([[0, height - 1],
                        [0, 0],
                        [width - 1, 0],
                        [width - 1, height - 1]], dtype="float32")

    persp_transf = cv2.getPerspectiveTransform(src_pts, dst_pts)

    warped = cv2.warpPerspective(gray2, persp_transf, (width, height))

    # cv2.imshow("Image", warped)
    # cv2.waitKey()

    save_image(warped, dest)

    return
```

Appendix B

cnn.py

```
from __future__ import absolute_import, division, print_function,
unicode_literals
import image_processing
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow_core.python.keras.callbacks import EarlyStopping
import os
import numpy as np
from sklearn import metrics
import matplotlib.pyplot as plt

pictures_path = os.path.join(os.getcwd(), 'pictures')
test_pictures_path = os.path.join(os.getcwd(), 'test')

list_subfolders = [f.name for f in os.scandir(pictures_path) if
os.path.isdir(f)]

os.makedirs(os.path.join(os.getcwd(), 'processed_images'),
exist_ok=True)
os.makedirs(os.path.join(os.getcwd(), 'processed_test_images'),
exist_ok=True)

image_path_dictionary = {}
test_path_dictionary = {}

for subfolder in list_subfolders:
    image_path_dictionary.update({subfolder: os.path.join(pictures_path,
subfolder)})
    test_path_dictionary.update({subfolder:
os.path.join(test_pictures_path, subfolder)})
    os.makedirs(os.path.join(os.getcwd(), 'processed_images\\' +
subfolder), exist_ok=True)
    os.makedirs(os.path.join(os.getcwd(), 'processed_test_images\\' +
subfolder), exist_ok=True)

for key, path in image_path_dictionary.items():
    pictures = [picture.name for picture in os.scandir(path) if
os.path.isfile(picture)]
    for picture in pictures:
        image_processing.extract_square_from_image(
            os.path.join(os.getcwd(), 'pictures\\' + key + "\\" +
picture),
            os.path.join(os.getcwd(), 'processed_images\\' + key + '\\'
+ picture))

for key, path in test_path_dictionary.items():
    pictures = [picture.name for picture in os.scandir(path) if
os.path.isfile(picture)]
    for picture in pictures:
        image_processing.extract_square_from_image(
            os.path.join(os.getcwd(), 'test\\' + key + "\\" + picture),
            os.path.join(os.getcwd(), 'processed_test_images\\' + key +
'\\' + picture))
```

```
pictures_path = os.path.join(os.getcwd(), 'processed_images')
test_pictures_path = os.path.join(os.getcwd(), 'processed_test_images')

batch_size = 1
epochs = 50
IMG_HEIGHT = 300
IMG_WIDTH = 300

image_generator = ImageDataGenerator(zoom_range=[0.9, 1.1],
vertical_flip=True, rotation_range=5)

train_data_gen =
image_generator.flow_from_directory(batch_size=batch_size,
classes=list_subfolders, directory=pictures_path, shuffle=True,
color_mode='grayscale', target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode='sparse')

test_data_gen =
image_generator.flow_from_directory(batch_size=batch_size,
classes=list_subfolders, directory=test_pictures_path, shuffle=False,
color_mode='grayscale', target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode='sparse')

print(train_data_gen.class_indices)

num_files = [len(os.listdir(os.path.join(pictures_path, subfolder))) for
subfolder in os.listdir(pictures_path)]

counter = sum(num_files)
while counter < 5500:
    batch, label = train_data_gen.next()
    for img in batch:
        class_label =
list(train_data_gen.class_indices.keys())[list(train_data_gen.class_indi
ces.values()).index(int(label))]
        name = str(label) + '_' + class_label + '_' + str(counter + 1) +
'.PNG'
        image_processing.save_image(img, pictures_path + '\\' +
class_label + '\\' + name)
    counter += 1

num_test_files = [len(os.listdir(os.path.join(test_pictures_path,
subfolder))) for subfolder in os.listdir(test_pictures_path)]
```

```
counter = sum(num_test_files)
while counter < 200:
    batch, label = test_data_gen.next()
    for img in batch:
        class_label = list(test_data_gen.class_indices.keys())[
list(test_data_gen.class_indices.values()).index(int(label))]
        name = str(label) + '_' + class_label + '_' + str(counter + 1) +
'.PNG'
        image_processing.save_image(img, test_pictures_path + '\\' +
class_label + '\\' + name)
    counter += 1

final_image_generator = ImageDataGenerator(validation_split=0.2)

train_data_gen =
final_image_generator.flow_from_directory(batch_size=batch_size,
classes=list_subfolders, directory=pictures_path, shuffle=True,
color_mode='grayscale', target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode='sparse', subset='training')

val_data_gen =
final_image_generator.flow_from_directory(batch_size=batch_size,
classes=list_subfolders, directory=pictures_path, shuffle=False,
color_mode='grayscale', target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode='sparse', subset='validation')

model = Sequential([
    Conv2D(16, 5, padding='valid', activation='relu',
input_shape=(IMG_HEIGHT, IMG_WIDTH, 1)),
    MaxPooling2D(),
    Conv2D(16, 5, padding='valid', activation='relu'),
    MaxPooling2D(),
    Conv2D(32, 3, padding='valid', activation='relu'),
    MaxPooling2D(),
    Conv2D(32, 3, padding='valid', activation='relu'),
    MaxPooling2D(),
    Conv2D(32, 3, padding='valid', activation='relu'),
    MaxPooling2D(),
    Conv2D(32, 3, padding='valid', activation='relu'),
    MaxPooling2D(),
    Flatten(),
    Dense(64, activation='relu'),
    Dense(4, activation='softmax')
])

nadam_custom = tensorflow.keras.optimizers.Nadam(
    learning_rate=0.000013)

es = EarlyStopping(monitor='val_loss', mode='min', verbose=1,
patience=6, min_delta=0.005)
```

```
model.compile(optimizer=nadam_custom,
loss=tensorflow.keras.losses.SparseCategoricalCrossentropy(from_logits=T
rue), metrics=['sparse_categorical_accuracy'])

history = model.fit(train_data_gen, steps_per_epoch=4400, epochs=epochs,
validation_data=val_data_gen, validation_steps=1100, callbacks=[es])

print(model.summary())

test_data_gen =
image_generator.flow_from_directory(batch_size=batch_size,
classes=list_subfolders, directory=test_pictures_path, shuffle=False,
color_mode='grayscale', target_size=(IMG_HEIGHT, IMG_WIDTH),
class_mode='sparse')

predictions = model.predict_generator(test_data_gen, steps=200)

predicted_classes = np.argmax(predictions, axis=1)
true_classes = test_data_gen.classes
class_labels = list(test_data_gen.class_indices.keys())
report = metrics.classification_report(true_classes, predicted_classes,
target_names=class_labels)
print(report)
matrix = metrics.confusion_matrix(true_classes, predicted_classes)

plt.imshow(matrix, interpolation='nearest', cmap="OrRd")
plt.title("Confusion Matrix")
plt.colorbar()
tick_marks = np.arange(len(class_labels))
plt.xticks(tick_marks, class_labels, rotation=45)
plt.yticks(tick_marks, class_labels)
plt.tight_layout()
plt.ylabel('True class')
plt.xlabel('Predicted class')

acc = history.history['sparse_categorical_accuracy']
val_acc = history.history['val_sparse_categorical_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(es.stopped_epoch+1)

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.show()
```