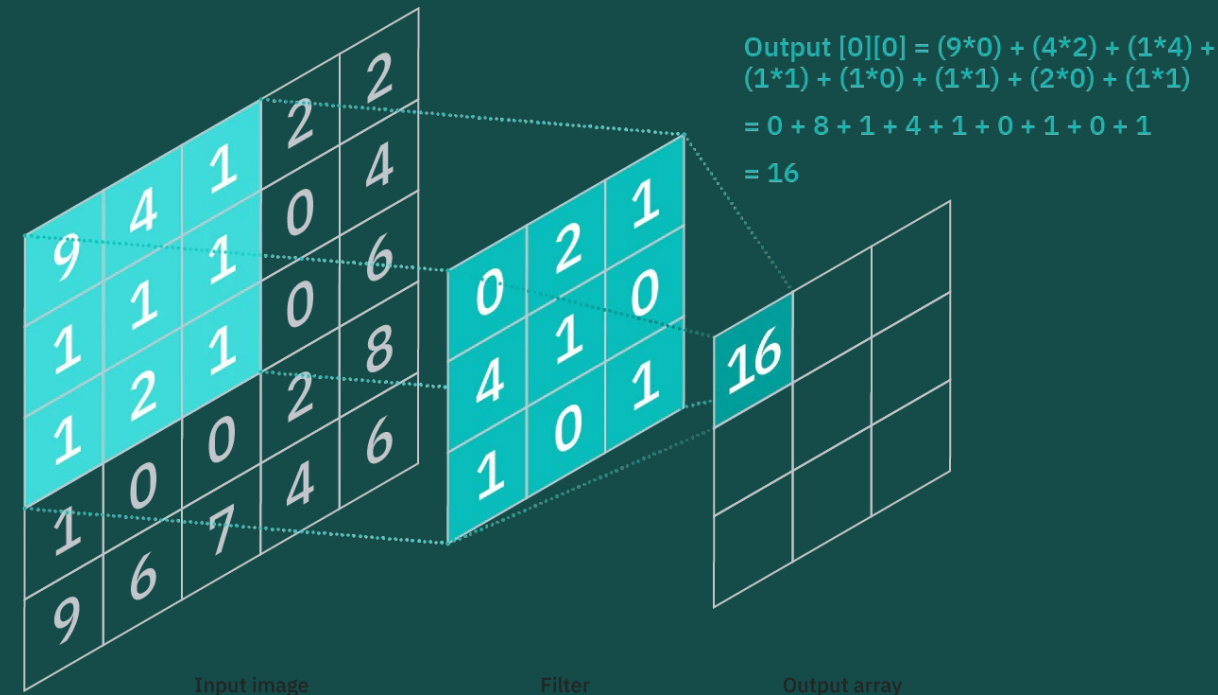# CLASSIFYING PLANT DISEASE IMAGES USING CONVOLUTED NEURAL NETWORKS

By Radhika Joshi, Harold Lee, Alex Papparis, and Daniel Zhou

INTRODUCTION

- Applications for plant identification are plentiful (e.g., PictureThis, Pl@ntNet)

- Working from modern interest in plants, provide enthusiasts with means to identify well-being of their plants.

- Large-scale image classification techniques, such as Convoluted Neural Networks, can be used to train more data and increase accuracy.
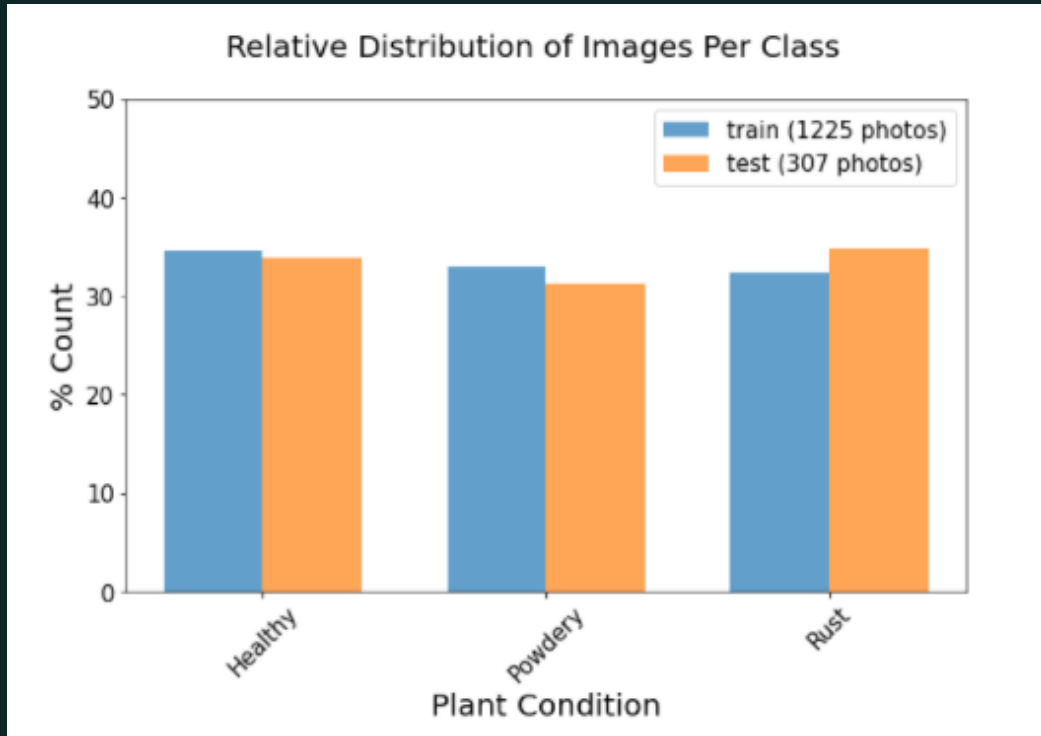
Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)

= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1

= 16

Input image   Filter   Output array

DATASET

- Open-source plant disease dataset (Kaggle)

- 1532 images of 3 classes: Healthy, Rusted, or Powdery (left, top to bottom)

- Pre-split into training, testing, and validation sets, but combined for our analysis

MODELS


Relative Distribution of Images Per Class

- Data was split into 80:20 training and testing sets

- Images resized to 128x128

- Channels normalized

- Classified using:

    - Traditional models (Naïve Bayes, Stochastic Gradient Descent, Random Forest Classifier, and Gradient Boosting Classifier)

    - Convoluted Neural Networks (Self-made and with transfer learning and image flipping)

TRADITIONAL MODELS

Naïve Bayes Classifier
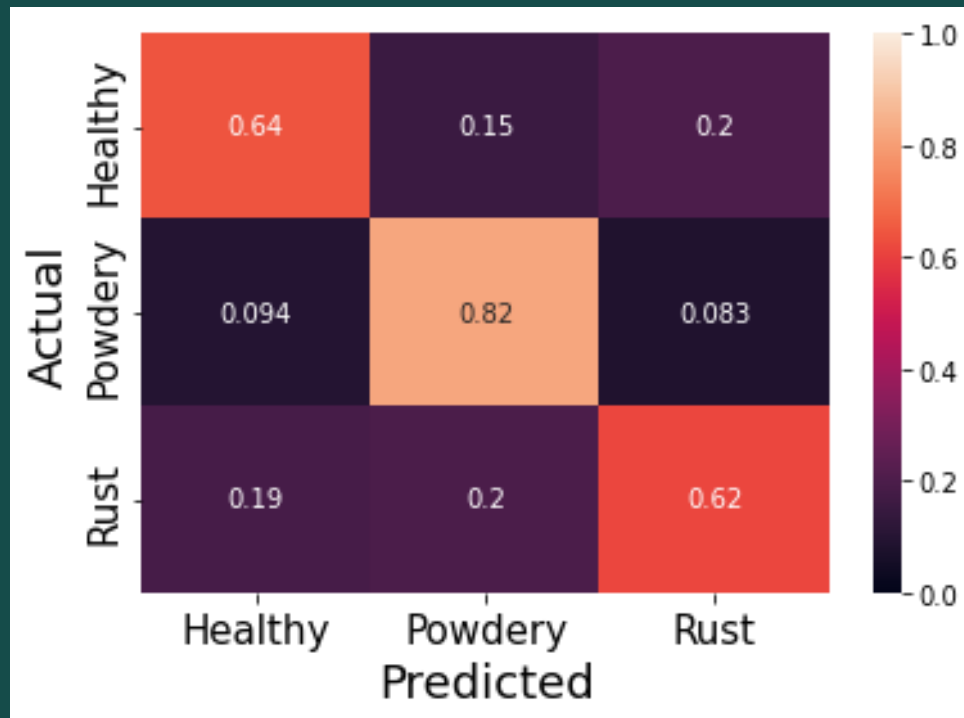
Testing Accuracy = 69 %

5-fold Cross Validation with Standard Scaler
Testing Accuracy = 68 %

TRADITIONAL MODELS

Linear Support Vector Machine with Stochastic Gradient Descent
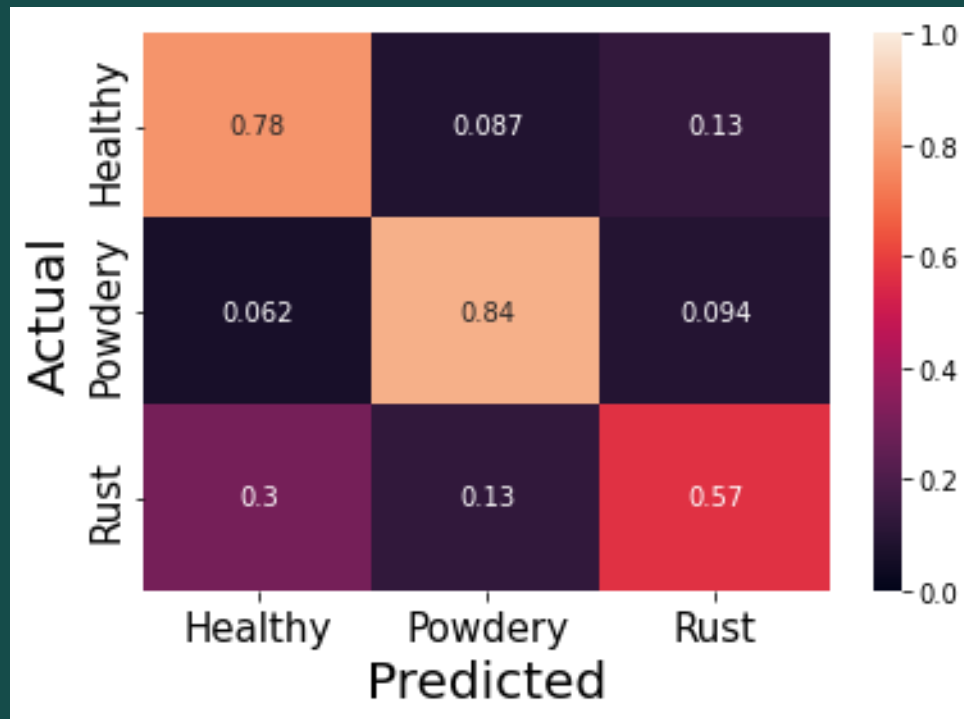
Grid Search for Hyperparameters:
- alpha: 0.0001
- loss: hinge
- Penalty: l2
- Max iterations = 200

Testing Accuracy = 73 %

5-fold Cross Validation with Standard Scaler
Testing Accuracy = 70 %

TRADITIONAL MODELS

Random Forest Classifier Without Feature Scaling
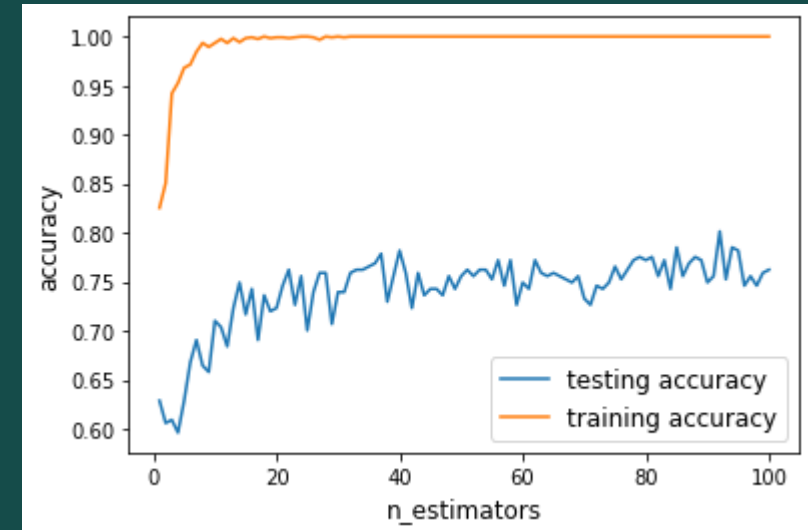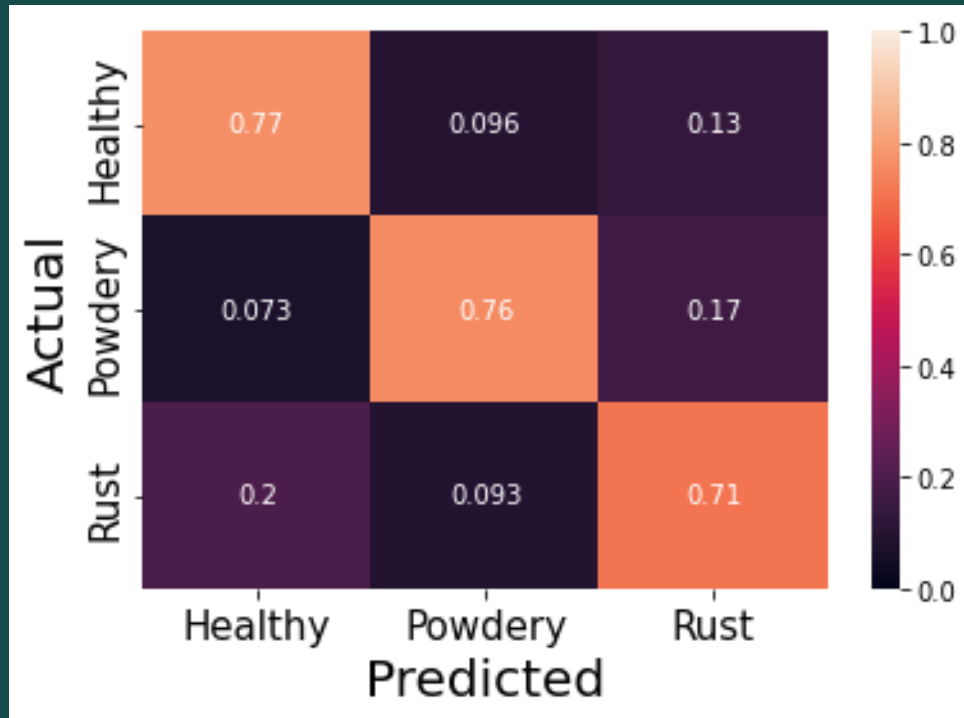
With 100 estimators:
Testing Accuracy = 75 %

5-fold Cross Validation:
Testing Accuracy = 75 %

TRADITIONAL MODELS

Gradient Boosting Classifier Without Feature Scaling

With 100 estimators:
Testing Accuracy = 74 %

5-fold Cross Validation:
Testing Accuracy = 72 %

CNN Model Building - base

- We began by basing our own CNN model on an official PyTorch CNN example model on their Github page [Github, 2020]. The example model was used for classifying the MNIST (handwritten digits) dataset. And is structured as per below. Our model training will run for 10 epochs.
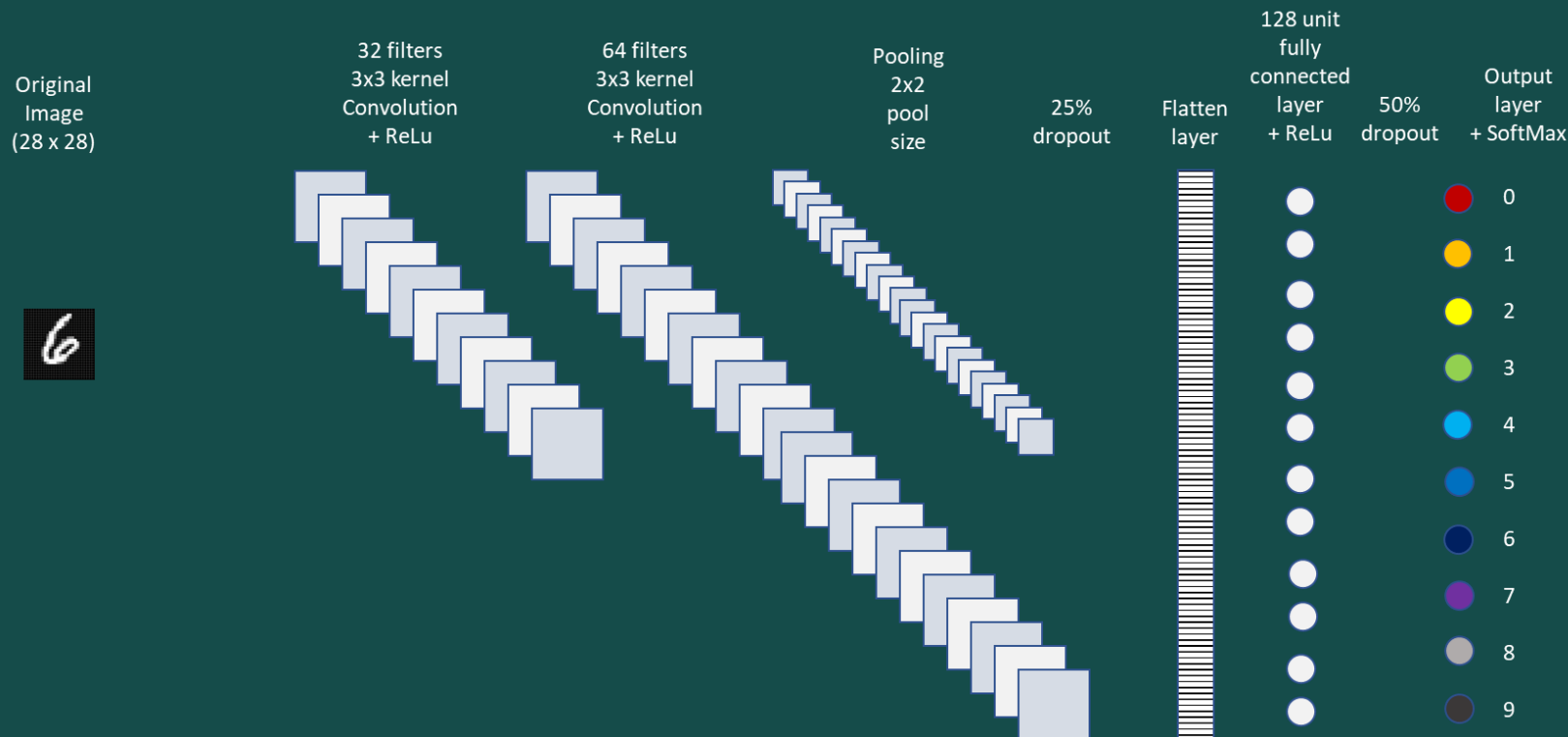

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```
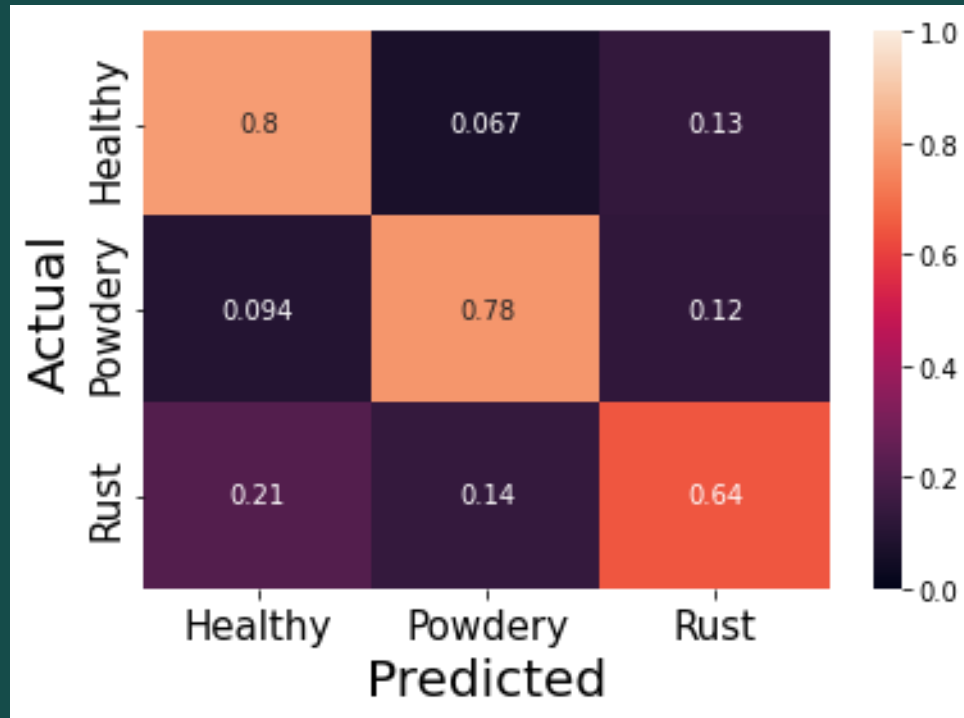
CNN Model Building - initial

- First, we wanted to run the model as is, so we had to resize our images as our images are much higher resolution (generally each image was around 12 megapixels, roughly 4000 x 3000) whereas the MNIST dataset had a resolution of 28 x 28. We will resize down to 28x28 before inputting the image for training.

	Train acc	Test acc	Train loss	Test loss
28x28 base	85%	81%	0.40	0.51

- We see that our base model testing accuracy has comfortably surpassed our traditional models (max of 75%) without any modification.
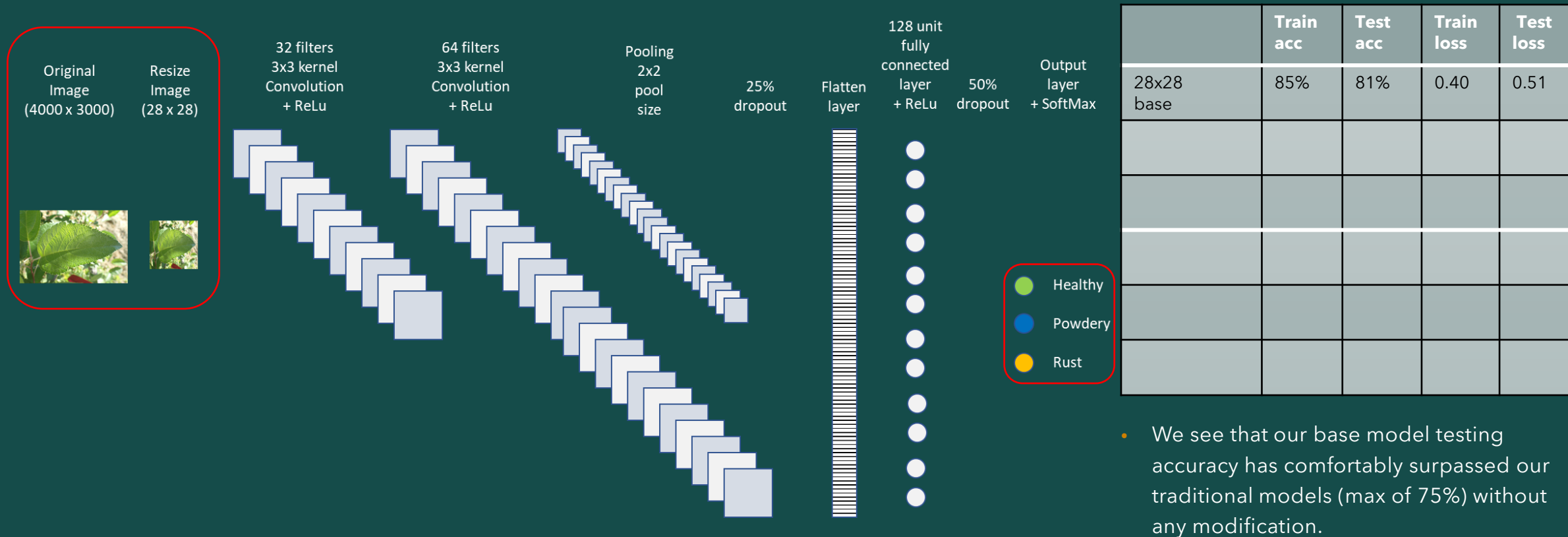
CNN Model Building – resize image

- Next we try using a larger image size as input for training, we try 128x128, which is a pretty standard image size for 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 on) for a more appropriate comparison.

	Train acc	Test acc	Train loss	Test loss
28x28 base	85%	81%	0.40	0.51
128x128 base	88%	82%	0.32	0.50

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

- No significant difference observed yet.

- Signs of more overfitting with the 128 x 128 model.

CNN Model Building – normalization

- Because we did not see a significant difference between our model with the 28x28 input image and our model with the 128x128 input image size. We will continue evaluating both input sizes.

- Next, we will 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.
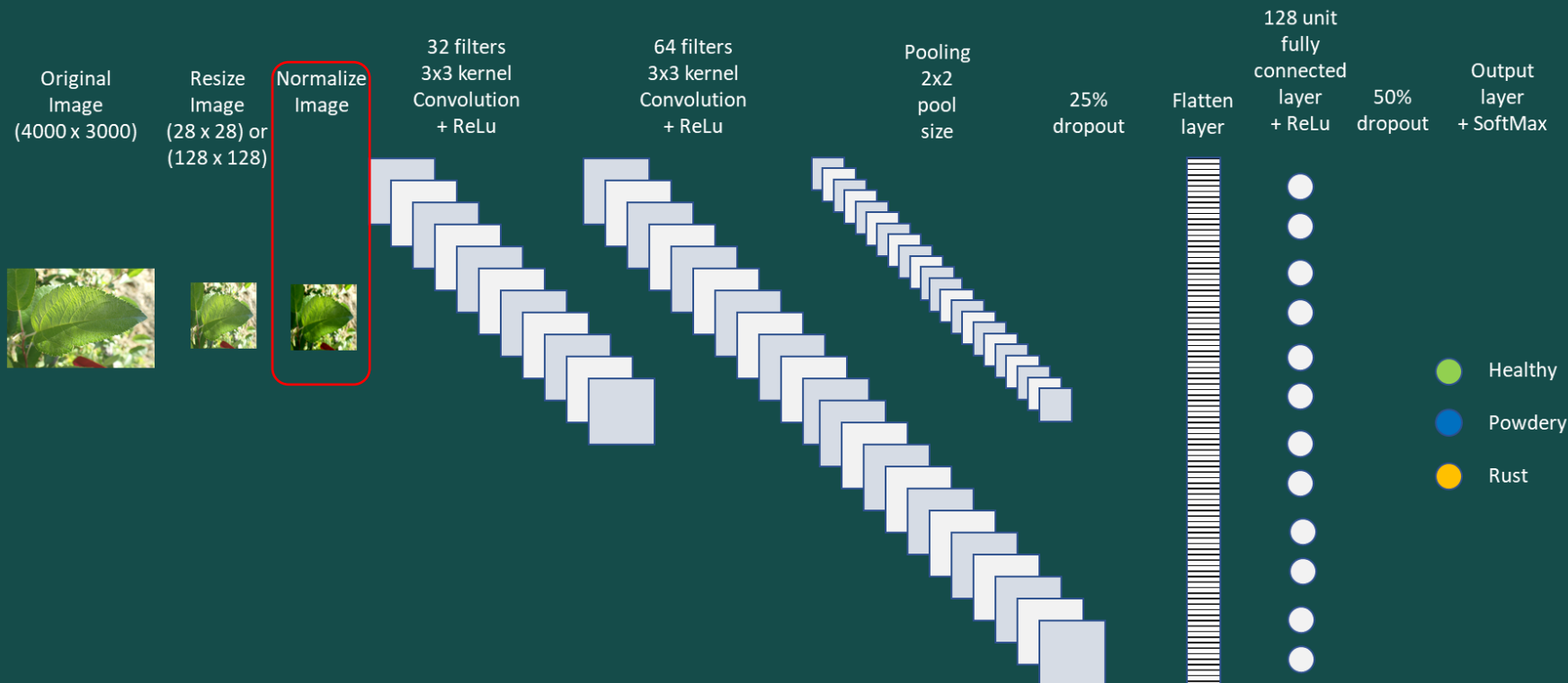
	Train acc	Test acc	Train loss	Test loss
28x28 base	85%	81%	0.40	0.51
28x28 normalized	96%	85%	0.13	0.38
128x128 base	88%	82%	0.32	0.50
128x128 normalized	100%	90%	0.02	0.34

- as expected, the convergence is faster. In fact by the 3rd epoch, both of our models surpassed the testing accuracy of their base models

- however, the comparison between training loss and testing loss tells us there is significant overfitting, that is especially obvious in the model with 128 x 128 input image.

CNN Model Building – random image flips

- Lastly, 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 add a 50% random flip rate for training image into our model (both horizontally and vertically).

	Train acc	Test acc	Train loss	Test loss
28x28 base	85%	81%	0.40	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	0.50
128x128 normalized	100%	90%	0.02	0.34
128x128 random flips	96%	91%	0.13	0.29

- We see comparable accuracies to the normalized model we had prior, with the 128 x 128 model especially having a much smaller difference between the training loss and testing loss while maintaining accuracy. Indicating less overfitting. We choose that model as our final model.

CNN Model Building – final model

- Our final model is as per below.

- Using our final model, we also did some sample testing drawn from the test set.

Training Accuracy and Testing Accuracy vs. Epochs

Training Loss and Testing Loss vs. Epochs
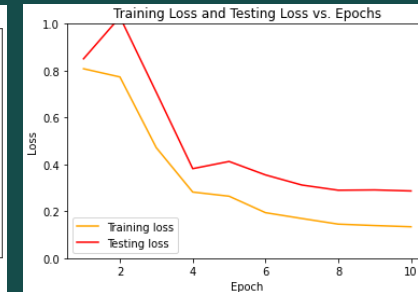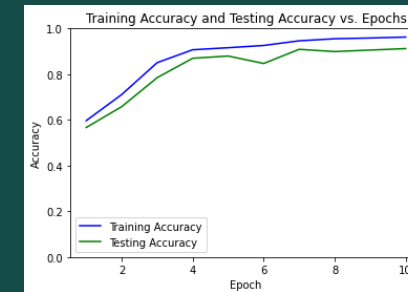
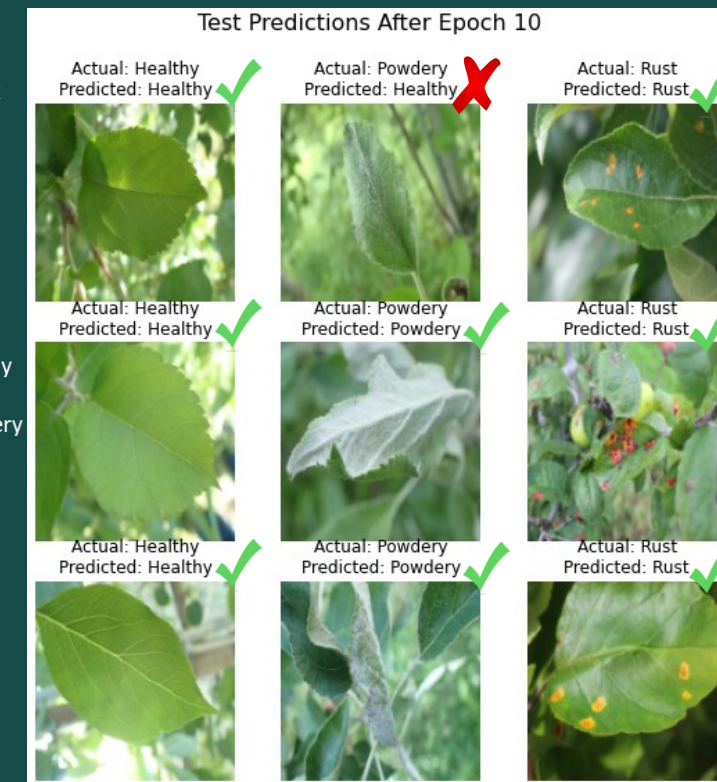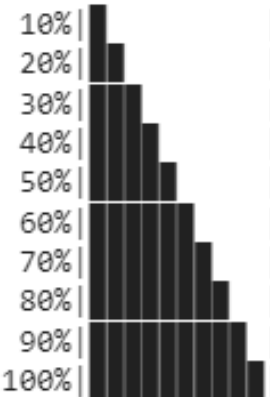|	Train acc	Test acc	Train loss	Test loss
Final Model	96%	91%	0.13	0.29

Original Image (4000 x 3000)
Resize Image (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

Test Predictions After Epoch 10

Actual: Healthy Predicted: Healthy ✓
Actual: Powdery Predicted: Healthy ✗
Actual: Rust Predicted: Rust ✓

Actual: Healthy Predicted: Healthy ✓
Actual: Powdery Predicted: Powdery ✓
Actual: Rust Predicted: Rust ✓

Actual: Healthy Predicted: Healthy ✓
Actual: Powdery Predicted: Powdery ✓
Actual: Rust Predicted: Rust ✓

CNN Model Building – learnings

- This was a valuable learning + practice for us to build a CNN model from scratch.

- We were able to tune some parameters but there are still lots of parameters we can try tuning for our future projects such as:

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

- The biggest 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:

  - Tuning and reducing the number of filters at each layer

  - Tuning and reducing the number of layers overall

  - use Regularizer in our convolution layers to optimize our model by penalizing complexity

- Having another separate testing set would allow us to separately test our final model again, the way we used our testing set during model building is akin to a validation set where we used its results for model selection, perhaps with a larger dataset in the future, we could look into a 60/20/20 split.

- Getting a personal rig may be a good investment, as we were constantly getting GPU banned on Google Colab...

- We'll next check out a transfer learning model to see how it does compared to building our own CNN.

CNN Transfer Learning Model (Reznet50)

```
 10%|█        | 1/10 [03:44<33:38, 224.29s/it]Epoch 1: mean training loss = 0.7033, test loss = 0.3651, test accuracy = 0.9121
 20%|██       | 2/10 [07:27<29:50, 223.84s/it]Epoch 2: mean training loss = 0.3387, test loss = 0.2892, test accuracy = 0.9088
 30%|███      | 3/10 [11:08<25:57, 222.44s/it]Epoch 3: mean training loss = 0.2796, test loss = 0.2207, test accuracy = 0.9316
 40%|████     | 4/10 [14:49<22:11, 221.87s/it]Epoch 4: mean training loss = 0.2723, test loss = 0.1877, test accuracy = 0.9544
 50%|█████    | 5/10 [18:33<18:32, 222.53s/it]Epoch 5: mean training loss = 0.2224, test loss = 0.1769, test accuracy = 0.9446
 60%|██████   | 6/10 [22:14<14:47, 221.96s/it]Epoch 6: mean training loss = 0.1877, test loss = 0.1866, test accuracy = 0.9381
 70%|███████  | 7/10 [25:53<11:03, 221.26s/it]Epoch 7: mean training loss = 0.1775, test loss = 0.1577, test accuracy = 0.9446
 80%|████████ | 8/10 [29:36<07:23, 221.65s/it]Epoch 8: mean training loss = 0.1933, test loss = 0.1615, test accuracy = 0.9577
 90%|█████████| 9/10 [33:22<03:42, 222.92s/it]Epoch 9: mean training loss = 0.1531, test loss = 0.1426, test accuracy = 0.9544
100%|██████████| 10/10 [37:04<00:00, 222.43s/it]Epoch 10: mean training loss = 0.1696, test loss = 0.1443, test accuracy = 0.9544
```
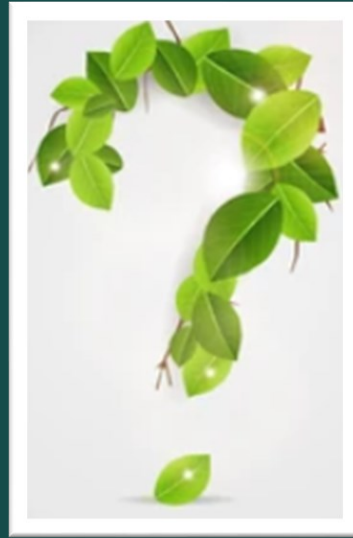
```python
class Model(torch.nn.Module):
    def __init__(self, feature_extractor, n_features):
        super().__init__()
        self.L = torch.nn.Linear(n_features, 3)
        self.fe=feature_extractor
        self.fe.requires_grad_(False)

    def forward(self, x):
        x = normalise(x)
        if self.training:
            x = random_horizontal_flip(x)
            x = random_vertical_flip(x)
        x = self.fe(x).squeeze()
        x = self.L(x).squeeze()
        return x
```

Conclusion

Model	Normalized channels	Random flipping	Test accuracy	Cross Validation
Naïve bayes	Y		68%	Y
Linear SVM with SGD	Y		70%	Y
Random Forest Classifier	Y		75%	Y
Gradient Boosting Classifier	Y		74%	Y
Self-made CNN	Y	Y	91%	
Transfer learning Reznet50	Y	Y	95%	

Thank you!

References

- https://www.kaggle.com/datasets/rashikrahmanpritom/plant-disease-recognition-dataset

- https://www.familyhandyman.com/list/most-common-plant-diseases/

- https://github.com/pytorch/examples/blob/main/mnist/main.py

- https://www.kaggle.com/code/vad13irt/plant-disease-classification

- https://www.sciencedirect.com/science/article/pii/S1738573319308587