Inteligencia Computacional - 2023/2024

Fase 3 - Desenvolvimento sustentável

Identificação da saúde de produtos vegetais

2021110042 a2021110042@isec.pt | Jorge Ricardo Marques Duarte

Índice

Descrição do problema

À medida que o planeta enfrenta desafios crescentes relacionados à segurança alimentar, e problemas com o uso responsável dos recursos naturais e mitigação das mudanças climáticas, a identificação precoce e eficaz de doenças em produtos vegetais como plantas e frutas emerge como uma solução vital no âmbito do desenvolvimento sustentável. Esta capacidade de detetar e responder a doenças nas culturas agrícolas não apenas promove práticas agrícolas mais eficientes, mas também desempenha um papel fundamental na preservação da biodiversidade, na redução do uso de produtos químicos e na promoção de sistemas alimentares sustentáveis, sendo este trabalho, o objetivo para a resolução destes problemas.

Metodologias utilizadas

No meu projeto, foquei em metodologias rigorosas e eficientes, onde inicialmente, procedi à recolha e preparação dos dados, redimensionando e normalizando imagens para treinamento neural.

Experimentei com arquiteturas de CNNs, como VGG16 e ResNet50, adaptando-as ao meu caso específico.

Na otimização dos hiperparâmetros, utilizei PSO, Random Search e Grid Search, procurando a configuração ideal.

Durante o treino e validação, utilizei validação cruzada e técnicas como Early Stopping e Model Checkpoint, visando maximizar a taxa de acerto e prevenir o overfitting, onde com isto avaliei os modelos com o conjunto de teste, alcançando alta taxa de acerto e robustez.

Apresentação da Arquitetura de Código

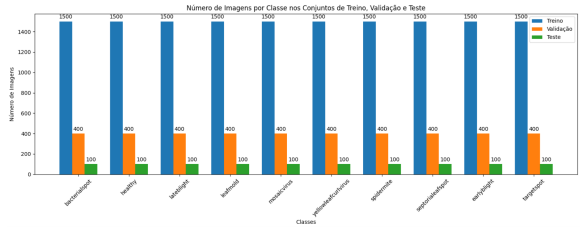
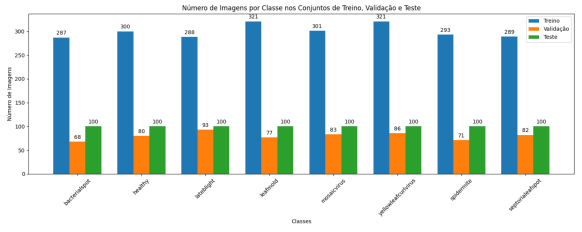
Recolha e preparação dos dados

Procedi à recolha e preparação dos dados para o treino do modelo de rede neural, onde inicialmente, recolhi imagens a partir de diretórios específicos para treino, validação e teste, organizadas em subdiretórios por categoria. Posteriormente, efetuei um pré-processamento destas imagens, redimensionando-as para o tamanho padrão de 64x64 pixels e normalizando os valores dos pixels através da divisão por 255. Estas etapas são fundamentais para adaptar as imagens ao formato necessário para um treino eficiente e rápido da rede neural.

Distribuições dos conjuntos para testes dos hiperparâmetros

Ao contrário da fase 1 e 2 onde os dados de treino foram usados todos em completo, sendo 2000 imagens por classe, 1500 treino, 400 validação e 100 para teste, nesta terceira fase foi feito de forma diferente.

Nesta terceira fase, os conjuntos de treino e validação foram reduzidos em 80%, ou seja, as imagens por cada classe de treino em vez de 1000 eram cerca de 280 enquanto os de validação cerca de 80 em vez das típicas 100.

Distribuição fase 1 e 2	Distribuição fase 3
	

Organização do Código

A organização do código deste projeto foi dividida em módulos que refletem diferentes etapas do processo de desenvolvimento de modelos de machine learning:

- **App.py**
  - É o script principal da aplicação de interface gráfica criada com Tkinter. Permite ao utilizador carregar modelos e imagens para fazer previsões de forma interativa.
- **Cnn_base-gridsearch.ipynb**
  - Jupyter notebook onde implementei o Grid Search para otimizar os hiperparâmetros da rede.
- **Cnn_base-pso-hyperparameters.ipynb**
  - Neste notebook, apliquei Particle Swarm Optimization para encontrar a melhor configuração dos hiperparâmetros.
- **Cnn_base-random-search.ipynb**
  - Aqui, utilizei o método Random Search para a otimização de hiperparâmetros, oferecendo uma abordagem mais aleatória e menos computacionalmente intensiva em comparação com o Grid Search.
- **Cnn_train_best_model.ipynb**
  - Este notebook é dedicado ao treino do modelo com os melhores hiperparâmetros encontrados através dos métodos de otimização mencionados.
- **Model_data.ipynb**
  - Um notebook que contém scripts para carregar e visualizar os dados do modelo, além de realizar avaliações.. A estrutura do projeto é desenhada para modularidade e reutilização, facilitando testes, ajustes e expansão futura.

Definição do modelo

Optei por experimentar com diferentes arquiteturas de CNN, incluindo **VGG16**, **ResNet50** e **Xception**, todas pré-treinadas no dataset ImageNet. As redes foram adaptadas para o meu caso específico, desativando o treino das camadas iniciais para aproveitar as características já aprendidas e adicionando novas camadas densas no topo. Estas novas camadas incluíam uma camada de achatamento (Flatten), seguida de uma camada densa com 512 neurônios e ativação ReLU, uma camada de Dropout para reduzir o overfitting, e finalmente uma camada de saída densa com ativação softmax, correspondendo ao número de categorias a serem classificadas. Este design foi pensado para combinar a eficácia das redes pré-treinadas com a especificidade do nosso conjunto de dados.

Estrutura definida para o modelo:

- **Base Model (VGG16/ResNet50/Xception):** Input: Imagens de tamanho 64x64x3. Camadas Convolutivas: Várias camadas, configuradas para não serem treináveis (trainable=False).
- **Custom Top Layers**:
    - Flatten: Achatamento da saída do modelo base.
    - Dense: 512 neurônios, ativação ReLU.
    - Dropout: Taxa de dropout definida pelo hiperparâmetro.
    - Dense: Número de neurônios igual ao número de categorias, ativação softmax.
- **Compilação:**
    - Otimizador: Adam, com taxa de aprendizagem definida pelo hiperparâmetro.
    - Função de Perda: Sparse Categorical Crossentropy.
    - Métricas: Taxa de acerto.

Esta estrutura Inicia com um modelo base pré-treinado (VGG16, ResNet50 ou Xception), ideal para aprendizagem por transferência, seguido de camadas personalizadas (top layers) que incluem Flatten, Dense e Dropout, atendendo à definição de uma rede densa para as últimas camadas. A otimização dos hiperparâmetros da rede densa, como a taxa de dropout e a taxa de aprendizagem, é realizada utilizando técnicas como PSO, Grid Search e Random Search.

Otimização dos hiperparâmetros

Na otimização dos hiperparâmetros do meu modelo de rede neural, apliquei três técnicas distintas: Particle Swarm Optimization (PSO), Random Search e Grid Search.

O PSO foi utilizado para encontrar combinações ideais de taxa de dropout e taxa de aprendizagem, através de um processo interativo e adaptativo baseado no comportamento de enxames.

Já o Random Search testou combinações aleatórias desses hiperparâmetros, proporcionando uma busca mais rápida, embora menos sistemática.

Por fim, utilizei o Grid Search para uma exploração exaustiva do espaço de hiperparâmetros, testando todas as combinações possíveis dentro dos limites definidos.

Estas abordagens foram cruciais para explorar amplamente o espaço de hiperparâmetros e identificar a configuração ótima para maximizar a performance do modelo.

Treino e validação

No treino e validação do modelo, usei um conjunto de dados separado em três partes: treino, validação e teste. O modelo foi treinado no conjunto de treino e validado no conjunto de validação para ajustar os hiperparâmetros e evitar o overfitting, onde durante esta etapa também utilizei técnicas como validação cruzada para garantir a generalização do modelo.

A performance foi avaliada com base na taxa de acerto e na área sob a curva ROC (AUC). Após cada fase de treino, o modelo foi validado para assegurar que as melhorias eram consistentes e aplicáveis a dados não vistos.

Classes dos modelos

Durante o treino dos modelos, dado as etapas anteriores, acabei por reduzir o número de classes de 10 para 8, dado a taxa de acerto das 8 classes andar a rondar os 84%. Com isto, nesta última fase, foram criados modelos destes 2 tipos. Modelos com 8 classes e modelos com 10 classes.

As categorias dos modelos de 8 classes são as seguintes:

["bacterialspot", "healthy","lateblight","leafmold", "mosaicvirus","yellowleafcurlvirus" ,"spidermite","septorialeafspot"]

As categorias dos modelos de 10 classes são as seguintes:

["bacterialspot", "healthy","lateblight","leafmold", "mosaicvirus","yellowleafcurlvirus" ,"spidermite","septorialeafspot","earlyblight", "targetspot"]

Descrição da Implementação dos algoritmos

Algoritmo PSO

Na implementação do algoritmo Particle Swarm Optimization (PSO), desenvolvi um processo iterativo de otimização de hiperparâmetros para a rede neural convolucional, com isto defini um espaço de busca bi-dimensional, representando a taxa de dropout e a taxa de aprendizagem, com limites específicos para cada um. Utilizei o PSO com um número definido de agentes e iterações, em conjunto com a validação cruzada K-Fold, para avaliar o desempenho das configurações de hiperparâmetros com base na métrica AUC. O objetivo foi encontrar o conjunto de hiperparâmetros que maximiza a AUC média, ajustando o modelo de maneira eficiente e eficaz.

O Particle Swarm Optimization (PSO) é uma técnica de otimização bioinspirada que simula o comportamento social de enxames, como peixes e pássaros. No PSO, cada 'partícula' representa uma solução potencial no espaço de busca do problema. Cada partícula ajusta sua posição baseando-se em sua própria experiência e na dos vizinhos, procurando pela melhor solução. Isso é feito considerando a melhor posição conhecida da partícula e a melhor posição global conhecida pelo enxame. O processo é iterativo e continua até que um critério de parada seja atingido, como um número máximo de iterações ou um limiar de desempenho.

```python
print("Iniciando a otimização com PSO...")
dimension = 2    # Número de dimensões (neste caso, dropout_rate e learning_rate)
lb = [0.1, 0.0001]   # Limites inferiores para dropout_rate e learning_rate
ub = [1, 0.01]    # Limites superiores para dropout_rate e learning_rate
num_agents = 2  # Número de agentes

max_iter = 10   # Número de iterações
total_epochs = 10
k_n_splits = 3 # k splits

best_auc_median_score = 0  # Variável global para armazenar o melhor auc_score

def evaluate_model_PSO(solution):
    global best_auc_median_score
    print(f" - Evaluating with cross validation, splits={k_n_splits}")
    dropout_rate, learning_rate = solution
    dropout_rate = abs(dropout_rate)
    learning_rate = abs(learning_rate)
    kfold = KFold(n_splits=k_n_splits, shuffle=True, random_state=42)
    auc_scores = []

    for train, val in kfold.split(x_train, y_train):
        model = create_cnn_model(dropout_rate, learning_rate)

        # Treina o modelo com os dados de treino e valida nos dados de validação
        model.fit(x_train[train], y_train[train], epochs=total_epochs, batch_size=32, verbose=0)
        y_pred = model.predict(x_train[val])
        y_val_categorical = to_categorical(y_train[val], num_classes=len(categories))
        auc_score = roc_auc_score(y_val_categorical, y_pred, multi_class='ovr')
        auc_scores.append(auc_score)
        print(f"   -- generating model with -> dropout_rate: {dropout_rate}, learning_rate: {learning_rate}, auc score: {auc_score}")

    auc_median_score_maximized = np.mean(auc_scores)
    auc_median_score_minimized = 1 - auc_median_score_maximized

    if(auc_median_score_maximized >= best_auc_median_score):
        best_auc_median_score = auc_median_score_maximized
    print(f" - generated cnn models -> dropout_rate: {dropout_rate}, learning_rate: {learning_rate}, auc median score (minimized): {auc_median_score_minimized}, maximized median auc: {auc_median_score_maximized}")

    # Retorna a média do AUC score (minimizado) 1- auc
    return auc_median_score_minimized


# Criando a instância do PSO
pso_instance = pso(n=num_agents, function=evaluate_model_PSO, lb=lb, ub=ub, dimension=dimension, iteration=max_iter)
best_solution = pso_instance.get_Gbest()

best_dropout_rate, best_learning_rate = best_solution
best_dropout_rate = abs(best_dropout_rate)
best_learning_rate = abs(best_learning_rate)

print(f"Melhores parâmetros encontrados: \nBest dropout rate: {best_dropout_rate}, Best Learning Rate: {best_learning_rate}")
print(f"Best AUC score: {best_auc_median_score}")
```

Algoritmo Random Search

Implementei este algoritmo como uma forma eficiente de sondar o espaço de hiperparâmetros para a rede neural convolucional. Diferentemente de métodos exaustivos, o Random Search seleciona aleatoriamente combinações de hiperparâmetros dentro de um intervalo pré-definido, testando-os por meio de validação cruzada. Utilizei K-Fold para garantir a validação robusta, e a métrica AUC para avaliação. Este método provou ser valioso por sua simplicidade e capacidade de descobrir configurações ótimas em um espaço de busca amplo e complexo.

```python
k_n_splits=3
n_iterations= 20
total_epochs = 5

def evaluate_model_cv(dropout_rate, learning_rate, x_train, y_train, n_splits=5):
    print(f" - Evaluating with cross validation, splits={n_splits}")
    kfold = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    auc_scores = []

    for train_index, val_index in kfold.split(x_train):
        model = create_cnn_model(dropout_rate, learning_rate)
        model.fit(x_train[train_index], y_train[train_index], epochs=total_epochs, batch_size=32, verbose=1)

        y_pred = model.predict(x_train[val_index])
        y_val_categorical = to_categorical(y_train[val_index], num_classes=len(categories))
        auc_score = roc_auc_score(y_val_categorical, y_pred, multi_class='ovr')
        auc_scores.append(auc_score)

    return np.mean(auc_scores)

def random_search(x_train, y_train, n_iter=100):
    best_score = -np.inf
    best_params = {'dropout_rate': None, 'learning_rate': None}
    best_auc_score = None

    for _ in range(n_iter):
        dropout_rate = abs(random.uniform(0.1, 0.99))
        learning_rate = abs(random.uniform(0.0001, 0.01))
        print(f"Testing params:  dropout_rate: {dropout_rate}, learning_rate: {learning_rate}")
        auc_score = evaluate_model_cv(dropout_rate, learning_rate, x_train, y_train, k_n_splits)

        if auc_score > best_score:
            best_score = auc_score
            best_params = {'dropout_rate': dropout_rate, 'learning_rate': learning_rate}
            best_auc_score = auc_score
            print(f"New best score: AUC={auc_score}, with parameters: dropout_rate={dropout_rate}, learning_rate={learning_rate}")

    return best_params, best_auc_score

best_params, best_auc_score = random_search(x_train, y_train, n_iter=n_iterations)
print(f"Melhores parâmetros encontrados: {best_params}, Melhor pontuação AUC: {best_auc_score}")
```

Algoritmo Grid Search

Para o Grid Search, estabeleci uma metodologia sistemática e abrangente, avaliando todas as combinações possíveis dos hiperparâmetros, taxa de dropout e taxa de aprendizagem. Com valores pré-definidos para cada parâmetro, utilizei a validação cruzada K-Fold para medir a eficácia de cada configuração. O desempenho foi quantificado pela média da área sob a curva ROC, e a combinação que resultou na maior métrica AUC foi selecionada como a melhor. Este processo meticuloso garantiu a escolha do conjunto de hiperparâmetros mais promissor para o modelo.

```python
import numpy as np
import random
from sklearn.model_selection import KFold
from tensorflow.keras.utils import to_categorical
from sklearn.metrics import roc_auc_score

dropout_rates = [0.1, 0.325, 0.55, 0.775, 0.9]  # Valores específicos para dropout_rates
learning_rates = [0.0001, 0.00031623, 0.001, 0.00316228, 0.01]  # Valores específicos para learning_rates
k_n_splits=3
total_epochs = 20

def evaluate_model_cv(dropout_rate, learning_rate, x_train, y_train, n_splits=5):
    print(f" - Evaluating with cross validation, splits={n_splits}")
    kfold = KFold(n_splits=n_splits, shuffle=True, random_state=42)
    auc_scores = []

    for train_index, val_index in kfold.split(x_train):
        model = create_cnn_model(dropout_rate, learning_rate)
        model.fit(x_train[train_index], y_train[train_index], epochs=total_epochs, batch_size=32, verbose=1)

        y_pred = model.predict(x_train[val_index])
        y_val_categorical = to_categorical(y_train[val_index], num_classes=len(categories))
        auc_score = roc_auc_score(y_val_categorical, y_pred, multi_class='ovr')
        auc_scores.append(auc_score)

    return np.mean(auc_scores)


def grid_search(x_train, y_train, dropout_rates, learning_rates, n_splits=5):
    best_score = -np.inf
    best_params = {'dropout_rate': None, 'learning_rate': None}
    best_auc_score = None

    for dropout_rate in dropout_rates:
        for learning_rate in learning_rates:
            print(f"Testing params: dropout_rate: {dropout_rate}, learning_rate: {learning_rate}")
            auc_score = evaluate_model_cv(dropout_rate, learning_rate, x_train, y_train, n_splits)

            if auc_score > best_score:
                best_score = auc_score
                best_params = {'dropout_rate': dropout_rate, 'learning_rate': learning_rate}
                best_auc_score = auc_score
                print(f"New best score: AUC={auc_score}, with parameters: dropout_rate={dropout_rate}, learning_rate={learning_rate}")

    return best_params, best_auc_score


best_params, best_auc_score = grid_search(x_train, y_train, dropout_rates, learning_rates, n_splits=k_n_splits)
print(f"Melhores parâmetros encontrados: {best_params}, Melhor pontuação AUC: {best_auc_score}")
```

Análise de Resultados da melhoria dos Hiperparâmetros

Resultados

% reduction data	swarm	agents	lb	up	iterations	epochs	k_splits	dimensions	application	image_size	best_dropout_rate	best_learning_rate	best_median_auc_score
80	pso	2	[0.1, 0.0001]	[1, 0.01]	5	5	3	2	vgg16	64x64	0.1	0.005393309682	0.9756082851
80	pso	2	[0.1, 0.0001]	[1, 0.01]	5	5	4	2	vgg16	64x64	0.2203281391	0.005290318267	0.9760516026
80	pso	2	[0.1, 0.0001]	[1, 0.01]	10	5	3	2	vgg16	64x64	0.1611461741	0.004681839026	0.976612035
80	pso	2	[0.1, 0.0001]	[1, 0.01]	20	5	3	2	vgg16	64x64	0.1	0.000982049731 4	0.9778650022
80	pso	2	[0.1, 0.0001]	[1, 0.01]	5	10	3	2	vgg16	64x64	0.2571024635	0.001466815143	0.9736031269
80	pso	2	[0.1, 0.0001]	[1, 0.01]	10	10	3	2	vgg16	64x64	0.1756737896	0.010107774 86	0.9758420142

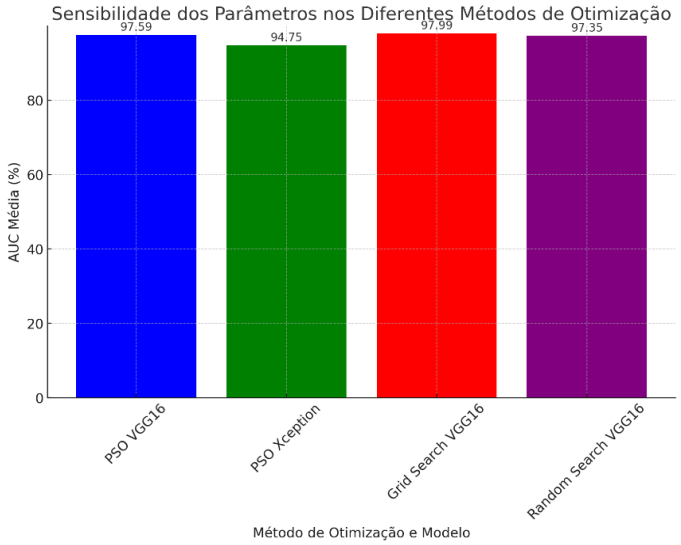
% reduction data	swarm	agents	lb	up	iterations	epochs	k_splits	dimensions	application	image_size	best_dropout_rate	best_learning_rate	best_median_auc_score
80	pso	2	[0.1, 0.0001]	[1, 0.01]	5	5	3	2	Xception	64x64	0.567004429 9	0.004303011936	0.9433017749
80	pso	2	[0.1, 0.0001]	[1, 0.01]	5	5	4	2	Xception	64x64	0.034528897775	0.001200486789	0.9533494316
80	pso	2	[0.1, 0.0001]	[1, 0.01]	10	5	3	2	Xception	64x64	0.1	0.009657809133	0.9465543291
80	pso	2	[0.1, 0.0001]	[1, 0.01]	20	5	3	2	Xception	64x64	0.5316690312	0.002107069786	0.950047628
80	pso	2	[0.1, 0.0001]	[1, 0.01]	5	10	3	2	Xception	64x64	0.5092727139	0.000802649070 9	0.955402628
80	pso	2	[0.1, 0.0001]	[1, 0.01]	10	10	3	2	Xception	64x64	0.5645662189	0.009108480917	0.9360720388

% reduction data	swarm	agents	dropout_rates	learning_rates	iterations	epochs	k_splits	dimensions	application	image_size	best_dropout_rate	best_learning_rate	best_median_auc_score
80	grid	2	[0.1, 0.325, 0.55, 0.775, 0.9]	[0.0001, 0.00031623, 0.001, 0.00316228, 0.01]	25	5	3	2	vgg16	64x64	0.1	0.00316228	0.9770894177
80	grid	2	[0.1, 0.325, 0.55, 0.775, 0.9]	[0.0001, 0.00031623, 0.001, 0.00316228, 0.01]	25	10	3	2	vgg16	64x64	0.1	0.001	0.9801659288
80	grid	2	[0.1, 0.325, 0.55, 0.775, 0.9]	[0.0001, 0.00031623, 0.001, 0.00316228, 0.01]	25	20	3	2	vgg16	64x64	0.325	0.001	0.982474924

% reduction data	swarm	agents	dropout_rates	learning_rates	iterations	epochs	k_splits	dimensions	application	image_size	best_dropout_rate	best_learning_rate	best_median_auc_score
80	random	2	[0.1 ... 0.99]	[0.0001, ... , 0,01]	5	5	3	2	vgg16	64x64	0.2921584934	0.0007337478094	0.9763715279
80	random	2	[0.1 ... 0.99]	[0.0001, ... , 0,01]	5	10	3	2	vgg16	64x64	0.5284691655	0.0077263 06312	0.9647729648
80	random	2	[0.1 ... 0.99]	[0.0001, ... , 0,01]	10	5	3	2	vgg16	64x64	0.3379288696	0.0040485 79834	0.9760248878
80	random	2	[0.1 ... 0.99]	[0.0001, ... , 0,01]	20	5	3	2	vgg16	64x64	0.2567627675	0.0013247 47899	0.9769568535

Análise

Em geral, consegui observar que os resultados dos diferentes métodos de otimização revelam nuances importantes sobre as suas aplicações. O PSO mostrou-se vantajoso em explorar espaços de solução contínuos e potencialmente alcançar um ótimo global, mas pode ser computacionalmente custoso com muitas iterações. O Grid Search, embora garantindo uma pesquisa exaustiva, pode ser impraticável em espaços de hiperparâmetros muito grandes devido ao seu alto custo computacional e tempo de execução. O Random Search oferece uma alternativa mais rápida, com a chance de encontrar soluções satisfatórias sem explorar todo o espaço, ideal para quando o tempo é um fator crítico, mas pode negligenciar áreas do espaço de busca que contém a solução ótima.



Relativamente aos resultados, podemos observar os seguintes pontos:

- O PSO com VGG16 alcançou um melhor AUC médio quanto maior o número de iterações, sugerindo que uma busca mais extensa no espaço de hiperparâmetros pode ser benéfica.
- Comparativamente ao Xception e VGG16, consegui observar que o VGG16 obtém resultados muito melhores de AUC comparativamente ao Xception, onde o VGG16 anda por médias de 97 e o Xception com médias de 94.
- Para a aplicação Xception, a taxa de dropout variou significativamente, indicando que a rede pode ser sensível a este parâmetro. Aqui, um maior número de iterações não correspondeu necessariamente a melhores resultados, sugerindo que o equilíbrio certo entre os hiperparâmetros é chave para otimização.

- Com o Grid Search, percebe-se que um aumento no número de épocas melhorou o AUC, indicando que um treinamento mais longo foi efetivo para o modelo VGG16.
- Os resultados da pesquisa aleatória (Random Search) também apresentaram melhorias com um número maior de épocas, mas com um intervalo mais amplo de hiperparâmetros, demonstrando que soluções eficientes podem ser encontradas fora dos valores tipicamente sugeridos.
- Por média de AUC podemos observar que o grid search acabou por obter a melhor média entre os 3 métodos.

Treino do melhor modelo e Análise

Melhoria e treino do melhor modelo

Após ter determinado os melhores hiperparâmetros para o modelo, passei para a fase da criação dos melhores modelos, onde fiz uso do early Stopping e Model Checkpoint para otimizar o treino do meu modelo de rede neural. O objetivo era alcançar uma taxa de acerto de validação superior determinada numa variável. Este limiar foi crucial para garantir que eu selecionasse o modelo mais eficiente, e após atingir essa taxa de acerto de validação, interrompi o treino, carreguei o modelo guardado e avaliei o desempenho sobre o conjunto de teste. A análise dos resultados mostrou a eficácia desta abordagem, não apenas em alcançar alta precisão, mas também em prevenir o overfitting, assegurando assim a generalização do modelo.

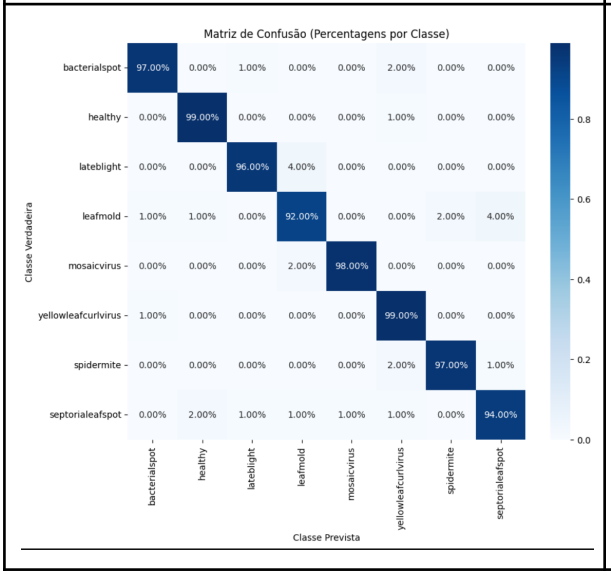
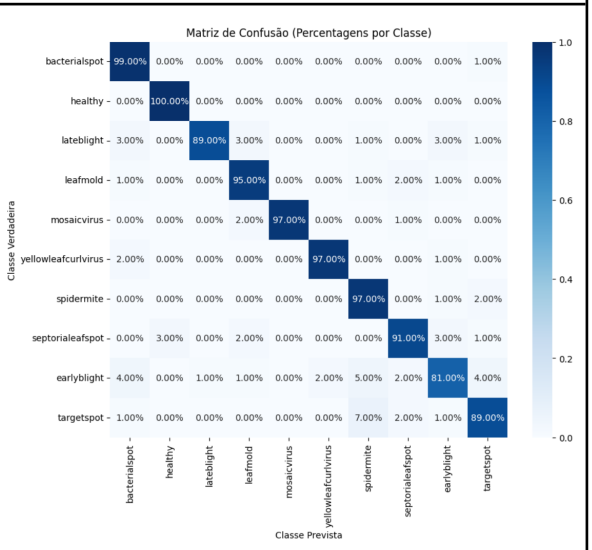
Para a finalidade de determinar o melhor modelo, baseei-me no AUC e na taxa de acerto sobre o conjunto de teste. Dado que os modelos nunca viram o conjunto de teste, o uso deste conjunto faz com que determinemos com certeza a taxa de acerto do modelo.

Este processo de avaliação foi o que tornou possível a criação e a existência dos dois melhores modelos no trabalho.

Para **10 classes**, obtive um modelo com taxa de acerto de **93.5%**, e para **8 classes** obtive um modelo com uma taxa de acerto de **96.4%**.

```python
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model

best_learning_rate = 0.001
dropout_rate = 0.225

val_accuracy_threshold = 0.965
early_stopping = EarlyStopping(monitor='val_loss', patience=20, min_delta=0.001, mode='min', verbose=1)
model_checkpoint = ModelCheckpoint('best_model.h5', monitor='val_accuracy', mode='max', save_best_only=True, verbose=0)

while True:
    print("A trainar o modelo...")
    best_model = create_cnn_model(dropout_rate, best_learning_rate)
    best_model_history = best_model.fit(x_train, y_train, epochs=60, batch_size=32, verbose=1, validation_data=(x_validation, y_validation),callbacks=[early_stopping, model_checkpoint])

    best_pso_train_accuracy = best_model_history.history['accuracy'][-1]
    best_pso_validation_accuracy = max(best_model_history.history['val_accuracy'])


    print("Melhor Accuracy de Treino: ", best_pso_train_accuracy)
    print("Melhor Accuracy de Validação: ", best_pso_validation_accuracy)

    if best_pso_validation_accuracy >= val_accuracy_threshold:
        print("Atingiu a acurácia desejada no conjunto de validação.")
        break

best_model = load_model('best_model.h5')

loss_test, test_accuracy = best_model.evaluate(x_test, y_test, verbose=0)
print(f"Acurácia no Teste: {test_accuracy}, Loss: {loss_test}")

plt.plot(best_model_history.history['accuracy'], label='Treinamento')
plt.plot(best_model_history.history['val_accuracy'], label='Validação')
plt.xlabel('Épocas')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

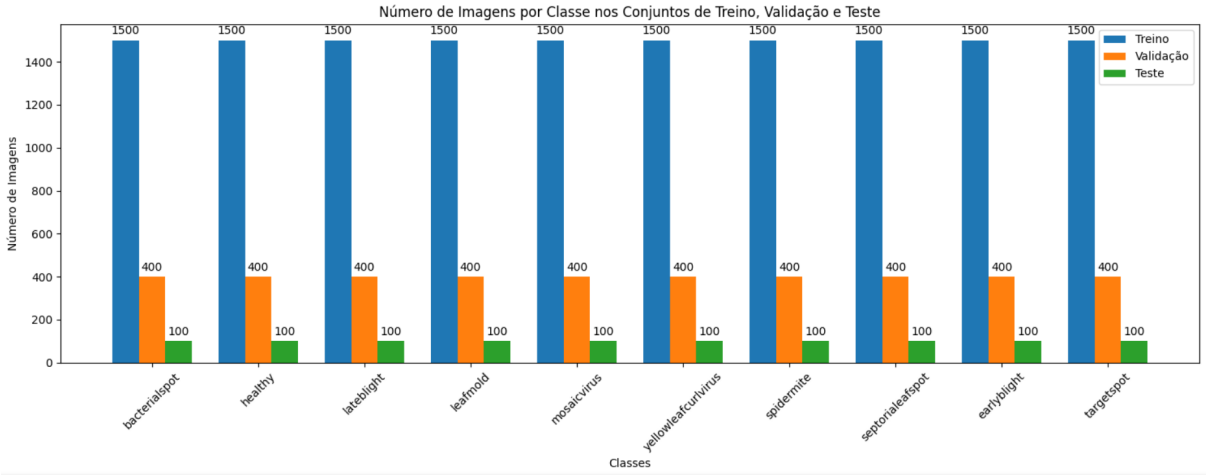
Matrizes de confusão sobre o conjunto de teste

Melhor modelo 8 Classes (**96.499%**)	Melhor modelo 10 Classes (**93.500%**)
	

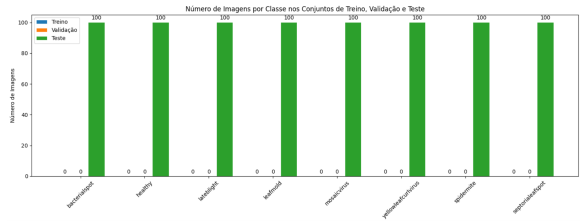
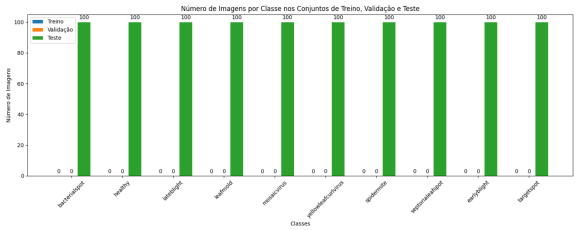
Como podemos observar, é visível as melhorias comparativamente com as etapas anteriores. Na primeira etapa deste projeto, estava a obter taxas de acerto de 91% para o modelo de 8 classes, e taxas de 85% para o modelo de 10 classes, onde agora obtivemos melhorias entre **5-9%**.

Distribuição dos conjuntos utilizados

Para a validação treino e teste do modelos, mantive as percentagens originais de 75% para treino, 10% para teste e 5% para validação.



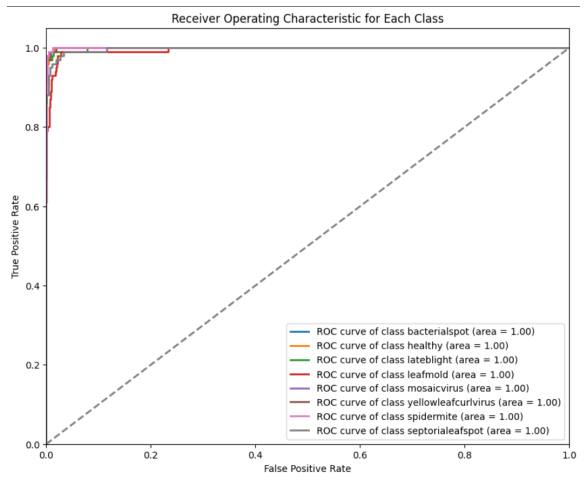
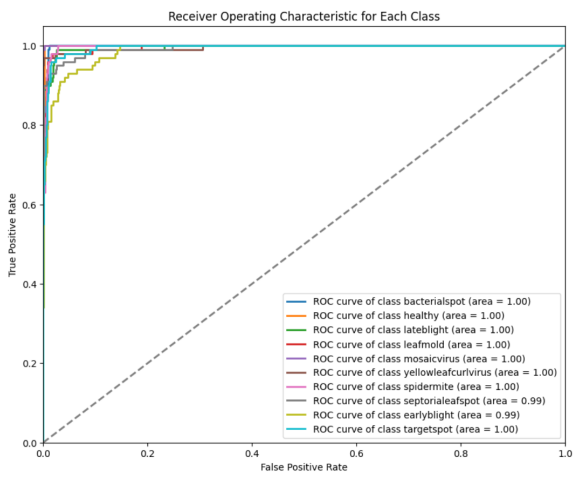
Relativamente aos conjunto de teste utilizado para avaliar estes modelos, foram os seguintes:

Conjunto de teste 8 classes	Conjunto de teste 10 classes
	

Esta parte está presente no model_data.ipynb onde aqui é feito e gerado vários gráficos de análise para os melhores modelos.

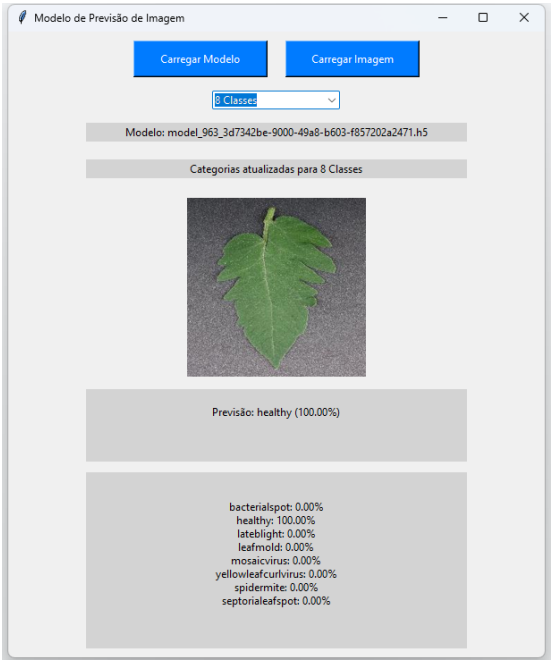
ROC AUC do conjunto de teste

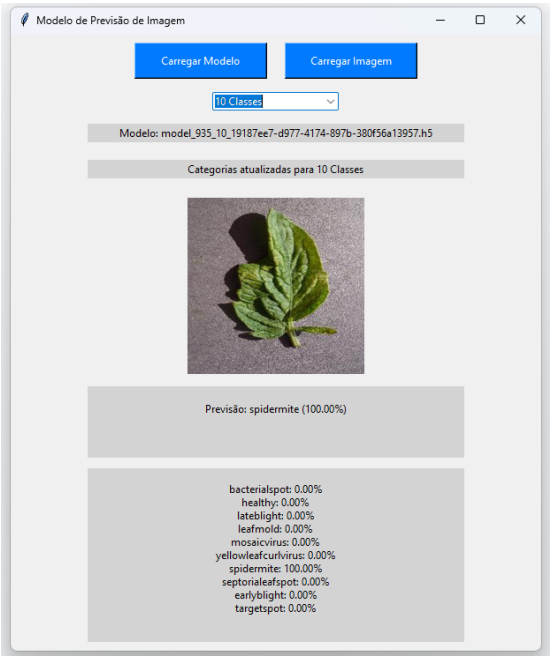
Relativamente ao ROC-AUC sobre o conjunto de teste:

ROC sobre o modelo 8 classes	ROC sobre o modelo 10 classes
	

Aplicação e uso dos modelos

Nesta última meta, aproveitei para desenvolver uma aplicação de teste utilizando Tkinter para o uso prático dos modelos gerados. Esta aplicação permite ao utilizador carregar modelos de rede neural e testar imagens para classificação. Implementei um seletor para alternar entre modelos de 8 e 10 classes, e um sistema de carregamento de imagens, onde as imagens são redimensionadas e processadas pelo modelo carregado. Os resultados, incluindo a classe prevista e a probabilidade associada, são exibidos na interface. Esta aplicação representa um passo importante para a aplicação prática dos modelos em um ambiente de utilizador final.





Conclusões

Para concluir, os métodos de otimização revelaram a eficácia do PSO e a precisão do Grid Search, com a VGG16 mostrando melhores resultados de AUC do que a Xception.

A implementação de EarlyStopping e ModelCheckpoint no treino resultou em modelos altamente precisos, com uma notável melhoria em relação às etapas anteriores.

O uso prático dos modelos foi demonstrado através de uma aplicação Tkinter que desenvolvi, permitindo testes interativos de classificação de imagens.

Esta terceira fase e a implementação de algoritmos para otimização dos hiperparâmetros não só melhorou o desempenho dos modelos em termos de taxa de acerto e generalização, mas também estabeleceu uma base sólida para aplicações práticas em ambientes reais.

Bibliografia

ChatGPT, OpenAI. "Discussão sobre Early Stopping e Model Checkpoint." Conversa com o usuário, [13/12/2023].

ChatGPT, OpenAI. "Desenvolvimento e Implementação de Aplicação de Teste em Tkinter." Conversa com o usuário, [14/12/2023].

ChatGPT, OpenAI. "Discussão sobre Otimização de Hiperparâmetros em Redes Neurais." Conversa com o usuário, [16/12/2023].