ALS-CLS: AUTONOMOUS LEAF SEGMENTATION IN COMPLICATED LIGHTING SITUATIONS

Report submitted to SASTRA Deemed to be University
As per the requirement for the course

CSE300: MINI PROJECT

Submitted By

Arshiya Obili (Reg no:125003035, B.Tech III year CSE)

Shalini R (Reg no: 125015113, B.Tech III Year IT)

Kappeta poojitha (Reg no:125003227, B Tech III Year CSE)

May 2024

Guided By

SRILAKSHMI A

Assistant Professor II

School Of Computing

SCHOOL OF COMPUTING

THANJAVUR, TAMIL NADU, INDIA – 613 401

SCHOOL OF COMPUTING

THANJAVUR – 613 401

Bonafide Certificate

This is to certify that the report titled **"ALS-CLS: AUTONOMOUS LEAF SEGMENTATION IN COMPLICATED LIGHTING SITUATIONS"** submitted as a requirement for the course, **CSE300 : MINI PROJECT** for B.Tech. is a Bonafide record of the work done by Ms.Arshiya obili(Reg. No:125003035,B.Tech-CSE),Ms.Shalini R(Reg.No:1245015113, B.Tech-IT),Ms.Poojitha kappeta(Reg.No:125003227 B.Tech-CSE) during the academic year 2023-2024, in the School of Computing, under my supervision.

Signature of Project Supervisor	**:**
Name with Affiliation	**:**
Date	**:**

Mini Project *Viva voice* held on ________________

Examiner 1 **Examiner 2**

ACKNOWLEDGEMENTS

We would like to thank our Honorable Chancellor **Prof. R. Sethuraman** for providing us with an opportunity and the necessary infrastructure for carrying out this project as a part of our curriculum.

We would like to thank our Honorable Vice-Chancellor **Dr. S. Vaidhyasubramaniam** and **Dr. S. Swaminathan**, Dean, Planning & Development, for the encouragement and strategic support at every step of our college life.

We extend our sincere thanks to **Dr. R. Chandramouli**, Registrar, SASTRA Deemed to be University for providing the opportunity to pursue this project.

We extend our heartfelt thanks to **Dr. V. S. Shankar Sriram**, Dean, School of Computing, **Dr. R. Muthaiah**, Associate Dean, Research, **Dr. K.Ramkumar**, Associate Dean, Academics, Dr. D. Manivannan, Associate Dean, Infrastructure**, Dr. R. Algeswaran** , Associate Dean, Students Welfare.

Our guide **SriLakshmi A**, Associate Professor, School of Computing was the driving force behind this whole idea from the start. Her deep insight in the field and invaluable suggestions helped us in making progress throughout our project work. We also thank the project review panel members for their valuable comments and insights which made this project better.

We would like to extend our gratitude to all the teaching and non-teaching faculties of the School of Computing who have either directly or indirectly helped us in the completion of the project.

We gratefully acknowledge all the contributions and encouragement from my family and friends resulting in the successful completion of this project. We thank you all for providing me an opportunity to showcase my skills through this project.

List of Figures

Fig No.	Title	Page No
1.3.1	Proposed Framework of the Project	4
1.4.1	Testing data under Purple Light	5
1.4.2	Testing data under Natural Light	5
3.1	Output of leaf Segmentation Model under supervised Learning	31
4.1	Purple image from Test dataset used for Model Testing	32
4.2	Generator Model produced a*b* color space by taking l*	32
4.3	The Image consists of the output for the leaf segmentation under Self Supervised Learning	33
4.4	Ground Truth mask and also Predicted Binary Mask used for FBD calculation (self-supervised Learning)	33
4.5	Ground Truth mask and also Predicted Binary Mask used for FBD calculation (self-supervised Learning)	34

List of Tables

Table No	Title	Page No
4.6	Leaf Segmentation Result in terms of FBD (%) of purple Testing Dataset	34

Abbreviations

RGB	Red Green Blue
ReLU	Rectified Linear Unit
PSNR	Peak Signal-to-Noise Ratio
SSIM	Structural Similarity Index
CNN	Convolutional Neural Network
CRF	Conditional Random Fields
GAN	Generative Adversary Network
CVPPP	Computation Vision Problem in Plant Phenotyping
FBD	Foreground Background Dice

ABSTRACT

Leaf segmentation has gained more attention recently as a crucial precondition work for image based plant phenotyping. Although self-supervised learning is showing promise as a substitute for a variety of computer vision tasks, its applicability to image based plant phenotyping is yet relatively studied and despite the significant advancements over the years, a number of obstacles continue to prevent the widespread application of current techniques in real-world implementation.

Firstly, training deep models takes a significant period of annotated data, however it could be extremely difficult to collect pixel-wise annotation for segmentation. Secondly, models of deep learning trained using datasets of certain plant species typically don't generalize well to new species that have not been observed, this issue is especially significant when it comes to plant leaf segmentation. Lastly, one further factor contributing to the difficulties encountered by image based plant phenotyping is the significant variations in the background and leaf look brought about by different lighting conditions. Many attempts have been made to create fresh synthetic data sets by 3D plant modelling in order to lessen the above issues with Generative Adversary Networks (GANs), Randomization of domains. But correctly simulating diverse environmental conditions,plant characteristics and the intricate interactions between environmental and genetic factors is challenging, if not impossible.

As a result, there will always be a discrepancy between the real and synthetic data. Traditional unsupervised clustering algorithms, such as the expectation maximization (EM) algorithm, K-means and fuzzy clustering based on the particular scenario of leaf segmentation. U-Net and Mask R-CNN are two examples of deep neural network architectures that have been effectively applied to category-level or the other one called instance-level leaf segmentation. In order to create artificial plant images, some other works use generative adversarial networks (GANs).Even while they are effective at producing realistic synthetic plant images, all of the aforementioned techniques still heavily rely on labelled pictures.

Keywords : Convolutional Neural Network ,U-Net, Generative Adversarial Networks

Table of Contents

Title	Page No.
Bonafide Certificate	ii
Acknowledgements	iii
List of Figures and Tables	iv
Abbreviations	v
Notations	vi
Abstract	vii
1. Summary of the base paper	1
1.1 Base Paper Details	1
1.2 Introduction	1
1.3 Solution Proposed	3
1.4 Work Done and Methodology	5
2. Merits and Demerits of the base paper	7
2.1 Literature survey	7
2.2 Merits	8
2.3 Demerits	8
3. Source Code	10
4. Output Pictures	32
5. Conclusion and Future Plans	35
6. References	36
7. Appendix -Base Paper	37

CHAPTER 1
SUMMARY OF THE BASE PAPER

1.1 BASE PAPER DETAILS

Title of the base paper	:	Self Supervised Leaf Segmentation Under Complex Lighting Conditions
Name of the Journal	:	Pattern Recognition
Indexing	:	ScienceDirect
Impact factor of Journal	:	8.518
Year of publication	:	2022
Link of the Base paper	:	

https://www.sciencedirect.com/science/article/pii/S0031320322005015

1.2 INTRODUCTION

In today's era Agriculture has been playing a very crucial part in economy sector of country and life style of a person. Plant based phenotyping is a scientific field that measures observable plant traits through the interaction of genotype with environmental conditions. The human inspection of leaves by agricultural professionals, which can be time-consuming, subjective, and prone to error, has traditionally been the primary method for phenotyping.

In order to overcome these difficulties, deep learning algorithms combined with advanced computer vision techniques have shown promise as tools for automating disease diagnosis and monitoring in crops. Category-level leaf segmentation is easier and more feasible for applications like plant growth monitoring.

1.3 SOLUTION PROPOSED

Despite significant progress, existing techniques for image-based plant phenotyping still face challenges. Deep learning models require large amounts of annotated data, which can be labour-intensive and error-prone. This is particularly challenging for plant leaf segmentation due to the dramatic variations in leaf appearances across different plant species. Additionally, varying lighting conditions, such as climate or weather or the times of the day, add to the complexity of leaf segmentation. Limited research has been conducted to know the productivity of leaf segmentation under different lighting conditions, particularly in greenhouse cultivation. To address these issues, synthetic data samples have been generated using 3D plant modelling, GANs, and domain randomization.

In this work, we propose to surmount the above challenges by developing a self-supervised leaf segmentation framework comprising a semantic segmentation model, The color-based leaf segmentation algorithm , and a self-supervised color correction model.

1. The semantic segmentation model allows the same semantic object's pixels to be jointly considered by the semantic algorithm.

2. The self-supervised color correction model is proposed for images taken under complex illumination conditions.

1.Self-Supervised Color Correction:

The architecture of a GAN-based pixel-by-pixel image translation network is mimicked by the color correction model. It is composed of a discriminator that determines the difference between authentic images captured in natural light and synthetic images produced by the generator, and a generator that produces color-corrected images.

Generator:
The generator consists of residual blocks and Convolution- BatchNormal ReLU modules. To extract high-level features from the input image, it down samples it. These features are then up sampled to create the target images. By enabling direct information flow between the downsampling and upsampling branches, skip connections are used to preserve low-level information lost during downsampling.

Discriminator:
A binary classifier called the discriminator is built by stacking Convolution-BatchNorm-ReLU blocks. It seeks to distinguish between generated (fake) and real images.

Training Data:
Natural images that were taken in daylight and lack labels make up the training set. Every picture is transformed into the Lab* color space, where the ab channels store the color values and the L* channel stores the lightness values. The generator can extract color channels from grayscale pictures because of its training.

Training process:
The generator is trained to output color-corrected images from input images that have been corrupted (by bad weather or artificial lighting, for example). A mixture of real and fake images is used to train the discriminator.

A weighting factor is used to balance the combination of GAN and L1 losses that make up the training loss.

Adversarial Training:
To maximize the color correction model, the discriminator and generator are trained in an adversarial fashion, one after the other.

2.Self- Supervised Semantic Segmentation:

The framework lets the neural network decide if two pixels belong to the same class instead of using human-defined pseudo labels. For leaf segmentation, a shallow neural network comprising two or three convolutional layers extracts discriminative features.
By first performing a 1x1 pointwise convolution and then a 3x3 depthwise convolution, the number of parameters is decreased and embedding learning is accelerated. Convolutional layers' output is concluded by a softmax layer, which models the label assignment uncertainty at each pixel. The probability distribution over semantic labels for every pixel is subjected to argmax classification in order to produce pseudo labels. Through backpropagation, these pseudo labels are used to iteratively update the network parameters.

In general, the goal of this self-supervised framework is to overcome the difficulties associated with leaf segmentation by utilizing the neural network's capacity to identify semantic similarities between pixels. This eliminates the need for manual annotation and allows the system to adjust to a variety of lighting conditions.

3.Color Based Leaf Segmentation:

The proposed self-supervised semantic segmentation algorithm eliminates the need for large-scale datasets for training, but requires additional effort to distinguish leaves from other objects.

Segmenting leaves based on color:
Previous research has demonstrated the potential of color-based features for leaf segmentation, especially in supervised settings or images with uniform backgrounds. In order to jointly process similar pixels of the same semantic label and extract trustworthy color information, the suggested algorithm makes use of the findings from the self-supervised semantic segmentation .It suggests a leaf segmentation algorithm that takes pixel "greenness" into account.

Steps in an Algorithm:

a. Color Replacement:

The average color of the segmented region that each pixel is a part of is substituted for its original color, which is determined locally using connected regions.

b. Conversion to HSV Color Space:

The RGB color space is replaced with the HSV color space for the image.

c. Greenness Measurement:

In the HSV color space, each pixel's "greenness" is determined using a multivariate normal distribution.

d. Thresholding:

A binary leaf segmentation mask is created using thresholding based on absolute and relative greenness, allowing object comparison within the same image, and defining thresholds for leaf regions.

e. Thresholding operation:

The binary leaf segmentation mask generates based on greenness measurements exceeding predefined thresholds, combining self-supervised semantic segmentation with color-based features to accurately segment leaves in cluttered backgrounds.

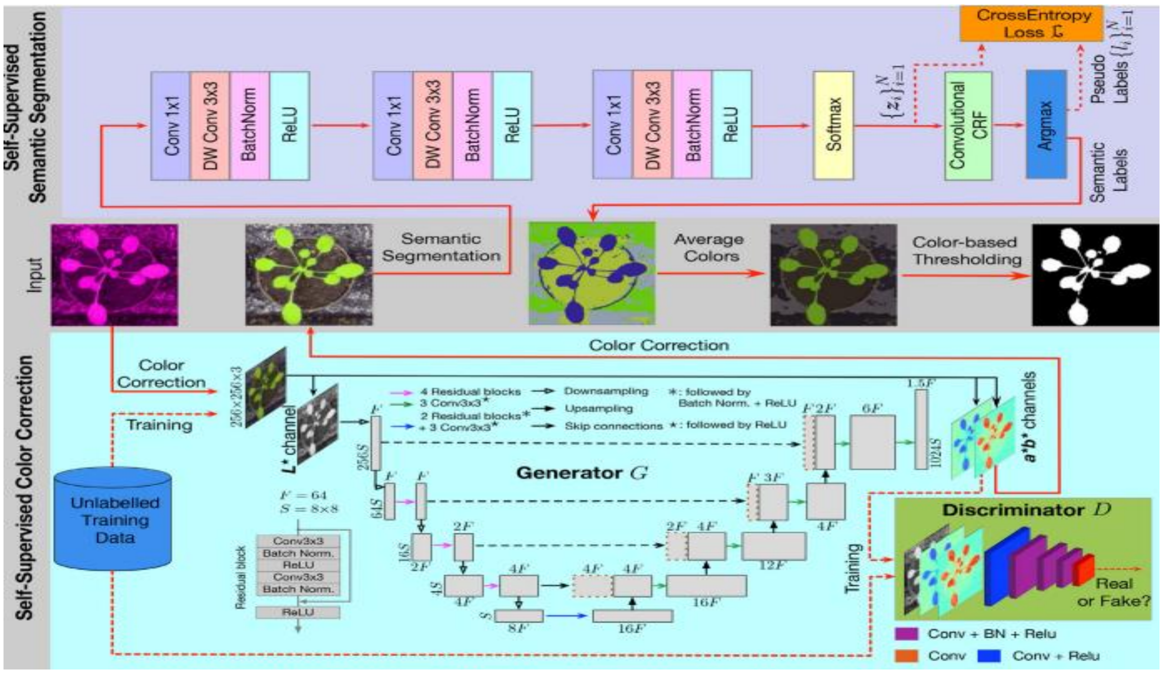


Fig 1.3.1: Proposed Framework of the project

1.4 WORK DONE AND METHODOLOGY

Dataset :

The Training Dataset CVPPP LSC of four folders, A1, A2, A3, and A4, each containing 128, 31, 27, 624 photos that were captured in natural illumination.Regarding the Testing dataset, it has 120 photos with 40 images in each of the three lighting situations (Natural, Yellow, and Purple). to replicate the CVPPP dataset's "Yellow" and "Purple" lighting conditions. Each image is converted to "Yellow" and "Purple" versions by adjusting the hue value of each pixel.

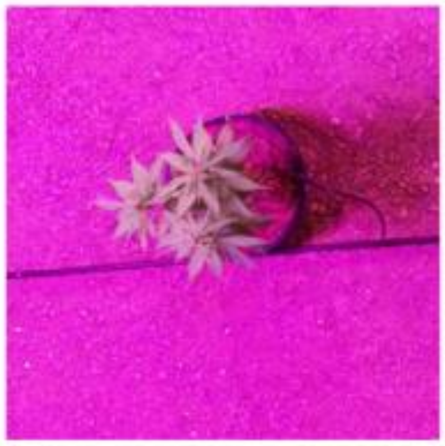


Fig 1.4.1: testing data under purple Light

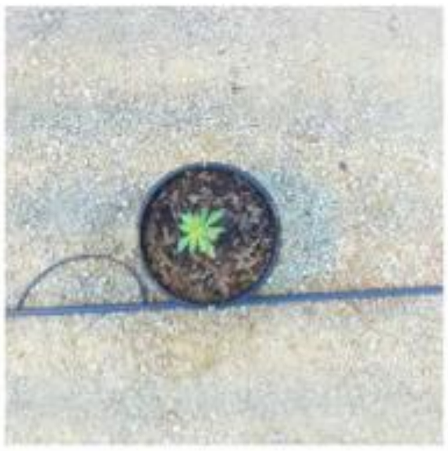


Fig 1.4.2: testing data under Natural

Data Augmentation:

The model is trained with variety of transformation of data. This is done to increase the size of the whole dataset, which improve the performance. We have applied: Rotation Translation Scaling Zoom It also help to prevent overfitting the model this makes it more robust to variations of data. Model Development This study uses CNN a popular neural network that is mostly used for image processing. We have built our own model in which the result of this model is far more equivalent to the result of VGG16 . On a total the designed Discriminator model has 9 convolutional layers, followed by 3 fully connected layers,3 max pooling layers and a softmax layer. Generator model has 3 convolutional layers.

Evaluation Metrics:

"Accuracy" is a common evaluation statistic that is frequently used to rate deep learning-based image categorization models when balanced datasets are used.

1)SSIM is a metric that measures the similarity between two signals (e.g. images) in terms of luminance, contrast, and structure.

2)LPIPS computes the similarity between two images by measuring the distance between their deep embeddings, which are obtained from pre-trained classic neural networks

3)FBD measures the similarity between the foreground (object of interest) regions in the predicted segmentation and the ground truth segmentation.

CHAPTER 2

MERITS AND DEMERITS OF THE BASE PAPER

2.1 LITERATURE SURVEY:

There are various algorithms available for Plant-Phenotyping, segmentation.

- **"A self-supervised overlapped multiple weed and crop leaf segmentation approach under complex light condition" Anand** Muni Mishra et al,[]. This work has used three different datasets, and collected 5090 images for training the model. Implementing the PSPUSegNet model for on-field applications may require substantial computational resources and processing power, potentially hindering its deployment on resource-constrained devices commonly used by farmers in remote or low-income areas

- **"Self-Supervised Plant Phenotyping by Combining Domain Adaptation with 3D plant model Simulations. Application to wheat Leaf Counting at Seedling Stage"** by Yinglun Li et al,[13]. The digital plant phenotyping platform has been used to simulate a large dataset of RGB images and corresponding leaf tip labels of wheat plants at seedling stages .One disadvantage is that the approach leverages a high-throughput method for generating a large dataset of RGB images, the realism of the simulated data may not fully capture the variability and complexity present in real-world environments

- **"Finely-grained annotated datasets for image-based plant phenotyping"**by Massimo Minervini et al,[6]. The model defines computer vision tasks, provide datasets and discuss annotation processes. One potential disadvantage of this paper is that the defined computer vision tasks and classification objectives may not cover the entire spectrum of challenges encountered in plant phenotyping

- **“Mask R-CNN based automated identification and extraction of oil well sites”** by Hongjie He et al,[13]. This model improves oil well site extraction from high-resolution satellite images in mining regions by replacing the backbone with D-Link Net and adding a semantic segmentation branch. But the disadvantage is that if the pre-training data does not adequately capture the characteristics of the target domain (i.e., oil well site extraction), it may limit the effectiveness of the proposed model.

- **“Convolutional Networks for Biomedical Image Segmentation”** by Olaf Ronneberger et al,[22]. The model architecture achieves superior segmentation of neuronal structures on electron microscopic stacks. The disadvantage is that The architecture's performance improvements are demonstrated primarily in the context of specific challenges (ISBI challenge) and microscopy imaging modalities, which may not extend to other domains.

2.2 Merits:

- Innovative Approach: The paper introduces a novel self-supervised framework for leaf segmentation, which can potentially improve accuracy and efficiency in image-based plant phenotyping.
- Comprehensive Evaluation: The study conducts a thorough evaluation of the proposed framework using various metrics and comparisons with existing methods, providing a clear understanding of its performance.
- Data Availability: The authors plan to make the data publicly available through a GitHub link, promoting transparency and reproducibility in research.
- Ablation Studies: The paper contains ablation studies to analyze the impact of different components on segmentation performance, enhancing the understanding of the proposed method.
- Performance Analysis: Detailed performance analyses and comparisons with unsupervised, supervised, and self-supervised methods demonstrate the effectiveness of the proposed framework.

2.3 Demerits:

- Complexity: The training process involves multiple models, epochs, and optimization techniques, which may make it challenging to implement for researchers with limited resources or expertise.
- Performance Variations: The color correction model shows performance variations on images taken under complex lighting conditions, indicating potential challenges in real-world applications.

CHAPTER 3

SOURCE CODE

```
import os
import cv2
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, Model, optimizers
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt
```

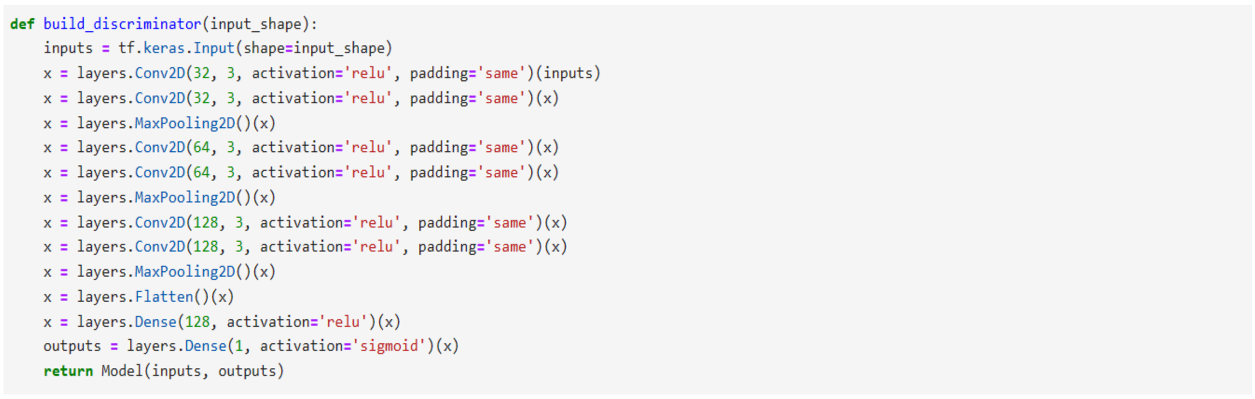
● Importing necessary libraries

```
def build_generator(input_shape):
    inputs = tf.keras.Input(shape=input_shape)
    x = layers.Conv2D(32, 3, activation='relu', padding='same')(inputs)
    x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
    outputs = layers.Conv2D(3, 3, activation='sigmoid', padding='same')(x)
    return Model(inputs, outputs)
```

- The function build_generator(input_shape) accepts an input_shape parameter, which specifies the shape of the input data.
- Input Layer: The function starts by creating an input layer with the given input_shape.
- Convolutional Layers: There are two convolutional layers with 32 filters, 3x3 kernels, ReLU activation and same padding. These layers help the model learn different features from the input data.
- Output Layer: The final layer is a convolutional layer with 3 filters (to match the number of channels in the output image, e.g., RGB channels), a 3x3

kernel, and sigmoid activation. The sigmoid activation ensures that the output values are in the range of [0,1] , which is suitable for images.

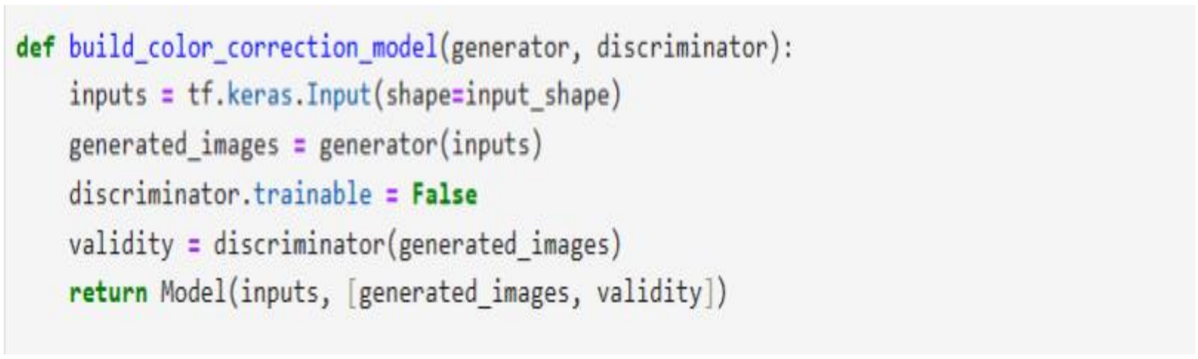
- Model Creation: The function returns a keras model using the defined input and output layers. This model is the generator.



```
def build_discriminator(input_shape):
    inputs = tf.keras.Input(shape=input_shape)
    x = layers.Conv2D(32, 3, activation='relu', padding='same')(inputs)
    x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Conv2D(64, 3, activation='relu', padding='same')(x)
    x = layers.Conv2D(64, 3, activation='relu', padding='same')(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Conv2D(128, 3, activation='relu', padding='same')(x)
    x = layers.Conv2D(128, 3, activation='relu', padding='same')(x)
    x = layers.MaxPooling2D()(x)
    x = layers.Flatten()(x)
    x = layers.Dense(128, activation='relu')(x)
    outputs = layers.Dense(1, activation='sigmoid')(x)
    return Model(inputs, outputs)
```

- The function build_discriminator takes one argument,input_shape, which specifies the shape of the input tensor. This shape represents the input data (e.g., an image’s height , width and channels).
- Input Layer: The function starts by creating an input layer with the specified input_shape.
- Convolutional Layers: The discriminator uses multiple convolutional layers with different numbers of filters. The first pair of convolutional layers uses 32 filters and ReLU activation, followed by a max pooling layer.
- The next pair uses 34 filters and ReLU activation, next comes another max pooling layer. The last pair uses 128 filters and ReLU activation, followed by a final max pooling layer.
- These layers process the input data and learn features that help distinguish between real and fake data.
- Max Pooling: After each set of convolutional layers, there is a max pooling layer which reduces the spatial dimensions of the data and retains important features.
- Flattening: The output from the last max pooling layer is flattened using layers.Flatten(). This converts the 3D output from the previous layer into a 1D array.
- Dense Layers: The flattened Data is then passed through a dense layer with 128 units and ReLU activation.
- Output Layer: The final layer is a dense layer with one unit and sigmoid activation. This layer gives output as a single value between 0 and 1, representing the probability that the input data is real.
-

- Model Creation: The function returns a Keras model using the defined input and output layers.

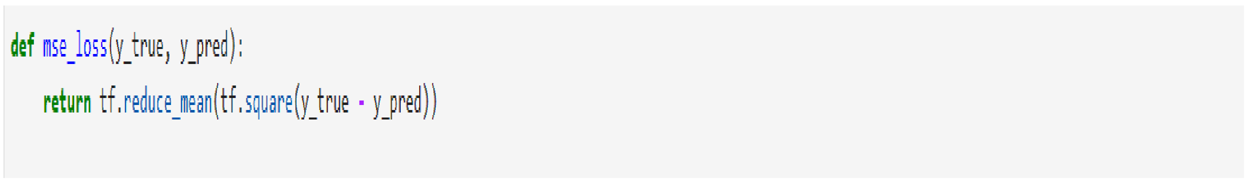


```
def build_color_correction_model(generator, discriminator):
    inputs = tf.keras.Input(shape=input_shape)
    generated_images = generator(inputs)
    discriminator.trainable = False
    validity = discriminator(generated_images)
    return Model(inputs, [generated_images, validity])
```

Build_color_correction_model(generator, discriminator)

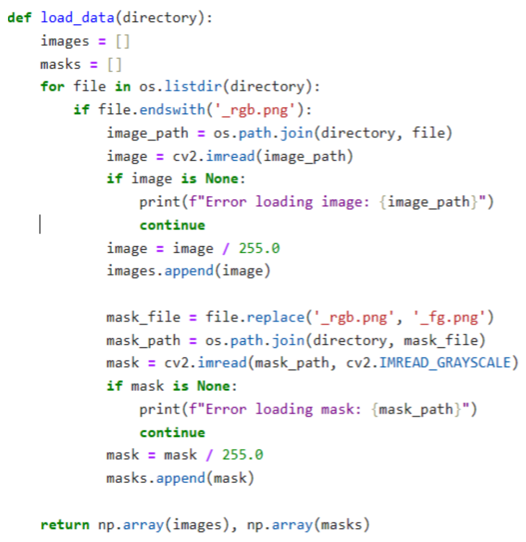
- This function creates a color correction model that integrates a generator and a discriminator.
- Input: The function takes a generator model (generator) and a discriminator model (discriminator) as inputs.
- Process: An input layer with the specified shape (input_shape) is created. The input is passed through the generator model to produce generated images.
- The discriminator is set to be non-trainable in this model to freeze its weights during training.
- The generated images are passed through the discriminator to obtain a validity score, which represents how real the generated images are.
- Outputs: The function returns a Keras model with two outputs: the generated images and their validity scope from the discriminator.

Purpose:



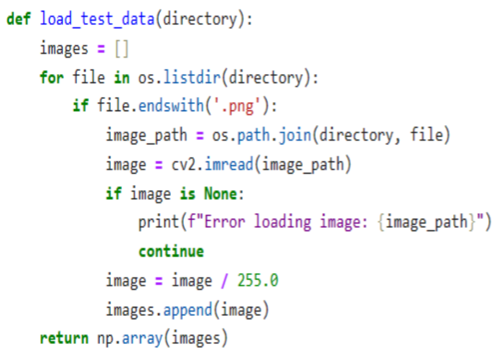
```
def mse_loss(y_true, y_pred):
    return tf.reduce_mean(tf.square(y_true - y_pred))
```

- This function computes the Mean Squared Error(MSE) loss between the true values and predicted values.
- The function calculates the squared differences between true and predicted values and then returns the mean of these sqauared differences as the loss.
  Load_data(directory)
- This function loads images and their corresponding masks from a specified directory.



```python
def load_data(directory):
    images = []
    masks = []
    for file in os.listdir(directory):
        if file.endswith('_rgb.png'):
            image_path = os.path.join(directory, file)
            image = cv2.imread(image_path)
            if image is None:
                print(f"Error loading image: {image_path}")
                continue
            image = image / 255.0
            images.append(image)

            mask_file = file.replace('_rgb.png', '_fg.png')
            mask_path = os.path.join(directory, mask_file)
            mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
            if mask is None:
                print(f"Error loading mask: {mask_path}")
                continue
            mask = mask / 255.0
            masks.append(mask)

    return np.array(images), np.array(masks)
```

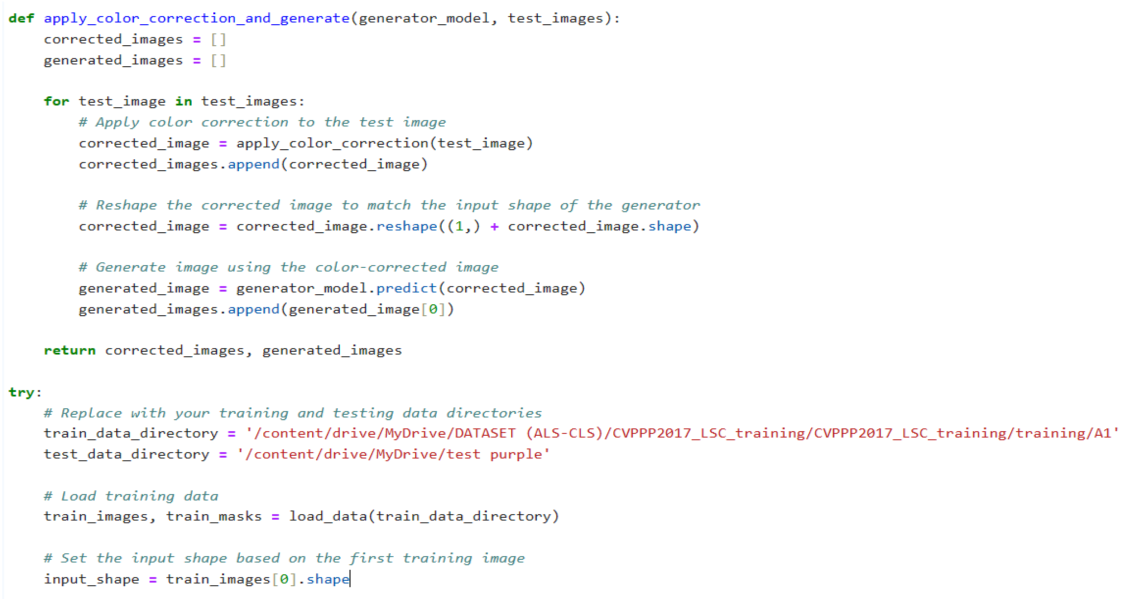
- It initializes empty lists for images and masks. Iterates through files in the given directory. If a file has the suffix _rgb.png, it reads the image using OpenCV and normalizes it by diving by 255. The image is then added to the list of images.
- For each image, it tries to load a corresponding mask file (with _fg.png suffix), which is read as grayscale and normalized by diving by 255.
- The function prints an error message if any image or mask cannot be loaded.
- Outputs: Returns the loaded images and masks as Numpy arrays.



```python
def load_test_data(directory):
    images = []
    for file in os.listdir(directory):
        if file.endswith('.png'):
            image_path = os.path.join(directory, file)
            image = cv2.imread(image_path)
            if image is None:
                print(f"Error loading image: {image_path}")
                continue
            image = image / 255.0
            images.append(image)
    return np.array(images)
```

Load_test_data(directory):

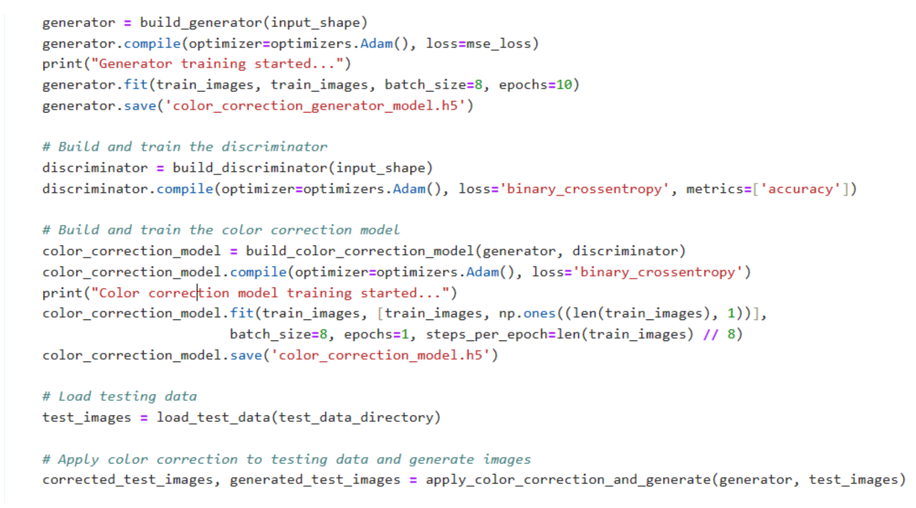
- This function loads test images from a specified directory. It reads images from files with .png extension and normalizes them by dividing pixel values by 255.
- Returns a NumPy array of loaded test images.



```python
def apply_color_correction_and_generate(generator_model, test_images):
    corrected_images = []
    generated_images = []

    for test_image in test_images:
        # Apply color correction to the test image
        corrected_image = apply_color_correction(test_image)
        corrected_images.append(corrected_image)

        # Reshape the corrected image to match the input shape of the generator
        corrected_image = corrected_image.reshape((1,) + corrected_image.shape)

        # Generate image using the color-corrected image
        generated_image = generator_model.predict(corrected_image)
        generated_images.append(generated_image[0])

    return corrected_images, generated_images

try:
    # Replace with your training and testing data directories
    train_data_directory = '/content/drive/MyDrive/DATASET (ALS-CLS)/CVPPP2017_LSC_training/CVPPP2017_LSC_training/training/A1'
    test_data_directory = '/content/drive/MyDrive/test purple'

    # Load training data
    train_images, train_masks = load_data(train_data_directory)

    # Set the input shape based on the first training image
    input_shape = train_images[0].shape
```

Generator_model:

- A trained generator model that takes image as input and produces color-corrected or transformed images as output.
- Test images are a list or array of images on which the function will apply color correction and image generation.
- Initializes empty lists to store corrected and generated images.
- For each test image calls the apply_color_correction() function to perform color correction on the test image.
- Appends the corrected image to the corrected_images list. Reshapes the corrected image to match the input shape of the generator model.
- Uses the generator model to predict (generate) a new image from the color-corrected image. Appends the first output from the generator model to the generated_images list.
- Outputs: Returns two lists- corrected_images containing the color-corrected images and generated_images containing the images generated by the generator model.
- The code initializes variables to hold training data directories. The load_data(train_data_directory) function call loads the training data, which includes both images and masks from the specified directory.
- This data is later used to train the generator and discriminator. The input_shape variable is set based on the shape of the first training image in the dataset. The shape will be used to build the models with compatible input shapes.



```
generator = build_generator(input_shape)
generator.compile(optimizer=optimizers.Adam(), loss=mse_loss)
print("Generator training started...")
generator.fit(train_images, train_images, batch_size=8, epochs=10)
generator.save('color_correction_generator_model.h5')

# Build and train the discriminator
discriminator = build_discriminator(input_shape)
discriminator.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy', metrics=['accuracy'])

# Build and train the color correction model
color_correction_model = build_color_correction_model(generator, discriminator)
color_correction_model.compile(optimizer=optimizers.Adam(), loss='binary_crossentropy')
print("Color correction model training started...")
color_correction_model.fit(train_images, [train_images, np.ones((len(train_images), 1))],
                       batch_size=8, epochs=1, steps_per_epoch=len(train_images) // 8)
color_correction_model.save('color_correction_model.h5')

# Load testing data
test_images = load_test_data(test_data_directory)

# Apply color correction to testing data and generate images
corrected_test_images, generated_test_images = apply_color_correction_and_generate(generator, test_images)
```

Building and Training the Generator:

- The code creates a generator model using the build_generator(input_shape) function. The input shape is specified based on the first training image.
- The generator model is compiled using the Adam optimizer and a Mean Squared Error(MSE) loss function.
- The generator model is trained using the training images as both inputs and targets (self-supervised learning) for a specified number of epochs and batch size.
- After training, the generator model is saved to a file named ‘color_correction_generator_model.h5’.

Building and Compiling the Discriminator:

- The code creates a discriminator model using the build_discriminator(input_shape) function.
- The discriminator is compiled using the Adam optimizer and a binary cross-entropy loss function with an accuracy metric.
- The discriminator is not trained separately in the provided code, but it is set up to be part of the training process for the color correction model.

Building and Training the Color Correction Model:

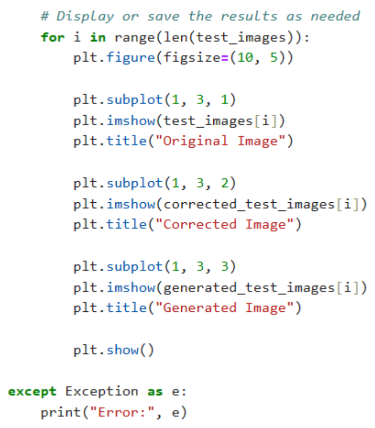
- A color correction model is built using build_color_correction_model(generator, discriminator), which combines the generator and discriminator.
- The color correction model is compiled using the Adam optimizer and a binary cross entropy loss function.
- The model is trained using the training images. The training data includes images as input and a list containing the same images(as target) and an array of ones (to represent the “real” label in GANs) as outputs.

- The training process includes a specified batch size, a single epoch, and a number of steps per epoch based on the training images and batch size.
- After training, the color correction model is saved to a file named 'color_correction_model.h5'.

  Loading Testing Data:
- The load_test_data(test_data_directory) function is called to load test images from the specified directory.
- The loaded images are used for further processing.

  Applying Color Correction and Generating Images:
- The apply_color_correction_and_generate(generator, test_images) function applies color correction to the test images using the apply_color_correction() function.
- The function uses the trained generator model to generate new images from the corrected test images.
- The corrected test images and generated images are returned and can be used

For evaluation:



```
    # Display or save the results as needed
    for i in range(len(test_images)):
        plt.figure(figsize=(10, 5))

        plt.subplot(1, 3, 1)
        plt.imshow(test_images[i])
        plt.title("Original Image")

        plt.subplot(1, 3, 2)
        plt.imshow(corrected_test_images[i])
        plt.title("Corrected Image")

        plt.subplot(1, 3, 3)
        plt.imshow(generated_test_images[i])
        plt.title("Generated Image")

        plt.show()

except Exception as e:
    print("Error:", e)
```

- ❿ Iteration through Test Images: The code iterates over the range of the length of test_images, which means it processes each test image.
- ❿ Creating a New Figure: For each test image, the code creates a new figure with a size of 10x5 inches.
- ❿ Displaying the Original Image: In the first subplot (position 1 out of 3 in a single row), the code displays the original test image using plt.imshow(test_images[i]). The subplot is titled “Original Image” using plt.title().
- ❿ Displaying the Corrected Image: In the second subplot (position 2 out of 3 in a single row), the code displays the color-corrected test image using plt.imshow(corrected_test_images[i]). The subplot is titled “Corrected Image” using plt.title().
- ❿ Displaying the Generated Image: In the third subplot (position 3 out of 3 in a single row), the code displays the image generated by the generator model using plt.imshow(generated_test_images[i]). The subplot is titled “Generated Image” using plt.title().
- ❿ Showing the Figure: After plotting the original, corrected, and generated images, the code displays the figure using plt.show(). This allows the user to see the images side by side for comparison.
- ❿ Exception Handling: If any exception occurs during the execution of the code, it is caught and printed using print(“Error:”,e).

```python
import lpips
from skimage.metrics import structural_similarity as ssim
import torch
import torchvision.transforms as transforms

def calculate_metrics(original_image_path, corrected_image_path, generated_image_path):
    # Load the images
    original_image = cv2.imread(original_image_path)
    corrected_image = cv2.imread(corrected_image_path)
    generated_image = cv2.imread(generated_image_path)

    # Check if any image failed to load
    if original_image is None or corrected_image is None or generated_image is None:
        print("Error: Failed to load one or more images.")
        return None, None

    # Resize images to a common size (optional)
    min_height = min(original_image.shape[0], corrected_image.shape[0], generated_image.shape[0])
    min_width = min(original_image.shape[1], corrected_image.shape[1], generated_image.shape[1])

    original_image = cv2.resize(original_image, (min_width, min_height))
    corrected_image = cv2.resize(corrected_image, (min_width, min_height))
    generated_image = cv2.resize(generated_image, (min_width, min_height))

    # Convert images to grayscale for SSIM calculation (assuming images are RGB)
    original_image_gray = cv2.cvtColor(original_image, cv2.COLOR_BGR2GRAY)
    generated_image_gray = cv2.cvtColor(generated_image, cv2.COLOR_BGR2GRAY)

    # Calculate SSIM
    ssim_score = ssim(original_image_gray, generated_image_gray)

    # Calculate LPIPS
    lpips_score = calculate_lpips(original_image, generated_image)

    return ssim_score, lpips_score
```

✦ The code imports necessary libraries:
lpips: A library for computing LPIPS scores.
ssim from skimage.metrics: A function for computing SSIM
scores.

✦ Function Definition (calculate_metrics):The function takes three arguments: paths to the original, corrected, and generated images.

✦ Image Loading: The code loads the original, corrected, and generated images using OpenCV's cv2.imread() function.

✦ Error Handling: It checks if any of the images fail to load. If any image fails to load, it prints an error message and returns None for both SSIM and LPIPS scores.

✦ Image Resizing (Optional): The images are resized to the minimum height and width among them. This step ensures that all images have the same dimensions, which might be necessary for certain similarity calculations.

Conversion to Grayscale (for SSIM): The original and generated images are converted to grayscale using OpenCV's cv2.cvtColor() function. This conversion is done because SSIM is typically calculated on grayscale images.

SSIM Calculation: The SSIM score is computed between the original and generated grayscale images using the ssim() function from skimage.metrics.

LPIPS Calculation: The LPIPS score is computed using the calculate_lpips() function, which is assumed to be defined elsewhere in the code. This function calculates the LPIPS score between the original and generated images.

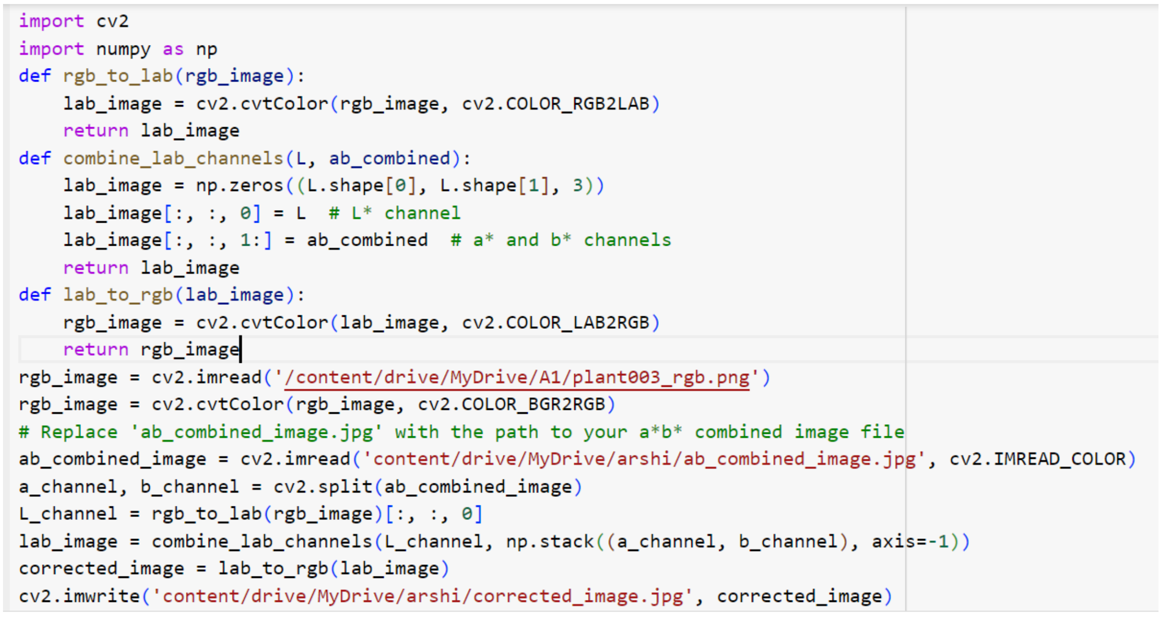
Return Values: The function returns the computed SSIM and LPIPS scores.

```python
def calculate_lpips(original_image, generated_image):
    # Ensure images are in float32 format and normalized to [0, 1]
    original_image = original_image.astype(np.float32) / 255.0
```

The Lpips Score is 0.568 and the SSIM value is 0.64

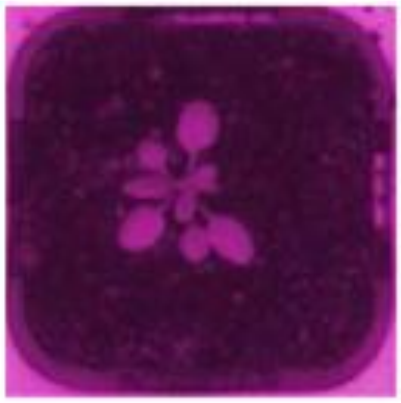
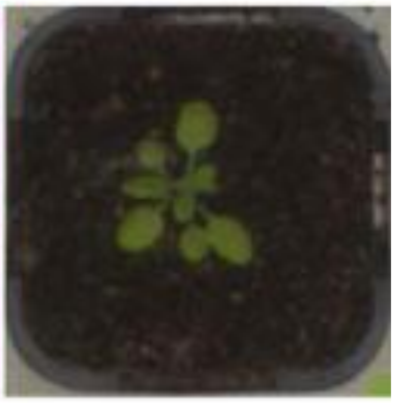
The original and generated images are normalized to the range [0, 1] by dividing them by 255.0 (assuming the images are represented as arrays of pixel values ranging from 0 to 255).
Conversion to PyTorch Tensors: The normalized images are converted from NumPy arrays to PyTorch tensors using transforms.ToTensor(). This transformation also adds a batch dimension to each image tensor using unsqueeze(0).
LPIPS Model Initialization: The LPIPS model is initialized with the AlexNet architecture (net='alex'). LPIPS models are designed to measure the perceptual similarity between images based on features extracted by deep neural networks.
Computing LPIPS Score: The LPIPS score is computed by passing the original and generated image tensors through the LPIPS model. The resulting score is returned as a scalar value using item().
Example Usage: An example usage of the calculate_lpips function is provided where paths to original, corrected, and generated images are passed to the function. After obtaining both SSIM (not defined in the provided code) and LPIPS scores, they are printed if both scores are not None.

GENERATING COLOR CORRECTED IMAGE USING CORRECTED a*b* CHANNELS AND COMBINING THEM WITH l* OF THE SAME IMAGE

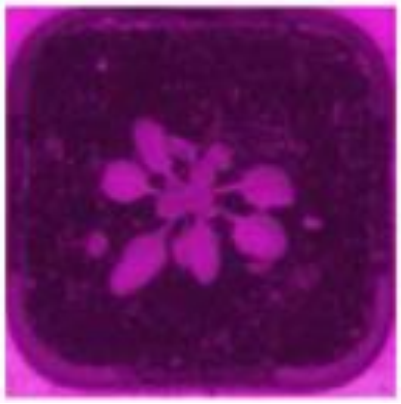
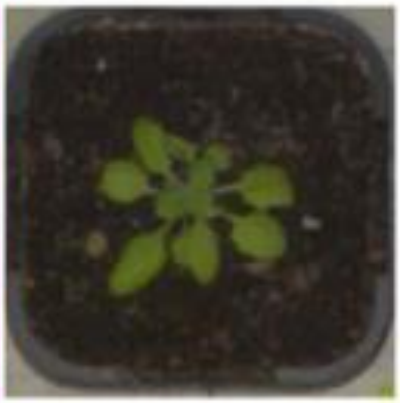


```python
import cv2
import numpy as np
def rgb_to_lab(rgb_image):
    lab_image = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2LAB)
    return lab_image
def combine_lab_channels(L, ab_combined):
    lab_image = np.zeros((L.shape[0], L.shape[1], 3))
    lab_image[:, :, 0] = L  # L* channel
    lab_image[:, :, 1:] = ab_combined  # a* and b* channels
    return lab_image
def lab_to_rgb(lab_image):
    rgb_image = cv2.cvtColor(lab_image, cv2.COLOR_LAB2RGB)
    return rgb_image
rgb_image = cv2.imread('/content/drive/MyDrive/A1/plant003_rgb.png')
rgb_image = cv2.cvtColor(rgb_image, cv2.COLOR_BGR2RGB)
# Replace 'ab_combined_image.jpg' with the path to your a*b* combined image file
ab_combined_image = cv2.imread('content/drive/MyDrive/arshi/ab_combined_image.jpg', cv2.IMREAD_COLOR)
a_channel, b_channel = cv2.split(ab_combined_image)
L_channel = rgb_to_lab(rgb_image)[:, :, 0]
lab_image = combine_lab_channels(L_channel, np.stack((a_channel, b_channel), axis=-1))
corrected_image = lab_to_rgb(lab_image)
cv2.imwrite('content/drive/MyDrive/arshi/corrected_image.jpg', corrected_image)
```

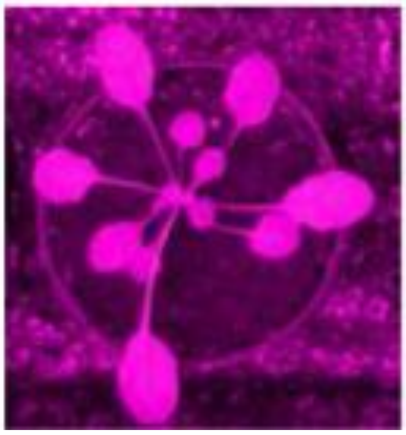
FINAL OUTPUT OF FIRST MODULE:

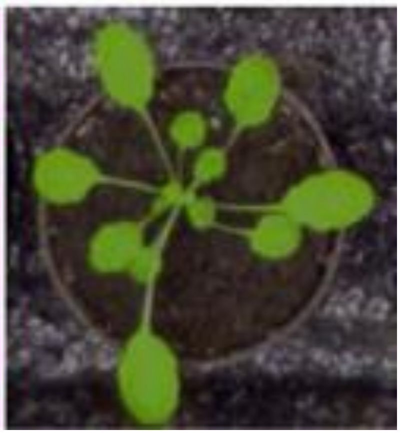



ORIGINAL TESTING IMAGE   COLOR CORRECTED IMAGE

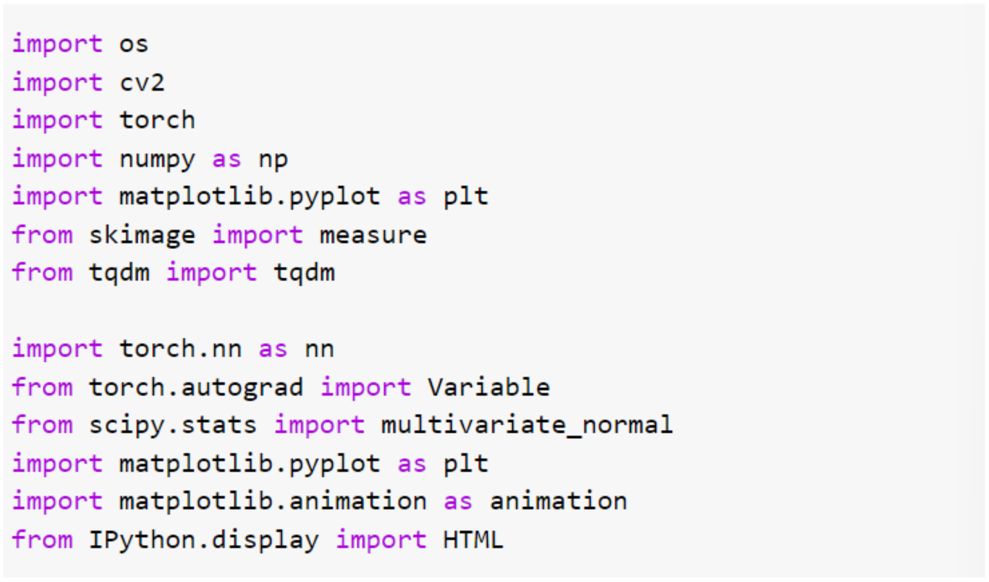
ORIGINAL TESTING IMAGE   COLOR CORRECTED IMAGE


ORIGINAL TESTING IMAGE


COLOR CORRECTED IMAGE

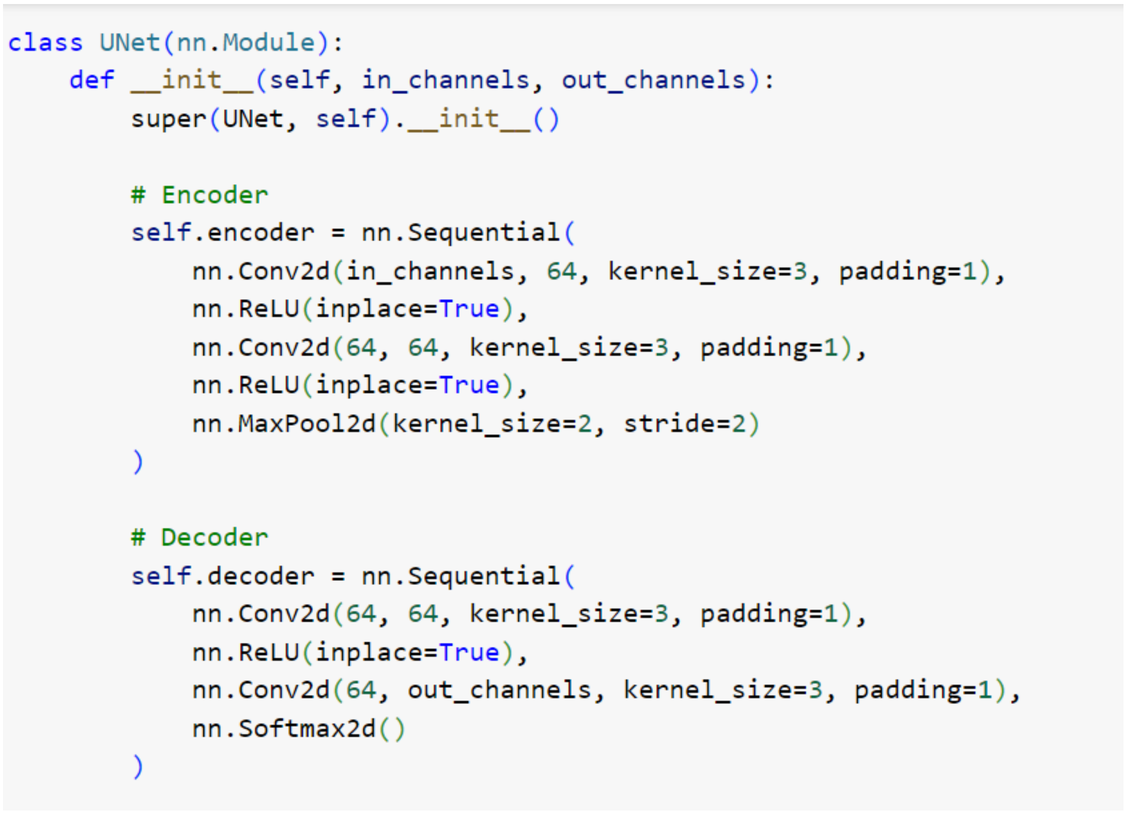
SELF-SUPERVISED LEAF SEGMENTATION (leafsegmenter.py)

1. Importing Python Libraries

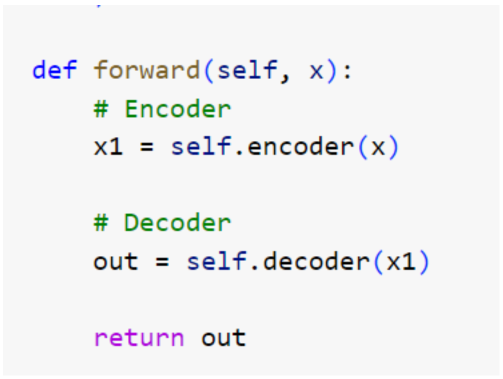


```python
import os
import cv2
import torch
import numpy as np
import matplotlib.pyplot as plt
from skimage import measure
from tqdm import tqdm

import torch.nn as nn
from torch.autograd import Variable
from scipy.stats import multivariate_normal
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
```

2.Defining U-Net class



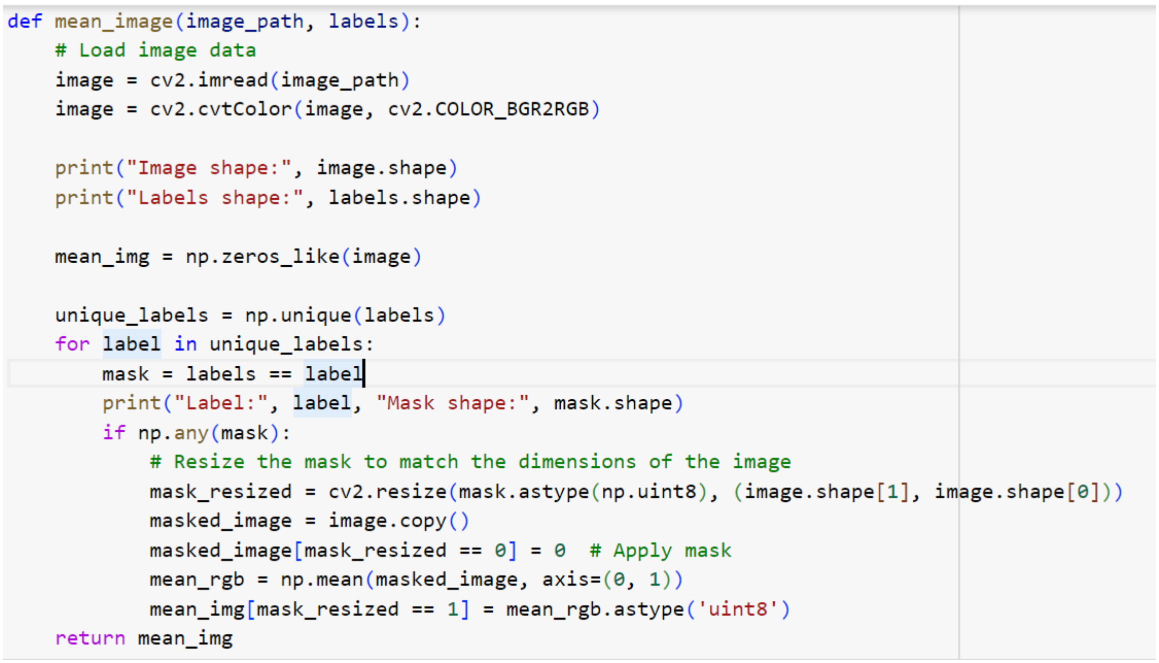
```python
class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Conv2d(in_channels, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, out_channels, kernel_size=3, padding=1),
            nn.Softmax2d()
        )
```



```python
    def forward(self, x):
        # Encoder
        x1 = self.encoder(x)

        # Decoder
        out = self.decoder(x1)

        return out
```

- Implementing a U-Net model for image segmentation, where the encoder and decoder extracts features from the input image and the generates a segmentation mask based on those features respectively.
- Encoder: Consists of two convolutional layers (each of 64 filters of 3x3 with padding of 1 pixel on both sides) followed by ReLU activation functions and max-pooling operations to extract features and reduce spatial dimensions.
- Decoder: Comprises of two convolutional layers (each of 64 filters of 3x3 with padding of 1 pixel on both sides) with ReLU activation functions to up-sample feature maps from the encoder and generate the segmentation output.
- Total of 10 Layers (4 convolution layers 2 each in encoder and decoder ,4 ReLU layers after each convolution layers ,max-pooling layer in encoder ,softmax in decoder)

3.Defining Mean Image calculation function:



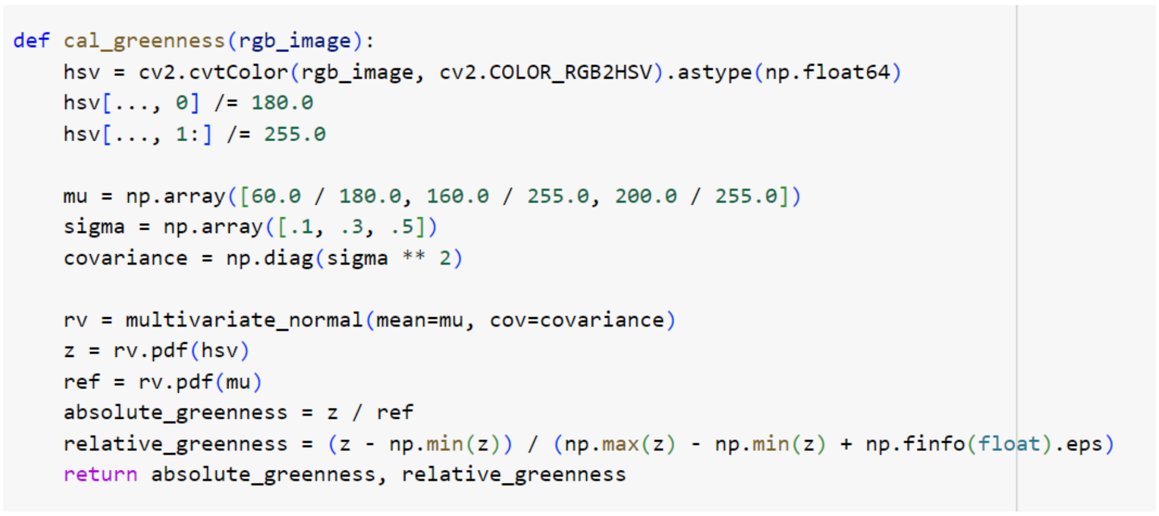
```python
def mean_image(image_path, labels):
    # Load image data
    image = cv2.imread(image_path)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    print("Image shape:", image.shape)
    print("Labels shape:", labels.shape)

    mean_img = np.zeros_like(image)

    unique_labels = np.unique(labels)
    for label in unique_labels:
        mask = labels == label
        print("Label:", label, "Mask shape:", mask.shape)
        if np.any(mask):
            # Resize the mask to match the dimensions of the image
            mask_resized = cv2.resize(mask.astype(np.uint8), (image.shape[1], image.shape[0]))
            masked_image = image.copy()
            masked_image[mask_resized == 0] = 0  # Apply mask
            mean_rgb = np.mean(masked_image, axis=(0, 1))
            mean_img[mask_resized == 1] = mean_rgb.astype('uint8')
    return mean_img
```

- Load Image: Reads the image from the specified path and converts it to RGB format.
- Initialize Mean Image: Creates an empty array with the same dimensions as the input image to store the mean RGB values.
- Compute Mean RGB: For each unique label in the label image, it computes the mean RGB value of the pixels in the input image corresponding to that

label. This is done by masking the input image with the label, resizing the mask to match the image dimensions, and then computing the mean RGB value of the masked pixels.

- Store Mean RGB: Sets the mean RGB value in the mean_img array for the pixels corresponding to each label.

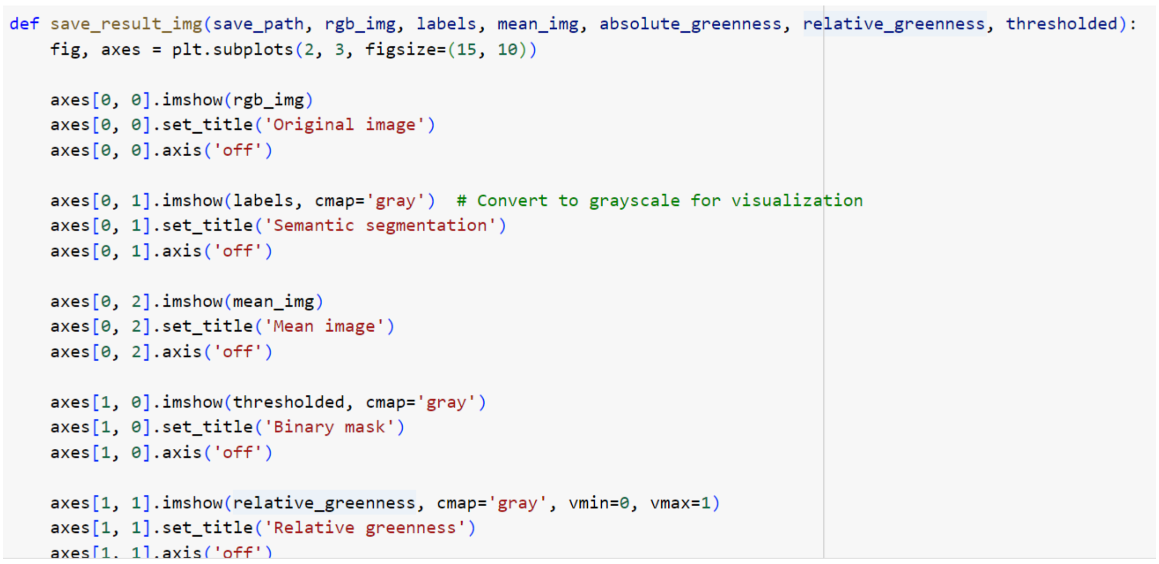
4.Defining calculating greenness function



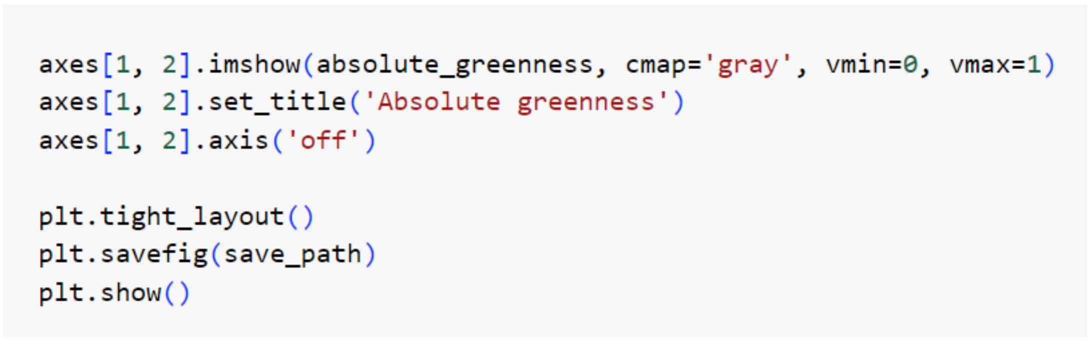
```python
def cal_greenness(rgb_image):
    hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV).astype(np.float64)
    hsv[..., 0] /= 180.0
    hsv[..., 1:] /= 255.0

    mu = np.array([60.0 / 180.0, 160.0 / 255.0, 200.0 / 255.0])
    sigma = np.array([.1, .3, .5])
    covariance = np.diag(sigma ** 2)

    rv = multivariate_normal(mean=mu, cov=covariance)
    z = rv.pdf(hsv)
    ref = rv.pdf(mu)
    absolute_greenness = z / ref
    relative_greenness = (z - np.min(z)) / (np.max(z) - np.min(z) + np.finfo(float).eps)
    return absolute_greenness, relative_greenness
```

- Convert to HSV: RGB image is converted to HSV color space and normalized.
- Define Mean and Covariance: Mean and covariance values are set to represent green color in HSV.
- Compute PDF: Probability density function (PDF) of green color is calculated for each pixel.
- Calculate Greenness: Absolute greenness is obtained by comparing each pixel's PDF to the mean PDF.
- Normalize: Relative greenness is scaled to [0, 1] range.
- Return: Absolute and relative greenness values are returned.

5.Defining a function to save the final result image which consists of binary mask,absolute greenness,relative greenness,original image,mean image,segmentation labels

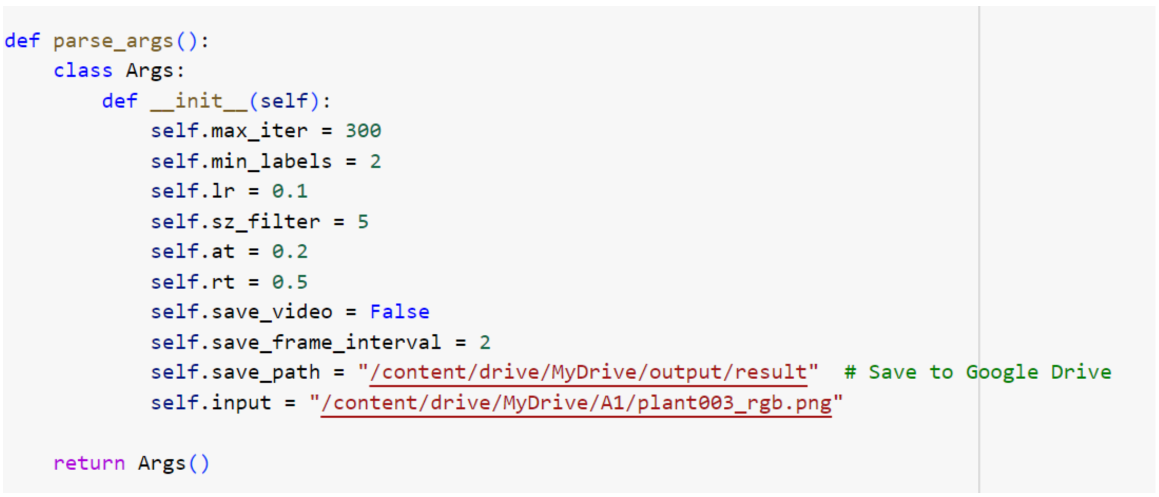


```
def save_result_img(save_path, rgb_img, labels, mean_img, absolute_greenness, relative_greenness, thresholded):
    fig, axes = plt.subplots(2, 3, figsize=(15, 10))

    axes[0, 0].imshow(rgb_img)
    axes[0, 0].set_title('Original image')
    axes[0, 0].axis('off')

    axes[0, 1].imshow(labels, cmap='gray')  # Convert to grayscale for visualization
    axes[0, 1].set_title('Semantic segmentation')
    axes[0, 1].axis('off')

    axes[0, 2].imshow(mean_img)
    axes[0, 2].set_title('Mean image')
    axes[0, 2].axis('off')

    axes[1, 0].imshow(thresholded, cmap='gray')
    axes[1, 0].set_title('Binary mask')
    axes[1, 0].axis('off')

    axes[1, 1].imshow(relative_greenness, cmap='gray', vmin=0, vmax=1)
    axes[1, 1].set_title('Relative greenness')
    axes[1, 1].axis('off')
```



```
    axes[1, 2].imshow(absolute_greenness, cmap='gray', vmin=0, vmax=1)
    axes[1, 2].set_title('Absolute greenness')
    axes[1, 2].axis('off')

    plt.tight_layout()
    plt.savefig(save_path)
    plt.show()
```

- Input Parameters: The function takes in several image-related data including the original RGB image (rgb_img), semantic segmentation labels (labels), mean image (mean_img), absolute greenness (absolute_greenness), relative greenness (relative_greenness), and a binary mask (thresholded).
- Visualization: It creates a subplot grid with 2 rows and 3 columns, adjusting the figure size accordingly. Each subplot displays a different image or visualization:
- Original RGB image,Semantic segmentation labels,Mean image,Binary mask,Relative greenness,Absolute greenness.
- Display and Save: The generated visualization is displayed using plt.show(), and then saved to the specified save_path using plt.savefig()
- 6.Creating an object with default parameter values

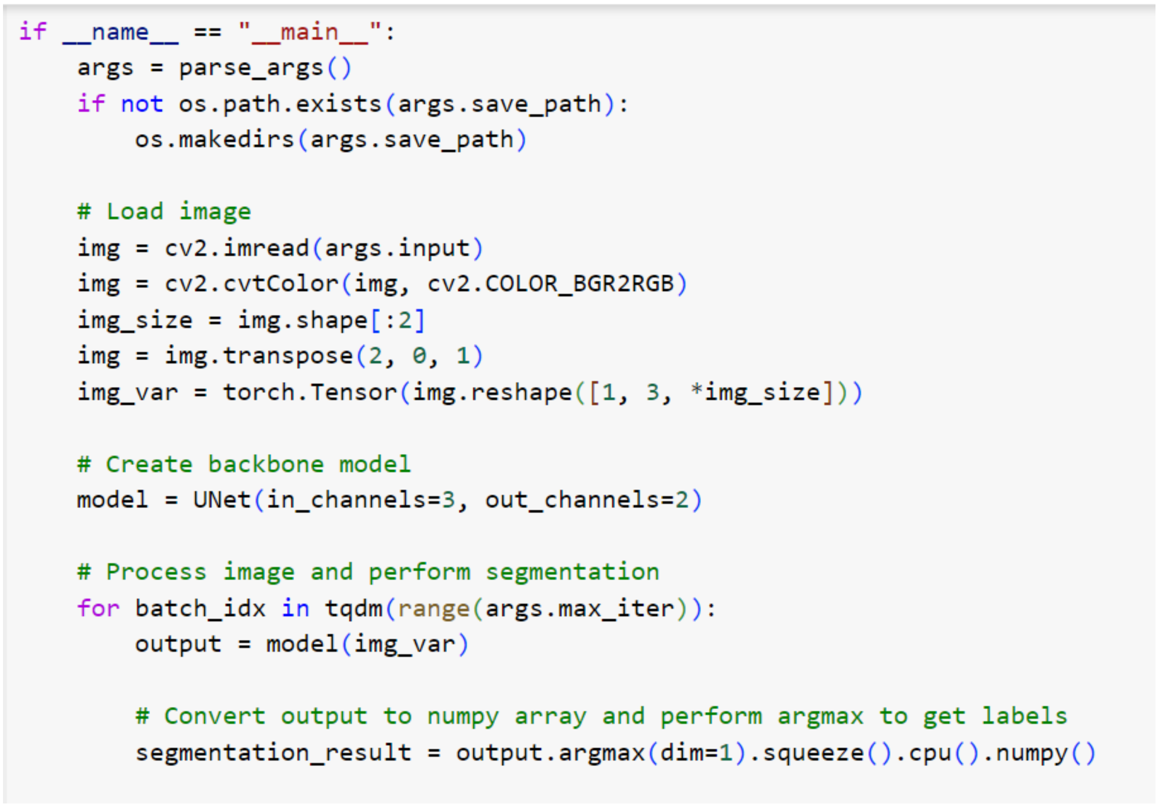


```python
def parse_args():
    class Args:
        def __init__(self):
            self.max_iter = 300
            self.min_labels = 2
            self.lr = 0.1
            self.sz_filter = 5
            self.at = 0.2
            self.rt = 0.5
            self.save_video = False
            self.save_frame_interval = 2
            self.save_path = "/content/drive/MyDrive/output/result"  # Save to Google Drive
            self.input = "/content/drive/MyDrive/A1/plant003_rgb.png"

    return Args()
```

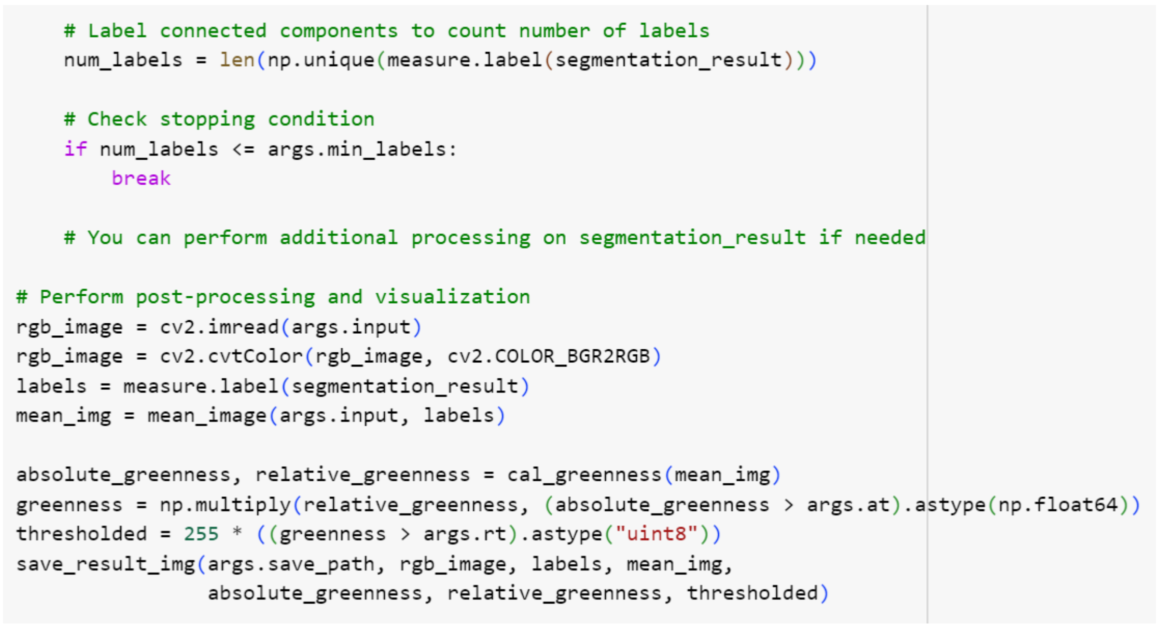
- Attributes: The attributes represent parameters such as max_iter, min_labels, lr, sz_filter, at, rt, save_video, save_frame_interval, save_path, and input.
- Default Values: Each attribute is initialized with a default value.
- Class Method: The class has a single method __init__ which initializes these attributes with default values.
- Return: Finally, an instance of the Args class with initialized attributes is returned from the function.

7.Defining main function

It performs image segmentation, analyzes the segmented regions, and saves the results

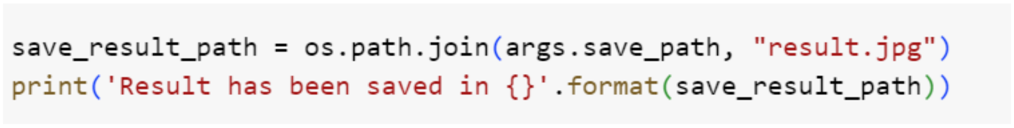


```python
if __name__ == "__main__":
    args = parse_args()
    if not os.path.exists(args.save_path):
        os.makedirs(args.save_path)

    # Load image
    img = cv2.imread(args.input)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img_size = img.shape[:2]
    img = img.transpose(2, 0, 1)
    img_var = torch.Tensor(img.reshape([1, 3, *img_size]))

    # Create backbone model
    model = UNet(in_channels=3, out_channels=2)

    # Process image and perform segmentation
    for batch_idx in tqdm(range(args.max_iter)):
        output = model(img_var)

        # Convert output to numpy array and perform argmax to get labels
        segmentation_result = output.argmax(dim=1).squeeze().cpu().numpy()
```



```
    # Label connected components to count number of labels
    num_labels = len(np.unique(measure.label(segmentation_result)))

    # Check stopping condition
    if num_labels <= args.min_labels:
        break

    # You can perform additional processing on segmentation_result if needed

# Perform post-processing and visualization
rgb_image = cv2.imread(args.input)
rgb_image = cv2.cvtColor(rgb_image, cv2.COLOR_BGR2RGB)
labels = measure.label(segmentation_result)
mean_img = mean_image(args.input, labels)

absolute_greenness, relative_greenness = cal_greenness(mean_img)
greenness = np.multiply(relative_greenness, (absolute_greenness > args.at).astype(np.float64))
thresholded = 255 * ((greenness > args.rt).astype("uint8"))
save_result_img(args.save_path, rgb_image, labels, mean_img,
                absolute_greenness, relative_greenness, thresholded)
```

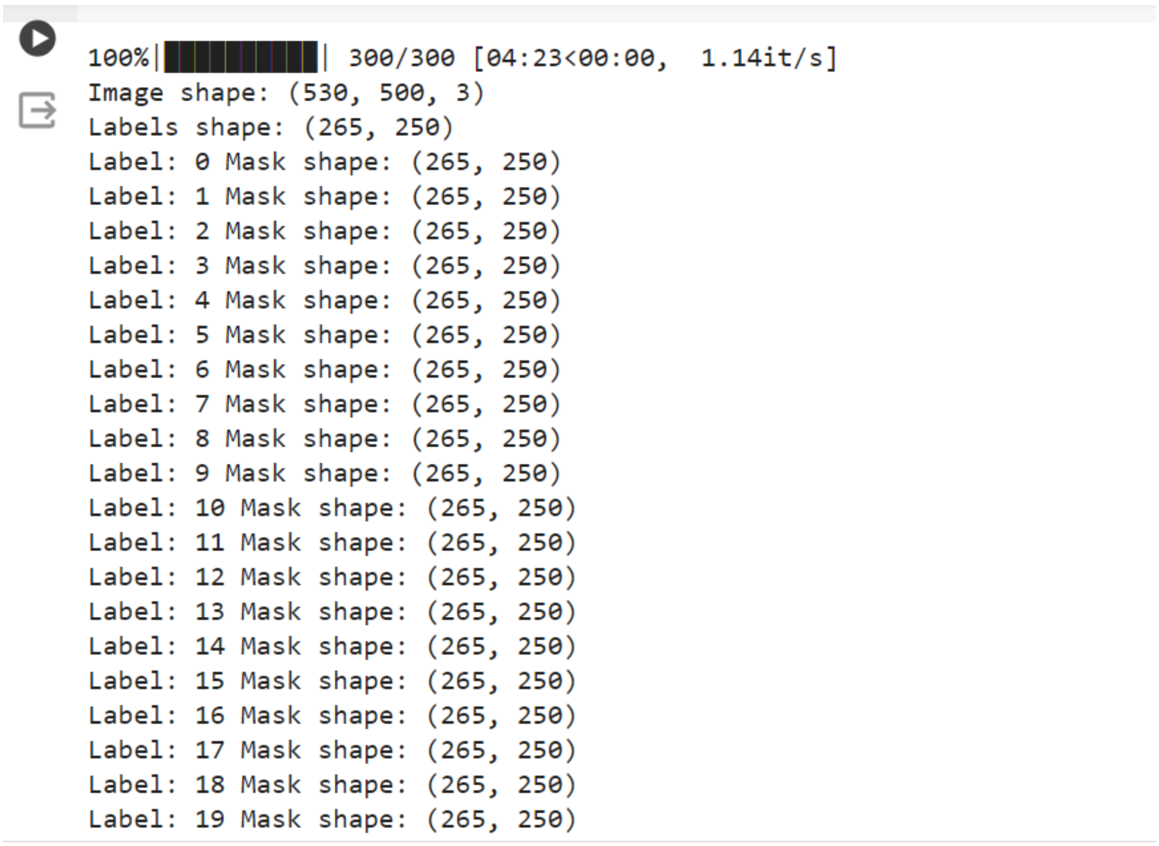
- Parsing Arguments: It initializes a set of parameters using the parse_args function.
- Creating Output Directory: Checks if the directory specified by args.save_path exists, and creates it if not.
- Loading and Preprocessing Image: Loads the input image, converts it to RGB format, and reshapes it for further processing.
- Model Initialization: Initializes an instance of the UNet model for image segmentation.
- Image Segmentation: Iteratively processes the image using the UNet model to perform segmentation. The process continues for a maximum number of iterations specified by args.max_iter, or until the number of labels in the segmentation result falls below a threshold specified by args.min_labels.



```
save_result_path = os.path.join(args.save_path, "result.jpg")
print('Result has been saved in {}'.format(save_result_path))
```

Post-processing and Visualization: Once segmentation is complete, it performs post-processing steps such as labeling connected components, calculating mean image, and computing greenness metrics. Finally, it visualizes and saves the results using the save_result_img function.

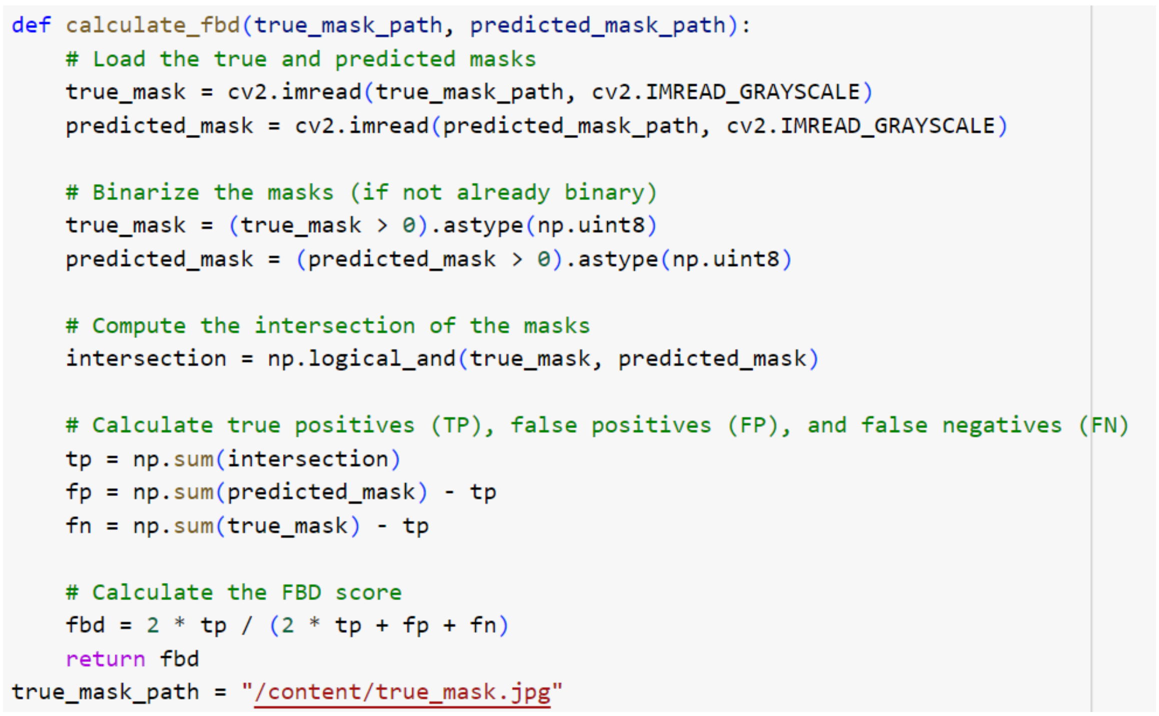
8.Running the python file



```
100%|██████████| 300/300 [04:23<00:00,  1.14it/s]
Image shape: (530, 500, 3)
Labels shape: (265, 250)
Label: 0 Mask shape: (265, 250)
Label: 1 Mask shape: (265, 250)
Label: 2 Mask shape: (265, 250)
Label: 3 Mask shape: (265, 250)
Label: 4 Mask shape: (265, 250)
Label: 5 Mask shape: (265, 250)
Label: 6 Mask shape: (265, 250)
Label: 7 Mask shape: (265, 250)
Label: 8 Mask shape: (265, 250)
Label: 9 Mask shape: (265, 250)
Label: 10 Mask shape: (265, 250)
Label: 11 Mask shape: (265, 250)
Label: 12 Mask shape: (265, 250)
Label: 13 Mask shape: (265, 250)
Label: 14 Mask shape: (265, 250)
Label: 15 Mask shape: (265, 250)
Label: 16 Mask shape: (265, 250)
Label: 17 Mask shape: (265, 250)
Label: 18 Mask shape: (265, 250)
Label: 19 Mask shape: (265, 250)
```

Output image is saved in result.png at the end

FBD Calculation:



```python
def calculate_fbd(true_mask_path, predicted_mask_path):
    # Load the true and predicted masks
    true_mask = cv2.imread(true_mask_path, cv2.IMREAD_GRAYSCALE)
    predicted_mask = cv2.imread(predicted_mask_path, cv2.IMREAD_GRAYSCALE)

    # Binarize the masks (if not already binary)
    true_mask = (true_mask > 0).astype(np.uint8)
    predicted_mask = (predicted_mask > 0).astype(np.uint8)

    # Compute the intersection of the masks
    intersection = np.logical_and(true_mask, predicted_mask)

    # Calculate true positives (TP), false positives (FP), and false negatives (FN)
    tp = np.sum(intersection)
    fp = np.sum(predicted_mask) - tp
    fn = np.sum(true_mask) - tp

    # Calculate the FBD score
    fbd = 2 * tp / (2 * tp + fp + fn)
    return fbd
true_mask_path = "/content/true_mask.jpg"
true_mask_path = "/content/true_mask.jpg"
predicted_mask_path = "/content/predicted_mask.jpg"

fbd_score = calculate_fbd(true_mask_path, predicted_mask_path)
print("FBD score:", fbd_score*100)
```

- Input Parameters: The function takes two parameters true_mask_path and predicted_mask_path, which are file paths to the true mask and the predicted mask images, respectively.
- Image Loading and Binarization: It loads the true and predicted masks using OpenCV's imread function, converting them to grayscale. Then, it binarizes the masks by thresholding, ensuring they are binary (values either 0 or 1).
- Intersection Calculation: It computes the intersection of the true and predicted masks using logical AND operation..
- FBD Score Calculation: Finally, it computes the FBD score using the formula:

$$\frac{2TP}{2 \times TP + FP + FN}$$

FBD Value :

```
FBD score: 89.79228121927237
```

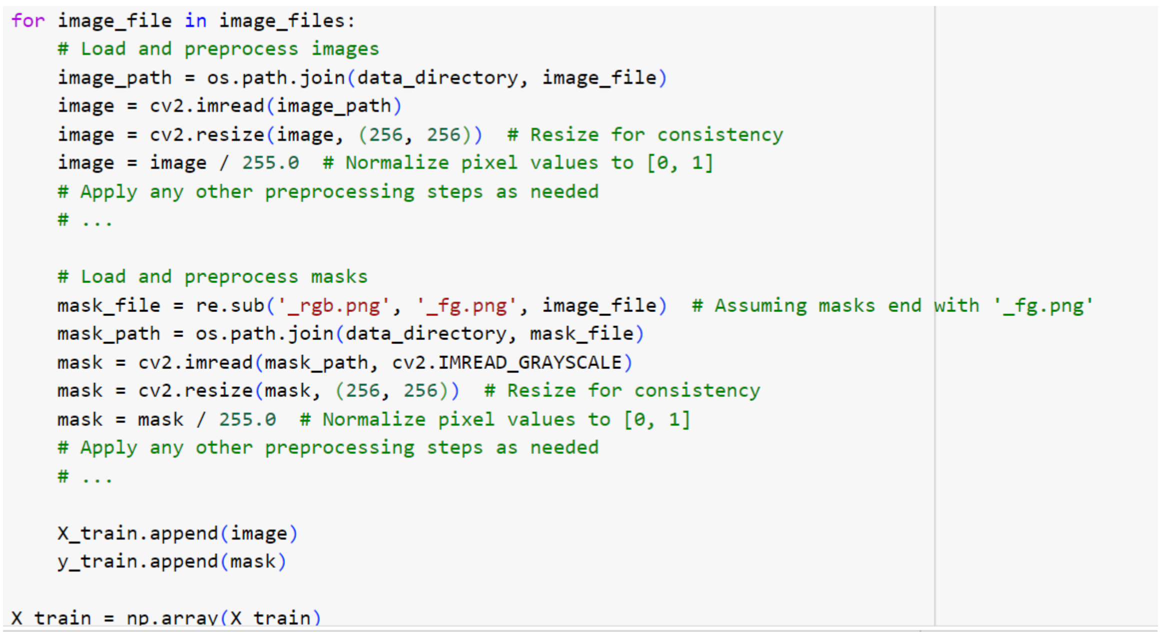
U-Net MODEL (Supervised Learning):

We also trained a U-Net Model using Supervised Learning just to compare results between Supervised and Self-Supervised

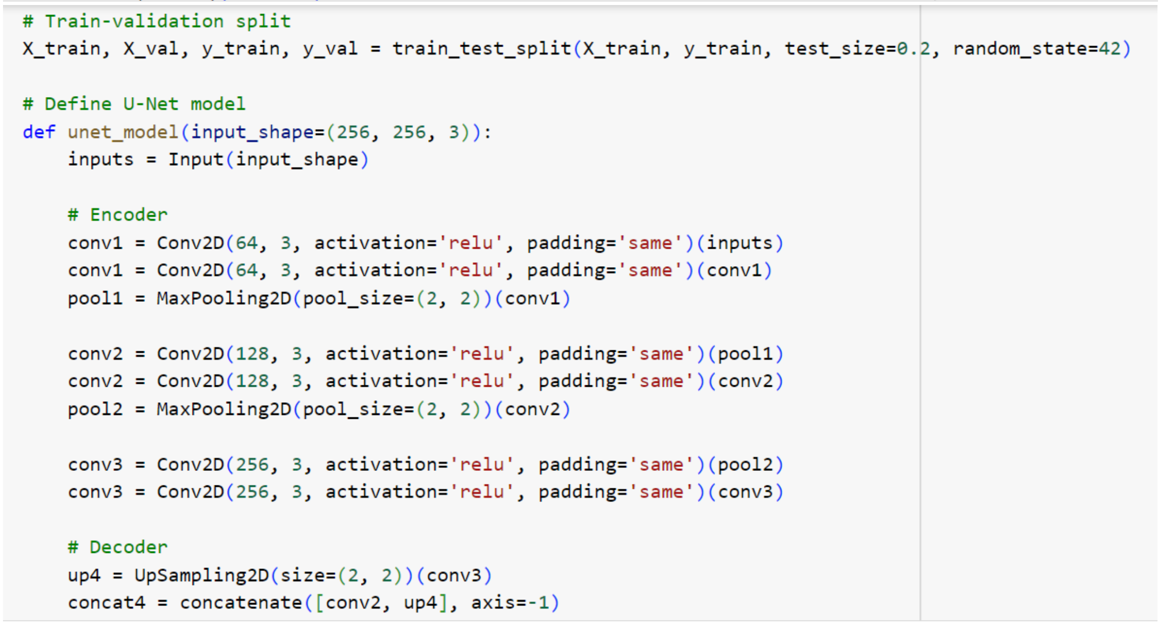
CODE:

Training the model:

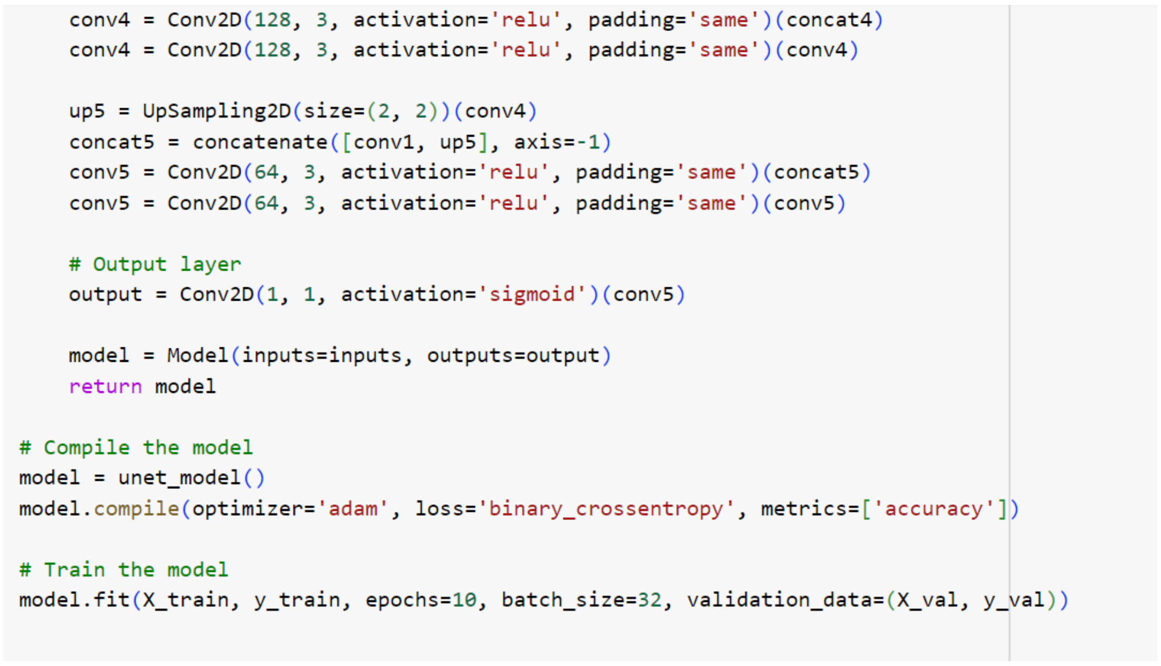
```python
import os
import cv2
import numpy as np
import re
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Set your data directory
data_directory = '/content/drive/MyDrive/DATASET (ALS-CLS)/CVPPP2017_LSC_training/CVPPP2017_LSC_training/training/A1'

# Get a list of all image files in the directory
image_files = [f for f in os.listdir(data_directory) if f.endswith('_rgb.png')]

# Sort the files to ensure images and masks are in the same order
image_files.sort()

# Load and preprocess images and masks
X_train = []
y_train = []
```



```python
for image_file in image_files:
    # Load and preprocess images
    image_path = os.path.join(data_directory, image_file)
    image = cv2.imread(image_path)
    image = cv2.resize(image, (256, 256))  # Resize for consistency
    image = image / 255.0  # Normalize pixel values to [0, 1]
    # Apply any other preprocessing steps as needed
    # ...

    # Load and preprocess masks
    mask_file = re.sub('_rgb.png', '_fg.png', image_file)  # Assuming masks end with '_fg.png'
    mask_path = os.path.join(data_directory, mask_file)
    mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
    mask = cv2.resize(mask, (256, 256))  # Resize for consistency
    mask = mask / 255.0  # Normalize pixel values to [0, 1]
    # Apply any other preprocessing steps as needed
    # ...

    X_train.append(image)
    y_train.append(mask)

X_train = np.array(X_train)
```

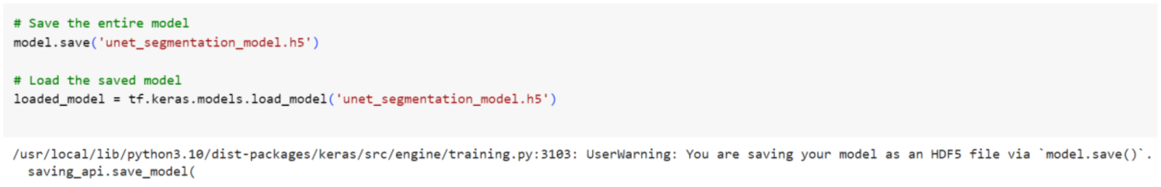


```python
# Train-validation split
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

# Define U-Net model
def unet_model(input_shape=(256, 256, 3)):
    inputs = Input(input_shape)

    # Encoder
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(inputs)
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(128, 3, activation='relu', padding='same')(pool1)
    conv2 = Conv2D(128, 3, activation='relu', padding='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    conv3 = Conv2D(256, 3, activation='relu', padding='same')(pool2)
    conv3 = Conv2D(256, 3, activation='relu', padding='same')(conv3)

    # Decoder
    up4 = UpSampling2D(size=(2, 2))(conv3)
    concat4 = concatenate([conv2, up4], axis=-1)
```



```
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(concat4)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(conv4)

    up5 = UpSampling2D(size=(2, 2))(conv4)
    concat5 = concatenate([conv1, up5], axis=-1)
    conv5 = Conv2D(64, 3, activation='relu', padding='same')(concat5)
    conv5 = Conv2D(64, 3, activation='relu', padding='same')(conv5)

    # Output layer
    output = Conv2D(1, 1, activation='sigmoid')(conv5)

    model = Model(inputs=inputs, outputs=output)
    return model

# Compile the model
model = unet_model()
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_val, y_val))
```

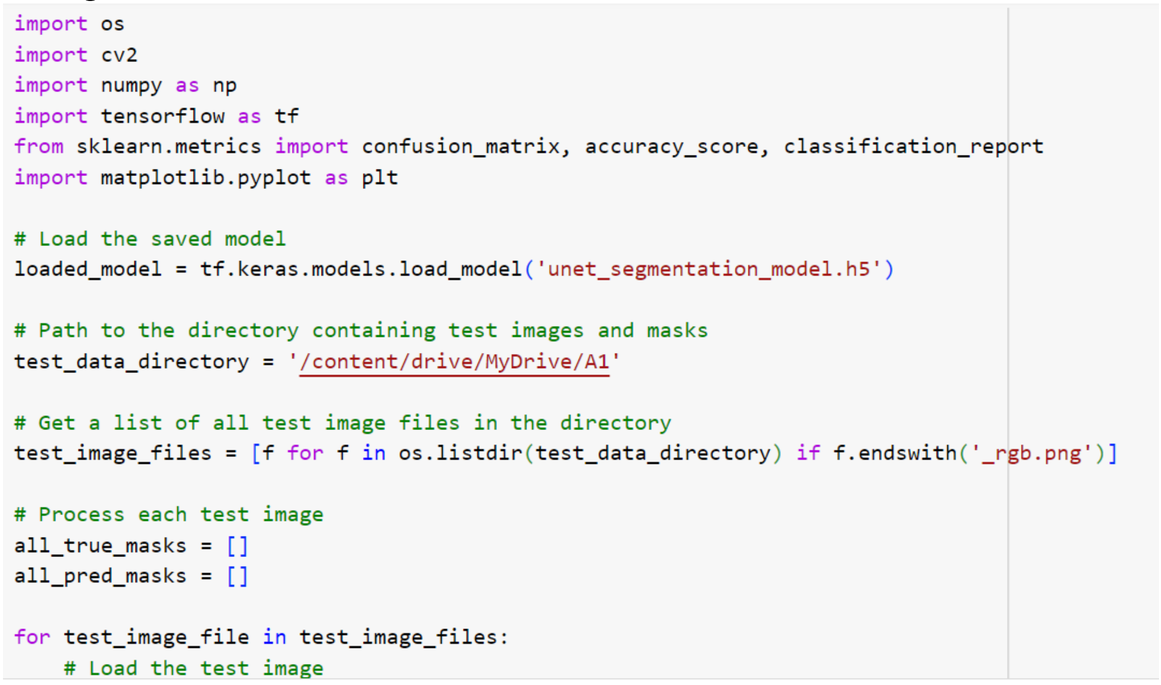
Saving the model :



```
# Save the entire model
model.save('unet_segmentation_model.h5')

# Load the saved model
loaded_model = tf.keras.models.load_model('unet_segmentation_model.h5')
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103: UserWarning: You are saving your model as an HDF5 file via `model.save()`.
  saving_api.save_model(
```

Testing the model:

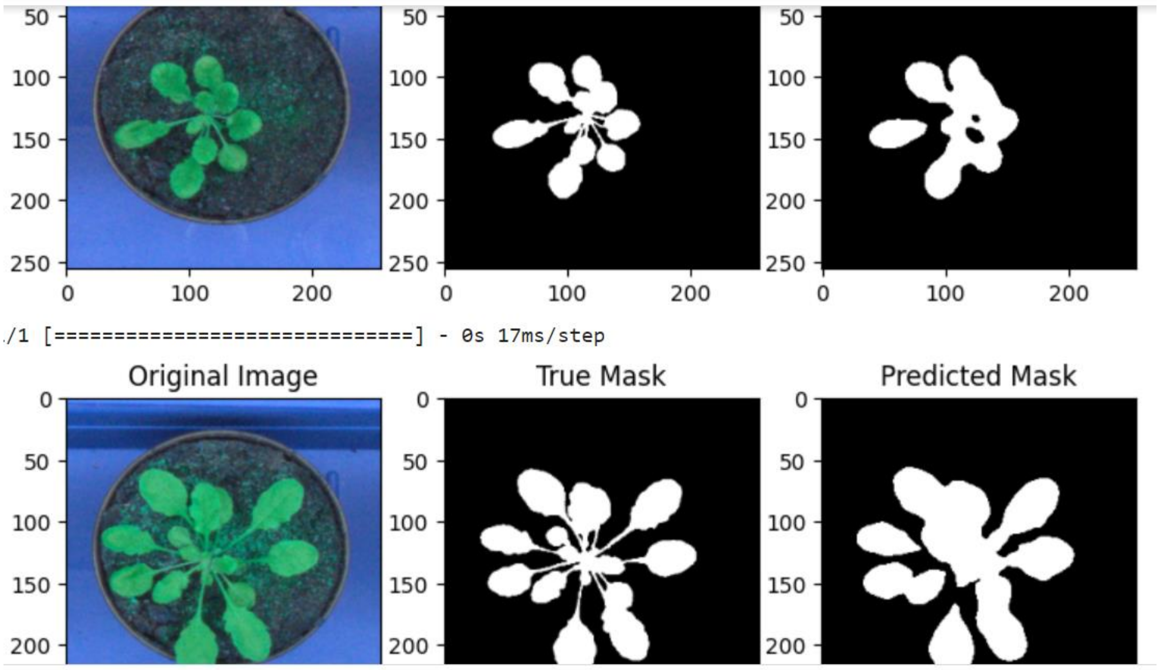


```
import os
import cv2
import numpy as np
import tensorflow as tf
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
import matplotlib.pyplot as plt

# Load the saved model
loaded_model = tf.keras.models.load_model('unet_segmentation_model.h5')

# Path to the directory containing test images and masks
test_data_directory = '/content/drive/MyDrive/A1'

# Get a list of all test image files in the directory
test_image_files = [f for f in os.listdir(test_data_directory) if f.endswith('_rgb.png')]

# Process each test image
all_true_masks = []
all_pred_masks = []

for test_image_file in test_image_files:
    # Load the test image
```

```
test_image_path = os.path.join(test_data_directory, test_image_file)
test_image = cv2.imread(test_image_path)
test_image = cv2.resize(test_image, (256, 256))  # Resize for consistency
test_image = test_image / 255.0  # Normalize pixel values to [0, 1]

# Make predictions on the test image
predictions = loaded_model.predict(np.expand_dims(test_image, axis=0))

# Load the true mask
true_mask_file = test_image_file.replace('_rgb.png', '_fg.png')
true_mask_path = os.path.join(test_data_directory, true_mask_file)
true_mask = cv2.imread(true_mask_path, cv2.IMREAD_GRAYSCALE)
true_mask = cv2.resize(true_mask, (256, 256))  # Resize for consistency
true_mask = true_mask / 255.0  # Normalize pixel values to [0, 1]

# Example post-processing (thresholding)
threshold = 0.5  # Adjust as needed
pred_mask = (predictions > threshold).astype(np.uint8).squeeze()

# Save the true and predicted masks for later evaluation
all_true_masks.append(true_mask)
all_pred_masks.append(pred_mask)
```

```
# Visualize the results
plt.figure(figsize=(12, 6))

plt.subplot(1, 4, 1)
plt.imshow(test_image)
plt.title('Original Image')

plt.subplot(1, 4, 2)
plt.imshow(true_mask, cmap='gray')
plt.title('True Mask')

plt.subplot(1, 4, 3)
plt.imshow(pred_mask.squeeze(), cmap='gray')  # Display the predicted mask
plt.title('Predicted Mask')

plt.show()
```

Output:



```
/1 [==============================] - 0s 17ms/step
```

Fig 3.1: Output of leaf segemntation model under supervised learning

FBD Calculation for U-Net using Supervised :

```
FBD score: 87.15623789354679
```

So Self-supervised Learning Performs better than supervised

CHAPTER 4
OUTPUT SNAPSHOTS

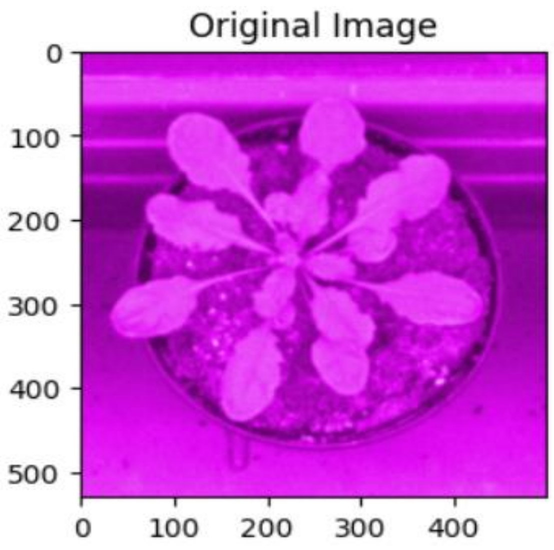


Fig 4.1: Purple Image from Test Dataset used for Model Testing

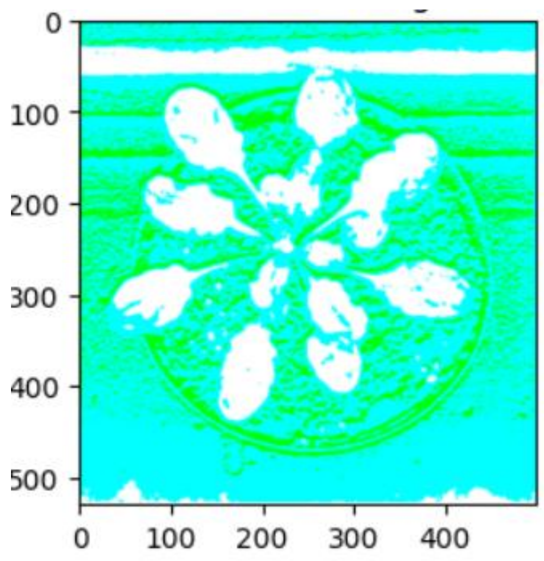


Fig 4.2: Generator model produced a*b* color space by taking l* as input

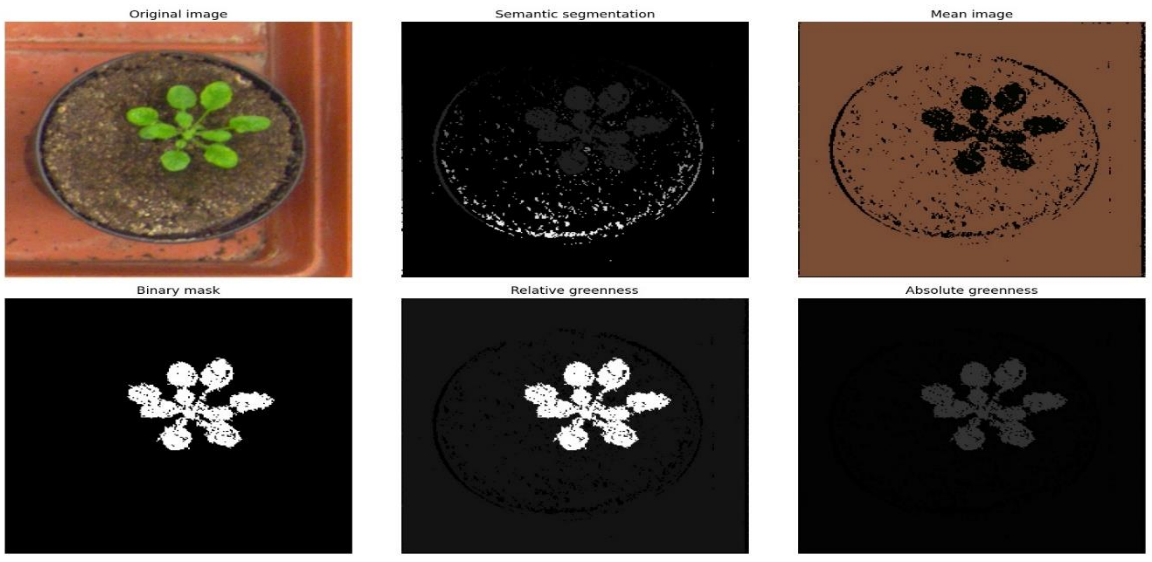


Fig 4.3: The Image consists of the output for the leaf segmentation under Self-Supervised Learning

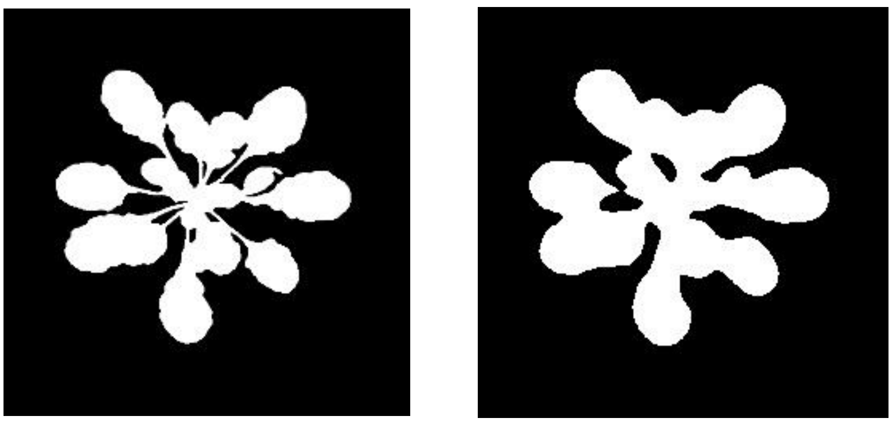
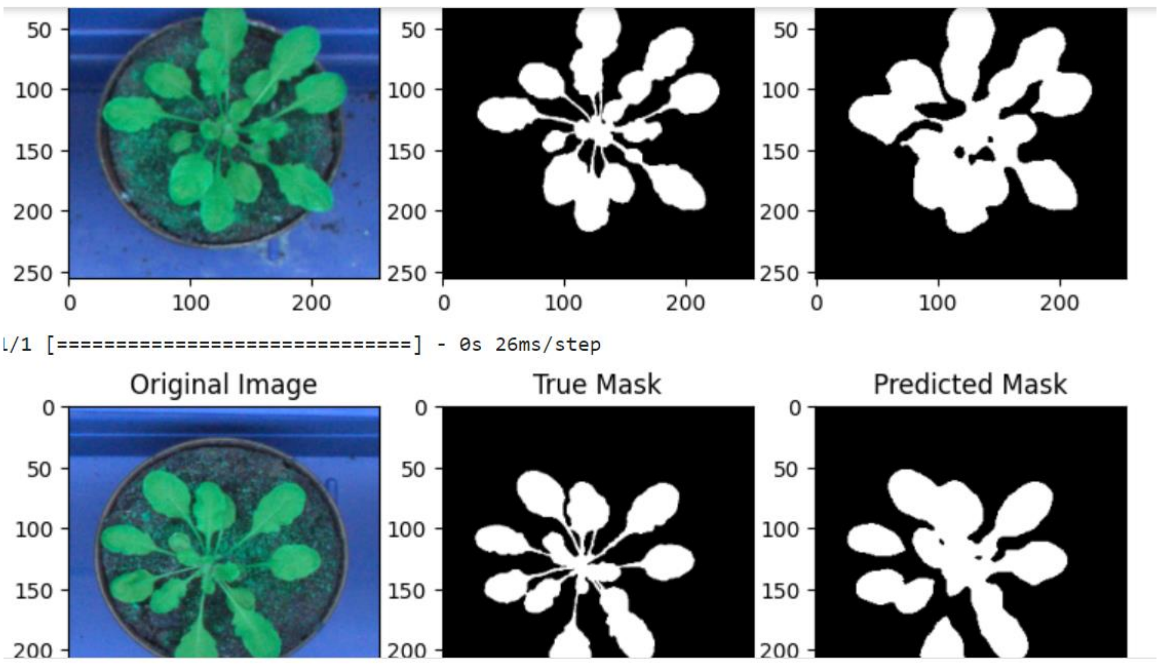


Fig 4.4: Ground Truth mask and Predicted Binary Mask used for FBD calculation (self-supervised Learning)



```
1/1 [==============================] - 0s 26ms/step
```

Fig 4.5: Ground Truth mask and Predicted Binary Mask used for FBD calculation (self-supervised Learning)

Dataset	Supervised	Self_Supervised
A1	87.1	89.7
A2	84.7	87.6
A3	85.9	89.8
A4	84.4	78.3

Table 4.6: Leaf Segmentation Result in terms of FBD (%) of Purple Testing Dataset

CHAPTER 5
CONCLUSION AND FUTURE PLANS

In our study, we introduced a self-supervised leaf segmentation framework that operates without the need for annotated training data, effectively delineating leaf regions from complex backgrounds under varying lighting conditions. Through extensive testing on both our Cannabis dataset and the CVPPP LSC dataset, we demonstrated the superior performance of our proposed approach. While capable of accurately segmenting leaf regions across diverse plant species and lighting scenarios, there remains scope for enhancement. Notably, the segmentation process currently requires 20-30 seconds to process an image of dimensions 512x512 pixels, prompting further optimization efforts.

Future Plan:

Implement heuristic early stopping criteria and pre-trained weights initialization to streamline training and improve segmentation accuracy. Develop self-supervised instance-level leaf segmentation by leveraging semantic segmentation results and incorporating mask outputs for precise object delineation. Investigate novel architectures like transformer-based models and innovative loss functions to further advance object segmentation and representation learning.Extend methodology to diverse domains such as medical imaging and robotics for broader impact and innovation.Include CRF for better segmentation results at the end.We would like to work on a machine with an Nvidia GTX 2080Ti GPU. This is what we would like to work on.

.

CHAPTER 6
REFERENCES

- Mishra, Anand Muni, Prabhjot Kaur, Mukund Pratap Singh, and Santar Pal Singh. **"A self-supervised overlapped multiple weed and crop leaf segmentation approach under complex light condition**." Multimedia Tools and Applications (2024): 1-26.

- Li, Yinglun, Xiaohai Zhan, Shouyang Liu, Hao Lu, Ruibo Jiang, Wei Guo, Scott Chapman et al. "**Self-supervised plant phenotyping by combining domain adaptation with 3D plant model simulations: application to wheat leaf counting at seedling stage**." Plant Phenomics 5 (2023): 0041.

- Minervini, Massimo, Andreas Fischbah, Hanno Scharr, and Sotirios A. Tsaftaris. "**Finely-grained annotated datasets for image-based plant phenotyping.**" Pattern recognition letters 81 (2016): 80-89.

- He, Hongjie, Hongzhang Xu, Ying Zhang, Kyle Gao, Huxiong Li, Lingfei Ma, and Jonathan Li. "**Mask R-CNN based automated identification and extraction of oil well sites**." International Journal of Applied Earth Observation and Geoinformation 112 (2022): 102875.

- Ogidi, Franklin C., Mark G. Eramian, and Ian Stavness. **"Benchmarking Self-Supervised Contrastive Learning Methods for Image-Based Plant Phenotyping**." Plant Phenomics 5 (2023): 0037.

- Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. "**U-net: Convolutional networks for biomedical image segmentation**." In Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October *5-9, 2015,* Proceedings, Part III 18, pp. 234-241. Springer International Publishing, 2015.

CHAPTER 7
APPENDIX

BASE PAPER:

Lin, Xufeng, Chang-Tsun Li, Scott Adams, Abbas Z. Kouzani, Richard Jiang, Ligang He, Yongjian Hu et al. "Self-supervised leaf segmentation under complex lighting conditions." *Pattern Recognition* 135 (2023): 109021.

Doi: 10.1016/j.patcog.2022.109021

keywords: {Self-supervised learning Convolutional neural networks Image-based plant phenotyping Leaf segmentation Color correction}

URL:
https://www.sciencedirect.com/science/article/pii/S0031320322005015