```
import os
import pandas as pd
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import random
from collections import defaultdict
import seaborn as sns
from keras.preprocessing.image import ImageDataGenerator
import tensorflow as tf
import tensorflow_addons as tfa
train_path = "Potato_Dataset/Train_set"
test_path = "Potato_Dataset/Test_set"
from tqdm import tqdm
from PIL import Image
from collections import defaultdict

X_train = []
Y_train = []

for target in os.listdir(train_path):
    target_path = os.path.join(train_path, target)
    for file in tqdm(os.listdir(target_path)):
        file_path = os.path.join(target_path, file)
        X_train.append(file_path)
        Y_train.append(target)

from tqdm import tqdm
from PIL import Image
from collections import defaultdict

X_test = []
Y_test = []

for target in os.listdir(test_path):
    target_path = os.path.join(test_path, target)
    for file in tqdm(os.listdir(target_path)):
        file_path = os.path.join(target_path, file)
        X_test.append(file_path)
        Y_test.append(target)
from sklearn.model_selection import train_test_split
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train,
test_size=0.2, random_state=42)
sns.countplot(x = Y_train)
sns.countplot(x = Y_test)
sns.countplot(x = Y_val)
df_train = pd.DataFrame(list(zip(X_train, Y_train)), columns
=['image_path', 'label'])
df_val = pd.DataFrame(list(zip(X_val, Y_val)), columns =['image_path',
'label'])
df_test = pd.DataFrame(list(zip(X_test, Y_test)), columns =['image_path',
'label'])
from keras.preprocessing.image import ImageDataGenerator

train_aug = ImageDataGenerator(
    horizontal_flip=True,
```

```
    width_shift_range=0.05,
    height_shift_range=0.05,
    zoom_range=0.05,
    rescale = 1./255,
    preprocessing_function=tf.keras.applications.vgg19.preprocess_input
)

test_aug = ImageDataGenerator(
    rescale = 1./255,
    preprocessing_function=tf.keras.applications.vgg19.preprocess_input
)

train_generator= train_aug.flow_from_dataframe(
    dataframe=df_train,
    x_col="image_path",
    y_col="label",
    batch_size=16,
    color_mode="rgb",
    target_size = (224, 224),
    class_mode="categorical")

val_generator= test_aug.flow_from_dataframe(
    dataframe=df_val,
    x_col="image_path",
    y_col="label",
    batch_size=16,
    color_mode="rgb",
    target_size = (224, 224),
    class_mode="categorical")

test_generator= test_aug.flow_from_dataframe(
    dataframe=df_test,
    x_col="image_path",
    y_col="label",
    color_mode="rgb",
    batch_size=16,
    shuffle = False,
    target_size = (224, 224),
    class_mode="categorical")
from tensorflow.keras.applications.vgg19 import VGG19
from keras.layers import Activation,Dense, Dropout, Flatten, Conv2D,
MaxPool2D, MaxPooling2D,AveragePooling2D, BatchNormalization, PReLU, ReLU
from keras.models import Model
from keras.applications.xception import Xception


def generate_model(pretrained_model = 'vgg19', num_classes=2):
    if pretrained_model == 'xception':
        base_model = Xception(weights = 'imagenet', include_top=False,
input_shape=(224, 224, 3))
    else:
        base_model = VGG19(weights = 'imagenet', include_top=False,
input_shape=(224, 224, 3)) # Topless

    x = base_model.output
    x = Flatten()(x)
```

```
    x = Dense(4096)(x)
    x = ReLU()(x)
    x = Dropout(0.5)(x)
    x = Dense(4096)(x)
    x = ReLU()(x)
    x = Dropout(0.5)(x)
    predictions = Dense(num_classes, activation='softmax')(x)
    model = Model(inputs=base_model.input, outputs=predictions)

    #Freezing Convolutional Base
    for layer in base_model.layers[:-3]:
        layer.trainable = False
    return model
def train_model(model, train_generator,val_generator, num_epochs,
optimizer, metrics):
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer,
                  metrics=metrics)
    early_stop =
tf.keras.callbacks.EarlyStopping(monitor="val_accuracy",patience=40,
verbose=1)
    rlr = tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
factor=0.1, patience=5)
    print(model.summary())

    history = model.fit(train_generator, epochs=num_epochs,
                        validation_data=val_generator, verbose=1,
                        callbacks = [early_stop, rlr])

    return model, history
import itertools
import sklearn
from sklearn.model_selection import train_test_split
from sklearn.utils import class_weight
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import roc_curve, auc, roc_auc_score,
confusion_matrix, classification_report

metrics = ['accuracy',
                tf.keras.metrics.AUC(),
                tfa.metrics.CohenKappa(num_classes = 2),
                tfa.metrics.F1Score(num_classes = 2),
                tf.keras.metrics.Precision(),
                tf.keras.metrics.Recall()]

def plot_loss(history):
    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')

def plot_acc(history):
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])
```

```
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'val'], loc='upper left')

# It prints & plots the confusion matrix, normalization can be applied by
setting normalize=True.

def plot_confusion_matrix(cm, classes,normalize=False,title='Confusion
Matrix for Potato Disease',cmap=plt.cm.Blues):

    plt.figure(figsize = (5,5))
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=90)
    plt.yticks(tick_marks, classes)
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]

    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]),
range(cm.shape[1])):
        plt.text(j, i, cm[i, j],
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

def plot_roc_curves(y_true, y_pred, num_classes, class_labels):

    lb = LabelBinarizer()
    lb.fit(y_true)
    y_test = lb.transform(y_true)

    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(num_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

#     # Plot all ROC curves
#     for i in range(num_classes):
        #fig, c_ax = plt.subplots(1,1, figsize = (6, 4))
        plt.plot(fpr[i], tpr[i],
                 label='ROC curve of class {0} (area = {1:0.2f})'
                 ''.format(class_labels[i], roc_auc[i]))
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC curve of class {0}'.format(class_labels[i]))
        plt.legend(loc="lower right")
    plt.show()
```

```
    return roc_auc_score(y_test, y_pred)
def plot_roc_curves(y_true, y_pred, num_classes, class_labels):

    lb = LabelBinarizer()
    lb.fit(y_true)
    y_test = lb.transform(y_true)

    # Compute ROC curve and ROC area for each class
    fpr = dict()
    tpr = dict()
    roc_auc = dict()
    for i in range(num_classes):
        fpr[i], tpr[i], _ = roc_curve(y_test[:, i], y_pred[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

#     # Plot all ROC curves
#     for i in range(num_classes):
        #fig, c_ax = plt.subplots(1,1, figsize = (6, 4))
        plt.plot(fpr[i], tpr[i],
                 label='ROC curve of class {0} (area = {1:0.2f})'
                 ''.format(class_labels[i], roc_auc[i]))
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC curve of class {0}'.format(class_labels[i]))
        plt.legend(loc="lower right")
    plt.show()
    return roc_auc_score(y_test, y_pred)
def evaluate_model(model, history, test_generator):
    # Evaluate model
    score = model.evaluate(test_generator, verbose=0)
    print('\nTest set accuracy:', score[1], '\n')

    y_true = np.array(test_generator.labels)
    y_pred = model.predict(test_generator, verbose = 1)
    y_pred_classes = np.argmax(y_pred,axis = 1)
    class_labels = list(test_generator.class_indices.keys())

    print('\n', sklearn.metrics.classification_report(y_true,
y_pred_classes, target_names=class_labels), sep='')
    confusion_mtx = confusion_matrix(y_true, y_pred_classes)
    plot_acc(history)
    plt.show()
    plot_loss(history)
    plt.show()
    plot_confusion_matrix(confusion_mtx, classes = class_labels)
    plt.show()
    print("ROC AUC score:", plot_roc_curves(y_true, y_pred, 9,
class_labels))
vgg_model = generate_model('vgg19', 2)
vgg_model, vgg_history = train_model(vgg_model, train_generator,
val_generator, 20, tf.keras.optimizers.SGD(lr=0.001, momentum=0.9),
metrics)
evaluate_model(vgg_model, vgg_history, test_generator)
import pandas as pd
from keras.preprocessing.image import load_img,img_to_array
image_path= df_test['image_path'][80]
```

```
img = load_img(image_path, target_size=(224,224,3)) # stores image in PIL
format
image_array=img_to_array(img)
from PIL import Image
display(Image.open(image_path))
def make_gradcam_heatmap(img_array, model, last_conv_layer_name,
pred_index=None):
    # First, we create a model that maps the input image to the
activations
    # of the last conv layer as well as the output predictions

    grad_model = tf.keras.models.Model([model.inputs],
[model.get_layer(last_conv_layer_name).output, model.layers[-2].output])

    # Then, we compute the gradient of the top predicted class for our
input image
    # with respect to the activations of the last conv layer
    with tf.GradientTape() as tape:
        last_conv_layer_output, preds = grad_model(img_array)
        if pred_index is None:
            pred_index = tf.argmax(preds[0])
        class_channel = preds[:, pred_index]

    # This is the gradient of the output neuron (top predicted or chosen)
    # with regard to the output feature map of the last conv layer
    grads = tape.gradient(class_channel, last_conv_layer_output)

    # This is a vector where each entry is the mean intensity of the
gradient
    # over a specific feature map channel
    pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))

    # We multiply each channel in the feature map array
    # by "how important this channel is" with regard to the top predicted
class
    # then sum all the channels to obtain the heatmap class activation
    last_conv_layer_output = last_conv_layer_output[0]
    heatmap = last_conv_layer_output @ pooled_grads[..., tf.newaxis]
    heatmap = tf.squeeze(heatmap)

    # For visualization purpose, we will also normalize the heatmap
between 0 & 1
    heatmap = tf.maximum(heatmap, 0) / tf.math.reduce_max(heatmap)
    return heatmap.numpy()
# Make model
model = vgg_model
last_conv_layer_name ="block5_conv3"
# Remove last layer's softmax
model.layers[-1].activation = None

img_array=np.expand_dims(image_array, axis=0)
# Prepare particular image

# Generate class activation heatmap
heatmap= make_gradcam_heatmap(img_array, model, last_conv_layer_name)
```

```
# Display heatmap
plt.matshow(heatmap)
plt.show()
import matplotlib.cm as cm
def save_and_display_gradcam(img_path, heatmap, cam_path, alpha=0.4):
    # Load the original image
    img = tf.keras.preprocessing.image.load_img(img_path)
    img = tf.keras.preprocessing.image.img_to_array(img)

    # Rescale heatmap to a range 0-255
    heatmap = np.uint8(255 * heatmap)

    # Use jet colormap to colorize heatmap
    jet = cm.get_cmap("jet")

    # Use RGB values of the colormap
    jet_colors = jet(np.arange(256))[:, :3]
    jet_heatmap = jet_colors[heatmap]

    # Create an image with RGB colorized heatmap
    jet_heatmap = tf.keras.preprocessing.image.array_to_img(jet_heatmap)
    jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
    jet_heatmap = tf.keras.preprocessing.image.img_to_array(jet_heatmap)

    # Superimpose the heatmap on original image
    superimposed_img = jet_heatmap * alpha + img
    superimposed_img =
tf.keras.preprocessing.image.array_to_img(superimposed_img)

    # Save the superimposed image
    superimposed_img.save(cam_path)

    # Display Grad CAM
    display(Image.open(cam_path))


# Display heatmap
plt.matshow(heatmap)
plt.show()
save_and_display_gradcam(image_path, heatmap,cam_path="GradCamTest1.jpg")
xception_model = generate_model('xception', 2)
xception_model, xception_history = train_model(xception_model,
train_generator, val_generator, 20, tf.keras.optimizers.SGD(lr=0.001,
momentum=0.9), metrics)
evaluate_model(xception_model, xception_history, test_generator)
# Make model
model = xception_model
last_conv_layer_name ="conv2d_296"
# Remove last layer's softmax
model.layers[-1].activation = None

img_array=np.expand_dims(image_array, axis=0)
# Prepare particular image

# Generate class activation heatmap
heatmap= make_gradcam_heatmap(img_array, model, last_conv_layer_name)
```

```
# Display heatmap
plt.matshow(heatmap)
plt.show()
save_and_display_gradcam(image_path, heatmap,cam_path="GradCamTest3.jpg")
```