

CLASSIFYING PLANT DISEASE IMAGES USING CONVOLUTED NEURAL NETWORKS

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



INTRODUCTION

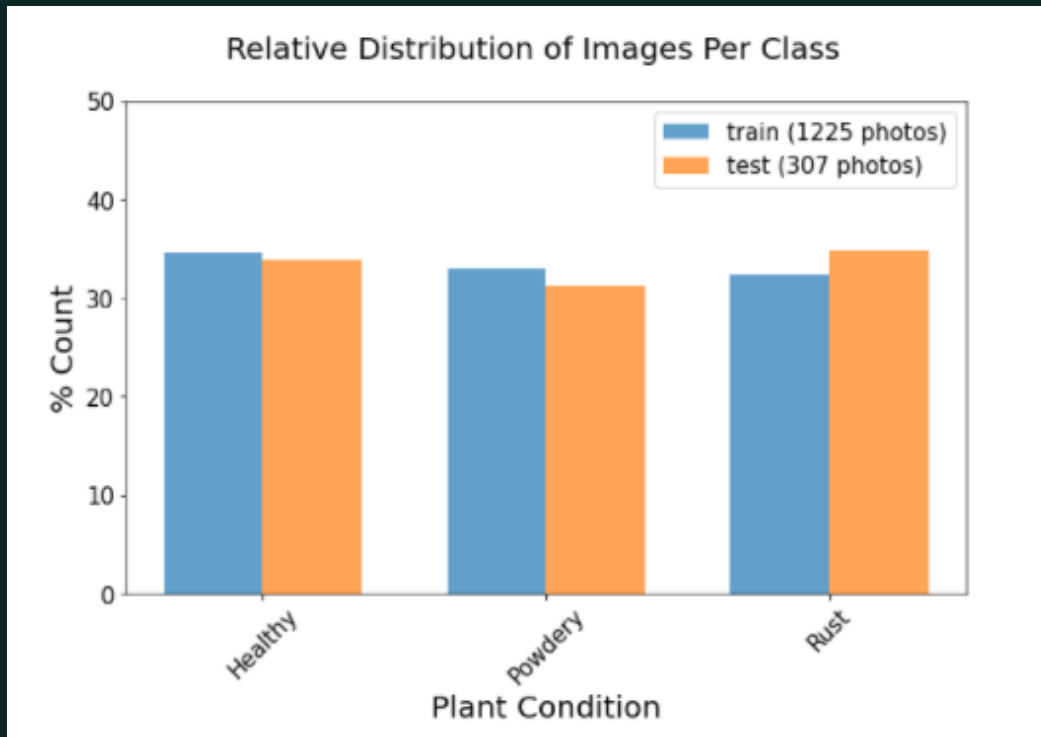
- Applications for plant identification are plentiful (egs)
- Working from modern interest in plants, provide enthusiasts with means to identify well-being of their plants.

The image is a vertical stack of three close-up photographs of leaves. The top leaf is healthy and green. The middle leaf shows signs of rust, with a small brown spot. The bottom leaf is covered in a thick layer of white powdery mildew. The right side of the image is a dark green gradient with a white semi-circle on the left edge.

DATASET

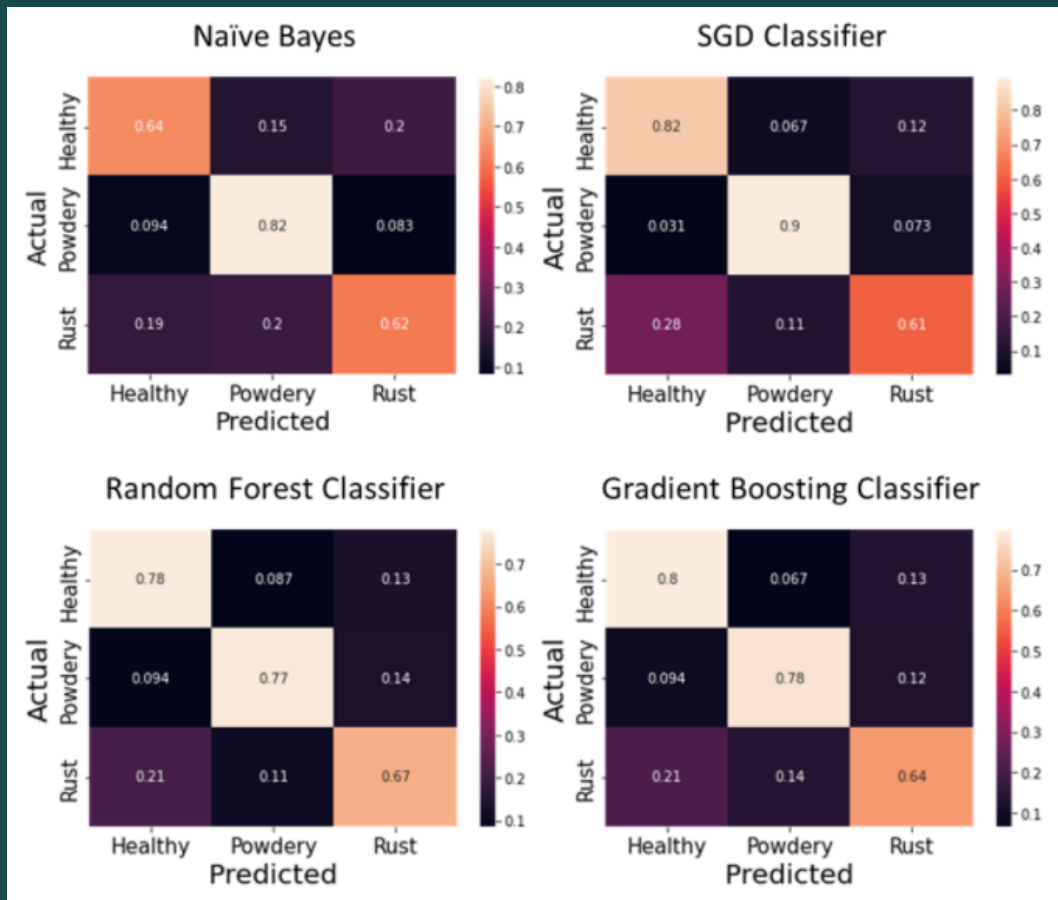
- Open-source plant disease dataset (Kaggle)
- 1532 images of 3 classes: Healthy, Rusted, or Powdery (left, top to bottom)
- Pre-split into training, testing, and validation sets

MODELS



- 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)
 - Image flipping

TRADITIONAL MODELS

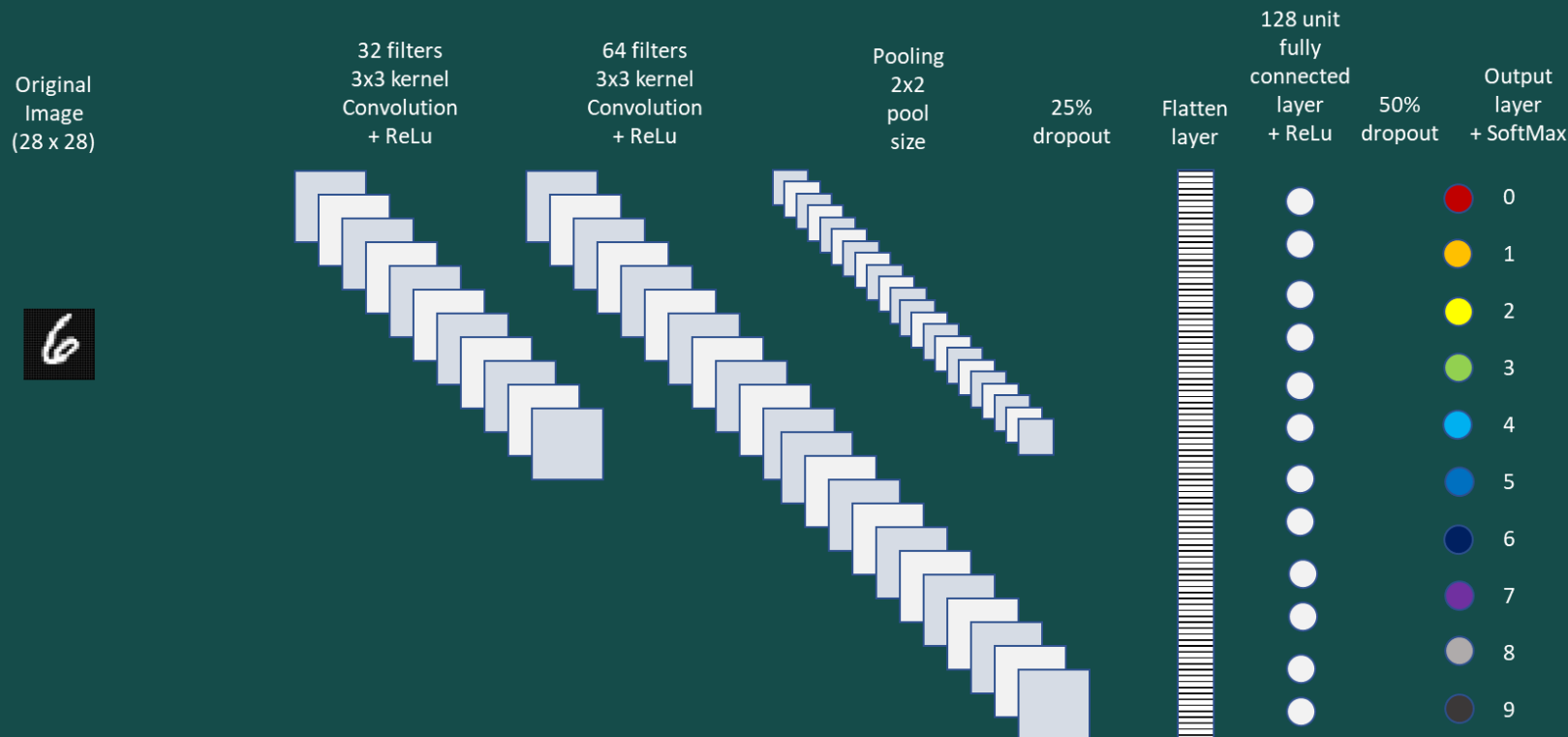


- Naïve Bayes, Stochastic Gradient Descent, Random Forest Classifier, and Gradient Boosting Classifier were all applied to the dataset
- Accuracies and Confusion matrices for each model were computed (shown below and left)

Model	Test accuracy	5-fold CV Accuracy
Naïve Bayes	69%	72%
Stochastic Gradient Descent (Linear Support Vector Machine) Optimized with <u>GridSearch</u>	72%	74%
Random Forest Classifier	74%	75%
Gradient Boosting Classifier	74%	Results Pending

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.

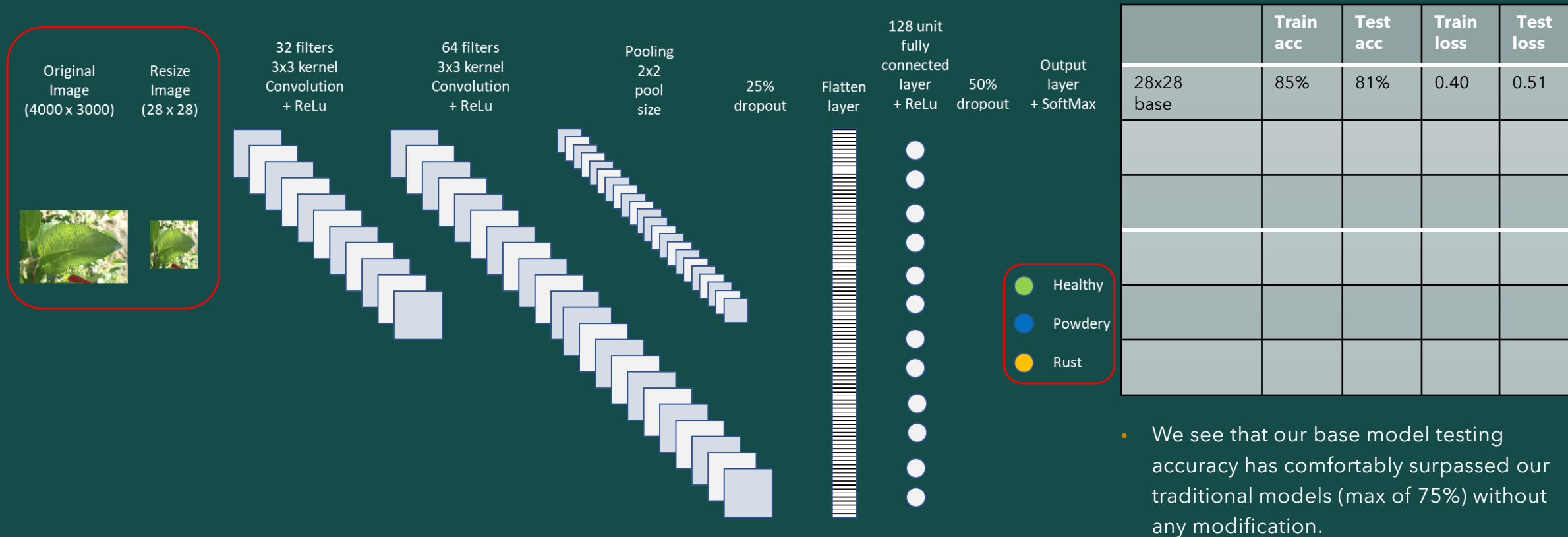


```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
    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

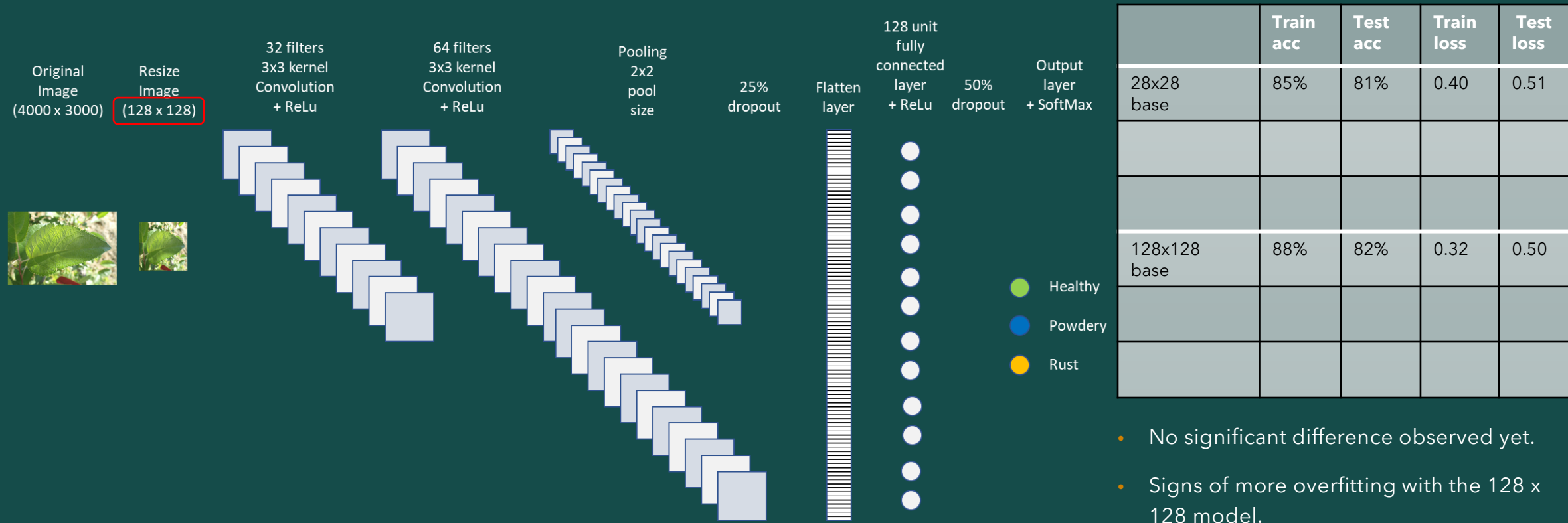
CNN Model Building - initial

- 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.



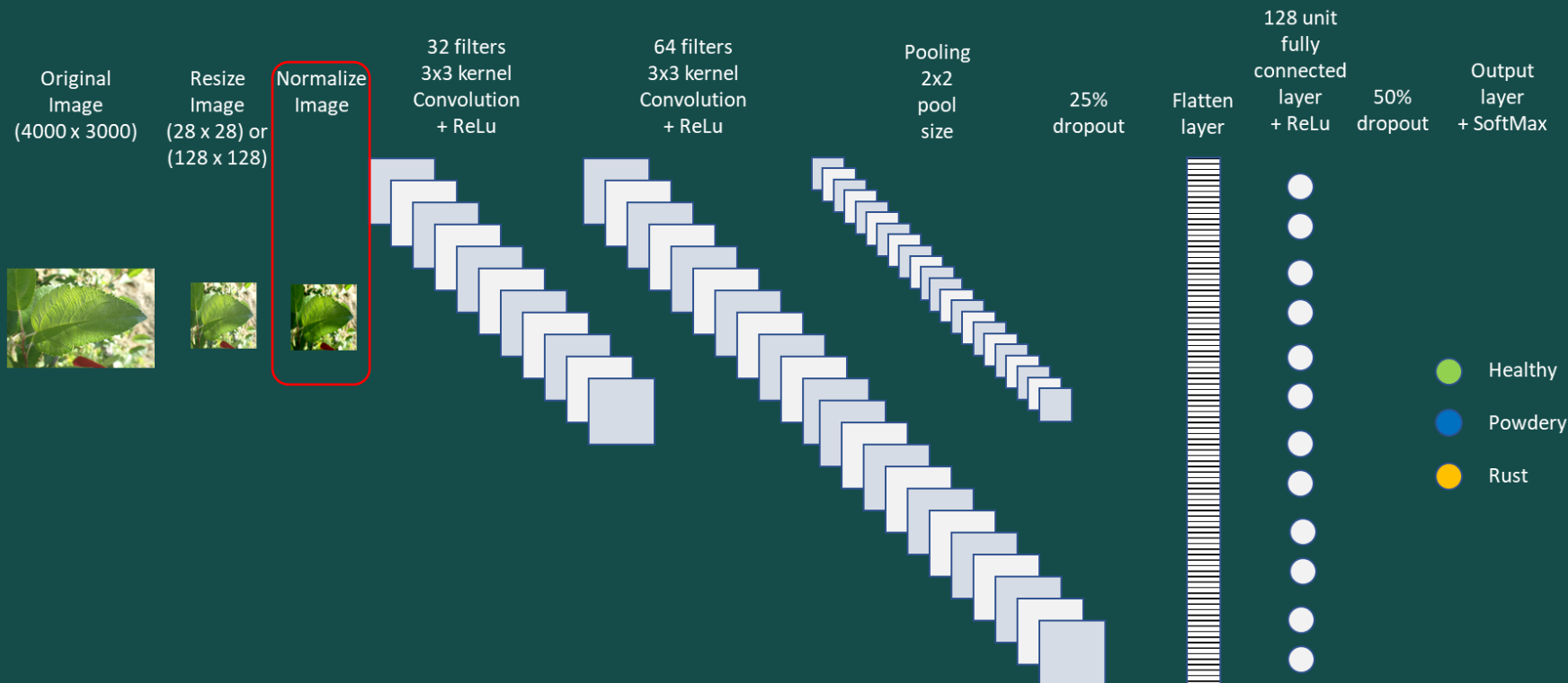
CNN Model Building – resize image

- 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.



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.

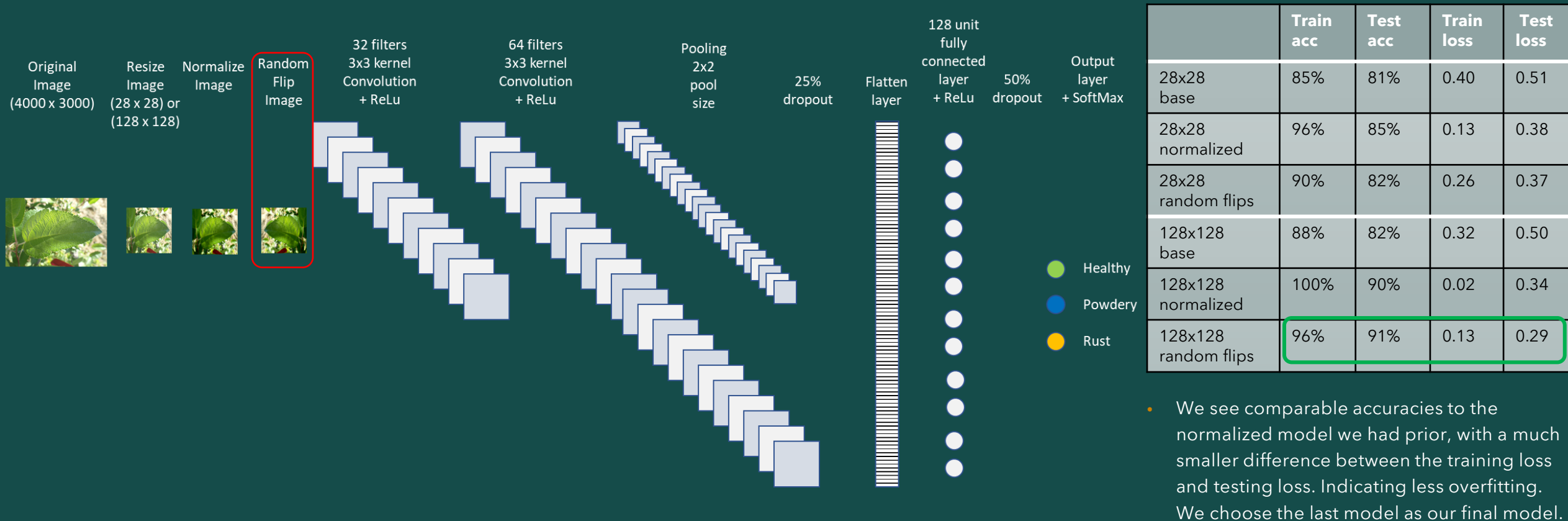


	Train acc	Test acc	Train loss	Test loss
28x28 base	85%	81%	0.40	0.51
28x28 normalized	96%	85%	0.13	0.38
128x128 base	88%	82%	0.32	0.50
128x128 normalized	100%	90%	0.02	0.34

- 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.

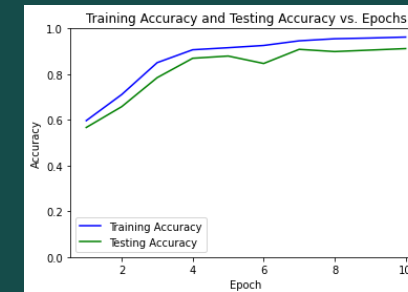
CNN Model Building – random image flips

- 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).

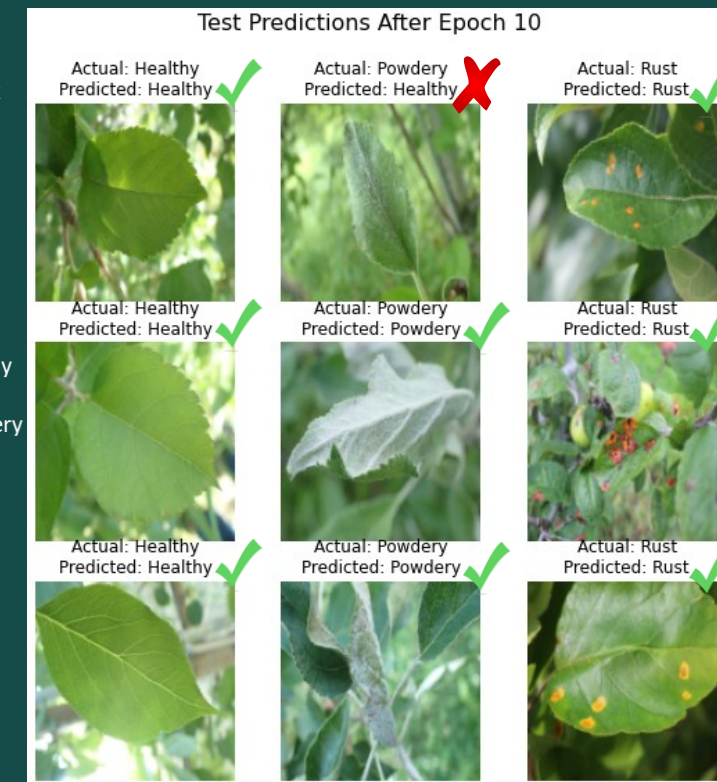
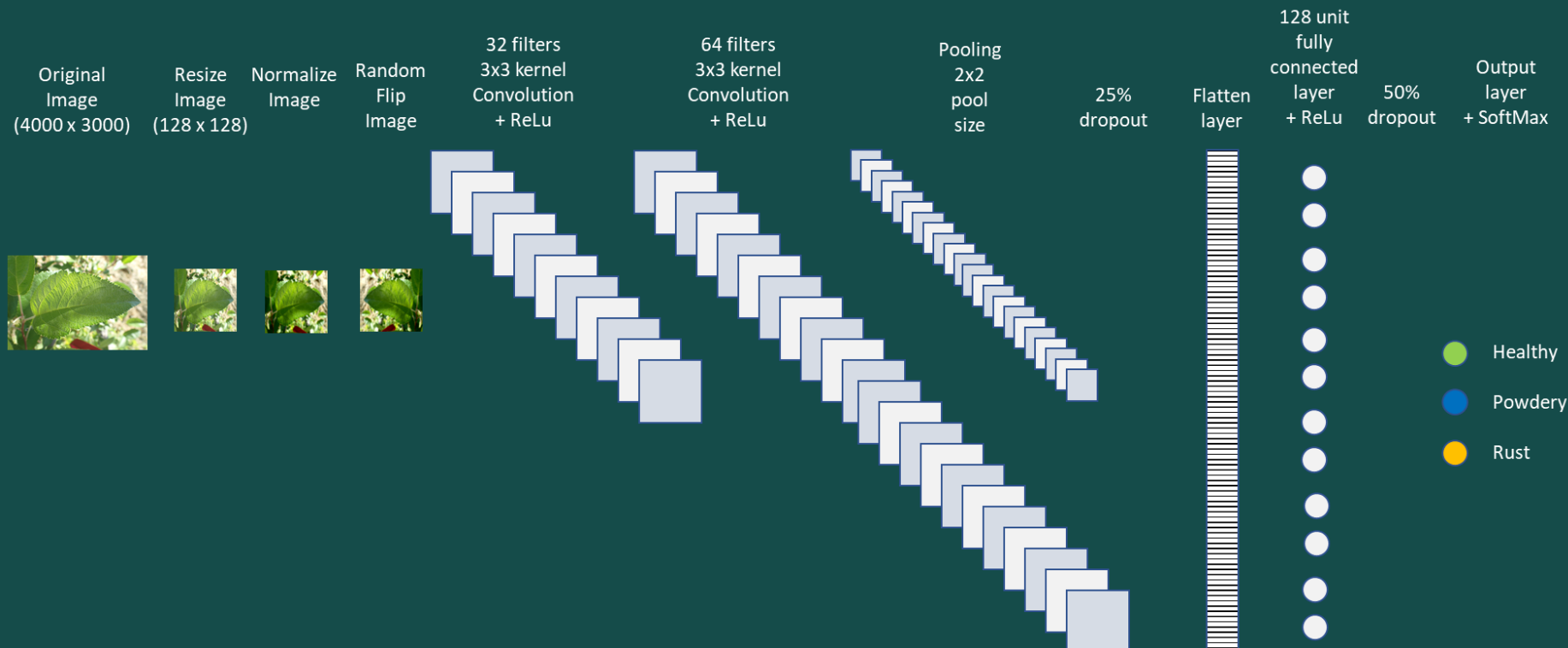


CNN Model Building – final model

- Our final model is as per below.
- Using our final model, we also did some sample testing drawn from the test set.



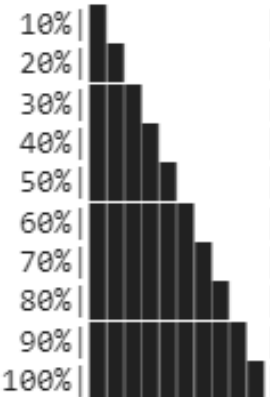
	Train acc	Test acc	Train loss	Test loss
Final Model	96%	91%	0.13	0.29



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, 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
- 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 (Resnet50)



Epoch	Progress	Time	Mean Training Loss	Test Loss	Test Accuracy
1/10	10%	[03:44<33:38, 224.29s/it]	0.7033	0.3651	0.9121
2/10	20%	[07:27<29:50, 223.84s/it]	0.3387	0.2892	0.9088
3/10	30%	[11:08<25:57, 222.44s/it]	0.2796	0.2207	0.9316
4/10	40%	[14:49<22:11, 221.87s/it]	0.2723	0.1877	0.9544
5/10	50%	[18:33<18:32, 222.53s/it]	0.2224	0.1769	0.9446
6/10	60%	[22:14<14:47, 221.96s/it]	0.1877	0.1866	0.9381
7/10	70%	[25:53<11:03, 221.26s/it]	0.1775	0.1577	0.9446
8/10	80%	[29:36<07:23, 221.65s/it]	0.1933	0.1615	0.9577
9/10	90%	[33:22<03:42, 222.92s/it]	0.1531	0.1426	0.9544
10/10	100%	[37:04<00:00, 222.43s/it]	0.1696	0.1443	0.9544

```
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		72%	Y
SGD	Y		74%	Y
Random Forest Classifier	Y		75%	Y
Gradient Boosting Classifier	Y		74%	
Self-made CNN	Y	Y	91%	
Transfer learning Resnet50	Y	Y	95%	

Thank you!

