DATA 607 Final Project

Classifying Plant Disease from Images Using ML and CNN

By: Radhika Joshi, Harold Lee, Alexandra Papparis, and Daniel Zhou

Introduction

Owning and caring for plants has become increasingly popular in recent years. Lots of applications have been created with the intent of properly identifying a plant. Websites like Pl@ntNet and phone apps like PictureThis are highly successful, allowing a user to input a picture of a plant and receive a series of suggestions as to what it is. Going further with this idea, the aim of this project is to investigate the health of a plant from a picture of its leaf. As they are living organisms, plants are susceptible to a variety of ailments. Anything from mold to wilting can be seen on a plant, and the ability to properly identify what disease is causing harm is the first step to treating it[1].

Convoluted neural networks are quickly becoming one of the leading image classification techniques[2]. One reason is the decrease in time needed to tune the high number of parameters needed when training multiple layers. They reduce dimensionality as well as the number of parameters without losing model quality. Compared to more traditional models, they can increase accuracy in a way that is ideal for computer vision.

Dataset

The dataset being explored is an open-source plant disease dataset found on Kaggle[3]. It contains a total of 1532 images (images vary in size) of plant leaves and is pre-split into Training, Validation, and Testing sets by the author. There are 3 classes for the images: Healthy, Powdered, or Rusted (see example images below). These classes refer to the conditions of the plant. A detailed branch of the structure of the dataset can be seen below.

- Dataset (1532 images)
    - Train (1322 images)
        - Healthy (458 images), Powdery (430 images), Rust (434 images)
    - Validation (60 images)
        - Healthy (20 images), Powdery (20 images), Rust (20 images)
    - Test (150 images)
        - Healthy (50 images), Powdery (50 images), Rust (50 images)

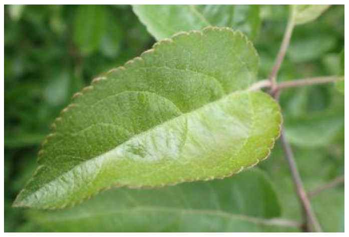


Figure 1 - example of ' Healthy'

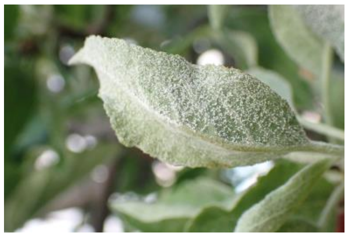


Figure 2 - example of ' Powdery'

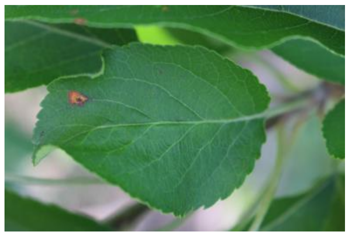
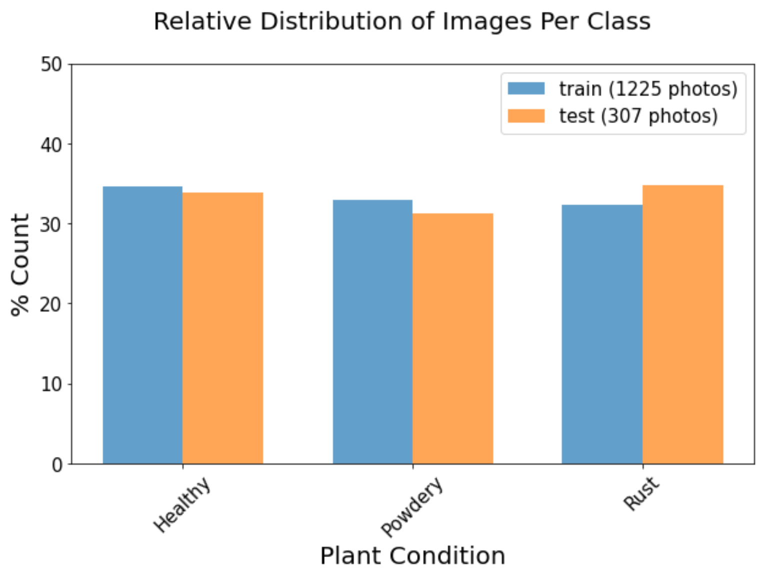


Figure 3 - example of ' Rust'

We decided that the validation set (consisting of 4% of total data) and testing set (consisting of 10% of total data) were too small for the purpose of our investigation, so we combined all the images together and then split 80/20 between training and testing set for all our models. Images were resized to 128x128 and the channels were normalized.

Distribution of Images between Training and Testing Sets



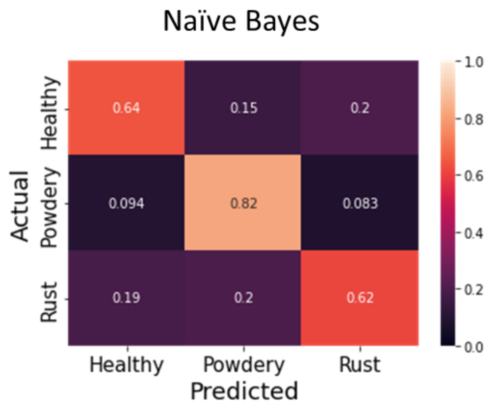
Traditional models

Preprocessing of Data for Applying traditional models from SKLearn

1. We split the data into 80% training and 20% test sets. The classes were largely balanced in the training and test sets.
2. We normalized the RGB image data using torchvision's transform method with the calculated mean and standard deviations for each channel from the training set. For our tree-based models, we created non-normalized training and test sets using the same random state.
3. For the tests with 5-fold cross-validation across models, we used SKlearn's StandardScaler for scaling the features in the non-normalized image data. This approach ensured that scaling was done based on the training data from each random split to avoid fitting the scaler on the test data.
4. We converted the torch tensors into NumPy arrays and flattened the 128 X 128 images into 2D arrays for fitting the traditional classification models.

Naïve Bayes Classifier

The model was fit to the normalized image data. It resulted in a prediction accuracy of 68 % with 5-fold Cross Validation.



Linear Support Vector Machine with Stochastic Gradient Descent Optimization

We initially fit the training data to the SGD classifier with default parameters including 'hinge' loss, which gives a linear SVM model. The default model resulted in a testing accuracy of 72 %.

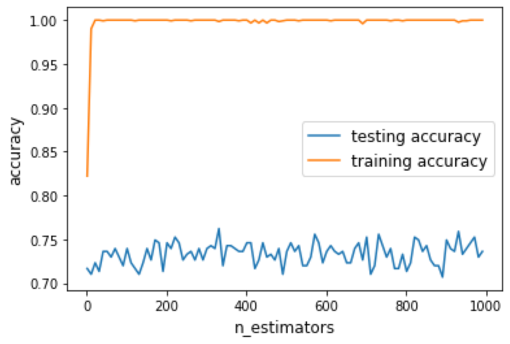
Next, we performed a grid search with 5-fold cross validation over the parameters:

```
params = {
    "loss": ["hinge", "log"],
    "alpha": [0.0001, 0.001],
    "penalty": ["l2", "l1", "elasticnet"],
}
```

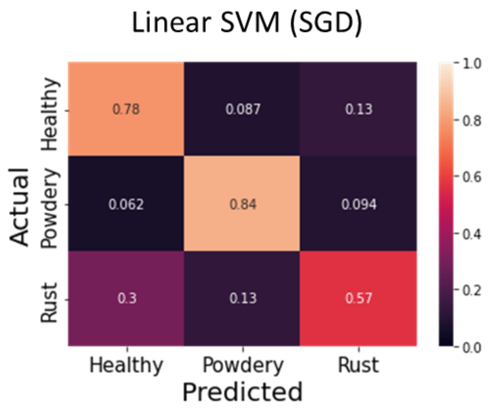
The best parameters selected by grid search were-

{'alpha': 0.0001, 'loss': 'hinge', 'penalty': 'l2'}.

We also tested the effect of the number of maximum iterations on the prediction accuracy.



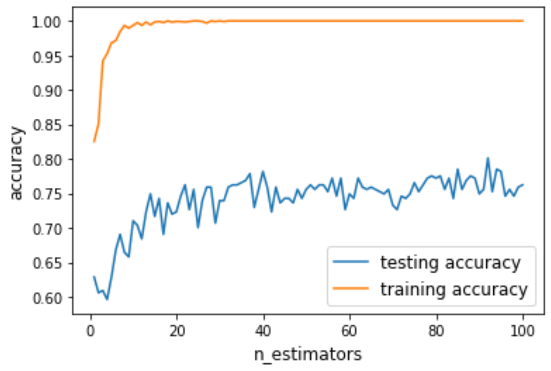
The graph shows that the accuracy didn't improve much beyond 200 iterations. Therefore, we decided to use 200 maximum iterations for convergence in our final SGD model.



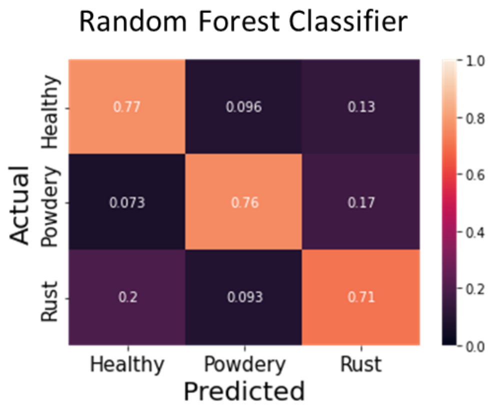
The optimized model resulted in a 73 % accuracy on the original train-test split and 70 % with 5-fold cross validation. The classification rate was lower for the 'Rust' class of plant diseases compared to the other two classes.

Random Forest Classifier

We fit this classifier to non-normalized RGB image data as normalization is not required for tree-based models. We used the default parameters for the model, only checking the effect of the number of estimators or trees on the prediction accuracy.



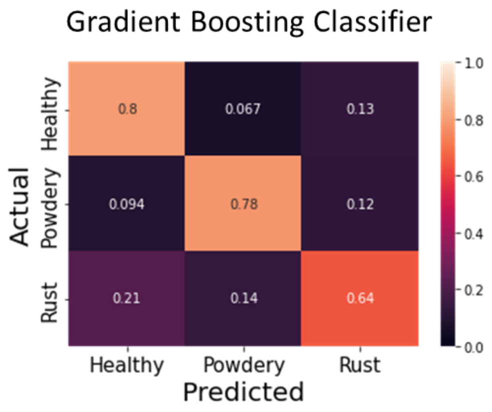
Based on the results from the figure, we proceeded with using 100 estimators in our model, which resulted in a 75% prediction accuracy with 5-fold cross validation. The model also performed better on classifying the 'Rust' images that the previous two models had performed poorly with.



Gradient Boosting Classifier

We fit this model to non-normalized data using the default parameters from SKlearn (n_estimators=100, learning_rate=1.0, max_depth=1). We did not tune any parameters for this model because convergence takes a very long time (SKlearn is not compatible with GPU).

With 100 estimators, the model resulted in a 74 % prediction accuracy on the original train-test split and 73 % with 5-fold cross-validation. The classification was worse for the 'Rust' class of images than the other classes.



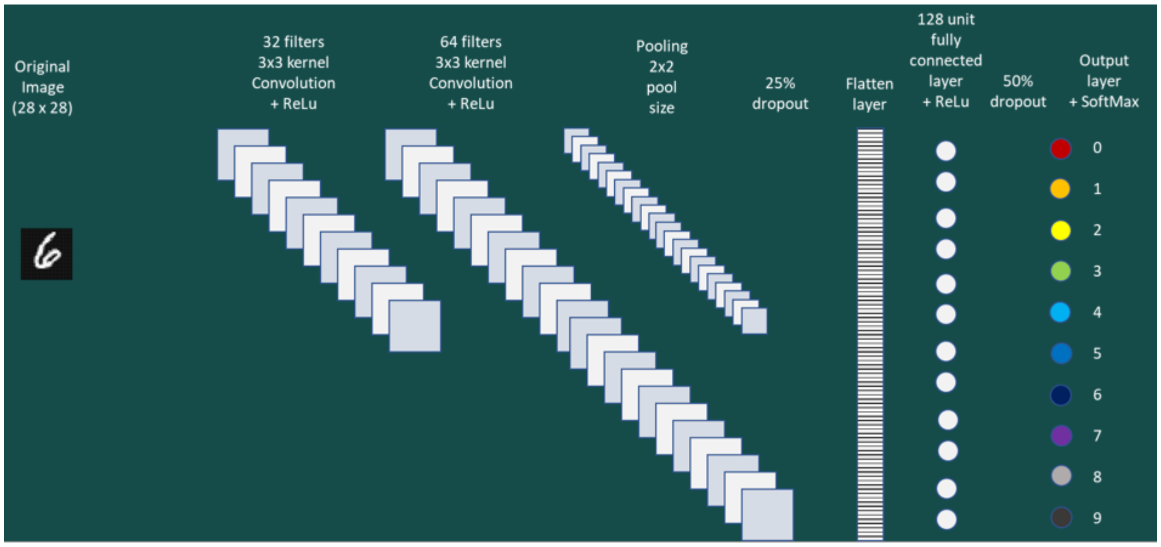
Traditional Model Summary

Model	Test accuracy	5-fold CV Accuracy
Naïve Bayes	69%	68%
Linear Support Vector Machine with Stochastic Gradient Descent	72%	73%
Random Forest Classifier (Without Feature Normalization)	75%	75% Number of Estimators: 100
Gradient Boosting Classifier (Without Feature Normalization)	74%	73% Number of Estimators: 100

CNN model built from scratch

Model building and selection

For our CNN model, we began by basing our own CNN model on an official PyTorch CNN example model on their GitHub page[4]. The example model was used for classifying the MNIST (handwritten digits) dataset and is structured as per model architecture diagram below.


Original Image (28 x 28)
32 filters 3x3 kernel Convolution + ReLu
64 filters 3x3 kernel Convolution + ReLu
Pooling 2x2 pool size
25% dropout
Flatten layer
128 unit fully connected layer + ReLu
50% dropout
Output layer + SoftMax
0
1
2
3
4
5
6
7
8
9

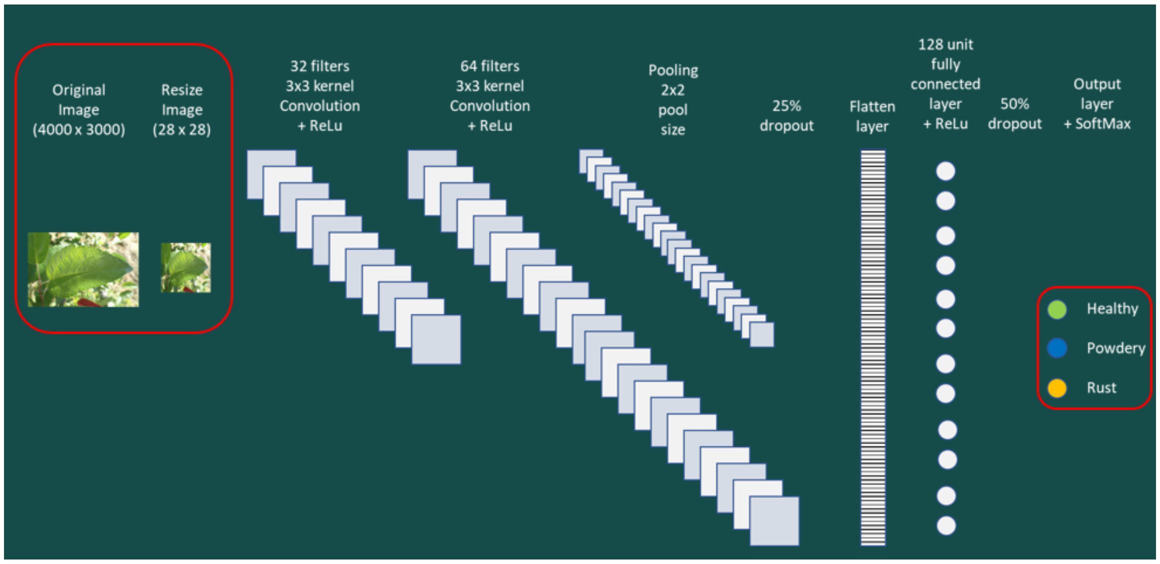
We can see that the model structure is:

- 2 x convolutional 2D layers with ReLu activation (3x3 kernal, with 32 and 64 filters each)
- 1 x pooling layer (with 2 x 2 pool size) to reduce the dimension of our feature maps
- 1 x 25 % dropout rate (to reduce overfitting)
- 1 x Flattening Layer
- 1 x Fully connected layer with ReLu activation (128 units)
- 1 x 50 % dropout rate (to reduce overfitting)
- 10 x output neurons with SoftMax activation (each representing 1 class)

Additionally, the model also had the following parameters:

- Training batch size of 64 images per batch
- Learning rate of 1
- Learning rate step gamma of 0.7 (this is the learning rate decay, if you are familiar with Tensorflow, it is sometimes referred as rho)
- PyTorch random seed set to 1
- The Adadelta optimizer was used. It is a stochastic gradient descent method that is based on adaptive learning rate per dimension to address the drawbacks of continual decay of learning rates throughout training and the need for a manually selected global learning rate. [5]
- The model initially had an epoch size of 14, which we decided to reduce to 10 to save time and to be consistent with our other models.

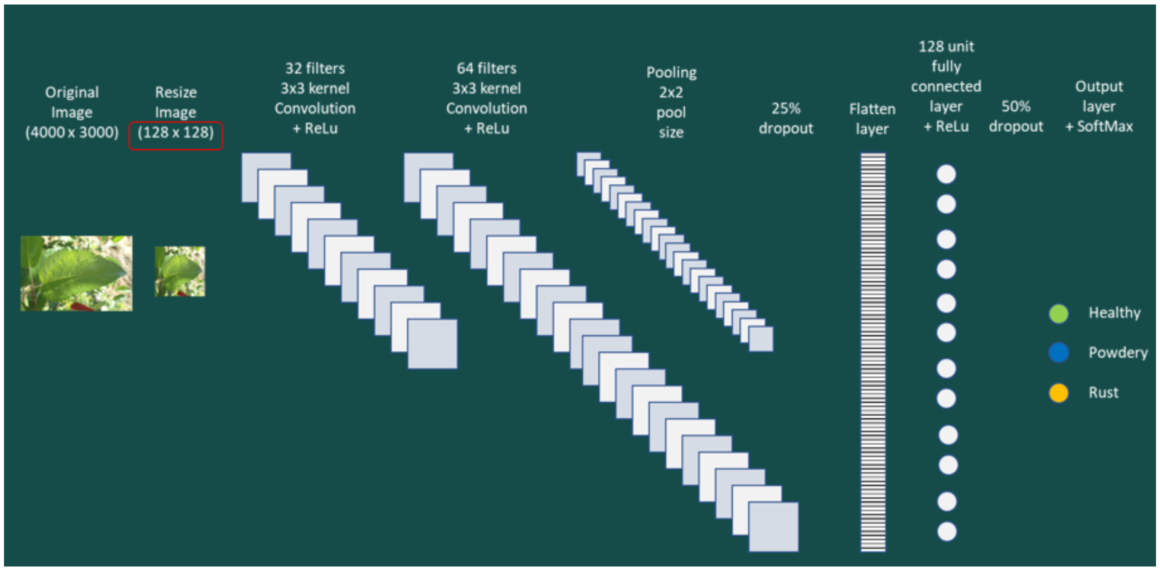
With this as the base model, we went ahead and adjusted the input and output layers to fit our problem, as shown in the model architecture diagram below. We decided for this first model we would resize our input image to the same size as the MNIST dataset's images (28 x 28). The resulting accuracies and losses of this model after 10 epochs are also shown below.


Original Image (4000 x 3000)
Resize Image (28 x 28)
32 filters 3x3 kernel Convolution + ReLu
64 filters 3x3 kernel Convolution + ReLu
Pooling 2x2 pool size
25% dropout
Flatten layer
128 unit fully connected layer + ReLu
50% dropout
Output layer + SoftMax
Healthy
Powdery
Rust

Model	Train acc	Test acc	Train loss	Test loss
28x28 base	85%	81%	0.4	0.51

As we can see, with this model, we ended up with a testing accuracy of 81%. This shows that, even without any modification, it comfortably surpassed our traditional models (max of 75%).

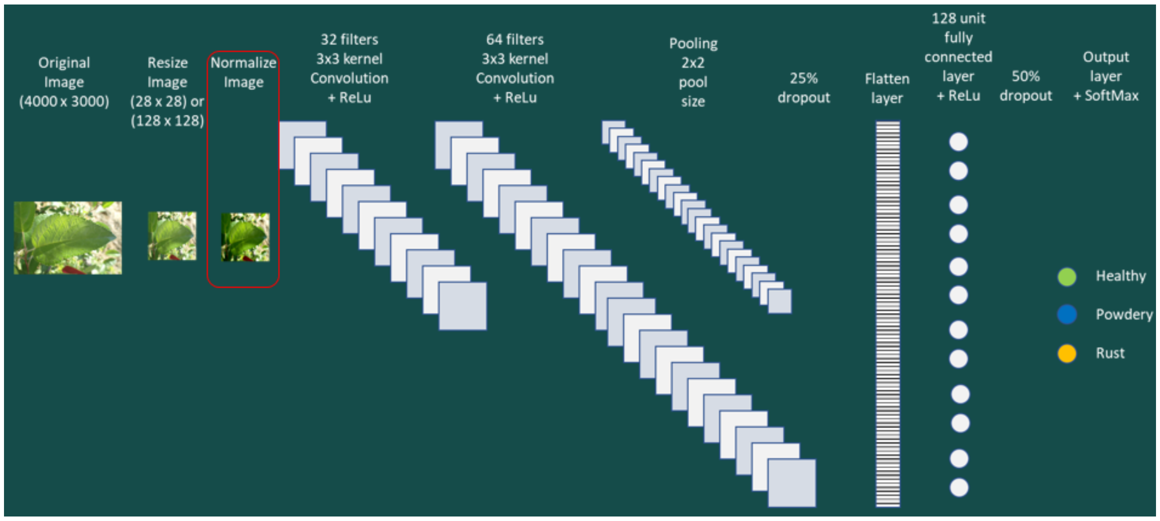
Next, we decided to tweak the input. Like our other models in the project, we decided to try an input size of 128 x 128 which is a standard image size for slightly more complicated images used in CNN classification. It also lets us match the input image sizes of our traditional models (and transfer learning model later) for a more appropriate comparison. Adjusted model architecture diagram is shown below. The resulting accuracies and losses of this model after 10 epochs are also shown below.


Original Image (4000 x 3000)
Resize Image (128 x 128)
32 filters 3x3 kernel Convolution + ReLu
64 filters 3x3 kernel Convolution + ReLu
Pooling 2x2 pool size
25% dropout
Flatten layer
128 unit fully connected layer + ReLu
50% dropout
Output layer + SoftMax
Healthy
Powdery
Rust

Model	Train acc	Test acc	Train loss	Test loss
128x128 base	88%	82%	0.32	0.5

With this model, we ended up with a testing accuracy of 82% after 10 epochs as shown in table above. This shows no significant difference compared to the 28 x 28 input sized model. However, based on the difference between the training and the testing loss, we can see that this model has signs of increasing overfitting. We will continue working with both input sizes as we try to improve our model.

Next, we would like to improve upon our accuracy. The next step we took is to add a normalization step for our input images. As learnt in class, normalization is important as it ensures that each input parameter (pixel, in this case) has a similar data distribution. Also makes convergence faster while training the network. Our adjusted model is as per model architecture diagram shown below. The resulting accuracies and losses of this model after 10 epochs are also shown below.

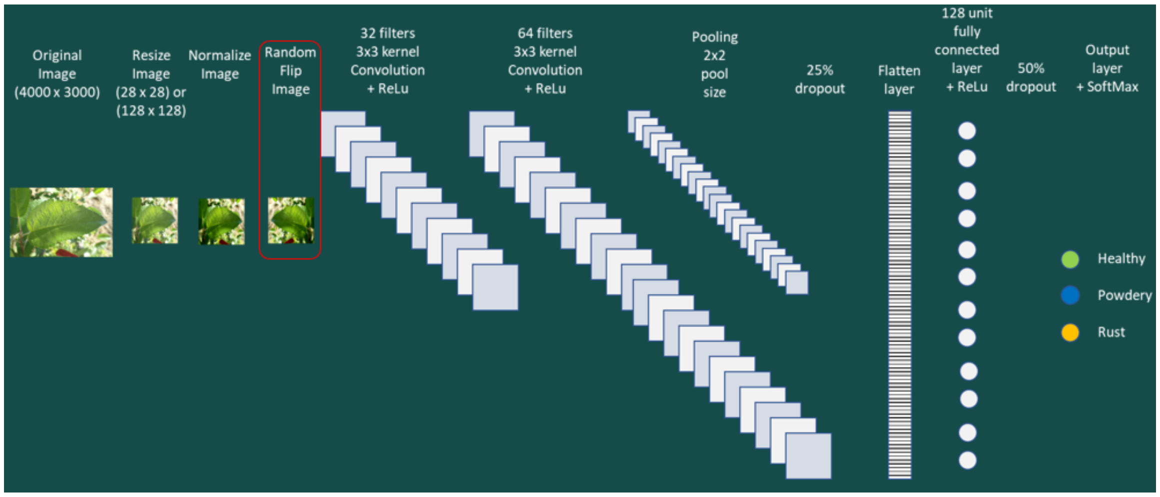

Original Image (4000 x 3000)
Resize Image (28 x 28) or (128 x 128)
Normalize Image
32 filters 3x3 kernel Convolution + ReLu
64 filters 3x3 kernel Convolution + ReLu
Pooling 2x2 pool size
25% dropout
Flatten layer
128 unit fully connected layer + ReLu
50% dropout
Output layer + SoftMax
Healthy
Powdery
Rust

Model	Train acc	Test acc	Train loss	Test loss
28x28 normalized	96%	85%	0.13	0.38
128x128 normalized	100%	90%	0.02	0.34

We see that the testing accuracy jumped higher for both models, especially the 128 x 128 model as per results table above. The convergence is faster, in fact by the 3rd epoch, both of our models surpassed the testing accuracy of their base models.

However, we can also see a big issue of overfitting. By comparing the training loss and testing loss we can see that there is a huge difference, especially with the 128 x 128 mode, where the training loss was 0.02 and testing loss was 0.34. This is a significant overfitting issue which we will have to address.

In order to address our overfitting issue, we use another technique learnt in class, random image flips. This lets us essentially increase the size of our training set by artificially generating more data. We proceeded to add a 50% random flip rate for training image into our models (both horizontally and vertically) as shown in the model architecture diagram below. The resulting accuracies and losses of this model after 10 epochs are also shown below.


Original Image (4000 x 3000)
Resize Image (28 x 28) or (128 x 128)
Normalize Image
Random Flip Image
32 filters 3x3 kernel Convolution + ReLu
64 filters 3x3 kernel Convolution + ReLu
Pooling 2x2 pool size
25% dropout
Flatten layer
128 unit fully connected layer + ReLu
50% dropout
Output layer + SoftMax
Healthy
Powdery
Rust

Model	Train acc	Test acc	Train loss	Test loss
28x28 random flips	90%	82%	0.26	0.37
128x128 random flips	96%	91%	0.13	0.29

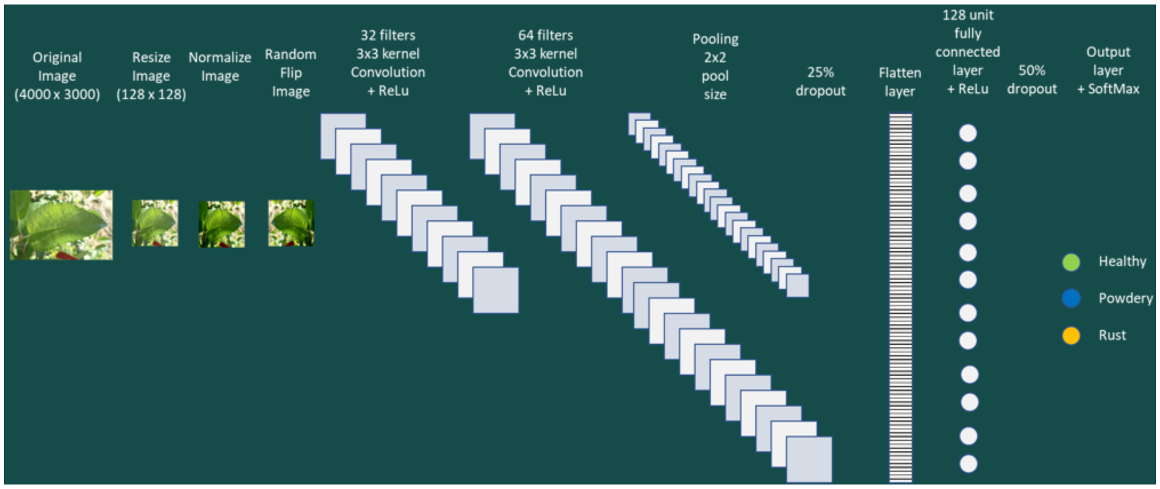
We can see comparable accuracies to the normalized model we had prior, especially with the 128 x 128 input sized model having a much smaller difference between the training loss and testing loss while maintaining accuracy. indicating less overfitting.

Below is a table summarizing and comparing the accuracies and losses of all the CNN models we've built thus far.

Model	Train acc	Test acc	Train loss	Test loss
28x28 base	85%	81%	0.4	0.51
28x28 normalized	96%	85%	0.13	0.38
28x28 random flips	90%	82%	0.26	0.37
128x128 base	88%	82%	0.32	0.5
128x128 normalized	100%	90%	0.02	0.34
128x128 random flips	96%	91%	0.13	0.29

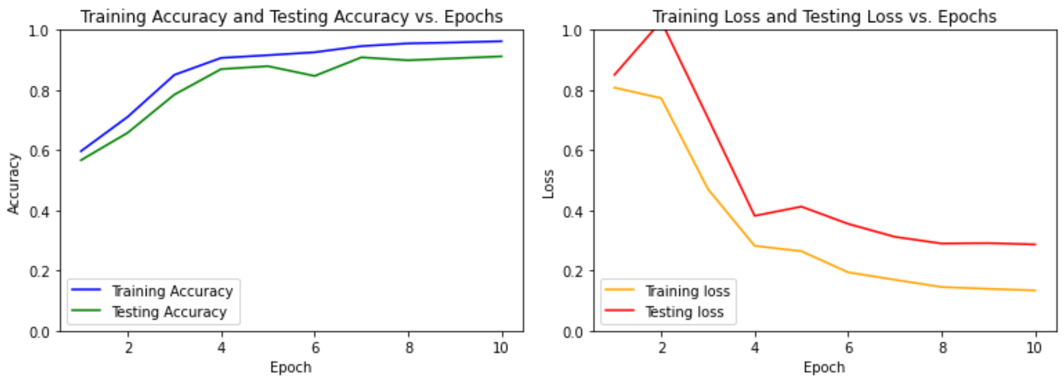
We will choose the latest 128 x 128 input sized model (with normalization and random flips) as our final model. This model had the highest accuracy (at 91%), and comparable training loss and testing loss difference as compared to the similar 28 x 28 input sized model.

Below is our final model architecture diagram.



Here is our final model's epoch vs accuracy / loss table as well as chart illustration.

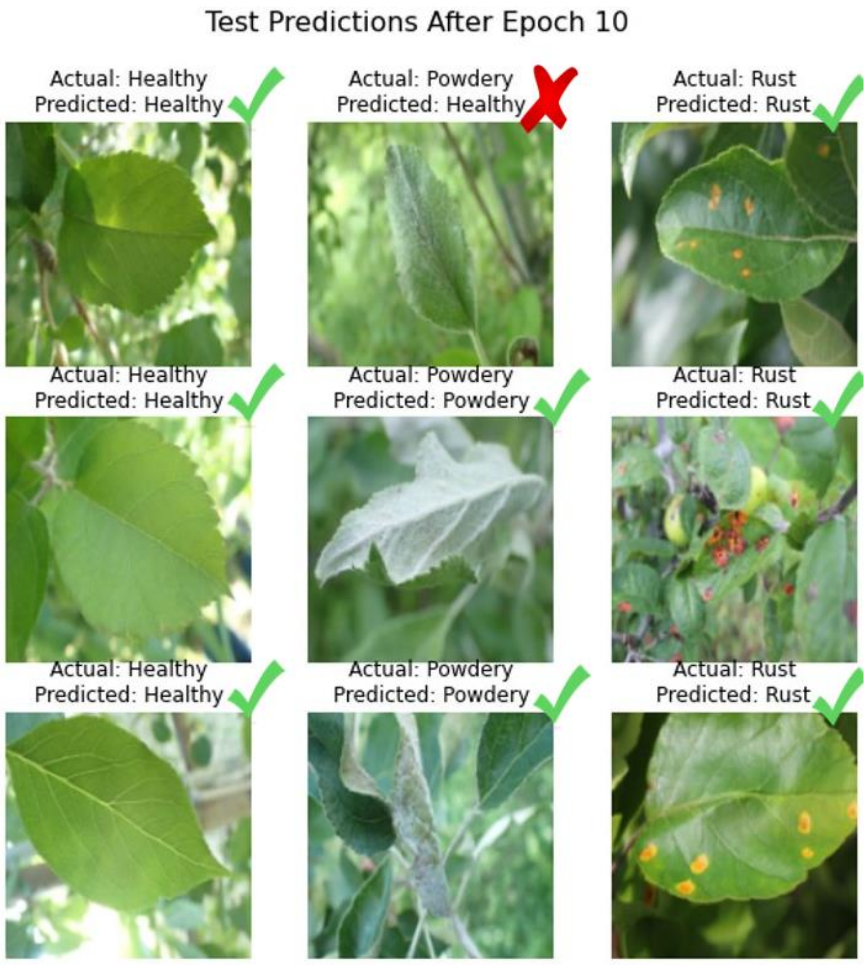
Epoch	Training accuracy	Testing accuracy	Training loss	Testing loss
1	59.67%	56.68%	0.8082	0.8506
2	71.10%	65.80%	0.7731	1.0289
3	85.06%	78.50%	0.4710	0.7072
4	90.69%	86.97%	0.2820	0.3818
5	91.59%	87.95%	0.2642	0.4125
6	92.57%	84.69%	0.1940	0.3551
7	94.61%	90.88%	0.1691	0.3120
8	95.51%	89.90%	0.1451	0.2898
9	95.84%	90.55%	0.1391	0.2911
10	96.24%	91.21%	0.1339	0.2867



We can see from the table and the charts that the accuracies and losses for our final model seem to stabilize by the 8th epoch.

Model Illustration

With this final model, we wanted to do a graphical illustration of the model. We randomly selected 9 samples (3 each for Healthy, Powdery, and Rust) from our testing set (where the model had a 91% accuracy) and passed them through our model. Results of the predictions are shown in the illustration below.



As we can see, our final model got 8 out of 9 predictions correct, so that’s in line with our 91% accuracy for the testing set from which these 9 pictures were selected from.

Transfer learning

Reznet50 was used in transfer learning with 10 epochs and a learning rate of 0.001. The Adam optimizer was used. The validation set was not used in the optimization of the model; hence it is essentially the test set. The result achieved a test accuracy of about 95% and is the highest yet.

Results:

```
Epoch 1: mean training loss = 0.7033, test loss = 0.3651, test accuracy = 0.9121
Epoch 2: mean training loss = 0.3387, test loss = 0.2892, test accuracy = 0.9088
Epoch 3: mean training loss = 0.2796, test loss = 0.2207, test accuracy = 0.9316
Epoch 4: mean training loss = 0.2723, test loss = 0.1877, test accuracy = 0.9544
Epoch 5: mean training loss = 0.2224, test loss = 0.1769, test accuracy = 0.9446
Epoch 6: mean training loss = 0.1877, test loss = 0.1866, test accuracy = 0.9381
Epoch 7: mean training loss = 0.1775, test loss = 0.1577, test accuracy = 0.9446
Epoch 8: mean training loss = 0.1933, test loss = 0.1615, test accuracy = 0.9577
Epoch 9: mean training loss = 0.1531, test loss = 0.1426, test accuracy = 0.9544
Epoch 10: mean training loss = 0.1696, test loss = 0.1443, test accuracy = 0.9544
```

With only 10 epochs, it is unclear if training and test loss has stabilized. Overfitting does not appear to be an issue as the train and test loss are comparable. The test accuracy plateaus from the 4th epoch.

Conclusion

Results

Model	Normalized channels	Random flipping	Test accuracy	Cross Validation
Naïve bayes	Y		68%	Y
SGD	Y		73%	Y
Random Forest Classifier	Y		75%	Y
Gradient Boosting Classifier	Y		73%	Y
Self-made CNN	Y	Y	91%	
Transfer learning Reznet50	Y	Y	95%	

The transfer learning model performed the best out of all models (95% test accuracy), but our self-made model performed remarkably well with just two convolution layers (91% test accuracy). The test accuracy of both models plateaued by about the 6-7th epoch. It may be possible for the mean training loss to stabilize at a lower value with more epochs in the transfer learning model.

The traditional Sklearn models performed about the same as each other. The CNN modes had a test accuracy of 15% to 20% higher than the non-neural network models. Our final self-made model had a test accuracy of 91%. The transfer learning model had about 95% test accuracy. We also saw that normalization gave a boost to the test accuracy of these CNN models.

Final thoughts and Learnings

Overall, this project was a rewarding experience. Building this CNN model from scratch and comparing it with other models was a valuable learning for us. We did find that computing power was a big limitation for us when training the CNN model as we used the free Google Colab IDE. We will look to invest in better setups for our future projects.

A big challenge we dealt with in our model building was reducing overfitting. Some things we can try in the future to further reduce overfitting on our models are:

1. reducing the number of filters at each layer
2. reducing the number of layers overall
3. use regularizer in our convolution layers to optimize our model by penalizing complexity

One thing we will have to do better in the future is our data splitting. For this model, we split our training set and testing set 80/20. However, the way we used our testing set during model building is akin to a validation set where we used its results for model selection. With a larger dataset in the future, we could investigate doing a 60/20/20 split between training/validation/testing as having another separate testing set would allow us to test our final model with a fresh set of data at the end.

Another thing that we learnt from this experience is that although we were able to tune some parameters, we did not fine tune other parameters such as:

- filter number,
- kernel size,
- pooling size,
- dropout rate,
- training batch size,
- learning rate,
- epochs,
- loss functions, etc.

We plan to try tuning those for our future projects.

The use of CNN to extract features and then XGBoost to classify is another interesting approach that could be tried out in the future. Finally, paying for Colab Pro may be a good investment, as we were constantly denied GPU access.

References

[1] 10 Common Plant Diseases (and How to Treat Them). Miller, L. Family Handyman. Retrieved from: https://www.familyhandyman.com/list/most-common-plant-diseases/

[2] Why are Convolutional Neural Networks good for image classification? Mishra, P. Medium. Retrieved from: https://medium.datadriveninvestor.com/why-are-convolutional-neural-networks-good-for-image-classification-146ec6e865e8#:~:text=diminished%20by%20CNNs.-,Network,above%20described%20abilities%20of%20CNNs.

[3] Plant disease recognition dataset. Kaggle. Retrieved from: https://www.kaggle.com/datasets/rashikrahmanpritom/plant-disease-recognition-dataset

[4] Pytorch CNN example. Retrieved from: https://github.com/pytorch/examples/blob/main/mnist/main.py

[5] ADADELTA: An Adaptive Learning Rate Method. Zeiler, M. D. (2012). Retrieved from: https://arxiv.org/abs/1212.5701

Plant Disease Classification. Kaggle. Retrieved from: https://www.kaggle.com/code/vad13irt/plant-disease-classification