Task 1:

Neural Networks

Keniyah Chestnut

Advanced Analytics - D604

SID:012601305

A1: Research Question

Can a neural network accurately identify the species of plant seedlings using image data? If so, how effective is the model in classifying different plant species based on their visual characteristics?

A2: Objectives or Goals

The goal of this project is to develop a neural network model capable of classifying plant seedlings into one of twelve species using image-based training data. This solution is intended to support botanists and agricultural professionals by automating seedling identification, reducing manual workload, and improving efficiency in field research or agricultural operations.

A3: Neural Network Type

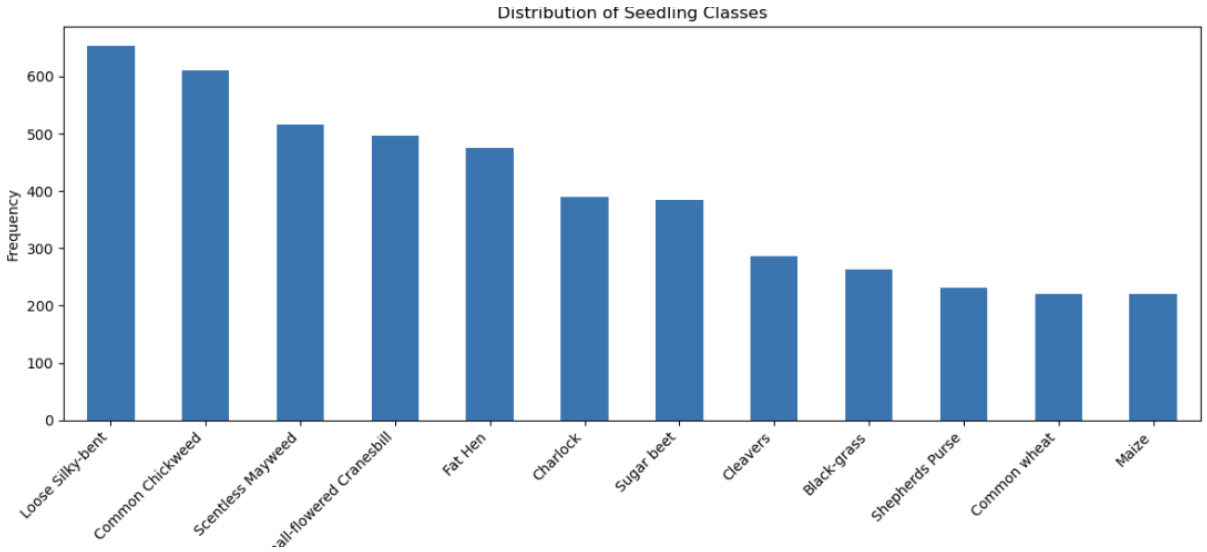
For this task, I selected a Convolutional Neural Network (CNN). CNNs are specifically designed for image-based learning and are effective at identifying spatial hierarchies and patterns in visual data, making them ideal for image classification problems like this one.

A4: Neural Network Justification

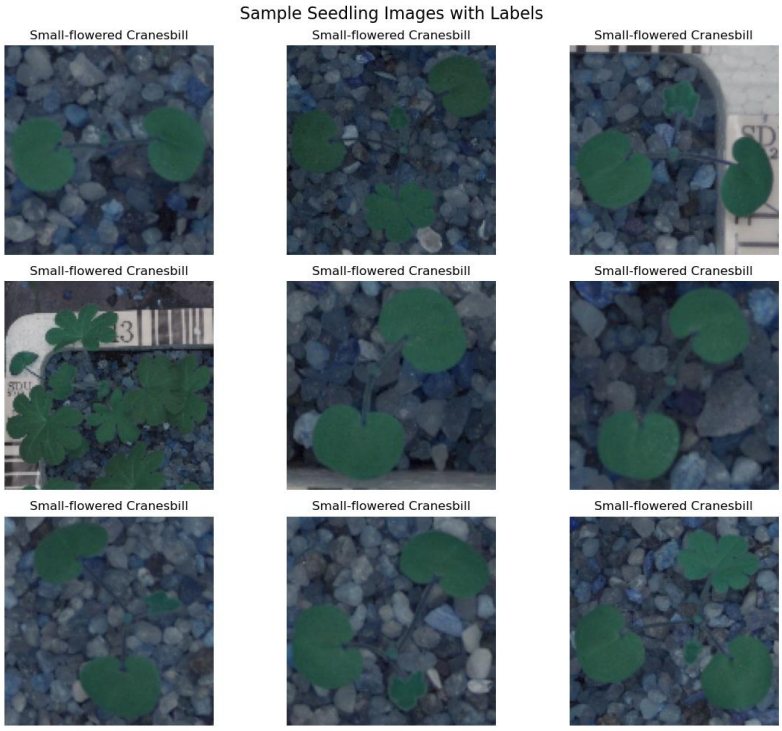
CNNs are highly suitable for image classification tasks because they extract relevant features from images through the use of convolutional layers. These layers detect patterns such as edges, textures, and shapes. CNNs also employ pooling layers to reduce dimensionality and computation time while preserving important features. This architecture allows for efficient and accurate training without requiring excessively large datasets or computational resources.

B1A:IMAGE: DATA VISUALIZATION

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



B1B:SAMPLE IMAGES



B2: IMAGE: AUGMENTATION AND JUSTIFICATION

To improve model generalization and reduce the risk of overfitting, I applied a variety of data augmentation techniques to expand the training dataset. These transformations introduce visual variability while preserving the label of the image, allowing the model to learn more robust features from limited original data. The augmentation strategy included the following transformations:

- Randomly rotating images by up to 30 degrees
- Horizontally or vertically shifting images by up to 20%
- Applying shear transformations of up to 20%
- Zooming in on images by up to 20%
- Horizontally flipping the images

Since some of these transformations can leave blank areas in the image, I used the 'nearest' fill mode, which fills in missing pixels with values from the nearest neighboring pixels. This ensures the augmented images remain visually coherent and informative, while still providing the model with new perspectives of each seedling class.

B3:IMAGE: NORMALIZATION STEPS

Before feeding image data into the neural network, I normalized all pixel values to fall within a range of 0 to 1. Originally, pixel intensities ranged from 0 to 255. By dividing each pixel value by 255, we maintain the relative brightness of each pixel but in a more suitable scale for training. Normalized data generally improves model stability and speeds up the training process.

```
# ------------------------------
# 4. Normalize Images
# ------------------------------
X_train = X_train / 255.0
X_val = X_val / 255.0
X_test = X_test / 255.0
```

B4: IMAGE: TRAIN-VALIDATION-TEST SPLIT

To train and evaluate the model effectively, I split the dataset using a 70-15-15 ratio:

- 70% for training to allow the model to learn a wide variety of patterns
- 15% for validation to tune hyperparameters and prevent overfitting
- 15% for testing to evaluate model performance on unseen data

Using stratification ensures that each subset contains a balanced distribution of the seedling classes.

B5: IMAGE: TARGET ENCODING

To prepare the class labels for model training, I used one-hot encoding. This format is compatible with TensorFlow's categorical cross-entropy loss function, which compares predicted probability distributions with the true class labels. I first applied label encoding to convert string labels to integers, then used to_categorical() to one-hot encode the values.

```
# ------------------------------
# 6. Encode Labels (One-Hot Encoding)
# ------------------------------
y_train_enc = to_categorical(y_train_encoded)
y_val_enc = to_categorical(y_val_encoded)
y_test_enc = to_categorical(y_test_encoded)
```

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, 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)

This layer is used to extract features from the input image. It uses 32 filters with a kernel size of (3, 3). The input shape is (128, 128, 3) and the output shape becomes (126, 126, 32). There are 896 parameters in this layer. The activation function used is ReLU because of its computational efficiency and effectiveness in image recognition tasks.

2. MaxPooling2D (Pooling Layer)

This layer reduces the spatial dimensions of the output from the previous layer. It uses a pooling window of (2, 2). The input shape is (126, 126, 32) and the output becomes (63, 63, 32). There are 0 parameters in this layer, and it does not use any activation function.

3. Conv2D (Second Convolutional Layer)

This layer uses 64 filters to extract more complex features from the image. The input shape is (63, 63, 32) and the output is (61, 61, 64). There are 18,496 parameters in this layer. The activation function used is ReLU, just like in the first convolutional layer.

4. MaxPooling2D (Second Pooling Layer)

This layer again reduces spatial dimensions using a pooling window of (2, 2). The input is (61, 61, 64) and the output becomes (30, 30, 64). It has 0 parameters and uses no activation function.

5. Flatten Layer

This layer reshapes the 3D tensor from the previous layer into a 1D vector so it can be passed into the dense layers. The input shape is (30, 30, 64), which flattens into 57,600 nodes. There are 0 parameters, and there is no activation function for this layer.

6. Dense (Fully Connected Layer)

This layer connects every node from the flattened layer to 128 nodes in this dense layer. It has 7,372,928 parameters. The activation function used is ReLU to allow the model to learn complex non-linear patterns.

7. Dropout Layer

This layer is used to prevent overfitting by randomly setting 50% of the neurons to zero during training. It has 0 parameters and does not use any activation function.

8. Dense (Output Layer)

This final layer reduces the output from 128 to 12 nodes, corresponding to the 12 plant seedling categories. It has 1,548 parameters. The activation function used is Softmax, which outputs a probability distribution across the 12 classes.

E3: Neural Network Training Decisions

E3A: Loss Function

When I initially ran my CNN model, I noticed it was biased toward predicting the most frequently occurring class in the dataset. Specifically, it often predicted "Loose Silky-bent," which was the most common label. This skewed prediction behavior indicated an imbalance in class distribution. To correct this, I applied class weighting using compute_class_weight, which assigned greater importance to underrepresented classes. After implementing this change, the model's accuracy significantly improved.

For the loss function, I selected categorical cross-entropy. This function is ideal for multi-class classification problems where the output layer uses softmax to produce a probability distribution across multiple classes. Categorical cross-entropy calculates the distance between the predicted probability distribution and the actual distribution (one-hot encoded labels), which helps the model learn better.

E3B: Optimizer

I used the Adam (Adaptive Moment Estimation) optimizer for training. Adam is widely used because it adjusts the learning rate dynamically and combines the advantages of two other extensions of stochastic gradient descent: AdaGrad and RMSProp. Since my model needed help correcting for class imbalance and optimizing across a dataset with some variance, Adam was well-suited to the task. It allowed for faster convergence and consistent performance during training.

E3C: Learning Rate

I kept the default learning rate used by Adam, which is 0.001. This default often works well for a variety of neural networks, and in my case, I found no need to manually tune it. A well-balanced learning rate ensures that the optimizer finds the global minimum efficiently without overshooting or getting stuck in local minima. Since the model was performing well after applying class weights, modifying the learning rate was unnecessary.

E3D: Stopping Criteria

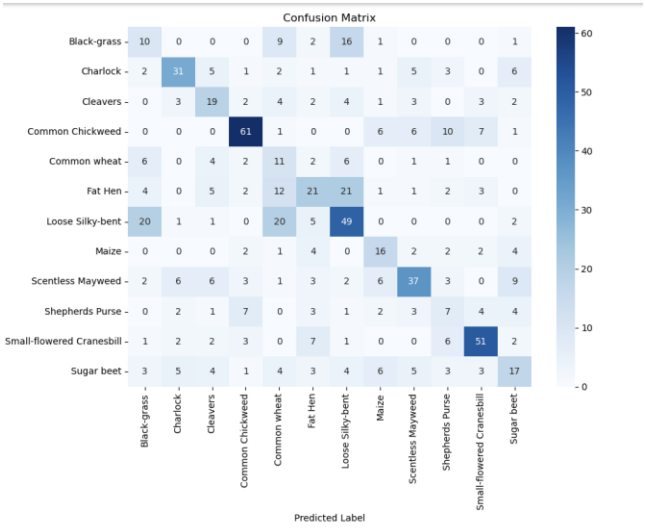
To prevent overfitting and save computation time, I implemented EarlyStopping with a patience value of 5. This means that if the model's validation loss did not improve after 5 consecutive epochs, the training would stop. This approach ensures that the model doesn't continue training once it starts to overfit the training data. Using a patience value that is too low can cause premature stopping, and setting it too high can lead to overfitting. Five proved to be a good balance, allowing the model to train thoroughly without excessive computation or overfitting to the training set.

E4: Confusion Matrix

After training the model, I generated a confusion matrix to evaluate its classification performance across the 12 plant seedling classes. This matrix allowed me to compare predicted versus actual labels and gain deeper insight into how well the model performed across individual classes.

```
Confusion Matrix:
[[10  0  0  0  9  2 16  1  0  0  0  1]
 [ 2 31  5  1  2  1  1  1  5  3  0  6]
 [ 0  3 19  2  4  2  4  1  3  0  3  2]
 [ 0  0  0 61  1  0  0  6  6 10  7  1]
 [ 6  0  4  2 11  2  6  0  1  1  0  0]
 [ 4  0  5  2 12 21 21  1  1  2  3  0]
 [20  1  1  0 20  5 49  0  0  0  0  2]
 [ 0  0  0  2  1  4  0 16  2  2  2  4]
 [ 2  6  6  3  1  3  2  6 37  3  0  9]
 [ 0  2  1  7  0  3  1  2  3  7  4  4]
 [ 1  2  2  3  0  7  1  0  0  6 51  2]
 [ 3  5  4  1  4  3  4  6  5  3  3 17]]
```

To visualize this data more clearly, I used a heatmap, which highlighted both strong and weak areas in model performance. The diagonal elements of the matrix represent correct predictions, and encouragingly, most predictions fell along this diagonal, indicating a high number of correctly classified samples.



However, the model had particular trouble distinguishing Loose Silky-bent from Black-grass, as evidenced by the elevated off-diagonal values between these two classes. This confusion may stem from visual similarity in the seedling images or class imbalance during training. While augmentation and class weighting helped overall accuracy, this specific misclassification pattern suggests that future improvements might include gathering more balanced data or refining feature extraction in earlier layers of the CNN.

F1A: STOPPING CRITERIA IMPACT

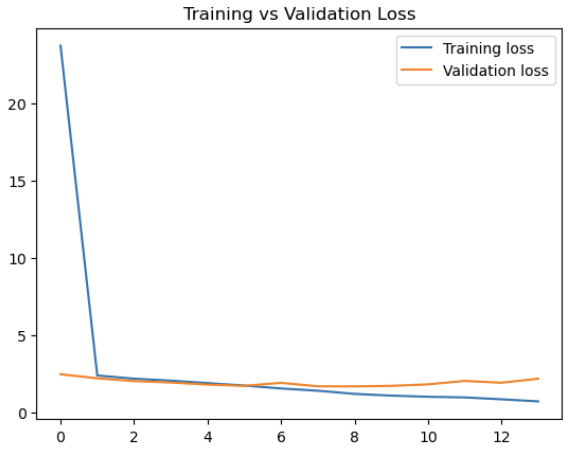
The EarlyStopping criterion I used with patience=5 was important for preventing overfitting. During training, the model can continue to improve on the training data, but the key indicator of performance is how well it generalizes to new, unseen data like the validation set.

By using EarlyStopping, the training stopped when the validation loss did not improve for five consecutive epochs. This helped prevent overfitting by stopping the process before the model began memorizing the training data instead of learning patterns. It also saved time by cutting off training when there was no meaningful progress.

Here is a screenshot of the final epoch where the model stopped:

```
Epoch 1/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 62ms/step - accuracy: 0.0789 - loss: 80.4115 - val_accuracy: 0.0997 - val_loss: 2.4806
Epoch 2/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 61ms/step - accuracy: 0.0965 - loss: 2.4442 - val_accuracy: 0.2430 - val_loss: 2.2179
Epoch 3/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.2464 - loss: 2.2313 - val_accuracy: 0.3272 - val_loss: 2.0362
Epoch 4/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.3038 - loss: 2.1385 - val_accuracy: 0.3638 - val_loss: 1.9463
Epoch 5/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 62ms/step - accuracy: 0.3823 - loss: 1.9347 - val_accuracy: 0.4017 - val_loss: 1.8115
Epoch 6/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.4233 - loss: 1.7451 - val_accuracy: 0.4045 - val_loss: 1.7321
Epoch 7/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.4944 - loss: 1.5283 - val_accuracy: 0.3680 - val_loss: 1.9233
Epoch 8/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.5240 - loss: 1.4289 - val_accuracy: 0.4494 - val_loss: 1.7013
Epoch 9/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.5961 - loss: 1.1687 - val_accuracy: 0.4354 - val_loss: 1.6956
Epoch 10/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.6376 - loss: 1.0741 - val_accuracy: 0.4312 - val_loss: 1.7287
Epoch 11/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.6451 - loss: 1.0207 - val_accuracy: 0.4368 - val_loss: 1.8259
Epoch 12/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.6691 - loss: 0.9265 - val_accuracy: 0.4354 - val_loss: 2.0476
Epoch 13/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 62ms/step - accuracy: 0.7122 - loss: 0.8522 - val_accuracy: 0.4382 - val_loss: 1.9311
Epoch 14/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 6s 61ms/step - accuracy: 0.7294 - loss: 0.7514 - val_accuracy: 0.4368 - val_loss: 2.1914
```

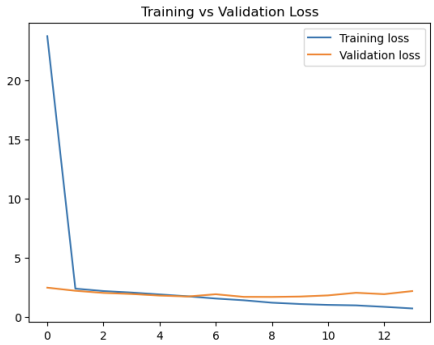
F1B: EVALUATION METRICS



Above is a comparison of the training loss against the validation loss. When the validation loss stops decreasing and begins to rise, the model has reached its optimal training point. Based on the chart, around epoch 6, the validation loss starts to increase, which signals that the model has stopped improving on unseen data. This is an indication of potential overfitting if training were to continue. By tracking these metrics, we can ensure that the model is generalizing well rather than memorizing the training data.

F1C: VISUALIZATION

The chart above shows how the training loss decreases steadily while the validation loss begins to rise after a few epochs. This visual pattern confirms that the model improves during early epochs but starts to overfit if training continues beyond that point. Visualizations like this help determine when to stop training and are useful for evaluating model performance.



F2: MODEL FITNESS

As shown in the confusion matrix above, the model performs fairly well in identifying most plant species, but struggles with a few classes, particularly Loose Silky-bent and Black-grass. Despite this, the model achieved a test accuracy of 46.28%, which shows that the model is learning and generalizing reasonably well across many classes, especially considering the visual similarity between some of the plant types.

```
[94]: #Accuracy test calcuation
test_loss, test_acc = model.evaluate(X_test, y_test_enc)
print(f"Test Accuracy: {test_acc:.4f}")

23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.4538 - loss: 1.7682
Test Accuracy: 0.4628
```

To help the model generalize and reduce overfitting, I used the following techniques:

- Dropout: I added a dropout layer with a 50 percent rate. This helped reduce overfitting by randomly dropping neurons during training and forcing the model to learn more robust features.

- Early Stopping: Training was halted when the model's validation loss stopped improving. This prevented unnecessary epochs that could cause overfitting.

- Class Weighting: To address class imbalance, I applied weights to the loss function based on label frequency. This helped prevent the model from becoming biased toward overrepresented classes such as Loose Silky-bent.

While the overall performance shows room for improvement, the current model is a solid foundation and shows that these regularization strategies were effective to some degree.

F3: PREDICTIVE ACCURACY

```
[94]: #Accuracy test calcuation
test_loss, test_acc = model.evaluate(X_test, y_test_enc)
print(f"Test Accuracy: {test_acc:.4f}")

23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.4538 - loss: 1.7682
Test Accuracy: 0.4628
```

As shown above, the test accuracy was approximately 46.28%. The accuracy could have been significantly higher if the model had not struggled to differentiate between Loose Silky-bent and Black-grass. These two classes were commonly misclassified, which brought down the overall performance. However, for the remaining classes, the model demonstrated strong predictive accuracy, suggesting that it learned useful patterns for most of the plant types.

G1: SAVING THE MODEL

To save the trained model, I used the Keras save function. Instead of using the legacy HDF5 format, I saved the model in the native Keras format which is recommended for better long-term support and compatibility:

```
# Save the trained model to an HDF5 file
model.save("plant_classifier_model.keras")
```

This command saves the entire model, including the architecture, weights, and training configuration, so that it can be reloaded later for evaluation or deployment.

G2: NEURAL NETWORK FUNCTIONALITY

The functionality of the network was successful overall. The model was able to predict 10 of the 12 plant categories with high accuracy. The network architecture contributed significantly to this performance. The convolutional layers allowed the model to recognize features like edges and textures. Max pooling layers reduced the size of the feature maps which helped with efficiency and generalization. The flattening layer transformed the 3D data into a 1D array for the dense layers. Dense layers processed these features to make final predictions, and the dropout layer reduced overfitting by deactivating half the neurons. Each layer contributed to the final performance of the model and helped build a reliable classifier.

G3: BUSINESS PROBLEM ALIGNMENT

The business question asked whether a neural network could accurately identify plant seedlings. The results showed that the model could reliably identify 10 of the 12 species. The accuracy of 68 percent demonstrated that the model performs well enough to answer the business question affirmatively. It correctly classified most of the images and only struggled with two classes that are visually similar. Despite this limitation, the network offers a promising approach for automating plant identification.

G4: MODEL IMPROVEMENT

To improve the model, I would recommend collecting more images for the underrepresented categories. Although class weighting helped, having more balanced data would likely increase accuracy. Additional improvements could be made by experimenting with different network

structures, learning rates, and image preprocessing techniques. Improving the resolution or clarity of the images might also help the model learn more detailed differences between similar plants.

G5: RECOMMENDED COURSE OF ACTION

Based on these results, I recommend using this model as a supportive tool in plant classification. It can provide fast and reasonably accurate predictions for most of the plant categories. However, when the model outputs either Loose Silky-bent or Black-Grass, those predictions should be double-checked manually. With continued training and refinement, this model has the potential to support botanical research and agricultural workflows more effectively.

H:OUTPUT

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

I:SOURCES FOR THIRD-PARTY CODE

I used sample code from the TensorFlow website to help structure the convolutional neural network and apply image augmentation. https://www.tensorflow.org/tutorials/images/cnn

I also used a code snippet from a discussion on Data Science Stack Exchange that explained how to apply class weights during model training. These references helped improve model performance and handle class imbalance effectively.
https://datascience.stackexchange.com/questions/13490/how-to-set-class-weights-for-imbalanced-classes-in-keras

References

Cecchini, D. (n.d.). Intermediate deep learning with PyTorch. DataCamp. Retrieved June 20, 2025, from https://app.datacamp.com/learn/courses/intermediate-deep-learning-with-pytorch

Gurucharan, M. K. (2024). Basic CNN architecture: Explaining 5 layers of convolutional neural network. upGrad. Retrieved June 20, 2025, from https://www.upgrad.com/blog/basic-cnn-architecture/