

image-captioning-for-smart-farming

July 4, 2024

```
[1]: from google.colab import files  
files.upload()  
  
<IPython.core.display.HTML object>  
Saving kaggle.json to kaggle.json  
  
[1]: {'kaggle.json':  
      b'{"username":"aloksingh077","key":"8d905146f1b8f911e33ecbb0a93e9869"}'}  
  
[2]: import os  
  
# Set the KAGGLE_CONFIG_DIR environment variable to point to the /content  
# directory.  
# This tells the Kaggle API where to find the kaggle.json configuration file,  
# which contains your Kaggle credentials.  
# The /content directory is commonly used in Google Colab for storing files.  
os.environ['KAGGLE_CONFIG_DIR'] = '/content'  
  
[3]: !kaggle competitions download -c plant-pathology-2020-fgvc7  
  
Warning: Your Kaggle API key is readable by other users on this system! To fix  
this, you can run 'chmod 600 /content/kaggle.json'  
Downloading plant-pathology-2020-fgvc7.zip to /content  
99% 775M/779M [00:09<00:00, 129MB/s]  
100% 779M/779M [00:09<00:00, 86.0MB/s]  
  
[4]: !unzip plant-pathology-2020-fgvc7.zip -d plant_pathology  
  
Archive: plant-pathology-2020-fgvc7.zip  
inflating: plant_pathology/images/Test_0.jpg  
inflating: plant_pathology/images/Test_1.jpg  
inflating: plant_pathology/images/Test_10.jpg  
inflating: plant_pathology/images/Test_100.jpg  
inflating: plant_pathology/images/Test_1000.jpg  
inflating: plant_pathology/images/Test_1001.jpg  
inflating: plant_pathology/images/Test_1002.jpg  
inflating: plant_pathology/images/Test_1003.jpg  
inflating: plant_pathology/images/Test_1004.jpg
```



```
inflating: plant_pathology/images/Train_97.jpg
inflating: plant_pathology/images/Train_970.jpg
inflating: plant_pathology/images/Train_971.jpg
inflating: plant_pathology/images/Train_972.jpg
inflating: plant_pathology/images/Train_973.jpg
inflating: plant_pathology/images/Train_974.jpg
inflating: plant_pathology/images/Train_975.jpg
inflating: plant_pathology/images/Train_976.jpg
inflating: plant_pathology/images/Train_977.jpg
inflating: plant_pathology/images/Train_978.jpg
inflating: plant_pathology/images/Train_979.jpg
inflating: plant_pathology/images/Train_98.jpg
inflating: plant_pathology/images/Train_980.jpg
inflating: plant_pathology/images/Train_981.jpg
inflating: plant_pathology/images/Train_982.jpg
inflating: plant_pathology/images/Train_983.jpg
inflating: plant_pathology/images/Train_984.jpg
inflating: plant_pathology/images/Train_985.jpg
inflating: plant_pathology/images/Train_986.jpg
inflating: plant_pathology/images/Train_987.jpg
inflating: plant_pathology/images/Train_988.jpg
inflating: plant_pathology/images/Train_989.jpg
inflating: plant_pathology/images/Train_99.jpg
inflating: plant_pathology/images/Train_990.jpg
inflating: plant_pathology/images/Train_991.jpg
inflating: plant_pathology/images/Train_992.jpg
inflating: plant_pathology/images/Train_993.jpg
inflating: plant_pathology/images/Train_994.jpg
inflating: plant_pathology/images/Train_995.jpg
inflating: plant_pathology/images/Train_996.jpg
inflating: plant_pathology/images/Train_997.jpg
inflating: plant_pathology/images/Train_998.jpg
inflating: plant_pathology/images/Train_999.jpg
inflating: plant_pathology/sample_submission.csv
inflating: plant_pathology/test.csv
inflating: plant_pathology/train.csv
```

```
[5]: # Install the EfficientNet package quietly.
# EfficientNet is a family of neural networks that are highly efficient and
# accurate for image classification tasks.
!pip install -q efficientnet
```

```
50.7/50.7 kB
2.4 MB/s eta 0:00:00
```

```
[6]: # Importing essential libraries for data manipulation, visualization, and building machine learning models.

import os # For interacting with the operating system.
import gc # For garbage collection to free up memory.
import re # For regular expression operations.
import cv2 # For image processing.
import math # For mathematical operations.
import numpy as np # For numerical operations on large arrays and matrices.
import scipy as sp # For scientific and technical computing.
import pandas as pd # For data manipulation and analysis.

import tensorflow as tf # For building and training machine learning models.
from IPython.display import SVG # For displaying SVG images in Jupyter notebooks.
import efficientnet.tfkeras as efn # For using EfficientNet models.
from keras.utils import plot_model # For visualizing Keras models.
import tensorflow.keras.layers as L # For defining layers in Keras models.
from keras.utils import model_to_dot # For converting models to DOT format.
import tensorflow.keras.backend as K # For Keras backend operations.
from tensorflow.keras.models import Model # For creating Keras models.
from tensorflow.keras.applications import DenseNet121 # For using DenseNet121 model.

import seaborn as sns # For data visualization.
from tqdm import tqdm # For progress bars.
import matplotlib.cm as cm # For colormap support.
from sklearn import metrics # For evaluating machine learning models.
import matplotlib.pyplot as plt # For plotting data.
from sklearn.utils import shuffle # For shuffling datasets.
from sklearn.model_selection import train_test_split # For splitting datasets into train and test sets.

tqdm.pandas() # For progress bars in pandas operations.
import plotly.express as px # For interactive visualizations.
import plotly.graph_objects as go # For creating visualizations.
import plotly.figure_factory as ff # For creating special visualizations.
from plotly.subplots import make_subplots # For creating subplot visualizations.

np.random.seed(0) # For reproducibility.
tf.random.set_seed(0) # For reproducibility.

import warnings
warnings.filterwarnings("ignore") # To ignore warnings.
```

```
[7]: EPOCHS = 20
SAMPLE_LEN = 100
IMAGE_PATH = "/content/plant_pathology/images"
TEST_PATH = "/content/plant_pathology/test.csv"
TRAIN_PATH = "/content/plant_pathology/train.csv"
SUB_PATH = "/content/plant_pathology/sample_submission.csv"

sub = pd.read_csv(SUB_PATH)
test_data = pd.read_csv(TEST_PATH)
train_data = pd.read_csv(TRAIN_PATH)
```

```
[8]: train_data.head()
```

```
[8]:   image_id  healthy  multiple_diseases  rust  scab
0  Train_0      0            0       0     1
1  Train_1      0            1       0     0
2  Train_2      1            0       0     0
3  Train_3      0            0       1     0
4  Train_4      1            0       0     0
```

```
[9]: from tqdm import tqdm # Import tqdm for progress bars
tqdm.pandas() # Add progress bars to pandas operations

IMAGE_PATH = "/content/plant_pathology/images/" # Correct image path
SAMPLE_LEN = 100 # Example value, adjust as needed

def load_image(image_id):
    file_path = image_id + ".jpg" # Construct file path for the image
    image = cv2.imread(IMAGE_PATH + file_path) # Read the image from the
    ↪specified path

    if image is None:
        print(f"Error loading image: {file_path}") # Print error message if
    ↪image is not found
        return None # Handle error appropriately by returning None

    return cv2.cvtColor(image, cv2.COLOR_BGR2RGB) # Convert image to RGB color
    ↪space

# Load images with error handling
train_images = train_data["image_id"][:SAMPLE_LEN].progress_apply(load_image) ↪
    ↪# Apply load_image function to each image_id

# Remove any None values from the list of images
train_images = train_images.dropna().tolist() # Drop None values and convert
    ↪to a list
```

```

# Explanation:
# - tqdm.pandas(): Adds progress bars to pandas operations.
# - IMAGE_PATH: Specifies the path to the directory containing the images.
# - SAMPLE_LEN: Number of images to load, can be adjusted.
# - load_image: Function to load and convert images to RGB, with error handling
#   ↪for missing images.
# - train_images: Loads images from the dataset, applies load_image function,
#   ↪and removes any None values.

```

100% | 100/100 [00:05<00:00, 19.19it/s]

[10]: fig = px.imshow(cv2.resize(train_images[0], (205, 136))) # Create an
 ↪interactive plot to display a resized image
fig.show() # Show the plot in the notebook

[11]: # Calculate mean pixel values for each color channel and overall for all images
 ↪in train_images
red_values = [np.mean(train_images[idx][:, :, 0]) for idx in
 ↪range(len(train_images))] # Mean red channel values
green_values = [np.mean(train_images[idx][:, :, 1]) for idx in
 ↪range(len(train_images))] # Mean green channel values
blue_values = [np.mean(train_images[idx][:, :, 2]) for idx in
 ↪range(len(train_images))] # Mean blue channel values
values = [np.mean(train_images[idx]) for idx in range(len(train_images))] # Mean
 ↪values across all channels (RGB)

[12]: fig = ff.create_distplot([values], group_labels=["Channels"], colors=["purple"]) # Create a distribution plot for pixel values across all
 ↪channels
fig.update_layout(showlegend=False, template="simple_white",
 ↪title_text="Distribution of channel values") # Customize plot layout and
 ↪add title
fig.data[0].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color to
 ↪black for better visibility
fig.data[0].marker.line.width = 0.5 # Set marker line width for better
 ↪appearance
fig

[13]: fig = ff.create_distplot([red_values], group_labels=["R"], colors=["red"]) # Create a distribution plot for red channel pixel values
fig.update_layout(showlegend=False, template="simple_white",
 ↪title_text="Distribution of red channel values") # Customize plot layout
 ↪and add title
fig.data[0].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color to
 ↪black for better visibility

```

fig.data[0].marker.line.width = 0.5 # Set marker line width for better
    ↵appearance
fig

[14]: fig = ff.create_distplot([green_values], group_labels=["G"], colors=["green"])
    ↵# Create a distribution plot for green channel pixel values
fig.update_layout(showlegend=False, template="simple_white",
    ↵title_text="Distribution of green channel values") # Customize plot layout
    ↵and add title
fig.data[0].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color to
    ↵black for better visibility
fig.data[0].marker.line.width = 0.5 # Set marker line width for better
    ↵appearance
fig

[15]: fig = ff.create_distplot([blue_values], group_labels=["B"], colors=["blue"]) #
    ↵Create a distribution plot for blue channel pixel values
fig.update_layout(showlegend=False, template="simple_white",
    ↵title_text="Distribution of blue channel values") # Customize plot layout
    ↵and add title
fig.data[0].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color to
    ↵black for better visibility
fig.data[0].marker.line.width = 0.5 # Set marker line width for better
    ↵appearance
fig

[16]: import plotly.graph_objects as go

fig = go.Figure()

# Assuming you have lists red_values, green_values, blue_values defined
    ↵somewhere

for idx, values in enumerate([red_values, green_values, blue_values]):
    if idx == 0:
        color = "Red"
    elif idx == 1:
        color = "Green"
    elif idx == 2:
        color = "Blue"

    # Add a box trace for each color channel
    fig.add_trace(go.Box(x=[color]*len(values), y=values, name=color,
        ↵marker=dict(color=color.lower())))

fig.update_layout(

```

```

        yaxis_title="Mean value",
        xaxis_title="Color channel",
        title="Mean value vs. Color channel",
        template="plotly_white"
    )

```

```
[17]: fig = ff.create_distplot([red_values, green_values, blue_values],
                            group_labels=["R", "G", "B"],
                            colors=["red", "green", "blue"]) # Create a
                                ↪distribution plot for red, green, and blue channel pixel values
fig.update_layout(title_text="Distribution of red channel values",
                    ↪template="simple_white") # Customize plot layout and add title
fig.data[0].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color for red
                                ↪channel to black for better visibility
fig.data[0].marker.line.width = 0.5 # Set marker line width for red channel
fig.data[1].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color for
                                ↪green channel to black for better visibility
fig.data[1].marker.line.width = 0.5 # Set marker line width for green channel
fig.data[2].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color for
                                ↪blue channel to black for better visibility
fig.data[2].marker.line.width = 0.5 # Set marker line width for blue channel
fig
```

```
[18]: def visualize_leaves(cond=[0, 0, 0, 0], cond_cols=["healthy"], is_cond=True):
    if not is_cond:
        cols, rows = 3, min([3, len(train_images) // 3])
        fig, ax = plt.subplots(nrows=rows, ncols=cols, figsize=(30, rows * 20 / 3))
        for col in range(cols):
            for row in range(rows):
                ax[row, col].imshow(train_images[-row * 3 - col - 1]) # Access
                                ↪images directly from the list
        plt.show()
        return None

    cond_0 = "healthy == {}".format(cond[0])
    cond_1 = "scab == {}".format(cond[1])
    cond_2 = "rust == {}".format(cond[2])
    cond_3 = "multiple_diseases == {}".format(cond[3])

    cond_list = []
    for col in cond_cols:
        if col == "healthy":
            cond_list.append(cond_0)
        if col == "scab":
            cond_list.append(cond_1)
```

```

if col == "rust":
    cond_list.append(cond_2)
if col == "multiple_diseases":
    cond_list.append(cond_3)

# Subset of data, adjust as needed
data = train_data.loc[:100]
for cond in cond_list:
    data = data.query(cond)

# Get indices from the filtered data
indices = data.index.tolist()

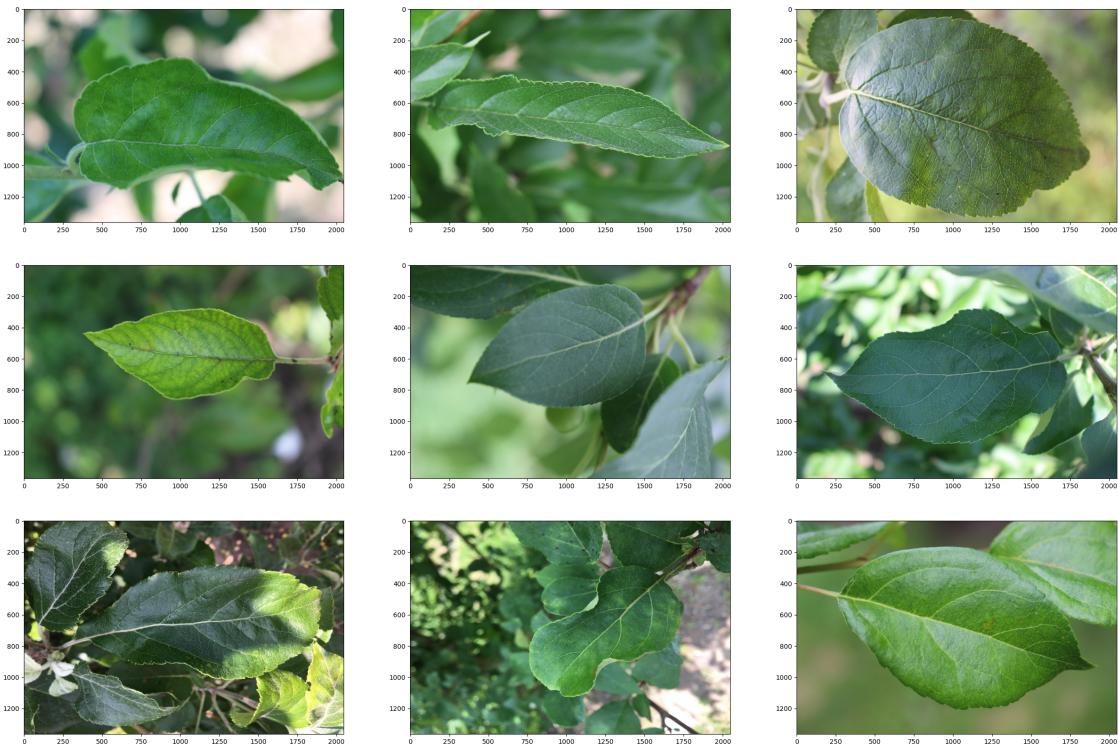
# Access images from the list using these indices if within range
images = [train_images[i] for i in indices if i < len(train_images)]

cols, rows = 3, min([3, len(images) // 3])

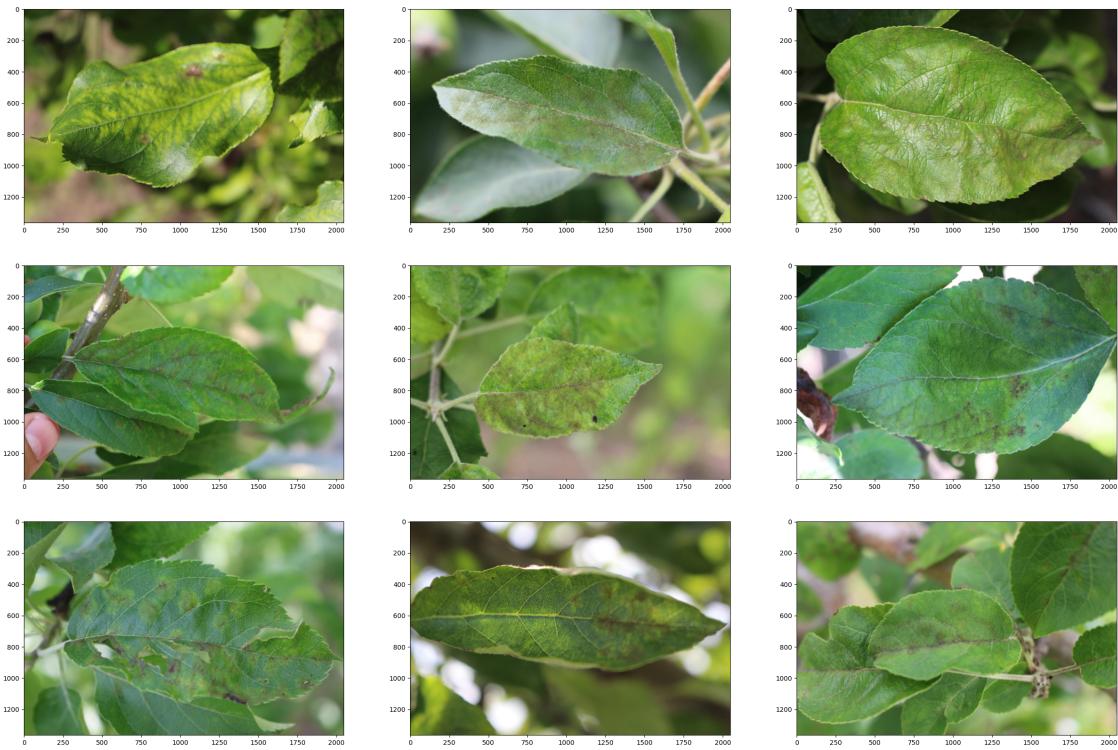
fig, ax = plt.subplots(nrows=rows, ncols=cols, figsize=(30, rows * 20 / 3))
for col in range(cols):
    for row in range(rows):
        idx = row * 3 + col
        if idx < len(images): # Check if index is within the range of
            images list
            ax[row, col].imshow(images[idx]) # Access images by simple
            indexing
plt.show()

# Example usage:
visualize_leaves(cond=[1, 0, 0, 0], cond_cols=["healthy"])

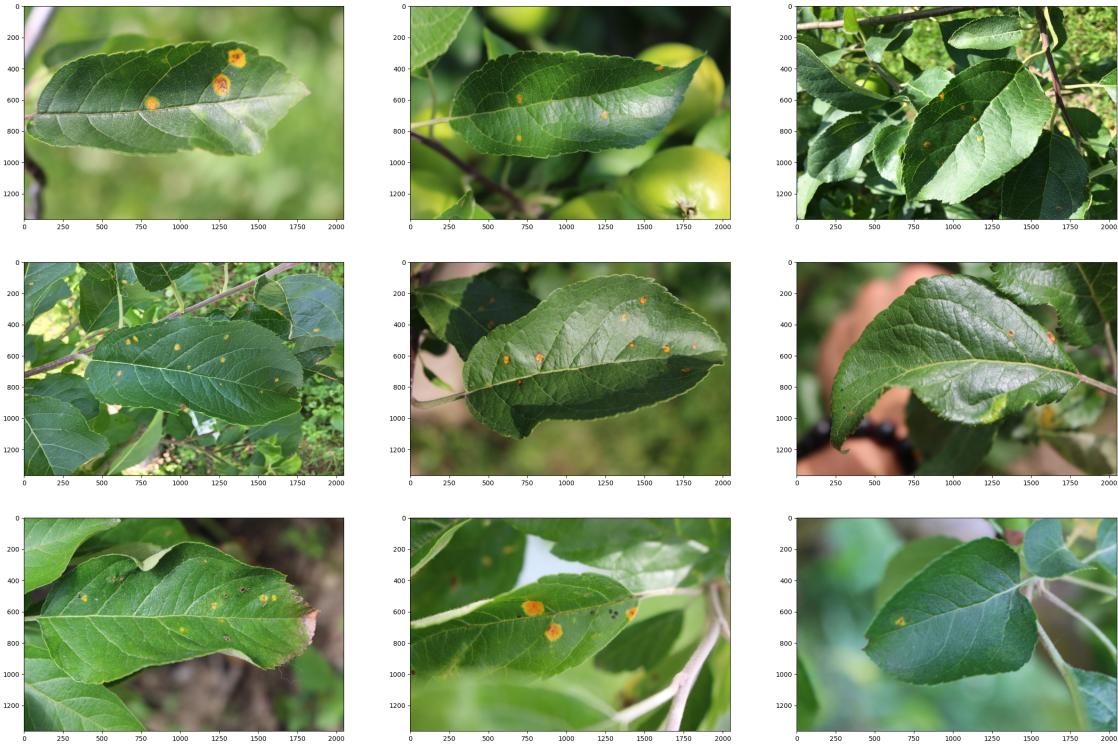
```



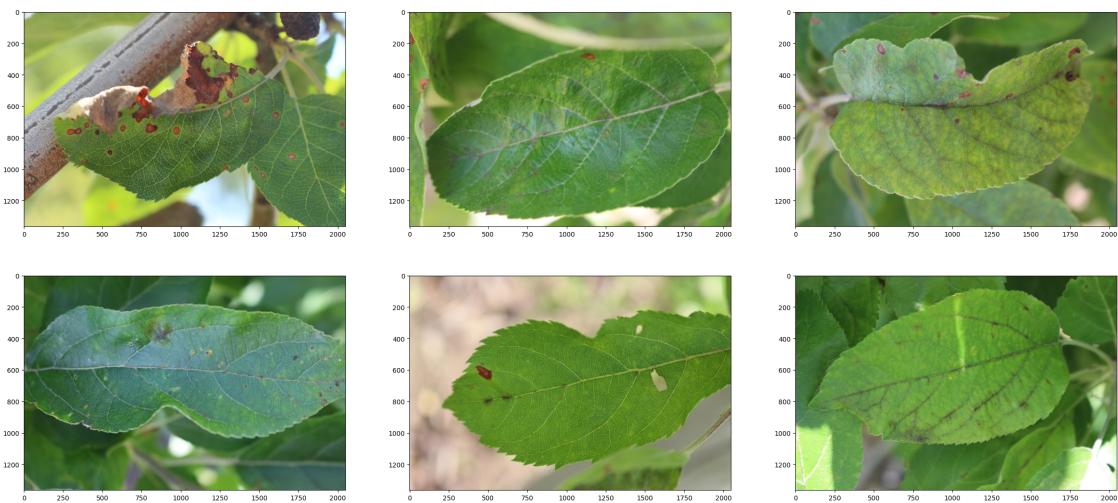
```
[19]: visualize_leaves(cond=[0, 1, 0, 0], cond_cols=["scab"])
```



```
[20]: visualize_leaves(cond=[0, 0, 1, 0], cond_cols=["rust"])
```



```
[21]: visualize_leaves(cond=[0, 0, 0, 1], cond_cols=["multiple_diseases"])
```

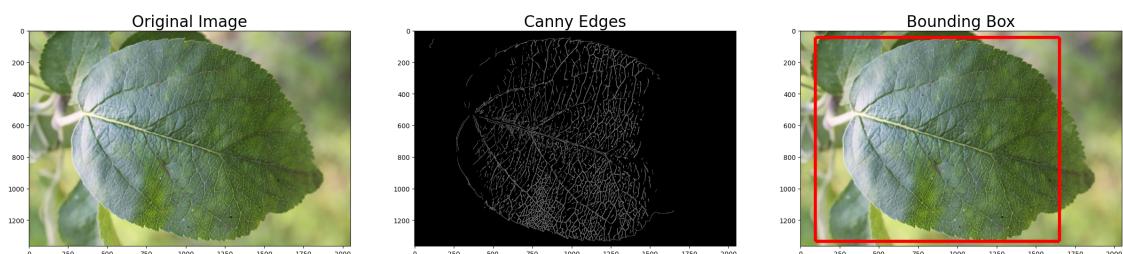
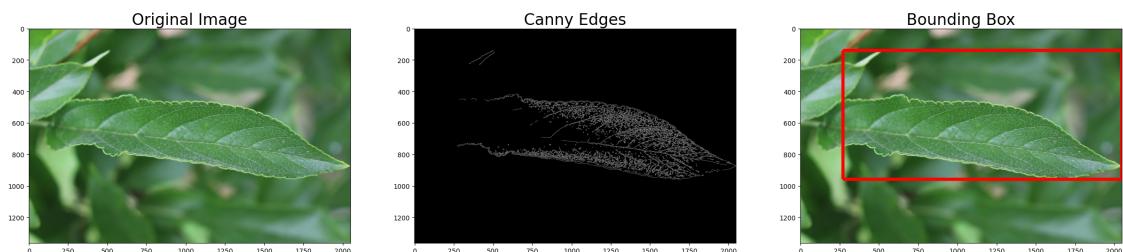
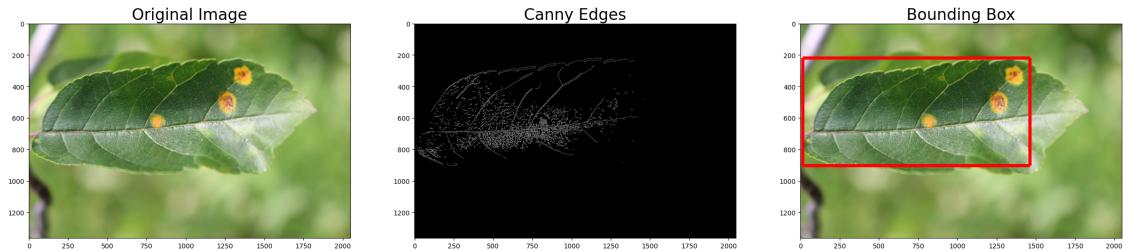


```
[22]: fig = go.Figure([go.Pie(labels=train_data.columns[1:],  
                           values=train_data.iloc[:, 1:].sum().values)]) # Create a pie chart  
    ↵showing the distribution of target labels  
fig.update_layout(title_text="Pie chart of targets", template="simple_white") #  
    ↵# Customize plot layout and add title  
fig.data[0].marker.line.color = 'rgb(0, 0, 0)' # Set marker line color to  
    ↵black for better visibility  
fig.data[0].marker.line.width = 0.5 # Set marker line width for better  
    ↵visibility  
fig.show() # Display the pie chart
```

```
[23]: def edge_and_cut(img):  
    emb_img = img.copy()  
    edges = cv2.Canny(img, 100, 200) # Detect edges using Canny edge detection  
    edge_coors = []  
    for i in range(edges.shape[0]):  
        for j in range(edges.shape[1]):  
            if edges[i][j] != 0:  
                edge_coors.append((i, j)) # Collect coordinates of edge points  
  
    # Determine bounding box coordinates based on edge points  
    row_min = edge_coors[np.argsort([coor[0] for coor in edge_coors])[0]][0]  
    row_max = edge_coors[np.argsort([coor[0] for coor in edge_coors])[-1]][0]  
    col_min = edge_coors[np.argsort([coor[1] for coor in edge_coors])[0]][1]  
    col_max = edge_coors[np.argsort([coor[1] for coor in edge_coors])[-1]][1]  
    new_img = img[row_min:row_max, col_min:col_max] # Crop the image based on  
    ↵bounding box  
  
    # Highlight bounding box in the original image  
    emb_img[row_min-10:row_min+10, col_min:col_max] = [255, 0, 0]  
    emb_img[row_max-10:row_max+10, col_min:col_max] = [255, 0, 0]  
    emb_img[row_min:row_max, col_min-10:col_min+10] = [255, 0, 0]  
    emb_img[row_min:row_max, col_max-10:col_max+10] = [255, 0, 0]  
  
    # Plot the original image, edges, and image with bounding box  
    fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(30, 20))  
    ax[0].imshow(img, cmap='gray')  
    ax[0].set_title('Original Image', fontsize=24)  
    ax[1].imshow(edges, cmap='gray')  
    ax[1].set_title('Canny Edges', fontsize=24)  
    ax[2].imshow(emb_img, cmap='gray')  
    ax[2].set_title('Bounding Box', fontsize=24)  
    plt.show()
```

```
[24]: # Apply the edge_and_cut function to specific images from the train_images list  
edge_and_cut(train_images[3]) # Apply to the 4th image in the list  
edge_and_cut(train_images[4]) # Apply to the 5th image in the list
```

```
edge_and_cut(train_images[5]) # Apply to the 6th image in the list
```



```
[25]: def invert(img):
    fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(30, 20))

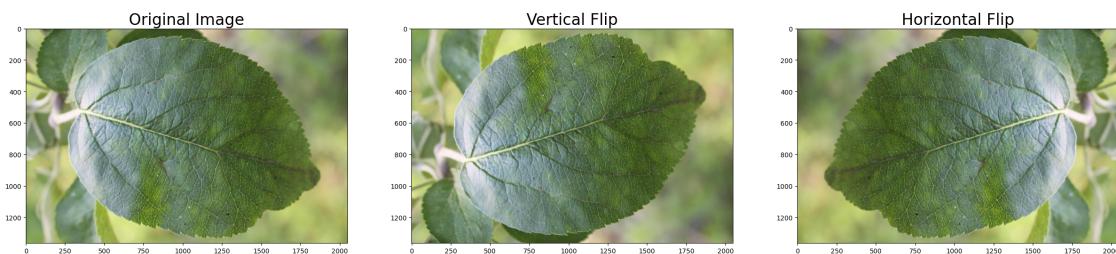
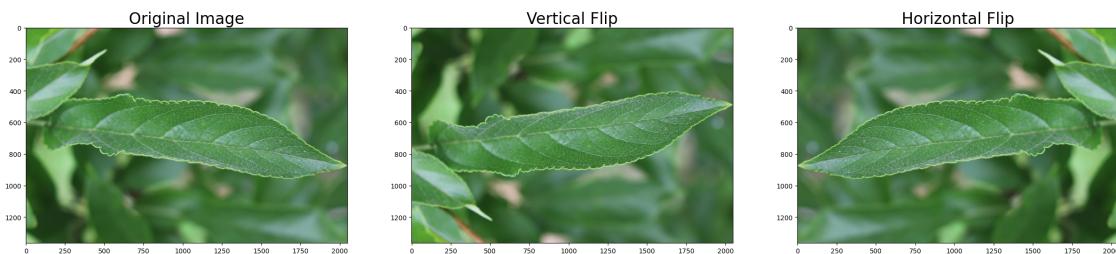
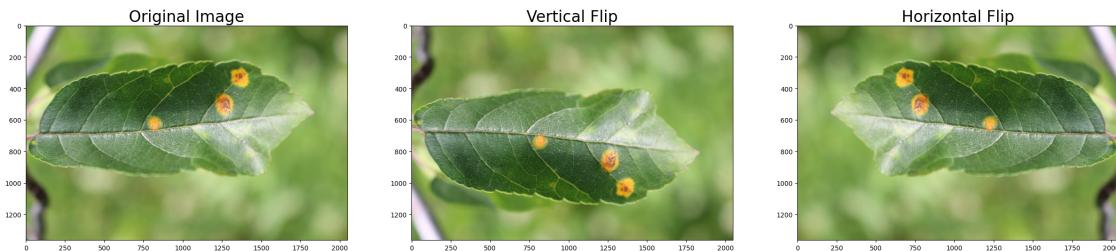
    # Display the original image
    ax[0].imshow(img)
    ax[0].set_title('Original Image', fontsize=24)

    # Display the vertically flipped image
    ax[1].imshow(cv2.flip(img, 0))
    ax[1].set_title('Vertical Flip', fontsize=24)

    # Display the horizontally flipped image
    ax[2].imshow(cv2.flip(img, 1))
    ax[2].set_title('Horizontal Flip', fontsize=24)
```

```
plt.show()
```

```
[26]: invert(train_images[3])  
invert(train_images[4])  
invert(train_images[5])
```

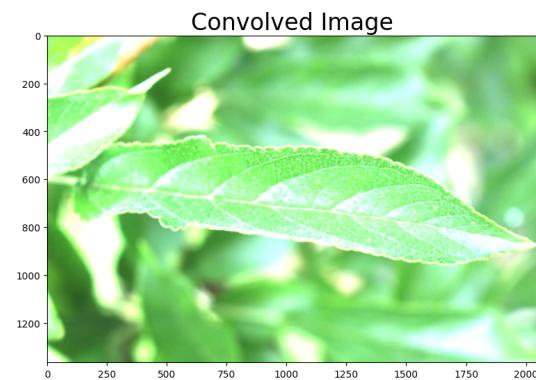
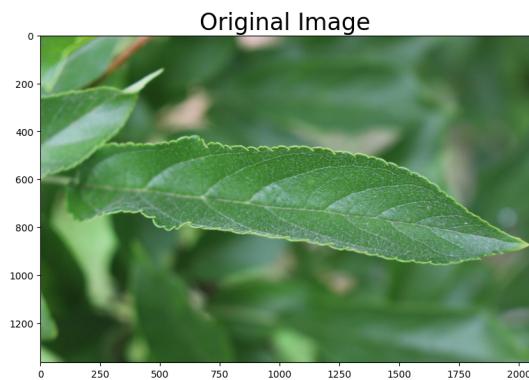
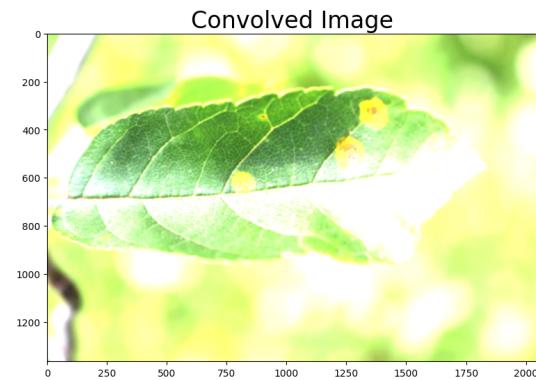
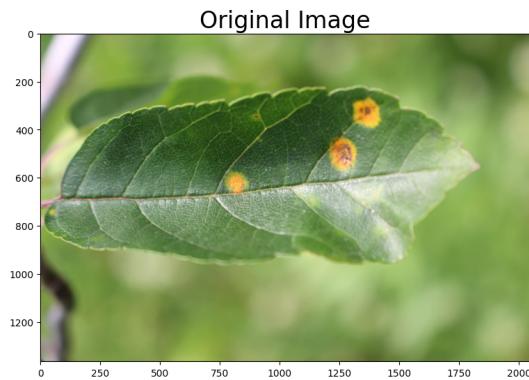


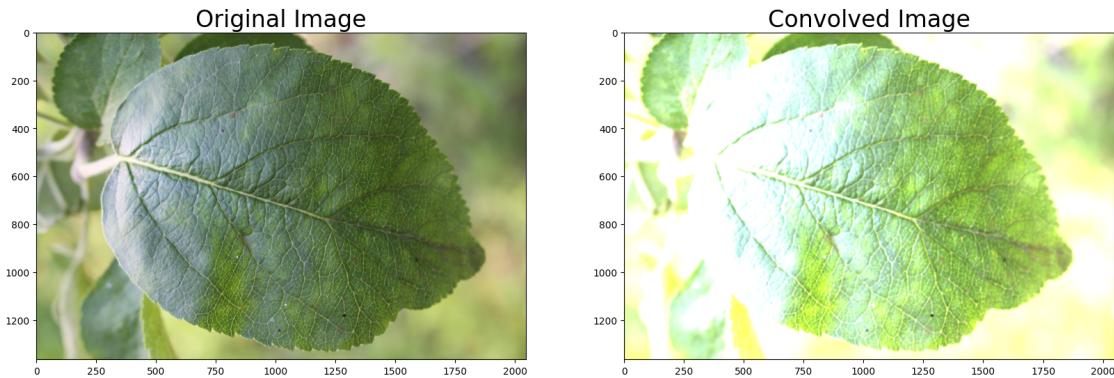
```
[27]: def conv(img):  
    fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(20, 20))  
  
    # Define a kernel for the convolution operation  
    kernel = np.ones((7, 7), np.float32) / 25  
    # Apply the convolution operation  
    conv = cv2.filter2D(img, -1, kernel)  
  
    # Display the original image  
    ax[0].imshow(img)  
    ax[0].set_title('Original Image', fontsize=24)
```

```
# Display the convolved image
ax[1].imshow(conv)
ax[1].set_title('Convolved Image', fontsize=24)

plt.show()
```

```
[28]: conv(train_images[3])
conv(train_images[4])
conv(train_images[5])
```





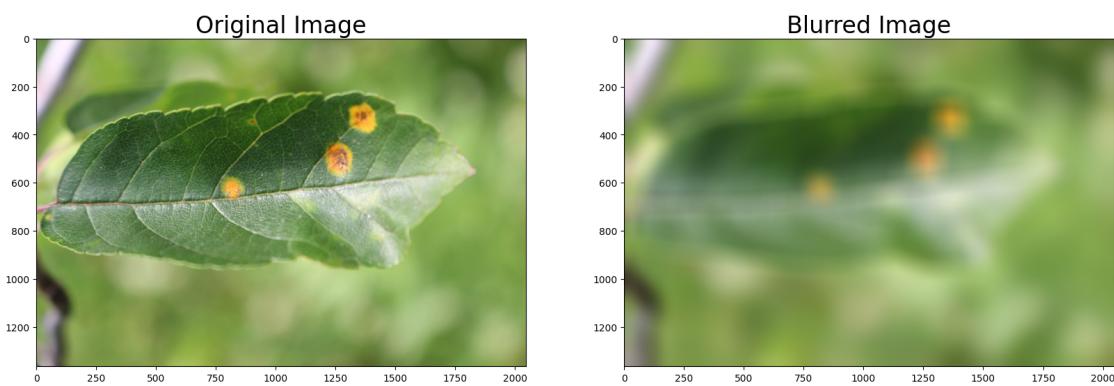
```
[29]: def blur(img):
    fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(20, 20))

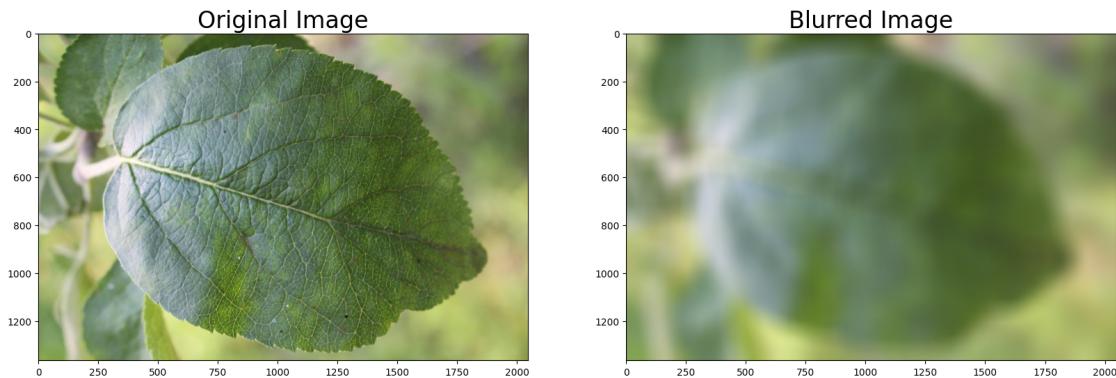
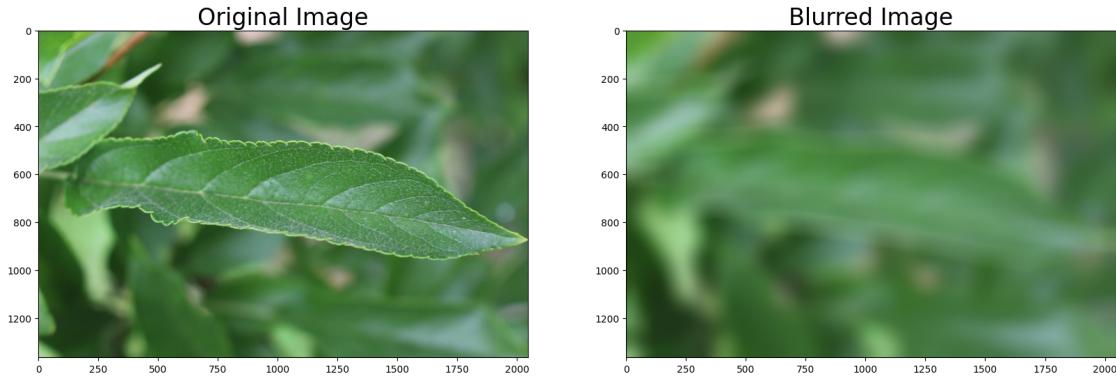
    # Display the original image
    ax[0].imshow(img)
    ax[0].set_title('Original Image', fontsize=24)

    # Apply a blurring operation with a 100x100 kernel size
    ax[1].imshow(cv2.blur(img, (100, 100)))
    ax[1].set_title('Blurred Image', fontsize=24)

    plt.show()
```

```
[30]: blur(train_images[3])
blur(train_images[4])
blur(train_images[5])
```





```
[31]: import tensorflow as tf
from sklearn.model_selection import train_test_split

# Use AUTOTUNE to automatically tune the number of threads
AUTO = tf.data.experimental.AUTOTUNE

# Since you are running on a CPU, no need to set up TPU
# Determine the batch size (you can adjust this based on your system's memory)
BATCH_SIZE = 16

# Manually set the path to your dataset
GCS_DS_PATH = '/content/plant_pathology' # Replace with the actual path to your dataset

# Function to format image paths based on GCS_DS_PATH
def format_path(st):
    return GCS_DS_PATH + '/images/' + st + '.jpg'

# Assuming you have already loaded train_data and test_data DataFrames
test_paths = test_data.image_id.apply(format_path).values
```

```

train_paths = train_data.image_id.apply(format_path).values

train_labels = np.float32(train_data.loc[:, 'healthy':'scab'].values)
train_paths, valid_paths, train_labels, valid_labels = \
    train_test_split(train_paths, train_labels, test_size=0.15, random_state=2020)

# Verify that the paths are correctly generated
print(train_paths[:5])
print(test_paths[:5])

```

```

['/content/plant_pathology/images/Train_602.jpg'
 '/content/plant_pathology/images/Train_1514.jpg'
 '/content/plant_pathology/images/Train_1571.jpg'
 '/content/plant_pathology/images/Train_607.jpg'
 '/content/plant_pathology/images/Train_58.jpg']
['/content/plant_pathology/images/Test_0.jpg'
 '/content/plant_pathology/images/Test_1.jpg'
 '/content/plant_pathology/images/Test_2.jpg'
 '/content/plant_pathology/images/Test_3.jpg'
 '/content/plant_pathology/images/Test_4.jpg']

```

```

[32]: def decode_image(filename, label=None, image_size=(512, 512)):
    """
    Decodes an image from a file, normalizes it, and resizes it.

    Args:
        filename (str): Path to the image file.
        label (optional): Label associated with the image. Default is None.
        image_size (tuple): Target size to resize the image. Default is (512, 512).

    Returns:
        If label is provided: (image, label) tuple.
        Otherwise: image.
    """
    # Read the image file
    bits = tf.io.read_file(filename)

    # Decode the image file as a JPEG image, expecting 3 color channels (RGB)
    image = tf.image.decode_jpeg(bits, channels=3)

    # Normalize the image to the range [0, 1]
    image = tf.cast(image, tf.float32) / 255.0

    # Resize the image to the target size
    image = tf.image.resize(image, image_size)

```

```

# Return the image and label (if provided), otherwise just the image
if label is None:
    return image
else:
    return image, label

def data_augment(image, label=None):
    """
    Applies random augmentations to the image for data augmentation.

    Args:
        image (Tensor): Input image tensor.
        label (optional): Label associated with the image. Default is None.

    Returns:
        If label is provided: (augmented image, label) tuple.
        Otherwise: augmented image.
    """
    # Apply random horizontal flip
    image = tf.image.random_flip_left_right(image)

    # Apply random vertical flip
    image = tf.image.random_flip_up_down(image)

    # Return the augmented image and label (if provided), otherwise just the image
    if label is None:
        return image
    else:
        return image, label

```

[33]:

```

# Create the training dataset
train_dataset = (
    tf.data.Dataset
    .from_tensor_slices((train_paths, train_labels)) # Create a dataset from the training paths and labels
    .map(decode_image, num_parallel_calls=AUTO) # Decode and preprocess the images in parallel
    .map(data_augment, num_parallel_calls=AUTO) # Apply data augmentation in parallel
    .repeat() # Repeat the dataset indefinitely (useful for training)
    .shuffle(512) # Shuffle the dataset with a buffer size of 512
    .batch(BATCH_SIZE) # Batch the data

```

```

    .prefetch(AUTO)                                # Prefetch the data for
    ↵better performance
)

# Create the validation dataset
valid_dataset = (
    tf.data.Dataset
    .from_tensor_slices((valid_paths, valid_labels)) # Create a dataset from
    ↵the validation paths and labels
    .map(decode_image, num_parallel_calls=AUTO)       # Decode and preprocess
    ↵the images in parallel
    .batch(BATCH_SIZE)                               # Batch the data
    .cache()                                       # Cache the data for
    ↵better performance
    .prefetch(AUTO)                                # Prefetch the data for
    ↵better performance
)

# Create the test dataset
test_dataset = (
    tf.data.Dataset
    .from_tensor_slices(test_paths)                 # Create a dataset from
    ↵the test paths
    .map(decode_image, num_parallel_calls=AUTO)       # Decode and preprocess
    ↵the images in parallel
    .batch(BATCH_SIZE)                               # Batch the data
)

```

[34]: # Define a function to build the learning rate schedule function

```

def build_lrfn(lr_start=0.00001, lr_max=0.00005,
               lr_min=0.00001, lr_rampup_epochs=5,
               lr_sustain_epochs=0, lr_exp_decay=.8):
    # Adjust the maximum learning rate based on the number of replicas in sync
    lr_max = lr_max * strategy.num_replicas_in_sync

    # Define the learning rate schedule function
    def lrfn(epoch):
        # Ramp up phase: gradually increase the learning rate
        if epoch < lr_rampup_epochs:
            lr = (lr_max - lr_start) / lr_rampup_epochs * epoch + lr_start
        # Sustain phase: maintain the learning rate at the maximum value
        elif epoch < lr_rampup_epochs + lr_sustain_epochs:
            lr = lr_max
        # Exponential decay phase: gradually decrease the learning rate
        else:
            lr = (lr_max - lr_min) *\
```

```

        lr_exp_decay** (epoch - lr_rampup_epochs \
                         - lr_sustain_epochs) + lr_min
    return lr

    return lrfn # Return the learning rate schedule function

```

```
[35]: # Define the learning rate function without TPU strategy
def build_lrfn(lr_start=0.00001, lr_max=0.00005,
               lr_min=0.00001, lr_rampup_epochs=5,
               lr_sustain_epochs=0, lr_exp_decay=.8):
    def lrfn(epoch):
        if epoch < lr_rampup_epochs:
            lr = (lr_max - lr_start) / lr_rampup_epochs * epoch + lr_start
        elif epoch < lr_rampup_epochs + lr_sustain_epochs:
            lr = lr_max
        else:
            lr = (lr_max - lr_min) *\
                  lr_exp_decay** (epoch - lr_rampup_epochs \
                                  - lr_sustain_epochs) + lr_min
    return lrfn
return lrfn

# Use the learning rate function
lrfn = build_lrfn()

# Calculate the number of steps per epoch based on the batch size and the size
# of the training dataset
STEPS_PER_EPOCH = train_labels.shape[0] // BATCH_SIZE

# Create a learning rate scheduler callback using the learning rate function
# (lrfn)
lr_schedule = tf.keras.callbacks.LearningRateScheduler(lrfn, verbose=1)

# Now print out to verify
print(f"STEPS_PER_EPOCH: {STEPS_PER_EPOCH}")
print(f"Learning rate schedule for the first few epochs:")
for epoch in range(10):
    print(f"Epoch {epoch + 1}: {lrfn(epoch)}")

```

```

STEPS_PER_EPOCH: 96
Learning rate schedule for the first few epochs:
Epoch 1: 1e-05
Epoch 2: 1.800000000000004e-05
Epoch 3: 2.600000000000002e-05
Epoch 4: 3.400000000000007e-05
Epoch 5: 4.200000000000004e-05
Epoch 6: 5e-05

```

```
Epoch 7: 4.200000000000004e-05
Epoch 8: 3.560000000000005e-05
Epoch 9: 3.048000000000006e-05
Epoch 10: 2.638400000000004e-05
```

```
[36]: # Define and compile the model
model = tf.keras.Sequential([
    tf.keras.applications.DenseNet121(input_shape=(512, 512, 3),
                                         weights='imagenet',
                                         include_top=False),
    tf.keras.layers.GlobalAveragePooling2D(),
    tf.keras.layers.Dense(train_labels.shape[1], activation='softmax')
])

model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['categorical_accuracy'])

model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/densenet/densenet121_weights_tf_dim_ordering_tf_kernels_notop.h5
29084464/29084464 [=====] - 0s 0us/step
Model: "sequential"

-----  

Layer (type)          Output Shape         Param #
-----  

densenet121 (Functional)  (None, 16, 16, 1024)    7037504  

global_average_pooling2d ( (None, 1024)           0
GlobalAveragePooling2D)  

dense (Dense)          (None, 4)            4100  

-----  

Total params: 7041604 (26.86 MB)
Trainable params: 6957956 (26.54 MB)
Non-trainable params: 83648 (326.75 KB)
```

```
[ ]: # Train the model using the train_dataset for the specified number of epochs
history = model.fit(
    train_dataset,
    epochs=EPOCHS,           # Number of epochs to train the model
    callbacks=[lr_schedule], # List of callbacks to apply during training, ↴
    in this case, a learning rate scheduler
```

```

    steps_per_epoch=STEPS_PER_EPOCH, # Number of steps (batches) per epoch, calculated based on the training data size and batch size
    validation_data=valid_dataset # Dataset for validation at the end of each epoch
)

```

```
[ ]: display_training_curves(
    history.history['categorical_accuracy'],
    history.history['val_categorical_accuracy'],
    'accuracy')
```

```
[ ]: import plotly.graph_objects as go
import plotly.express as px
import cv2

# Function to process an image for model input
def process(img):
    return cv2.resize(img / 255.0, (512, 512)).reshape(-1, 512, 512, 3)

# Function to make predictions using the model
def predict(img):
    return model.predict(process(img))[0]

# Initialize Plotly figure with subplots
fig = make_subplots(rows=4, cols=2)

# Function to visualize predictions for each image
def visualize_image_predictions(image, index):
    preds = predict(image)
    class_names = ["Healthy", "Multiple diseases", "Rust", "Scab"]
    pred_class = class_names[np.argmax(preds)]

    # Color coding for predicted class
    colors = {name: px.colors.qualitative.Plotly[0] for name in class_names}
    colors[pred_class] = px.colors.qualitative.Plotly[1]
    colors[pred_class] = "seagreen"
    bar_colors = [colors[name] for name in class_names]

    # Add image to subplot
    fig.add_trace(go.Image(z=cv2.resize(image, (205, 136))), row=index+1, col=1)

    # Add bar chart with predictions to subplot
    fig.add_trace(go.Bar(x=class_names, y=preds, marker=dict(color=bar_colors)), row=index+1, col=2)

    # Visualize each image with predictions
    for i, image in enumerate(train_images[:4]):
```

```

    visualize_image_predictions(image, i)

# Update layout and display the plot
fig.update_layout(height=1200, width=800, title_text="DenseNet Predictions",
                  showlegend=False, template="plotly_white")
fig.show()

[ ]: probs_dnn = model.predict(test_dataset, verbose=1)
sub.loc[:, 'healthy'] = probs_dnn
sub.to_csv('submission_dnn.csv', index=False)
sub.head()

[ ]: with strategy.scope():
      model = tf.keras.Sequential([
          efn.EfficientNetB7(input_shape=(512, 512, 3), # Define input shape for
EfficientNetB7 model
          weights='imagenet', # Initialize with
pretrained ImageNet weights
          include_top=False), # Exclude the top
classification layer

          L.GlobalAveragePooling2D(), # Reduce spatial
dimensions using Global Average Pooling
          L.Dense(train_labels.shape[1], # Output layer with
softmax activation for multiclass
          activation='softmax')
      ])

      model.compile(optimizer='adam', # Use Adam optimizer for
training
          loss='categorical_crossentropy', # Categorical
crossentropy loss for multiclass classification
          metrics=['categorical_accuracy']) # Track categorical
accuracy during training

      model.summary() # Print a summary of the model architecture

[ ]: SVG(tf.keras.utils.model_to_dot(Model(model.layers[0].input, model.layers[0].
                                         layers[11].output), dpi=70).create(prog='dot', format='svg'))

[ ]: SVG(tf.keras.utils.model_to_dot(model, dpi=70).create(prog='dot', format='svg'))

[ ]: history = model.fit(train_dataset,
                        epochs=EPOCHS,
                        callbacks=[lr_schedule],
                        steps_per_epoch=STEPS_PER_EPOCH,

```

```

        validation_data=valid_dataset)

[ ]: display_training_curves(
    history.history['categorical_accuracy'],
    history.history['val_categorical_accuracy'],
    'accuracy')

[ ]: fig = make_subplots(rows=4, cols=2)

[ ]: # Example for train_images[2]
preds = predict(train_images[2])
colors = {
    "Healthy": px.colors.qualitative.Plotly[0],
    "Scab": px.colors.qualitative.Plotly[0],
    "Rust": px.colors.qualitative.Plotly[0],
    "Multiple diseases": px.colors.qualitative.Plotly[0]
}

# Determine the predicted class
if list.index(preds.tolist(), max(preds)) == 0:
    pred = "Healthy"
elif list.index(preds.tolist(), max(preds)) == 1:
    pred = "Scab"
elif list.index(preds.tolist(), max(preds)) == 2:
    pred = "Rust"
elif list.index(preds.tolist(), max(preds)) == 3:
    pred = "Multiple diseases"

# Set color for the predicted class
colors[pred] = px.colors.qualitative.Plotly[1]
colors["Healthy"] = "seagreen"

# Add image and bar chart to subplot
fig.add_trace(go.Image(z=cv2.resize(train_images[2], (205, 136))), row=1, col=1)
fig.add_trace(go.Bar(x=["Healthy", "Multiple diseases", "Rust", "Scab"], y=preds, marker=dict(color=colors)), row=1, col=2)

# Repeat the above for other train_images[n] (0, 1, 3)

# Update layout and display
fig.update_layout(height=1200, width=800, title_text="EfficientNet\u2192Predictions", showlegend=False)
fig.update_layout(template="plotly_white")

[ ]: probs_efn = model.predict(test_dataset, verbose=1)
sub.loc[:, 'healthy'][:] = probs_efn
sub.to_csv('submission_efn.csv', index=False)

```

```
sub.head()
```

```
[ ]: #Considering these factors, EfficientNet might be slightly better
#if you value efficiency and scalability alongside accuracy.
#EfficientNet is known for achieving comparable or better performance
#with fewer parameters and lower computational cost.
#If the choice is purely based on the provided accuracy and epochs,
#both models are very similar, but EfficientNet's design for efficiency gives it
#it a slight edge.
```