D604 Advanced Analytics Task 1
By Eric Williams

A1:RESEARCH QUESTION

Can a neural network identify the species of plant seedlings? If so, how accurately can the network identify plant species in images?

A2:OBJECTIVES OR GOALS

The goal of this project is to create a neural network that can classify the species of 12 seedlings using images of the seedlings as training. This model could be able to assist botanists and make their work more efficient by allowing the network to identify seedlings for the. This could save them valuable time and effort during the seedling identification process.

A3:NEURAL NETWORK TYPE

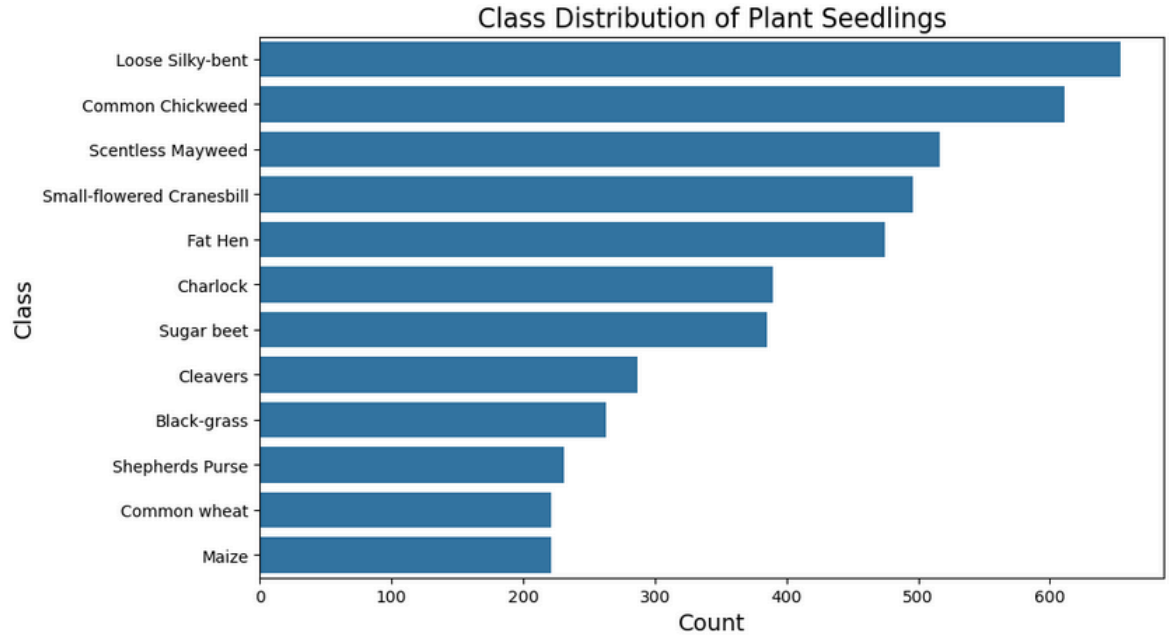
I have chosen a Convolutional Neural Network (CNN) for this project. The network will be designed to process images and identify the plants based on the images provided for the project.

A4:NEURAL NETWORK JUSTIFICATION

CNNs are great for image classification because they specialize in extracting features from images. They use convolutional operations to extract key points of images, such as edges and other features. By pooling layers and using convolutional filters, CNNs can be effective while not being overly computationally heavy.

B1A:IMAGE: DATA VISUALIZATION

Here is a bar plot of the class distribution of plant seedlings:



B1B:IMAGE: SAMPLE IMAGES

Here is a sample of the images I was given for this project with labels:

B2:IMAGE: AUGMENTATION AND JUSTIFICATION

To create a larger training set, we can augment our data by applying random transformations to the pictures we have and including them as new data points. We can rotate and flip the images, change the brightness or contrast of the images, or zoom in on the existing pictures to create new pictures of the same plants. By expanding our training set, our model should become more accurate and less prone to overfitting. In my augmentation, I included these possibilities:

- Randomly rotating the images by up to 30 degrees
- Randomly shifting the images horizontally or vertically by 20%
- Apply shear transformations up to 20%
- Zoom in on images up to 20%
- Flip the images horizontally

Because some of these transformations would leave the pictures incomplete or with missing data, I chose to fill in the blank spaces with the ‘nearest’ pixels. Because this fill data should be randomly dispersed across the images, it is unlikely to significantly affect our training model.

B3:IMAGE: NORMALIZATION STEPS

Here is the code for normalizing the pixels:

```
#Normalizing images, pixel values to range [0, 1]
images = images / 255.0
```

Now the values of the pixels are all between 0 and 1 rather than from 0 to 255. They still have the same proportional value relative to each other, but the numbers are smaller and should be easier to process.

B4:IMAGE: TRAIN-VALIDATION-TEST

I decided to do a 70-15-15 train, validation, test split. The largest dataset should be the training model to ensure the model has plenty of data to create an accurate model. I decided to take the remaining 30% and split it equally for validation and testing.

B5:IMAGE: TARGET ENCODING

Although I encoded the data in the previous step, the data needs to be one-hot encoded to be evaluated using TensorFlow:

```
#Target Feature Encoding (One-hot encoding)
from tensorflow.keras.utils import to_categorical

y_train_encoded = to_categorical(y_train)
y_val_encoded = to_categorical(y_val)
y_test_encoded = to_categorical(y_test)
```

B6:IMAGE: DATASETS COPY

The datasets are attached to this submission and will not be included here.

E1:MODEL SUMMARY OUTPUT

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 126, 126, 32)	896
max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
flatten (Flatten)	(None, 57600)	0
dense (Dense)	(None, 128)	7,372,928
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 12)	1,548

```
Total params: 7,393,868 (28.21 MB)
Trainable params: 7,393,868 (28.21 MB)
Non-trainable params: 0 (0.00 B)
```

E2: Components of the neural network

E2A:NUMBER OF LAYERS, E2B:TYPES OF LAYERS, E2C:NODES PER LAYER, E2D:NUMBER OF PARAMETERS, and **E2E:ACTIVATION FUNCTIONS**

To avoid repeating a lot of the same information, below I will break down my 8 layers. Layer by layer, I will discuss the number of layers, the type of layer, the purpose and function of each layer, the nodes per layer, the number of parameters, and the activation functions for each layer.

As shown above, the model has 8 layers. Here is each layer and all the necessary information needed for each layer:

1. **Conv2D (Convolutional Layer)**

When creating a CNN, the convolutional layers help the model learn specific features from the images we processed for the model. Including more layers deepens the network, but it would be computationally heavier to include more than a few layers. This is why I included two convolutional layers.

As you can see from the table listed in E1, this particular convolutional layer has 32 filters that produce a feature map. Although they are not truly necessarily nodes, they are distinct filtering points and including 32 is a reasonable practice. If we included too many, our model may be at risk of overfitting. Starting with 32 layers is fairly standard for finding features like edges and basic shape of the leaves.

The input shape is (128, 128, 3) as the image is 128 by 128 with three colors, while listed above the output shape is (126, 126, 3). The number of parameters is 896, which is a byproduct of a calculation based on the number of filters and the shape of the filter.

The activation function used in this layer is the ReLU (Rectified Linear Unit) function. This function is helpful in convolutional layers because it is simple, is computationally efficient, and it helps with training efficiency.

2. **MaxPooling2D (Pooling Layer)**

The pooling layers are necessary to help reduce dimensionality and reduce the required computational power for the network while simultaneously helping prevent overfitting. The pool size is (2,2), the input shape is (126, 126, 3) and the output shape is (63, 63, 32) because the pooling reduces the dimensions by half. If you consider the number of feature maps to be the number of nodes, then there are 32. The number of parameters is 0 because there are no parameters in pooling layers--it is a mere calculation to decrease dimension. There are no activation functions for this layer.

3. **Conv2D (Second Convolutional Layer)**

When creating a CNN, the convolutional layers help the model learn specific features from the images we processed for the model. Including more layers deepens the network, but it would be

computationally heavier to include more than a few layers. This is why I included two convolutional layers.

As you can see from the table listed in E1, this particular convolutional layer has 64 filters that produce a feature map. Although they are not truly necessarily nodes, they are our distinct filtering points and including 64 points will help the module capture more information than the previous layer did. To distinguish between plants, we'll need more information than just the general shape or edges. Yet, if we included too many filters, our model may be at risk of overfitting.

The input shape is (63, 63, 32), while listed above the output shape is (61, 61, 64). The number of parameters is 18,496, which is a byproduct of a calculation based on the number of filters and the shape of the filter.

The activation function used in this layer is the ReLU (Rectified Linear Unit) function. This function is helpful in convolutional layers because it is simple, is computationally efficient, and it helps with training efficiency.

4. **MaxPooling2D (Second Pooling Layer)**

The pooling layers are necessary to help reduce dimensionality and reduce the required computational power for the network while simultaneously helping prevent overfitting. The pool size is (2,2), the input shape is (61, 61, 64) and the output shape is (30, 30, 64) because the pooling reduces the dimensions by half. If you consider the number of feature maps to be the number of nodes, then there are 64.The number of parameters is 0 because there are no parameters in pooling layers--it is a mere calculation to decrease dimension. There are no activation functions for this layer.

5. **Flatten**

The flattening layer converts the two dimensional maps created above into a one dimensional array. While the input is (61, 61, 64), the output is a one dimensional array with size 57,600 to make it more efficient computationally. This is the total number of features. Because there are 57,600 points in the array, there are that many nodes. There are no parameters because flattening doesn't create parameters--it just reshapes the existing data. There are no activation functions for this layer.

6. **Dense (Fully Connected Layer)**

The fully connected dense layer combines features to make predictions. It takes the flattened data (an array of 57,600 points) and does calculations to restructure the data into 128 neurons, or units/nodes. There are 7,372,928 parameters, but this is not necessarily the direct result of a decision--this number comes from a calculation dependent on the number of units and the number of data points in the input.

The activation function used in this layer is the ReLU (Rectified Linear Unit) function. This function is helpful here because it is simple, is computationally efficient, and it helps with training efficiency.

7. **Dropout**

The purpose of the Droupout rate is to turn off 50% of the nodes to prevent the model from overfitting. It does not change the number of nodes, though it does deactivate half of them because of the 0.5 specification. Because it is just turning off neurons, there are no parameters. Similarly, there are no activation functions for this layer.

8. **Dense (Output Layer)**

The final Dense layer is the output layer. Similar to the previous Dense layer, this layer combines features to make predictions. Although it takes in 128 nodes, it reduces them to 12. This is because we have 12 seedlings we are trying to identify. This layer has 1,548 parameters, which is a result of the calculation involving the number of nodes and the number of datapoints that are used as input. The activation function I used for the output layer is Softmax, which converts raw output into probabilities. Whichever class ends up with the highest probability, that is the class the model predicts.

E3A:LOSS FUNCTION
Backpropagation: When I first ran the model, the accuracy was abysmal. After troubleshooting the problem, I realized the network was predicting the most common data input (Loose silky-bent) far too often. This meant the network was skewing toward the more common data entries and I fixed the problem by weighting the compute class in an attempt to have the network give more weight to underrepresented data. It worked, and the accuracy of the model was much higher moving forward.

For my **loss function**, I chose categorical cross-entropy, which is especially effective when a model is calculating the probability distribution to make predictions, just as the Dense layer above does with the softmax activation. When dealing with several classes, it can be especially effective.

E3B:OPTIMIZER
For my optimizer, I chose Adam (Adaptive Moment Estimation). Because my model had trouble with the weighting of the variables, Adam was perfect as it helps adjust the weights of the neural network. It is also a versatile network that works for many neural networks and is especially useful when data has high variance.

E3C:LEARNING RATE
For my learning rate, I decided to leave it unchanged from the Adam default. The default case will generally work well unless you have a reason to change it and I did not have such a reason. Having an appropriate learning rate can help find the optimal solution and will increase the effectiveness of the optimizer.

E3D:STOPPING CRITERIA

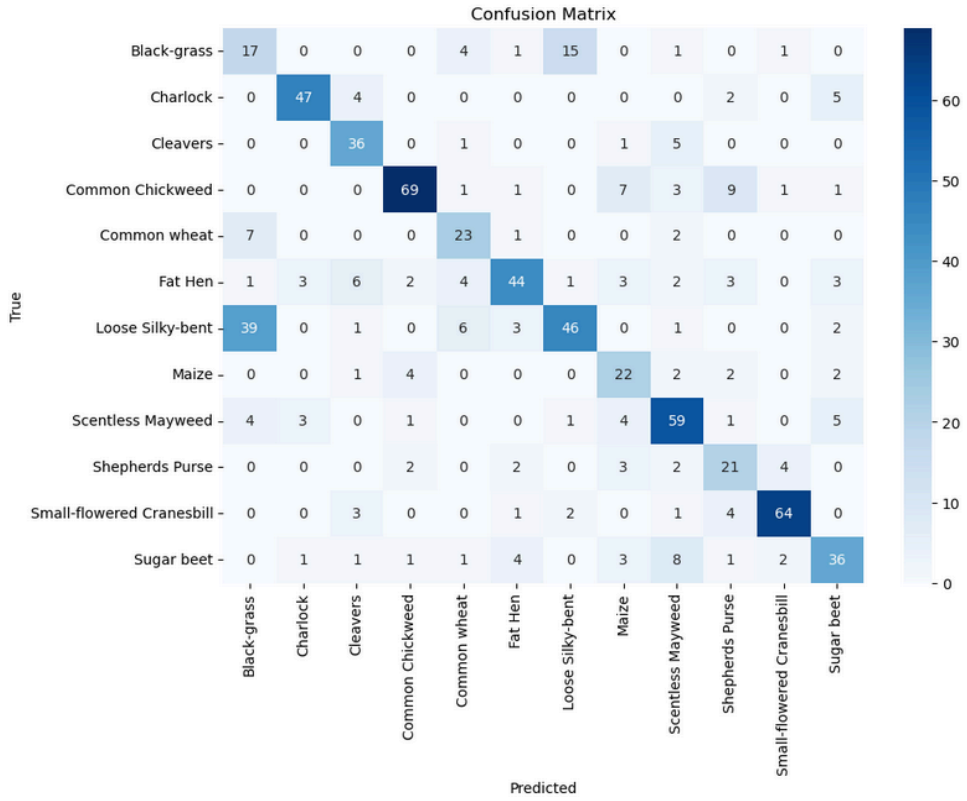
For my stopping criteria, I chose EarlyStopping with patience=5. If you have too high of a patience, the model will overfit the data and will perform poorly when you introduce new data. Having a patience of 5 will stop the training if the validation loss stops improving for 5 epochs in a row, which indicates that the model is performing worse on the validation set and thus it won't do well when given new data. If I chose a lower patience, the model might stop training before the optimal epoch, which would cause our model to be less accurate. Apart from overfitting, the having a stopping criteria saves computation time when the model stops improving.

E4:CONFUSION MATRIX

First I created a simple confusion matrix:

```
array([[13,  0,  0,  0,  6,  0, 18,  1,  0,  0,  1,  0],
       [ 0, 37, 12,  0,  0,  3,  0,  0,  2,  0,  0,  4],
       [ 0,  2, 36,  0,  0,  2,  0,  0,  3,  0,  0,  0],
       [ 0,  1,  1, 77,  1,  1,  0,  3,  1,  5,  1,  1],
       [ 4,  0,  0,  0, 14,  3,  9,  0,  2,  0,  0,  1],
       [ 1,  1,  8,  3,  2, 39,  9,  0,  4,  2,  1,  2],
       [14,  0,  0,  0, 25,  2, 56,  0,  1,  0,  0,  0],
       [ 0,  2,  1,  4,  0,  1,  0, 20,  1,  2,  1,  1],
       [ 1,  2,  3,  6,  2,  1,  0,  0, 47,  6,  0, 10],
       [ 0,  0,  3,  6,  0,  2,  0,  1,  2, 14,  6,  0],
       [ 1,  3,  4,  1,  0,  2,  0,  0,  0,  2, 60,  2],
       [ 2,  2,  7,  1,  4, 12,  0,  4,  4,  0,  2, 20]], dtype=int64)
```

Then I decided to create a heat map matrix to better observe the accuracies and inaccuracies:



As you can see, the entries on the diagonal are the accurate predictions by the model and those are the most common entries. However, it was especially common for the model to mix up Loose Silky-bent with Black-Grass. The model was not able to discern between these two plants better than a coin flip. Whether this is because of some issue with the data, the training, or if the plants just look alike, this issue should be addressed in the future.

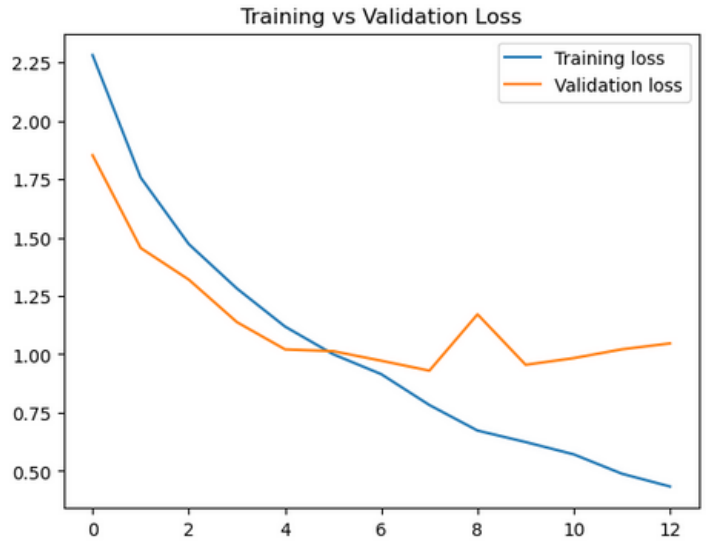
F1A:STOPPING CRITERIA IMPACT

As mentioned above, the stopping criteria is important to prevent overfitting. Because I used an EarlyStopping of patience=5, the model stopped training after the data did worse when seeing new data. Even if the model is improving at the other datasets, if it is not performing well on new data (the validation set) then the model is not truly improving. Thus, the impact of stopping criteria in this case helped prevent overfitting and cut down on computational time by stopping the training when the model was no longer improving.

Here is a screenshot of the final epochs:

```
Epoch 5/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 16s 150ms/step - accuracy: 0.6008 - loss: 1.1094 - val_accuracy: 0.6784 - val_loss: 1.0207
Epoch 6/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 15s 145ms/step - accuracy: 0.6449 - loss: 1.0167 - val_accuracy: 0.6756 - val_loss: 1.0130
Epoch 7/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 16s 151ms/step - accuracy: 0.6833 - loss: 0.9147 - val_accuracy: 0.6840 - val_loss: 0.9721
Epoch 8/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 16s 152ms/step - accuracy: 0.7102 - loss: 0.7991 - val_accuracy: 0.7008 - val_loss: 0.9296
Epoch 9/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 16s 154ms/step - accuracy: 0.7747 - loss: 0.6764 - val_accuracy: 0.6180 - val_loss: 1.1710
Epoch 10/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 15s 147ms/step - accuracy: 0.7720 - loss: 0.6265 - val_accuracy: 0.7135 - val_loss: 0.9547
Epoch 11/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 16s 152ms/step - accuracy: 0.8151 - loss: 0.5576 - val_accuracy: 0.7079 - val_loss: 0.9833
Epoch 12/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 15s 145ms/step - accuracy: 0.8380 - loss: 0.4861 - val_accuracy: 0.7093 - val_loss: 1.0211
Epoch 13/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 15s 143ms/step - accuracy: 0.8560 - loss: 0.4223 - val_accuracy: 0.6966 - val_loss: 1.0463
```

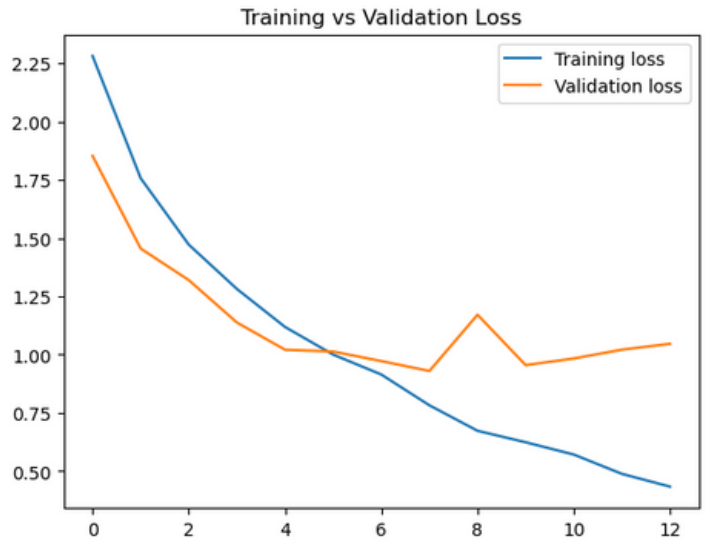
F1B:EVALUATION METRICS



Above is a comparison of the training loss against the validation loss. When the validation loss stops decreasing and starts increasing, the model is at its optimal training epoch. This is when the model has not begun overfitting, but has gained as much accuracy through training that it can. According to our graph, around epoch 9, our model is no longer performing better on the validation set. If we let this go on without early stopping, the model would become better at predicting the test or training set, but would be worse at predicting for new data.

F1C:VISUALIZATION

As pictured above, here is a visualization comparing the training loss to the validation loss:



F2:MODEL FITNESS

As you can gather from the confusion matrix above, the model does great in identifying 10 of the 12 plants with a high level of accuracy--but the network has trouble differentiating between two plants (Loose Silky-bent and Black-Grass). But even with this confusion, the model is still about 68% accurate:

```
23/23 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.6701 - loss: 1.0045
Test Accuracy: 0.678821861743927
```

This means the model fits well and my attempts to prevent overfitting were largely successful. Here is a list of the prevention methods that I used to prevent overfitting:

- **Dropout**: Creating a Dropout layer with a 50% rate prevented the model from becoming overfit by reducing the number of neurons in the fully connected layer.
- **Early Stopping**: Stopping the model when it is improving accuracy but is getting worse on the validation set is an excellent way to prevent overfitting as discussed above.
- **Balancing Plant Distribution**: By weighting the uneven plant distribution, the model did not become fixated on the Loose Silky-bent plant like it did previously.

F3:PREDICTIVE ACCURACY

```
23/23 ━━━━━━━━━━━━━━━━━━━━ 1s 31ms/step - accuracy: 0.6701 - loss: 1.0045
Test Accuracy: 0.678821861743927
```

As shown above, the test accuracy was around 68%. The accuracy would have been much higher, but the network was not able to accurately differentiate between Loose Silky-bent and Black-Grass. Besides these two plants, the accuracy was very high.

G1:CODE

Here is the code I used to save the trained network within the neural network:

```
model.save(r"C:\\Users\\18014\\Desktop\\plant_seedling_classifier.h5")
```

G2:NEURAL NETWORK FUNCTIONALITY

The network's functionality was overall a success with one particular exception, as mentioned above. The network is functional in predicting 10 of the 12 plants. As mentioned above, the architecture of the neural network helped contribute to its effectiveness. The use of convolutional layers helped the model detect patterns in edges and texture, while max pooling layers helped reduce dimension and improve computation efficiency. The flattening layer helped transform the data into an array that could be more easily processed. The Dense layers combined the data to create predictions, while the Dropout layer helped prevent overfitting. These layers all worked together to create a neural network that was largely accurate in identifying plants while still being computationally efficient.

G3:BUSINESS PROBLEM ALIGNMENT

My business question was: Can a neural network identify the species of plant seedlings? If so, how accurately can the network identify plant species in images? The answer is that yes, the neural network can accurately identify 10 out of 12 plant seedlings, and the other 2 plant seedlings are too difficult to differentiate. Regardless, the model performs with 68% accuracy.

G4:MODEL IMPROVEMENT

The model could easily be improved by providing more data so the number of seedlings to identify is equal. Although weighting the data helped the model perform much better, having more data would be an improvement. The model had particular trouble differentiating between Loose Silky-bent, which was the most over-represented plant and thus the most trained plant, and Black-Grass, which only accounted for 5% of the training data. The model could also possibly be improved by better photos, a reconfiguration of the neural network layers to optimize the process.

G5:RECOMMENDED COURSE OF ACTION

My recommendation is for botanists to use this model as a shortcut to identify plants with the exception of identifying Loose Silky-bent and Black-Grass. If the model identifies either of those two plants, they should be differentiated by hand. This model could save botanists time and energy when identifying plants.

H:OUTPUT

Here is the codel used to save the neural network and to output in an html format:

```
#Saving the
model.save(r"C:\\Users\\18014\\Desktop\\plant_seedling_classifier.h5")
```

```
#Exporting as html
!jupyter nbconvert --to html "D604 Task 1A.ipynb" --output "C:/Users/18014/Desktop/D604_Task_1A.html"
```

I:SOURCES FOR THIRD-PARTY CODE

Code was used from the Tensorflow website for data augmentation and neural network architectural design.

Code for class weights was used from Data Science Stack Exchange.

J:SOURCES

Cecchini, David. (n.d.). Intermediate Deep Learning with PyTorch. DataCamp. Retrieved December 2024 from https://app.datacamp.com/learn/courses/intermediate-deep-learning-with-pytorch

Gurucharan, MK. (2024). Basic CNN Architecture: Explaining 5 Layers of Convolutional Neural Network. upGrad. Retrieved December 2024 from https://www.upgrad.com/blog/basic-cnn-architecture/