INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL MECÂNICA
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

Igor Rhiân Dalavechia

DESENVOLVIMENTO DE UM SISTEMA DE CLASSIFICAÇÃO DE DOENÇAS EM FOLHAS DE TOMATE EMPREGANDO REDES NEURAIS CONVOLUCIONAIS

FLORIANÓPOLIS, 2024.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA - CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL MECÂNICA
CURSO DE GRADUAÇÂO EM ENGENHARIA MECATRÔNICA

Igor Rhiân Dalavechia

DESENVOLVIMENTO DE UM SISTEMA DE CLASSIFICAÇÃO DE DOENÇAS EM FOLHAS DE TOMATE EMPREGANDO REDES NEURAIS CONVOLUCIONAIS

Trabalho de Conclusão de Curso submetido ao Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina como parte dos requisitos para obtenção do título de Engenheira Mecatrônica

Orientador:
Prof. MAURICIO EDGAR STIVANELLO, Dr. Eng

FLORIANÓPOLIS, 2024.

Ficha de identificação da obra elaborada pelo autor.

DALAVECHIA, IGOR RHIÂN
DESENVOLVIMENTO DE UM SISTEMA DE CLASSIFICAÇÃO DE DOENÇAS EM FOLHAS DE TOMATE EMPREGANDO REDES NEURAIS CONVOLUCIONAIS / IGOR RHIÂN DALAVECHIA; orientação de MAURICIO EDGAR STIVANELLO. - Florianópolis, SC, 2024.
66 p.

Trabalho de Conclusão de Curso (TCC) - Instituto Federal de Santa Catarina, Câmpus Florianópolis. Bacharelado em Engenharia Mecatrônica. Departamento Acadêmico de Metal Mecânica.
Inclui Referências.

1. Visão computacional. 2. CNN. 3. Folhas de tomate. 4. Aprendizado de máquina. 5. FASTAI. I. STIVANELLO, MAURICIO EDGAR . II. Instituto Federal de Santa Catarina. III. DESENVOLVIMENTO DE UM SISTEMA DE CLASSIFICAÇÃO DE DOENÇAS EM FOLHAS DE TOMATE EMPREGANDO REDES NEURAIS CONVOLUCIONAIS.

DESENVOLVIMENTO DE UM SISTEMA DE CLASSIFICAÇÃO DE DOENÇAS EM FOLHAS DE TOMATE EMPREGANDO REDES NEURAIS CONVOLUCIONAIS

IGOR RHIÂN DALAVECHIA

Este trabalho foi julgado adequado para obtenção do título de Engenheiro em Mecatrônica e aprovado na sua forma final pela banca examinadora do Curso de Engenharia Mecatrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Florianópolis, 25 de Setembro, 2024.

Banca Examinadora:

gov.br
Documento assinado digitalmente
MAURICIO EDGAR STIVANELLO
Data: 26/09/2024 14:44:59-0300
Verifique em https://validar.iti.gov.br

MAURICIO EDGAR STIVANELLO, Dr. Eng (Orientador)

gov.br
Documento assinado digitalmente
VALDIR NOLL
Data: 25/09/2024 18:38:44-0300
Verifique em https://validar.iti.gov.br

Valdir Noll, Dr. Eng

gov.br
Documento assinado digitalmente
DELCINO PICININ JUNIOR
Data: 26/09/2024 13:36:06-0300
Verifique em https://validar.iti.gov.br

Delcino Picinin Júnio. Dr. Eng

AGRADECIMENTOS

Agradeço à minha família pelo suporte incondicional ao longo desta jornada. Aos meus amigos, que tornaram esse percurso leve, alegre e agradável, meu sincero obrigado por estarem sempre presentes. À minha namorada, que sempre esteve ao meu lado, oferecendo apoio e motivação. Sua presença foi essencial para que eu pudesse superar os desafios e seguir em frente sou eternamente grato.

Agradeço também ao corpo docente, especialmente ao professor Maurício Edgar Stivanello, pela paciência, compreensão e valiosas contribuições ao meu trabalho, sem elas, este projeto não teria sido possível.

RESUMO

Este trabalho apresenta o desenvolvimento de um aplicativo móvel voltado para a detecção de doenças em folhas de tomate, utilizando técnicas de visão computacional, com foco em redes neurais convolucionais (RNCs). As doenças em folhas de tomate são um desafio significativo na agricultura, pois comprometem a saúde das plantas e a produtividade das culturas. O aplicativo desenvolvido permite a captura de imagens das folhas de tomate e, através de um modelo de RNC treinado, obtenham-se a classificação dessas folhas, identificando a presença de doenças. O projeto envolveu a preparação de um conjunto de dados, o treinamento do modelo, a criação de uma API para processar as imagens e a implementação de uma interface para dispositivos moveis. Os resultados obtidos demonstraram a eficácia do aplicativo na classificação de folhas saudáveis e doentes, com alta precisão.

Palavras-chave: Visão computacional. Aprendizado de máquina. Detecção de doenças. Folhas de tomate. FASTAI.

ABSTRACT

This work presents the development of a mobile application aimed at detecting diseases in tomato leaves using computer vision techniques, with a focus on Convolutional Neural Networks (CNNs). Diseases in tomato leaves pose a significant challenge in agriculture as they compromise plant health and crop productivity. The developed application enables the capture of images of tomato leaves and, through a trained CNN model, classifies these leaves to identify the presence of diseases. The project involved preparing a dataset, training the model, creating an API to process the images, and implementing an interface for mobile devices. The results demonstrated the effectiveness of the application in classifying healthy and diseased leaves with high accuracy.

Keywords: Computer vision. Machine learning. Disease detection. Tomato leaves. FASTAI.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

As doenças que afetam as folhas de tomate são um desafio significativo para os produtores, influenciando diretamente a saúde das plantas e, consequentemente, a produtividade das lavouras. Essas doenças, muitas vezes causadas por fungos, bactérias e vírus, podem se manifestar de várias formas, incluindo manchas, murchas e descolorações, que enfraquecem a planta e prejudicam o desenvolvimento dos frutos.

A visão computacional é amplamente utilizada na indústria, partindo de sensores de visão até algoritmos clássicos, já que os erros encontrados em uma produção tendem a seguir um padrão, como a presença de peças em um determinado produto. Porém, quando queremos tratar padrões orgânicos e classificação de erros, como diferentes tipos de manchas em uma folha de tomate, é necessário partir para o aprendizado de máquina *(Machine Learning)*. *Machine learning* é um conjunto de regras que permite a computadores aprenderem a partir de um conjunto de dados para gerar e aperfeiçoar previsões (MOLNAR, 2019).

O aprendizado profundo é uma técnica computacional de *machine learning* para extrair e transformar dados, com casos de uso que vão desde o reconhecimento de fala humana até a classificação de imagens de animais, usando múltiplas camadas de redes neurais. Cada uma dessas camadas recebe informações das camadas anteriores e as refina progressivamente. As camadas são treinadas por algoritmos que minimizam seus erros e melhoram sua precisão. Dessa forma, a rede aprende a realizar uma tarefa específica (HOWARD e GUGGER, 2020).

Nas últimas décadas, houve um crescimento imenso na tecnologia, principalmente em telefones celulares. Esse avanço permitiu o uso desses dispositivos de forma casual, e a difusão foi tanta que, atualmente, 96,3% dos domicílios brasileiros possuem celulares, sendo que, em áreas urbanas, esse valor representa 97,2%, e em áreas rurais, 90% da população (IBGE, 2021). O número de aplicativos para pequenos, médios e grandes produtores terá elevado desenvolvimento, especialmente para a gestão das áreas agrícolas, manejo de rebanhos, cotação de insumos, previsão do clima, identificação de doenças, uso de defensivos, irrigação, código florestal e comercialização (EMBRAPA, 2022).

Considerando o cenário apresentado, este trabalho busca estudar o âmbito da classificação de imagem utilizando *deep learning* e a comunicação dos seus resultados para um dispositivo móvel através de requisições a um servidor API. Isso tem como objetivo sanar a necessidade de detecção de doenças em tomates de forma rápida, eficaz e portátil, desprovido de análise humana.

1.1 Objetivo Geral

Desenvolver uma ferramenta de visão computacional que consiga classificar tipos de doenças em folhas de tomate utilizando técnicas de *machine learning*.

1.2 Objetivos Específicos

Para auxiliar na sua conclusão, o trabalho foi dividido nos seguintes objetivos especificos:

a) Seleção do banco de dados;

b) Implementação do algoritimo para criação do modelo;

c) Realizar o treinamento dos diferentes modelos;

d) Avaliar os resultados obtidos;

e) Criação e comunicação do aplicativo e da API;

f) Documentação dos resultados.

2 FUNDAMENTAÇÃO TEÓRICA

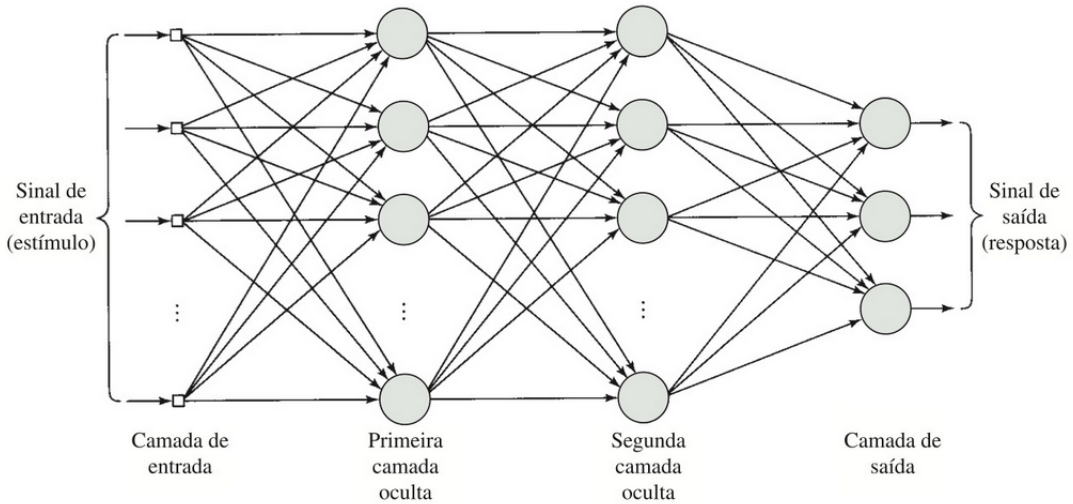
Nesse capítulo são abordados os conceitos essenciais para o entedimento do trabalho.

2.1 Redes Neurais Artificiais

As Redes Neurais Artificiais (RNA) trabalham através do método de aprendizado e são inspiradas nos estudos da maneira como se organizam e comunicam os neurônios humanos. A representação de conhecimentos nas redes conexionistas, como diz o próprio nome, é fortemente ligada a noção de conexão entre neurônios, elementos processadores de informação, que interagem uns com os outros através destas ligações. O modelo conexionista possui sua origem nos estudos feitos sobre as estruturas de nosso cérebro, sofrendo uma grande simplificação do modelo original, onde encontramos no modelo artificial, que é simulado, elementos como os neurônios e as suas conexões, chamadas de sinapses.(OSÓRIO e BITTENCOURT,2000).

Um dos primeiros modelo de RNA criado foi o Perceptron, por Rosenblatt , como é possível ver na figura 1. Neste modelo os neurônios estão dispostos em várias camadas, que podem ser classificadas em camadas de entrada, camadas ocultas e camadas de saída, onde, respectivamente, os pesos dos dados de entrada são definidos, o processamento é feito e o resultado da rede é apresentado.

Figura 1 – Arquitetura Peceptron com multiplas camdas



FONTE: HAYKIN, Simon(2001)

O primeiro modelo de neurônio artificial foi proposto por Warren McCulloch e Walter Pitts em 1943, através desse modelo é possivel ter um melhor entendimento de como funciona uma RNA. Eles propuseram, matematicamente, que a sinapse em um neurônio computacional pode ser representada pela seguinte equação:

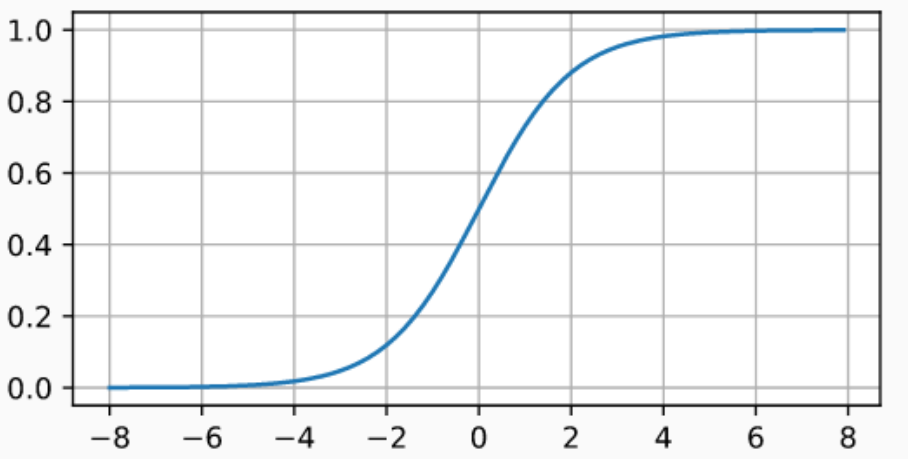
$$y = g(\sum_{i=1}^{n} W_i X_i + b) \tag{1}$$

- n é o nº de entradas;
- W é o peso;
- X é o valor da entrada;
- b viés ou bias;
- g() é a função ativação;
- y é a saida.

O modelo calcula a soma ponderada das entradas, adiciona um viés(b) e passa o resultado através de uma função de ativação para obter a saída do neurônio. Este é o conceito fundamental por trás de um neurônio artificial em uma rede neural.

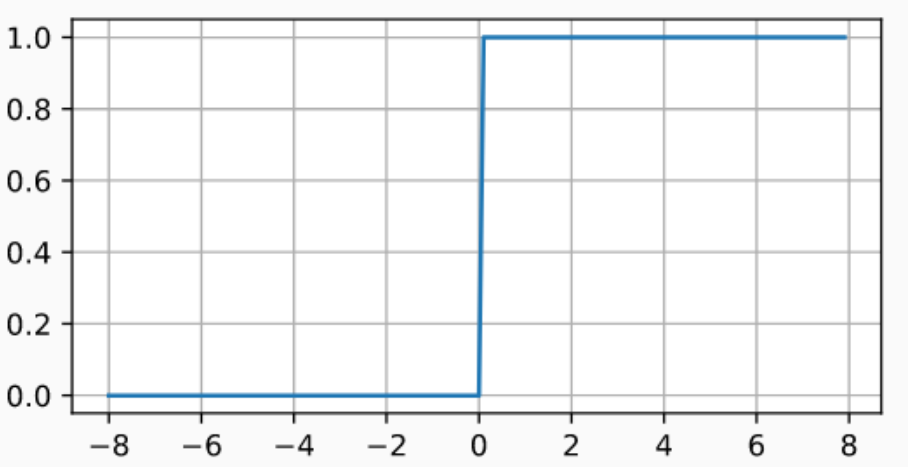
O viés (b) desempenha um papel crucial na ativação do neurônio, determinando se ele será ativado ou inibido com base na entrada e na função de ativação. A função de ativação, por sua vez, normaliza a saída resultante, garantindo que ela esteja dentro de um intervalo específico, como [0, 1] ou [-1, 1], dependendo da função escolhida. Isso é essencial para que a saída do neurônio possa ser interpretada corretamente pelo próximo neurônio na rede neural. Nas figuras abaixo, podemos ver 2 funções de ativação comuns : uma senoide e uma limiar, ambas dentro do intervalo de [0,1].

Figura 2 – Função Ativação Sigmoide



FONTE: Do Autor(2023)

Figura 3 – Função Ativação Limiar



FONTE:Do Autor(2023)

2.2 Redes Neurais Convolucionais (RNC)

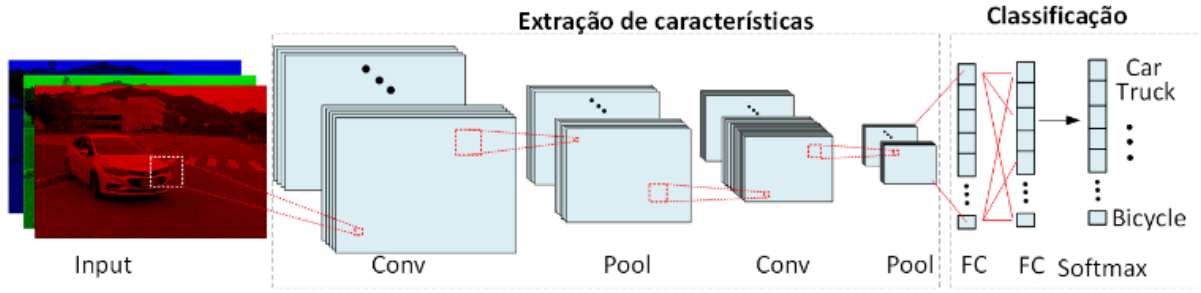
A capacidade de distinguir, classificar e comparar imagens e vídeo é inerente a seres humanos e muitos seres vivos, porém para um computador isso é algo muito mais complexo. Dentro de todas as abordagens atuais do aprendizado profundo a arquitetura de Redes Neurais Convolucionais(RNC) é a mais usada para identificação de dados gráficos, como videos e imagens. As camadas das RNC's foram criadas para identificar padrões simples como contornos para então começar a identificar diferenças entre rostos e objetos, como é descrito pelo autores Bhatt et al.(2021).

As RNC's foram influenciadas pela organização e funções do córtex visual, projetada para se assemelhar às conexões entre os neurônios no cérebro humano. Como dito anteriormente, as rede convolucionais são compostas de múltiplas camadas que podem ser descritas como: entrada, camadas de convolução, camada de

agrupamento e camada de saída como demonstrado na figura 4. A diferença entre as RNC's e outras redes/métodos de classificação é a eficiência com que os dados são tratados, sendo possivel trabalhar com dados mais complexos. Assim como a autora (LEMOS,2021,p.43) conclui:

> As redes neurais tradicionais recebem uma entrada, um único vetor que é transformado através de uma série de camadas ocultas. Cada camada oculta é composta de um conjunto de neurônios, onde cada neurônio está completamente conectado a todos os neurônios da camada anterior e onde os neurônios de uma única camada trabalham de forma totalmente independente e não compartilham nenhuma conexão. Essas redes não podem ser facilmente escalonadas para imagens complexas, por exemplo, uma imagem de tamanho 32x32x3, deve ter um neurônio totalmente conectado em uma primeira camada oculta com 32*32*3 = 3072 pesos, isso significa que imagens maiores aumentarão para um número excessivo de pesos que serão difíceis de manejar, tornando sua conectividade um desperdício.

Figura 4 – Representação da estrutura de uma RNC



FONTE: ERAZO(2021)

2.2.1 Camada convolucional

A camada convolucional recebe da camada de entrada os dados inseridos na rede e aplica o processo de convolução neles. Esse processo consiste em aplicar filtros nas imagens de entrada de cada neurônio. Segundo os autores Bhatt et al. (2021), o objetivo principal da convolução é extrair informações significativas de pontos específicos desses dados.

Um filtro, também chamado de Kernel, é uma matriz que caminha pela imagem aplicando operações lineares nos pixels das imagens. Dependendo de qual característica ou padrão é procurado para se destacar, os filtros podem ser alterados pelos hiperparâmetros: *Stride* e *Padding*. O resultado das operações dos filtros compõem o que é chamado de mapa de características. Ainda segundo os autores Bhatt et al. (2021), a saída dos kernels convolucionais, mapa de características, alimenta a função de ativação da RNC, o que auxilia no aprendizado e incorpora a não linearidade. Isso significa que a rede neural pode prever múltiplas classes, não se limitando a respostas de sim ou não.

Dependendo da aplicação, é necessário escolher um tipo de filtro específico. Alguns exemplos são: binarização preto e branco ou colorida, aumento ou diminuição de contraste e/ou brilho, suavização para eliminar ruídos, detecção de contornos, entre outros. Na imagem abaixo, é possível ver o exemplo do filtro Laplacian:

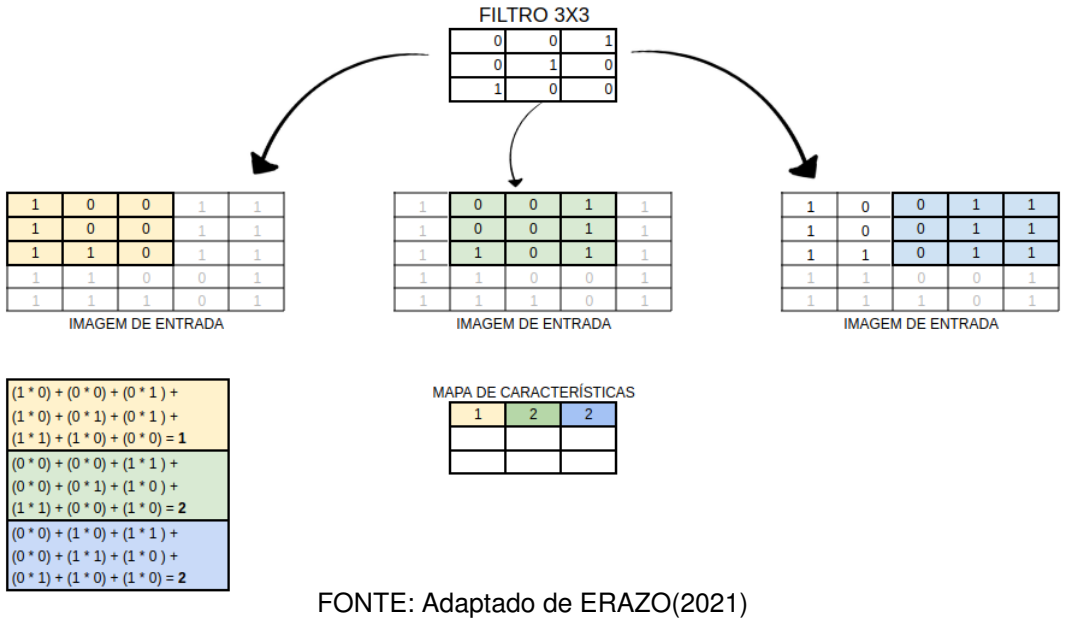
Figura 5 – Exemplo de filtro Laplacian - Filtrada esquerda e original a direita

Imagem com filtro laplacian aplicado — Imagem original

FONTE:Do Autor(2024)

Normalmente a entrada é uma imagem que pode ser representada por uma matriz de pixels (ixj), o filtro, matriz (NxM), percorre pela entrada multiplicando os valores (i,j) pelos seus pesos. Ao final o produto de cada posição é somado para formar o mapa de características ERAZO(2021), como demonstrado na figura 6.

Figura 6 – Exemplo do funcionamento de um filtro/kernel 3x3

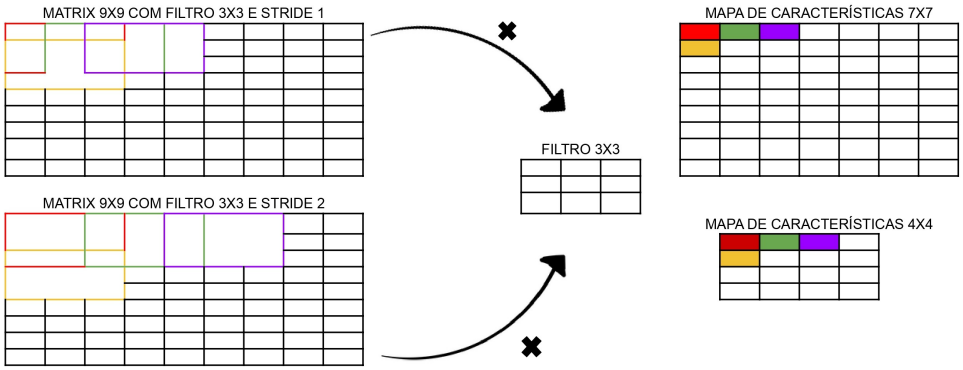


FONTE: Adaptado de ERAZO(2021)

2.2.1.1 Stride (passo)

O *stride* (passo) é o quanto o *kernel* se movimenta na imagem. Por exemplo um passo de valor 1 fará com que o *kernel* se movimente 1 pixel por vez, um passo de valor 2 fará o *kernel* se movimentar 2 pixels por vez e assim sucessivamente. Grandes valores de passo resultam em volumes de saída menores e segundo ERAZO(2021) isso inviabiliza modelos profundos pois o mapa de características fica menor a cada convolução, como é possivel ver na figura 7, onde um filtro 3x3 é aplicado sobre a entrada de tamanho 9x9 com valores de passo 1 e 2 obtendo-se, respectivamente, um mapa de tamanho 7x7 e 4x4.

Figura 7 – Exemplo de filtro 3x3 com stride de valor 1 e 2



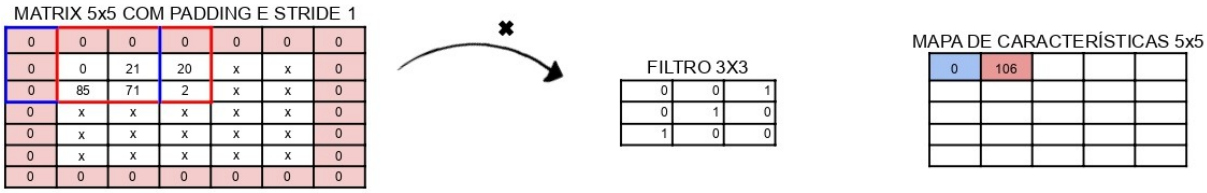
FONTE: Adaptado de ERAZO(2021)

Ainda segundo ERAZO(2021) grande valores de passo acarretam a perdas de informações, por que o filtro não é capaz de navegar toda a imagem, assim deixando os dados das bordas de fora da convolução. Não só isso os dados encontrados no centro da imagem tem grande reptibilidade o que pode fazer a RNC focar em detalhes indesejados.

2.2.1.2 Padding (preenchimento)

Padding é um método criado para solucionar os problemas citados em 2.2.1.1, onde um filtro não consegue circular por todos os pixels da imagem. Esse hiperparâmetro consiste em criar pixels ao redor da imagem, contornando todo o seu tamanho, com o valor de 0. Isso aumenta o tamanho dos dados de entrada e possibilita que o tamanho da saída, o mapa de características, seja controlado, como é mostrado na Figura 8. Isso permite a criação de redes mais profundas e a obtenção de mais informações nas laterais da imagem.

Figura 8 – Exemplo de Padding com filtro 3x3 com stride de valor 1



FONTE: Adaptado de ERAZO(2021)

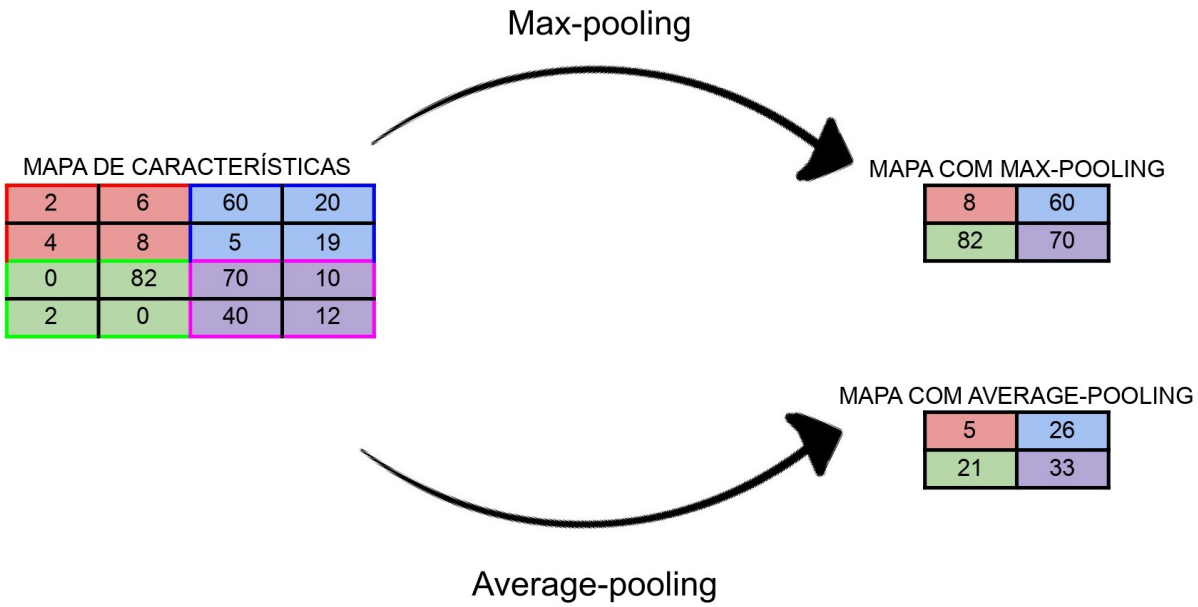
2.2.2 Camada de Pooling (subamostragem)

Após a camada de convolução, é necessário adicionar uma camada de pooling, também chamada de camada de agrupamento ou subamostragem. Segundo LEMOS(2021), o objetivo dessa etapa é reduzir progressivamente o tamanho das imagens de entrada. Consequentemente, isso reduz a carga computacional da rede e controla o sobreajuste (overfitting). Os autores Bhatt et al. (2021) afirmam que essa camada também ajuda na extração de informações que são sensíveis a rotações e translações, ajuda a identificar, por exemplo, números rotacionados, sem alterar a profundidade da entrada e reduz o tempo de treinamento. Essa camada irá atuar nos mapas de características de forma individual.

Pooling é normalmente utilizado na forma de duas funções:

- **Max-pooling:** Normalmente é utilizado um filtro 2x2 com um passo de 2 para reduzir o tamanho do mapa pela metade. Nessa função como podemos ver na figura 9, o max-pooling extrairá o pixel com o maior velor dentro do filtro e formará um novo mapa de características.
- **Average-pooling:** Assim como o Max-pooling, normalmente é utilizado um filtro 2x2 com um passo de 2 para reduzir o tamanho do mapa pela metade. Nessa função como podemos ver na figura 9, o average-pooling extrairá o valor médio dos pixels dentro do filtro e formará um novo mapa de características.

Figura 9 – Exemplo de Pooling com filtro 2x2 e stride de valor 2



FONTE: Adaptado de ERAZO(2021)

Conforme as camadas se aprofundam o modelo começar a aprender características mais complexas usando informações de outros neurônios. Na figura abaixo podemos ver vários mapas de características em diferente camadas:

Figura 10 – Diferentes niveis de extração de caracteristicas

FONTE:ZEILER,(2013)https://arxiv.org/pdf/1311.2901.pdf

2.2.3 Camada totalmente conectada e Softmax

A camada totalmente conectada (FC) é a última camada da Rede Neural Convolucional. Como mostrado anteriormente na Figura 4, ela se comporta como uma rede neural *feedforward* e é a responsável pela interpretação dos mapas de características da última camada de pooling.

O mapa de características da última pooling sofre o processo de *flatten*. O vetor gerado por esse processo é alimentado à FC. Toda FC possui uma função não linear dentro dela, como podemos ver na Figura 11 do pseudocódigo de Eberman et al. (2018):

Figura 11 – Pseudo Código de uma Camada totalmente conectada

se camada anterior é de *pooling* ou de convolução
então
  redimensionar(x)
fim se
para todo neurônio j da camada totalmente conectada **faça**
  para toda entrada i da camada anterior **faça**
    $u_j = \sum_{i=1}^{n} x_i \cdot w_{i,j} + b_j$
    $y_j = f(u_j)$
  fim para
fim para

FONTE:Ebermam, E.; Krohling. R. Uma Introdução Compreensiva às Redes Neurais Convolucionais: Um Estudo de Caso para Reconhecimento de Caracteres Alfabéticos,(2018)

Porém a última camada totalmente conectada possui uma função de ativação diferente, chamada de Softmax. A quantidade de neurônios nessa última camada, onde a softmax é aplicada, é a mesma da quantidade de classes pré definidas para a rede. Nela cada neurônio equivale a uma classe, o valor externado por ele é de [0 a 1] e o maior deles é a previsão da rede neural. A softmax possui muitas variações mas o modelo mais comum é o mostrado na equação abaixo:

$$z_i = \frac{e^{z^i}}{\sum_{j=1}^{K} e^{z^j}} \quad \text{para} \quad i = 1, \ldots, K \quad \text{e} \quad z = (z_1, \ldots, z_K) \in R^K \tag{2}$$

A função exponencial é aplicada a cada elemento do vetor de entrada z e os valores resultantes são normalizados dividindo pelo somatório de todas as exponenciais. A normalização garante que os elementos do vetor de saída sm(z) somem 1. Por exemplo, se os resultados da FC forem $z_1 = 1, z_2 = 2, z_3 = 0.5$ *e* $z_4 = 1,5$ então os resultados da rede seriam:

$$\begin{aligned}
zi &= \frac{e^{z^i}}{e^1 + e^2 + e^{0.5} + e^{1.5}} \\
zi &= \frac{e^{z^i}}{2.71 + 7.38 + 1.64 + 4.48} \\
zi &= \frac{e^{z^i}}{16.21} \\
z1 &= \frac{e^{z^1}}{16.21} = \frac{e^1}{16.21} = 0.167 \\
z2 &= \frac{e^{z^2}}{16.21} = \frac{e^2}{16.21} = 0.455 \\
z3 &= \frac{e^{z^{0.5}}}{16.21} = \frac{e^{0.5}}{16.21} = 0.101 \\
z4 &= \frac{e^{z^{1.5}}}{16.21} = \frac{e^{1.5}}{16.21} = 0.276
\end{aligned} \tag{3}$$

2.3 Treinamento de uma RNC

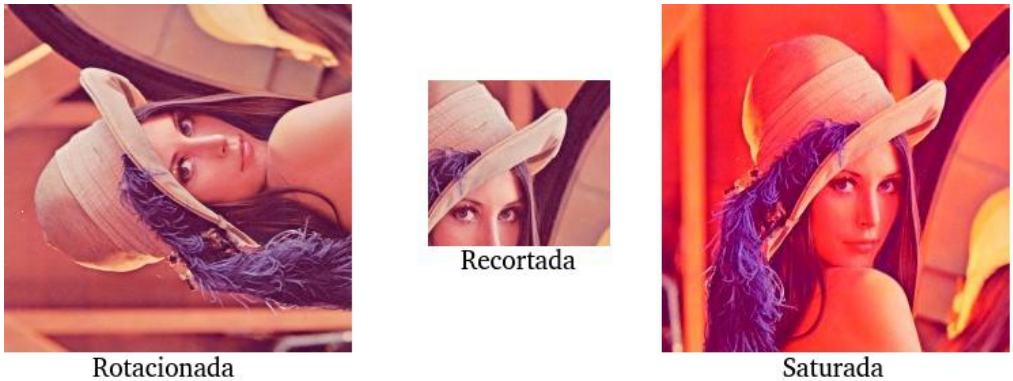
O processo de aprimorar as camadas de uma rede neural convolucional é chamado de treinamento. Para realizar o treinamento, é preciso ter o conhecimento de algumas ferramentas e conceitos como: escolha de filtros, pesos nas camadas FC, aplicação de algoritimos, arquiteturas e entre outros. O objetivo de um treinamento é reduzir o erro nas suas previsões, no caso, reduzir o erro entre os dados de saída e as classes de treinamento da rede.

Antes do treinamento, é necessário preparar os dados de entrada e seus respectivos rótulos. Isso pode incluir tarefas como carregar os dados, pré-processamento, como redimensionamento e normalização, e divisão em conjuntos de treinamento, validação e teste. Nesse capítulo, serão abordados alguns métodos e ferramentas que são utilizados na preparação e execução do treinamento de uma RNC.

2.3.1 Data Augmentation (Aumento de Dados)

O uso de diversas transformações nas imagens para aumentar a variedade de um dataset é uma habilidade muito útil, pois revoga a ideia de que é necessário um grande banco de dados para treinar um modelo de aprendizagem de máquina. Como mencionado em "Deep Learning for Coders with Fastai and PyTorch: AI Applications Without a PhD"(Howard e Gugger, 2020), também é muito utilizado em aplicações onde a capacidade de obter dados ou imagens é muito difícil. As transformações são variadas, por exemplo: rotação, translação, aumento ou redução de brilho, recorte da imagem, esmagamento da imagem tanto na horizontal quanto na vertical, zoom, etc. A figura abaixo mostra exemplos de Data Augmentation:

Figura 12 – Exemplo de Augmentation



FONTE:Do Autor(2024)

2.3.2 Hiperparâmetros

Os hiperparâmetros são parâmetros que não são aprendidos durante o treinamento do modelo, mas precisam ser definidos antes do treinamento começar. Eles afetam diretamente o processo de aprendizado da rede, influenciando fatores como a velocidade e a qualidade do treinamento, bem como a estrutura e comportamento da rede neural.

2.3.2.1 Batch

Batch é o termo utilizado para definir o conjunto de exemplos de treinamento que serão utilizados em uma iteração. O Batch Size é o números de exemplos e possui três opções:

- Batch mode: onde o tamanho do lote é igual ao conjunto de dados total, logo os valores de iteração e épocas são iguais.
- Mini-batch mode: onde o tamanho do lote é maior que um, mas menor que o tamanho total do conjunto de dados.
- Stochastic mode: onde o tamanho do lote é igual a um. Portanto, o gradiente e os parâmetros da rede neural são atualizados após cada amostra.

Segundo a equipe DataScienceAcademy (2022), um Batch size grande causa uma degradação significativa na qualidade do modelo, que é medida pela sua baixa capacidade de generalização. Por outro lado, batch sizes menores demostram empiricamente uma convergência mais rápida para boas soluções. Isso é intuitivamente explicado pelo fato de que tamanhos de lote menores permitem que o modelo inicie o aprendizado antes de ver todos os dados.

2.3.2.2 Taxa de aprendizado (learning rate)

A taxa de aprendizado é um hiperparâmetro fundamental em algoritmos de RNC. Essa taxa controla o tamanho dos passos que um algoritmo de otimização, como o Gradiente Descendente Estocástico, dá ao ajustar os pesos da rede durante o treinamento. O tamanho do passo refere-se à magnitude da mudança nos valores dos pesos durante cada atualização.

Segundo Jason Brownlee(2021), a taxa de aprendizado controla o quão rapidamente o modelo se adapta. Pequenas taxas de aprendizado requerem mais épocas de treinamento, dada a menor taxa de atualização nos pesos a cada época, enquanto taxas de aprendizado maiores resultam em mudanças rápidas e requerem menos épocas de treinamento.

Encontrar a taxa de aprendizado adequada é crucial para garantir que o modelo seja treinado de maneira eficaz e eficiente. Uma taxa de aprendizado muito grande pode fazer com que o modelo dirija-se muito rapidamente para uma solução subótima, enquanto uma taxa de aprendizado muito pequena pode fazer com que o processo fique preso.

Utilizando o Gradiente Descendente Estocástico com valor de 2, uma taxa de 0.1, tendo o valor do novo peso como Np e o antigo valor como Ap, podemos ver um exemplo de como a taxa de aprendizado afeta os pesos do modelo na equação 4.

$$\begin{aligned} &\text{Novo\_Peso} = \text{Antigo\_Peso} - (\text{learning\_rate} \times \text{gradiente}) \\ &\text{Np} = \text{Ap} - (0.1 \times 2) \\ &\text{Np} = \text{Ap} + 0.2 \end{aligned} \tag{4}$$

2.3.3 Função de perda

Para monitorar as saídas das redes neurais, é utilizada a função de perda, que mensura a divergência entre o valor de saída e o valor esperado para a entrada. O objetivo é que o valor de perda diminua à medida que o número de épocas aumenta, já que a perda representa a discrepância entre o valor esperado e o valor previsto.

Buscando minimizar essa discrepância durante o treinamento do modelo, os parâmetros do modelo são ajustados de forma iterativa. Segundo Karpathy (2020), a função de perda serve como guia para o processo de otimização, levando a uma solução que minimize o erro nos dados e balanceie os pesos dos dados de entrada durante o treinamento.

2.3.4 Otimização (Gradiente descendente)

O Gradiente descendente é um algoritmo com o objetivo de encontrar um mínimo em uma função arbitrária. É amplamente utilizado em otimização de *deep learning* para minimizar a função de perda durante o treinamento. Seu objetivo é encontrar os parâmetros do modelo que resultam no menor valor da função de perda, ou seja, na melhor performance do modelo.

Este algoritmo ajusta os parâmetros do modelo na direção oposta ao gradiente da função de perda a cada iteração do treinamento. Por exemplo, se a função de perda estiver diminuindo em uma direção, os parâmetros do modelo serão atualizados nessa direção para reduzir ainda mais a perda. Por outro lado, se a função de perda estiver aumentando, os parâmetros serão atualizados na direção oposta para tentar minimizar a perda. Este algoritmo utiliza o hiperparâmetro *learning rate* para fazer a otimização da função de perda.

2.3.5 Overfitting (Sobreajuste)

O *overfitting* é um dos principais desafios do aprendizado de máquina, uma vez que um modelo com *overfitting* não pode ser generalizado. O sobreajuste ocorre quando o modelo memoriza os detalhes dos dados de treino. Por isso, uma série de técnicas foi proposta para ajudar a mitigá-lo (LEMOS, 2021). Ações que podem evitar o *overfitting* incluem: aumento de dados no treinamento e regularização com *Dropout*.

O *Dropout* é uma das técnicas de regularização utilizadas para evitar o sobreajuste. Esta técnica desativa aleatoriamente um grupo de neurônios da camada oculta. Utilizando os resultados da rede neural modificada, é utilizado um mini-lote de exemplos para atualizar os pesos e vieses apropriados. Em seguida, os neurônios removidos são restaurados e o processo é repetido com outro conjunto aleatório de neurônios. Logo, é como se estivéssemos treinando redes neurais diferentes.

> Esta técnica reduz co-adaptações complexas de neurônios, já que um neurônio não pode confiar na presença de outros neurônios em particular. É, portanto, forçado a aprenda recursos mais robustos que são úteis em conjunto com muitos subconjuntos aleatórios diferentes dos outros neurônios.(DataScienceAcademy ,2022,cap.23)

2.3.6 Transfer Learning

Transfer Learning corresponde a utilizar um modelo pré-treinado em uma tarefa conhecida, por exemplo, ImageNet. Para a criação de modelos pré-treinados são utilizados datasets com grande volume de imagens e com uma grande diversidade de classes. De acordo com Fastai (2023), a ideia é utilizar as características aprendidas, no pré-treinamento, e utilizar esse conhecimento para acelerar e melhorar o seu modelo. É visível a eficiência no treinamento, principalmente se é utilizado um dataset pequeno.

Na visão computacional, por exemplo, as RNC's geralmente tentam detectar bordas e formas nas camadas inciais e intermediarias. No transfer learning, essas camadas são usadas e apenas treinamos novamente as últimas camadas, onde são aprendidas as caracteristicas mais especificas do objeto.

Segundo Donges (2022), o *transfer learning* é a reutilização de um modelo pré-treinado em um novo problema. O autor ainda define o transfer learning como uma metodologia de projeto, que é comumente utilizado em RNAs, pois facilita o treinamento de novos modelos utilizando poucos dados de entrada.

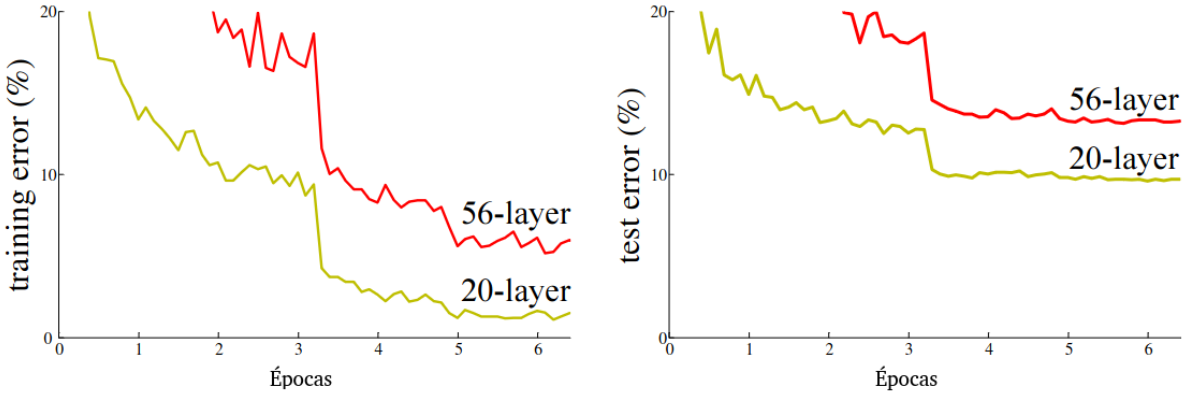
2.3.7 ResNet

Segundo os atuores HE et. al. (2016), quando as redes mais profundas começam a convergir, atingir valores de saída desejaveis, a degradação aparece: com o aumento da profundidade da rede, a precisão diminui e então decai rapidamente.

Inesperadamente, essa degradação não é causada por overfitting, e adicionar mais camadas ao modelo leva a um erro de treinamento mais alto.

Esse problema pode ser observado na figura 13, onde a esquerda temos o erro de treinamento e a direita o erro de teste. Nesse teste foram utilizadas são rede simples, sem blocos residuais, com 56 e 20 camadas de profundidade. Tento isso em consideração é possível observar que a rede mais profunda, devido a degradação, teve maiores valores de erro.

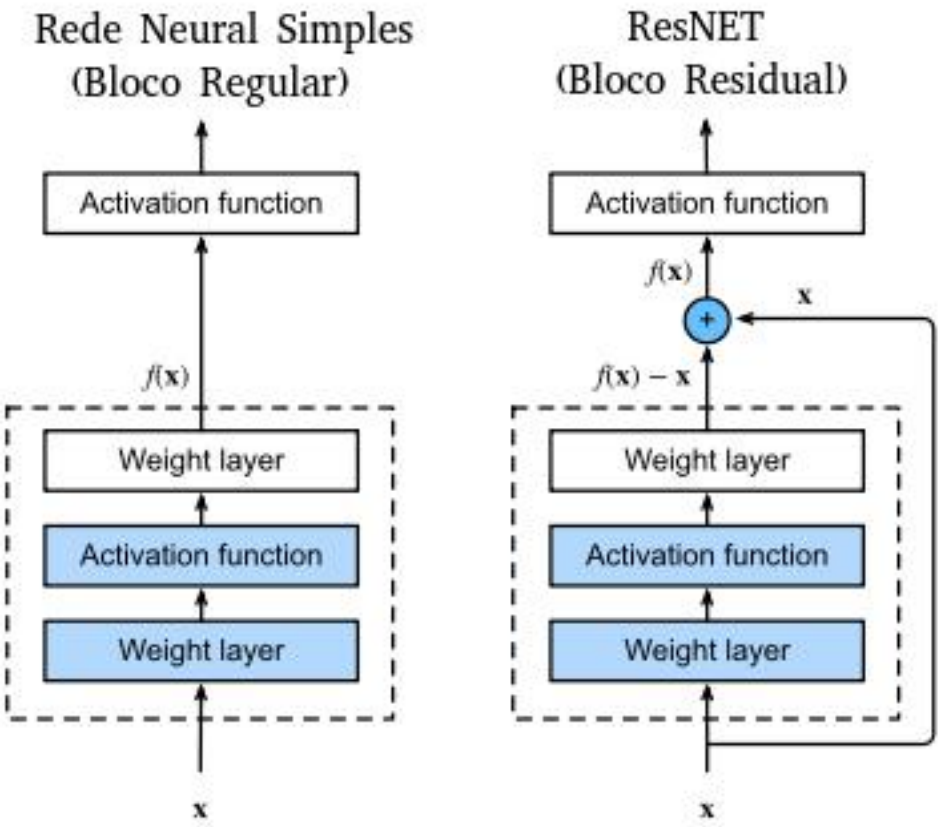
Figura 13 – Degradação occorendo em rede neural simples



FONTE: Adptado de HE et. al.(2016)

Podemos usar a figura 14 para ter melhor entendimento de como a resnet contornou o problema de degradação. Tendo o valor de entrada como x, segundo Aston et. al. (2021), podemos assumir que a transformação ideal, tranformação aplicada à entrada para obter a saída desejada, seja F(x), que será usado como entrada para a função de ativação no topo.

Figura 14 – Bloco regular e bloco residual



FONTE: Adaptado Aston et. al. (2021)

Ainda segundo os autores Aston et. al. (2021), a ideia principal por trás dos blocos residuais é que a rede neural pode aprender a ajustar apenas as diferenças entre a entrada x e a saída desejada F(x), em vez de ter que aprender toda a transformação F(x) diretamenteIsso é feito através da conexão residual ou conexão de atalho, representada na figura 14 como a linha que conecta x a F(x). Essa conexão também funciona como uma regularização pois otimiza e estabiliza o treinamento da rede e permite que arquiteturas possuam grandes profundidades.

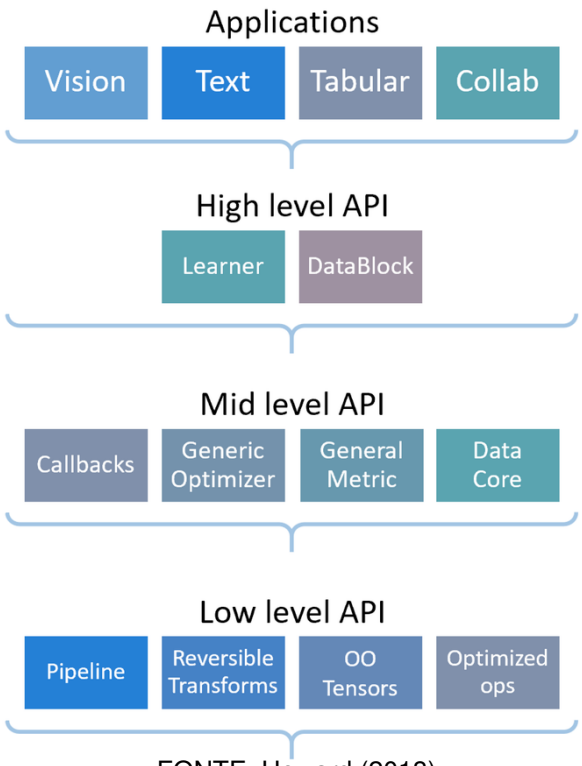
2.3.8 FastAI

PyTorch é um framework de aprendizado profundo de código aberto, conhecido por sua flexibilidade e facilidade de uso. Isso é, em parte, possibilitado por sua compatibilidade com a popular linguagem de programação de alto nível Python, preferida por desenvolvedores de aprendizado de máquina e cientistas de dados. Ele se destaca pelo excelente suporte a GPUs.

Fastai é uma biblioteca de *Deep Learning*, desenvolvida em cima do *PyTorch*, que fornece uma forma mais simples de trabalhar com redes neurais e possui ferramentas de baixo e alto nível. Ela é organizada em torno de dois objetivos principais de design: ser acessível e rapidamente produtiva, ao mesmo tempo que é profundamente

configurável. Podemos ver na figura abaixo as aplicações que esse biblioteca consegue trabalhar e os niveis das API's e suas ferramentas.

Figura 15 – Hierarquia dos niveis de API



FONTE: Howard (2018)

Essa biblioteca possui muitas funções importantes que serão usadas mais adiante no desenvolvimento deste trabalho. O *Datablock* é uma delas, sendo uma forma de facilitar o treinamento de um modelo, ja que ele é um esquemático que orgazina de forma clara os parâmetros de treinamento.

No *Datablock*, é necessário entender e definir alguns métodos que são de suma importância para o treinamento, a saber: o caminho das imagens que serão usadas no treinamento, a formatação do banco de dados, o divisor de dados, e as transformações.

O divisor de dados, chamado de *splitter* possui dois parâmetros, que são: *valid_pct* e *seed*. O primeiro parâmetro especifica a porcentagem dos dados que serão alocados para o conjunto de validação, exemplo: um *valid_pct* de valor 0.2 terá 20% dos dados como dados de validação, os outros 80% serão usado durante o treinamento. Já o segundo parâmetro é usado para controlar a aleatoriedade na divisão dos dados. A *seed* é um valor numérico que permite reproduzir os mesmos resultados aleatórios em diferentes execuções do código, sendo útil para reproduzir resultados e garantir a consistência dos experimentos.

Na função *Datablock*, ainda é possível aplicar aumento de dados através de transformações, como: espelhamento, recorte, zoom, entre outras, como visto em 2.3.1.

Nesse método também é possível fazer a transformação de *resize*, essa transformação é muito importante pois permite tornar a imagem quadrada e reduzir o seu tamanho, sendo essencial para o treinamento pois auxilia na aplicação dos filtros e reduz o processamento da rede neural.

Ao final da função *Datablock* é possivel chamar o *DataLoader*, para carregar os dados, de acordo com os parâmetros que foram definidos, em grupos, chamados de *batch size*, como visto em 2.3.2.1.

Para o treinamento da RNC foi utilizada função *vision_learner*. Essa função ajuda o usuário, escolhendo automaticamente um modelo pré-treinado da arquitetura escolhida. Utilizando as informações anteriormente explicadas a função é capaz de realizar o treinamento do modelo e devolver ao usuário um gŕafico do desempenho do mesmo.

Com o resultado, da *vision_learner*, pode ser usado o método Learner, para exportar e salvar modelos, sendo possível reutilizar o modelo para re-treino.

2.4 Métricas

Para acompanhar e validar o desempenho de um modelo e/ou treinamento existem várias técnicas que podem ser utilizadas em aprendizado de máquinas. Tais técnicas podem ser chamadas de métricas e elas quantificam e apresentam visualmente o desempenho e resultados do modelo.

Os termos usados para calcular as métricas são: verdadeiros positivos(VP), verdadeiros negativos(VN), falsos positivos(FP) e falsos negativos(FN). Esses termos representam previsões que estão corretas ou erradas, logo, um falso negativos significa uma predição errada de uma classe verdadeira positiva. A lista abaixo apresenta as métricas discutidas.

- Acurácia: Essa métrica representa a proporção de predições corretas para o total de predições, como podemos ver na equação abaixo. A acurácia é a quantificação da performace do modelo e é representada pela equação 5:

$$\text{Acurácia} = \frac{\text{VP} + \text{VN}}{\text{Total de Previsões}} \tag{5}$$

- *Recall* : Essa métrica mede a capacidade do modelo de classificar corretamente verdadeiros positivos. Um valor alto de recall indica que o modelo previu a maioria das imagens como True OKenquanto um valor baixo indica a presença de falsos negativos. Isso fica mais claro quando se olha para a equação 6:

$$\text{Recall} = \frac{\text{VP}}{\text{VP} + \text{FN}} \tag{6}$$

- Precisão: Essa métrica mede a capacidade do modelo de evitar classificações de falsos positivos. A precisão mede a precisão das previsões de verdadeiros positivos, como podemos ver na equação 7:

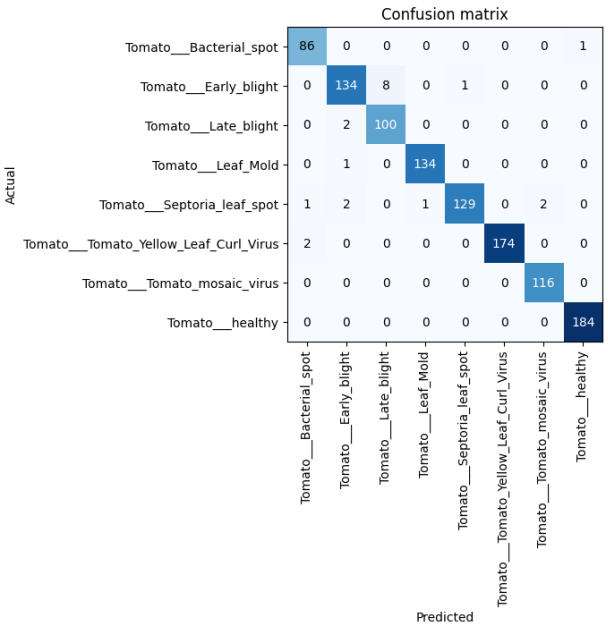
$$\text{Precisão} = \frac{\text{VP}}{\text{VP} + \text{FP}} \tag{7}$$

- *F1-score*: É a media harmônica entre *recall* e precisão. O *F1-Score* varia de 0 a 1, onde 1 indica um modelo perfeito e 0 indica um modelo que não conseguiu fazer previsões corretas. Essa métrica mede a performace do modelo, como mostra a equação 8:

$$\text{F1-Score} = 2 \text{ x } \frac{\text{Precisão} x \textit{Recall}}{\text{Precisão} + \textit{Recall}} \tag{8}$$

- Matriz confusão: é utilizada para medir o desempenho do modelo e diferente das outras métricas ela é fornecida em formato de tabela. A matriz de confusão compara as previsões com as respostas reais. Ela é uma ferramenta muito util pois permite facilmente visualizar aonde estão os problemas do modelo, e também fornece os dados para fazer os calculos das outras métricas ja apresentadas. Por exemplo, na figura 16, podemos ver, na primeira linha e coluna, que a classe *bactrial spot* possui um total de 86 previsões corretas, verdadeiros oks, e 1 falso negativo que foi previsto como *healthy*.

Figura 16 – Matriz confusão de treinamento Resnet 34



FONTE: Do Autor(2024)

- *train_loss* e *valid_loss*: Essas duas métricas são usadas durante o treinamento para acompanhar o desempenho do modelo. *train_loss* é o calculo da perda dos dados de treinamento e o quão bem eles se ajustaram ao novos pesos. *valid_loss* é o calculo da perda para os dados de validação, esses são usados para monitorar o desempenho do modelo em imagens que ele nunca viu.
- *error_rate*: É métrica que análisa o desempenho do modelo atráves do número de previsões incorretas, como é apresentado na equação 9.

$$error\_rate = \frac{FP + FN}{\text{Total de Previsões}} \tag{9}$$

2.5 Trabalhos correlatos

A visão computacional tornou-se muito mais poderosa e popular com a constante evolução computacional, e do aprendizado de máquina, ganhando viabilidade de implementação em diversas áreas de atuação.(DAL BO e CORSO, 2023).

Brignoli (2021) desenvolveu um projeto com RNC para a classificação de 6 classes, que representavam ambientes residenciais, juntamente com a implementação do modelo RNC em um servidor API. A solução foi desenvolvida utilizando a biblioteca *FASTAI* e comparando os resultados de 3 modelos treinados com 3 diferentes arquiteturas *Resnet, Resnet* 18, 34 e 50. O autor conseguiu provar a eficacia do seu modelo alcançando uma acurácia de 96% com apenas 10 épocas de treinamento.

Lemos(2021) apresentou uma solução de inspeção automática de ovos de galinha através da comparação de 3 diferentes algoritmos de visão: técnicas classicas, aprendizado profundo com RNC e segmentação semântica com RNC . O projeto buscou classificar 4 diferentes classes de ovos: normais, sujos, geometricamente anormais e fissurados. Os 3 métodos obtiveram sucesso de detecção alcançando acurácias acima de 80%. Lemos tamém utilizou da biblioteca *FASTAI* e das*Resnet's* 18, 24 e 50.

No trabalho dos autores AL-HIARY et al.(2011) foi proposto um sistema de detecção e classificação automática de doenças em plantas usando métodos de processamento de imagens e aprendizado de máquina. A solução melhora abordagens anteriores com maior velocidade e precisão,aplicando um algoritmo *K-means* para segmentação. A técnica também remove pixels sem informações relevantes e usa redes neurais artificiais (RNA) para classificar doenças com precisão entre 83% e 94%.

No artigo de NAG et al.(2023) foi desenvolvido um sistema de identificação de doenças em folhas de tomate baseado em redes neurais convolucionais (CNN) aplicadas em um aplicativo móvel. Utilizando técnicas de aprendizado por transferência, modelos como AlexNet, ResNet-50, VGG19 e DenseNet-121 foram ajustados para classificar imagens de folhas de tomate do conjunto PlantVillage. O estudo se destaca por

alcançar mais de 95% de acurácia, com DenseNet-121 apresentando um desempenho de 99,85%.

3 METODOLOGIA

Nesse capítulo são apresentados a metodologia e o desenvovilmento deste trabalho.

3.1 Discussão Sobre os Requisitos do Sistema

Este capítulo apresenta os requisitos funcionais e não funcionais do sistema. Esses requisitos são os objetivos e delimitações do projeto.

3.1.1 Requisitos funcionais

- O sistema de visão deverá classificar corretamente as doenças em folhas de tomate que são dividas em 8 classes. Essas estão apresentadas na tabela 1:

Classe	Imagem	Classe	Imagem
Bacterial Spot		Early Blight	
Healthy		Late Blight	
Septoria Spot		Mosaic Virus	
Leafy Mold		Yellow Curl Virus	

Tabela 1 – Classes a serem identificadas pelo sistema

3.1.2 Requisitos não funcionais

- O sistema de visão deverá ter a classificação em uma API, com comunicação ao aplicativo.

3.2 Descrição da solução

Com o objeito de cumprir os requisitos do projeto, foi selecionada o método de classificação de imagem com aprendizado profundo com redes neurais convolucionais. Esse método foi implementado utilizando a bilbioteca Fastai. A seleção dos dados foi feito no Kaggle, os treinamentos dos modelos foram feitos em um computador

utilizando uma GPU Nvidia GTX1650 de 4GB. Para começar o desenvolvimento do sistema foi criado um plano de ação, que se consistiu em:

- Obtenção e formatação do banco de dados;
- Treinamento e validação do modelo classificador;
- Criação de API com o modelo;
- Criação de aplicativo com requisição à API;

3.3 Treinamento do modelo

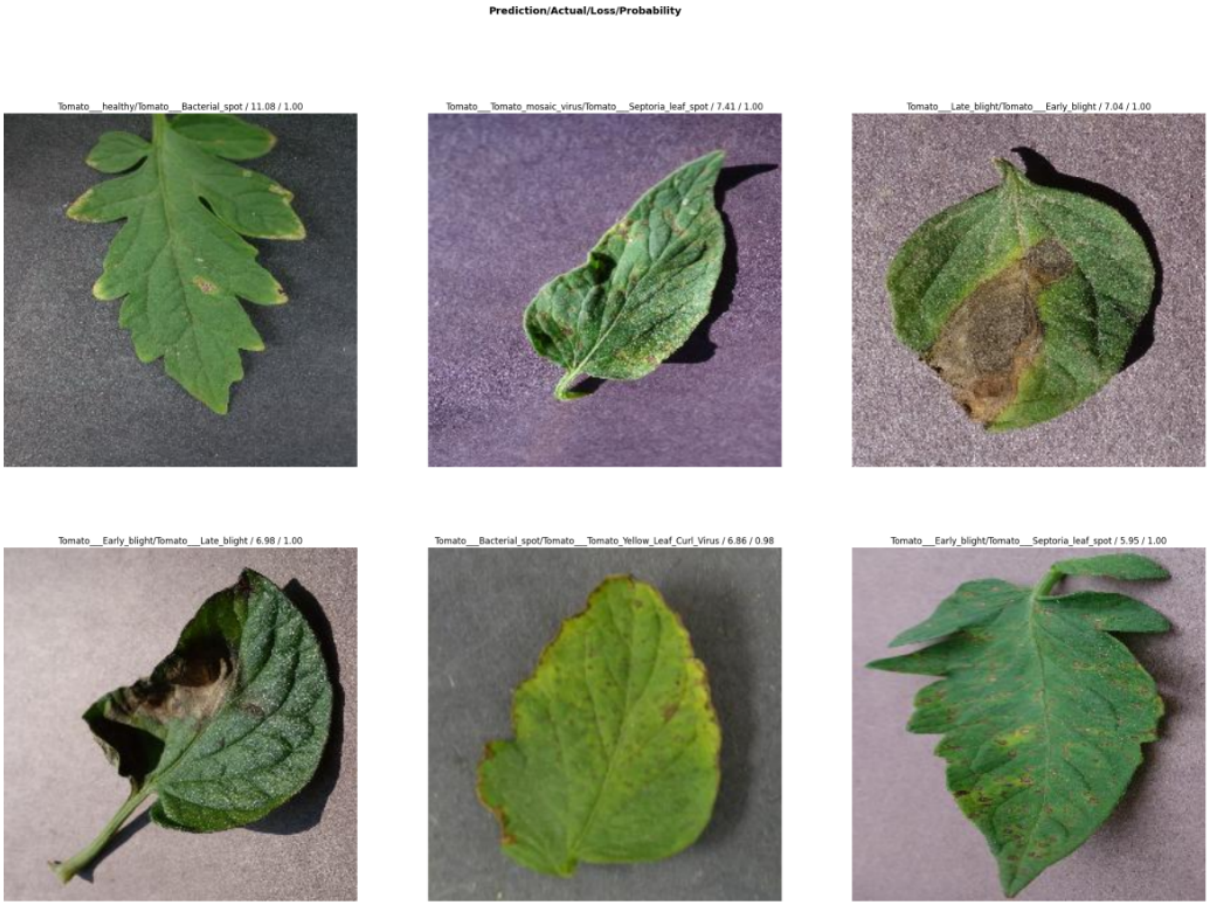
O próximo passo é o treinamento do modelo utilizando o banco de dados criado.

3.3.1 Obtenção e formatação do banco de dados

Para esta aplicação o banco de dados foi procurado na internet em site como: *Kaggle*, *Roboflow* e *Data.mendeley*. O objeto de estudo escolhido foi a folha do tomate e assim foi montado um banco com as 8 classes definidas nos requisitos do projeto. Após o carregamento do banco foi necessário fazer a sua formatação.

Como o banco de dados foi obtido da internet, foi necessário verificar todas as classes para identificar possíveis imagens com anotações incorretas, ou seja, imagens que podem estar na classe errada. Esse processo é crucial, pois a falta de cuidado pode resultar em modelos com desempenhos insatisfatórios. Um método eficaz para a verificação de grandes bancos de dados é treinar modelos de forma gradual e adicionar imagens da classe ao longo do tempo. Dessa maneira, é possível usar esses modelos para classificar as imagens e, com base nos resultados, avaliar as maiores perdas, exemplo na figura 17, do treinamento e identificar as anotações incorretas.

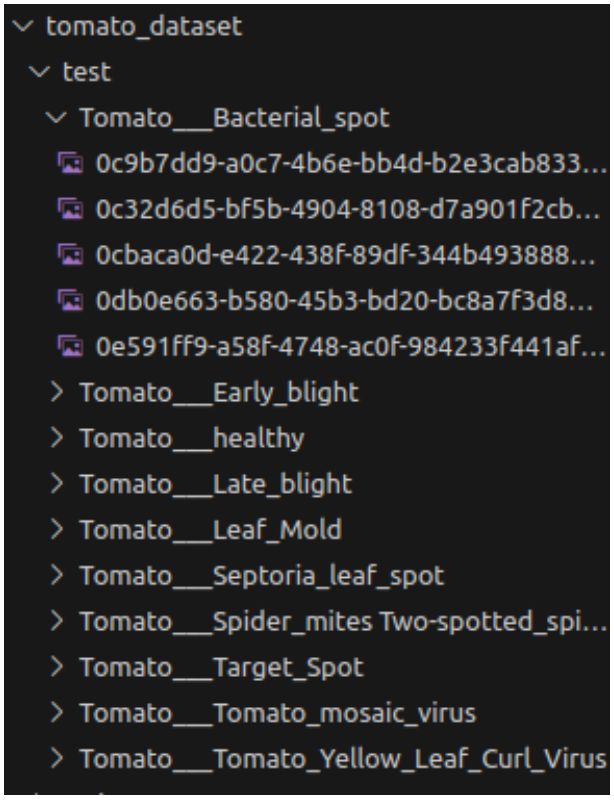
Figura 17 – Exemplo de *top losses*



FONTE:Do Autor(2024)

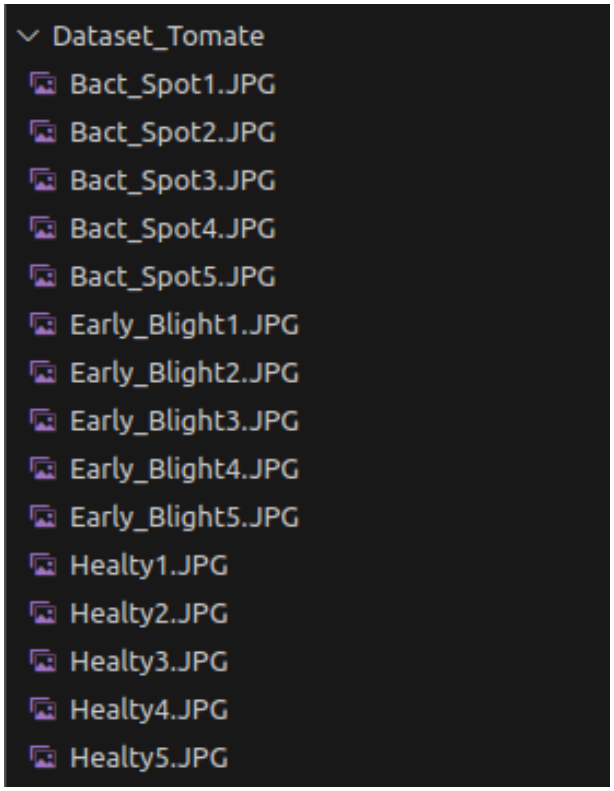
A formatação de um banco de dados pode ser feita de muitas maneiras, como podemos ver na figura 18 o banco possui a seguinte organização: Diretório do dataset -> Diretórios de treinamento ou validação -> Diretórios das classes -> Fotos referentes às classes. Já na figura 19 a seguinte forma é tomada: Diretório do dataset -> Fotos referentes às classes.

Figura 18 – Exemplo de formatação do banco de dados 1



FONTE:Do Autor(2024)

Figura 19 – Exemplo de formatação do banco de dados 2



FONTE:Do Autor(2024)

A formatação da figura 19 foi o escolhida para este projeto, apenas fazendo alteração no nome das fotos para: classe(X).jpg, sendo x a posição da foto no diretório. O dataset usado nesse projeto consta na tabela 2. Além do parâmetro *valid_pct*, como falado em 3.3.2, foi criado um dataset apenas para validação, como consta na tabela 3. O banco de treinamento consiste num total de 3445 fotos e o de validação 14777.

Classe	Quantidade	Classe	Quantidade
Bacterial Spot	426	Early Blight	433
Healthy	481	Late Blight	261
Septoria Spot	436	Mosaic Virus	448
Leafy Mold	470	Yellow Curl Virus	490

Tabela 2 – Banco de dados de treinamento

Classe	Quantidade	Classe	Quantidade
Bacterial Spot	1702	Early Blight	1920
Healthy	1926	Late Blight	1851
Septoria Spot	1745	Mosaic Virus	1790
Leafy Mold	1882	Yellow Curl Virus	1961

Tabela 3 – Banco de dados de validação

3.3.2 DataBlock

No datablock, foi definida a aplicação de um random splitter, utilizado para dividir os dados em: 40% para validação e 60% para treinamento. Note que o datablock realiza automaticamente essa separação, eliminando a necessidade de fazê-lo manualmente. No entanto, é uma boa prática reservar 40% das imagens para validação, a fim de avaliar o modelo após o término do treinamento. Neste projeto, foi utilizado um banco de validação significativamente maior.

Além disso, foram realizadas as seguintes transformações: redimensionamento das imagens para 233x233 pixels, aplicação de uma técnica de *data augmentation* de esmagamento (squish) e definição de um seed igual a 42. O seed é essencial para a comparação entre diferentes arquiteturas, pois define a divisão dos dados pelo splitter. Dessa forma, a divisão dos dados será sempre a mesma em cada treinamento.

3.3.3 Vision_learner

A função *vision_learner*, em sua forma padrão, ajuda na aquisição automática de um modelo pré treinado, que se enxaixe no tipo de dado sendo usado. Logo, essa função utiliza de *transfer learning* e ela requer três parametros para seu funcionamento: o dls(Datablock), a arquitetura e as métricas.

Para este trabalho, foram escolhidas as arquiteturas ResNet18, ResNet34 e ResNet50, e as métricas avaliadas foram a acurácia e a *error rate*. Além disso, foi

necessário definir o número de *epochs*, que representa um ciclo de aprendizagem completo. Um ciclo é considerado concluído quando todas as imagens do dataset foram processadas pelo modelo pelo menos uma vez.

Os valores de *epochs* e *splitter* foram ajustados experimentalmente após vários treinamentos. Isso foi possível devido ao grande volume de imagens no dataset, que permitiu observar, após alguns ciclos, uma alta acurácia e um baixo *error rate*.

No treinamento foi utilizada a técnica de *fine tuning*, que é uma das muitas técnicas de *transfer learning*, como discutido em 2.3.7 . Essa técnica tem como parâmetro o número de *epoch* e, como é visto no capítulo 2, é usado para ajustar os pesos dos dados de entrada modelo. No caso da *FastAi*, essa técnica é aplicada automaticamento pela função *vision_learner*, como dito anteriormente.

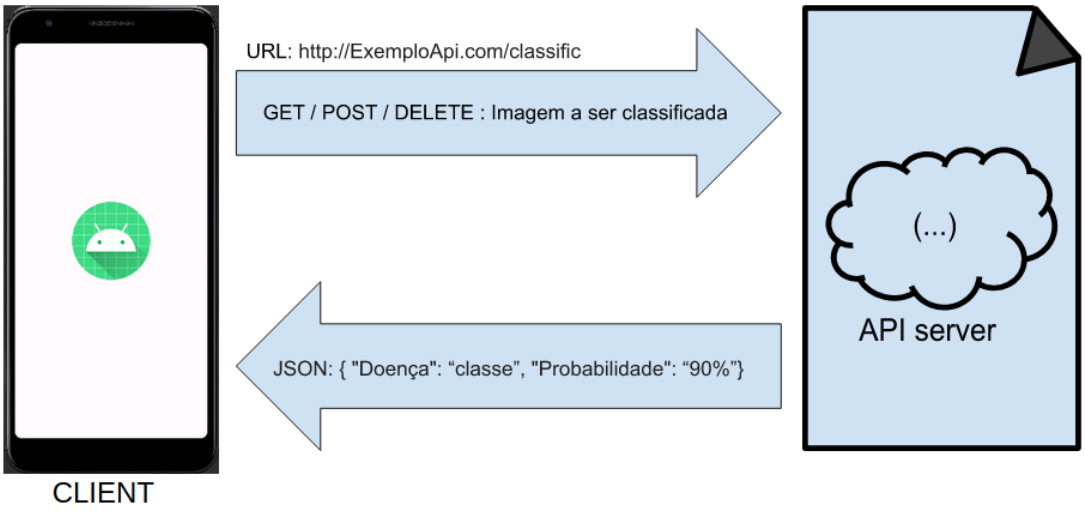
Após a formulação do Datablock e do *vision_learner* com seus respectivos parâmetros, é possível treinar o modelo para classificar doenças em folhas de tomate. A função *Learner.export()* permite salvar o modelo treinado e utilizá-lo posteriormente. Esta função armazena todas as informações essenciais, incluindo: a arquitetura do modelo, as configurações de treinamento, o estado do otimizador e as transformações aplicadas.

3.4 Desenvolvimento da API

Tendo em vista os requisitos do trabalho, foi necessário a criação de uma API para receber requisições e enviar os resultados do modelo classificador. A API foi feita em Python utilizando o framework FastAPI para criar um servidor de API.

API é uma integração com um conjunto de rotinas e padrões que facilitam a comunicação e troca de informações entre sistemas, no caso desse trabalho será entre APP <-> API. Como é possivel observar na imagem abaixo:

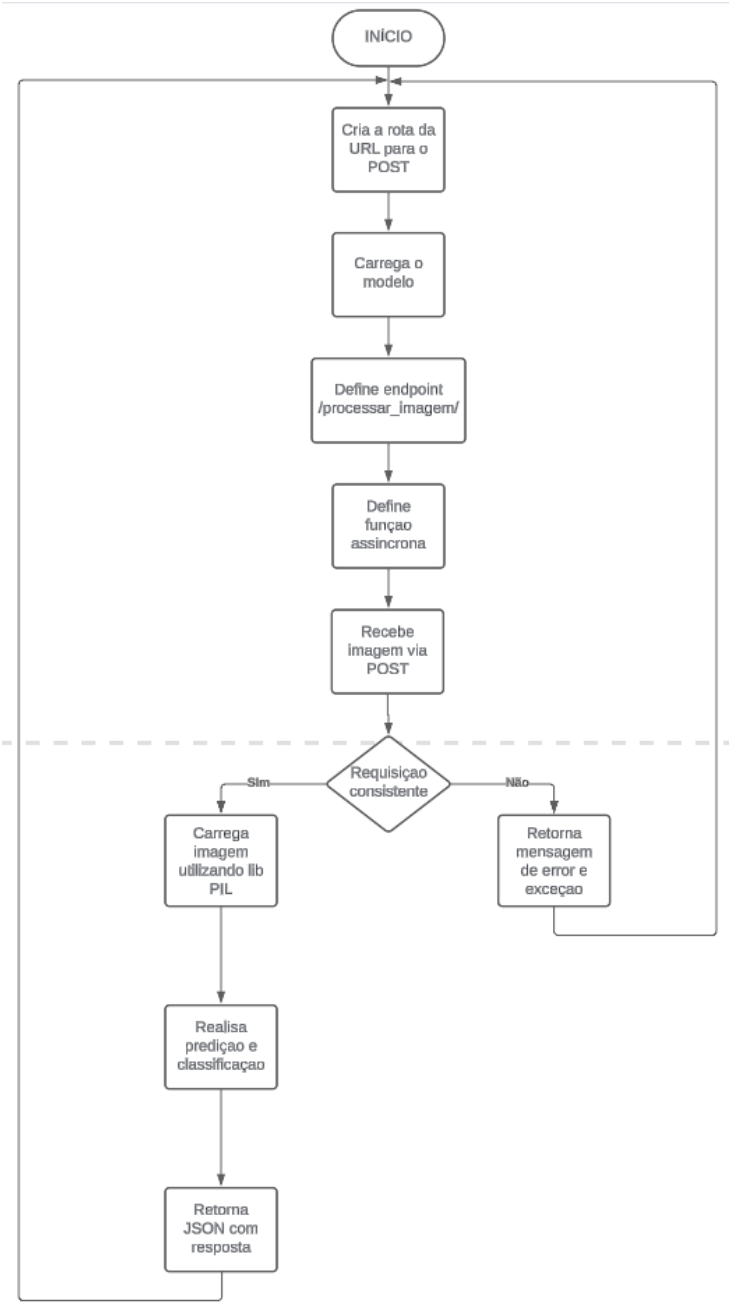
Figura 20 – Exemplo de funcionamento de uma API



FONTE:Do Autor(2024)

Através desse framework, uma rota foi criada com um endpoint /processar_imagem/. Utilizando o método POST, é possível enviar um arquivo, que será a imagem enviada pelo aplicativo. Se a requisição for bem-sucedida, o arquivo será passado pelo modelo classificador. Ao final do processo, as informações de classe e probabilidade, que são a resposta do modelo, são formatadas em JSON e enviadas para o aplicativo. A figura 21 demonstra o funcionamento da API.

Figura 21 – Fluxograma do funcionamento da API



FONTE:Do Autor(2024)

Na API também é feita a conversão da imagem para o modo de cor RGB, para isso utilizou-se a biblioteca *Pillow* e a função *.convert('RGB')*, isso é necessário para garantir que as imagens passadas no modelo estejam no mesmo formato que as utilizadas no treinamentos.

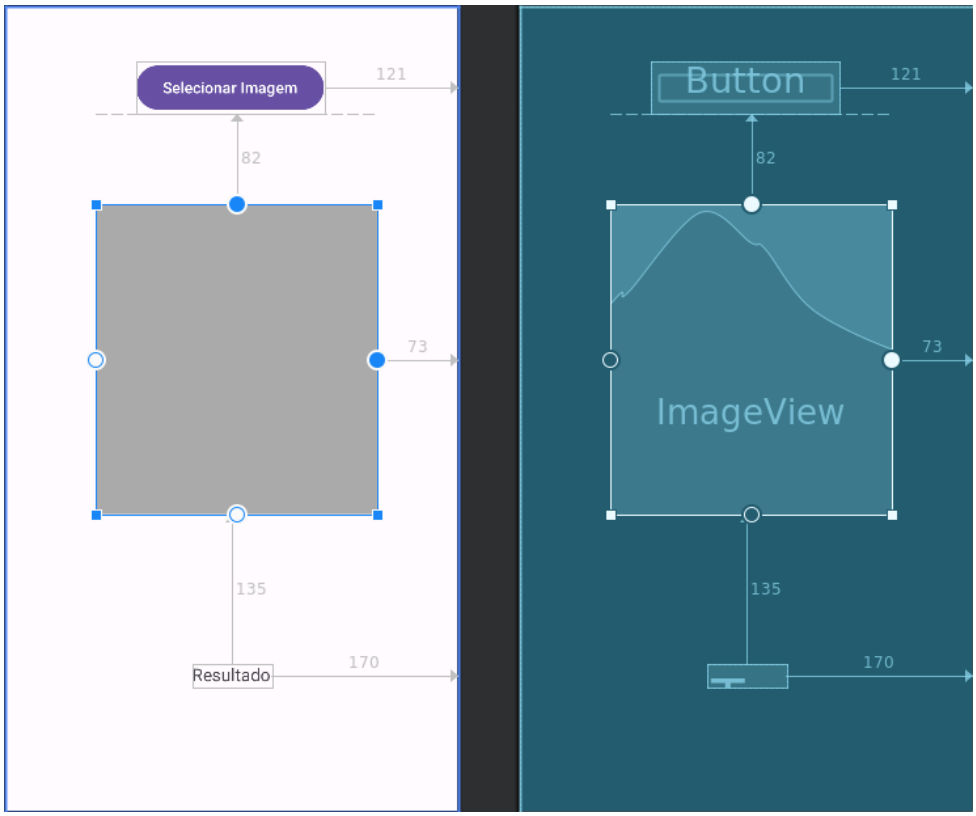
Para facilitar a leitura dos resultados, a API retorna apenas o resultado mais provável, com o nome da classe e o seu percentual. E para evitar predições erradas foi colocado um limite de 60% de acurácia, logo, se o resultado da classificação for menor que esse valor, a API retorna um valor de acurácia igual a 0 e de classe igual a "Classe

não registrada".

3.5 Desenvolvimento do APP

Após a criação da API foi necessário desenvolver um aplicativo, assim cumprindo com o último passo do plano de ação. Para isso foi optado por utilizar o software Android Studio da JetBrains. Trata-se de um ambiente de desenvolvimento integrado projetado especificamente para criar aplicativos Android. Essa IDE (Integrated Development Environment) pode ser essencialmente dividida em duas partes principais: a MainActivity, responsável pela lógica de execução do programa, e a MainActivity.xml, que representa a interface gráfica do aplicativo como mostra a Figura 22.

Figura 22 – Interface gráfica Android Studio



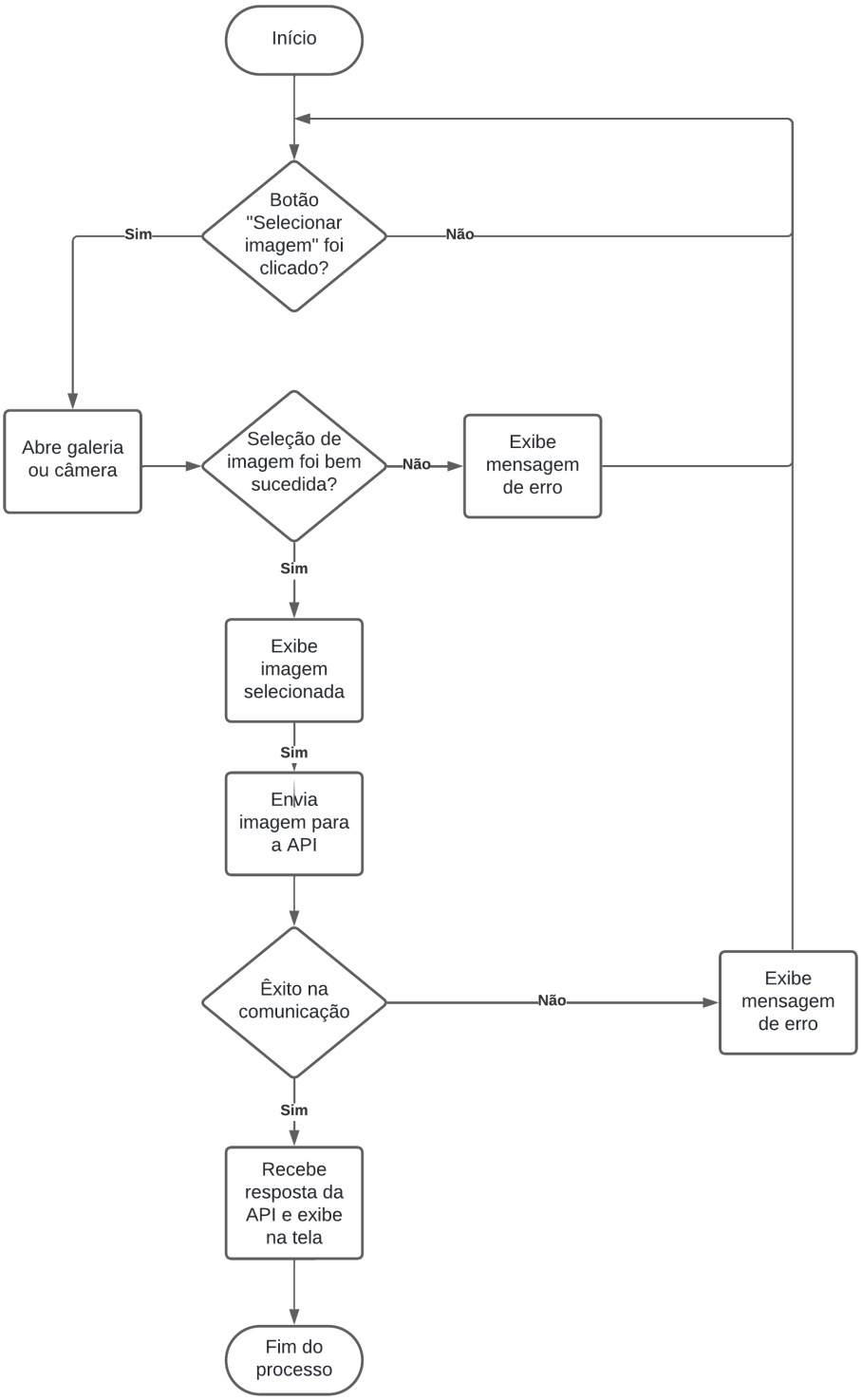
FONTE:Do Autor(2024)

O código foi escrito de forma objetiva e com o intuito de sanar os requisitos do projeto. O aplicativo precisa acessar a galeria ou camera do celular, carregar uma imagem, enviar ela para a API e por fim receber os resultados e apresenta-los na tela.

Quando o botão "Selecionar Imagem"é clicado, abre-se a opção de escolha entre galeria e câmera do dispositivo para selecionar uma imagem. Após a seleção, são disponibilizadas as edições de zoom e recorte na imagem, que, por fim, será exibida na tela. Após a confirmação da escolha, a imagem é convertida em bytes e enviada para a API por meio de uma solicitação HTTP. A resposta da API é, então, exibida em

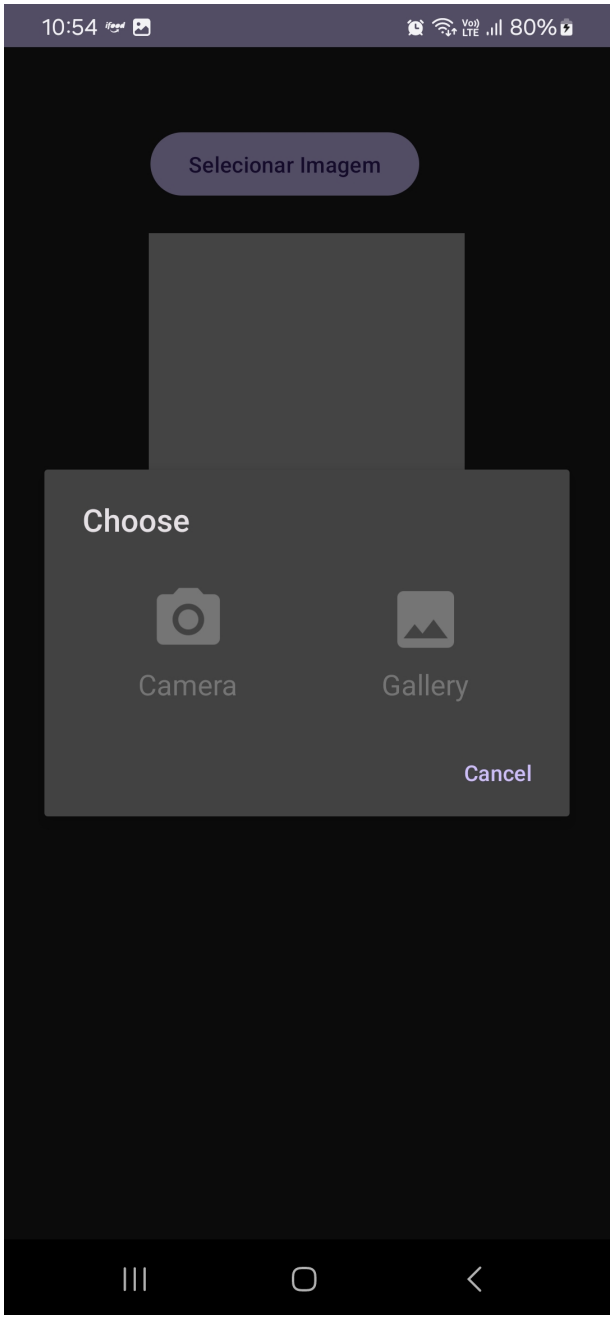
um campo de texto na interface do usuário. Esse fluxo é demonstrado na figura 23. E a figura 24 mostra o passo após o botão "Selecionar Imagem".

Figura 23 – Fluxograma do funcionamento do aplicativo



FONTE:Do Autor(2024)

Figura 24 – Tela de escolha de camera ou galeria para selecionar a imagem



FONTE:Do Autor(2024)

Para a comunicação com a API, foram utilizadas as bibliotecas okhttp3 e retrofit2, elas facilitam a realização de operações de rede, como enviar solicitações HTTP para um servidor, receber respostas e lidar com tarefas relacionadas à rede em aplicativos Android. Assim foi criado o corpo da requisão e criado um bitmap da imagem. Utilizando a rota criada na secção 3.4, é realizada uma requisição HTTP POST para o envio da imagem e recebimento dos dados JSON.

4 APRESENTAÇÃO DOS RESULTADOS

Nesse capítulo são apresentados os melhores resultados obtidos de cada arquiterura e os melhores resultados gerais, a comparação entre os métodos e arquiteturas utilizados em cada um e os resultados da implementação do modelo.

4.1 Análise dos treinamentos e modelos

Para alcançar o melhor desempenho do modelo, foram realizados diversos treinamentos utilizando diferentes arquiteturas. Após concluir os treinamentos e realizar a validação com o banco de dados apresentado na Tabela 3, é possível analisar e comparar os resultados de cada arquitetura. A Tabela 4 apresenta quais foram os treinamentos analisados.

Arquitetura	Época	Arquitetura	Época	Arquitetura	Época
Resnet 18	15	Resnet 18	25	Resnet 18	40
Resnet 34	15	Resnet 34	25	Resnet 34	40
Resnet 50	15	Resnet 50	25	Resnet 50	40

Tabela 4 – Arquitetura X nº de Épocas

Dos treinamentos apresentandos na Tabela 4, os melhores resultados de cada arquitetura foram *Resnet* 34 - 25 época, *Resnet* 50 - 40 época, *Resnet* 18 - 40 época, como visto na tabela abaixo:

Arquitetura	Época	Acurácia	Recall	F1
Resnet 18	40	0.9293	0.9296	0.93
Resnet 34	25	0.9542	0.9543	0.95
Resnet 50	40	0.9454	0.9454	0.94

Tabela 5 – Melhores resultados

É possível notar que a acurácia e o *recall* são valores muito próximos ou até mesmo iguais, isso significa que os modelos tem um desempenho consistente ao longo de todas as classes, como dito em 2.4. No apêndice A, é possível ver que os resultados dos outros modelos foram bem próximos aos apresentados na Tabela 5, assim podemos concluir que todos os treinos alcançaram um bom resultado.

Nas listas abaixo consta outro tipo dado que é analisável: a precisão dos modelos em cada classe específica. Avaliar a precisão das classes permite visualizar melhor o desempenho do modelo para cada situação de classificação. Isso ajuda a identificar os pontos fortes e as falhas do modelo, orientando ajustes e melhorias.

Resnet 18 época 40 acurácia específica de classe

- ___Tomato_Bacterial_spot: 0.9800000000000001
- ___Tomato_Early_blight: 0.8604089560936374
- ___Tomato_Late_blight: 0.7493076923076923
- ___Tomato_Leaf_Mold: 0.9612359550561798
- ___Tomato_Septoria_leaf_spot: 0.9610169491525424
- ___Tomato_Tomato_Yellow_Leaf_Curl_Virus: 0.9825825825825826
- ___Tomato_Tomato_mosaic_virus: 0.9006024096385543
- ___Tomato_healthy: 0.9994813278008299

Resnet 34 época 25 acurácia específica de classe

- Tomato ___ Bacterial_spot: 0.98236092265956
- Tomato ___ Early_blight: 0.9239752333586626
- Tomato ___ Late_blight: 0.8130637514231499
- Tomato ___ Leaf_Mold: 0.9888155568899387
- Tomato ___ Septoria_leaf_spot: 0.9696123643403597
- Tomato ___ Tomato_Yellow_Leaf_Curl_Virus: 0.9805690113651477
- Tomato ___ Tomato_mosaic_virus: 0.9770612200425074
- Tomato ___ healthy: 0.9989615784008308

Resnet 50 época 40 acurácia específica de classe

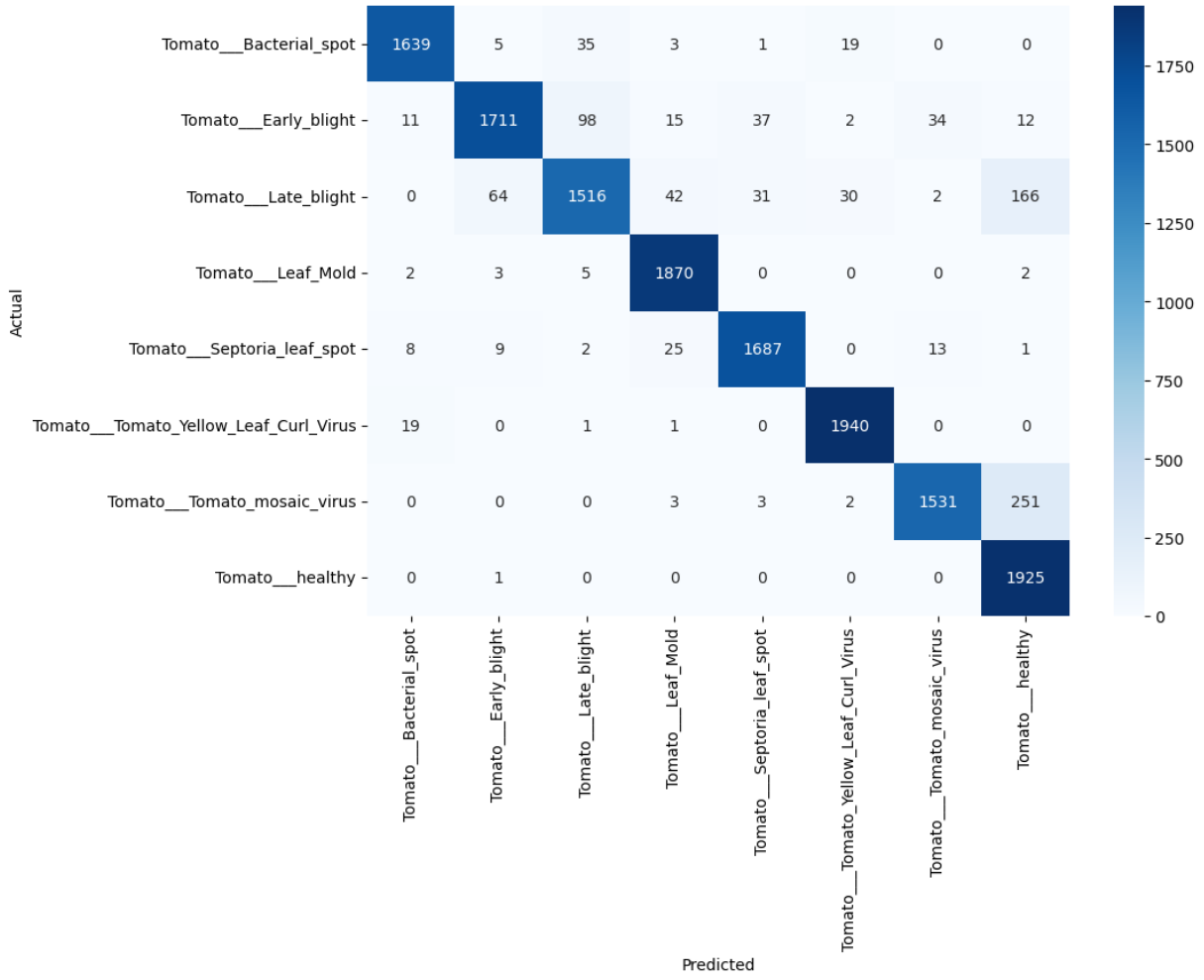
- ___ Tomato_Bacterial_spot: 0.977141065830721
- ___ Tomato_Early_blight: 0.9063478487367087
- ___Tomato_Late_blight: 0.7898391453120654
- ___Tomato_Leaf_Mold: 0.9936385973279506
- ___Tomato_Septoria_leaf_spot: 0.9759487750556793
- ___Tomato_Tomato_Yellow_Leaf_Curl_Virus: 0.9872546378959263

- ___ Tomato_Tomato_mosaic_virus: 0.9335276967930029
- ___Tomato_healthy: 0.9994807892004154

As listas mostram que todos os modelos tiveram mais dificuldade na classificação das classes *Late Blight* e *Early Blight*. Esse foi um problema notado logo no começo do trabalho, quando foi feita a verificação do banco de dados, por que as duas doenças possuem caracteristicas muito similares, e há poucos pontos que as diferenciam. Uma forma de melhorar os resultados seria utilizar imagens com maior resolução, o que forneceria ao modelo mais informações para aprender e classificar as classes.

Esses problemas também podem ser visualizados por imagem, através de uma matriz confusão, como na figura 25. Pode-se observar no apendice X, eixo das predições, que o modelo confundiu as classes *Late Blight* e *Early Blight* respectivamente 96 e 64 classificações. O modelo também classificou erroneamente muitas fotos como *Healthy*, o motivo desses erros pode ser *miss labels*, como discutido previamente, ou parâmetros de treinamento que precisam ser ajustados. Mas para os parâmetros pré estabelecidos para esse trabalho o modelo apresentado na imagem teve um mal desempenho.

Figura 25 – Matriz confusão dos dados de validação - resnet 50 época 25



Também é possivel comparar o tempo de treinamento e validação de cada um dos modelos. Para melhor análise, foi montada a Tabela 6, que mostra o tempo de treinamento dos 3 melhores modelos e os seu tempo de processamento para 1 imagem. O cálculo para tempo de classificação de 1 imagem foi feito utilizando tempo que o modelo demorou para classificar o banco de dados de validação divido pela quantidade de imagens do banco, o banco possui 14777 imagens como falado em 3.3.1.

Arquitetura	Época	T(s) de treinamento	T(ms) de classificação
Resnet 18	40	291	16
Resnet 34	25	292	25
Resnet 50	40	676	61

Tabela 6 – Tempos de treinamento do modelo e validação de 1 imagem.

Tem que se ter em mente que o número de épocas muda e isso afeta o tempo de treinamento. E quanto mais complexa a arquitetura mais demorado deverá ser seu tempo de validação pois mais profunda ela é, e a quantidade de cálculos e

neurônios que ela possui são maiores e mais complexos. Modelos maiores usam mais memória de processamento e podem levar mais tempo para processar seus dados. Justamente como mostrado na Tabela 6.

Outra informação que pode ser olhada é a tabela fornecida pela função *vision_learner* que mostra as métricas do treinamento: *train_loss, valid_loss, error_rate, accuracy, epoch e time*. Focando nas 4 primeiras é possivel notar quando o treinamento está com muitas épocas, *overfitting*, convergência ou atigindo estabilidade. A figura 26 mostra um recorte das ultimas épocas dos 3 modelos discutidos até agora.

Figura 26 – Métricas dos treinamentos.

Resnet 18

epoch	train_loss	valid_loss	error_rate	accuracy	time
32	0.002553	0.093099	0.023191	0.976809	00:07
33	0.001584	0.102279	0.024119	0.975881	00:07
34	0.001231	0.083587	0.023191	0.976809	00:07
35	0.001093	0.087394	0.020408	0.979592	00:07
36	0.000441	0.085330	0.022263	0.977737	00:07
37	0.000612	0.083059	0.021336	0.978664	00:07
38	0.000564	0.082600	0.019481	0.980519	00:07
39	0.000796	0.092841	0.022263	0.977737	00:07

Resnet 50

epoch	train_loss	valid_loss	error_rate	accuracy	time
32	0.001573	0.087931	0.023191	0.976809	00:16
33	0.001848	0.080881	0.017625	0.982375	00:16
34	0.002149	0.072977	0.017625	0.982375	00:16
35	0.004674	0.068484	0.016698	0.983302	00:16
36	0.002769	0.073691	0.015770	0.984230	00:16
37	0.001627	0.077169	0.019481	0.980519	00:16
38	0.001097	0.074387	0.015770	0.984230	00:16
39	0.000990	0.077905	0.016698	0.983302	00:16

Resnet 34

epoch	train_loss	valid_loss	error_rate	accuracy	time
17	0.017009	0.101202	0.023191	0.976809	00:11
18	0.011084	0.115880	0.027829	0.972171	00:11
19	0.007686	0.080897	0.022263	0.977737	00:11
20	0.010361	0.079858	0.015770	0.984230	00:11
21	0.004615	0.081713	0.014842	0.985158	00:11
22	0.004023	0.083684	0.015770	0.984230	00:11
23	0.003784	0.083555	0.015770	0.984230	00:11
24	0.002535	0.075348	0.015770	0.984230	00:11

FONTE:Do Autor(2024)

Na *Resnet 18* e *Resnet 50* nota-se que os valores de acurácia, *train_loss, valid_loss, error_rate* pioram nas ultimas épocas, isso indica que pode estar ocorrendo *overfitting* no modelo. Já para a *Resnet 34* as métricas se mantêm ou diminuem, isso indica *overfitting* ou convergência, que é quando o modelo esta atingindo o seu limite de aprendizado. E pelo resultados apresentados por essa arquitetura conclui-se que o modelo atingiu a convergência.

4.1.1 Análise do modelo com melhor desempenho

Utilizando os dados e conclusões do secção anterior é possivel afirmar que o melhor modelo foi o gerado pelo treinamento com a arquitetura *Resnet 34* com 25 épocas de treinamento. Como visto na tabela 5 esse modelo atigiu o maior valor de acurácia, recall e F1-score todos com o valor de 95%.

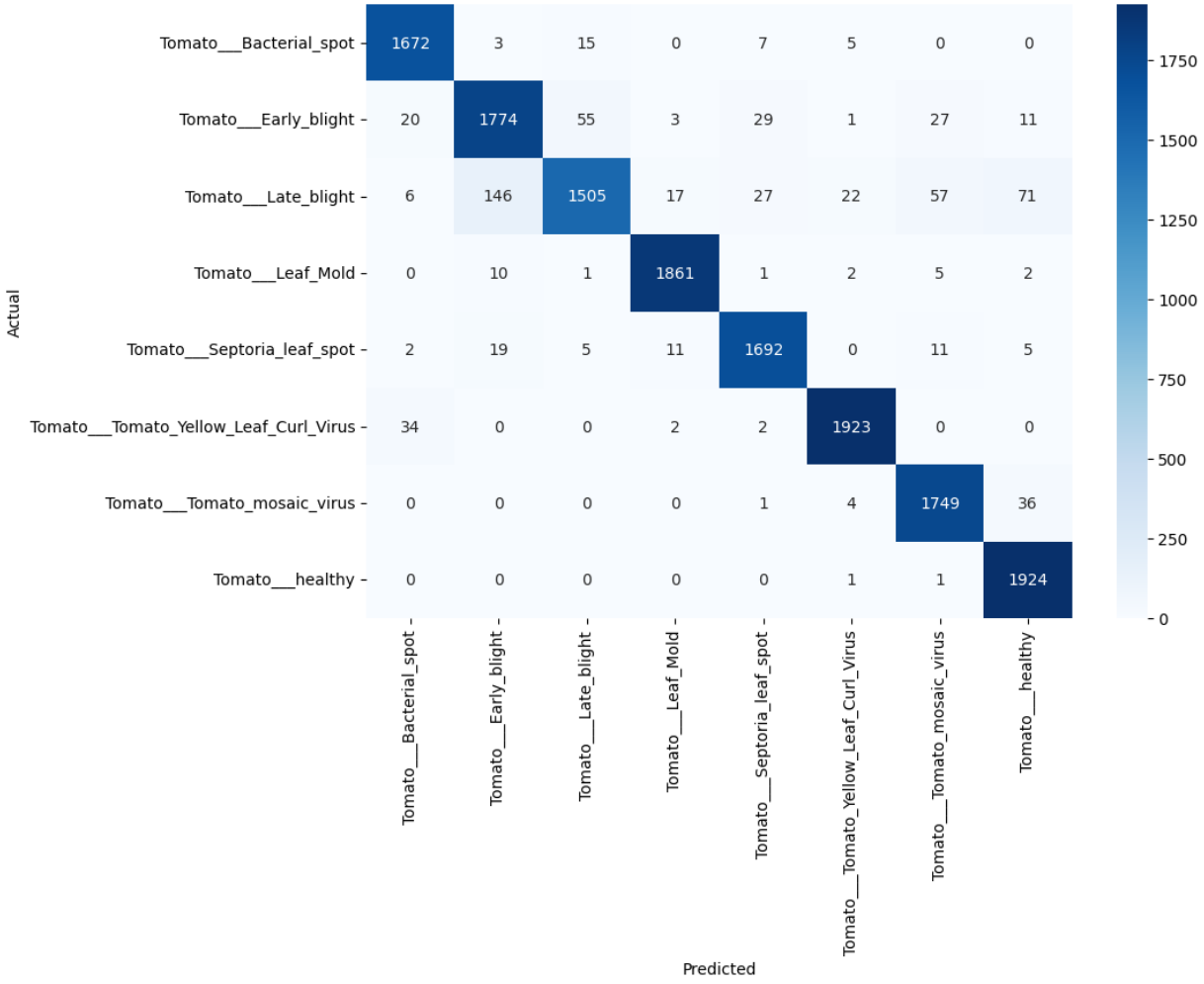
Para melhor entedimento é possivel analisar os resultados de cada uma das classes do modelo. Como consta na tabela abaixo:

	precision	**recall**	**f1-score**	**support**
Tomato_Bacterial_spot	0.96	0.98	0.97	1702
Tomato_Early_blight	0.91	0.92	0.92	1920
Tomato_Late_blight	0.95	0.81	0.88	1851
Tomato_Leaf_Mold	0.98	0.99	0.99	1882
Tomato_Septoria_leaf_spot	0.96	0.97	0.97	1745
Tomato_Tomato_Yellow_Leaf_Curl_Virus	0.98	0.98	0.98	1961
Tomato_Tomato_mosaic_virus	0.95	0.98	0.96	1790
Tomato_healthy	0.94	1.00	0.97	1926
accuracy			0.95	14777
macro avg	0.95	0.95	0.95	14777
weighted avg	0.95	0.95	0.95	14777

Tabela 7 – Desempenho de classificação por classe.

Observando a a Tabela 7 e matriz confusão na figura 27 percebe-se algumas classes com pior desempenho. Os principais erros ocorrem com as classes *Early_blight* e *Late_blight*, pois como dito anteriormente são 2 classes que possuem caracteristicas muito parecidas. Já os resultados previdos errados em outras classes podem estar ligados a *miss labels* no banco de validação, imagens com muito ruído ou qualidade ruim.

Figura 27 – Matriz confusão Resnet 34



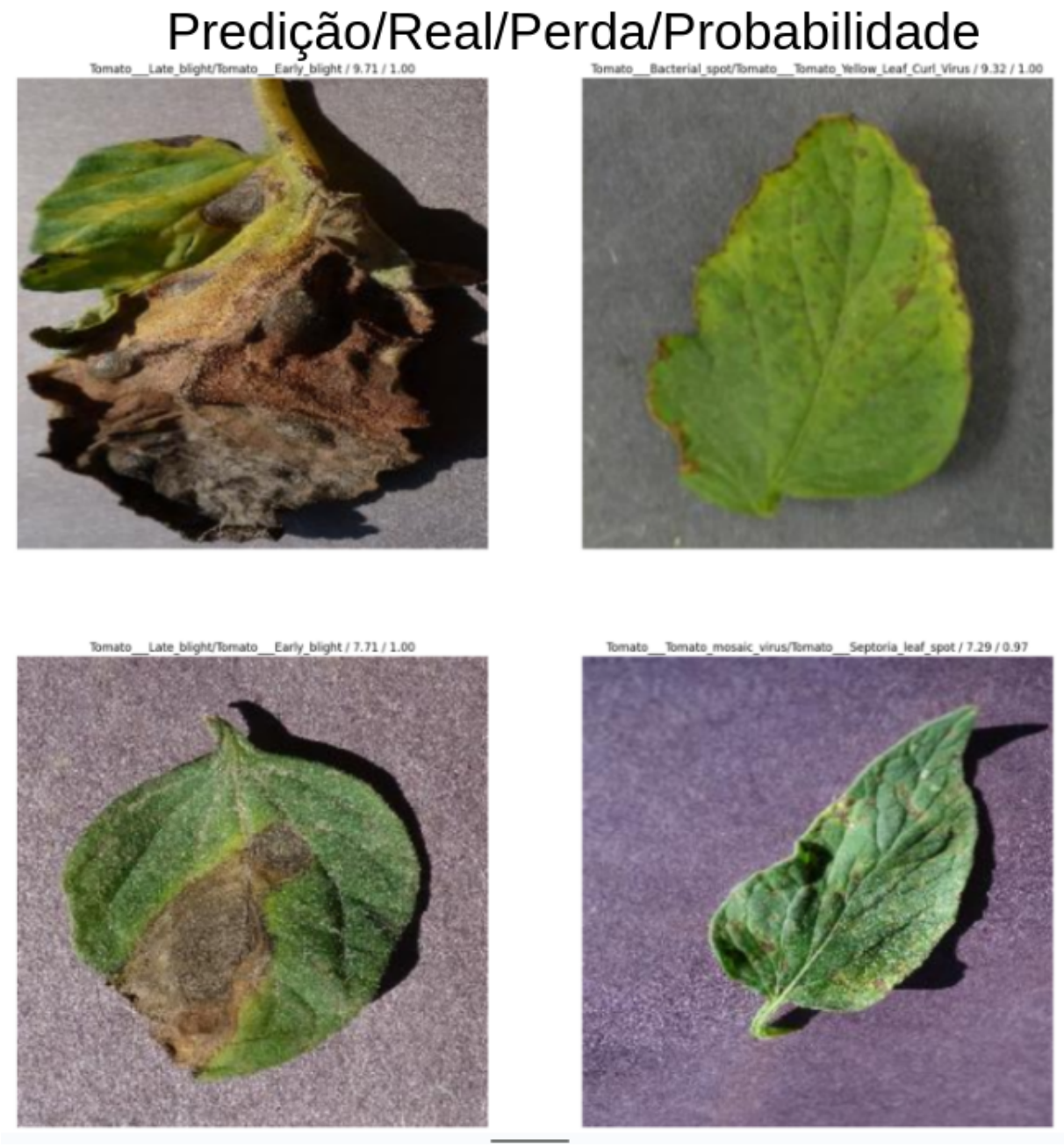
FONTE:Do Autor(2024)

A figura 28 mostra as classes que o modelo teve a maior dificuldade de classificação. Nela podemos ver como os padrões são muito parecidos e talvez com mais resolução ou qualidade na imagem seria possível reconhecer algum padrão chave para a diferenciação dessas duas classes.

Figura 28 – Imagens que o modelo classificou errado - Early e Late Blight

Na figura 29, é possível ver como *miss label*, qualidade e o ângulo da foto podem afetar o modelo. Na segunda imagem, o modelo previu a doença *yellow leaf curl virus* como *bacterial spot*. Essa imagem está borrada, com baixa qualidade e possui características que definem *bacterial spot*. Já na quarta imagem, o modelo previu *septoria leaf spot* como *mosaic virus*. Um dos motivos desse erro pode ser a angulação da folha, gerando pequenas sombras sobre a folha que podem confundir com a doença errada, além de que a resolução da folha na foto é baixa.

Figura 29 – *Top losses* da Resnet 34 época 25



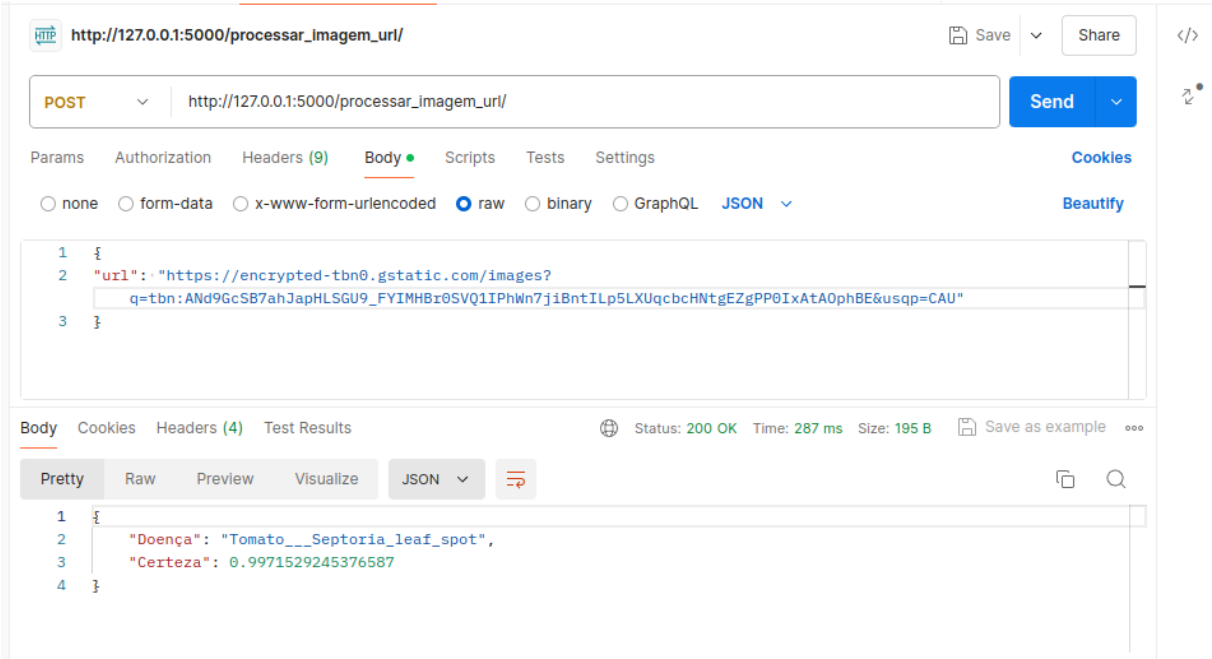
FONTE:Do Autor(2024)

4.2 Análise da API, APP e requisitos gerais do sistema

Com o modelo treinado e exportado, é possível carregá-lo na API que se comunica com o aplicativo, permitindo a classificação pelo celular. Para validar o funcionamento da API de forma isolada, sem ser com o aplicativo, foi utilizada a ferramenta *Postman*. Essa ferramenta permite o envio de requisições HTTP, que simulam as do aplicativo, para uma API e a visualização de suas respostas. Na figura 30, é possível ver o método POST sendo usado para enviar uma URL de uma imagem

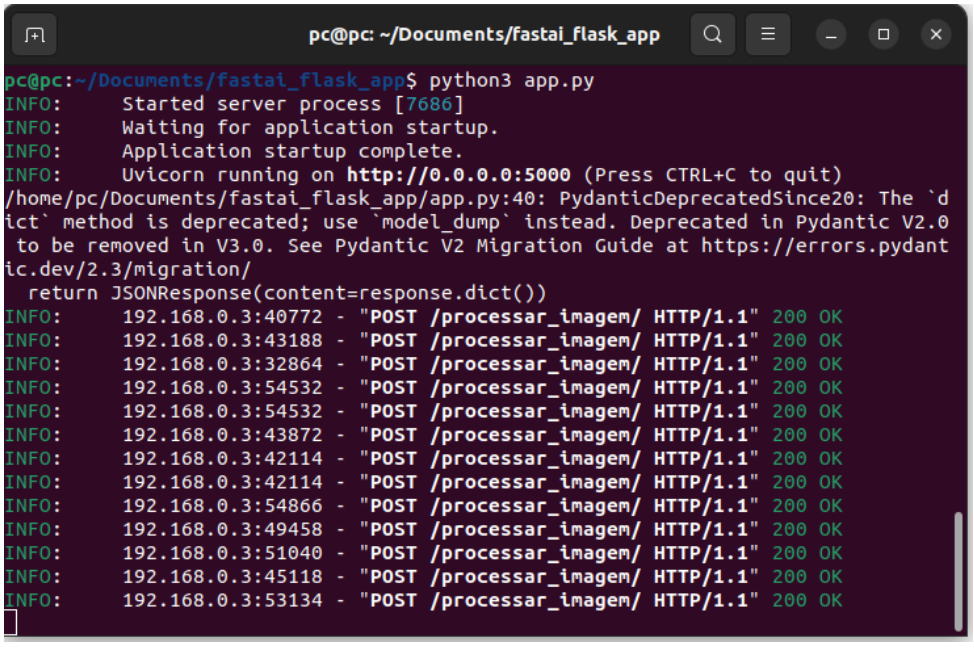
do *Google* para a API e, na resposta do body, o resultado do classificador enviado pela API. Esse simples processo valida a comunicação da API, que está rodando em máquina local, como mostra a Figura 31.

Figura 30 – Validação da API pelo Postman utilizando imagens da internet



FONTE:Do Autor(2024)

Figura 31 – Terminal Linux com API rodando em máquina local

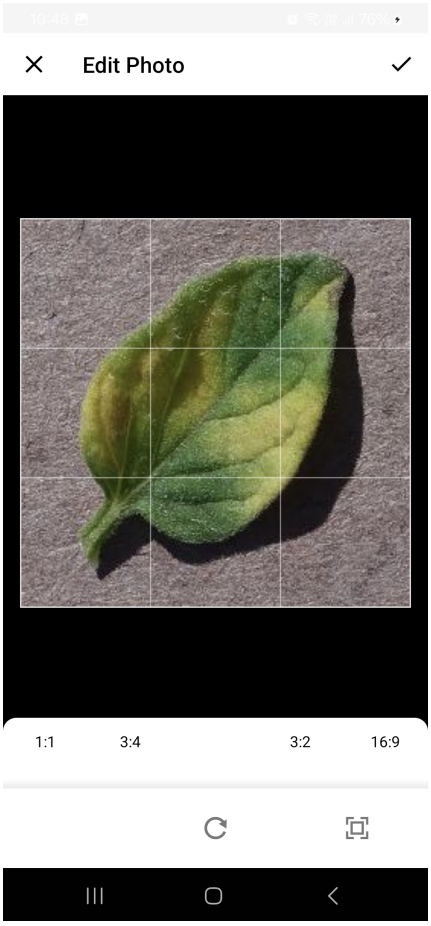


FONTE:Do Autor(2024)

Para fazer as requisições para a API, foi criado o aplicativo com um fluxo-

grama bem simples, como demonstrado em 3.5. Após a criação do aplicativo, ele foi instalado em um celular que estava conectado à mesma rede que o servidor API, para então fazer a sua validação. Nas figuras 32, 33 e 34, podemos, respectivamente, ver a seleção de uma imagem na galeria, o efeito de zoom colocado na imagem e o resultado da requisição enviado pela API. Na figura 35, pode-se ver a representação gráfica do limite de acurácia colocado lá na API.

Figura 32 – Confirmação da imagem escolhida e manipulações

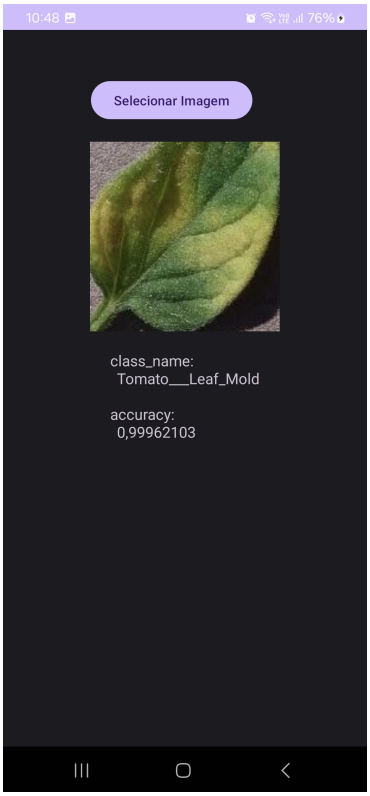


FONTE:Do Autor(2024)

Figura 33 – Manipulação de zoom aplicada na imagem

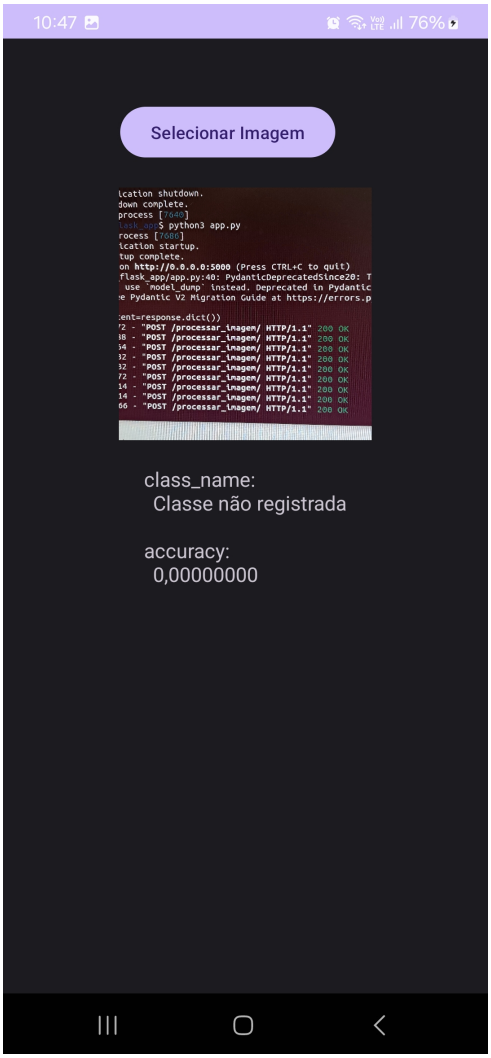
FONTE:Do Autor(2024)

Figura 34 – Tela do aplicativo apresentando resultado da classificação



FONTE:Do Autor(2024)

Figura 35 – Classificação de imagem que não é uma folha de tomate e resposta da API com resultado limitado pela acuracia menor que 60%



FONTE:Do Autor(2024)

Com base nos resultados obtidos, é possível constatar que os três melhores modelos classificadores de doença em folhas de tomate, treinados ao longo do projeto, conseguem fazer a classificação das oito classes requiridas. A comparação de vários modelos permitiu que as configurações dos treinamentos fossem aprimoradas ao longo do desenvolvimento do projeto, até que o modelo atingisse seu desempenho máximo.

As análises dos tempos de validação e treinamento dos modelos classificadores evidenciam a eficácia do sistema, com um tempo médio de resposta de 34 ms para a classificação de uma imagem. Além disso, a complexidade das arquiteturas e o número de épocas se mostram fatores decisivos para o tempo de validação e treinamento. Conclui-se também que, dependendo da aplicação, pode ser mais vantajoso utilizar uma arquitetura menos complexa, com um maior número de épocas, para alcançar altas taxas de acurácia, reduzindo, assim, o tempo de validação e possivelmente de treinamento.

E por fim, o sistema atendeu todos os requisitos do seu desenvolvimento. Sendo capaz de classificar corretamente as 8 classes de doenças e comunicar o resultados do modelo da através API para o aplicativo.

5 CONCLUSÕES

Neste trabalho foi desenvolvido um sistema de classificação de imagens de doença em folhas de tomate integrado a um serviço API acessado via aplicativo celular. Para atingir os requisitos, foram criados modelos classificação utilizando redes neurais convolucionais com técnicas de visão computacional e *deep learning*.

Este trabalho demonstra que soluções de visão computacional com CNN são capazes de resolver problemas com padrões orgânicos, isso é, padrões que são frequentemente irregulares, não repetitivos, e possuem complexidade que muitas vezes difere de padrões geométricos ou artificiais.

Para cumprir os requisitos do sistema foram executados uma variedade de treinamentos com 2 parâmetros definitivos, *batch_size* igual a 23 e *valid_pct* igual a 0.4, e 3 arquiteturas, Resnet18, Resnet34 e Resnet50, variando a quantidade de épocas. O modelo final com melhor resultado foi o treinado utilizando Resnet34 com 25 épocas. Os 3 melhores modelos do trabalho demonstraram eficiência na classificação das 8 classes todos com tempo de inferência menor que 65ms. Por fim o melhor modelo foi integrado ao servidor API.

Portanto, pode-se afirmar que o sistema compriu todos os requisitos requiridos, permitindo ao usário classificar qualquer uma das 8 classes através de um celular.

5.1 Sugestões para trabalhos futuros

Para trabalhos futuros são sugeridas as seguintes implementações no sistema:

- Utilizar a técnica de segmentação no treinamento do modelo. Esta técnica possibilita a classificação de cada pixel da imagem em uma classe específica. Essa abordagem aumenta a precisão e a confiabilidade da detecção, eliminando interferências na classificação, como o chão e a parte saudavel da folha. Assim, a doença será identificada como uma classe distinta, em vez de se classificar a imagem como um todo.
- Para futura produção seria necessário uma solução mais robusta de API, que consegui-se lidar com mais requisições, outros tipos de informações e dados. E também a implentação de mais telas no aplicativo e um botão de reteste.

6 REFERÊNCIAS

Zhang, Aston; Lipton, Zachary C; Li, Mu; Smola, Alexander J.**Dive into Deep Learning** In: arXiv preprint arXiv:2106.11342, 2021.

Kaiming, He; X, Zhang; Shaoqing, Ren; Jian, Sun.**Deep Residual Learning for Image Recognition**. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016. 770-778. Disponível em: https://api.semanticscholar.org/CorpusID:206594692.

Kirtan Jha, Aalap Doshi, Poojan Patel, Manan Shah.**A comprehensive review on automation in agriculture using artificial intelligence**, 2019.

MOLNAR,Christoph.**Interpretable machine learning: a guide for making black box models interpretable**.1ªedição.Lulu.com, 2019.

HOWARD, Jeremy;GUGGER, Sylvain.**Deep Learning for Coders with fastai and PyTorch: AI Applications Without a PhD**.1ªedição.Canada:O'Reilly Media, 2020.

OSÓRIO, Fernando;BITTENCOURT, João.R. **Sistemas Inteligentes baseados em redes neurais artificiais aplicados ao processamento de imagens**. In: I WORKSHOP DE INTELIGÊNCIA ARTIFICIAL UNISC, nº, 2000, Santa Cruz do Sul.*Anais*... São Leopoldo: Centro de Ciências Exatas e Tecnológicas da UNISINOS, 2000. 2-25.

ERAZO,J.J.M.**DESENVOLVIMENTO DE UM SISTEMA DE CONTAGEM E CLASSIFICAÇÃO DE VEÍCULOS UTILIZANDO REDES NEURAIS CONVOLUCIONAIS**.Dissertação(Mestrado em Engenharia de Automação e Sistemas)- UNIVERSIDADE FEDERAL DE SANTA CATARINA CTC - CENTRO TECNOLÓGICO DA UFSC. Florianópolis, p.135.2021.

LEMOS,Y.M.V.**Inspeção automática de defeitos em ovos comerciais usando visão computacional**.Dissertação(Mestrado em Engenharia de Automação e Sistemas)- UNIVERSIDADE FEDERAL DE SANTA CATARINA CAMPUS FLORIANÓPOLIS. Florianópolis, p.169.2021.

Bhatt, D.;Patel, C.;Talsania, H.;Patel, J.;Vaghela, R.;Pandya, S.;Modi, K.; Ghayvat, H. **CNN Variants for Computer Vision: History, Architecture, Application, Challenges and Future Scope**.Electronics 2021.Disponível em:https://doi.org/10.3390/electronics10202470.

MARTINS, Lucimara; MURAMATSU JÚNIOR, Mario; SERAPIÃO, Adriane. **Classificação de imagens de ideogramas Kuzushiji com Redes Neurais Convolucionais**. In: ENCONTRO NACIONAL DE INTELIGÊNCIA ARTIFICIAL E COMPUTACIONAL (ENIAC), 16. , 2019, Salvador. Anais [...]. Porto Alegre: Sociedade Brasileira de Computação, 2019 . p. 309-320. ISSN 2763-9061. DOI: https://doi.org/10.5753/eniac.20

19.9293.

Ebermam, E.; Krohling. R. **Uma Introdução Compreensiva às Redes Neurais Convolucionais: Um Estudo de Caso para Reconhecimento de Caracteres Alfabéticos**. Revista de Sistemas de Informação da FSMA n. 22 (2018) pp. 49-59.

Kunal Banerjee, Vishak Prasad C, Rishi Raj Gupta, Karthik Vyas, Anushree H.**Exploring Alternatives to Softmax Function** Intel Corporation Bangalore, Karnataka, India. Disponível em:https://arxiv.org/abs/2011.11538.

Howard, Jeremy;Gugger, Sylvain. **Deep Learning for Coders with Fastai and PyTorch: AI Applications Without a PhD**. Disponível em: https://course.fast.ai/Resources/book.html

KARPATHY, Andrej. **Transfer Learning. Notas do curso CS231n: Convolutional Neural Networks for Visual Recognition**. Disponível em: https://cs231n.github.io/transfer-learning.

FASTAI. **Fastai-visionlearner.2023.** Disponivel em: https://fastai1.fast.ai/vision.learner.html

FASTAI. **Fastai-datablocks.2023**. Disponivel em:https://docs.fast.ai/ tutorial.datablock.html

FASTAI. **Fastai-dataloader.2023.** Disponivel em:https://docs.fast.ai/data.load.html

FASTAI. **Fastai-docs.2023.** Disponivel em: https://docs.fast.ai

Equipe DSA. **Deep Learning Book Brasil.2022.** Disponível em: https://www.deeplearningbook.com.br/

Brownlee, Jason. **How to Code a Neural Network with Backpropagation In Python (from scratch)** . Code Algorithms From Scratch 22, 2021. Disponivel em: https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

Donges, Niklas. **What Is Transfer Learning? Exploring the Popular Deep Learning Approach**. 2022. Disponivel em: https://builtin.com/data-science/transfer-learning

Brownlee, Jason. **Understand the Impact of Learning Rate on Neural Network Performance**. Deep Learning Performance, 12, 2020. Disponivel em: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Dal Bo, G.; Corso, L. L. **Aplicação de aprendizado de máquina para aumento de precisão de um sistema automatizado de nutrição de suínos**. Revista Produção Online, 2428–2451.(2023). Disponivel em: https://doi.org/10.14488/1676-1901.v22i1.4586

BRIGNOLI, Ramon. **ESTUDO DE REDES NEURAIS CONVOLUCIONAIS E SUA APLICAÇÃO PARA CLASSIFICAÇÃO DE AMBIENTES**. Monografia (Graduação em Engenharia Mecâtronica) - INSTITUTO FEDERAL DE SANTA CATARINA. Florianópolis. 2021.

EMPRESA BRASILEIRA DE PESQUISA AGROPECUÁRIA – EMBRAPA. O papel da ciência, tecnologia e inovação. Disponível em: https://www.embrapa.br/visao/o-papel-da-ciencia-tecnologia-e-inovacao.

INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA – IBGE. Informativo. Disponível em: https://biblioteca.ibge.gov.br/visualizacao/livros/liv101963_informativo.pdf.

AL-HIARY, H.; BANI-AHMAD, S.; REYALAT, M.; BRAIK, M.; ALRAHAMNEH, Z. Fast and Accurate Detection and Classification of Plant Diseases. International Journal of Computer Applications, v. 17, n. 1, p. 31-38, mar. 2011.

NAG, Amitava; CHANDA, Priya Raj; NANDI, Sukumar. Mobile app-based tomato disease identification with fine-tuned convolutional neural networks. Computers and Electrical Engineering, v. 112, p. 108995, 2023. Disponível em: https://doi.org/10.1016/j.compele

APÊNDICES

APÊNDICE A – TREINAMENTOS, CÓDIGOS FONTE E RESULTADOS

https://github.com/IgorDalavechia/TCC.git