CISC5800: Machine Learning Final Project

Ryan Cruise, S.J.
M.S. in Data Science
Fordham University
Bronx, U.S.A
rc6@fordham.edu

***Abstract*—Plant maladies often present themselves visually on the leaves of plants and are treated with topical chemicals. These chemicals can lead to unintended consequences for the environment and for humans. This project focuses on developing a convolutional neural network (CNN) that can classify plant diseases from images of leaves using the Plant Leaf Diseases Training Dataset from Kaggle. The CNN is inspired by the VGG-16 model and was trained on a local desktop PC. A variety of different fine-tuning methods were used to increase the performance of the model. However, the best performance came from doing Stratified K-Fold Cross Validation, achieving a validation accuracy of ~88%.**

Keywords—Machine Learning, Convolutional Neural Networks, Image Processing, Leaf Disease Detection, Stratified K-Fold

I. Introduction

One practical consideration for farmers, and a major field of agronomic study, is disease and pest control prevention. Evidence of diseases and pest predation can often be seen on the leaves of crops through spots, holes, or other forms of discoloration. The presence of leaf-based maladies has long since been known to be inversely correlated with yield [1]. Thus, a critical aspect of growing crops to maximize yield includes identifying and treating pathogens and pests that cause harm to crop leaves. Currently, much of the work of identifying leaf-based pathogens has been done by farmers themselves, agronomists, or other agricultural employees or technicians. Further, the treating of these pathogens often involves large-scale application of pesticides, bactericides, and fungicides on crops. These crop treatments often do not involve application on specific or individual plants, but instead treatments are applied across entire areas of farmland, often on the order of acres. While there is a host of different active ingredients in pesticides, bactericides, and fungicides, it has been shown that their application on crops can lead to unintended consequences in humans and the environment [2]. Therefore, reducing the amount of chemicals used on crops by applying them to specific plants or localized sections of farmland could reduce any unintended consequences, while also increasing the farmer's profit by saving in chemical costs.

Until recently, this would only be possible by someone individually inspecting and identifying pathogens on plants, which can quickly become impractical when dealing with large sections of farmland. However, recent advancements in Convolutional Neural Networks (CNN) have shown promising results in recognizing plant pathogens from images of leaves [3], [4], [5]. In fact, as recently as July of 2024, Jawadul Karin et al. were able to use Nvidia's Jeston Nano and a "lightweight CNN" to identify grape leaf diseases in real-time [6]. Developing the technology behind Convolutional Neural Networks in plant disease detection can reduce the time needed to identify diseases, as well as reduce the quantity of chemicals needed to treat the disease. Hence, my final project will be working with a dataset from Kaggle of plant images to create a CNN that classifies each image based on the presence of disease on the leaf. I will use a CNN structure that is loosely based on the VGG-16 CNN model, created by the Visual Geometry Group at the University of Oxford [7]. Due to the cost and limited availability of AI GPUs and TPUs, I will be doing the majority of the model refinement on my personal desktop PC, using my consumer-grade Nvidia GPU instead of Google Colaboratory.

II. Plant Leaf Dataset

A. Dataset Source

I retrieved this dataset from Kaggle.com. This dataset is titled "Plant Leaf Diseases Training Dataset" and was created by Nirmal Sankalana [8]. It was last updated 10 months ago and has a public domain license.

B. Dataset Description

The Plant Leaf Dataset has a total of 116, 147 sample images of plant leaves. There is a total of 20 different plant species represented in the samples with various different maladies, creating 71 different classes. For example, the class "Sugarcane___healthy" has 522 images of sugarcane plant leaves that are healthy; and the class "Rose___rust" has 4,953 images of rose leaves that are suffering from a specific kind of fungus that is called "rust." The Plant Leaf Diseases Training Dataset is very imbalanced, with the most represented class, "Cassava___mosaic_disease," having 13,158 samples, and the least represented class, "Potato___nematode," having only 68 samples. This class imbalance will become the primary concern for my model accuracy later on in this project.

Below is a table of the five most frequent classes and the five least frequent classes. As you can see, the classes do not only include healthy and diseased plants, but also leaves that have evidence of being eaten by bugs or have images of bugs physically present on the leaves.

Class	Plant	Samples
Cassava___mosaic_disease	Cassava	13,158
Orange___citrus_greening	Orange	5,507
Tomato___leaf_curl	Tomato	5,357

Soybean___healthy	Soybean	5,090
Rose___slug_sawfly	Rose	4,979
…..	…..	…..
Apple___brown_spot	Apple	215
Watermelon___healthy	Watermelon	205
Coffee___red_spider_mite	Coffee	167
Watermelon___anthracnose	Watermelon	155
Potato___nematode	Potato	68

Table 1—Plant Dataset Class Samples

The Plant Leaf Diseases Training Dataset is structured by one parent directory containing all the sub-directories of the class data. Each class is a sub-directory, containing all the image samples of that class within it. This structure is visualized in Figure 1 below:

```
Parent_directory/
    class_a/
        a_img1.jpg
        a_img2.jpg
        ….
    class_b/
        b_img1.jpg
        b_img2.jpg
        ….
```

Figure 1 — Dataset Directory Structure

The directory structure of the dataset will become very relevant later on in the paper because the directory structure plays a role in how the samples are processed and loaded into the CNN pipeline.

C. *Examples of Plant Leaf Images*

The majority of leaf images are sized as 256 pixels by 256 pixels. So, this dimension became the size that all images were processed into the CNN. Since all the images are in color, the input dimensions of each image is (256, 256, 3), to include the three RGB channels. I have included here a few examples of the leaf images so you have a sense of what the images look like before they get processed and classified.

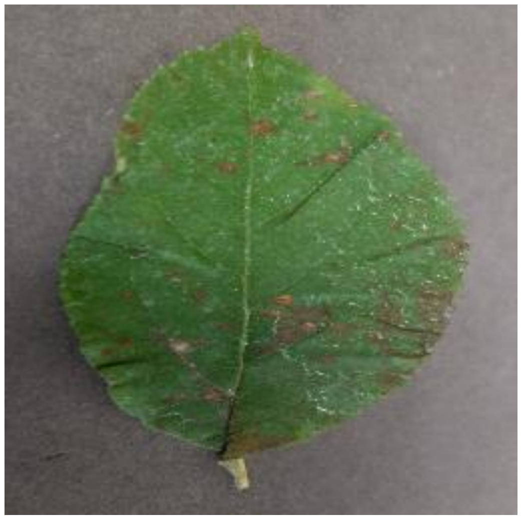


Figure 2 —Apple Leaf with Rust Fungus

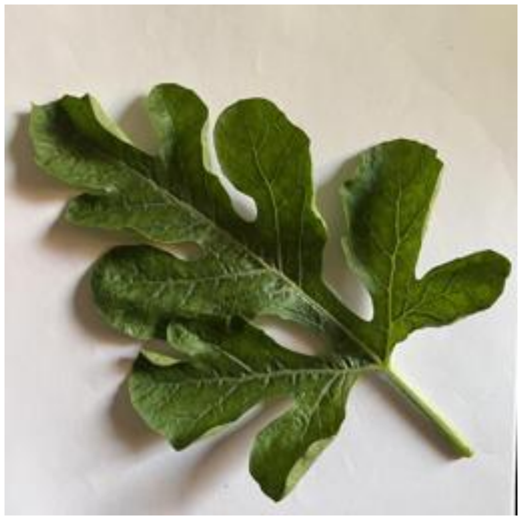


Figure 3—Healthy Watermelon Leaf

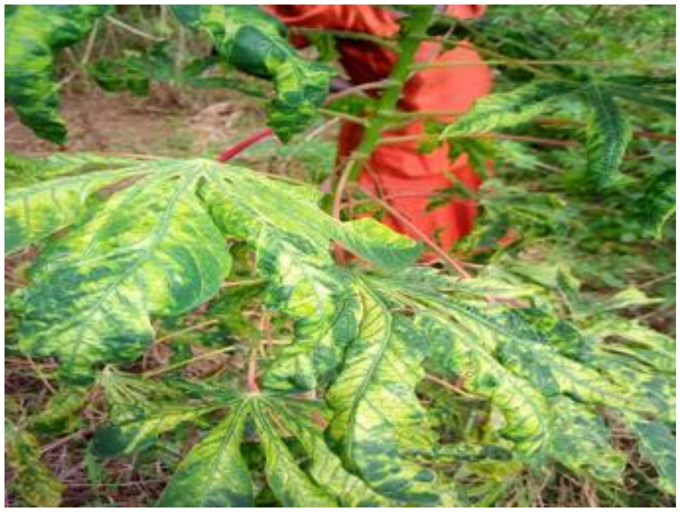


Figure 4—Cassava Leaf with Mosaic Disease

III. Why CNN and VGG-16?

A. Convolutional Neural Networks

Convolutional Neural Networks perform particularly well at image classification tasks because they are designed to be *invariant*. Pattern recognition models that are invariant allow for transformations within the training data without altering the prediction, making the predictions relatively *invariant* to variances in the input variables [9]. CNN models achieve this by creating *feature maps* within each input, which are “small subregions of the image, and all of the units in a feature map are constrained to share the same weight values” [9]. CNNs are able to create these feature maps based on a property of images that states that nearby pixels are more strongly correlated than pixels that are farther apart. Thus, CNNs are able to identify local features within the image that are only dependent on small subregions of the image [9]. CNNs identify local features by applying kernels, or filters, to images. Filters identify features by “sliding across” the image and calculating the inner product of the filter matrix and image subregion. There are a variety of filters, with each filter specializing in highlighting certain image features. Multiple filters are often applied at each convolutional neural network layer, as in the case of VGG-16.

The Convolutional Layers, where the *feature maps* are created using filters, are the primary way in which CNNs differ from other neural network models. As shown by Figure 5, the process of CNNs starts with the image data being processed by the Convolutional Layers. A non-linear activation function, such as ReLU, is often introduced once the feature maps have been identified to increase the complexity of the input data. The pooling layer is used to reduce the spatial size (dimensions) of the feature maps, which helps reduce the number of computations needed and the time the model takes to train. Once the spatial size has been reduced, the feature maps are then passed through at least one fully-connected (FC) neural network layer before making a classification. The softmax function, which is often used for classifying multiple categorical variables, is typically used for the final FC layer. The size of the output layer is a Kx1 matrix, where K is the number of classes.

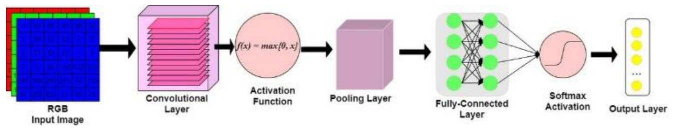


Figure 5—Convolutional Neural Network Architecture [10]

Therefore, Convolutional Neural Networks are useful for this project because they allow for variances in position and orientation of leaves. Each disease, fungus, or pest manifests itself similarly in all leaves for each plant species, but it is not reasonable to assume that each leaf image will be positioned and oriented in the same way. Also, even if each leaf were in the same position for all image samples, there is no way of controlling in which region of the leaf that the malady presents itself. For example, in the case of Figure 2 (the apple leaf suffering from the rust fungus), the brown rust spots are going to look similarly to all rust spots on all apple leaves, but the position and orientation of each rust spot will be different on each leaf. Thus, the model needed for the task of identifying plant diseases must be relatively invariant to the location and orientation of the disease on the plant leaf.

B. VGG-16 Background

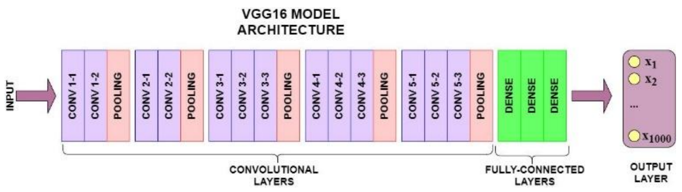


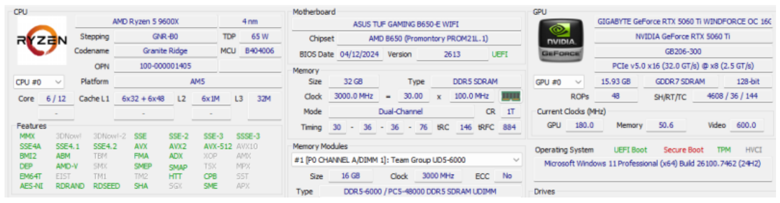
Figure 6—VGG-16 Architecture [10]

The VGG-16 Convolutional Neural Network model was first introduced in 2014 by the Visual Geometry Group at the University of Oxford [7]. It is based on the 2012 AlexNet CNN model that achieved an accuracy of 92% when trying to classify 1,000 classes [11]. As shown by Figure 6, the VGG-16 model is comprised of 13 convolutional layers and 3 fully-connected layers, creating 16 layers in total. VGG-16 is a very famous and simple model, so it became a useful convolutional neural network model to use as a starting point for creating my own CNN. As stated above, I will be primarily using my personal desktop PC with a consumer-grade Nvidia GPU to train the custom CNN model.

IV. System Configuration

A. System Specifications

For this project, I am using a custom-built PC that I assembled about one year ago. Its operating system is Windows 11 Pro that has Window’s Subsystem for Linux (WSL2) installed on it. The CPU is AMD’s Ryzen 5 9600X and the GPU is Nvidia’s 5060 Ti with 16GB GDDR7 VRAM. The 5060 Ti is part of Nvidia’s Blackwell Architecture and has 4,608 CUDA cores and 759 Tensor Cores. I also have available to me Nvidia’s 1070 GPU with 8GB VRAM, but I would eventually decide not to use this GPU. The PC has 32GB of DDR5 SDRAM running at 6,000 MT/s. There are 11 fans installed in the PC for cooling, not including the two fans on the GPU. Typically, I use the Nvidia 5060 Ti for video output, but for the duration of this project I switched the DisplayPort cable to my motherboard so that I could use the CPU’s integrated graphics (AMD Radeon) for video output instead. Generally speaking, this is a fairly capable PC, but this project pushed the hardware to its limit on multiple occasions. I found that my biggest hardware limitation for this project was the lack of VRAM (16GB) on the GPU.



```
CPU
          AMD Ryzen 5 9600X                 4 nm
Stepping  GNR-B0                  TDP       65 W
Codename  Granite Ridge           MCU       B404006
OPN       100-000001405
CPU #0    Platform  AM5
Core 6 / 12   Cache L1  6x32 + 6x48   L2  6x1M   L3  32M
Features
MMX     3DNow!   3DNow!-2  SSE    SSE-2   SSE-3    SSSE-3
SSE4A   SSE4.1   SSE4.2    AVX    AVX2    AVX-512  AVX10
BMI2    ABM      TBM       FMA    ADX     XOP      AMX
DEP     AMD-V    SMX       SMEP   SMAP    TSX      MPX
EM64T   EIST     TM1       TM2    HTT     CPB      SST
AES-NI  RDRAND   RDSEED    SHA    SGX     SME      APX

Motherboard
           ASUS TUF GAMING B650-E WIFI
Chipset    AMD B650 (Promontory PROM21L.1)
BIOS Date  04/12/2024   Version  2613   UEFI
Memory
Size    32 GB          Type  DDR5 SDRAM
Clock   3000.0 MHz  =  30.00  x  100.0 MHz
Mode    Dual-Channel                  CR  1T
Timing  30 - 36 - 36 - 76  tRC  146  tRFC  884
Memory Modules
#1 [P0 CHANNEL A/DIMM 1]: Team Group UD5-6000
Size  16 GB      Clock  3000 MHz      ECC  No
Type  DDR5-6000 / PC5-48000 DDR5 SDRAM UDIMM

GPU
         GIGABYTE GeForce RTX 5060 Ti WINDFORCE OC 16G
         NVIDIA GeForce RTX 5060 Ti
         GB206-300
         PCIe v5.0 x16 (32.0 GT/s) @ x8 (2.5 GT/s)
GPU #0   15.93 GB    GDDR7 SDRAM    128-bit
ROPs     48          SH/RT/TC       4608 / 36 / 144
Current Clocks (MHz)
GPU  180.0     Memory  50.6     Video  600.0
Operating System   UEFI Boot   Secure Boot   TPM   HVCI
Microsoft Windows 11 Professional (x64) Build 26100.7462 (24H2)
Drives
```

Figure 7—PC System Specifications

B. Software Configuration

The TensorFlow package for Python provides the option to use a PC’s GPU for processing instead of the CPU, so I exclusively used this library for training the CNN model. However, there was one significant problem—stable versions of TensorFlow do not yet support the Nvidia 5060 Ti because it is too new (released in April 2025). I considered using the older Nvidia 1070 (released in June 2016) instead, but this GPU is not supported by Nvidia's CUDA Toolkit because it is too old. The Nvidia CUDA Toolkit *and* TensorFlow are both needed for the GPU to run machine learning operations. Thankfully, TensorFlow offers an unstable version of its library that is updated every night (tf-nightly) that supports Nvidia's 5060 Ti [12]. Since the 5060 Ti has over twice as many cores as the 1070 (4,608 vs. 1,920), and both options would require significant software workarounds, I decided to go with the 5060 Ti. After many failed attempts to get TensorFlow to recognize my GPU as available for machine learning tasks, I finally found success by installing a custom Python wheel of TensorFlow and the Nvidia CUDA Toolkit within a Conda environment on WSL2 [13]. Please refer to Figure 8 for a visual representation of the software stack I used to run TensorFlow operations on my Nvidia 5060 Ti.

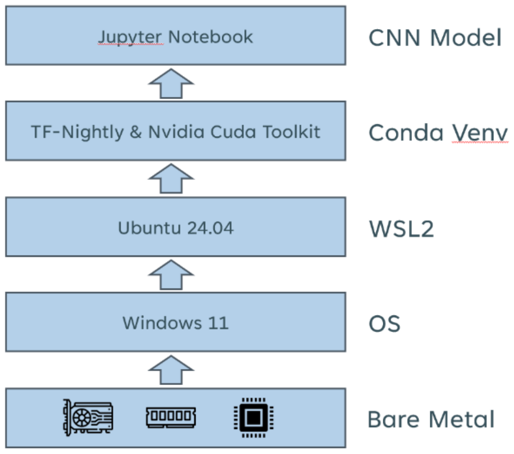
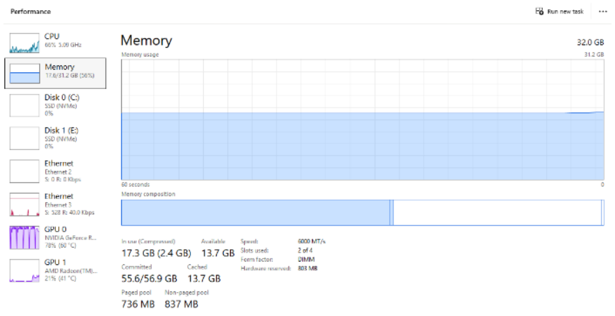


Figure 8—PC Software Stack for the CNN

C. *Proof of Concept*

Once TensorFlow recognized my GPU, I was ready to do a trail run to see if my computer could handle training a CNN. Before creating my own CNN that was customized for the Plant Leaf Diseases Training Dataset, I wanted to check to see if it could run any CNN model. So, I used the prebuilt VGG-16 package that is a part of the tensorflow.keras library. I followed an example on Kaggle by Subas Paudel to make sure my system was working correctly. My system was able to get an accuracy score that was similar to Subas's model on Kaggle (~67% validation accuracy) [14].

While my system was able to run the VGG-16 model, it did not do so without any issues. My biggest problem was that my GPU kept running out of VRAM. While 16GB of VRAM is ample for most GPU tasks, it was not quite enough for processing the CNN. However, running out of VRAM did not result in a critical failure. Once my GPU used up all of its VRAM, the system started to use the computer's SDRAM. I would frequently receive a warning message that looked something like this: "*ptxas warning: Registers are spilled to local memory in function 'gemm_fusion_dot_676', 280 bytes spill stores, 280 bytes spill loads*." GEMM stands for "General Matrix Multiplications," and is typically used for processing neural networks [15]. So, this warning message essentially means that there was not enough memory in the GPU to store all of the matrix multiplications, so some of the memory "spilled over" into the computer's SDRAM. As Figure 9 shows, the Nvidia 5060 Ti (GPU 0) is working at full-throttle, while a significant portion of the computer's memory is being consumed as overflow from the GEMM processes.



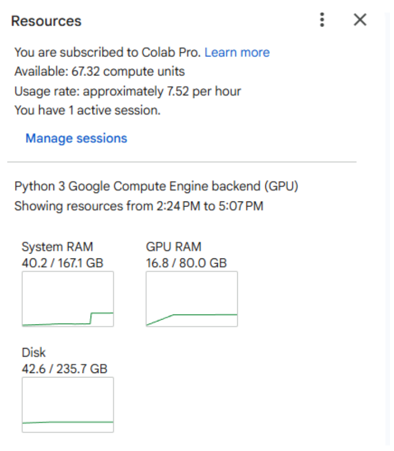
In use (Compressed) 17.3 GB (2.4 GB)
Available 13.7 GB
Committed 55.6/56.9 GB
Cached 13.7 GB
Paged pool 736 MB
Non-paged pool 837 MB
Speed: 6000 MT/s
Slots used: 2 of 4
Form factor: DIMM
Hardware reserved: 803 MB

Figure 9—GPU and Memory Usage During CNN Training

There were times during training that the RAM overflow consumed almost all of the RAM that is available to my CPU. The only reason this did not completely crash my system is because I limited WSL2 to only use 30GB of RAM, leaving only 2GB remaining for the operating system (which was just enough to keep everything running).

Another issue I ran into when running TensorFlow's CNN packages on my local desktop came from CPU instructions. Every time I imported TensorFlow, I would receive this warning message: "*This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations. To enable the following instructions: AVX512F AVX512_VNNI AVX512_BF16 AVX_VNNI, in other operations, rebuild TensorFlow with the appropriate compiler flags*." These instructions are part of AMD's x86-64 Architecture that the Ryzen 5 9600X is capable of using, but the current version of TensorFlow I was using was not able to use them [16]. I am fairly certain that this negatively impacted runtimes, and it may also have also reduced validation accuracy.

Given these warnings, I would eventually purchase a monthly subscription to Google Colaboratory's "Colab Pro" to get access to 100 Compute Units. However, as Figure 10 shows, I consumed about one third of the compute units in 2.5 hours, so this was not a viable option for me to use for the entire project. Also, Google Colab is showing that I am using 16.8 GB of GPU RAM, which means that my GPU is just shy of the necessary amount of VRAM to run the project. So, the benefits of running in Google Colab were not good enough for me to pay the premium price for Compute Units for this project.



Resources

You are subscribed to Colab Pro. Learn more
Available: 67.32 compute units
Usage rate: approximately 7.52 per hour
You have 1 active session.

Manage sessions

Python 3 Google Compute Engine backend (GPU)
Showing resources from 2:24 PM to 5:07 PM

System RAM 40.2 / 167.1 GB

GPU RAM 16.8 / 80.0 GB

Disk 42.6 / 235.7 GB

Figure 10—Google Colab Compute Unit Usage

V. CNN Experimentation

I explored a variety of different methods for improving the model from its base score (~67% validation accuracy). While I was somewhat limited by computational resources and time, I was able to find ways to improve the model without the use of GridSearchCV for hyperparameter testing. I was able to improve the model by about 20% but could not break the 90% validation accuracy barrier. Below are some of the changes I made to the base model that showed notable improvements to the model's validation accuracy.

A. Reducing Classes

Given that some of the original 71 classes have fewer than 100 samples, one of the first actions I took that improved accuracy was to reduce the number of classes to only include classes that had a significant sample size (and were agriculturally relevant). I created a "small" dataset that only included 21 classes with 56,244 samples. While I eventually returned to the original dataset, removing the underrepresented classes did improve the validation score.

Another reason for reducing the number of classes was to reduce the computational load for each run of the CNN model, which improved the runtime. This gave me more time to test different fine-tuning adjustments to the model. Given there are over 100,000 samples in the dataset, I wrote a .bash script to identify and move directories and folders when I created the small dataset. I will include the .bash script with the Python code in my GitHub repository.

B. Batch Size

I tried batch sizes of 16, 32, 64, and 128. I found that the batch size of 64 gave me the most consistently good results. If I had more computational resources, I would have tested the batch size with GridSearchCV using all four numbers above to truly see which batch size worked best for my data.

C. CNN Depth

I may have spent the most time on this aspect of the project. I started this project using the VGG-16 model directly, which did not produce great validation accuracy (~67%). One of the biggest problems with using the VGG-16 model is that it uses an image size of 224x224 pixels, but all the images in my dataset are a minimum of 256x256 pixels. So, my next step was to "reproduce" the VGG-16 model (i.e., create the same number of layers in the same order), but increase the image size to match my training data. This did not prove to be very effective either, which I believe is due to an excess of trainable features in comparison to the number of samples within my dataset and the number of classes the model is predicting. After this, I began pairing down the number of convolutional and FC layers until I found a combination that produced the highest test validation. Figure 11 is a graphical representation of my final CNN model, which one could call a "mock" VGG-8.

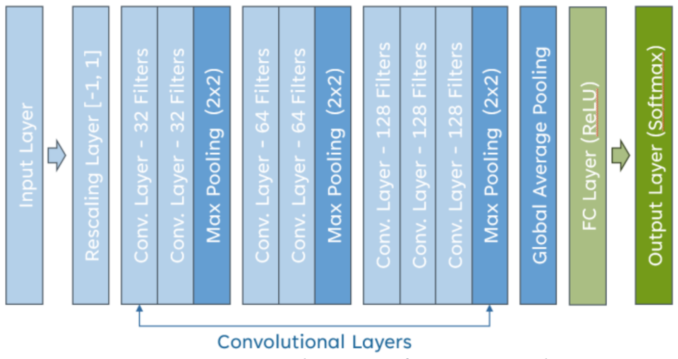


Figure 11—Custom CNN with Best Performance on Plant Dataset

D. Freezing CNN Layers

The VGG-16 model freezes many of its layers, which greatly reduces the number of trainable parameters the model must calculate. One reason for doing this is to reduce computational complexity and to help prevent overfitting. I experimented with this feature but ultimately decided to go with all the layers having trainable parameters. I found that a model with around 500,000 – 1,000,000 trainable parameters worked best—likely because my dataset was too small to benefit from more parameters.

E. Augmenting Trainig Data

I attempted to augment the training data by adding images that have been rotated or reflected, but my system could not handle the additional data. I could not even add a single translation to the training data without maxing out my GPU and system RAM. Given I am training a machine learning model, I am confident that adding training data would improve the validation accuracy. This is especially true since I am training a Convolutional Neural Network, which specializes in handling translations and rotations in the training data. However, unfortunately I was limited by my hardware to only use the original dataset.

F. Image Rescaling

I first rescaled the data from [0, 255] to [0,1] by dividing by 1/255, but I later found that rescaling to [-1,1] performed better. Rescaling to [-1,1] simply requires dividing by 1/127.5 and subtracting 1 to all the values. So, I decided to use the new scale of [-1,1] for all of models.

G. *Dropout Layers*

Dropout layers randomly drop a specified percentage of inputs before sending the input data to the next layer. Dropout layers are useful before and after fully-connected layers to prevent overfitting. I varied the dropout percentage from [0.1-0.5], with a step size of 0.1. I found that the dropout percentage of 0.5 worked best, both before and after the FC layer immediately prior to the output layer.

H. *Number of Filters per Convolutional Layer*

I tried different filter counts per convolutional layer, ranging from 16 filters to 512 filters. The VGG-16 model uses multiple stacks of 512-filter convolutional layers, but that is way too many filters for my relatively small training dataset. However, I found that starting with two 16-filter convolutional layers did not produce a validation score as high as starting with two 32-filter convolutional layers.

I. *Epoch Number and Steps per Epoch*

Ideally, I would have used 50 epochs for each iteration of the CNN model, but I preferred to use my time testing different fine-tuning parameters. Most of the time the model reached its highest score before the 50th epoch, but with the case of the Stratified K-Fold model, I likely left some performance on the table by cutting off the training at 20 epochs. I tried reducing the steps per epoch, but I found that leaving the number of steps at the default value, training samples / batch size, to work the best. I was willing to wait a bit longer for the additional steps to gain in validation accuracy.

J. *Imbalanced Data*

I found the data imbalance to be the greatest and most persistent issue with the model. As Figure 12 shows, the classes are heavily imbalanced toward the "Cassava___mosaic_disease" class. The CNN model was able to recognize all the Cassava leaf images as belonging to one of the Cassava leaf classes, but misidentified many Cassava classes as "Cassava___mosaic_disease."

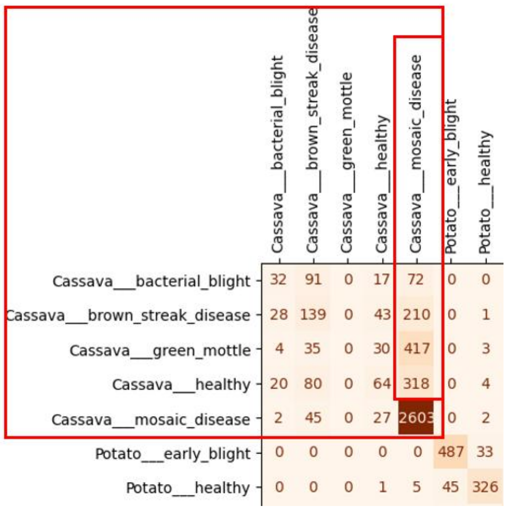


Figure 12—Confusion Matrix of Small Dataset

VI. Solving for Imbalanced Data

I tried two methods to solve for class imbalance:

1. Using image_dataset_from_directory() function to import the image data. Then, calculating the class weights using class_weight. compute_class_weight() and passing the class weights into the model.fit() function. This method uses the folder directory structure to determine the class labels, using the sub-directory names for each class label.
2. Using ImageDataGenerator() to import the image data. Then, I used Stratified K-Folds to create the training and test indices and process 5 folds of the data. I used a Medium article by Siladittya Manna as a reference guide for this method [17]. This method needs all the images to be under one directory with no sub-directories separating the classes. Instead, the class labels have been added to the name of each file. I wrote a .bash program to do restructure the image dataset.

The Stratified K-Fold method worked the best, producing the highest validation accuracy of any model I had made thus far. Each of the folds produced a validation accuracy close to 88%. Another issue with only using the weights to correct the data imbalance is that sometimes the classes with very few samples sometimes did not make it into the validation dataset, throwing off the accuracy. Therefore, if I were to continue working on the model, I would exclusively use the Stratified K-Fold method (despite the runtime increase).

VII. Future Improvements

At the risk of stating the obvious, one of the greatest limitations to this project was hardware—Particularly a shortage in RAM. This was so limiting because I could not add more translated images to the dataset. Increasing the training data set would almost certainly improve the performance of the model. Also, a larger dataset would give me more options to fine-tune the layers in the CNN model.

Another limitation of this project was not running GridSearchCV on the model to fine-tune various hyperparameters (e.g., batch size, learning rate, step size, epoch number). I did not do this more from a time-constraint perspective than due to hardware constraints. Setting up GridSearchCV and letting it run for however long it needed to run would likely improve the model by finding the best combination of hyperparameters for the CNN.

Another possibility to improve the model would be to use some sort of adaptive model that can take input images of varying sizes. While most of the image samples were 256 x 256 pixels, not all were the same size. Processing each image in its original size could increase the accuracy of the model.

I also noticed how sometimes misclassifications would occur when the CNN identified the correct disease but assigned it to the wrong plant leaf. I wonder if the model would perform better if the classes were just the diseases, and not the plant species *and* disease. I would like to test this out in the future to see if it would be better at classifying the disease, regardless of plant species. This could be practical as well, given that a plant disease can be treated with the same chemical regardless of which plant species it is infecting.

VIII. Conclusion

Overall, this was a very instructive project for me. I was able to learn more about AI-related hardware, how CNNs work, and how to process images. This was my first time working with a CNN and the first time using Deep Learning. If I were to do this project in the future, I would probably just pay the $50 for 500 computational credits and hope that was enough to get me through. One consistent challenge of this project was the fact that my desktop PC was constantly being used to train these models. I was constantly worried that it would crash or overheat, which is not something I considered before I started the project. One benefit of working with the cloud is the security server uptime and that someone else is maintaining the hardware for you. While I wish my model was able to produce better results, I am glad that I better understand how CNNs work and will be able to produce a better model in the future.

IX. References

[1] R. E. Gaunt, "The Relationship Between Plant Disease Severity and Yield," *Annual Review of Phytopathology,* vol. 33, no. 1, pp. 119-144, 1995.

[2] W. Aktar, D. Sengupta and A. Chowdhury, "Impact of Pesticides Use in agriculture: Their Benefits and Hazards," *Interdisciplinary Toxicology,* vol. 2, no. 1, pp. 1-12, 2009.

[3] A. Alnuaim, A. Altheneyan and A. A. AlZubi, "Early Leaf Disease Detection of Soybean Plants using Convolution Neural Network Algorithm," *LEGUME RESEARCH - AN INTERNATIONAL JOURNAL,* vol. 48, no. 6, 2025.

[4] S. J. Pethybridge and S. C. Nelson, "Leaf Doctor: A New Portable Application for Quantifying Plant Disease Severity," *Plant Disease,* vol. 99, no. 10, pp. 1310-1316, 2015.

[5] H. K. Gupta and R. S. Hare, "Potato Plant Disease Detection Using Convolution Neural Network," *NeuroQuantology,* vol. 20, no. 11, pp. 7244-7250, 2022.

[6] M. J. Karim, M. O. F. Goni, M. Nahiduzzaman, M. Ahsan, J. Haider and M. Kowalski, "Enhancing agriculture through real-time grape leaf disease classification via an edge device with a lightweight CNN architecture and Grad-CAM," *Scientific Reports,* vol. 14, no. 1, 2024.

[7] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," 2014.

[8] N. Sankalana, "Plant Leaf Diseases Training Dataset," Kaggle, [Online]. Available: https://www.kaggle.com/datasets/nirmalsankalana/plant-diseases-training-dataset/data. [Accessed 14 December 2025].

[9] C. M. Bishop, Pattern Recognition and Machine Learning, Springer, 2006, pp. 261-268.

[10] J. McDermott, "Hands-on Transfer Learning with Keras and the VGG16 Model," Learn Data Science, [Online]. Available: https://www.learndatasci.com/tutorials/hands-on-transfer-learning-keras/. [Accessed 14 December 2025].

[11] "VGG-16 - An Overview," ScienceDirect, [Online]. Available: https://www.sciencedirect.com/topics/computer-science/vgg-16. [Accessed 14 December 2025].

[12] T. N. Team, "Profile of tf-nightly," Pypi.org, 14 December 2025. [Online]. Available: https://pypi.org/user/tf-nightly/. [Accessed 14 December 2025].

[13] mypapit, "Tensorflow 2.20dev (Ubuntu 24.04) with AVX2," GitHub, 23 June 2025. [Online]. Available: https://github.com/mypapit/tensorflowRTX50/releases. [Accessed 14 December 2025].

[14] S. Paudel, "plant disease," Kaggle, 25 June 2025. [Online]. Available: https://www.kaggle.com/code/subaspaudel/plant-disease. [Accessed 14 December 2025].

[15] "Matrix Multiplication Background User's Guide," Nvidia, [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html. [Accessed 14 December 2025].

[16] Intel 64 and IA-32 Architectures Software Developer's Manual, Vols. 2 (2A, 2B, 2C, & 2D), 2024.

[17] S. Manna, "K-Fold Cross Validation for Deep Learning using Keras," Medium, 26 June 2020. [Online]. Available: https://medium.com/the-owl/k-fold-cross-validation-in-keras-3ec4a3a00538. [Accessed 14 December 2025].