Task 1: Neural Networks & Computer Vision

Kevin Budzisch

Western Governors University

D604: Advanced Analytics

Professor Hanan Swidan

1/13/2026

A1: Research Question

For this task, I chose to use the provided images for my analysis. My research question is, can a neural network be used to identify the species of plant seedlings? If so, how accurately does the network identify plant species from seedling images?

A2: Objectives of the Data Analysis

The objective of this data analysis is to create a neural network that can identify the species of a plant seedling using images of seedlings as training data. This model could help botanists identify plants, making their work more efficient by allowing the neural network to identify seedlings on their behalf. This could save valuable time, money, and effort during the seedling identification process.

A3: Type of Neural Network

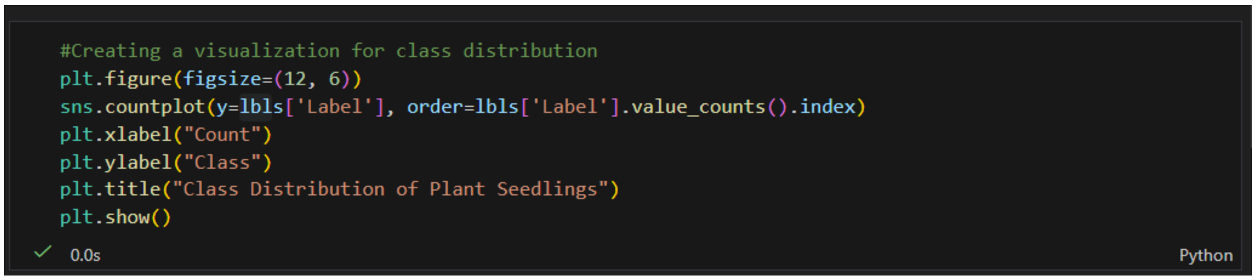
For this analysis, I have chosen to use images of the seedlings; therefore, I have selected a Convolutional Neural Network (CNN). This network will be designed to process images and identify plants using the images provided for this assignment.

A4: Justification of Neural Network

Using a CNN will be great for image classification. This type of neural network specializes in extracting features from images. CNNs use convolutional operations to extract key image features. By pooling multiple layers together and utilizing convolutional filters, CNNs can be effective while remaining computationally efficient.

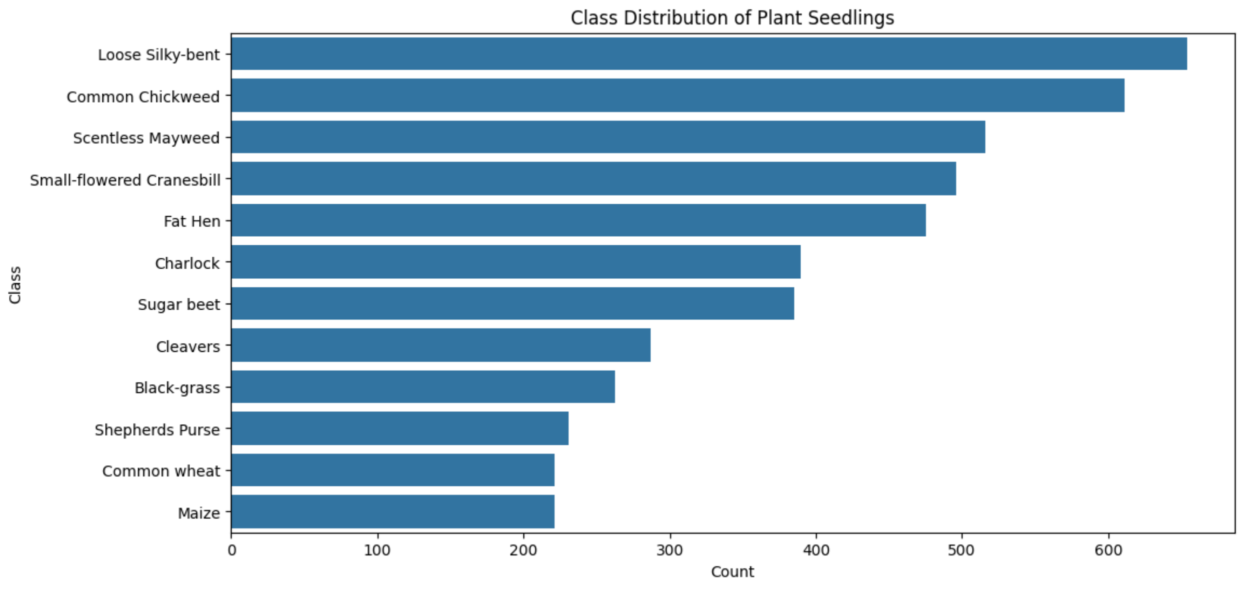
B1: Exploratory Data Analysis

Below is the code to create the bar plot of the class distribution of plant seedlings.

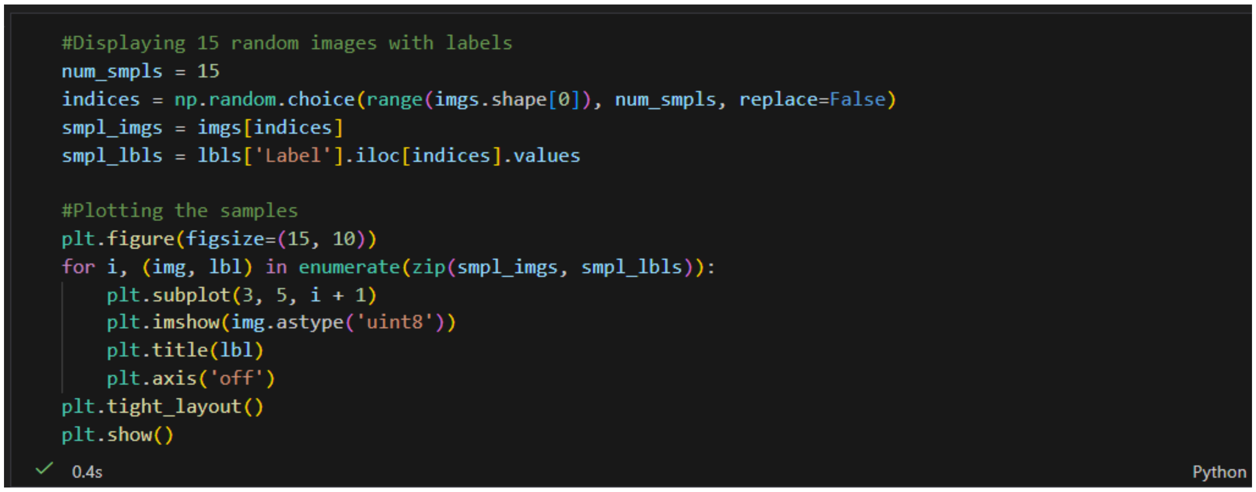


```python
#Creating a visualization for class distribution
plt.figure(figsize=(12, 6))
sns.countplot(y=lbls['Label'], order=lbls['Label'].value_counts().index)
plt.xlabel("Count")
plt.ylabel("Class")
plt.title("Class Distribution of Plant Seedlings")
plt.show()
```

Below is the bar plot.



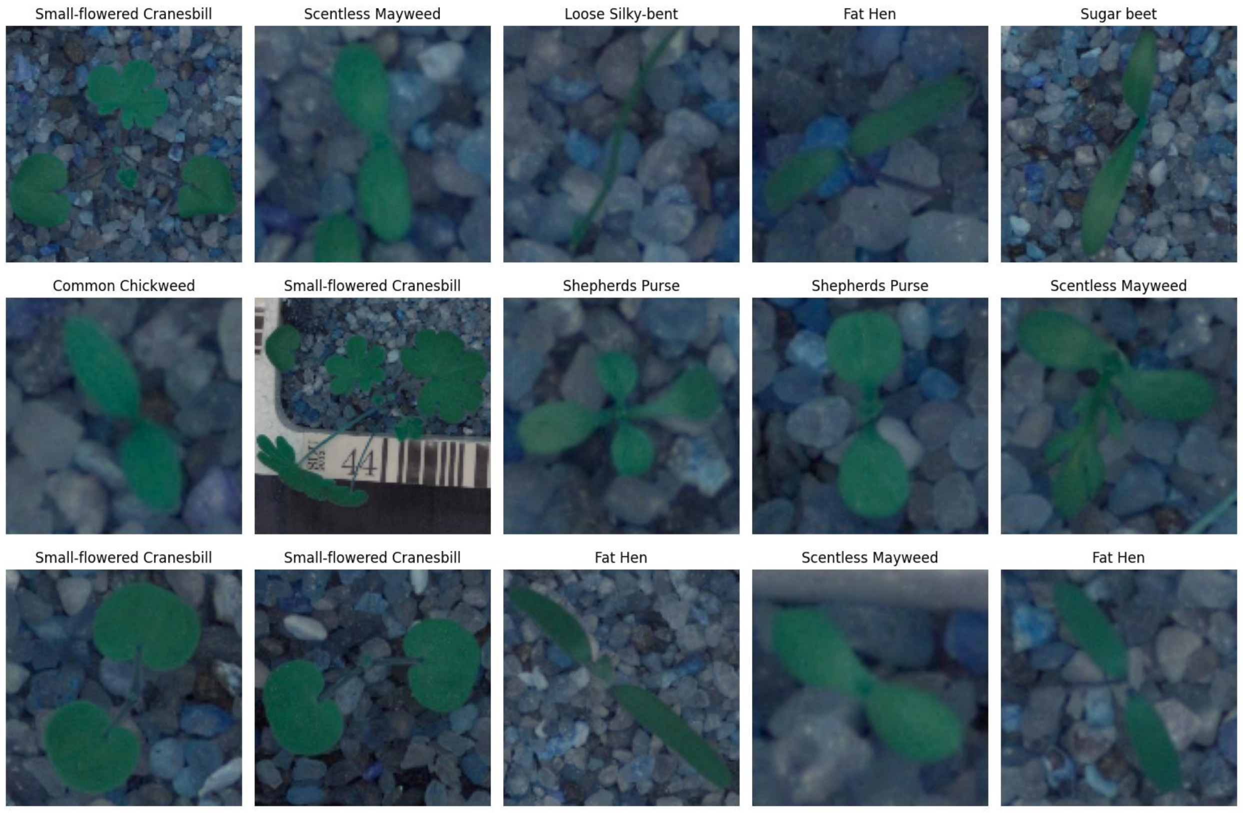
Below is the code for a sample of 15 images from this project, along with their labels.



```
#Displaying 15 random images with labels
num_smpls = 15
indices = np.random.choice(range(imgs.shape[0]), num_smpls, replace=False)
smpl_imgs = imgs[indices]
smpl_lbls = lbls['Label'].iloc[indices].values

#Plotting the samples
plt.figure(figsize=(15, 10))
for i, (img, lbl) in enumerate(zip(smpl_imgs, smpl_lbls)):
    plt.subplot(3, 5, i + 1)
    plt.imshow(img.astype('uint8'))
    plt.title(lbl)
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Below is a sample of 15 images.

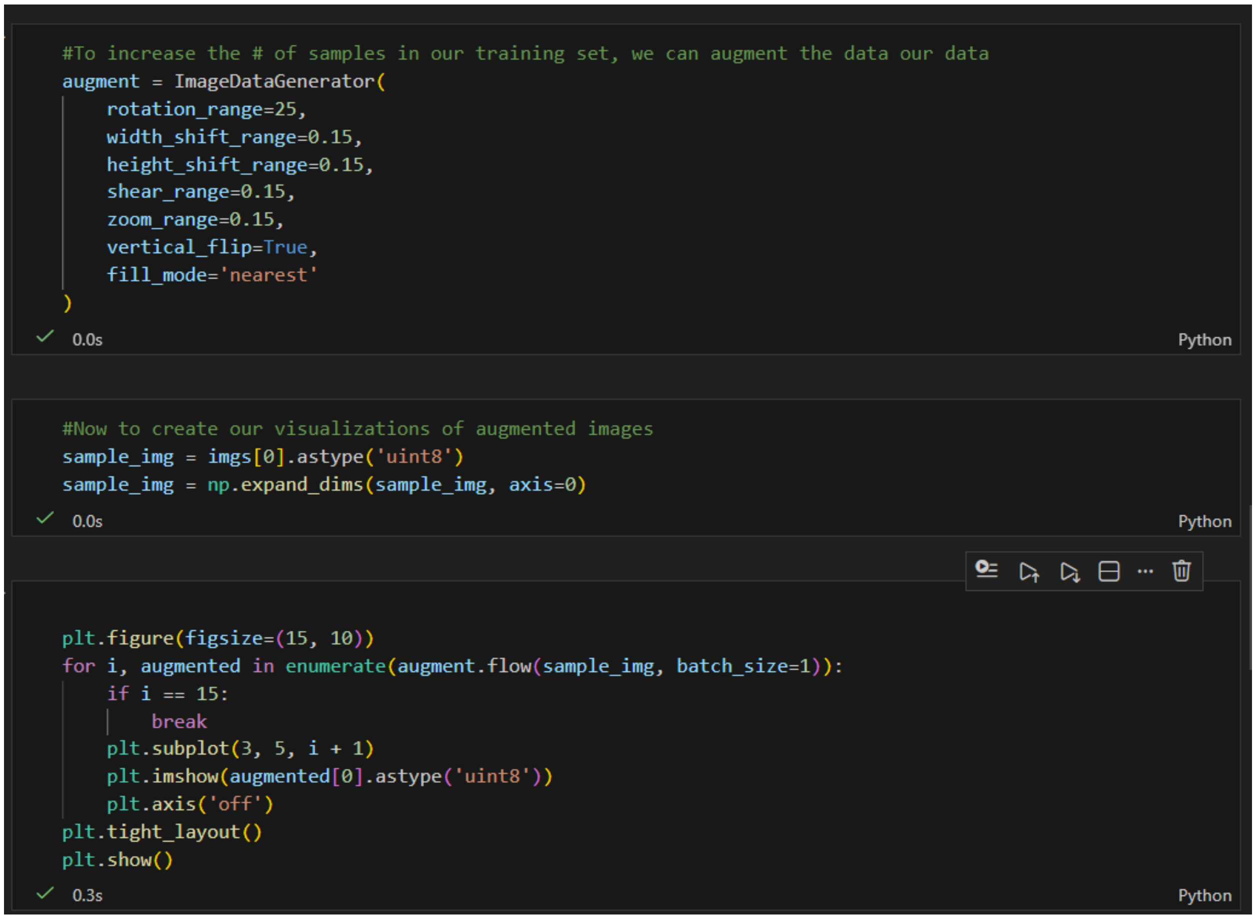


B2: Data Preprocessing

To create a larger training set, we can augment our data by applying random transformations to the sample we were given and treating the resulting points as new data points. Examples of image augmentation include rotating, adjusting brightness, and flipping images vertically or horizontally. By using these augmentations, we can increase the number of samples used by our model, making it more accurate and less prone to overfitting. For my preprocessing, I used the following augmentation possibilities;

- Randomly rotating the images by up to 25 degrees
- Randomly shifting the images horizontally or vertically by 15%
- Apply shear transformations up to 15%
- Zoom in on images up to 15%
- Flip the images vertically

The data was augmented with the code below:



```python
#To increase the # of samples in our training set, we can augment the data our data
augment = ImageDataGenerator(
    rotation_range=25,
    width_shift_range=0.15,
    height_shift_range=0.15,
    shear_range=0.15,
    zoom_range=0.15,
    vertical_flip=True,
    fill_mode='nearest'
)
```

✓ 0.0s Python

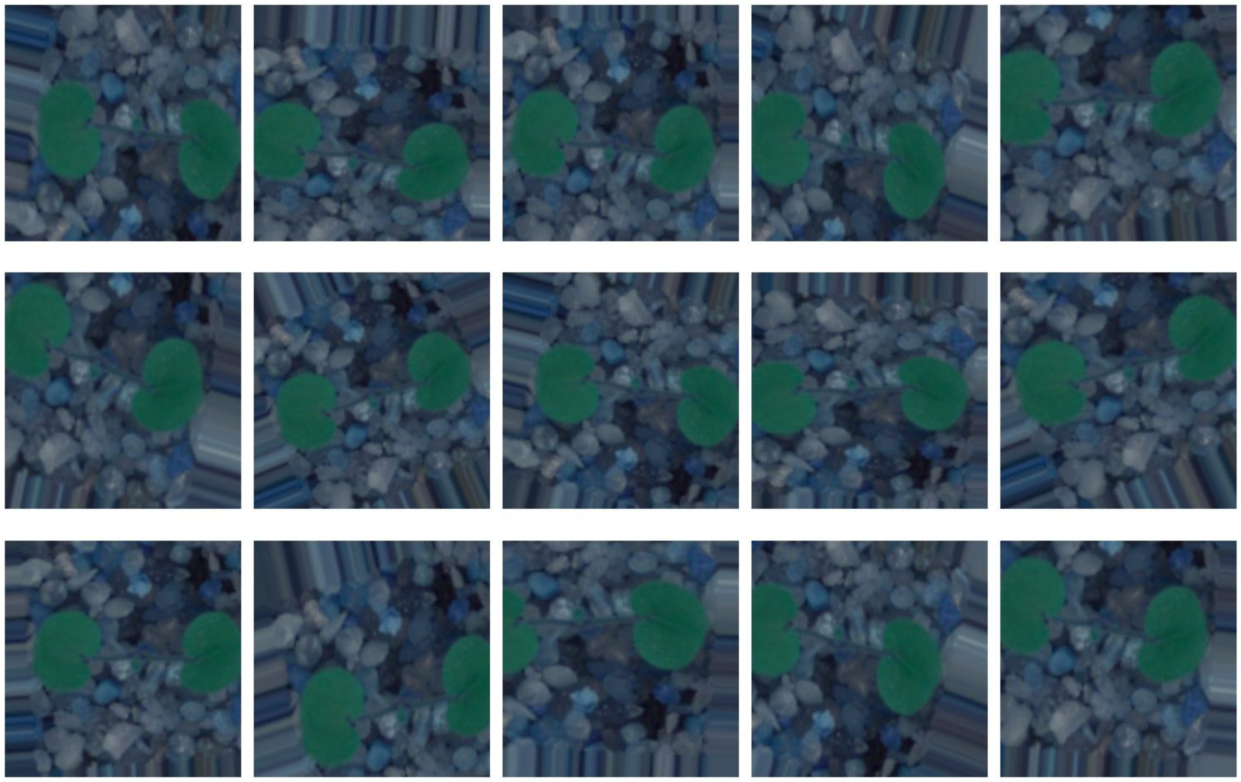
```python
#Now to create our visualizations of augmented images
sample_img = imgs[0].astype('uint8')
sample_img = np.expand_dims(sample_img, axis=0)
```

✓ 0.0s Python

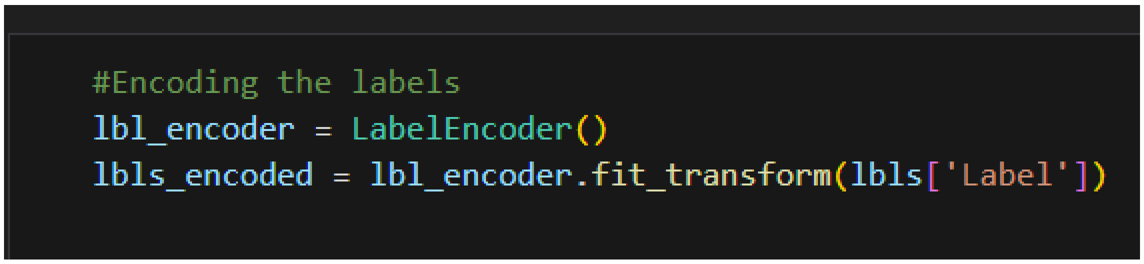
```python
plt.figure(figsize=(15, 10))
for i, augmented in enumerate(augment.flow(sample_img, batch_size=1)):
    if i == 15:
        break
    plt.subplot(3, 5, i + 1)
    plt.imshow(augmented[0].astype('uint8'))
    plt.axis('off')
plt.tight_layout()
plt.show()
```

✓ 0.3s Python

Below is a sample of 15 images with transformations applied:

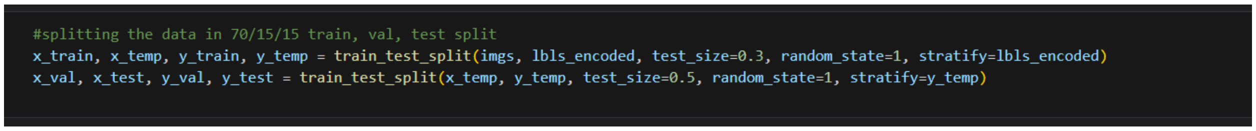


Then, I normalized the pixels. The pixel values were converted from 0-255 to 0-1. They will still have the same proportional value relative to each other, but the numbers are smaller and should be easier to process. This was done with the code below.



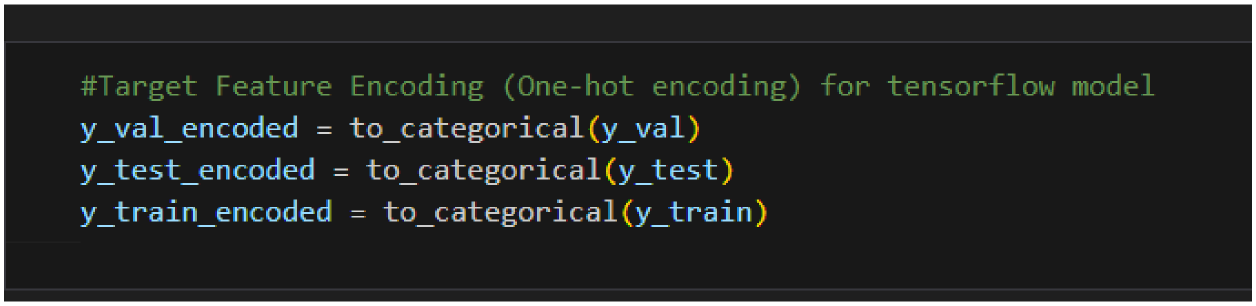
```
#Encoding the labels
lbl_encoder = LabelEncoder()
lbls_encoded = lbl_encoder.fit_transform(lbls['Label'])
```

I then decided to split the data into a 70/15/15 training, validation, and testing split. The code to do so is below.



```
#splitting the data in 70/15/15 train, val, test split
x_train, x_temp, y_train, y_temp = train_test_split(imgs, lbls_encoded, test_size=0.3, random_state=1, stratify=lbls_encoded)
x_val, x_test, y_val, y_test = train_test_split(x_temp, y_temp, test_size=0.5, random_state=1, stratify=y_temp)
```

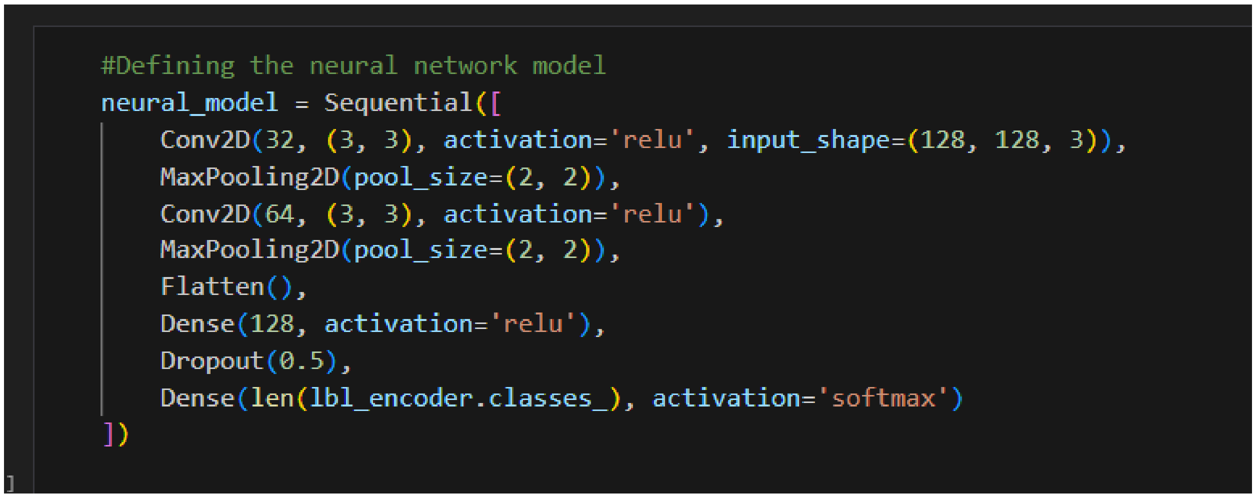
Finally, I encoded the data from the training, validation, and testing splits. This was done because the data needs to be Target Feature Encoded to be evaluated using TensorFlow.



```
#Target Feature Encoding (One-hot encoding) for tensorflow model
y_val_encoded = to_categorical(y_val)
y_test_encoded = to_categorical(y_test)
y_train_encoded = to_categorical(y_train)
```

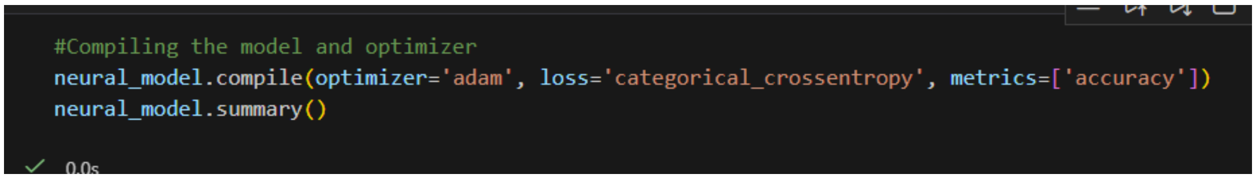
C1: Model Summary Output

I defined the neural network model using the code below.



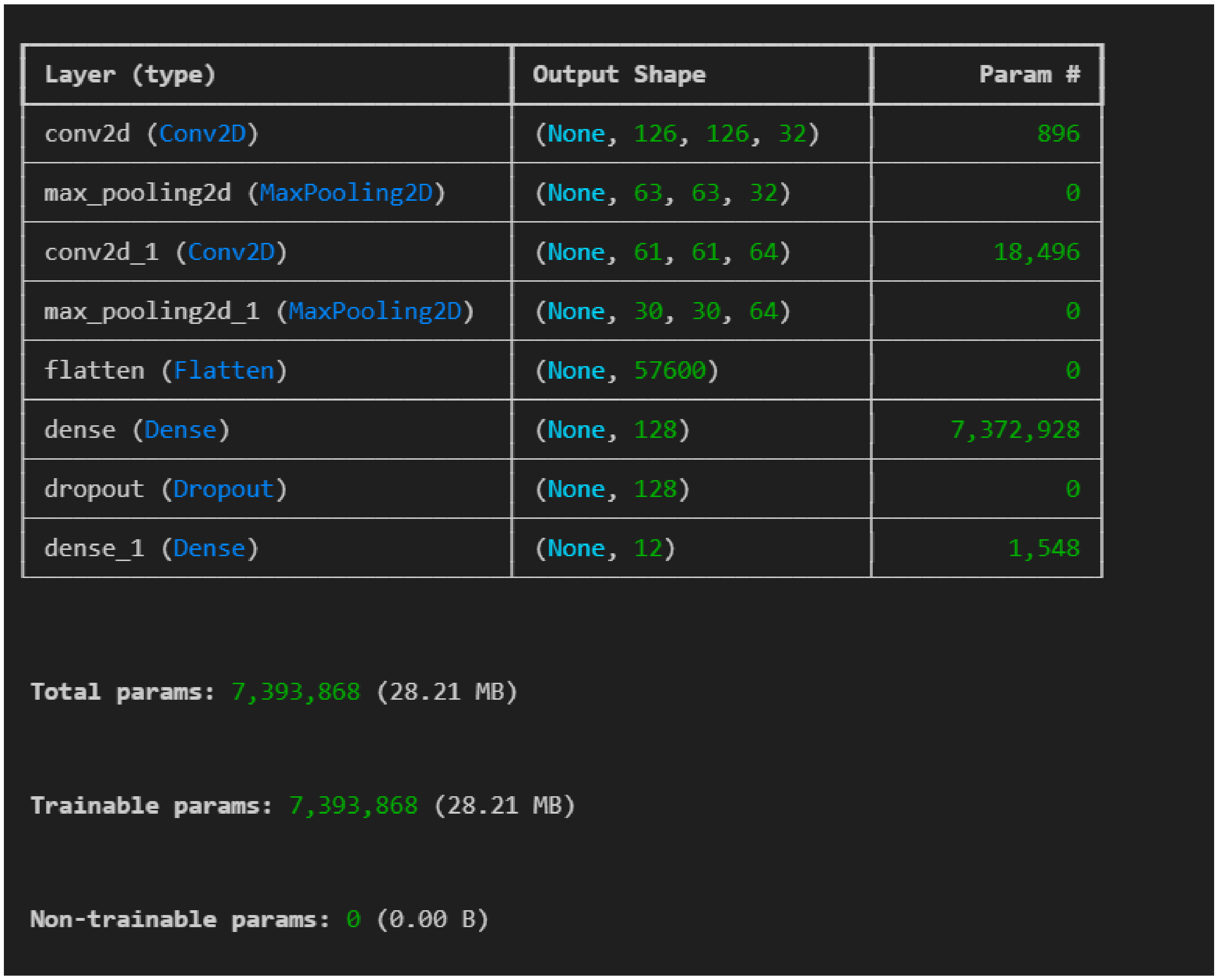
```
#Defining the neural network model
neural_model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(128, 128, 3)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(len(lbl_encoder.classes_), activation='softmax')
])
```

Then, I could compile the model and optimizer using the following code.



```
#Compiling the model and optimizer
neural_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
neural_model.summary()
```

```
0.0s
```



```
┃ 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)
```

From the model, we can identify that there are 8 layers, which are listed below with their layer type

#	Layer name	Layer type
1	conv2d	Conv2D
2	max_pooling2d	MaxPooling2D
3	conv2d_1	Conv2D
4	max_pooling2d_1	MaxPooling2D
5	flatten	Flatten
6	dense	Dense
7	dropout	Dropout
8	dense_1	Dense

The number of nodes per layer is described in the table below.

Layer	Output shape	Nodes/units explanation
Conv2D (1)	(126, 126, 32)	32 feature maps
MaxPooling2D (1)	(63, 63, 32)	32 pooled feature maps
Conv2D (2)	(61, 61, 64)	64 feature maps
MaxPooling2D (2)	(30, 30, 64)	64 pooled feature maps
Flatten	-57,600	57,600 nodes
Dense (1)	-128	128 neurons
Dropout	-128	128 neurons
Dense (output)	-12	12 neurons

The total number of parameters including weights and biases are included below.

Layer	Parameters
Conv2D (1)	896
Conv2D (2)	18,496
Dense (1)	7,372,928
Dense (output)	1,548
Total	**7,393,868**

The activation functions in all layers are included below.

Layer	Activation function
Conv2D (1)	**ReLU**
MaxPooling2D (1)	**None**
Conv2D (2)	**ReLU**
MaxPooling2D (2)	**None**
Flatten	**None**
Dense (128)	**ReLU**
Dropout	**None**
Dense (output)	**Softmax**

C2: Backpropagation and Hyperparameters justification

Backpropagation operates in my model by adjusting model weights to reduce classification error during training. In the forward pass (Conv2D, MaxPooling, Flatten + Dense, Output Dense) input images pass through convolutional and pooling layers to extract features, then through the SoftMax output producing class probabilities. Then during the backward pass, gradients of the loss are propagated backwards through the network, updating weights in both dense and convolutional layers. Then, the Adam optimizer uses these gradients to update the weights, improving the model's performance for each successive epoch.

The loss function for this model is categorical cross-entropy. This is appropriate for the model because the task is multiclass with 12 possible outputs and uses one-hot encoded labels. It measures how well the distribution from the Softmax layer matches the true class distribution. When dealing with several classes, it can be especially effective.

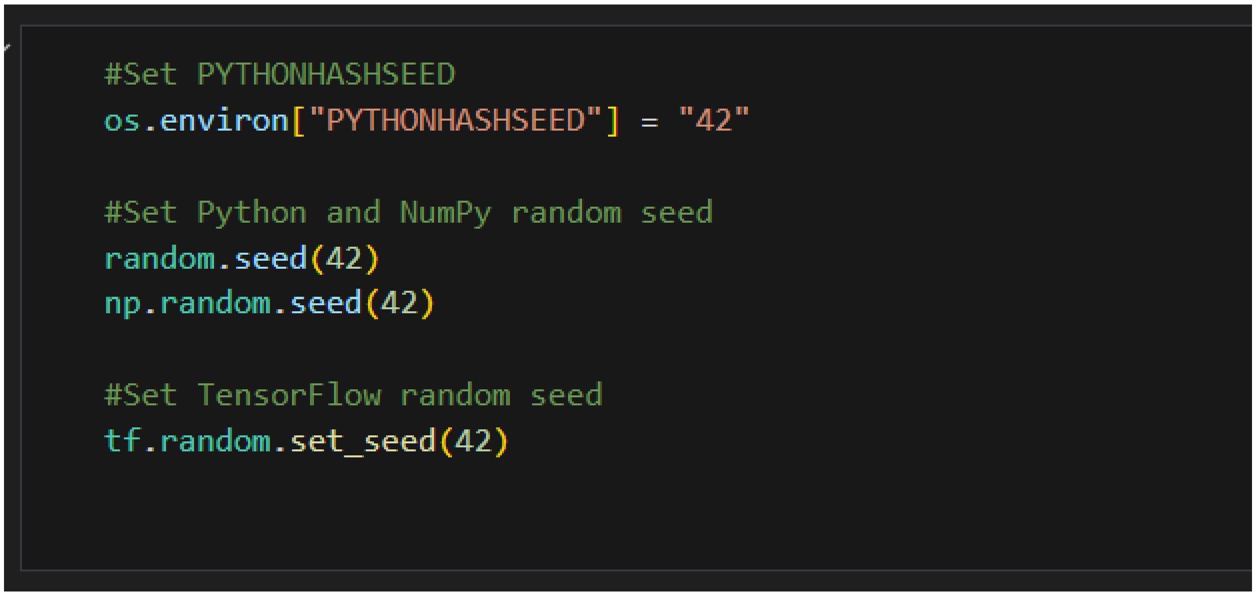
The optimizer for this model is the Adam optimizer. This is appropriate for the model because it can efficiently handle a large number of parameters and noisy gradients. It adapts the learning rate for each parameter individually, leading to faster and more stable convergence. For a model with a large dataset, this optimizer is effective.

The model uses the Adam optimizer's default learning rate of 0.001, which provides a balance between convergence speed and training stability. This rate is appropriate for the model's depth and large dataset size.

The stopping criteria is using EarlyStopping. This criteria, monitors validation loss to prevent overfitting as training progresses. With patience set to 5, training is stopped when the validation loss stops for 5 epochs in a row, which indicates the model is performing worse and will not do well. Having a stopping criteria saves computation time when the model stops improving

C3: Training the Model Using Best Practices

A fixed random seed was set to ensure reproducibility using the code below.

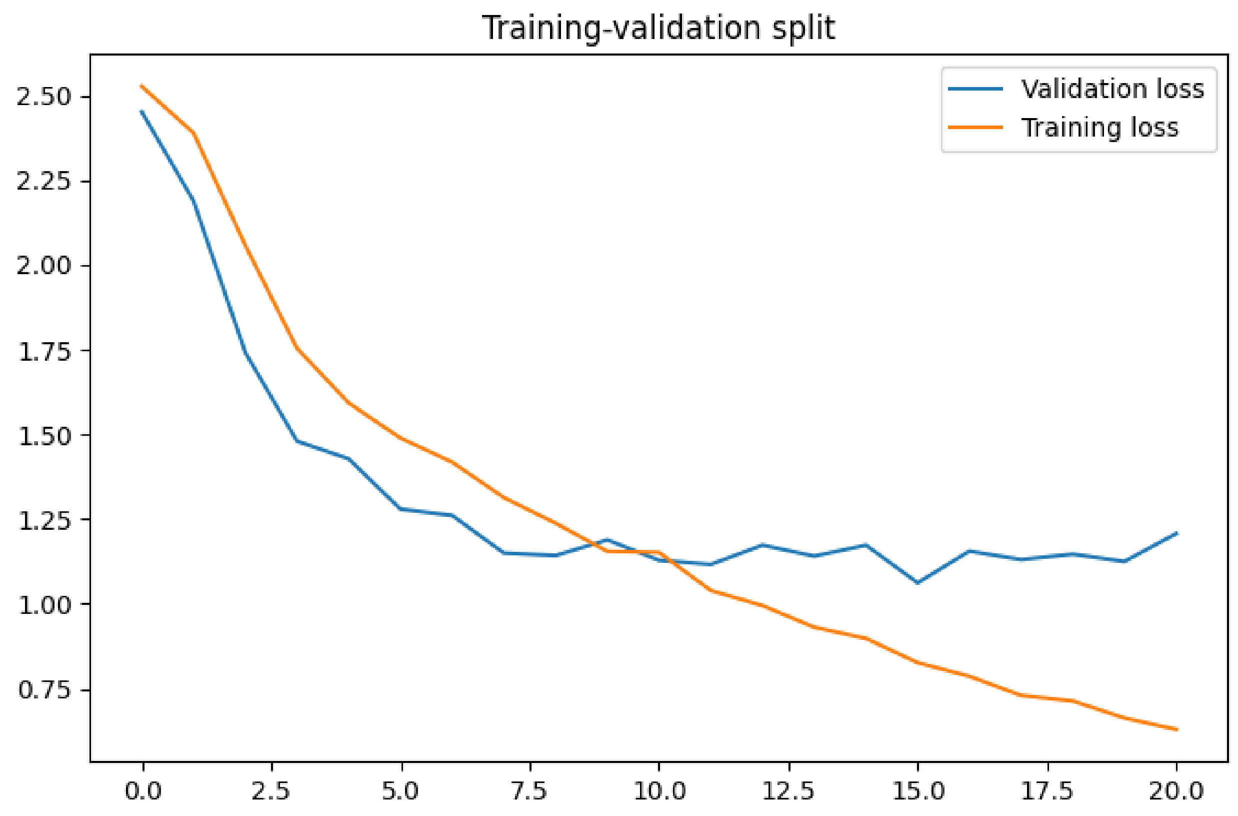


```
#Set PYTHONHASHSEED
os.environ["PYTHONHASHSEED"] = "42"

#Set Python and NumPy random seed
random.seed(42)
np.random.seed(42)

#Set TensorFlow random seed
tf.random.set_seed(42)
```

I chose to use a training-validation split to evaluate the model performance. Below is the training-validation split plotted for easier analysis.

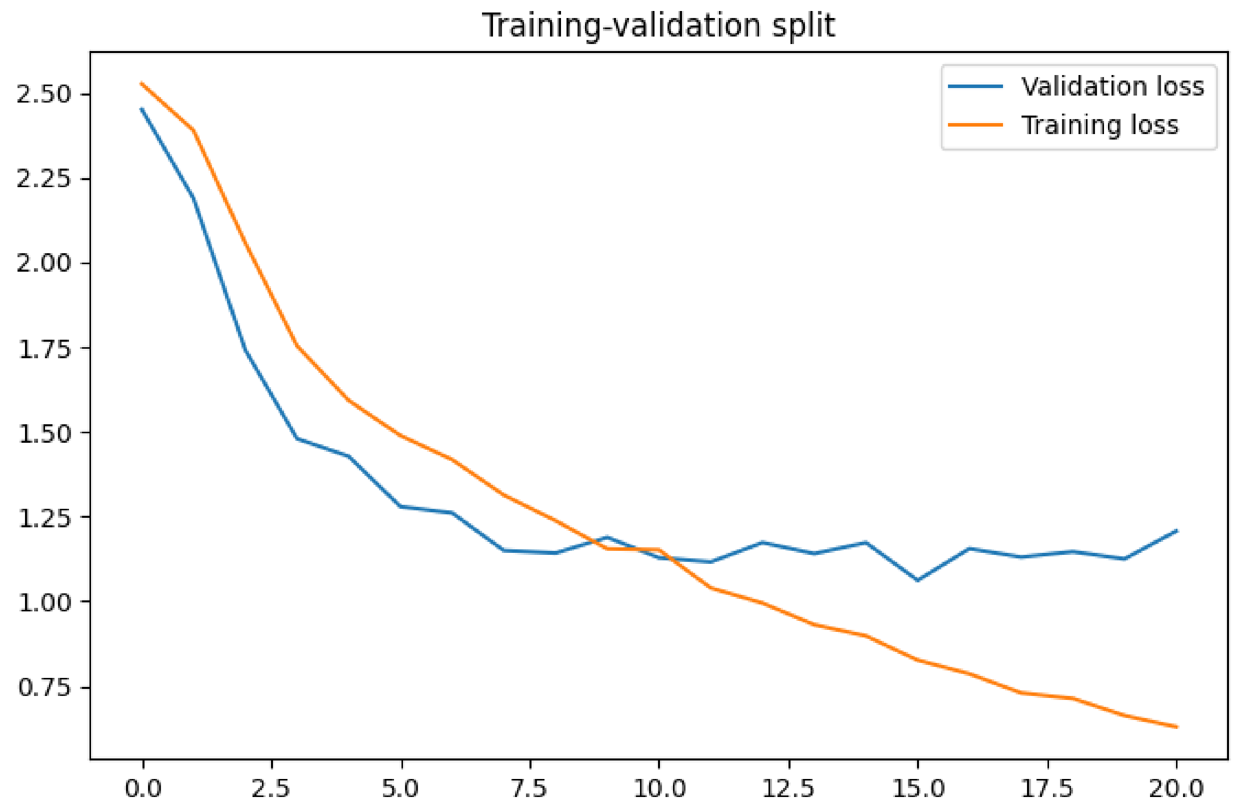


When the validation loss stops decreasing and starts to plateau or increase, the model is at its optimal training epoch. At that point, the model has not yet begun overfitting; instead, it has achieved as much accuracy as it can through training. According to the graph, at about epoch 9, the validation and training loss start to diverge. This is where our model stops performing better on the validation set. This is when early stopping kicks in. Without early stopping, our model might become better at predicting the test or training sets, at the expense of being worse at predicting new data.

A training-validation split was chosen to evaluate this neural network because it provides a reliable estimate of the model's ability to generalize unseen data. Assessing performance on a separate validation set helps to detect overfitting and ensures that observed performance is not due to memorization. The split allows for fair comparison between different hyperparameter choices, as well as the use of techniques such as early stopping, which was used in this neural network.

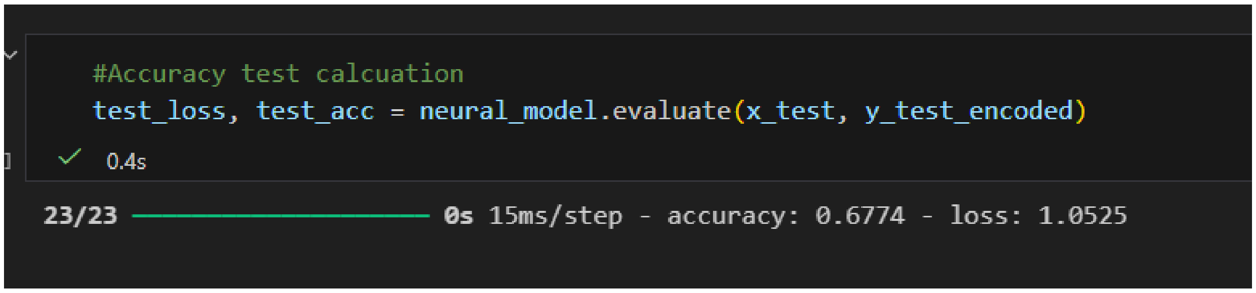
D1. Evaluate the Training and Validation Process

The training-validation split was evaluated in the previous section. It will be copied down here.



When the validation loss stops decreasing and starts to plateau or increase, the model is at its optimal training epoch. At that point, the model has not yet begun overfitting; instead, it has achieved as much accuracy as it can through training. According to the graph, at about epoch 9, the validation and training loss start to diverge. This is where our model stops performing better on the validation set. This is when early stopping kicks in. Without early stopping, our model might become better at predicting the test or training sets, at the expense of being worse at predicting new data.

The accuracy and loss were tested using the code below.



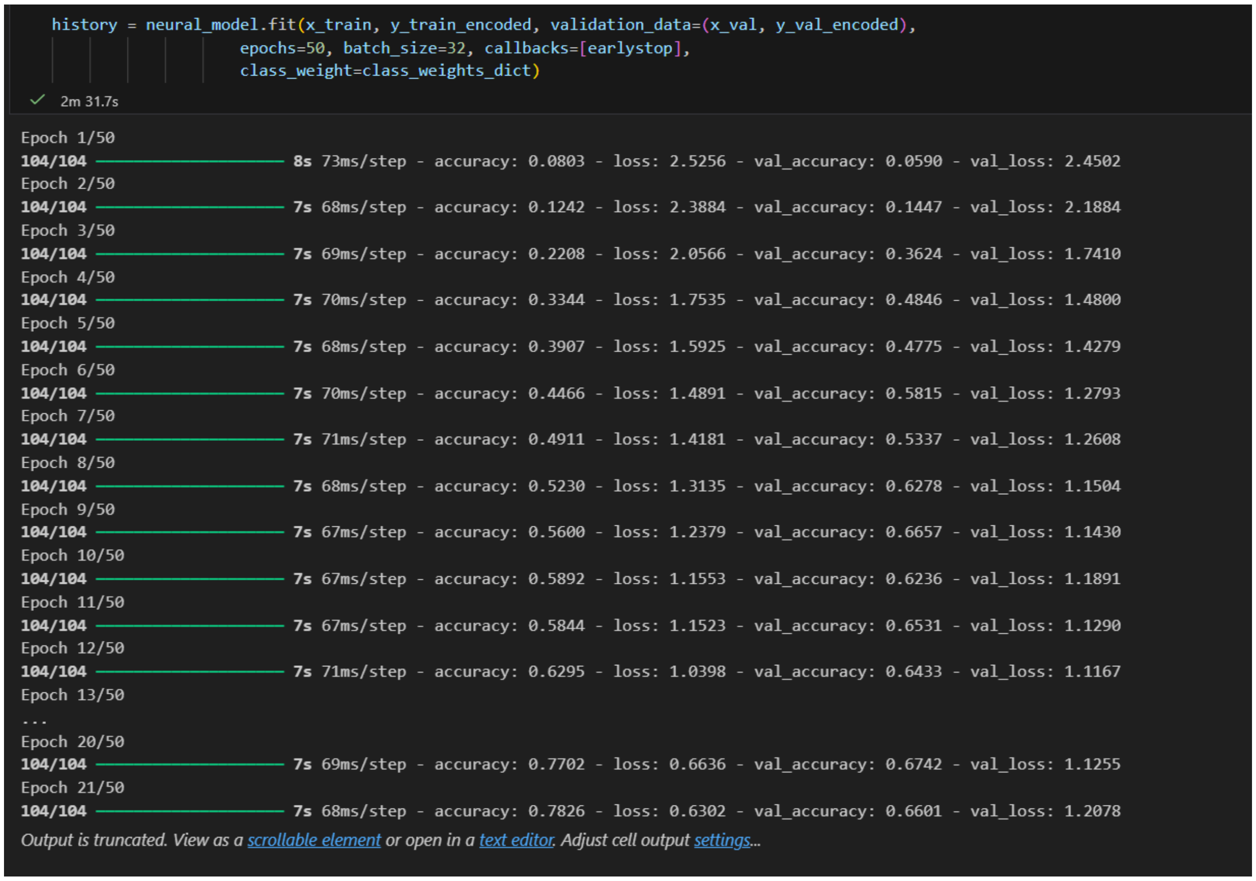
```
#Accuracy test calcuation
test_loss, test_acc = neural_model.evaluate(x_test, y_test_encoded)
✓ 0.4s
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.6774 - loss: 1.0525
```

The test accuracy was 0.6774, and the loss was 1.0525. The accuracy suggests that the model has learned patterns from the training data but still has room for improvement. With an accuracy of 67.7% across 12 classes, it is much better than random guessing. A loss of 1.05 implies that the model is often uncertain or overconfident in incorrect predictions. The combination of high loss and moderate accuracy indicates limited generalization, but, as with accuracy, using the model for identification is much better than guessing.

The model fits well, and the hyperparameter tuning to prevent overfitting was largely successful. Creating a dropout layer with a 0.5 (50%) rate prevented overfitting by reducing the number of neurons in the fully connected layer. Using EarlyStopping with patience = 5 caused the model to stop training after the data performed worse when seeing

new data. This helped to prevent overfitting and cut down on computational time by stopping the training when the model was no longer improving. Class weights were used due to an uneven distribution of plants. This resulted in a balanced strategy being applied during training to address class imbalance, ensuring that classes with fewer images contributed to the loss function. The hyperparameter choices were validated by improved stability in validation loss and contributed to more reliable generalization behavior.

As mentioned above, I used EarlyStopping with patience = 5. This caused the model to stop training after the data performed worse when seeing new data. Even if the model is improving on other datasets, if it is not performing well on new data, such as the validation set, then the model is not truly improving. Thus, using the stopping criteria helped prevent overfitting and reduced computational time by stopping training when the model was no longer improving. For the neural network model, I chose to use 50 epochs. Below is a screenshot of the final epoch training.



```
history = neural_model.fit(x_train, y_train_encoded, validation_data=(x_val, y_val_encoded),
                           epochs=50, batch_size=32, callbacks=[earlystop],
                           class_weight=class_weights_dict)
✓ 2m 31.7s

Epoch 1/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 8s 73ms/step - accuracy: 0.0803 - loss: 2.5256 - val_accuracy: 0.0590 - val_loss: 2.4502
Epoch 2/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 68ms/step - accuracy: 0.1242 - loss: 2.3884 - val_accuracy: 0.1447 - val_loss: 2.1884
Epoch 3/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 69ms/step - accuracy: 0.2208 - loss: 2.0566 - val_accuracy: 0.3624 - val_loss: 1.7410
Epoch 4/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 70ms/step - accuracy: 0.3344 - loss: 1.7535 - val_accuracy: 0.4846 - val_loss: 1.4800
Epoch 5/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 68ms/step - accuracy: 0.3907 - loss: 1.5925 - val_accuracy: 0.4775 - val_loss: 1.4279
Epoch 6/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 70ms/step - accuracy: 0.4466 - loss: 1.4891 - val_accuracy: 0.5815 - val_loss: 1.2793
Epoch 7/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 71ms/step - accuracy: 0.4911 - loss: 1.4181 - val_accuracy: 0.5337 - val_loss: 1.2608
Epoch 8/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 68ms/step - accuracy: 0.5230 - loss: 1.3135 - val_accuracy: 0.6278 - val_loss: 1.1504
Epoch 9/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 67ms/step - accuracy: 0.5600 - loss: 1.2379 - val_accuracy: 0.6657 - val_loss: 1.1430
Epoch 10/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 67ms/step - accuracy: 0.5892 - loss: 1.1553 - val_accuracy: 0.6236 - val_loss: 1.1891
Epoch 11/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 67ms/step - accuracy: 0.5844 - loss: 1.1523 - val_accuracy: 0.6531 - val_loss: 1.1290
Epoch 12/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 71ms/step - accuracy: 0.6295 - loss: 1.0398 - val_accuracy: 0.6433 - val_loss: 1.1167
Epoch 13/50
...
Epoch 20/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 69ms/step - accuracy: 0.7702 - loss: 0.6636 - val_accuracy: 0.6742 - val_loss: 1.1255
Epoch 21/50
104/104 ━━━━━━━━━━━━━━━━━━━━ 7s 68ms/step - accuracy: 0.7826 - loss: 0.6302 - val_accuracy: 0.6601 - val_loss: 1.2078
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

The final epoch training shows accuracy: 0.7826 - loss: 0.6302 - val_accuracy: 0.6601 - val_loss: 1.2078. What this indicates about the overall model fitness is that the model is overfitting. The divergence in training accuracy and loss and that of validation shows that the model is fitting the training data well but is struggling to generalize those learned patterns to unseen validation data. These results indicate the model demonstrates adequate learning capacity but insufficient generalization.

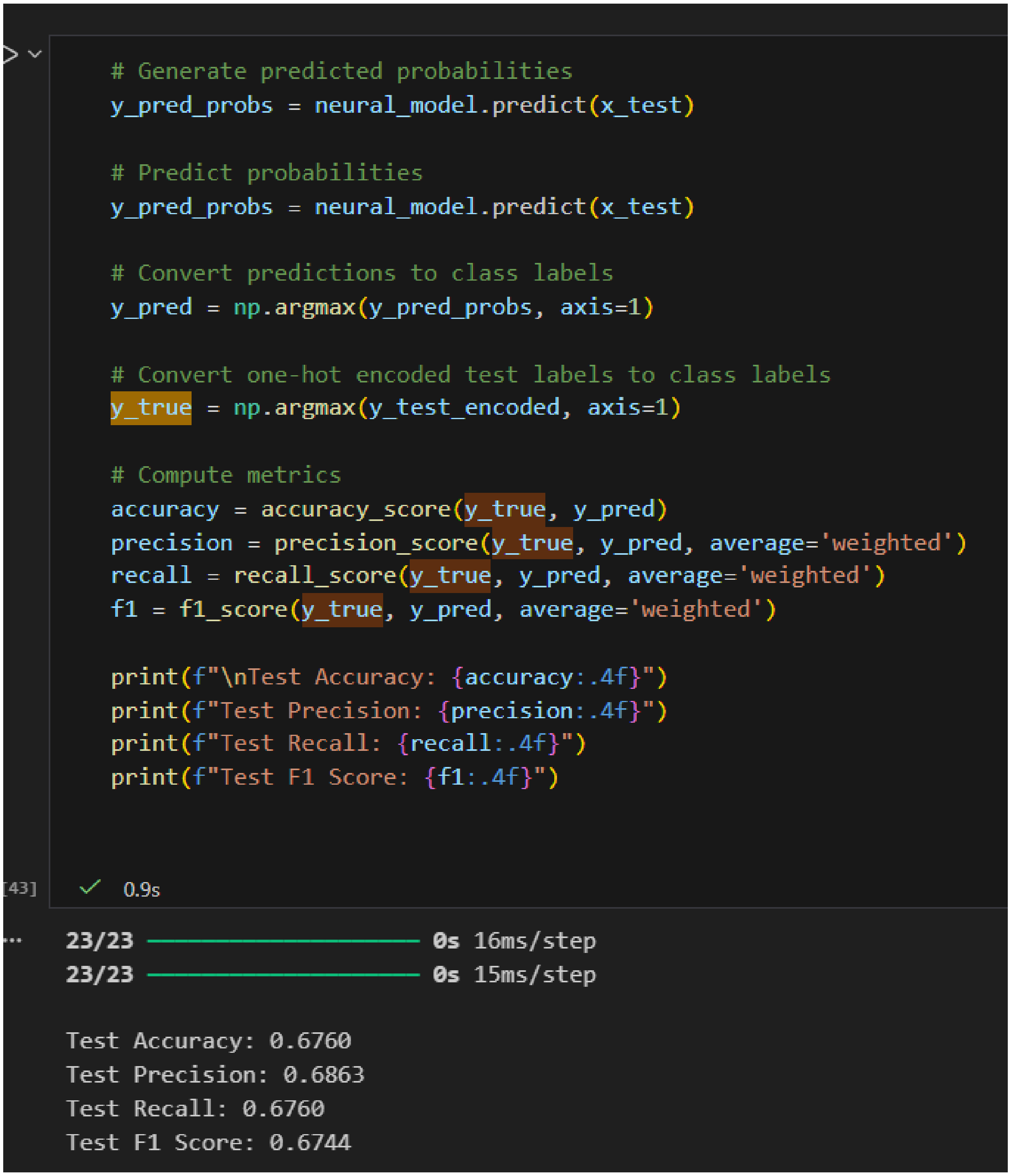
D2. Model Overfitting and Corrective Action

In the previous section, it was determined that the model is overfitting. Some corrective actions that can be taken to prevent this include

- Adding or increasing dropout to force the network to rely on more distributed representations rather than memorizing training data.
- Simplifying the model architecture by reducing the number of layers or neurons, which would make it harder for the network to overfit the training set.
- Increasing the size or diversity of the training data to help the model learn more generalizable features.
- Setting a stricter early stopping by reducing the patience value. This is done to stop training earlier, before overfitting intensifies.
- Further refining hyperparameters or experimenting with different optimizers, which can lead to smoother convergence and better generalization.

D3. Final Predictive Performance

The accuracy, precision, recall, and F1 score of the model were calculated using the code below.



```
# Generate predicted probabilities
y_pred_probs = neural_model.predict(x_test)

# Predict probabilities
y_pred_probs = neural_model.predict(x_test)

# Convert predictions to class labels
y_pred = np.argmax(y_pred_probs, axis=1)

# Convert one-hot encoded test labels to class labels
y_true = np.argmax(y_test_encoded, axis=1)

# Compute metrics
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, average='weighted')
recall = recall_score(y_true, y_pred, average='weighted')
f1 = f1_score(y_true, y_pred, average='weighted')

print(f"\nTest Accuracy: {accuracy:.4f}")
print(f"Test Precision: {precision:.4f}")
print(f"Test Recall: {recall:.4f}")
print(f"Test F1 Score: {f1:.4f}")
```

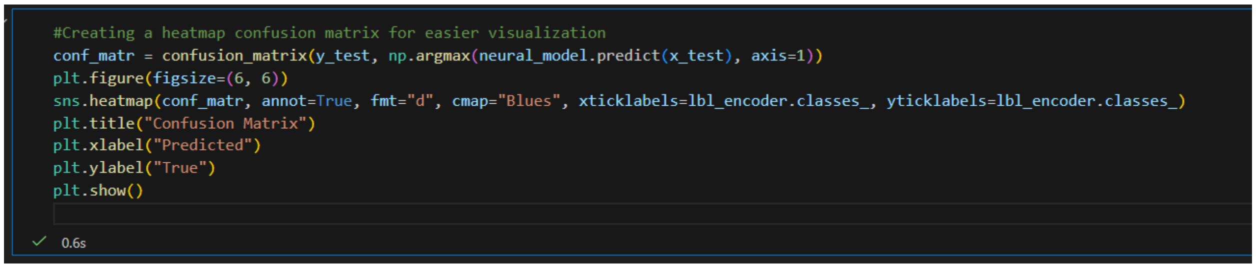
[43] ✓ 0.9s

```
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
23/23 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step

Test Accuracy: 0.6760
Test Precision: 0.6863
Test Recall: 0.6760
Test F1 Score: 0.6744
```

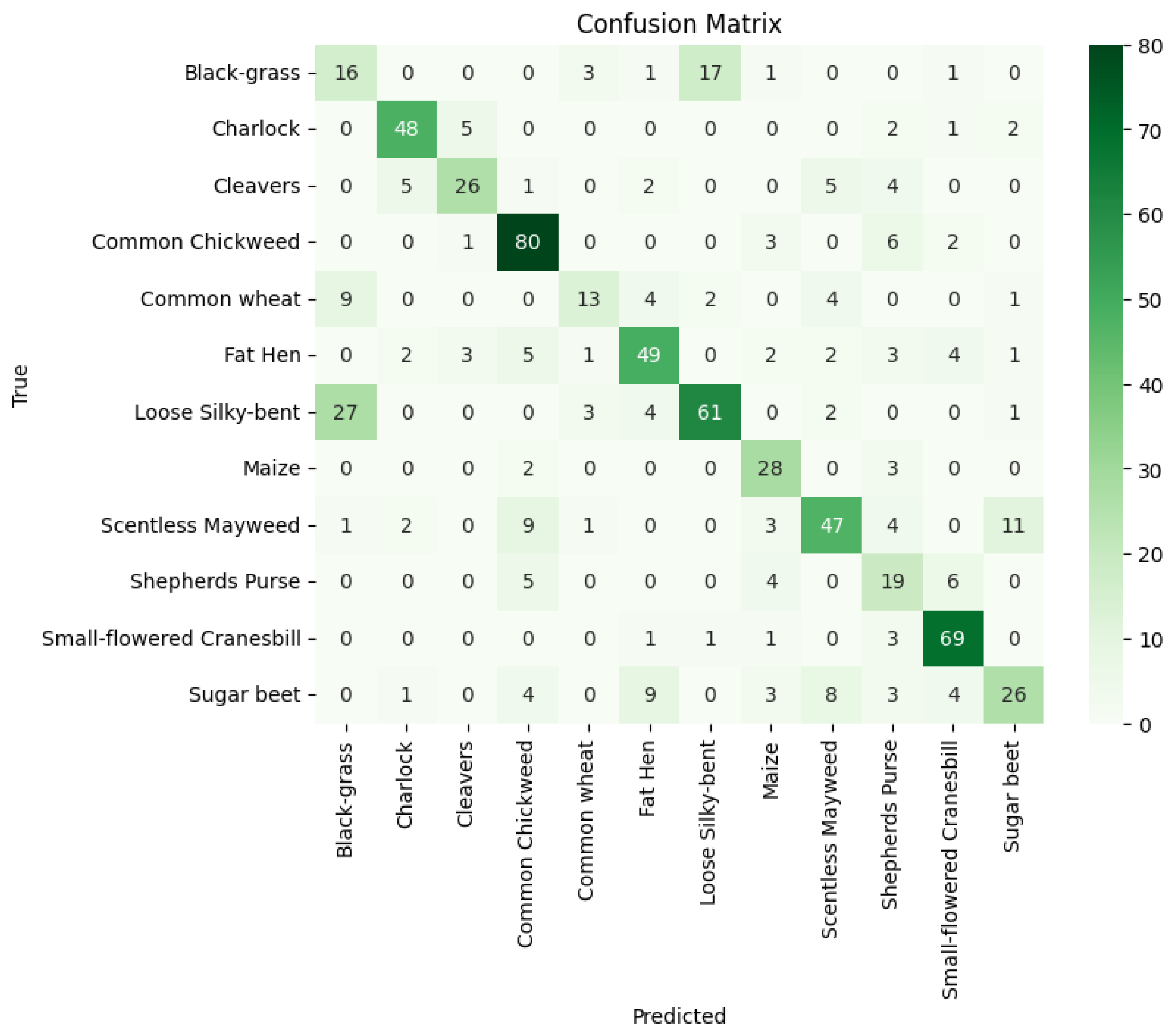
From the code, we can conclude that the test accuracy was 0.6760, the test precision was 0.6863, the test recall was 0.6760, and the F1 score was 0.6744. A test accuracy of 0.6760 shows that the model correctly classifies roughly two-thirds of the unseen test samples, which is much better than guessing but not strong enough to be considered highly reliable. The precision (0.6863) suggests that when the model predicts a class, it is correct in most cases. The recall (0.6760) closely matches the accuracy, showing that the model identifies a similar proportion of true instances across classes, but still fails to capture a substantial number of correct cases. The F1 score (0.6744) confirms that performance is consistent but modest, with no single metric substantially outperforming the others. These results suggest that the model has learned meaningful patterns but still struggles to generalize fully. Further refinement to the model is necessary to achieve better predictive performance.

The code used to create a heatmap confusion matrix is included below.



```
#Creating a heatmap confusion matrix for easier visualization
conf_matr = confusion_matrix(y_test, np.argmax(neural_model.predict(x_test), axis=1))
plt.figure(figsize=(6, 6))
sns.heatmap(conf_matr, annot=True, fmt="d", cmap="Blues", xticklabels=lbl_encoder.classes_, yticklabels=lbl_encoder.classes_)
plt.title("Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

Converting the confusion matrix to a heatmap allows for easier visualization compared to one that is just text. The resulting confusion matrix is below.


Confusion Matrix
True
Predicted
80
70
60
50
40
30
20
10
0

True \ Predicted	Black-grass	Charlock	Cleavers	Common Chickweed	Common wheat	Fat Hen	Loose Silky-bent	Maize	Scentless Mayweed	Shepherds Purse	Small-flowered Cranesbill	Sugar beet
Black-grass	16	0	0	0	3	1	17	1	0	0	1	0
Charlock	0	48	5	0	0	0	0	0	0	2	1	2
Cleavers	0	5	26	1	0	2	0	0	5	4	0	0
Common Chickweed	0	0	1	80	0	0	0	3	0	6	2	0
Common wheat	9	0	0	0	13	4	2	0	4	0	0	1
Fat Hen	0	2	3	5	1	49	0	2	2	3	4	1
Loose Silky-bent	27	0	0	0	3	4	61	0	2	0	0	1
Maize	0	0	0	2	0	0	0	28	0	3	0	0
Scentless Mayweed	1	2	0	9	1	0	0	3	47	4	0	11
Shepherds Purse	0	0	0	5	0	0	0	4	0	19	6	0
Small-flowered Cranesbill	0	0	0	0	0	1	1	1	0	3	69	0
Sugar beet	0	1	0	4	0	9	0	3	8	3	4	26

The X-axis is the predicted class of the photos, and the Y-axis is the true class. When the predicted class matches that of the y-axis, we get accurate predictions, which creates the diagonal line we see present in the confusion matrix. There are some cases where the predicted class did not match the true class. Most notably, the model would confuse Loose Silky-bent with Black Grass and vice versa. This could be caused by issues with the data, the training, hyperparameter tuning, or because the plants look too similar. Further refinement to the model will be required.

E1. Summarizing the Neural Network Design and Behavior

The neural network is designed to perform multiclass image classification, with the purpose of automatically assigning an input image to one of 12 predefined categories. The architecture is a convolutional neural network (CNN), selected because CNNs are well-suited for image data due to their ability to learn spatial hierarchies of features while reducing sensitivity to translation and noise. The model consists of two Conv2D layers with ReLU activation to extract low- and mid-level visual features, each followed by MaxPooling layers to downsample feature maps and reduce computational complexity. A Flatten layer converts the learned spatial features into a one-dimensional vector, which is processed by a fully connected Dense layer with ReLU activation to learn high-level class-specific representations, while a Dropout layer provides regularization by randomly disabling neurons to reduce overfitting. The final Dense layer uses a Softmax activation to output a probability distribution over the 12 classes, where each value represents the model's confidence for a given class, and the highest probability corresponds to the predicted label.

The network inputs are RGB images of size 128×128×3, and the outputs are 12-element probability vectors representing class membership. Training behavior is governed by the categorical cross-entropy loss function, which is appropriate for multiclass classification with one-hot encoded labels, and the Adam optimizer with its default learning rate of 0.001 to provide stable and efficient gradient-based optimization. The model is trained using a batch size of 32 for up to 50 epochs, with early stopping based on validation loss to prevent overfitting and restore the best-performing weights. Class weights are applied during training to address class imbalance by increasing the loss contribution of underrepresented classes. Testing behavior involved evaluating the trained model on a held-out test dataset, which showed test accuracy of

0.6760, test precision of 0.6863, test recall of 0.6760, and an F1 score of 0.6744. These results suggest that the model has learned meaningful patterns but still struggles to generalize fully. Further refinement of the model is necessary to improve predictive performance.

E2. Neural Network Functionality

The neural network's functionality was successful at identifying the species of a plant seedling using images of seedlings as training data. It is functional at predicting 10 of the 12 plants, with the 2 remaining plants being able to be predicted with some hiccups. The architecture of the neural network helped contribute to effectiveness. Use of convolutional layers helped the model detect patterns in edges and texture, while max pooling layers helped reduce dimension and improve computation efficiency. Then the flattening layer helped transform the data into an array that could be more easily processed. Then the Dense layers combined the data to create predictions, and finally the Dropout layer helped prevent overfitting. These layers all worked together to create a neural network that had a test accuracy of 0.6760, a test precision of 0.6863, a test recall of 0.6760, and a F1 score of 0.6744. These results suggest that the model has learned meaningful patterns but still struggles to generalize fully. Further refinement to the model is necessary to achieve better predictive performance.

E3. Suggestions for Improvement

The model could be improved by providing more data so that each of the classes has the same number of samples. Right now, each class is weighted to make it seem like the size of each class is similar. Although this does work, having more data would be an improvement. The model did have difficulty differentiating between the Loose Silky-Bent and the Black Grass classes. The Loose Silky-Bent was the most over-represented plant and thus the most trained plant, and then the Black-Grass, which only accounted for 5% of the training data. Reducing the number of samples for Loose Silky-Bent and increasing the number of samples for Black Grass would help the model differentiate between the two more easily. Further tuning of the hyperparameters, using better photos, and possibly reconfiguring the neural network layers are all suggestions for improving the model and further refining the process.

E4. Recommended Course of Action

My recommendation is for a botanist using this model to use it as a shortcut to identify plants, but to double-check Loose Silky-bent and Black-Grass. If the model identifies either of those two plants, it should be double-checked to ensure accuracy. This model will save the botanists time and energy when identifying the plants.

References

No sources were used.