Biology Project Report

Semantic Segmentation of Herbarium Specimen Sheets with a Branched CNN/DNN Model

James Della-Giustina

IDNM 680

Fisher College of Science and Mathematics

Dr. David Hearn

May 23, 2023

1. Group Work Summary

The goal of the Biology CRE project is to create a program that will take an image of a plant specimen and output the image with specific parts highlighted and classified. For instance, any arbitrary image could consist of all the following: the specimen itself, accession information, data labels, bar-code, determination slip, tape, envelope, color bar, ruler, and institutional insignia. If an image contains any (or all) of the objects mentioned, the goal of the program is to classify (segment) each of the parts. The data set we were provided consists of over 48,000 images (for each of the image sizes 1x1, 7x7, 31x31, and 75x75) of herbarium sheets. Therefore, this type of program would be immensely useful in digitizing a herbarium collection and gaining additional insights.

2. Code Instructions

The CNN/DNN and segmentation code uses numerous libraries, which are essential to successfully execute these programs. The following text should be placed in a file called `requirements.txt`

```
opencv-python
imageio
keras
matplotlib
numpy
pandas
tensorflow
tensorflow-addons
Pillow
scikit-learn
tqdm
scipy
pathlib
```

Navigate to the directory where `requirements.txt` is saved on the terminal and execute the command `pip install -r requirements.txt`. Additionally, in the CNN/DNN code, there are two file paths that need to be adjusted based on the location of the dataset, as well as where the final model should be saved.

3. Branched CNN/DNN Code

The following is the complete CNN/DNN code. Note that the list variable `categorical_columns` can be changed to include/exclude any of the 32 present classes.

```python
import os
import random
import sys
import cv2
import imageio.v2 as imageio
import keras
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
from PIL import Image
from keras.callbacks import EarlyStopping, LearningRateScheduler,
   ModelCheckpoint
from keras.layers import BatchNormalization, Conv2D, Dense, Dropout, Flatten,
   MaxPooling2D
from keras.models import Sequential
from keras.optimizers import Adadelta, Adam
from keras_preprocessing import image
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.applications import ResNet50, MobileNetV2
from tensorflow.keras.losses import BinaryCrossentropy, CategoricalCrossentropy
from tqdm import tqdm
from keras.initializers import GlorotNormal
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Dense, Flatten
   , Concatenate
from tensorflow.keras.models import Model
from pathlib import Path
from sklearn.metrics import precision_score, recall_score
from sklearn.utils import class_weight
import tensorflow_addons as tfa
import seaborn as sns
```

```
# Categorical columns of the data frame, ordered by the number of occurrences
   of each class
categorical_columns = ['HerbariumSheet', 'Plant', 'Inflorescence', 'External',
   'Text', 'FlowerFruit',
                     'Vegetative', 'Typed', 'Leaf', 'Border', 'DataLabel', '
                        BorderDifferentComponents',
                     'ColorBarRuler', 'DeterminationSlip', 'Stem', 'Lamina', '
                        Background', 'Barcode',
                     'ColorBoxMatrix', 'InstitutionalInsigniaIcon', '
                        Handwritten', 'Envelope',
                     'AccessionInformation', 'BarsDemarcationLines', 'Damage',
                        'TapeString', 'Glue',
                     'Petiole', 'Tendril', 'BorderSameComponents', '
                        HerbariumSheetMargin', 'Other']
# Single Class
categorical_columns = ['External', 'FlowerFruit',
                     'Typed', 'DataLabel', 'BorderDifferentComponents',
                     'ColorBarRuler', 'DeterminationSlip', 'Stem', 'Lamina', '
                        Background', 'Barcode',
                      'InstitutionalInsigniaIcon', 'Handwritten', 'Envelope',
                     'AccessionInformation', 'BarsDemarcationLines', 'Damage',
                     'Petiole', 'Tendril', 'Other']

# Specify the number of classes that you want the model to consider and extract
    them to a list
```

```
num_classes = len(categorical_columns)
print("Number of classes: ", num_classes)
# Initialize a StandardScaler object for normalizing the images before training
scaler = StandardScaler()
# Set a random seed for reproducibility
random.seed(42)
```

- Choose the number of categorical classes that you want to take from the metadata file.
- Classes are ordered by the number of occurrences in the dataset.
- We chose to use only single classes, i.e., each class is mutually exclusive.

```
# Set the path to the InterdisciplinaryDataAnalysisClass_FinalDataSet folder on
    your local machine
data_folder_path = '/Users/fluidlab/Desktop/JamesDeepXDE/Seg/
   InterdisciplinaryDataAnalysisClass_FinalDataSet'

# Construct the file path for the metadata file
data_folder = Path(data_folder_path)
metadata_file_path = data_folder / 'sampleMetadata_concatenated.txt'

# Read the metadata file as a pandas DataFrame and define folder_names for
   image folders
metaData = pd.read_csv(metadata_file_path, sep='\t', dtype={'Timestamp': str})
folder_names = ['Local', 'Overview', 'Window', 'Pixel']

# Create a dictionary to store image arrays for 'Local', 'Overview', and '
   Window' folders
image_arrays = {folder_name.lower() + '_images': [] for folder_name in ['Local'
   , 'Overview', 'Window']}

num_samples = 48000
indices = random.sample(range(metaData.shape[0]), num_samples)

# Create a new DataFrame with only the selected entries
metaData_sample = metaData.iloc[indices].reset_index(drop=True)

# Add new columns to store the extracted RGB values
metaData_sample['R'] = 0
metaData_sample['G'] = 0
metaData_sample['B'] = 0

# Loop through the selected indices and process the images
for i in tqdm(indices):
    # Loop through the folder names (Local, Overview, Window, Pixel)
    for folder_name in folder_names:
        # Construct the image file path using the data folder path, folder name,
            sample number, and timestamp
        img_path = f'{data_folder_path}/{folder_name}/{folder_name.lower()}.{
            metaData["Sample Number"][i]}.{metaData["Timestamp"][i]}.png'

        # Check if the image file exists
        if os.path.exists(img_path):
            # Read the image using OpenCV and convert from BGR to RGB color
                space
            img = cv2.imread(img_path)
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

            # For 'Pixel' folder, extract RGB values and store them in the
                metadata dataframe
            if folder_name == 'Pixel':
                center_rgb = img[0, 0] / 255
                metaData_sample.loc[metaData_sample['Sample Number'] == metaData
                    ["Sample Number"][i], ['R', 'G', 'B']] = center_rgb
            else:
                # Reshape the image into a 2D array for scaling
                img_2d = img.reshape(-1, img.shape[-1])

                # Scale the image using StandardScaler
                img_scaled = scaler.fit_transform(img_2d)

                # Reshape the scaled image back to its original shape
                img = img_scaled.reshape(img.shape)

                # Append the processed image to the corresponding image_arrays
                    entry
                image_arrays[folder_name.lower() + '_images'].append(img)
```

- For a user-specified `num_samples`, read in `num_samples` files from the Pixel, Local, Overview, and Window files.
- For Pixel images, the R,G,B values are appended to new columns in the meta data file.
- For all other images, use the standard scalar from `sklearn.preprocessing` to normalize the image.
- Create a variable containing the categorical column information from the specified columns we are using.

```
# Check the number of images loaded in each folder
folder_names = ['Local', 'Overview', 'Window']
for folder_name in folder_names:
    print(f"Number of {folder_name.lower()} images: {len(image_arrays[
        folder_name.lower() + '_images'])}")

# Display a random image from each folder
for folder_name in folder_names:
    random_index = np.random.randint(len(image_arrays[folder_name.lower() + '
        _images']))
    img = image_arrays[folder_name.lower() + '_images'][random_index]
    print(f"{folder_name} image shape: {img.shape}")
    plt.imshow(img)
    plt.title(f"{folder_name} image")
    plt.show()
```

- Check that each image array has the correct number/dimension of images.

```
# Extract the numerical data from the specified columns in the metaData_sample
    DataFrame
top_classes = categorical_columns[:num_classes]

mlp_input = metaData_sample[['R', 'G', 'B']].values

# Convert the list of 'local_images' to a NumPy array
local_images = np.array(image_arrays['local_images'])

# Convert the list of 'overview_images' to a NumPy array
overview_images = np.array(image_arrays['overview_images'])

# Convert the list of 'window_images' to a NumPy array
window_images = np.array(image_arrays['window_images'])

# Extract the target values (y) from the metaData_sample DataFrame
y = metaData_sample[top_classes].values
```

- Convert each list to a `np.array` and extract the values from the categorical columns to store in our y variable.
- Although we tried to use the `X-scaled` and `Y-scaled` as input to our MLP branch, we found that this greatly reduced the efficiency of the model as we trained and produced extremely poorly segmented images; we are unsure on why this was the case.

```
# Set the split ratio for training and validation sets
split_ratio = 0.8

# Split the data into training and validation sets using the train_test_split
    function
(
    local_train, local_val,
    overview_train, overview_val,
    window_train, window_val,
    mlp_train, mlp_val,
    y_train, y_val
) = train_test_split(
    local_images, overview_images, window_images, mlp_input, y,
    train_size=split_ratio, random_state=42
)
```

- Split training and testing datasets from 80% and 20% of the data respectively.

```
class_freqs = y.sum(axis=0)

# Calculate the weights for each class using the inverse of their frequencies
class_weights = 1 / class_freqs

# Normalize the weights so they sum up to 1
class_weights = class_weights / class_weights.sum()

class_weights_dict = dict(zip(categorical_columns, class_weights))
```

- Based on the number of occurrences of each class in the metadata, create weights to address the class imbalance. This effectively tells our network to focus on getting the minority classes correct.

```
# Define the window branch of the CNN
def create_window_branch(input_shape):
    input_layer = Input(shape=input_shape, name="window_input")
    x = Conv2D(32, kernel_size=(3, 3), kernel_initializer=GlorotNormal(),
        activation='relu', name="window_conv1")(input_layer)
    x = BatchNormalization()(x)
    x = MaxPooling2D(pool_size=(4, 4), name="window_maxpool1")(x)
    x = Dropout(0.25)(x)
    x = Conv2D(64, kernel_size=(3, 3), kernel_initializer=GlorotNormal(),
        activation='relu', name="window_conv2")(x)
    x = BatchNormalization()(x)
    x = MaxPooling2D(pool_size=(4, 4), name="window_maxpool2")(x)
    x = Dropout(0.25)(x)
    x = Flatten(name="window_flatten")(x)
    return input_layer, x

# Define the overview branch of the CNN
def create_overview_branch(input_shape):
    input_layer = Input(shape=input_shape, name="overview_input")
    x = Conv2D(32, kernel_size=(3, 3), kernel_initializer=GlorotNormal(),
        activation='relu', name="overview_conv1")(input_layer)
    x = BatchNormalization()(x)
    x = MaxPooling2D(pool_size=(2, 2), name="overview_maxpool1")(x)
    x = Conv2D(64, kernel_size=(2, 2), kernel_initializer=GlorotNormal(),
        activation='relu', name="overview_conv2")(x)
    #x = BatchNormalization()(x)
    x = MaxPooling2D(pool_size=(2, 2), name="overview_maxpool2")(x)
    x = Conv2D(256, kernel_size=(2, 2), kernel_initializer=GlorotNormal(),
        activation='relu', name="overview_conv3")(x)
    x = MaxPooling2D(pool_size=(2, 2), name="overview_maxpool3")(x)
    x = Conv2D(256, kernel_size=(2, 2), kernel_initializer=GlorotNormal(),
        activation='relu', name="overview_conv4")(x)

    #x = Dropout(0.25)(x)
    x = Flatten(name="overview_flatten")(x)
    return input_layer, x

# Define the local branch of the CNN
def create_local_branch(input_shape):
    input_layer = Input(shape=input_shape, name="local_input")
    x = Conv2D(16, kernel_size=(3, 3), kernel_initializer=GlorotNormal(),
        activation = 'relu', name = "local_conv1")(input_layer)
    x = BatchNormalization()(x)
    x = MaxPooling2D(pool_size=(2, 2), name="local_maxpool1")(x)
    x = Dropout(0.25)(x)
    x = Flatten(name="local_flatten")(x)
    return input_layer, x

# Define the MLP for categorical data
def create_mlp_branch(input_shape, name):
    input_layer = Input(shape=input_shape, name=f"{name}_input")
    x = Dense(128, activation='relu', name=f"{name}_dense1")(input_layer)
    x = Dropout(0.25)(x)
    x = Dense(64, activation='relu', name=f"{name}_dense2")(x)
    return input_layer, x
```

- Create the architecture of our branched CNN/DNN model.

```
# Create CNN branches for each image type
local_input, local_branch = create_local_branch(input_shape=(7, 7, 3), name='
    local')
overview_input, overview_branch = create_overview_branch(input_shape=(75, 75,
    3), name='over')
window_input, window_branch = create_window_branch(input_shape=(31, 31, 3),
    name='window')

# Create MLP branch
categorical_input, categorical_branch = create_mlp_branch(input_shape=(3,),
    name='mlp')

# Concatenate the outputs of each branch
concatenated = Concatenate()([local_branch, overview_branch, window_branch,
    categorical_branch])

# Add a fully connected layer
fc = Dense(128, activation='relu')(concatenated)
fc = Dropout(0.15)(fc)
fc = Dense(64, activation='relu')(fc)
fc = Dropout(0.15)(fc)
fc = Dense(32, activation='relu')(fc)
fc = Dropout(0.15)(fc)
fc = Dense(32, activation='relu')(fc)
```

```
categorical_output = Dense(num_classes, activation='softmax', name='
    categorical_output')(fc)
# tf.keras.metrics.MeanIoU(num_classes=2)
optimizer = Adam(learning_rate=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-07,
    amsgrad=False)

# Import the F1 score to use as a metric which provides a more robust metric
# than just categorical accuracy
f1_score_metric = tfa.metrics.F1Score(num_classes=num_classes, average='
    weighted', threshold=0.5)
# Create and compile the model
model = Model(inputs=[local_input, overview_input, window_input,
    categorical_input], outputs=[categorical_output])
model.compile(optimizer = optimizer, loss= 'binary_crossentropy', metrics = [
    f1_score_metric,
             'categorical_accuracy'])

# Display the model summary
model.summary()
```

- Create the inputs for our CNN/DNN model, concatenate the outputs of each individual branch and feed into a dense neural net.
- The output layer has `num_classes` nodes, with each having the `softmax` activation function. Better results and similar segmentation images were obtained using the 'sigmoid' activation function on the last layer, even though we are in a situation where every class is mutually exclusive.
- A `binary_crossentropy` loss was used, as `categorical_crossentropy` quickly approached infinity within the first 2 epochs.

```
# Produce a graph of the models architecture
dot_img_file = 'pathToModelStructureFile.png'
tf.keras.utils.plot_model(model, to_file=dot_img_file, show_shapes=True)
```

- This produces a plot which displays the architecture of our branched CNN/DNN model, seen in Figure 1.

```
# Initialize the ModelCheckpoint callback, which will save the model weights
    after each epoch
# if the model's f1_score on the validation set has improved. The weights are
    saved in a file named 'best_model_singlefix_datagen10.h5'.
checkpoint = ModelCheckpoint('best_model_singlefix_datagen10.h5',
    save_best_only=True, monitor='f1_score', mode='max')

# Initialize the ReduceLROnPlateau callback, which will reduce the learning
    rate when the f1_score has stopped improving.
# 'factor' indicates by how much the learning rate will be reduced. 'patience'
    is the number of epochs with no improvement
# after which learning rate will be reduced. 'min_lr' is the lower bound on the
     learning rate.
```

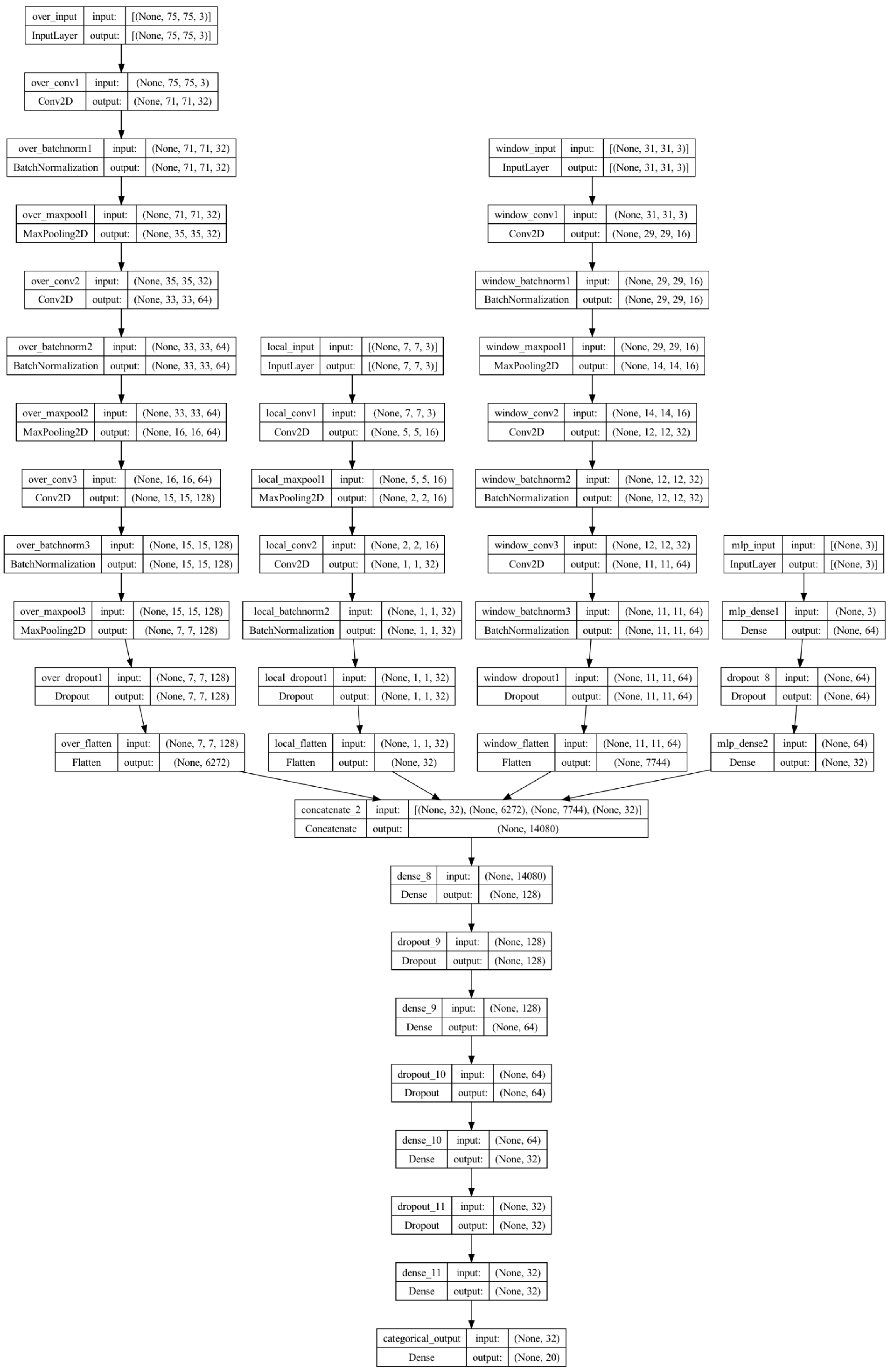

over_input | input: | [(None, 75, 75, 3)]
InputLayer | output: | [(None, 75, 75, 3)]
over_conv1 | input: | (None, 75, 75, 3)
Conv2D | output: | (None, 71, 71, 32)
over_batchnorm1 | input: | (None, 71, 71, 32)
BatchNormalization | output: | (None, 71, 71, 32)
over_maxpool1 | input: | (None, 71, 71, 32)
MaxPooling2D | output: | (None, 35, 35, 32)
over_conv2 | input: | (None, 35, 35, 32)
Conv2D | output: | (None, 33, 33, 64)
over_batchnorm2 | input: | (None, 33, 33, 64)
BatchNormalization | output: | (None, 33, 33, 64)
over_maxpool2 | input: | (None, 33, 33, 64)
MaxPooling2D | output: | (None, 16, 16, 64)
over_conv3 | input: | (None, 16, 16, 64)
Conv2D | output: | (None, 15, 15, 128)
over_batchnorm3 | input: | (None, 15, 15, 128)
BatchNormalization | output: | (None, 15, 15, 128)
over_maxpool3 | input: | (None, 15, 15, 128)
MaxPooling2D | output: | (None, 7, 7, 128)
over_dropout1 | input: | (None, 7, 7, 128)
Dropout | output: | (None, 7, 7, 128)
over_flatten | input: | (None, 7, 7, 128)
Flatten | output: | (None, 6272)
local_input | input: | [(None, 7, 7, 3)]
InputLayer | output: | [(None, 7, 7, 3)]
local_conv1 | input: | (None, 7, 7, 3)
Conv2D | output: | (None, 5, 5, 16)
local_maxpool1 | input: | (None, 5, 5, 16)
MaxPooling2D | output: | (None, 2, 2, 16)
local_conv2 | input: | (None, 2, 2, 16)
Conv2D | output: | (None, 1, 1, 32)
local_batchnorm2 | input: | (None, 1, 1, 32)
BatchNormalization | output: | (None, 1, 1, 32)
local_dropout1 | input: | (None, 1, 1, 32)
Dropout | output: | (None, 1, 1, 32)
local_flatten | input: | (None, 1, 1, 32)
Flatten | output: | (None, 32)
window_input | input: | [(None, 31, 31, 3)]
InputLayer | output: | [(None, 31, 31, 3)]
window_conv1 | input: | (None, 31, 31, 3)
Conv2D | output: | (None, 29, 29, 16)
window_batchnorm1 | input: | (None, 29, 29, 16)
BatchNormalization | output: | (None, 29, 29, 16)
window_maxpool1 | input: | (None, 29, 29, 16)
MaxPooling2D | output: | (None, 14, 14, 16)
window_conv2 | input: | (None, 14, 14, 16)
Conv2D | output: | (None, 12, 12, 32)
window_batchnorm2 | input: | (None, 12, 12, 32)
BatchNormalization | output: | (None, 12, 12, 32)
window_conv3 | input: | (None, 12, 12, 32)
Conv2D | output: | (None, 11, 11, 64)
window_batchnorm3 | input: | (None, 11, 11, 64)
BatchNormalization | output: | (None, 11, 11, 64)
window_dropout1 | input: | (None, 11, 11, 64)
Dropout | output: | (None, 11, 11, 64)
window_flatten | input: | (None, 11, 11, 64)
Flatten | output: | (None, 7744)
mlp_input | input: | [(None, 3)]
InputLayer | output: | [(None, 3)]
mlp_dense1 | input: | (None, 3)
Dense | output: | (None, 64)
dropout_8 | input: | (None, 64)
Dropout | output: | (None, 64)
mlp_dense2 | input: | (None, 64)
Dense | output: | (None, 32)
concatenate_2 | input: | [(None, 32), (None, 6272), (None, 7744), (None, 32)]
Concatenate | output: | (None, 14080)
dense_8 | input: | (None, 14080)
Dense | output: | (None, 128)
dropout_9 | input: | (None, 128)
Dropout | output: | (None, 128)
dense_9 | input: | (None, 128)
Dense | output: | (None, 64)
dropout_10 | input: | (None, 64)
Dropout | output: | (None, 64)
dense_10 | input: | (None, 64)
Dense | output: | (None, 32)
dropout_11 | input: | (None, 32)
Dropout | output: | (None, 32)
dense_11 | input: | (None, 32)
Dense | output: | (None, 32)
categorical_output | input: | (None, 32)
Dense | output: | (None, 20)

Figure 1. Branched CNN/DNN model architecture.

```
lr_scheduler = ReduceLROnPlateau(monitor='f1_score', factor=0.5, patience=3,
    min_lr=1e-10, verbose=1)
```

```python
# Define the batch size and number of epochs
batch_size = 64
epochs = 50

# Train the model using the fit method. 'local_train', 'overview_train', '
    window_train', and 'mlp_train' are the input features,
# and 'y_train' are the labels. The same structure applies to the validation
    data.
# 'class_weight' parameter is used to specify weights for each class, which can
     be useful for imbalanced datasets.
# The training process will use the defined learning rate scheduler and
    checkpoint callbacks.
history = model.fit(
    [local_train, overview_train, window_train, mlp_train],
    y_train,
    batch_size=batch_size,
    epochs=epochs,
    validation_data=([local_val, overview_val, window_val, mlp_val], y_val),
    class_weight=dict(enumerate(class_weights)),
    callbacks=[lr_scheduler, checkpoint]
)

# After training, save the entire model (including the architecture, optimizer,
     and learned weights) to a specified directory.
# Here the model is saved
model.save('/Users/fluidlab/Desktop/JamesDeepXDE/Seg')
```

The training process gives output

```
Epoch 1/50
600/600 [==============================] - 59s 96ms/step - loss: 0.0048 -
↪  f1_score: 0.3522 - categorical_accuracy: 0.4434 - val_loss: 0.0901 -
↪  val_f1_score: 0.5061 - val_categorical_accuracy: 0.6026 - lr: 0.0010
Epoch 2/50
600/600 [==============================] - 54s 91ms/step - loss: 0.0026 -
↪  f1_score: 0.5671 - categorical_accuracy: 0.6055 - val_loss: 0.0646 -
↪  val_f1_score: 0.6062 - val_categorical_accuracy: 0.6435 - lr: 0.0010
Epoch 3/50
600/600 [==============================] - 54s 90ms/step - loss: 0.0019 -
↪  f1_score: 0.6432 - categorical_accuracy: 0.6341 - val_loss: 0.0580 -
↪  val_f1_score: 0.6549 - val_categorical_accuracy: 0.6528 - lr: 0.0010
.
.
.
600/600 [==============================] - 55s 91ms/step - loss: 2.7924e-05 -
↪  f1_score: 0.7790 - categorical_accuracy: 0.7496 - val_loss: 0.0338 -
↪  val_f1_score: 0.7292 - val_categorical_accuracy: 0.7081 - lr: 3.9063e-06
Epoch 50/50
```

```
600/600 [==============================] - 55s 91ms/step - loss: 2.7380e-05 -
↪  f1_score: 0.7782 - categorical_accuracy: 0.7476 - val_loss: 0.0339 -
↪  val_f1_score: 0.7293 - val_categorical_accuracy: 0.7071 - lr: 1.9531e-06
```

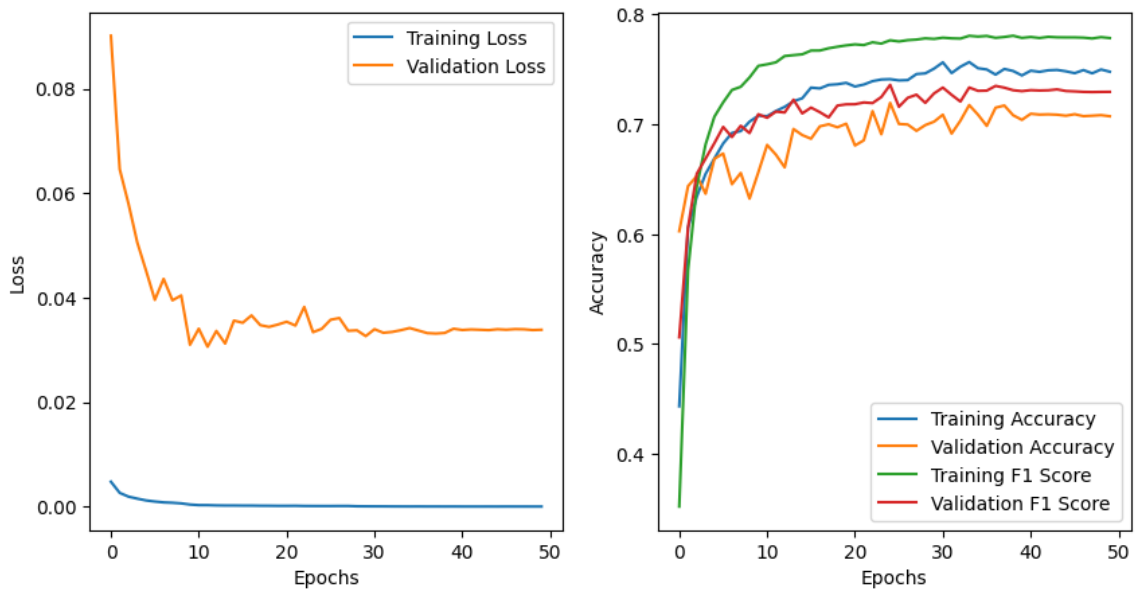


FIGURE 2. Training & Validation Categorical Accuracy and F1 Scores for increasing epochs.

4. Image Segmentation Code & Results

The following code uses our trained CNN/DNN model and generates segmented images:

```python
import os
import random
import sys
import cv2
import imageio.v2 as imageio
import keras
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_addons as tfa
from PIL import Image
from keras.callbacks import EarlyStopping, LearningRateScheduler,
    ModelCheckpoint
from keras.layers import BatchNormalization, Conv2D, Dense, Dropout, Flatten,
    MaxPooling2D
from keras.models import Sequential
from keras.optimizers import Adadelta, Adam
from tensorflow.keras.losses import BinaryCrossentropy, CategoricalCrossentropy
from keras_preprocessing import image
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.callbacks import ReduceLROnPlateau
from tensorflow.keras.applications import ResNet50, MobileNetV2
from tqdm import tqdm
from keras.initializers import GlorotNormal
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, Dense, Flatten
    , Concatenate, MultiHeadAttention
from tensorflow.keras.models import Model
from pathlib import Path
from scipy import ndimage
from tensorflow.keras.metrics import Precision, Recall, MeanIoU
import gc
gc.collect()
```

```
# Categorical columns of the data frame, ordered by the number of occurrences
    of each class
categorical_columns = ['HerbariumSheet', 'Plant', 'Inflorescence', 'External',
    'Text', 'FlowerFruit',
                        'Vegetative', 'Typed', 'Leaf', 'Border', 'DataLabel', '
                           BorderDifferentComponents',
                        'ColorBarRuler', 'DeterminationSlip', 'Stem', 'Lamina', '
                           Background', 'Barcode',
                        'ColorBoxMatrix', 'InstitutionalInsigniaIcon', '
                           Handwritten', 'Envelope',
                        'AccessionInformation', 'BarsDemarcationLines', 'Damage',
                           'TapeString', 'Glue',
                        'Petiole', 'Tendril', 'BorderSameComponents', '
                           HerbariumSheetMargin', 'Other']
# Single Class
categorical_columns = ['External', 'FlowerFruit',
                        'Typed', 'DataLabel', 'BorderDifferentComponents',
                        'ColorBarRuler', 'DeterminationSlip', 'Stem', 'Lamina', '
                           Background', 'Barcode',
                         'InstitutionalInsigniaIcon', 'Handwritten', 'Envelope',
                        'AccessionInformation', 'BarsDemarcationLines', 'Damage',
                        'Petiole', 'Tendril', 'Other']

num_classes = len(categorical_columns)
print(num_classes)
# Initialize a StandardScaler object for normalizing the images before training
scaler = StandardScaler()
output_directory = '/Users/fluidlab/Desktop/JamesDeepXDE/Seg/Images/prob'
```

- Choose the appropriate variables from the `categorical columns` list that correlate to the model that we are applying. In our case, we used the mutually exclusive single classification.

```
#Generate distinct colors for each num_class
colors = []
```

```python
for i in range(num_classes):
    hue = int(255 * i / num_classes) # Equally spaced hue values
    hsv_color = np.array([[[hue, 255, 255]]], dtype=np.uint8) # Create an HSV
        color with full saturation and value
    rgb_color = cv2.cvtColor(hsv_color, cv2.COLOR_HSV2BGR) # Convert the HSV
        color to RGB
    colors.append(tuple(rgb_color[0, 0])) # Append the RGB color as a tuple to
        the colors list
print(colors)
```

- Create unique colors corresponding to each of the entries in `categorical_columns`.

```python
custom_objects = {
    "F1Score": tfa.metrics.F1Score,
}

model = tf.keras.models.load_model('/Users/fluidlab/Desktop/JamesDeepXDE/Seg/
    best_model_single_softmaxfinal.h5', custom_objects=custom_objects)

# Load the image
img = cv2.imread('/Users/fluidlab/Desktop/JamesDeepXDE/Seg/PreImages/
    Passiflora_amalocarpa_6d1afcc4-7092-40a7-854a-45e4885fe778.jpg')

# Get the original dimensions of the image
original_height, original_width, _ = img.shape

# Calculate the new height based on the aspect ratio
new_width = 700
aspect_ratio = original_height / original_width
new_height = int(new_width * aspect_ratio)

# Resize the image while maintaining aspect ratio
img = cv2.resize(img, (new_width, new_height))

# Normalize the image using StandardScaler
scaler = StandardScaler()
img_2d = img.reshape(-1, img.shape[-1])
img_scaled = scaler.fit_transform(img_2d)
img = img_scaled.reshape(img.shape)

# Define the border size
border_size = 50

# Add padding to the image
img = cv2.copyMakeBorder(img, border_size, border_size, border_size,
    border_size, cv2.BORDER_CONSTANT, value=0)
```

```python
# Get the dimensions of the image
h, w, _ = img.shape
```

- Read in the saved Keras model, creating a `custom_objects` directory that accounts for functions that are not explicitly buily into Keras. In our case, this was the F1Score metric that we imported from `tensorflow-addons`.
- Read in the image that we wish to segment and resizing to have a width of 700 pixels while also maintaining the aspect ratio.
- Normalize the image using `StandardScaler` before adding padding of 50 pixels on each edge of the image.

```python
# Create arrays to hold the patches for each image type
local_patches = np.zeros((h*w, 7, 7, 3), dtype=np.uint8)
window_patches = np.zeros((h*w, 31, 31, 3), dtype=np.uint8)
overview_patches = np.zeros((h*w, 75, 75, 3), dtype=np.uint8)
mlp_data = np.zeros((h*w, 3), dtype=np.float32)

# Loop through each pixel in the image and generate patches
index = 0
for i in range(border_size, h - border_size):
    for j in range(border_size, w - border_size):
        local_patches[index] = img[i-3:i+4, j-3:j+4]
        window_patches[index] = img[i-15:i+16, j-15:j+16]
        overview_patches[index] = img[i - 37:i + 38, j - 37:j + 38]
        mlp_data[index] = img[i, j]
        index += 1
```

- We loop through every pixel in the image, creating and storing patches of size 7x7, 31x31, and 75x75 as well as storing the RGB values; these will be the input to our model.

```python
# The following line instructs the CPU to carry out applying the model, since
# the majority of these computations were done on a M1 Mac, which doesn't
# have support for CUDA
with tf.device('/cpu:0'):
    predictions = model.predict([local_patches, overview_patches,
        window_patches, mlp_data])
```

```python
# Calculate the dimensions of the image without the border
borderless_height = h - 2 * border_size
borderless_width = w - 2 * border_size

# Get the total number of center pixels
num_center_pixels = borderless_height * borderless_width

# Reshape the predictions array by removing the excess predictions
predictions = predictions[:num_center_pixels]

# Reshape the output
```

```
output = predictions.reshape(borderless_height, borderless_width, num_classes)

# Calculate basic statistics for each class
min_values = np.min(output, axis=(0, 1))
max_values = np.max(output, axis=(0, 1))
mean_values = np.mean(output, axis=(0, 1))

# Print the statistics
for i in range(num_classes):
   print(f"Class {i}: Min={min_values[i]:.4f}, Max={max_values[i]:.4f}, Mean={
      mean_values[i]:.4f}")
```

- After applying the model, we reshape the output to the original size of the image before padding.
- We then calculate statistics on output data, finding the min, max, and mean for each class, helping us immediately identify if our model is *actually* performing well.
  - For instance, when our MLP input data consisted of `R, G, B, X-scaled, Y-scaled`, the model trained and reported back great accuracy and F1Score. However, when examining the output of these basic quantities, almost every class had a mean of 0 and most had a max value of 0.

```
import matplotlib.pyplot as plt

# Visualize probabilities for each class using a heatmap
for i in range(num_classes):
   class_name = categorical_columns[i]
   plt.imshow(output[:, :, i], cmap='jet')
   plt.colorbar()
   plt.title(f'{class_name} probabilities (Class {i})')
   plt.savefig(f'class_{i}_{class_name}_probabilities.png')
   plt.show()

from scipy import ndimage

# Assume 'output' is your input 3D numpy array
# Shape: (height, width, num_classes)

log_images = np.empty_like(output)

for i in range(output.shape[2]): # For each class...
   smooth_image = ndimage.gaussian_filter(output[:,:,i], sigma=5)
   log_images[:,:,i] = ndimage.laplace(smooth_image)

# Visualize LoG images for each class using a heatmap
for i in range(output.shape[2]):
   class_name = categorical_columns[i]
   plt.imshow(log_images[:, :, i], cmap='jet')
   plt.colorbar()
```

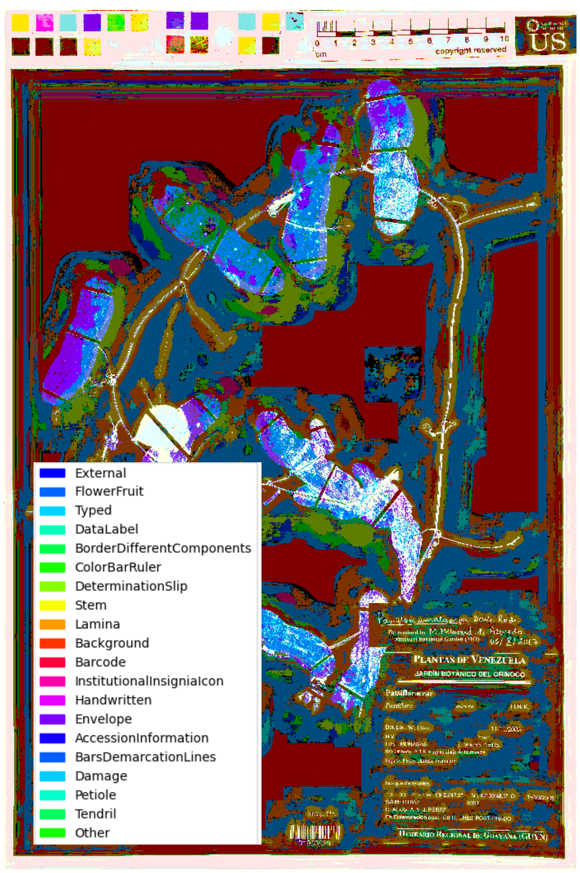
```
    plt.title(f'{class_name} edges (LoG) (Class {i})')
    plt.savefig(f'class_{i}_{class_name}_log_edges.png')
    plt.show()
output += log_images
```

- Here we use Gaussian smoothing and Laplacian filters from `scipy.ndimage` to our output array.
  - The Gaussian filter uses a normal distribution of two variables
  $$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$
  and creates a matrix to perform discrete convolution; just as we have already done in our branched CNN/DNN model. This helps 'smooth' the values in our output array and reduce noise.
  - The Laplacian of a function
  $$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$
  is a measure of concavity of a function f at a point $(x, y) \in \mathbb{R}^2$. For our purposes[1], it helps detect horizontal and vertical edges by examining the pixel intensity values for all n classes $I_n(x, y)$ and calculating
  $$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$
  which is again accomplished through discrete convolution.
- We take the output of the Gaussian/Laplacian filters and add them to our output array; we are unsure if this was the right sequence of steps for post-processing.

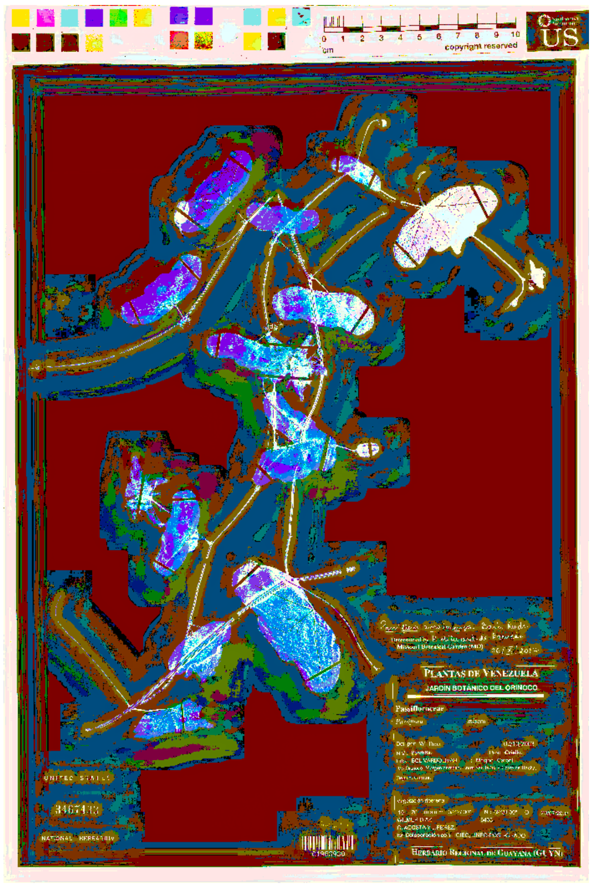
```
# This block looks at each of the pixels probabilities and if above the
    threshold, adds to the class color maps

# Remove the border from the image
img_without_border = img[border_size:border_size+borderless_height, border_size
    :border_size+borderless_width]

# Create the segmented images
threshold = 0.95 # Set the threshold value

segmented_images = []
for k in range(num_classes):
    segmented_image = np.zeros((borderless_height, borderless_width, 4), dtype=
        np.uint8)
    segmented_image[:, :, :3] = colors[k] # Set the color of the segmented
        image

    # Set the alpha channel based on the threshold
    alpha_channel = np.where(output[:, :, k] >= threshold, (output[:, :, k] *
        255).astype(np.uint8), 0)
    segmented_image[:, :, 3] = alpha_channel

    segmented_images.append(segmented_image)
```

```
# Combine the segmented images and overlay on the original image
combined_segmented_image = np.zeros((borderless_height, borderless_width, 4),
    dtype=np.float32)
for segmented_image in segmented_images:
    alpha = segmented_image[:, :, 3][:, :, np.newaxis] / 255.0
    combined_segmented_image[:, :, :3] += alpha * segmented_image[:, :, :3]
    combined_segmented_image[:, :, 3] = np.maximum(combined_segmented_image[:,
        :, 3], alpha[:, :, 0])

combined_segmented_image = combined_segmented_image.astype(np.uint8)

# Ensure the data types of img_without_border and combined_segmented_image are
    the same
img_without_border = img_without_border.astype(np.uint8)

# Merge the combined segmented image with the original image
segmented_overlay = cv2.addWeighted(img_without_border, .9,
    combined_segmented_image[:, :, :3], .5, 0)
output_filename = '/Users/fluidlab/Desktop/JamesDeepXDE/Seg/Images/
    output_image_singleclassLOG4.png'
cv2.imwrite(output_filename, segmented_overlay)
# Display the image using matplotlib
plt.imshow(cv2.cvtColor(segmented_overlay, cv2.COLOR_BGR2RGB))
plt.axis('off')
plt.show()
```

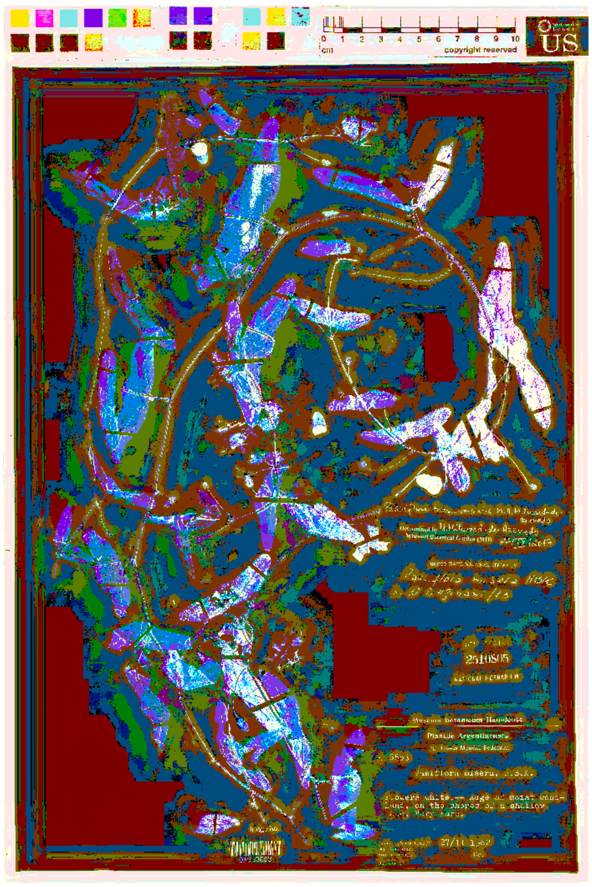
- We take the output array after post-processing, and for each class, we add the pixel to that color map if the probability that the pixel is above a certain threshold (95% in this instance).
- We then overlay each color map onto the original image and save the file, the results of which can be seen in Figure 3.
- Although our model was able to identify different components of the herbarium sheets, there is definitely room for improvement. All classes are **not** truly mutually exclusive, and many pixel groupings are incorrectly classified.
- We suspect a problem with the underlying dataset for training, as no amount of tweaking could produce better results than those shown.



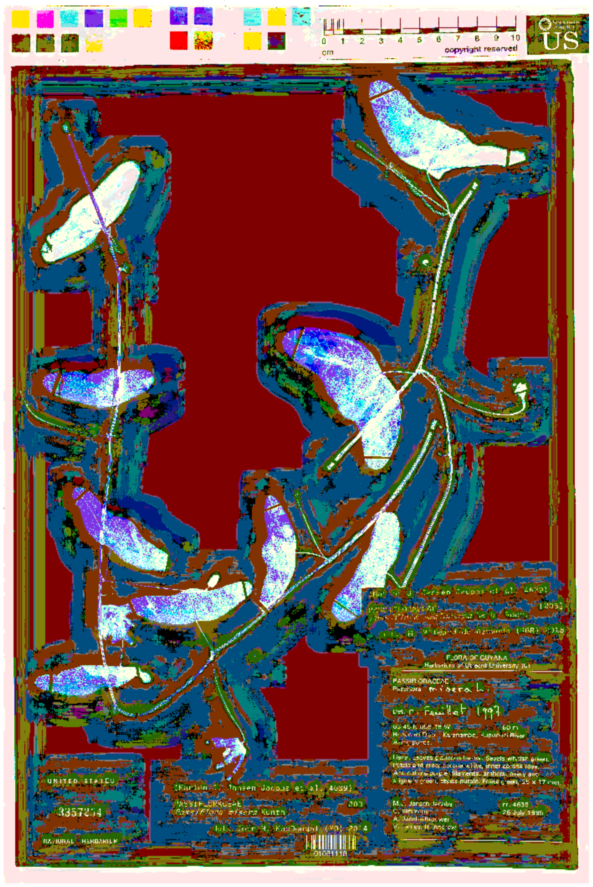
(A) Seg Image 1



(B) Seg Image 2



(C) Seg Image 3



(D) Seg Image 4

FIGURE 3. Segmented images and corresponding legend showing the unique color of each class.

References

[1] URL: https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm.

[2] G. Bradski. “The OpenCV Library”. In: *Dr. Dobb’s Journal of Software Tools* (2000).

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] Abadi Martín et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[5] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.