CLASSIFICADOR DE PATOLOGIAS VEGETAIS A PARTIR DE IMAGENS UTILIZANDO REDES NEURAIS CONVOLUCIONAIS

MURILO PEREIRA BOTELHO

TRABALHO DE CONCLUSÃO DE CURSO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

Universidade de Brasília

Faculdade de Tecnologia

Departamento de Engenharia Elétrica

Classificador de Patologias Vegetais a partir de Imagens utilizando Redes Neurais Convolucionais

Murilo Pereira Botelho

Trabalho final de graduação submetido ao Departamento de Engenharia Elétrica da Faculdade de Tecnologia da Universidade de Brasília, como parte dos requisitos necessários para a obtenção do grau de Engenheiro Eletricista.

APROVADO POR:

Prof. Alexandre Ricardo Soares Romariz, Ph.D (ENE-UnB)
(Orientador)

Prof. Eduardo Peixoto Fernandes da Silva, Ph.D. (ENE-UnB)
(Examinador Interno)

Prof. João Luiz Azevedo de Carvalho, Ph.D. (ENE-UnB)
(Examinador Interno)

Brasília/DF, 27 de julho de 2023.

FICHA CATALOGRÁFICA

MURILO PEREIRA BOTELHO

Classificador de Patologias Vegetais a partir de Imagens utilizando Redes Neurais Convolucionais. [Distrito Federal] 2023.

xiii, XXp., 210 x 297 mm (ENE/FT/UnB, Engenheira Eletricista, Engenharia Elétrica, 2022).

Trabalho de Conclusão de Curso - Universidade de Brasília, Faculdade de Tecnologia.

Departamento de Engenharia Elétrica

1. Classificação de imagem
2. Rede Neural Convolucional
3. Patologias Vegetais
4. Aplicativo

I. ENE/FT/UnB
II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

BOTELHO, M. P. (2023). Classificador de Patologias Vegetais a partir de Imagens utilizando Redes Neurais Convolucionais, Trabalho de Conclusão de Curso, Publicação 2023, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF.

CESSÃO DE DIREITOS

AUTOR: Murilo Pereira Botelho

TÍTULO: Classificador de Patologias Vegetais a partir de Imagens utilizando Redes Neurais Convolucionais

GRAU: Bacharel ANO: 2023

Murilo Pereira Botelho

Departamento de Eng. Elétrica (ENE) - FT

Universidade de Brasília (UnB)

Campus Darcy Ribeiro

CEP 70919-970 - Brasília - DF - Brasil

DEDICATÓRIA

À todos aqueles que sempre
me disseram que
cada um tem seu tempo.

AGRADECIMENTOS

Aos meus pais, Denise e Geraldo, e à minha irmã, Melina, por todo o apoio durante esta época como aluno da UnB e pela compreensão durante a escrita deste trabalho. Espero que se orgulhem do que consegui fazer.

À Carol, por tornar os tempos de UnB mais incríveis e que, com sua experiência e apoio, tornou a escrita deste trabalho menos estressante e mais fluida.

Aos meus amigos de faculdade, por compartilharem os momentos incríveis e fazerem dos ruins, toleráveis.

Ao professor Alexandre Romariz, por ter aceitado me orientar, mesmo sabendo da correria que seria, por ter acreditado na ideia que, no início, não era tão clara e por me deixar sempre tranquilo que este trabalho poderia ser muito bom, contanto que eu me esforçasse para isso.

Ao meu psicólogo, Arthur, pelas conversas, que me ajudam a entender que o caminho do perfeccionismo não é possível.

RESUMO

As patologias vegetais representam uma parte considerável das perdas de colheitas todo ano e é notável que, apesar da quantidade de usuários de *smartphones* ser considerável (SNA, 2021), não se utiliza todo o potencial desses dispositivos no meio rural. Este trabalho tem por objetivo detalhar e discutir a criação de um aplicativo para classificação de imagens contendo patologias vegetais, utilizando redes neurais convolucionais, como uma solução para diminuição do impacto das patologias vegetais na perda de colheitas. Serão avaliadas as escolhas do banco de dados, do modelo de rede neural, os parâmetros da rede, as tecnologias do aplicativo e suas limitações. Foi possível obter um modelo com uma acurácia de 99,38% utilizando a mesma métrica de avaliação da referência bibliográfica e superando-a. Os resultados encontrados possibilitam um maior entendimento para a construção de uma solução que permita aos agricultores uma atuação mais barata e específica para gerenciar problemas relacionados às patologias de plantações.

Palavras-chave: Classificação de imagem; rede neural convolucional; patologia vegetal; aplicativo.

ABSTRACT

Plant pathologies represent a significant portion of crop losses each year, and it is noteworthy that, despite the considerable number of smartphone owners (SNA, 2021), the full potential of these devices is not being utilized in rural areas. This study aims to detail and discuss the creation of a mobile app for image classification of plant pathologies, using convolutional neural networks, as a solution to mitigate the impact of plant pathologies on crop loss. Choices regarding the dataset, the neural network model, network parameters, mobile app technologies, and their limitations will be evaluated. It was possible to build a model with an accuracy of 99.38%, using the same evaluation metric as the bibliographical reference and exceeding it. The results found allow for a greater understanding in building a solution that enables farmers to act more cheaply and specifically to manage problems related to crop pathologies.

Keywords: Image classification; convolutional neural network; plant pathology; app.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

GLOSSÁRIO E LISTA DE SÍMBOLOS

CNN — *Convolutional Neural Network* (Rede Neural Convolucional): rede neural especializada em análise de imagens, como classificação destas e identificação de objetos.

ILSVRC — *ImageNet Large-Scale Visual Recognition Challenge*: uma das principais competições para classificação de imagens e detecção de objetos em larga escala.

GPU — *Graphics Processing Unit* (Unidade de Processamento Gráfico): também conhecida como placa de vídeo, é responsável, principalmente, por acelerar o processamento gráfico de computadores e, atualmente, também utilizada para acelerar treinamentos de aprendizado de máquina.

API — *Application Programming Interface* (Interface de Programação de Aplicação): é um programa que cria uma interface de comunicação entre outros dois programas, por meio de protocolos e regras bem definidas.

TCP — *Transmission Control Protocol* (Protocolo de Controle de Transmissão): é um protocolo da camada de transporte responsável por fornecer uma conexão confiável entre dois dispositivos de uma rede, garantindo a entrega na ordem correta e sem perda.

HTTP — *Hypertext Transfer Protocol* (Protocolo de Transferência de Hipertexto): protocolo da camada de aplicação que utiliza do TCP para transferir informações pela internet. Ele define como as solicitações de dados são feitas pelos clientes (navegadores, aplicativos, etc.) e como servidores respondem a essas solicitações.

CAPÍTULO 1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO DO TEMA

O objetivo deste trabalho é apresentar e discutir uma solução de rápido resultado e de baixo custo para identificação de patologias vegetais presentes em propriedades rurais, por meio de imagens das folhas das plantas.

Estima-se que aproximadamente 40% das plantações sejam perdidas devido a doenças e pragas (FAO, 2022), tornando as patologias vegetais uma das principais causas de perda de lavoura em todo o mundo.

Embora grandes esforços sejam feitos para reduzir a incidência dessas doenças nas lavouras, uma parcela considerável dessas perdas poderia ser evitada com a aplicação correta de soluções específicas para cada tipo de doença. Muitas vezes, essas doenças apresentam sintomas semelhantes aos olhos humanos, mas suas causas podem ser diferentes, originando-se de agentes biológicos distintos, como pode ser observado na Figura 1.1.

Normalmente, os agricultores contratam técnicos especializados para identificar essas doenças e sugerir soluções adequadas. No entanto, dada a extensão territorial brasileira, o grande número de estabelecimentos agropecuários (aproximadamente 5 milhões, de acordo com o Censo Agro 2017 (IBGE, 2017a)) e a necessidade de monitorar constantemente as plantações, é evidente que precisamos de tecnologia para aprimorar esse processo. Além disso, a maioria das inspeções é realizada manualmente por meio de análise visual direta, e a tecnologia poderia até mesmo auxiliar os próprios técnicos (BRUNO *et al.*, 2022).

O setor agrícola é muito importante para o Brasil. Em 2023, estima-se que o faturamento das lavouras alcance um valor de R$ 835,5 bilhões, o que representa 71%

(a) (b)

Figura 1.1. Exemplo de folha com Podridão Negra em 1.1(a), causada pelo fungo *Xanthomonas campestris pv. campestris* e exemplo de folha com Esca em 1.1(b), causada por um complexo de fungos: *Phaeomoniella clamydospora*, *Phaeoacremonium aleophiluma* e *Fomitiporia spp.*

do Valor Bruto da Produção Agropecuária (VBP) (GOV.BR, 2023). Em 2022, o setor do agronegócio representou 24,8% do PIB brasileiro, mas, no setor agrícola, houve uma queda de 6,39% em relação a 2021, principalmente devido ao alto custo dos insumos, incluindo os defensivos agrícolas (ESALQ/USP, 2023).

Além disso, no Brasil, muitas pessoas dependem financeira e diretamente da agricultura, como na agricultura familiar, em que grupos de pessoas donas de estabelecimentos agrícolas utilizam de sua própria mão-de-obra e de sua família para todo o processo do cultivo.

A agricultura familiar abrange 23% da área cultivada e 77% dos estabelecimentos agrícolas, de acordo com o Censo Agro 2017 (IBGE, 2017b). Ela também representa 23% da produção total dos estabelecimentos agropecuários e é a base da economia em 90% dos municípios brasileiros com até 20 mil habitantes, de acordo com o Governo Federal. Ainda, segundo o censo, os agricultores familiares têm uma participação significativa na produção dos alimentos consumidos pelos brasileiros. Nas culturas permanentes, esse segmento responde por 48% do valor da produção de café e banana, enquanto que nas culturas temporárias, são responsáveis por 80% do valor da produção de mandioca, 69% do abacaxi e 42% do feijão.

Entretanto, esse grupo não possui a mesma possibilidade de investimentos em tecnologias mais avançadas. No setor agropecuário, as tecnologias para identificação de doenças de plantas podem ser muito caras e de difícil acesso, como um capacete que,

quando usado por um especialista, é capaz de aprender e, posteriormente, compreender, por meio dos sinais cerebrais do usuário, se uma folha de planta está saudável ou não (CANALRURAL.COM.BR, 2023).

Porém, com a ampla presença de *smartphones* na vida das pessoas, com sua capacidade de processamento e a capacidade de tirar fotos de alta qualidade, surge uma oportunidade promissora para o desenvolvimento de tecnologias acessíveis e de baixo custo que possam auxiliar na identificação de controle das doenças que afetam a agricultura. Com o uso de aplicativos e técnicas de processamento de imagens, os agricultores podem capturar imagens das plantas afetadas e receber análises rápidas e precisas, permitindo uma detecção precoce de doenças e uma resposta mais eficiente no combate a elas. Isso não só ajudaria a reduzir perdas econômicas, mas também a otimizar o uso de recursos, reduzindo tanto os custos quanto a quantidade de defensivos utilizados, promovendo uma agricultura mais sustentável.

Com isso em mente, foi desenvolvido um aplicativo de celular que, utilizando redes neurais convolucionais (CNNs, em inglês [1]), é capaz de analisar imagens de folhas e detectar a presença de doenças nas plantas. Esse sistema utiliza algoritmos de aprendizado de máquina para identificar os padrões característicos das doenças nas folhas e, assim, oferece uma avaliação precisa e sugerir soluções.

1.2 OBJETIVOS DO TRABALHO

- Apresentar e discutir as dificuldades de se identificar doenças vegetais e as consequências para a segurança alimentar global;

- Explicar e detalhar como funcionam redes neurais simples e redes neurais convolucionais, descrevendo o papel de cada parâmetro e hiperparâmetro e suas limitações;

[1] A maioria das traduções utilizadas neste trabalho foram retiradas de (BURKOV, 2019). Foram incluídas as traduções para o inglês entre parênteses quando se achou necessário para ficar mais claro algumas palavras que ainda não são tão utilizadas em português e também, em relação às siglas, foram incluídas as originais, pelo mesmo motivo. Em algumas situações sem tradução, os termos em inglês foram usados.

- Modelar uma rede neural convolucional para classificar imagens de folhas de plantas e identificar possíveis patologias;
- Avaliar os diferentes modelos propostos, discutindo a escolha dos hiperparâmetros e comparando com modelos de referência;
- Implementar um aplicativo com comunicação com a rede neural, avaliando sua viabilidade para identificação de patologias vegetais e discutindo suas limitações.

1.3 ORGANIZAÇÃO DO TRABALHO

Este trabalho está organizado seguindo a seguinte estrutura de capítulos:

- no capítulo 2 será abordada a teoria por trás das redes neurais, redes neurais convolucionais, seus parâmetros e limitações e uma breve teoria sobre o funcionamento de aplicativos;
- no capítulo 3 serão abordados os detalhes do método escolhido e da modelagem da rede neural e do aplicativo;
- no capítulo 4 serão apresentados os resultados encontrados, assim como uma discussão sobre a viabilidade da solução apresentada;
- no capítulo 5 serão feitas as conclusões sobre o trabalho, baseadas nos resultados encontrados, limitações vivenciadas e propostas para eventuais trabalhos futuros que possam aperfeiçoar a solução.

CAPÍTULO 2

FUNDAMENTAÇÃO TEÓRICA

2.1 REDES NEURAIS

A essência deste trabalho está no aprendizado de uma máquina para que esta consiga identificar, com uma acurácia satisfatória, o estado e a doença de uma determinada planta, por meio de imagens de suas folhas. Atualmente, para esta e outras formas de classificações de padrões em imagens, utilizam-se principalmente as já citadas CNNs. Para melhor compreensão, entretanto, é desejável entender como funcionam modelos mais simples de redes neurais.

Redes neurais surgiram como um novo paradigma de programação. No paradigma anterior, é necessário escrever para a máquina exatamente o que se deseja que ela faça. Em um programa de processamento de imagens, por exemplo, define-se exatamente quais cálculos a máquina deve realizar com os dados de entrada para que se obtenha um resultado. Com redes neurais, no entanto, as informações que um programador fornece à rede correspondem à entrada (como no paradigma anterior) e à saída, sendo esta a diferença ao paradigma anterior. Com essas duas informações, a própria rede aprenderá quais cálculos deve fazer para que, dada uma entrada, ela consiga obter o resultado pretendido.

2.1.1 Abordagens

Dentro do novo paradigma, existem também diversas abordagens para que a máquina realize seu aprendizado. O último exemplo mencionado representa o chamado aprendizado supervisionado, uma vez que dada uma entrada, dizemos exatamente qual resultado a máquina deveria ter obtido e iterativamente a máquina cria o que chamamos

de modelo para, posteriomente, a partir de uma nova entrada e com acurácia suficiente, possa nos dizer qual é a saída mais provável, sem que providenciemos uma a ela. O maior uso desta abordagem é, então, realizar predições a partir de determinados dados. Outras abordagens são os aprendizados não supervisionados, semissupervisionados e por reforço (BURKOV, 2019).

Nos aprendizados não supervisionados, não são disponibilizadas as saídas desejadas para cada entrada. A própria máquina é responsável por identificar padrões, estruturas ou relações entre os dados fornecidos. Eles são usados para problemas de agrupamento de dados (*clustering*) e também para problemas de redução de dimensão (BURKOV, 2019), pois a rede, por meio da identificação de relações entre os dados iniciais, consegue informar quais dados são mais relevantes, diminuindo a quantidade e simplificando os dados iniciais.

Na abordagem semissupervisionada, temos uma mistura dos dois aprendizados anteriores e pode ser utilizada quando temos uma quantidade de dados rotulados e não rotulados, mas é desejável aproveitar ambos (BURKOV, 2019). Assim, enquanto os dados rotulados ajudam na predição da saída, os dados não rotulados ajudam a identificar relações entre os dados e espera-se que o algoritmo sabia aproveitar ambas as informações.

Por fim, os aprendizados por reforço possuem uma maior similaridade em relação à forma como seres humanos aprendem no mundo real. Diz-se que a máquina "vive" em um ambiente, que possui um estado. Neste ambiente, a máquina pode realizar ações e o objetivo é aprender uma política, ou uma forma de se "viver", com o objetivo de maximizar as recompensas das ações (BURKOV, 2019). Em analogia a uma criança, vivendo em uma festa de aniversário (estado), esta experimenta um doce (ação), tem uma sensação positiva do seu corpo (recompensa) e tende a repetir este comportamento, dada a recompensa positiva. Entretanto, em um dado momento, os pais da criança podem brigar com ela para que não coma tantos doces (recompensa negativa). Com o tempo, a criança passa a entender como viver (política) naquele mundo de "festas de aniversário".

Neste trabalho, daremos foco ao aprendizado supervisionado, pois foi a estratégia

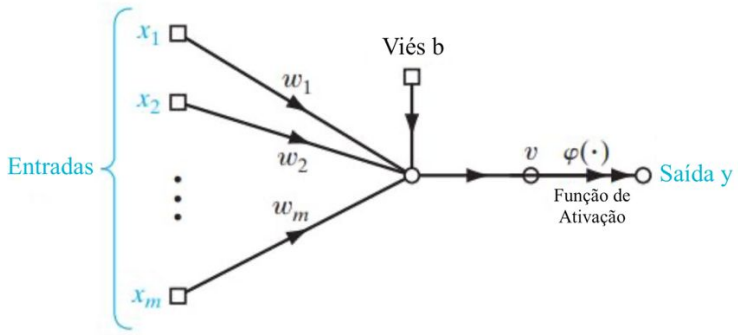


Figura 2.1. Perceptron, também chamado de Perceptron de Rosenblatt. w_m representa os pesos, x_m, os valores de entrada, b, o viés, v, o argumento da função de ativação, φ, a função de ativação e y, o valor de saída. Fonte: (HAYKIN, 2009), com modificações.

escolhida para resolver o problema posto.

2.1.2 Perceptron

Uma rede neural é essencialmente uma máquina projetada para modelar a forma como o cérebro humano processa uma determinada tarefa (HAYKIN, 2009). Esta modelagem é composta principalmente de uma grande quantidade de conexões (os chamados **pesos sinápticos**, ou somente pesos) entre unidades básicas de processamento, os chamados **neurônios** e lembra o cérebro humano em dois principais aspectos:

- conhecimento é **adquirido** pela rede por meio do ambiente, através de um **processo de aprendizado**;
- conhecimento é **retido** por meio dos pesos sinápticos, que representam a força entre as conexões entre neurônios.

O modelo mais simples de uma rede neural é conhecido por perceptron, que pode ser visto na Figura 2.1. O perceptron é formado por cinco partes principais: as entradas (representadas na figura por x_1, x_2, ... x_m) , as conexões das entradas com o neurônio em si (representadas por w_1, w_2, ..., w_m), a função de ativação (representada por φ), o viés (*bias*, representado por b) e a saída (representada por y). Essas cinco partes tem inspiração no funcionamento do neurônio biológico.

O neurônio biológico, que pode ser visto na Figura 2.2, é considerado ativado quando os sinais que recebe, por meio dos dendritos, é forte o suficiente. Quando ativado, o neurônio transmite o sinal nervoso, por meio de seu axônio e suas ramificações até

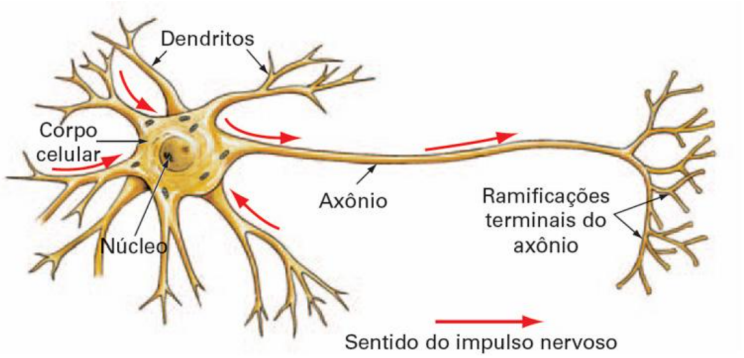


Figura 2.2. Neurônio biológico. Fonte: (ACADEMY, 2022)

outros neurônios. No perceptron, a saída é determinada pela aplicação da função de ativação na soma do viés com a soma ponderada dos sinais com os pesos, o que pode ser definido pela seguinte equação:

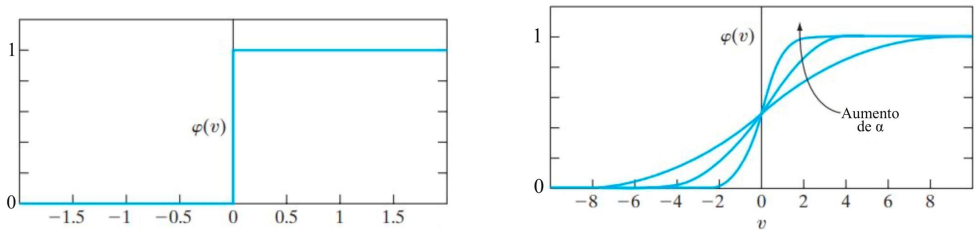
$$y = \varphi(\sum_{i=1}^{m} w_i x_i + b). \tag{2.1}$$

Enquanto o entendimento da soma ponderada é mais simples de entender, pois representa a soma das influências dos sinais, isso não vale para o viés e a função de ativação.

O viés representa um parâmetro com capacidade de alterar a saída do perceptron, mas sem depender dos sinais de entrada. É um parâmetro que indica, assim como o nome sugere, uma tendência que aquele neurônio específico tem de estar ativado ou não.

A função de ativação, por outro lado, possui duas responsabilidades: adicionar não-linearidade na saída do perceptron, fazendo com que ele consiga representar relações mais complexas entre os dados de entrada e servir como um limitador da ativação do neurônio, para que ele não possa assumir qualquer valor.

Uma função de ativação clássica é a função degrau (MCCULLOCH; PITTS, 1990), mostrada na Figura 2.3(a), pois possui uma resposta simples binária que pode ser observada pela equação seguinte:



(a) (b)

Figura 2.3. Função degrau 2.3(a) e função logística 2.3(b). Fonte: (HAYKIN, 2009), com modificações.

$$\varphi(v) = \begin{cases} 0, & \text{se } v < 0 \\ 1, & \text{se } v \geq 0. \end{cases} \tag{2.2}$$

De fato, notamos que, para qualquer valor de v menor que 0, o perceptron possui uma saída 0 e, para qualquer valor maior ou igual a 0, ele possui uma saída igual a 1.

O modelo binário de resposta do perceptron, no entanto, se mostrou limitado. Como veremos mais para frente, é um requerimento que a função de ativação seja diferenciável e a função degrau não é. Com essa necessidade, surgiram outras alternativas, como a função logística , mostrada na Figura 2.3(b) que, por ser mais suave, é diferenciável e também apresentou bons resultados empiricamente (BURKOV, 2019).

No processo de aprendizagem, os pesos vão ter seus valores atualizados, de acordo com o quão próxima a saída estará do valor desejável. Neste contexto, define-se o erro como:

$$e = d - y, \tag{2.3}$$

em que e é o erro, d é o valor desejável e y é a saída do perceptron.

Encontrado um valor de erro, a atualização do valor dos pesos segue a seguinte equação:

$$w_{(n+1)} = w_{(n)} + a[d_{(n)} - y_{(n)}]x_{(n)}, \tag{2.4}$$

em que a é a taxa de aprendizado, n representa a iteração atual (cada iteração corres-

ponde ao processo completo para uma entrada), $d_{(n)}$ é o valor desejável de saída para uma determinada iteração e $y_{(n)}$ é o valor de saída para uma determinada iteração.

Nota-se, portanto, que o novo valor de um determinado peso é proporcional ao erro e a um novo valor, chamado de taxa de aprendizado. A proporcionalidade ao erro permite que um erro muito grande cause uma alteração muito grande no peso e um erro pequeno cause uma alteração pequena.

Por outro lado, a taxa de aprendizado é um novo valor, chamado de hiperparâmetro, e possui esse nome, em contraste aos parâmetros comuns, como os pesos e o viés, por não serem aprendidos pela rede neural, mas sim definidos no início do treinamento. Ela assume valores na faixa $0 < x \leq 1$ e permite controlar o quão rápido a rede deve fazer alterações nos pesos. Entretanto, enquanto um número muito baixo pode ajudar na estabilidade da rede, mas também aumentar a carga de processamento pela demora, um número muito alto pode diminuir o tempo de treinamento, mas também causar instabilidade no sistema, fazendo com que ele nunca consiga convergir para valores bons para os pesos e o viés.

Perceptrons são usados principalmente para tarefas de classificação binária, nas quais eles aprendem a separar pontos de dados em duas classes com base em uma fronteira de decisão, um hiperplano que separa os dados de uma classe com os dados da outra. Diz-se, então, que o perceptron é capaz somente de resolver problemas linearmente separáveis. Entretanto, vários perceptrons podem ser combinados para criar arquiteturas de rede neural mais complexas, como o perceptron multicamadas, capaz de lidar com problemas de classificação não lineares.

2.1.3 Perceptron Multicamadas

Para superar a limitação imposta pelo perceptron de uma camada e para aumentar a similaridade com o cérebro humano, foi criado o modelo chamado perceptron multicamadas, como mostrado na Figura 2.4. As camadas da extremidade, são chamadas de camada de entrada e camada de saída, enquanto as camadas intermediárias tem um nome em comum e são chamadas de camadas ocultas. Elas possuem esse nome, porque

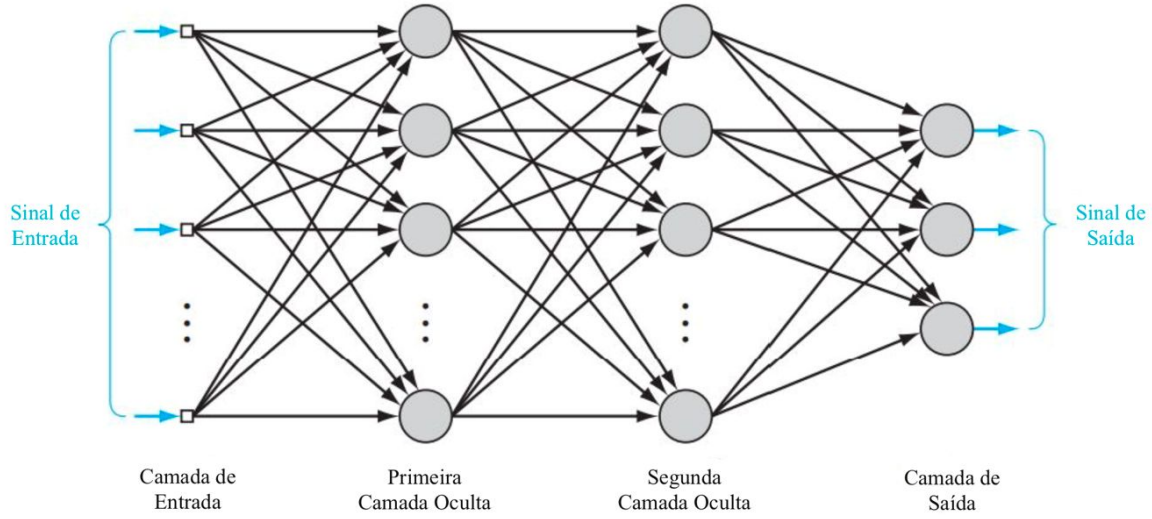


Figura 2.4. Perceptron Multicamadas. Fonte: (HAYKIN, 2009), com modificações.

elas não tem contato direto com a entrada e a saída da rede (BURKOV, 2019).

O valor de cada neurônio do perceptron multicamadas corresponderia, então, à soma das multiplicações dos valores dos neurônios da camada anterior pelos pesos das conexões entre as duas camadas. Entretanto, para introduzir não-linearidades ao sistema, é adicionada uma função de ativação que vai limitar o valor que o neurônio pode ter.

Além desses parâmetros, soma-se, para cada neurônio, ao final da soma ponderada e antes da aplicação da função de ativação, o viés, já comentado anteriormente. Assim, o valor de um neurônio j na camada L pode ser dado pela seguinte equação (generalização da Equação 2.1):

$$y_j^{(L)} = \varphi(\sum_{i=1}^{m} w_{ji}^{(L-1)} y_i^{(L-1)} + b_j^{(L)}), \tag{2.5}$$

em que $w_{ji}^{(L-1)}$ representa o valor de cada conexão que o neurônio j possui.

Depois de passar pela maioria das camadas, temos a já mencionada camada de saída, com cada neurônio, novamente, possuindo um valor, resultado da soma ponderada dos valores e pesos dos neurônios da camada anterior. Os valores dos neurônios da camada de saída são, então, comparados com um vetor que possuirá valores representando as classes determinadas no início do treinamento.

Para facilitar o entendimento, podemos supor o vetor de valores desejáveis como tendo a mesma dimensão da quantidade de classes disponíveis, em que cada elemento terá valor igual a 0 ou 1, sendo 1 se for a classe desejada e 0 se não for, processo chamado de codificação *one-hot*. Desta forma, podemos interpretar, na camada de saída, a classe do neurônio de maior valor como sendo a classe identificada. Entretanto, este pode não ser sempre o caso, e a interpretação da classe identificada dependerá da combinação dos valores dos neurônios, incluindo múltiplas interpretações dos valores de um neurônio específico, a depender de limites preestabelecidos.

Utilizando o exemplo com codificação *one-hot*, para avaliar o desempenho da rede, comparamos cada neurônio com o respectivo valor do vetor da classe a qual deveria pertencer, utilizando uma de diversas funções de custo. Uma delas é o erro quadrático médio (MSE, em inglês), que realiza uma média das diferenças quadráticas dos valores da camada de saída com o vetor com os valores de cada classe:

$$C(w, b) = \frac{1}{2n} \sum_{x} \|y(x) - d\|^2, \tag{2.6}$$

em que C é a função de custo, w são os pesos da rede, b são os vieses, n é o total de exemplos, d é o vetor de valores desejáveis para uma determinada entrada x e $y(x)$ é o vetor de valores da saída, para uma determinada entrada.

Vale citar que, embora a princípio não fosse necessário, a operação de elevar ao quadrado tem os benefícios de: permitir que apenas valores positivos sejam resultantes, penalizar mais valores muito distantes do esperado e permitir que a função de custo seja diferenciável.

A diferenciabilidade da função de custo é uma característica desejável, uma vez que para calcular os ajustes que tem que ser feitos nos pesos da rede, utilizamos da derivada da função. Como queremos diminuir o valor da função para o mínimo possível, a ideia inicial é calcular os valores de pesos e vieses que garantem que a derivada da função de custo seja igual a 0, assim como em Cálculo para encontrar o mínimo de funções.

Entretanto, nem sempre é possível encontrar este valor diretamente e este é o caso para a maioria dos problemas envolvendo redes neurais, pois a função de custo depende

de uma quantidade muito grande de interações entre os valores dos neurônios e valores de pesos e vieses (de várias camadas) e, assim, podem existir vários mínimos locais que não são globais. Resolve-se este problema, então, iterativamente a partir de dois algoritmos complementares conhecidos como descida por gradiente (*gradient descent*) e retropropagação (*backpropagation*).

A descida por gradiente possui esse nome por utilizar do gradiente da função de custo (calculado pela retropropagação) para determinar a direção de maior diminuição do valor da função de custo, que é justamente o oposto da direção do vetor gradiente. Como a função de custo é uma função de todos os pesos e vieses da rede neural, o gradiente pode ser representado da seguinte forma:

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \frac{\partial C}{\partial w^{(2)}} \\ \frac{\partial C}{\partial b^{(2)}} \\ \vdots \end{bmatrix}, \tag{2.7}$$

em que $\frac{\partial C}{\partial w^{(L)}}$, representa a derivada parcial da função de custo em relação a todos os pesos w de uma camada L e $\frac{\partial C}{\partial b^{(L)}}$, representa a derivada parcial da função de custo em relação a todos os vieses w de uma camada L.

Assim, para cada iteração de aprendizado, realiza-se um passo na direção contrária do gradiente e repete-se esse processo até encontrar um mínimo. O valor deste passo também é proporcional e controlado pela taxa de aprendizado, que ser deve testada a fim de evitar *overshooting*. O *overshooting* acontece quando a taxa de apredizado é muito alta; o algoritmo de descida por gradiente, então, pode ultrapassar um mínimo local (possivelmente global) em um determinado passo, possivelmente nunca convergindo para um valor; entretanto, valores muito baixos fazem com que a rede aprenda muito devagar e aumente a carga do sistema.

2.1.3.1 Retropropagação

O algoritmo de retropropagação é responsável por calcular o gradiente da função de custo para que seja utilizado pelo algoritmo de descida por gradiente. Iniciamos o entendimento utilizando uma versão simplificada do exemplo da função de custo MSE em 2.6, o raciocínio se mantém para outros tipos de funções de custo. Ainda, utilizaremos um perceptron multicamadas, mas supondo, inicialmente, somente um neurônio por camada (SANDERSON, 2017).

O desejo é que possamos saber o quanto devemos alterar nossos parâmetros (pesos e vieses) para que uma determinada alteração no custo seja possível. Assim, o interesse é na derivada da função de custo em relação a, por exemplo, um dos pesos da camada anterior, calculada por: $\frac{\partial C_0}{\partial w^{(L)}}$, em que L representa uma cada específica.

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial((y^{(L)} - d)^2)}{\partial w^{(L)}}, \tag{2.8}$$

em que substituímos $y(x)$ por $y^{(L)}$, para que y represente o resultado de uma função de ativação, não somente da última camada. Além disso, o parâmetro x foi retirado, pois, para uma determinada entrada x, y é uma constante em relação a x.

Para facilitar entendimento posterior, agruparemos alguns fatores em funções separadas. Definiremos $z^{(L)}$ como sendo o valor de um neurônio da camada L, antes de ocorrer a aplicação da função de ativação:

$$z^{(L)} = w^{(L)} y^{(L-1)} + b^{(L)}. \tag{2.9}$$

Definiremos, também, $d^{(L)}$ como a generalização do valor d da Equação 2.8, isto é, o valor de um neurônio após a aplicação da função de ativação:

$$d^{(L)} = \varphi(z^{(L)}). \tag{2.10}$$

Utilizando essas definições e sabendo que a função de custo é uma função composta, a derivada parcial da Equação 2.8 pode ser expandida por meio da regra da cadeia da seguinte forma:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial C_0}{\partial y^{(L)}} \frac{\partial y^{(L)}}{\partial z^{(L)}} \frac{\partial z^{(L)}}{\partial w^{(L)}}. \tag{2.11}$$

Nota-se que a variação da função de custo para um determinado peso de uma camada L depende da derivada parcial $\frac{\partial z^{(L)}}{\partial w^{(L)}}$. Mas como em 2.9, $z^{(L)}$ depende do valor da função de ativação de um neurônio da camada anterior, existe uma dependência, para o cálculo do custo de um peso de uma camada específica, com a camada anterior. É essa dependência que dá nome ao algoritmo de retropropagação.

2.1.3.2 Descida por gradiente

Calculado o gradiente, temos que agora utilizá-lo para otimizar a rede, por meio de um algoritmo de otimização, como a descida por gradiente. No algoritmo tradicional de descida por gradiente, também chamado de descida por gradiente em lote (*batch gradient descent*), encontramos a direção de maior diminuição da função de custo para todos os exemplos da rede e só então atualizamos os parâmetros. Entretanto, este algoritmo exige muita memória do sistema de processamento, uma vez que é necessário guardar informações de todos os exemplos para se realizar uma alteração nos parâmetors.

Com o objetivo de ser mais eficiente, foi criado um algoritmo que atualiza os parâmetros após cada amostra, chamado de descida por gradiente estocástico (*stochastic gradient descent*). Para uma amostra, encontramos a direção de maior diminuição da função de custo e atualizamos os pesos e vieses com os novos valores. Este algoritmo, por sua vez, também possui problemas: como se realiza um passo na direção de diminuição da função de custo a cada iteração, ou seja, sem que seja considerado todos os exemplos, pode ser que este passo não seja o passo mais rápido na direção de um mínimo.

Foi criado, com o objetivo de ser um algoritmo que se benefica de ambos os anteriores, o algoritmo de descida por gradiente em mini lotes *mini batch gradient descent*. Neste algoritmo, cada iteração de atualização dos pesos é feita em mini lotes (*mini batches*), um hiperparâmetro da rede responsável pelo agrupamento de vários dados de entrada. Assim, realizamos um passo mais certeiro na direção do mínimo da função, não

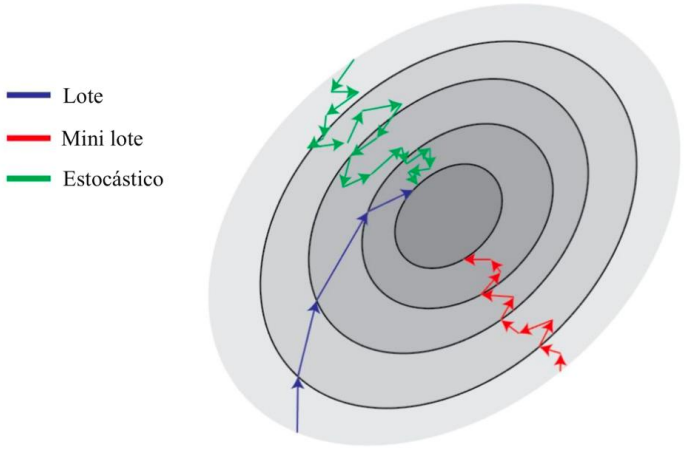


Figura 2.5. Algoritmos de descida por gradiente. Na descida por gradiente em lote, o algoritmo realiza passos mais certeiro, enquanto que na descida por gradiente estocástico, o algoritmo realiza passos menos certeiros, mas mais rapidamente. A descida por gradiente em mini lotes representa um balanço entre os outros dois. Fonte: (ZVORNICANIN, 2023).

sendo necessário manter todos os exemplos na memória do sistema e também aproveita-se o processamento em paralelo para cada exemplo dentro de um lote. Visualizações dos diferentes tipos de algoritmos de descida por gradiente podem ser observados na Figura 2.5.

Quando todos os mini lotes (e, portanto, todos os dados de entrada) realizam uma etapa do treinamento, uma nova etapa é iniciada, começando do primeiro lote novamente. Cada uma dessas etapas é chamada de época, outro hiperparâmetro da rede. Ao final de todas as épocas, o treinamento é concluído e possuímos então o modelo final da rede, com os melhores pesos e vieses, dadas às configurações de hiperparâmetros.

Com estes dados, possuímos a acurácia da rede para os dados de treinamento e conseguimos calcular a acurácia para os dados de validação. Os dados de validação correspondem a uma parte de todo o banco de dados que foi separado antes do treinamento para verificar se o modelo treinado apresenta uma boa acurácia para exemplos que nunca foram apresentados para ele antes.

Um outro hiperparâmetro importante na construção de um modelo é a quantidade de camadas ocultas. Quanto mais camadas são adicionadas, entende-se que a rede consegue abstrair mais características para melhor representar os dados. Inclusive, a verificação de aumento no desempenho de uma rede por meio de um aumento na

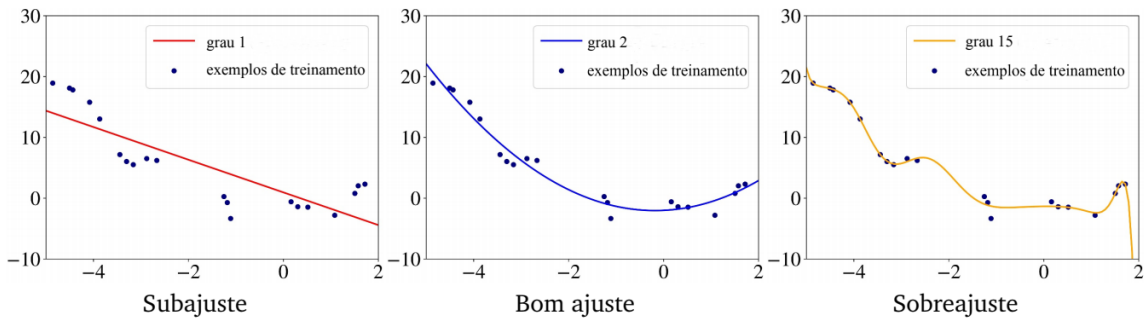


Figura 2.6. Visualização dos problemas de sobreajuste e subajuste.

quantidade de camadas ocultas deu origem ao nome "aprendizagem profunda" (*deep learning*), que representa a maior parte das redes mais modernas.

Entretanto, não se deve aumentar a quantidade de camadas ocultas sem qualquer critério, não somente porque isso exigirá mais tempo de treinamento e mais memória do sistema, mas também porque essa quantidade pode gerar sobreajuste (*overfitting*), fenômeno que acontece quando uma rede se ajusta de forma exagerada aos dados de treinamento, possuindo poucas características genéricas, o que pode ser visualizado pela Figura 2.6. O contrário também pode acontecer, gerando um subajuste (*underfitting*) no sistema, no qual a o modelo não se ajustou o suficiente e fica muito genérico.

Comentamos anteriormente a respeito da codificação *one-hot*. Ela é uma técnica muito utilizada para problemas de classificação, pois o vetor de valores desejáveis terá um tamanho igual à quantidade de classificações possíveis e os valores de cada elemento serão representados por 0 ou 1, que vão indicar a classe correta (1 para a classe correta e 0 para todas as outras classes), o que evita ter que atribuir outra padronização de números que podem possuir uma ordem de importância e confundir o treinamento (como atribuir 1 para uma classe, 2 para outra, 3 para uma terceira, etc.) e facilita a interpretação.

Para que a técnica de codificação *one-hot* seja aplicada da forma correta, é necessário que o vetor de saída da rede neural contenha apenas valores que representem a probabilidade de cada classe ser correta. Para que isso seja possível, aplicamos, apenas na última camada oculta para a camada de saída, uma função de ativação que possibilite este resultado, como a *softmax*.

A função *softmax* é uma generalização da função logística para problemas de várias classes, em que um exemplo pertence somente a uma classe e resulta em um vetor de valores cuja soma é igual a 1, possibilitando a interpretação de cada valor como sendo a probabilidade de uma classe ocorrer (GOOGLE, 2022).

Em conjunto com a *softmax*, geralmente utiliza-se uma função de custo como a entropia cruzada categórica *categorical cross-entropy*. Essa função de custo é muito utilizada em problemas de classificação com mais de duas classes e tem o benefício adicional de ajudar em problemas de desbalanceamento de banco de dados, isto é, quando uma classe possui muitos exemplos e outras nem tanto, penalizando mais erros em classes com menos exemplos e menos em classes com mais exemplos.

2.1.4 Redes Neurais Convolucionais

Apesar de o perceptron multicamadas poder resolver diversos problemas, podemos notar que será sempre necessário adicionar na entrada um vetor de valores. Imaginemos, então, o problema de classificação de uma imagem. Poderíamos, a princípio, adicionar os valores de cada pixel da imagem como sendo esse vetor para a entrada. Isso poderia até funcionar para imagens de pequena resolução e em preto e branco, como no caso das redes neurais treinadas para resolver o problema do MNIST (TENSORFLOW, 2023).

Quando lidamos com imagens mais complexas, entretanto, possuímos imagens coloridas e com resoluções maiores. Assim, se trabalharmos, por exemplo, com uma imagem 256x256 colorida, notamos que, apenas na entrada possuímos $256 \cdot 256 \cdot 3 = 196608$ valores para o vetor de entrada. Para a primeira camada oculta, supondo uma camada de 16 neurônios (o que pode não ser suficiente, dada a quantidade de informação dos valores de entrada) teríamos $196608 \cdot 16 + 196608 = 3.342.336$ parâmetros. Essa quantidade de parâmetros pode escalar muito rapidamente e isso exigiria muito da máquina de treinamento.

Além disso, para imagens do mundo real, pixels adjacentes, em todas as direções, possuem relação entre si e representar todos esse pixels em um vetor unidimensional

acarretará em perdas dessas informações de espacialidade.

Para superar esse desafio, foram propostas as redes neuras convolucionais. Nesse tipo de rede, teremos a imagem principal como a entrada da rede, porém, o cálculo a ser feito será uma convolução em cima dela com determinados filtros. Os filtros são matrizes quadradas (*kernels*) de valores arbitrários a serem treinados, que podem aqui serem entendidos como as conexões (pesos) entre os neurônios do perceptron multicamadas. Em efeitos práticos, uma convolução de uma imagem representa uma varredura que os filtros fazem em cima da imagem.

Para exemplificar, consideremos um filtro 3×3, com os seguintes valores:

$$\mathbf{B} = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}. \tag{2.12}$$

Com esta matriz e considerando uma imagem em preto-e-branco (e de baixa resolução, para melhorar a compreensão), devemos iniciar nossa varredura a partir do primeiro pixel em que a matriz 3×3 não ultrapassa a borda da imagem (destacado na Figura 2.7(a)). Os valores dos pixels deste destaque são:

$$\mathbf{D} = \begin{bmatrix} 206 & 205 & 247 \\ 244 & 161 & 137 \\ 192 & 154 & 74 \end{bmatrix}. \tag{2.13}$$

O resultado do novo pixel (destacado na Figura 2.7(b)) é calculado por:

$$\begin{aligned} \textbf{novo pixel} = 206 \cdot \frac{1}{16} + 205 \cdot \frac{1}{8} + 247 \cdot \frac{1}{16} \\ +244 \cdot \frac{1}{8} + 161 \cdot \frac{1}{4} + 137 \cdot \frac{1}{8} \\ +192 \cdot \frac{1}{16} + 154 \cdot \frac{1}{8} + 75 \cdot \frac{1}{16} = 178. \end{aligned}$$

Uma vez com este valor, pulamos um pixel à direita (Figura 2.9(a)) e calculamos o valor do segundo pixel da nova imagem:

(a) (b)

Figura 2.7. Exemplo do processo de convolução da imagem 2.7(a) a partir do filtro da Equação 2.12. Fonte: (POWELL, 2015). Imagem resultante em 2.7(b).

(a) (b)

Figura 2.8. Exemplo do processo de convolução da imagem 2.7(a) a partir do filtro da Equação 2.12, para o segundo pixel. Fonte: (POWELL, 2015). Imagem resultante em 2.8.

$$\begin{aligned}\textbf{segundo novo pixel} = 205 \cdot \frac{1}{16} + 247 \cdot \frac{1}{8} + 245 \cdot \frac{1}{16} \\ +161 \cdot \frac{1}{8} + 137 \cdot \frac{1}{4} + 244 \cdot \frac{1}{8} \\ +154 \cdot \frac{1}{16} + 75 \cdot \frac{1}{8} + 200 \cdot \frac{1}{16} = 194.\end{aligned}$$

Seguimos esse procedimento para os pixels restantes da imagem e temos como resultado uma nova imagem, mostrada na Figura 2.9(b).

Podemos perceber que o filtro utilizado realiza um borrão (*blur*) na imagem. Isso ocorre por conta dos valores usados dentro do filtro. Notamos que o maior valor é o do centro, e os valores ao redor vão decrescendo com o aumento da distância ao centro e, ao mesmo tempo, de forma simétrica ao centro, o que é análogo ao racioncínio de uma curva Gaussiana. Por este motivo, o nome deste tipo de filtro é *Gaussian blur*.

(a) (b)

Figura 2.9. Exemplo do processo de convolução da imagem 2.7(a) a partir do filtro da Equação 2.12, para todos os pixels. Imagem resultante em 2.9(b). Fonte: (POWELL, 2015).

Além disso, podemos notar que a imagem resultante possui uma borda preta e que a imagem de fato possui tamanho menor que a imagem original. Isso acontece por duas razões principais e é um comportamento necessário. As duas causas dessa diminuição são:

- *padding*: podemos notar que o primeiro pixel que foi varrido não foi o primeiro pixel da imagem, mas sim o primeiro pixel que encaixa no meio do filtro 3×3. Uma técnica para resolver essa limitação consiste em aumentar o tamanho da imagem, adicionando pixels com determinados valores ao redor da imagem, para que o primeiro pixel da imagem original consiga encaixar no meio do filtro 3×3. Essa técnica é chamada de *padding* e os valores dos pixels adicionados geralmente é 0 ou algo próximo dos valores do pixels ao redor, para não alterar muito a imagem. Esta técnica nem sempre aumenta a acurácia da rede e deve ser testada caso a caso. No nosso exemplo, foram adicionados pixels pretos na imagem resultante;

- o passo da movimentação da varredura: ao concluir o cálculo do primeiro pixel, observamos que movimentamos um pixel à direita para calcular o valor do segundo pixel; entretanto, nem sempre essa movimentação consiste em um pulo de apenas um pixel à direita, podendo pular mais pixels à direita. O valor dessa movimentação tem o nome de avanço (*stride*) e assume valores maiores para imagens cuja variação de pixels próximos é baixa para diminuir a carga do treinamento.

Este comportamento é necessário, pois, no final da rede, desejamos obter um vetor

unidimensional capaz de nos dizer, assim como no perceptron multicamadas, se aquela imagem pertence a uma determinada classe.

Esta diminuição da dimensionalidade também pode ser obtida por meio de uma outra técnica, chamada de *pooling*. Um dos tipos de *pooling* mais comuns é o *max pooling*, que seleciona o valor máximo em uma região pré-definida. Isso ajuda a reduzir o número de parâmetros da rede e a extrair características invariantes de escala e posição. Ela também auxilia no processo de generalização, fornecendo uma representação resumida das características mais relevantes nas imagens.

Geralmente, no entanto, não são utilizadas imagens em preto e branco, mas sim imagens coloridas, isto é, imagens que possuem três canais de informação. Na prática, isso significa que, em vez de realizarmos uma convolução em duas dimensões, realizaremos uma convolução em três dimensões, em que a terceira dimensão são os canais da imagem.

A Figura 2.10 mostra um exemplo de uma convolução em três dimensões e também com adição do viés. Na prática, além da soma tradicional que vimos na Equação 2.1.4, que seria a soma de apenas um canal, adicionamos a ela a soma ponderada dos outros canais. Assim, o valor do pixel da Figura 2.10 foi calculado como:

$$\begin{gathered} 3 \cdot (-2) + 1 \cdot 3 + 4 \cdot 5 + 1 \cdot (-1) + \\ 2 \cdot (-2) + (-1) \cdot 3 + (-3) \cdot 5 + 1 \cdot (-1) + \\ 1 \cdot (-2) + (-1) \cdot 3 + 2 \cdot 5 + (-1) \cdot (-1) - 2 = -3. \end{gathered}$$

Utilizamos, neste exemplo, apenas um filtro 3×3 para descrever em detalhes o processo que acontece durante a varredura. Entretanto, assim como no perceptron multicamadas temos várias camadas e vários neurônios, aqui na CNN temos várias camadas e vários filtros. Assim, para cada imagem de entrada, passamos por uma quantidade definida (um outro hiperparâmetro) de filtros e cada varredura resultará em 3 imagens menores, uma para cada canal da imagem, sendo geralmente esses os canais RGB. O conjunto dessas imagens menores são chamadas de mapas de características (*feature maps*), pois cada um dos resultados é um mapa de pixels (ou, efetivamente, uma outra imagem), representando uma abstração de uma característica da imagem

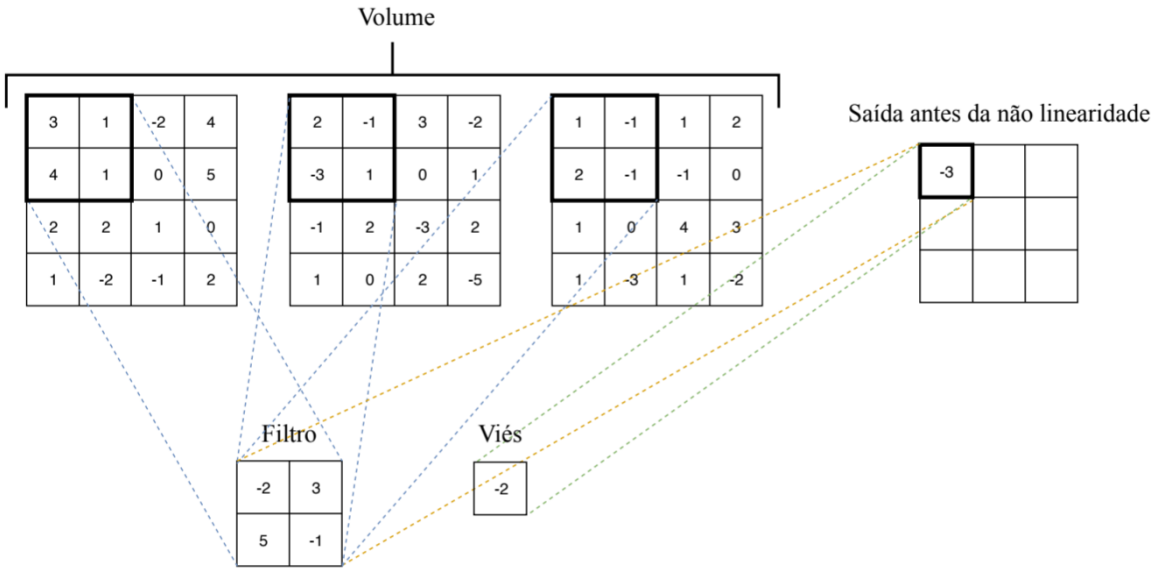


Figura 2.10. Processo de convolução para um pixel de uma imagem com volume, isto é, com três dimensões. Fonte: (BURKOV, 2019)

para compreendê-la como de uma classe diferente ou igual das outras.

2.1.4.1 *Data augmentation*

Por vezes, em se tratando de um treinamento com imagens, existe uma dificuldade maior em criar bancos de dados com o mesmo tamanho que outros tipos de dados, dado o trabalho de se tirar uma foto e o processo de tratamento para padronização do banco, como redimensionamento dessas imagens.

Para aumentar a quantidade de dados do banco, utiliza-se uma técnica chamada de *data augmentation*. Nesta técnica, aplica-se determinadas transformações nas imagens, criando outras e aumentando a quantidade de dados, como: espelhamento, ampliação, rotação, translação, aumento e diminuição de brilho e contraste, etc..

Essas transformações podem ajudar na generalização da rede, uma vez que as imagens resultantes são consideradas realistas. Por vezes, fotos são tiradas em posições diferentes, com condições de iluminação diferentes, com níveis de ampliação diferentes, entre outras possibilidades.

A aplicação desta técnica também pode ajudar a reduzir custos de obtenção, processamento e marcação dos dados. Exemplos de imagens com a técnica aplicada podem

Figura 2.11. Exemplo de transformações de uma imagem original para *data augmentation*. Fonte: (PAWARA *et al.*, 2017), com modificações.

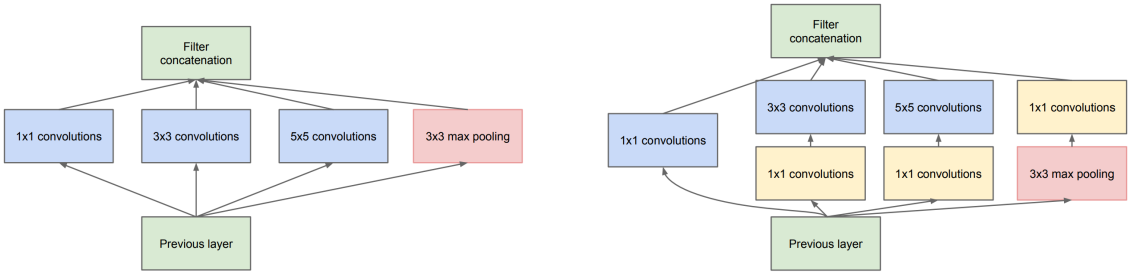
ser vistos na Figura 2.11.

2.1.4.2 Inception

Acima, foi descrito o processo tradicional de aprendizado de uma CNN, utilizando o processo tradicional de convolução. No entanto, os modelos de redes neurais que se utilizam somente de convoluções tradicionais necessitam de muitas camadas para conseguir uma alta precisão, aumentando-se a complexidade do sistema de treinamento e os requisitos de memória.

O modelo Inception foi, então, desenvolvido por uma equipe de pesquisa do Google (SZEGEDY *et al.*, 2014) em 2014, como uma solução para o compromisso entre profundidade e eficiência computacional em CNNs e foi responsável por definir o estado da arte da época no ImageNet Large-Scale Visual Recognition Challenge 2014 (ILSVRC14), uma das principais competições para classificação de imagens e detecção de objetos em larga escala.

O modelo Inception introduziu o conceito de "módulos Inception". Esses módulos são blocos de camadas com diferentes tamanhos de filtros que operam em paralelo,



(a) (b)

Figura 2.12. Módulos ingênuo em 2.12(a) e com redução de dimensionalidade em 2.12(b). Fonte: (SZEGEDY *et al.*, 2014).

diferentemente do tamanho único de filtros utilizados anteriormente. Cada módulo Inception possui várias ramificações convolucionais, cada uma usando um tamanho diferente de filtro (como 1×1, 3×3, 5×5). Essas ramificações são concatenadas para formar um único conjunto de características que é passado para a próxima camada e permitem uma captura tanto de detalhes locais (pelos filtros menores) quanto de um contexto mais amplo (pelos filtros maiores), resultando em uma melhor representação de características.

Na chamada implementação ingênua do modelo Inception 2.12(a), as ramificações convolucionais dentro de um módulo Inception têm tamanhos de filtro diferentes para capturar diferentes tipos de características. Normalmente, os tamanhos de filtro são de 1×1, 3×3 e 5×5. No entanto, isso resulta em um aumento significativo no número de parâmetros do modelo, já que cada ramificação convolucional requer uma quantidade considerável de parâmetros.

Já na implementação com redução de dimensionalidade 2.12(b), é adicionada uma camada de convolução 1×1 antes das ramificações convolucionais com tamanhos maiores de filtro. Essa camada de convolução 1×1 tem o objetivo de reduzir a dimensionalidade dos mapas de características antes de aplicar os filtros maiores.

2.1.4.3 Xception

Na prática, "a hipótese fundamental do Inception é que relações entre canais e relações espaciais são suficientemente desacopladas para ser preferível não realizar uma

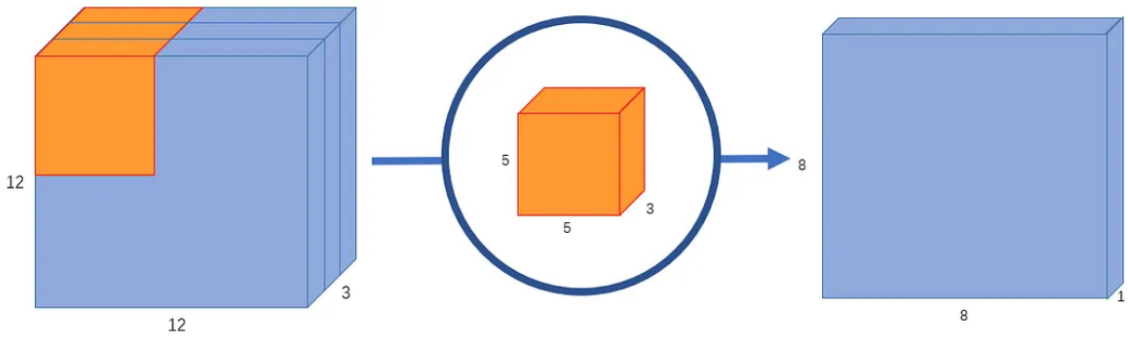


Figura 2.13. Convolução tradicional de uma imagem 12×12×3, filtro 5×5×3 e saída 8×8×1. Fonte: (WANG, 2018).

convolução única para verificação dessas relações" (CHOLLET, 2017). Em 2017, François Chollet, autor da biblioteca Keras, propôs que essas relações realmente poderiam ser completamente desacopladas, desenvolendo um modelo chamado de Xception (*Extreme Inception*), capaz de igualar a acurácia e, às vezes, superar a do Inception, mas utilizando uma quantidade bem menor de parâmetros, garantido uma menor carga para o treinamento.

No Xception, em vez de usar filtros 1×1 diferentes e separados e depois filtros maiores para cada saída, utilizam-se os mesmos filtros 1×1 em todas as entradas e depois são aplicados filtros maiores em cada canal de saída dos filtros 1×1, sem sobreposição. Na realidade, aplica-se primeiro uma convolução para verificação das relações espaciais (de dimensões maiores), chamada de convolução por profundidade (*depthwise convolution*) e, em seguida, aplica-se uma convolução para verificação de relações entre canais (utilizando filtros 1×1), chamada de convolução convolução ponto-a-ponto (*pointwise convolution*), em conjunto chamadas de convolução separável por profundidade (*depthwise separable convolution*).

Para exemplificar, consideremos uma imagem 12×12×3. Para uma convolução tradicional, deve-se utilizar um filtro também com 3 canais e, supondo que deseja-se utilizar filtros 5×5, sem *padding*, a saída da convolução será uma nova imagem 8×8×1 (Figura 2.13).

Considerando que deseja-se realizar 256 filtragens na imagem, a imagem da saída terá um formato de 8×8×256, que nada mais é que um agrupamento dos 256 resultados

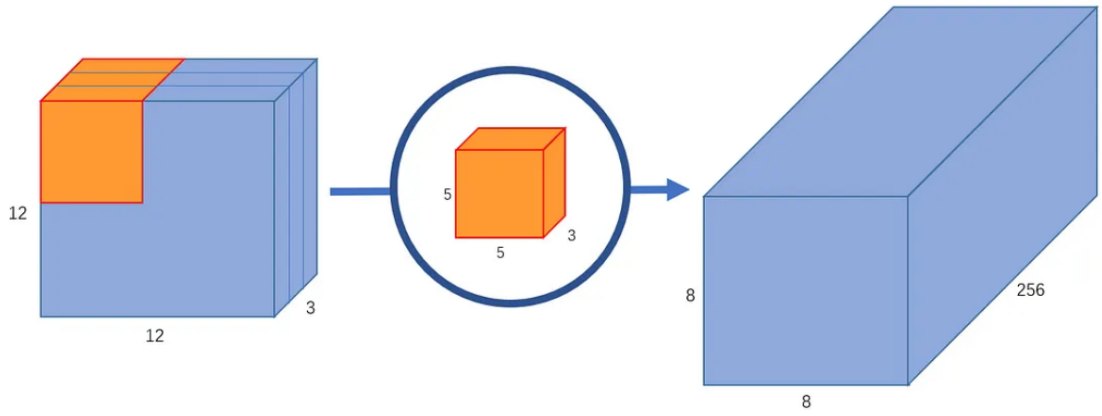


Figura 2.14. Convolução tradicional de uma imagem 12×12×3, 256 filtros 5×5×3 e saída 8×8×256. Fonte: (WANG, 2018).

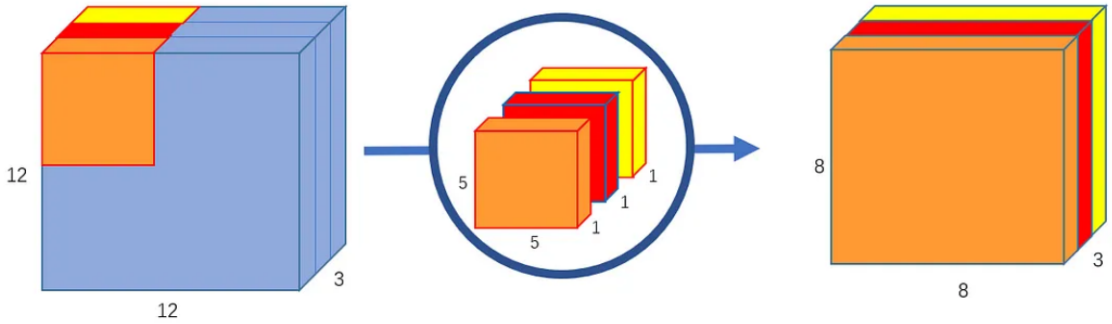


Figura 2.15. Convolução por profundidade de uma imagem 12×12×3, 3 filtros 5×5×1 e saída 8×8×3. Fonte: (WANG, 2018).

dos filtros 5×5×3 (Figura 2.14).

Na convolução por profundidade, entretanto, primeiramente é realizada uma convolução da imagem com filtros 5×5×1 diferentes para cada canal, resultando em uma nova imagem 8×8×3, isto é, sem alterar a quantidade de canais (Figura 2.15). Após, realiza-se uma convolução ponto-a-ponto, que consiste na aplicação de um filtro 1×1 com 3 canais (Figura 2.16). Como, neste caso, foram aplicados 256 filtros na convolução tradicional, aqui, deve-se utilizar a mesma quantidade para os filtros da convolução ponto-a-ponto (Figura 2.17).

A estrutura do Xception pode ser visualizada na Figura 2.19. Nela, temos as os chamados fluxo de entrada, fluxo intermediário e fluxo de saída. Para compreender os fluxos, precisamos entender antes o que são conexões residuais e normalização de lotes (*batch normalization*).

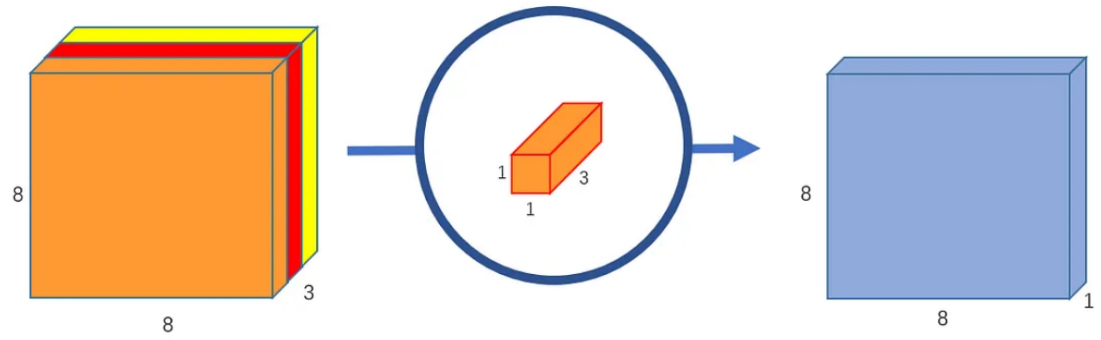


Figura 2.16. Convolução ponto-a-ponto de uma imagem 8×8×3, 1 filtro 1×1×3 e saída 8×8×1. Fonte: (WANG, 2018).

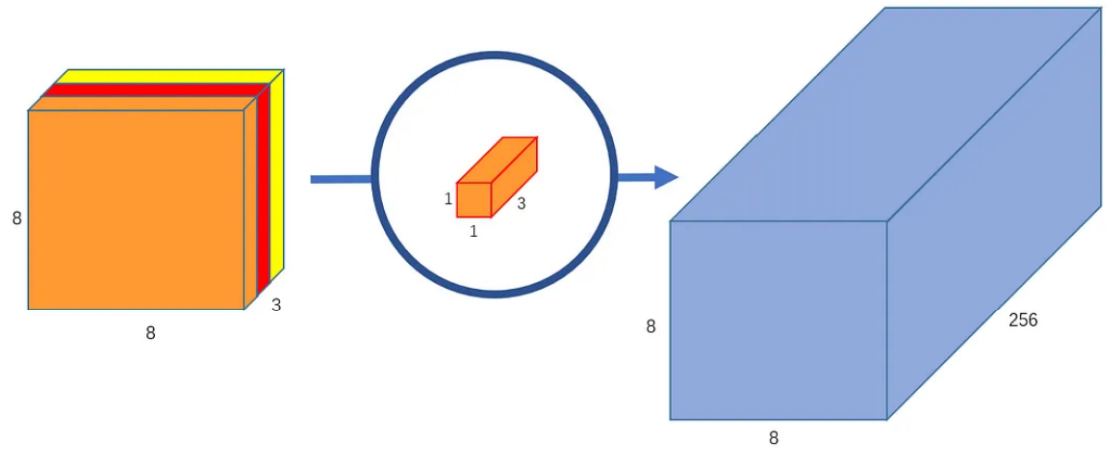


Figura 2.17. Convolução ponto-a-ponto de uma imagem 8×8×3, 256 filtros 1×1×3 e saída 8×8×256. Fonte: (WANG, 2018).

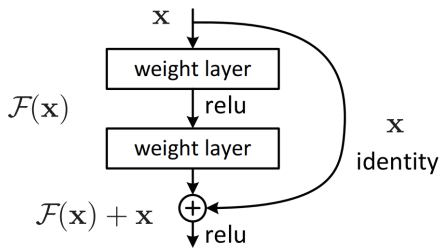


Figura 2.18. Bloco de construção de uma conexão residual. Fonte: (HE *et al.*, 2015).

Conexões residuais foram introduzidas no contexto de CNNs nos modelos chamados de ResNets (HE *et al.*, 2015), que ganharam o primeiro lugar na tarefa de classificação da ILSVRC 2015. A motivação para a criação deste modelo veio de um problema de degradação: quanto mais camadas se adicionavam em uma rede, quando ela começava a convergir, a acurácia se saturava e depois decaía rapidamente e foi verificado que não era um problema de sobreajuste.

Um bloco básico que implementa uma conexão residual pode ser visualizado na Figura 2.18. O objetivo é fazer com que a rede aprenda uma função

$$F(x) = H(x) - x, \tag{2.14}$$

em que $H(x)$ é o resultado desejado da rede e x representa a entrada.

Ou seja, deseja-se que a rede aprenda a diferença entre a entrada e a saída. Posteriormente, obtém-se a saída, $H(x)$, alterando a ordem dos fatores: $H(x) = F(x) + x$. Chamamos a conexão em si que pula uma parte da rede neural de conexão de atalho.

Na prática, essa abordagem permite o gradiente fluir mais facilmente pelas conexões residuais durante a retropopagação, o que verificou-se facilitar o treinamento de redes profundas.

A normalização de lotes é responsável por solucionar o problema de *internal covariate shift* (IOFFE; SZEGEDY, 2015), uma mudança da distribuição das entradas de uma camada, dada as alterações nos parâmetros de camadas anteriores durante o treinamento e foi observado que este fenômeno deixa os treinamentos mais lentos. A normalização de lotes ajuda nesse problema ao normalizar as entradas das camadas, transformando a entrada para que possua uma média próxima a 0 e o desvio padrão

próximo a 1.

Com isso em mente, o fluxo de entrada do Xception consiste em algumas camadas convolucionais tradicionais seguidas por várias conexões residuais, onde cada módulo residual é composto por um bloco de convolução separável por profundidade seguido por uma camada de *max pooling*. Essas camadas são projetadas para capturar características de baixo nível, como arestas, texturas e cores.

O fluxo intermediário repete o mesmo módulo 8 vezes. Cada módulo é uma série de três convoluções separáveis por profundidade. A entrada e a saída deste módulo são conectadas por uma conexão residual. Este módulo é projetado para fornecer a profundidade necessária para o modelo aprender representações mais abstratas.

O fluxo de saída novamente consiste em uma série de convoluções separáveis por profundidade e conexões de atalho, seguidas por uma camada de *global average pooling* para reduzir as dimensões espaciais dos mapas de características, responsável por fazer as previsões finais usando as características extraídas dos fluxos de entrada e intermediário.

2.1.5 Avaliação

Acima, descrevemos como são realizados diversos treinamentos de modelos de redes neurais diferentes. Ao final do treinamento, devemos realizar uma avaliação do quão bom aquele modelo é. Como também mencionado, a princípio, separamos os dados disponíveis em dois grupos separados, um de treinamento e outro de validação, sendo este último usado para avaliação do modelo para dados nunca antes vistos, o que indicaria a real acurácia do modelo.

Entretanto, devemos ter cuidado para não utilizar os dados de validação para realizar uma alteração na rede, por exemplo, dos hiperparâmetros, com o objetivo de melhorar a acurácia. Se isso for feito, o conjunto de validação perde o seu sentido de servir como uma forma de avaliar o quão bom um modelo desempenha para dados não vistos; afinal, agora, a rede está "vendo" estes dados, pois estamos realizando modificações nela a partir deles, isto é, estamos ajustando o modelo para os dados de validação,

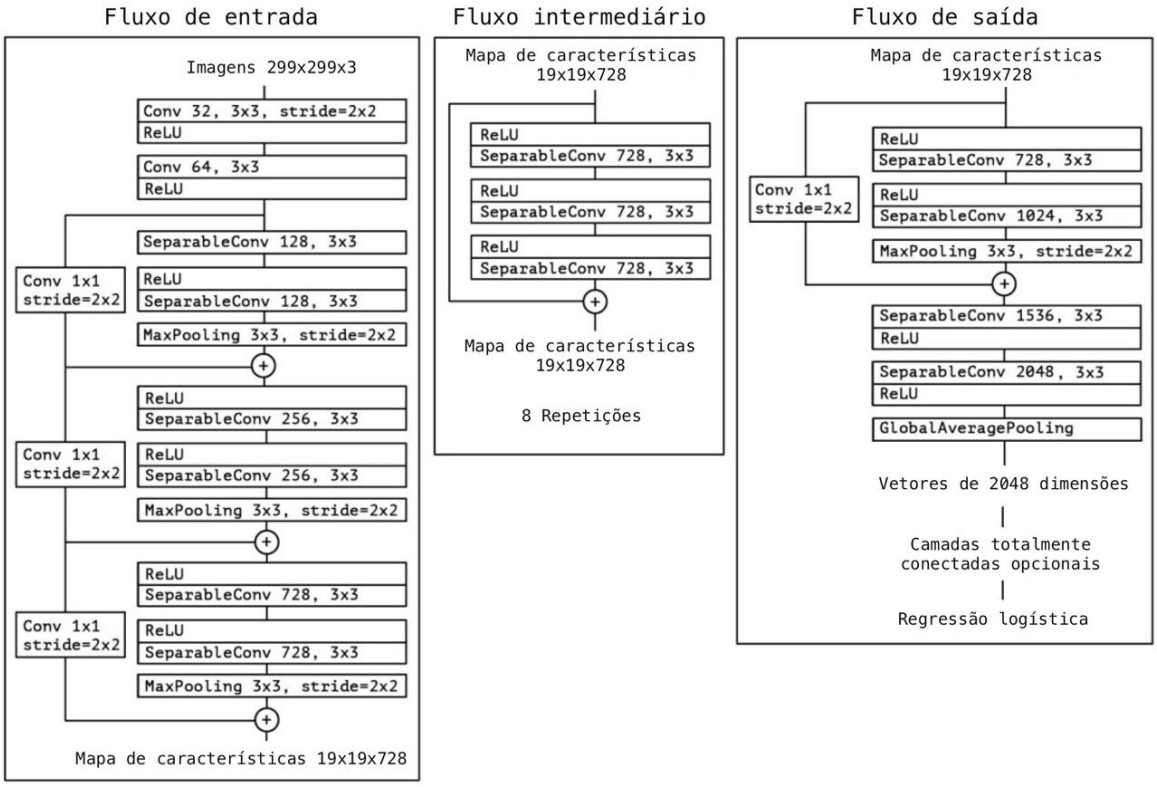


Figura 2.19. Modelo Xception. Todas as camadas de SeparableConvolution são seguidas por uma camada de BatchNormalization, que não estão explicitadas no diagrama. Fonte: (CHOLLET, 2017), com modificações.

o que pode gerar um sobreajuste para esses dados.

Contudo, é de interesse realizar alterações na rede com o objetivo de melhorar o seu desempenho para dados não vistos durante o treinamento. No treinamento, realiza-se apenas uma avaliação geral da rede e o quão bom um modelo consegue aprender, entretanto, não se realiza avaliação de diferentes escolhas de hiperparâmetros, uma avaliação mais fina e que é realizada geralmente nos dados de validação (BURKOV, 2019).

Assim, foram criadas duas estratégias principais para ser possível a alteração de hiperparâmetros nos dados de validação: a separação dos dados em mais um conjunto e a validação cruzada.

Na primeira estratégia, mais direta, separamos os dados em mais um conjunto: o de teste. Assim, realiza-se os seguinetes passos:

1. separam-se os dados em três conjuntos: treinamento, validação e teste;

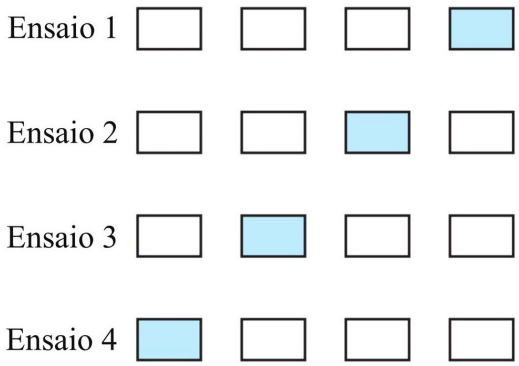


Figura 2.20. Processo de validação cruzada *k-fold*. Fonte: (HAYKIN, 2009), com modificações.

2. realiza-se o treinamento nos dados de treinamento, repetindo-os para diversas combinações de hiperparâmetros, avaliando a acurácia nos dados de validação e buscando aquela combinação com mais benefícios (entre acurácia e custo computacional);

3. realiza-se um novo treinamento utilizando ambos os dados de treinamento e validação e avalia-se a acurácia final do modelo para os dados de teste.

Esta estratégia, no entanto, possui uma grande desvantagem: se necessitamos dividir os dados em mais um conjunto, não teremos esses dados para realizar o treinamento e isso se torna uma limitação para problemas com uma quantidade baixa de dados. Neste caso, sugere-se a estratégia da validação cruzada.

Na validação cruzada, também chamada de validação cruzada *k-fold*, separamos os dados em k conjuntos e o treinamento será realizado utilizando $k-1$ dos conjuntos e a avaliação para escolha dos melhores hiperparâmetros é realizada no conjunto restante (HAYKIN, 2009). Este processo, ilustrado na Figura 2.20, é repetido, para os mesmos hiperparâmetros, até que a validação ocorra para todos os conjuntos. A acurácia final do modelo será uma medida de média entre as acurácias de cada iteração.

Esta estratégia possui uma desvantagem clara: para cada hiperparâmetro, é necessário realizar o treinamento k vezes, aumentando o custo computacional. Assim, a escolha da melhor estratégia deve levar em conta os recursos de processamento e a quantidade de dados disponíveis.

Finalizada a avaliação de um modelo, deseja-se carregá-lo em um sistema para que

ele possa realizar predições. Uma utilização comum é o carregamento do modelo em um servidor que irá expor rotas para que sites e aplicativos possam enviar uma entrada e receber a predição como resultado. Este modelo de programa é chamado de API.

2.2 API

Uma interface de programação de aplicação (API) é um programa que cria uma interface de comunicação entre outros dois programas, por meio de protocolos e regras bem definidas.

APIs são muito utilizadas em sistemas web para servir de intermediador entre um aplicação final que um usuário utiliza (site, aplicativo de celular, etc.) e um banco de dados, sendo geralmente responsável por garantir que as regras de negócio de uma empresa, por exemplo, sejam seguidas, apesar de todas as possibilidades de interação do usuário com a aplicação final. Elas também permitem que várias aplicações sejam criadas, seguindo as mesmas regras para listar, criar, editar ou deletar dados do banco e isolando estas aplicações do banco de dados em si, inclusive com regras de permissão de acesso.

O caso de APIs para visualização de horários e marcações de voos é clássico. É graças às APIs disponibilizadas por cada empresa aérea que sites como Google Flights e Decolar.com podem existir, garantindo uma experiência personalizada para os usuários, de acordo com as preferências da empresa criadora da aplicação.

A forma de comunicação de uma aplicação final com uma API é geralmente feita por meio do Protocolo de Transferência de Hipertexto (HTTP), protocolo da camada de aplicação que utiliza do Protocolo de Controle de Transmissão (TCP) para transferir informações pela internet. Ele define como as solicitações de dados são feitas pelos clientes (navegadores, aplicativos, etc.) e como servidores web respondem a essas solicitações.

Cada API define exatamente as rotas (essencialmente, URLs) disponíveis para consulta, cadastro, edição ou remoção de um registro do banco de dados. As rotas disponíveis geralmente são disponibilizadas para consulta por meio de uma documentação, com

o objetivo de facilitar a integração pelos interessados. Essa requisição, então, passará pela lógica interna da API (a depender da regra de negócio) e realizará a modificação no banco de dados.

2.3 APLICATIVO

Atualmente, para a criação de aplicativos de celular (comumente chamados de *app*), considera-se principalmente os sistemas operacionais Android e iOS. Cada sistema possui seus próprios requerimentos para rodar um app e, por isso, linguagens diferentes são utilizadas para criar um.

Para Android, a principal linguagem utilizada é Kotlin (O'REILLY, 2023), criada pela empresa JetBrains em 2010 e com adoção de mais de 60% dos desenvolvedores (DEVELOPER, 2023), por possuir uma aboradgem mais moderna e ser mais fácil de ser utilizada, diferentemente de outras linguagens, também com capacidade para criação desses apps, como Java.

Para iOS, a principal linguagem é o Swift (EDUCATIVE, 2023), criada pela própria Apple e anunciada em 2014, com intenções análogas aos problemas resolvidos pelo Kotlin: ser uma linguagem mais moderna e mais fácil de ser utilizada, em comparação com suas linguagens antecessoras, como Objective-C (APPLE, 2014), uma linguagem criada como um *superset* da linguagem C.

Como geralmente as empresas querem atender a ambos os usuários de Android e iOS, antigamente era necessário que elas sempre criassem dois aplicativos diferentes, nas duas linguagens, com as mesmas funções, o que gera um gasto adicional com equipe e manutenção.

Gerou-se um interesse, portanto, por linguagens que pudessem permitir que apps fossem escritos somente uma vez e que pudessem ser compilados para cada plataforma posteriormente: são as chamadas linguagens híbridas, em contraste com as originais, as chamadas linguagens nativas. Entretanto, inicialmente, o desempenho dessas linguagens híbridas não se equiparava ao das linguagens nativas.

Em 2015, foi anunciada pelo Facebook (hoje, Meta) uma biblioteca de código aberto de JavaScript, a linguagem mais popular da web para criação de interatividade em sites, chamada de React Native.

React Native permite construir aplicativos, escrevendo-os somente uma vez, com desempenho similar ao das linguagens nativas (a depender da aplicação), utilizando APIs nativas de cada plataforma para mostrar visualizações.

O sucesso da biblioteca se deu, principalmente, pela quantidade de desenvolvedores que já tinham experiência com JavaScript e que também tinha interesse de desenvolver apps, além de sites.

CAPÍTULO 3

MÉTODO

A seguir é descrito em detalhes e em seções, o método que foi utilizado no trabalho, desde a aquisição de dados até a validação dos resultados, e também as tecnologias e ferramentas utilizadas para a criação do aplicativo.

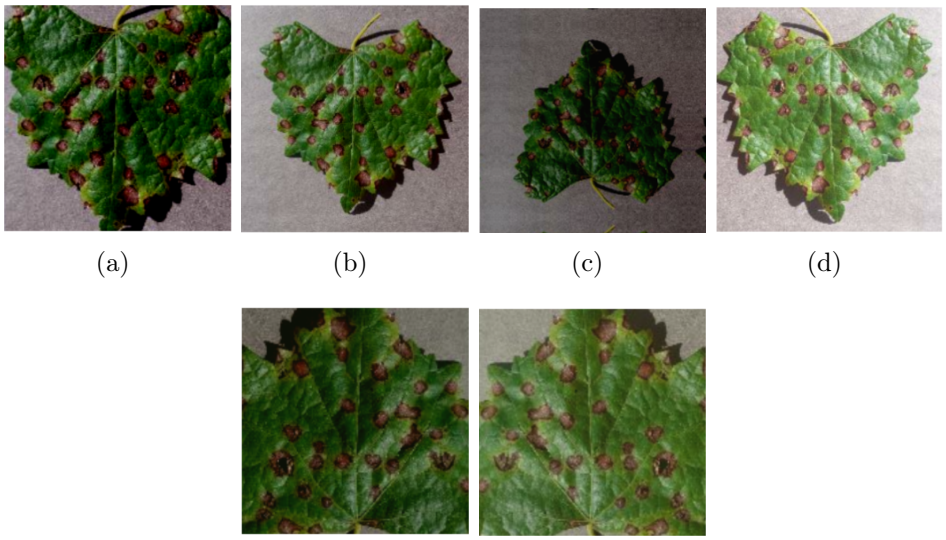
3.1 AQUISIÇÃO DE DADOS E PRÉ-PROCESSAMENTO

Para treinar a CNN foi utilizado um banco de dados com imagens originalmente disponibilizadas pela organização PlantVillage (HUGHES; SALATHE, 2016). Entretanto, as imagens não estão mais disponíveis pelo site da organização, mas sim por um artigo que usou o banco de dados e o republicou na plataforma da Mendeley (G.; J., 2017).

O banco é formado por 55448 imagens de tamanho 256×256, divido em dados já com *data augmentation* e sem. Aqui, foram utilizados apenas os dados sem *data augmentation*, pois realizamos um procedimento próprio e desejamos, no Capítulo 4, comparar os resultados com e sem a introdução de transformações nas imagens.

O procedimento próprio de *data augmentation* consistiu na introdução de uma camada que utiliza a classe Sequential do Keras, isto é, teremos as imagens como entradas e a saída será uma nova imagem, com as transformações aplicadas. Essas novas imagens são adicionadas ao conjunto original. As transformações realizadas foram:

- rotações de 90, 180 ou 270, utilizando a classe RandomFlip, com o valor de "horizontal_and_vertical";
- ampliações, utilizando a classe RandomZoom, com o valor de 0,4;



(a) (b) (c) (d)

(e) (f)

Figura 3.1. Exemplos de imagens do banco de dados com *data augmentation* aplicada.

- alteração no brilho, utilizando a classe RandomBrightness, com um valor de 0,3;
- alteração no contraste, utilizando a classe RandomContrast, com um valor de 0,3.

Essas transformações foram escolhidas por considerarmos serem alterações realistas, isto é, mudanças que podem aparecer em imagens reais. Exemplos dos resultados das transformações podem ser visualizadas na Figura 3.1.

Para realizar uma limpeza do dados, identificando imagens possivelmente corrompidas, cada uma passou por um algortimo em Python (BOTELHO, 2023c) e foi verificada a presença ou não do identificador "JFIF" nos 10 primeiros bytes do arquivo, o que verifica a validez de uma imagem JPEG. Neste banco, entretanto, não foram identificadas imagens corrompidas.

No total, são 39 classes, incluindo uma classe de imagens sem nenhuma folha específica, apenas com uma imagem de fundo para melhorar a distinção que a rede faz entre o que é um fundo e o que é uma folha. As classes com seus nomes originais e traduções podem ser encontradas na Tabela 3.1, enquanto que a quantidade de imagens disponíveis para cada classe e o agente causador da doença podem ser encontrados na Tabela 3.2.

Para realizar a tradução das classes originais, foi utilizado principalmente o site

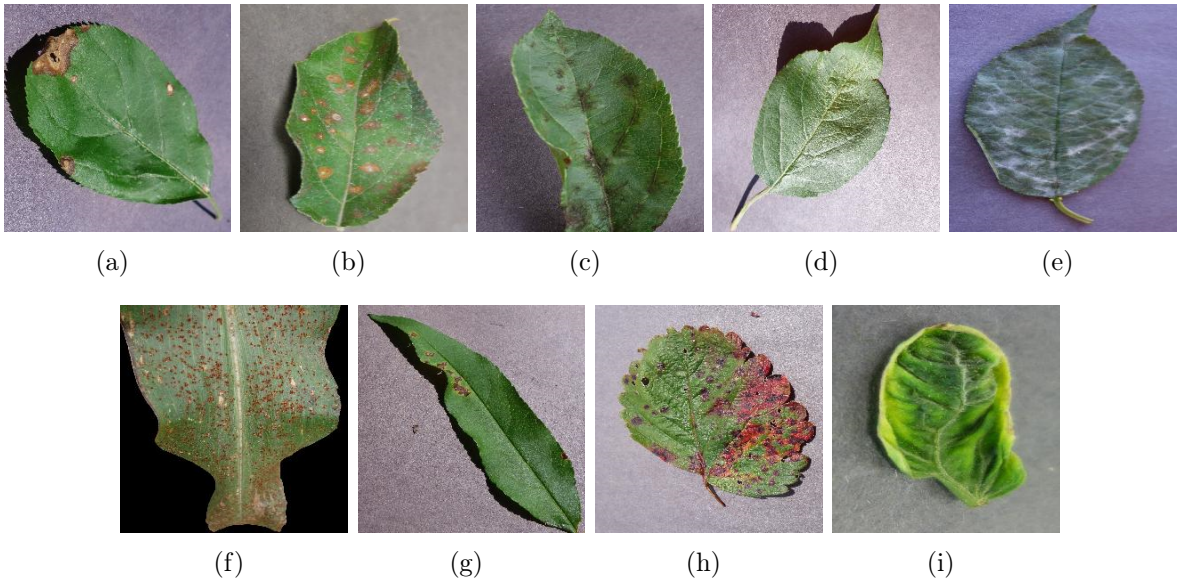


Figura 3.2. Exemplos de imagens do banco de dados. Folha de macieira com podridão negra em 3.2(a), folha de macieira com ferrugem do cedro em 3.2(b), folha de macieira com "Sarna da macieira" em 3.2(c), folha de macieira saudável em 3.2(d), folha de cerejeira com oídio 3.2(e), folha de milho com ferrugem comum 3.2(f), folha de pessegueiro com cancro bacteriano em 3.2(g), folha de morangueiro com queima de folhas 3.2(h) e folha de tomateiro com vírus da ondulação da folha amarela 3.2(i).

(AGROLINK, 2023), utilizando a ferramenta de pesquisa e inserindo o nome biológico dos agentes causadores das doenças, nomes estes disponibilizados por um artigo da organização do conjunto de dados original em (HUGHES; SALATHE, 2016).

Alguns exemplos das imagens utilizadas podem ser visualizados na Figura 3.2.

3.2 CONFIGURAÇÃO EXPERIMENTAL

Para criar o modelo e realizar o treinamento foi utilizada a biblioteca Keras e a linguagem Python. Em questões de hardware, foi utilizada uma GPU GeForce RTX 4090, com 24 GB de memória VRAM, 16384 núcleos NVIDIA CUDA e uma capacidade CUDA de 8.9. Foi utilizada também uma memória RAM de 64 GB. Os treinamentos duraram de 2 horas e meia a 4 horas para finalizarem, cada um.

O treinamento foi realizado em uma máquina local, em um ambiente Docker, um programa que permite a criação de ambientes virtuais reutilizáveis e isolados do sistema principal. Apesar de ser possível configurar um ambiente do zero, utilizamos uma imagem (um ambiente pré-pronto) disponibilizada pelo próprio Tensorflow dentro do

Classe original	Classe traduzida
Apple with scab	Sarna da macieira
Apple with black rot	Macieira com podridão negra
Apple with cedar apple rust	Macieira com ferrugem do cedro
Healthy apple	Macieira saudável
Healthy blueberry	Mirtilo saudável
Cherry with powdery mildew	Cerejeira com oídio
Healthy cherry	Cerejeira saudável
Grape with black rot	Videira com podridão negra
Corn with grey leaf spot	Milho com mancha cinzenta
Corn with common rust	Milho com ferrugem comum
Corn with northern leaf blight	Milho com mancha foliar do norte
Healthy corn	Milho saudável
Grape with black measles	Videira com sarampo negro
Grape with leaf blight	Videira com mancha foliar
Healthy grape	Videira saudável
Orange with Huanglongbing	Laranjeira com Huanglongbing
Peach with bacterial spot	Pessegueiro com cancro bacteriano
Healthy peach	Pessegueiro saudável
Pepper with bacterial spot	Pimentão com cancro bacteriano
Healthy pepper	Pimentão saudável
Potato with early blight	Batata com requeima precoce
Healthy potato	Batata saudável
Potato with late blight	Batata com requeima tardia
Healthy raspberry	Framboesa saudável
Healthy soybean	Soja saudável
Squash with powdery mildew	Abobrinha com oídio
Healthy strawberry	Morangueiro
Strawberry with leaf scorch	Morangueiro com queima de folhas
Tomato with bacterial spot	Tomateiro com mancha bacteriana
Tomato with early blight	Tomateiro com requeima precoce
Healthy tomato	Tomateiro saudável
Tomato with late blight	Tomateiro com requeima tardia
Tomato with leaf mold	Tomateiro com mofo das folhas
Tomato with septoria leaf spot	Tomateiro com mancha foliar de septoria
Tomato with two spotted spider mite	Tomateiro com ácaro-rajado
Tomato with target spot	Tomateiro com mancha-alvo
Tomato with mosaic virus	Tomateiro com vírus do mosaico
Tomato with yellow leaf curl virus	Tomateiro com vírus da ondulação da folha amarela
Background without leaf	Fundo sem folha

Tabela 3.1. Classes do banco de dados, com o nome original e traduzido.

Classe	Agente	Imagens
Sarna da macieira	Fungo (*Venturia inaequalis*)	630
Macieira com podridão negra	Fungo (*Botryosphaeria obtusa*)	621
Macieira com ferrugem do cedro	Fungo (*Gymnosporangium juniperi-virginianae*)	275
Macieira saudável	-	1645
Mirtilo saudável	-	1502
Cerejeira com oídio	Fungo (*Podosphaera spp*)	1052
Cerejeira saudável	-	854
Milho com mancha cinzenta	Fungo (*Cercospora zeae-maydis*)	513
Milho com ferrugem comum	Fungo (*Puccinia sorghi*)	1192
Milho com mancha foliar do norte	Fungo (*Exserohilum turcicum*)	985
Milho saudável	-	1162
Videira com podridão negra	Fungo (*Guignardia bidwellii.*)	1180
Videira com sarampo negro	Fungo (*Phaeomoniella spp.*)	1383
Videira com mancha foliar	Fungo (*Pseudocercospor a vitis*)	1076
Videira saudável	-	423
Laranjeira com Huanglongbing	Bactéria (*Candidatus Liber ibacter*)	5507
Pessegueiro com cancro bacteriano (mancha bacteriana)	Bactéria (*Xanthomonas campestris*)	1197
Pessegueiro sauável	-	360
Pimentão com cancro bacteriano (mancha bacteriana)	Bactéria (*Xanthomonas campestris*)	997
Pimentão saudável	-	1478
Batata com requeima precoce	Fungo (*Alternaria solani*)	1000
Batata saudável	-	152
Batata com requeima tardia	Mofo (*Phytophthora infestans*)	1000
Framboesa saudável	-	371
Soja saudável	-	5090
Abobrinha com oídio	Fungo (*Erysiphe cichoracearum / Sphaerotheca fuliginea*)	1835
Morangueiro saudável	-	456
Morangueiro com queima de folhas	Fungo (*Diplocarpon earlianum*)	1109
Tomateiro com mancha bacteriana	Bactéria (*Xanthomonas campestris pv. Vesicatoria*)	2127
Tomateiro com requeima precoce	Fungo (*Alternaria solani*)	1000
Tomateiro saudável	-	1591
Tomateiro com requeima tardia	Mofo (*Phytophthora Infestans*)	1909
Tomateiro com mofo das folhas	Fungo (*Fulvia fulva*)	952
Tomateiro com mancha foliar de septoria	Fungo (*Septoria lycopersici*)	1771
Tomateiro com ácaro-rajado	Ácaro (*Tetranychus urticae*)	1676
Tomateiro com mancha-alvo	Fungo (*Corynespora cassiicola*)	1404
Tomateiro com vírus do mosaico	Vírus (*Tomato Mosaic Virus*)	373
Tomateiro com vírus da ondulação da folha amarela	Vírus (*Tomato Yello Leaf Curl Virus*)	5357
Fundo sem folha	-	1143

Tabela 3.2. Classes do banco de dados traduzidas, o agente causador da doença e a quantidade de imagens.

Docker Hub, um site de hospedagem de imagens Docker. Assim, instalamos Docker e executamos o seguinte comando:

```
docker run --gpus all -it --rm tensorflow/tensorflow:latest-gpu.
```

O comando irá baixar e instalar a imagem, criando um ambiente com Jupyter e outras ferramentas para treinamento de redes neurais já instaladas, inclusive Python, e que já possui toda a configuração para utilização de aceleração por GPUs, uma configuração que tentamos implementar na máquina local, mas sem sucesso, porque exige muitos passos e é muito difícil, por possui muitas interações entre diferentes programas.

De qualquer forma, a implementação utilizando Docker isola todos os arquivos de ambas máquinas virtual e original, o que evita possíveis conflitos com instalações de programas com diferentes versões, como a própria linguagem Python e suas bibliotecas utilizadas.

3.3 MODELAGEM E TREINAMENTO

3.3.1 Estrutura do modelo

O modelo escolhido foi o do Xception, mas decidiu-se não utilizar a versão pré-treinada da biblioteca Keras, mas sim um modelo construído do zero (utilizando as classes do Keras), com escala menor do que o original e baseado em (CHOLLET, 2020). O principal motivo para isto foi que a versão pré-treinada não é capaz de realizar as classificações de doenças das folhas, que é o que desejamos. Isto pode ser observado analisando as categorias do banco de dados utilizado para este treinamento, disponível em (IMAGENET, 2020). Além disso, o modelo em menor escala foi disponibilizado pelo próprio criador do modelo original e um modelo em menor escala vai ao encontro ao objetivo de diminuição de custos de toda a solução.

No Capítulo 2, explicamos como funciona o modelo Xception, cuja estrutura contém três fluxos: o de entrada, o intermediário e o de saída. Nesta versão reduzida,

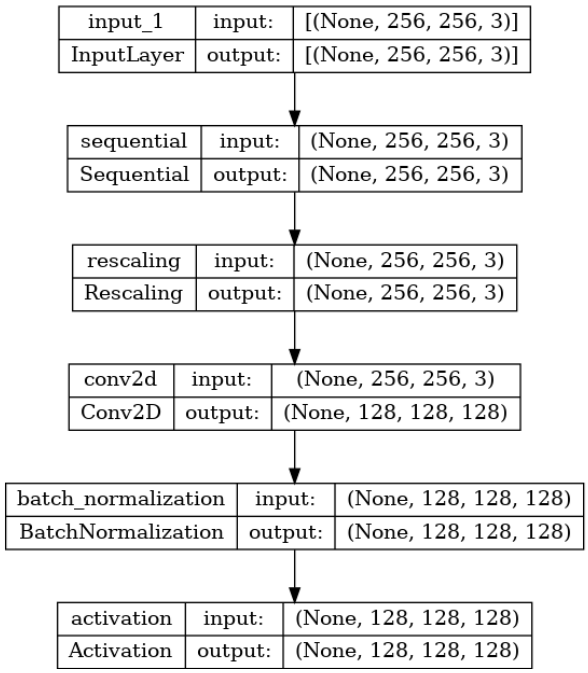


Figura 3.3. Parte inicial do fluxo de entrada do modelo utilizado.

teremos apenas o fluxo de entrada e de saída, com o objetivo de diminuir o custo computacional para o treinamento do modelo. Assim, foi retirado o fluxo intermediário, que é responsável pela abstração de características mais complexas, o que será avaliado no Capítulo 4. Então, podemos entender este modelo da seguinte forma:

Primeiramente, temos a parte inicial do fluxo de entrada, que pode ser visualizado na Figura 3.3. Aqui, temos as seguintes camadas:

1. a primeira camada, *input_ 1*, utiliza a classe InputLayer do Keras e representa os dados da imagem que serão entregues à rede para treinamento. Como cada imagem possui 256×256 pixels de tamanho e 3 canais (RGB), esta camada possui uma entrada e saída no formato (em **Tuple** do Python) de (256, 256, 3);

2. a segunda camada, chamada de *rescaling*, utiliza a classe Rescaling e é uma camada que, como o próprio nome indica, irá realizar um escalonamento dos valores dos pixels de cada canal e, em vez de estarem na faixa [0, 255], estarão na faixa [0, 1]. Assim, a saída desta camada tem o mesmo formato de sua entrada;

3. a terceira camada, chamada de *conv2d*, utiliza a classe Conv2D e é responsável por realizar a convolução em duas dimensões da imagem, abstraindo relações

espaciais. Utilizamos 64 filtros 3×3 com um *stride* de valor 2 e um valor de *padding* "same", que irá preencher regiões das imagens que antes não existiam para que todos os pixels da imagem original sejam varridos (com exceção dos pixels pulados pelo stride). Ao se utilizar o valor "same" de *padding*, o formato das imagens de saída podem ser calculados pela seguinte fórmula:

$$formato\_de\_saída = \left\lfloor \frac{formato\_de\_entrada - 1}{stride} \right\rfloor + 1, \tag{3.1}$$

em que: *formato_de_entrada* se refere ao tamanho da imagem de entrada sem os canais, no formato de Tuple, e *stride* representa o valor de stride mencionado acima. No nosso caso, como o tamanho da imagem é (256, 256), temos que, *formato_de_entrada* terá o valor de (128, 128), exatamente a metade do tamanho original, assim como comentado na fundamentação teórica. Como são utilizados 128 filtros, temos que o formato da saída da camada é (128, 128, 128);

4. a quarta camada, chamada de *batch_normalization*, utiliza a classe BatchNormalization e é responsável por aplicar uma transformação para manter a média da saída próxima a 0 e o desvio padrão da saída próximo a 1, dando maior estabilidade para a rede. A saída da rede tem o mesmo formato da entrada: (128, 128, 128);

5. a quinta camada, chamada de *activation*, utiliza a classe Activation e é responsável por aplicar a função de ativação, neste caso a ReLU, pixel a pixel.

Na segunda parte do fluxo de entrada, que pode ser visualizada na Figura 3.4, temos uma estrutura que se repete três vezes para valores diferentes de quantidade de filtros (256, 512 e 728), com destaque para as seguintes camadas:

1. a camada *separable_conv2d*, que utiliza a classe SeparableConv2D para realizar a *depthwise separable convolution*, processo diferencial do Xception;

2. a camada *conv2d_1* de convolução tradicional, utilizando a classe Conv2D, cuja saída servirá de residual. A convolução tem de ser tal que a saída tenha o mesmo formato da camada que será somada a ela ao final;

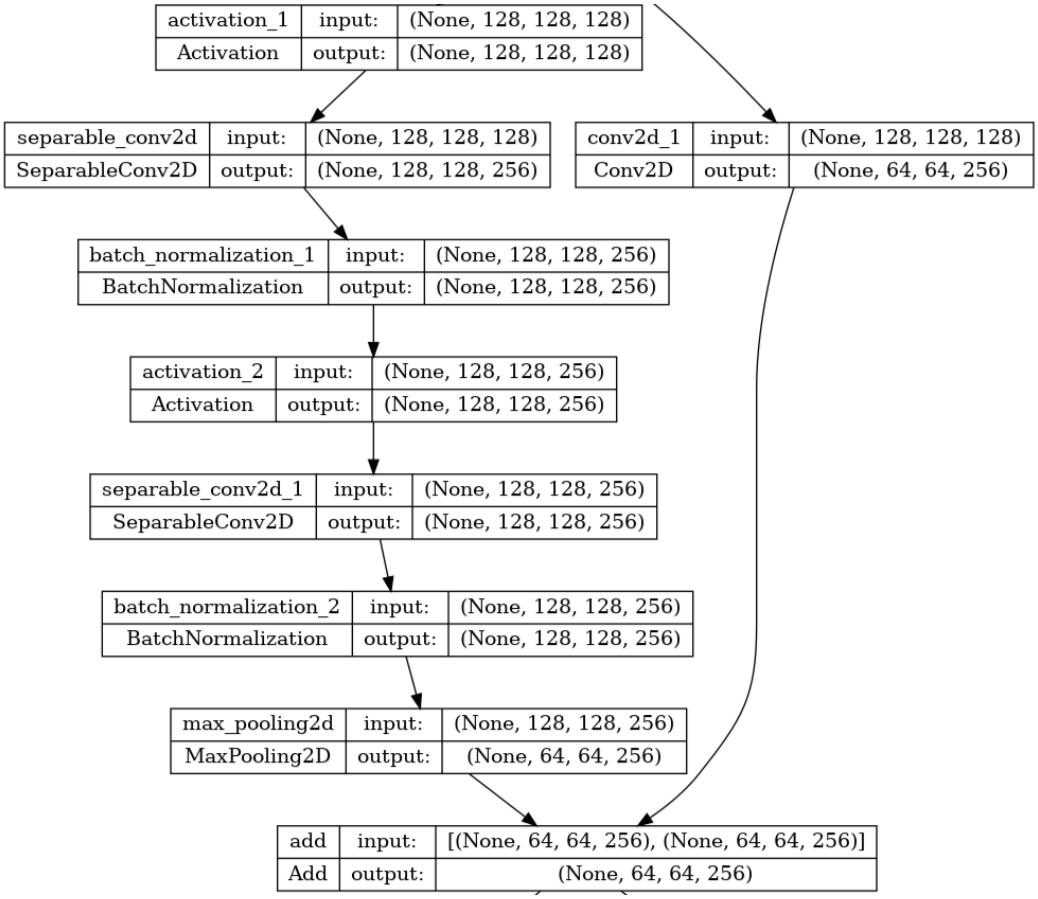


Figura 3.4. Segunda parte do fluxo de entrada do modelo utilizado.

3. a camada de *max pooling*, *max_pooling2d*, que utiliza a classe MaxPooling2D, com o objetivo de reduzir a dimensão dos mapas de características, abstraindo as características mais relevantes;

4. a camada de adição, *add*, que utiliza a classe Add, e vai realizar a soma das abstrações das *depthwise separable convolutions* com o residual.

No fluxo de saída, que pode ser visualizado na Figura 3.5, os mapas de características são preparados para serem avaliados, por meio das seguintes camadas:

1. a camada *global_ average_ pooling2d*, que utiliza a classe GlobalAveragePooling2D e irá reduzir a dimensão de sua entrada para um vetor unidimensional, realizando, como o nome indica, uma média global de cada mapa de característica, retornando apenas um valor para cada;

2. a camada *dropout*, que utiliza a classe Dropout e é responsável por aplicar a

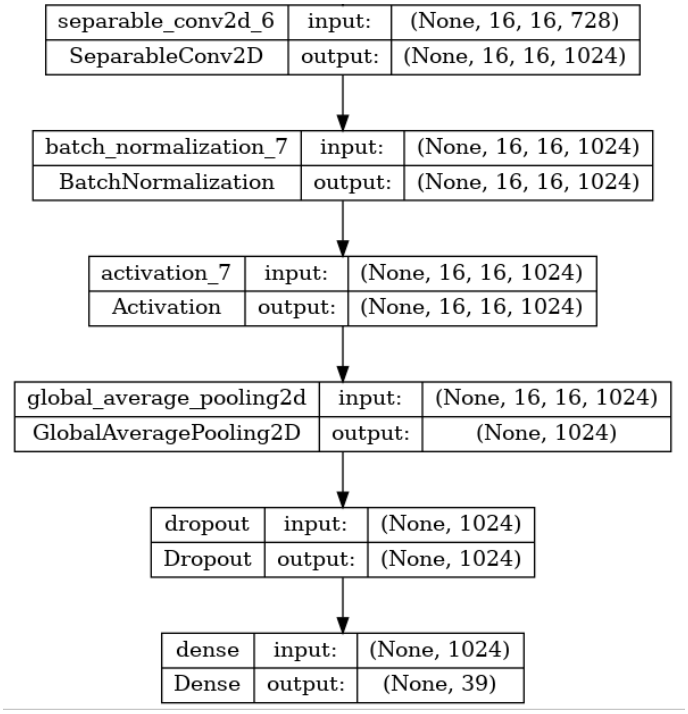


Figura 3.5. Fluxo de saída do modelo utilizado.

regularização de *dropout* para prevenir sobreajuste e melhorar a generalização da rede;

3. a camada *dense*, que utiliza a classe Dense e é reponsável, utilizando uma função de ativação *softmax*, por reduzir a entrada para um vetor com o mesmo tamanho da quantidade total de classes, em que cada valor do vetor corresponde à probabilidade daquela classe ser a correta.

Dada a estrutura do modelo, gostaríamos de definir uma notação para nos referirmos à quantidade de filtros por camada e no total. Podemos observar pelas figuras do modelo que foram utilizados 5 filtros (1 tradicional e 4 *depthwise separable*), portanto, utilizaremos a notação de lista "[128, 256, 512, 728, 1024]" para nos referirmos a estes filtros.

3.3.2 Otimizador e função de custo

Como função de custo, foi utilizada a entropia cruzada categórica. Como otimizador (algoritmo de descida do gradiente), utilizou-se o Adam (KINGMA; BA, 2017). Ele

usa o conceito de momento, de forma que gradientes passados afetam os próximos gradientes e também utiliza a ideia do RMSProp (um outro otimizador) de possuir uma taxa de aprendizado diferente para cada um dos pesos da rede neural, se adaptando aos cenários (permitindo, por exemplo, atualizações menores para características que ocorrem com mais frequência e atualizações maiores para aqueles menos frequentes). Essa combinação permite que o Adam "navegue na superfície" de forma eficiente e convirja rapidamente.

3.3.3 Modelos

Todos os modelos utilizaram um *dropout* de 0,5 e seus outros hiperparâmetros podem ser encontrados na Tabela 3.3. O Modelo 1 foi utilizado como referência, dentro os modelos testados, isto é, partimos dos seus hiperparâmetros para realizar alterações e comparações com outros modelos. Também compararemos os modelos com o da referência bibliográfica (G.; J., 2019), que é a mesma que disponibilizou as imagens e utilizou um modelo mais clássico de CNN, formado por empilhamento de camadas convolucionais.

3.3.4 Avaliação

Para avaliar cada modelo, foram utilizadas três abordagens. As duas primeiras são: a de separar o banco de dados em três conjuntos (treinamento, validação e teste) e a da validação cruzada *k-fold*, foram explicadas no Capítulo 2. A terceira abordagem, na verdade, consistiu no mesmo processo da primeira, mas com o uso de uma outra métrica de cálculo da acurácia: enquanto a primeira abordagem utilizou uma acurácia categórica tradicional, em que se calcula a quantidade de acertos dividida pela quantidade total de previsões, na terceira abordagem, utilizamos a métrica *F1 Score*.

A *F1 Score* utiliza de mais informações para ser calculada: a quantidade de falsos positivos e negativos e a quantidade de verdadeiros positivos. Esta métrica foi incluída, pois foi utilizada pelo artigo de referência (G.; J., 2019); assim, possibilitaremos com-

Modelo	Parâmetros	
(G.; J., 2019)	Taxa de aprendizado	0,01–0,0001
	Épocas	3000
	Dropout	0,2
	Tamanho do mini lote	64
	Com *Augmentation*?	Sim
Modelo 1 (referência)	Taxa de aprendizado	0,001
	Épocas	100
	Tamanho do mini lote	64
	Com *Augmentation*?	Sim
	Filtros	[64, 128, 256, 512, 1024]
Modelo 2	Taxa de aprendizado	0,01
	Épocas	100
	Tamanho do mini lote	64
	Com *Augmentation*?	Sim
	Filtros	[64, 128, 256, 512, 1024]
Modelo 3	Taxa de aprendizado	0,0001
	Épocas	100
	Tamanho do mini lote	64
	Com *Augmentation*?	Sim
	Filtros	[64, 128, 256, 512, 1024]
Modelo 4	Taxa de aprendizado	0,001
	Épocas	100
	Tamanho do mini lote	64
	Com *Augmentation*?	Sim
	Filtros	[128, 256, 512, 728, 1024]
Modelo 5	Taxa de aprendizado	0,001
	Épocas	100
	Tamanho do mini lote	64
	Com *Augmentation*?	Sim
	Filtros	[64, 128, 512, 728, 1024, 2048]
Modelo 6	Taxa de aprendizado	0,001
	Épocas	100
	Tamanho do mini lote	32
	Com *Augmentation*?	Sim
	Filtros	[64, 128, 256, 512, 1024]
Modelo 7	Taxa de aprendizado	0,001
	Épocas	100
	Tamanho do mini lote	64
	Com *Augmentation*?	Não
	Filtros	[64, 128, 256, 512, 1024]

Tabela 3.3. Descrição dos modelos testados e referência da bibliografia. Todos os modelos testados tiveram valor de *dropout* igual a 0,5 e utilizaram o otimizador Adam.

parações mais justas no Capítulo 4.

Para a abordagem da separação em três conjuntos, dividimos o banco nas seguintes proporções: 70% dos dados foram utilizados para treinamento, 20% para validação (nos quais foram realizadas otimizações dos valores de hiperparâmetros) e 10% para teste. Os dados foram previamente embaralhados e utilizou-se o parâmetro *seed* das funções do Keras para garantir o mesmo embaralhamento para todos os modelos. Para se obter a acurácia e o custo finais, ambos os dados de treinamento e validação foram utilizados para o treinamento do modelo final e a as métricas foram calculadas a partir do desempenho da rede para os dados de teste.

Além disso, vale citar, ainda para a abordagem da separação em três conjuntos, que todos os resultados dos modelos são consequência de apenas uma execução do treinamento. Ou seja, por questões de custo computacional, o treinamento não foi executado várias vezes para o mesmo modelo com o objetivo de, ao final, se obter uma média.

Para a abordagem da validação cruzada *k-fold*, utilizamos um k igual a 4, realizamos um embaralhamento do banco e o separamos em apenas dois conjuntos: de treinamento e validação. Realizamos alterações dos hiperparâmetros analisando os resultados de validação, e a acurácia final foi calculada como a média das acurácias dos *folds* do melhor modelo.

Todo o código usado para criação, treinamento e validação do modelo pode ser encontrado em (BOTELHO, 2023c).

3.4 APLICATIVO

Para construção do aplicativo, utilizou-se React Native, com o objetivo de avaliar se a utilização de uma biblioteca híbrida (o que facilita a implementação) é suficiente para a resolução do problema proposto e, consequentemente, mostrar que os custos para construção de um aplicativo para o mundo real, com o mesmo objetivo, pode ser diminuído.

(a) (b)

Figura 3.6. Telas inicial e final do aplicativo em um iPhone.

Seguindo a mesma ideia, para a interface do aplicativo, não houve uma grande preocupação em possibilitar uma ótima "experiência do usuário", uma vez que isto pode ser melhorado depois (com um custo). Assim, na tela principal, que pode ser visualizada na Figura 3.6(a), são apresentadas algumas instruções e dois botões para que o usuário possa selecionar uma imagem na galeria ou tirar uma nova foto.

Com uma imagem selecionada, o aplicativo fará o envio deste arquivo para a Amazon S3, um serviço da Amazon de armazenamento de arquivos na nuvem. Feito o envio com sucesso, é enviada automaticamente uma requisição para a API, cujo modelo será apresentado na próxima seção. Se der tudo certo, o aplicativo receberá de volta uma resposta, em formato JSON, com a classificação e qual a confiança que o modelo tem de que aquela classificação está correta. Um exemplo de como fica a tela do aplicativo

ao final é mostrada na Figura 3.6(b).

Todo o código do aplicativo pode ser encontrado em (BOTELHO, 2023b).

3.5 API

Para a construção da API, foi utilizada a biblioteca Python, o que vai permitir criar um mesmo programa tanto para receber requisições do aplicativo e respondê-las adequadamente, quanto para executar o modelo da rede neural de forma simples, uma vez que este foi criado e salvo utilizando Keras (que, por sua vez, também usa Python).

Para facilitar a escalabilidade da API, seria necessário que uma parte do sistema que executa o código da API ficasse responsável por lidar com as requisições do aplicativo e outra parte, responsável por executar o modelo da rede neural, pois a execução de um modelo pode demorar alguns segundos e, quando muitos usuários (centenas, milhares ou até milhões) tentam realizar classificações de uma só vez, o aplicativo pode se tornar lento.

Uma das estratégias que pode ser utilizada para resolver esta questão é o uso de uma avalição assíncrona do modelo para o dado desejado. Essa abordagem tem muito uso em outras funcionalidades, como envio de e-mails. Quando uma pessoa tenta enviar um e-mail, não é desejável que o usuário espere até que o e-mail realmente chegue até o destinatário, pois esta é uma ação que pode demorar alguns segundos. Assim, ao apertar o botão de enviar, o usuário não fica esperando na mesma tela e já pode realizar outras ações, por exemplo, na sua caixa de entrada. Caso o e-mail falhe na entrega, aí sim o usuário é avisado de que houve um erro e pode tentar novamente.

Nesta aplicação, entretanto, não estamos buscando resolver o problema de escalabilidade e sim mostrar que se pode criar todo um sistema de classificação de imagens de uma forma simples. Portanto, não se implementou nenhuma otimização para avaliação de imagens enviadas pelo usuário: os mesmos recursos são utilizados para lidar com as requisições dos usuários e realizar a classificação. Entretanto, ressalta-se que, pela API possuir poucas funcionalidades, escalá-la não seria uma tarefa muito difícil.

Na API, decidiu-se implementar uma pequena simulação de um banco de dados que vai permitir que sugestões sejam dadas aos usuários, a depender de qual foi a doença identificada pelo sistema. Dizemos uma simulação, pois foi implementado um mapeamento, dentro do código da API, que enviará para o usuário uma resposta pré-definida para cada doença. Esse armazenamento dentro do próprio código não é ideal, uma vez que qualquer mudança necessitaria de uma alteração no código, que é algo que deve ser feito com muito cuidado, pois uma alteração errada pode afetar muitos usuários. Na prática, essas sugestões poderiam ser armazenadas de duas formas:

- por meio de um sistema de gerenciamento de banco de dados relacional, como MySQL, o que diminuiria o tamanho do código e permitiria que as sugestões pudessem ser alteradas sem a necessidade de alteração do código e possibilitaria funcionalidades a mais, como *backup* automático, e uma segurança melhor para os dados;

- por meio de uma outra rede neural, possibilitando sugestões mais efetivas para um determinado usuário, caso este pudesse fornecer ao sistema maiores detalhes sobre sua situação específica, como onde vive (o que poderia alterar sugestões de onde encontrar possíveis produtos) e qual a situação financeira (o que poderia alterar quais produtos comprar), etc..

Ao receber a requisição do aplicativo, utilizamos a biblioteca *requests* de Python para solicitar informações da imagem da Amazon S3, criamos um arquivo e escrevemos as informações da imagem neste arquivo, para criá-la localmente. Com a imagem local, utilizamos a função *load_img* da biblioteca *keras.utils* para carregá-la, redimensionando e cortando as bordas se necessário para forçar as dimensões 256×256, e utilizamos a função *img_to_array*, da mesma biblioteca, para que possamos classificá-la com o modelo. Feita a classificação, o mapeamento mencionado acima é utilizado para se identificar possíveis soluções para lidar com a doença identificada e essas informações são enviadas de volta para o usuário.

Todo o código da implementação da API pode ser encontrado em (BOTELHO, 2023a).

CAPÍTULO 4

RESULTADOS

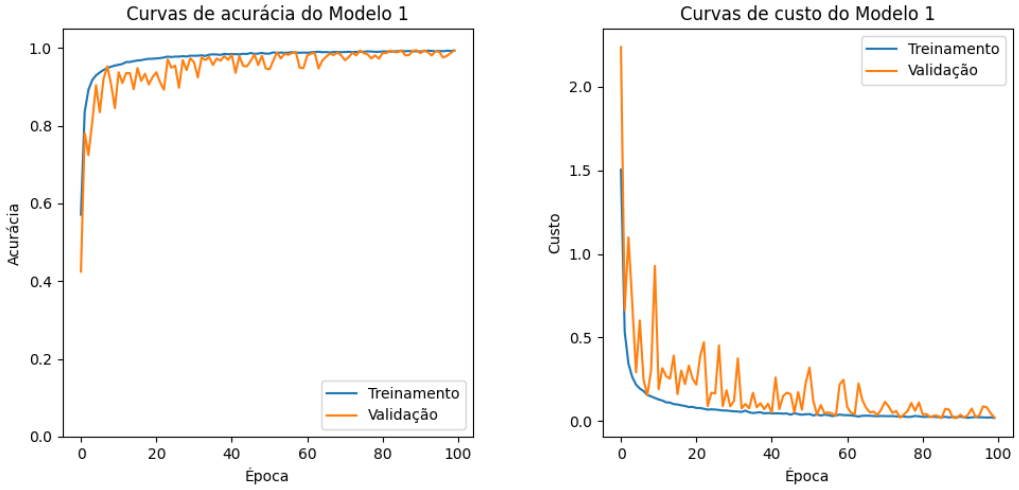
4.1 ABORDAGEM DA DIVISÃO EM 3 CONJUNTOS

Os resultados encontrados para a abordagem da separação dos dados em três conjuntos separados, para os dados de validação e utilizando a métrica tradicional de acurácia categórica, são mostrados na Tabela 4.1. As curvas de acurácia e custo, por época, para os dados de treinamento e validação podem ser encontrados da Figura 4.1 até a Figura 4.7.

Modelo	Acurácia
Modelo 1	99,38%
Modelo 2	91,27%
Modelo 3	98,25%
Modelo 4	97,14%
Modelo 5	99,08%
Modelo 6	99,56%
Modelo 7	99,61%

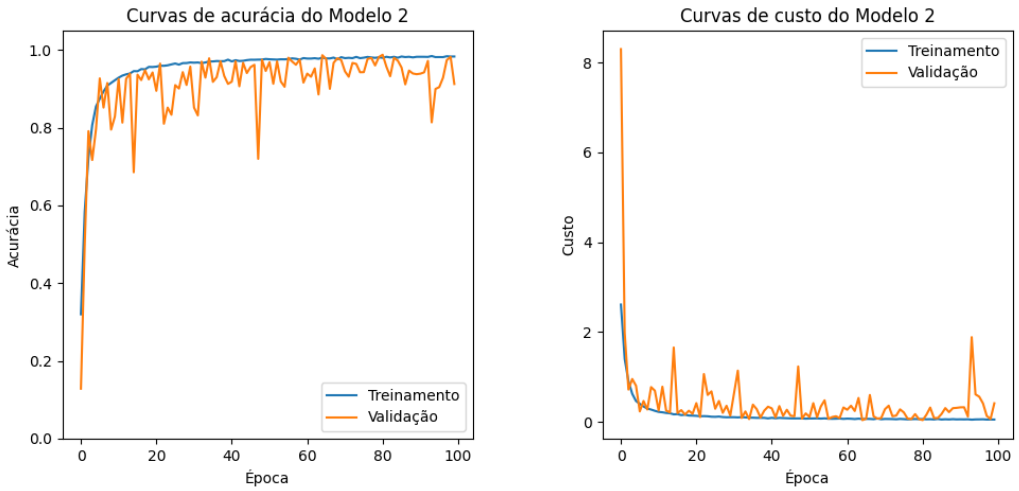
Tabela 4.1. Resultados dos modelos de Redes Neurais Convolucionais testadas para os dados de validação, utilizando a métrica de acurácia categórica tradicional e entropia cruzada categórica como função de custo.

Primeiramente, nota-se que o modelo de melhor acurácia foi o Modelo 7, que não utilizou da técnica de *data augmentation*. Entretanto, notamos que o Modelo 6 obteve uma acurácia muito próxima. Para decidirmos qual foi o melhor modelo de fato, podemos observar as curvas acurácia e custo para os dados de validação de ambos o modelos: na Figura 4.6 para o Modelo 6 e na Figura 4.7 para o Modelo 7.



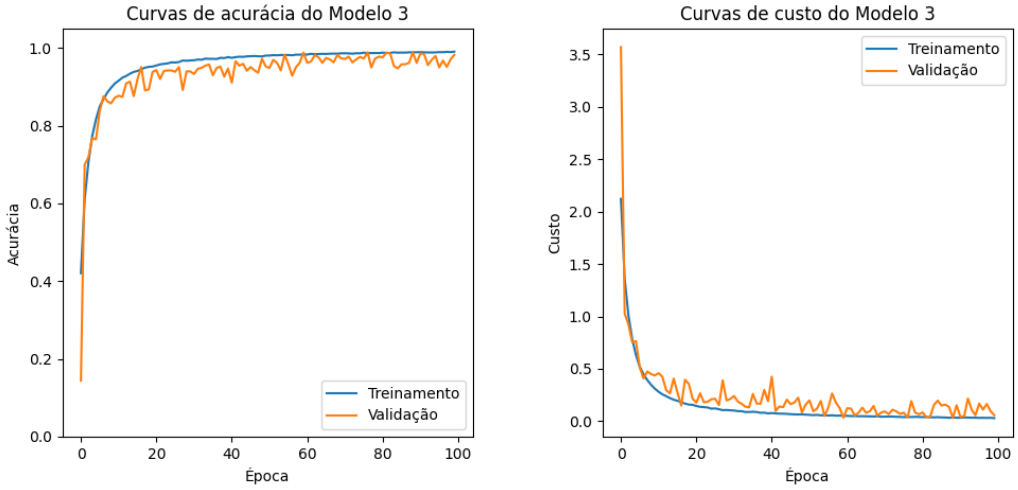
(a) (b)

Figura 4.1. Resultados do treinamento e da validação do Modelo 1.



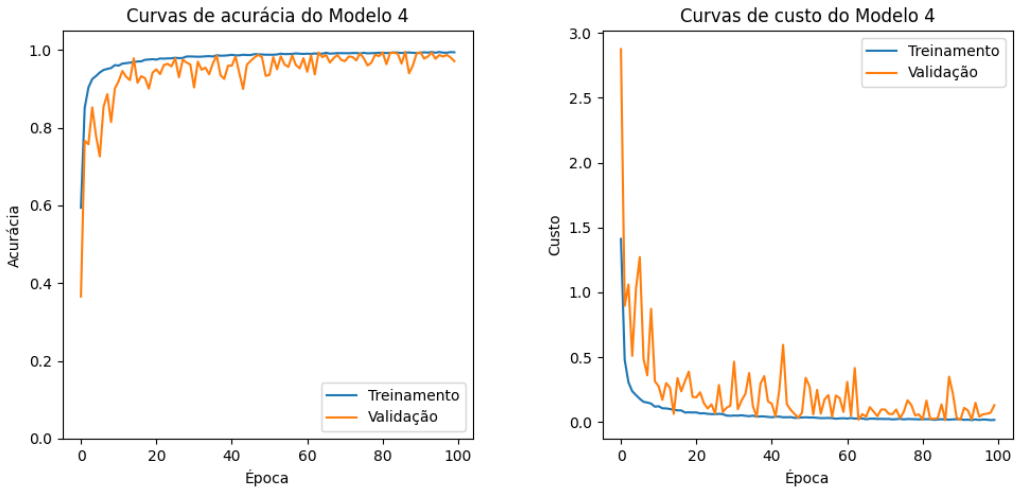
(a) (b)

Figura 4.2. Resultados do treinamento e da validação do Modelo 2.



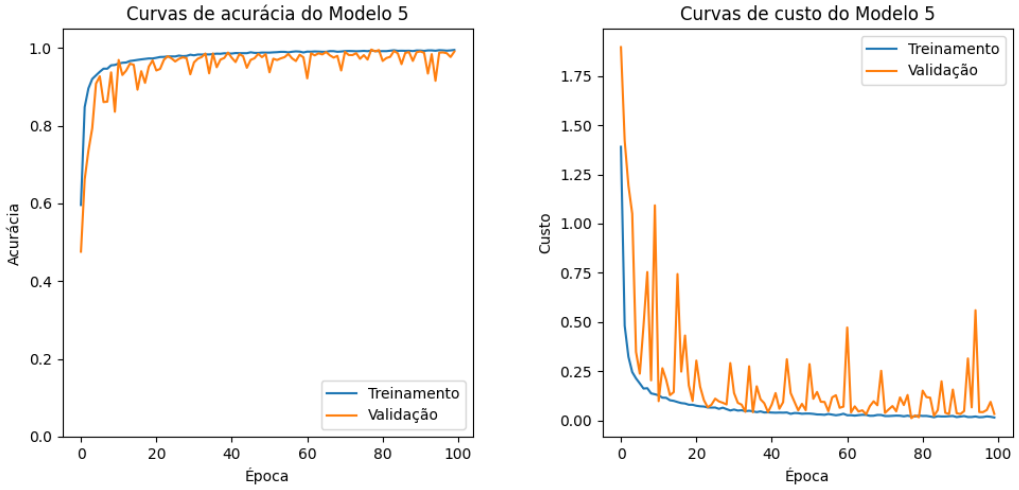
(a) (b)

Figura 4.3. Resultados do treinamento e da validação do Modelo 3.



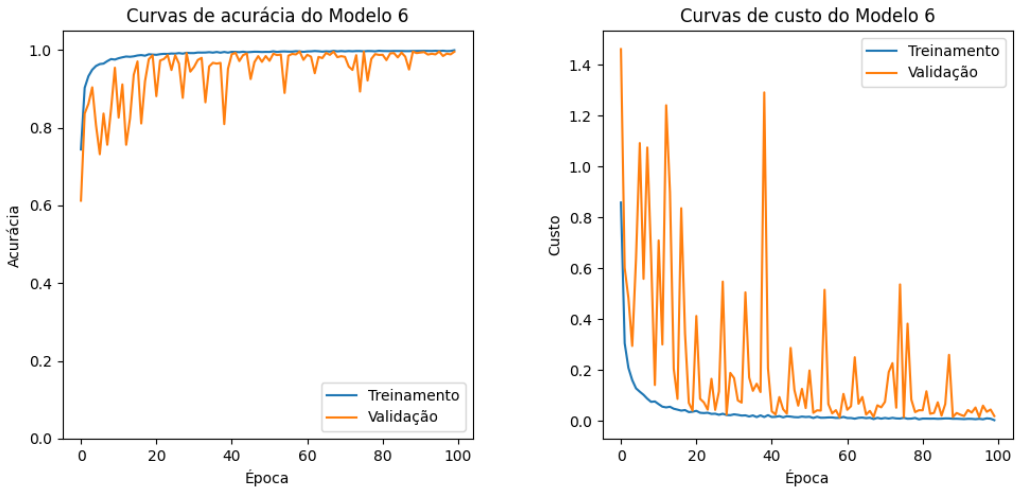
(a) (b)

Figura 4.4. Resultados do treinamento e da validação do Modelo 4.



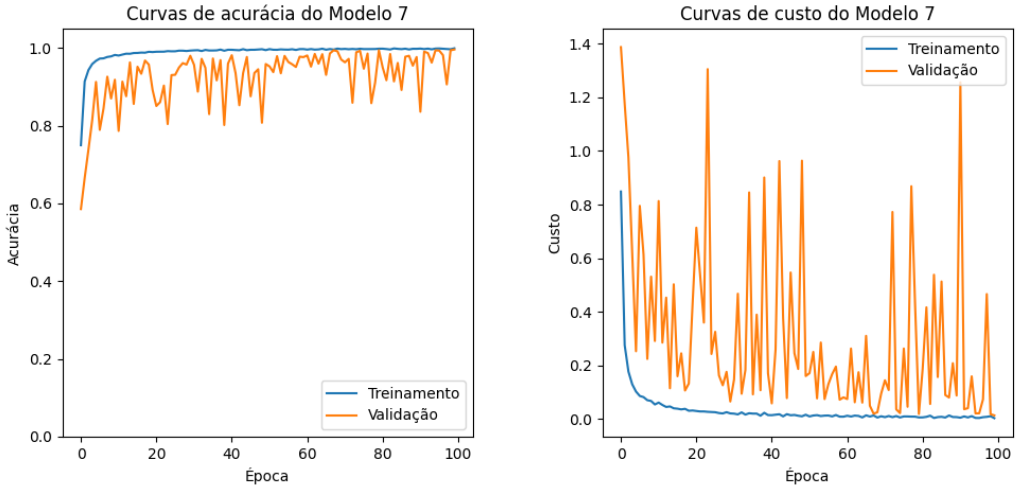
(a) (b)

Figura 4.5. Resultados do treinamento e da validação do Modelo 5.



(a) (b)

Figura 4.6. Resultados do treinamento e da validação do Modelo 6.



(a) (b)

Figura 4.7. Resultados do treinamento e da validação do Modelo 7.

Analisando as figuras, podemos observar que as curvas de acurácia e custo do Modelo 6, para os dados de validação, são mais estáveis que as curvas do Modelo 7, quando o fim das épocas se aproxima. Ou seja, como a quantidade de épocas é um número escolhido, se tivéssemos definido um número um pouco maior que 100, pode ser que a acurácia do Modelo 7 tivesse outro valor, possivelmente pior. No geral, essa instabilidade para os dados de validação indicam uma menor capacidade de generalização. Assim, escolheu-se o Modelo 6 como o modelo de melhor desempenho.

O Modelo 6 obteve um valor de 99,56% para acurácia e sua diferença para o modelo de referência (Modelo 1) foi a alteração do tamanho dos mini lotes de 64 para 32, o que indica que foi benéfico diminuir a quantidade de dados que deve ser considerada para ser dar um passo no algortimo de otimização.

Entretanto, obtiveram-se bons resultados em outros modelos também, como o Modelo 1, com 99,38% de acurácia e o Modelo 5, que obteve 99,08%. O primeiro é o modelo testado de referência e indica que aumentar a quantidade de dados dos mini lotes não causou um grande prejuízo (em relação ao Modelo 6). No segundo, a alteração foi o aumento de camadas e filtros, de 5 (com quantidade de filtros de: [64, 128, 256, 512, 1024]) para 6 (com quantidade de filtros de: [64, 128, 512, 728, 1024, 2048]), o que indica que o algortimo se beneficiou mais de uma estrutura mais simples, o que,

por sua vez, pode indicar que as imagens possuem características menos complexas de se aprender. Outro exemplo que sustenta essa afirmação foi o caso do Modelo 4, em que aumentamos apenas a quantidade de filtros em cada camada (de [64, 128, 256, 512, 1024] para [128, 256, 512, 728, 1024]) e ainda sim a acurácia encontrada foi menor que o modelo de referência.

Dada essa observação, podemos afirmar também que a utilização de uma versão menor do Xception original foi acertada, uma vez que a introdução do fluxo intermediário poderia não trazer tantos benefícios à rede, mas com certeza aumentaria o custo computacional.

O resultado mais baixo foi do Modelo 2, com 91,27% de acurácia, em que aumentamos apenas a taxa de aprendizado de 10^{-3} para 10^{-2}, o que indica que o algortimo não conseguiu se beneficiar de uma taxa maior para a quantidade de épocas utilizadas. Podemos observar o efeito do aumento da taxa de aprendizado na Figura 4.2(a), em que há uma maior variação da acurácia entre épocas e que, até a época 100, a curva ainda varia bastante. Além disso, no Modelo 3, que obteve uma acurácia de 98,25%, diminuímos a taxa de aprendizado para 10^{-4}, o que indica que, fixados os outros parâmetros, a taxa de aprendizado do modelo testado de referência, de 10^{-3}, foi a melhor taxa encontrada.

Como o Modelo 6 apresentou os melhores resultados para o conjunto de validação, utilizamos-no para obter a acurácia final, calculando-a para os dados de teste. O resultado encontrado junto do resultado da referência bibliográfica podem ser encontrados na Tabela 4.2.

Modelo	Acurácia
(G.; J., 2019)	98,15%
Modelo 6	99,35%

Tabela 4.2. Resultado do melhor modelo, o Modelo 6, para o conjunto de teste e resultado da referência bibliográfica.

Assim, nota-se que foi possível obter uma acurácia melhor que a referência, entretanto, nela, os autores utilizaram uma outra métrica para avaliação de desempenho da

rede, a chamada *F1 Score*. Então, assim como mencionado no Capítulo 3, decidiu-se também calcular o desempenho do Modelo 6 utilizando este cálculo, pois é um cálculo mais adequado para bancos de dados desbalanceados e é uma forma mais justa de se comparar com a referência bibliográfica. Os dados obtidos podem ser visualizados na Tabela 4.3.

Modelo	*F1 Score*
(G.; J., 2019)	98,15%
Modelo 6	99,38%

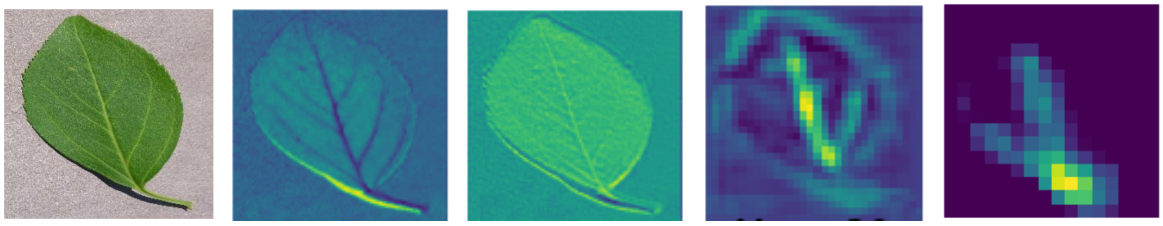
Tabela 4.3. Resultado do melhor modelo, o Modelo 6, para o conjunto de teste e resultado da referência bibliográfica, utilizando a métrica *F1 Score*.

Nota-se, portanto, que foi possível de fato obter um desempenho melhor que a referência, o que mostra o potencial do modelo Xception, mesmo em uma versão menor. Além disso, apesar dos autores da referência não citarem explicitamente, existe uma possibilidade do treinamento do Modelo 6 possuir menor custo computacional, uma vez que eles realizaram um treinamento com 3000 épocas, enquanto que o modelo mencionado foi treinado por apenas 100 (duração de 3 horas).

Uma outra observação é que utilizamos uma separação diferente dos dados: enquanto a referência bibiográfica possuía 1950 dados para teste, possuíamos 5545, ou seja, treinamos com menos dados e possuíamos mais dados para realizar a avaliação da acurácia final e uma acurácia boa, nesse contexto, indica uma boa capacidade de generalização para dados não vistos pela rede.

4.2 MAPAS DE CARACTERÍSTICAS E MATRIZ DE CONFUSÃO

Uma visualização interessante de se obter de uma CNN são os mapas de características das camadas ocultas. Elas dão uma maior noção do que está acontecendo internamente na rede e, de certa forma, mostram o que a rede está entendendo como características das imagens. A Figura 4.8 mostra os mapas de características para diferentes camadas da rede, dada uma entrada de uma imagem de uma folha saudável.



(a) (b) (c) (d) (e)

Figura 4.8. Exemplo de mapas de características para diversas camadas ocultas. Na Figura 4.8(a), a imagem original, na Figura 4.8(b) um exemplo de mapa de uma camada com 64 filtros (mais exemplos na Figura 4.9), na Figura 4.8(c) um exemplo de mapa de uma camada com 128 filtros (mais exemplos na Figura 4.10), na Figura 4.8(d) um exemplo de mapa de uma camada com 512 filtros (mais exemplos na Figura 4.11) e na Figura 4.8(b) um exemplo de mapa de uma camada com 1024 filtros (mais exemplos na Figura 4.12).

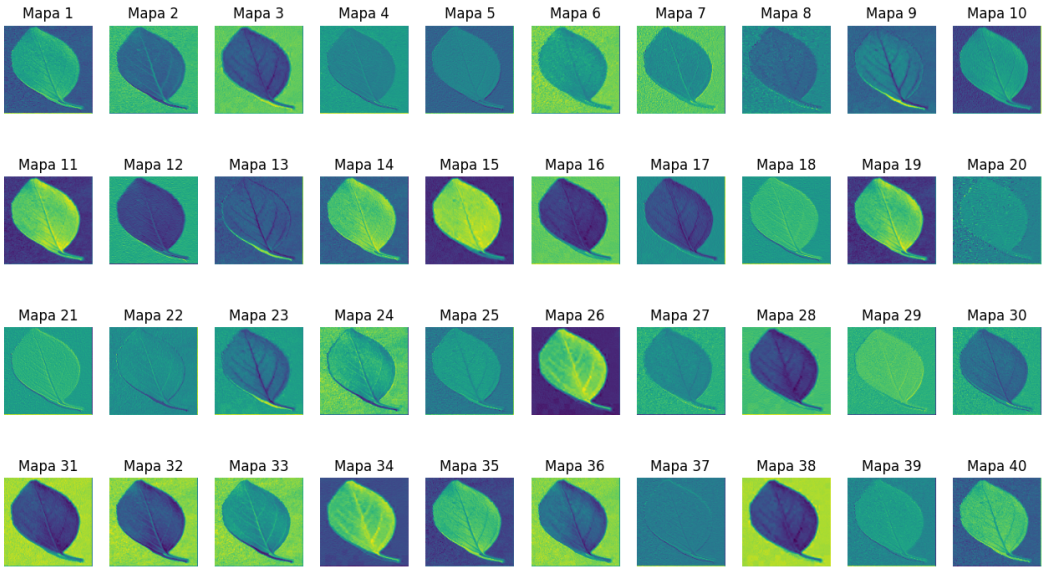


Figura 4.9. Mapas de características para a camada de 64 filtros.

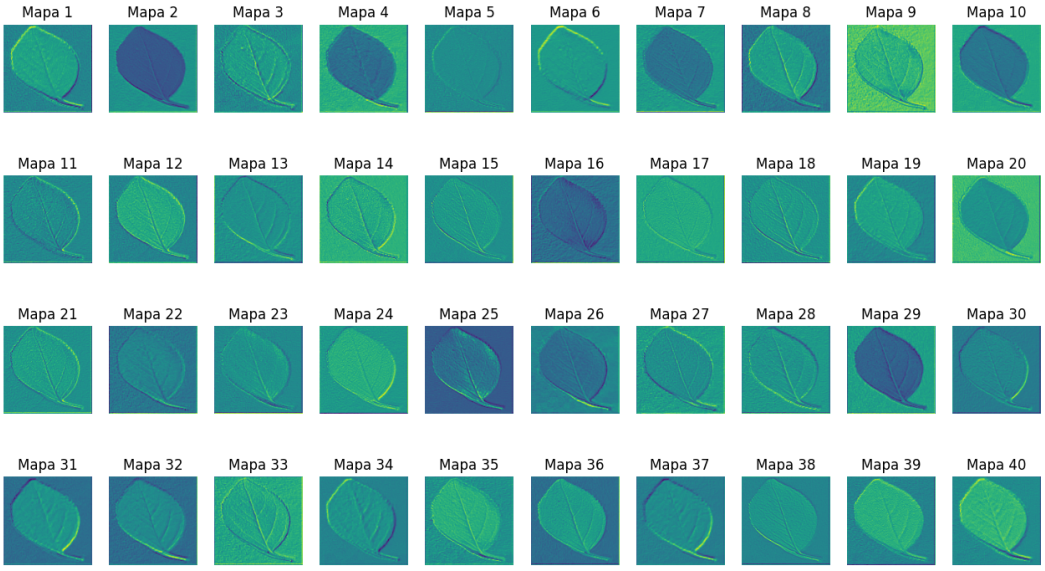


Figura 4.10. Mapas de características para a camada de 128 filtros.

Assim como esperado pela teoria, podemos observar que, com o aumento da quantidade de filtros utilizados em uma convolução e quanto mais avançamos nas camadas ocultas, mais abstratos ficam os mapas, destacando características mais específicas. Ou seja, enquanto que na Figura 4.8(b) os 64 filtros não modificaram tanto a imagem original e conseguimos visualizar o destaque de algumas linhas, na Figura 4.12 não é possível dar um sentido concreto para o que a rede está destacando.

Como um exemplo de uma folha com alguma patologia, a Figura 4.13 mostra os mapas de características para diferentes camadas da rede para um exemplo de folha de uma planta doente. Nela, podemos fazer as mesmas observações acima, mas é interessante notar, pelas Figuras 4.13(b) e 4.13(c), que de fato a rede destaca os pontos de ação do agente causador da doença (pontos amarronzados na folha original).

Além dos mapas de características, a matriz de confusão, calculada a partir dos dados de teste e mostrada na Figura 4.18, também pode providenciar um entendimento melhor do resultado da rede, uma vez que mostra quais categorias tiveram uma maior ou menor taxa de acerto, mostrando a quantidade de falsos positivos e negativos e verdadeiros positivos e negativos, isto é, ela possibilita entender quais classes mais confundem a rede. No nosso modelo, como obtivemos um bom F1 Score, métrica que depende das quantidades mencionadas, não há muito o que tirar de conclusão da

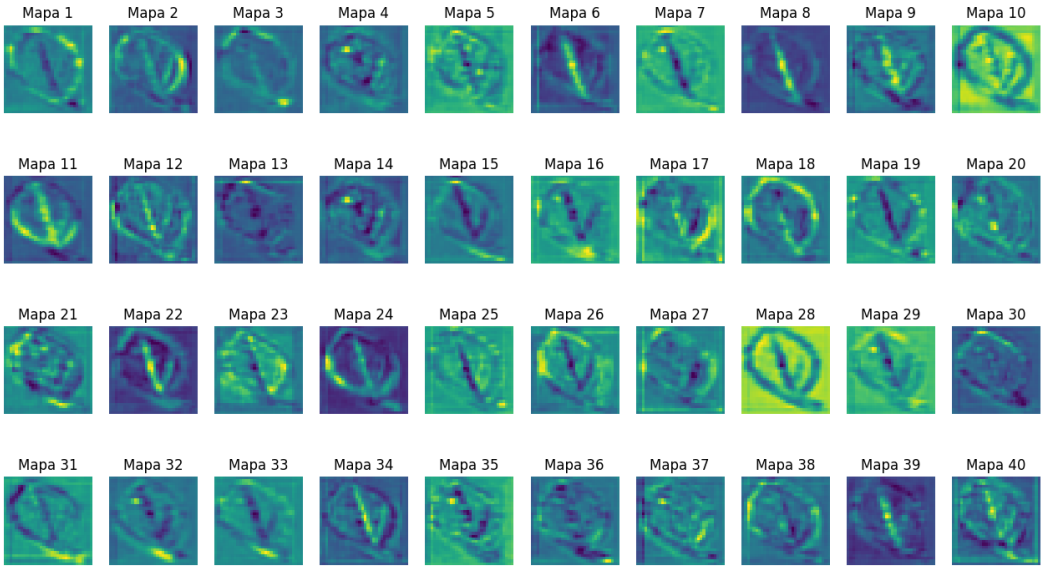


Figura 4.11. Exemplos de mapas de características para a camada de 512 filtros.

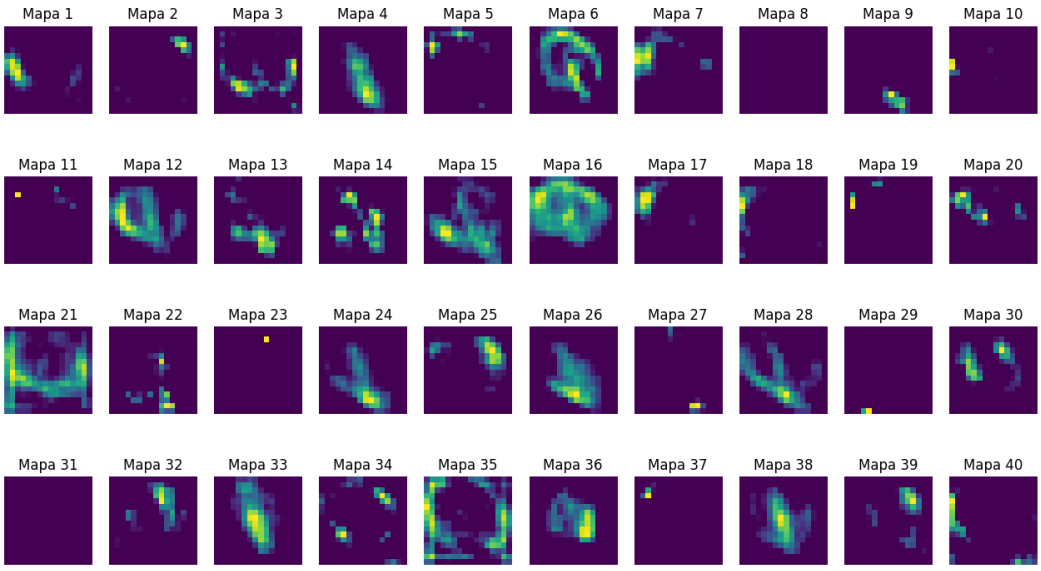
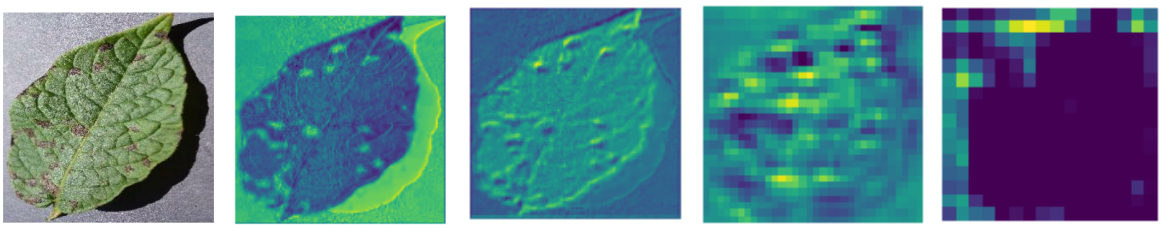


Figura 4.12. Exemplos de mapas de características para a camada de 1024 filtros.



(a) (b) (c) (d) (e)

Figura 4.13. Exemplo de mapas de características para diversas camadas ocultas. Na Figura 4.8(a), a imagem original, na Figura 4.13(b) um exemplo de mapa de uma camada com 64 filtros (mais exemplos na Figura 4.14), na Figura 4.13(c) um exemplo de mapa de uma camada com 128 filtros (mais exemplos na Figura 4.15), na Figura 4.13(d) um exemplo de mapa de uma camada com 512 filtros (mais exemplos na Figura 4.16) e na Figura 4.13(b) um exemplo de mapa de uma camada com 1024 filtros (mais exemplos na Figura 4.17).

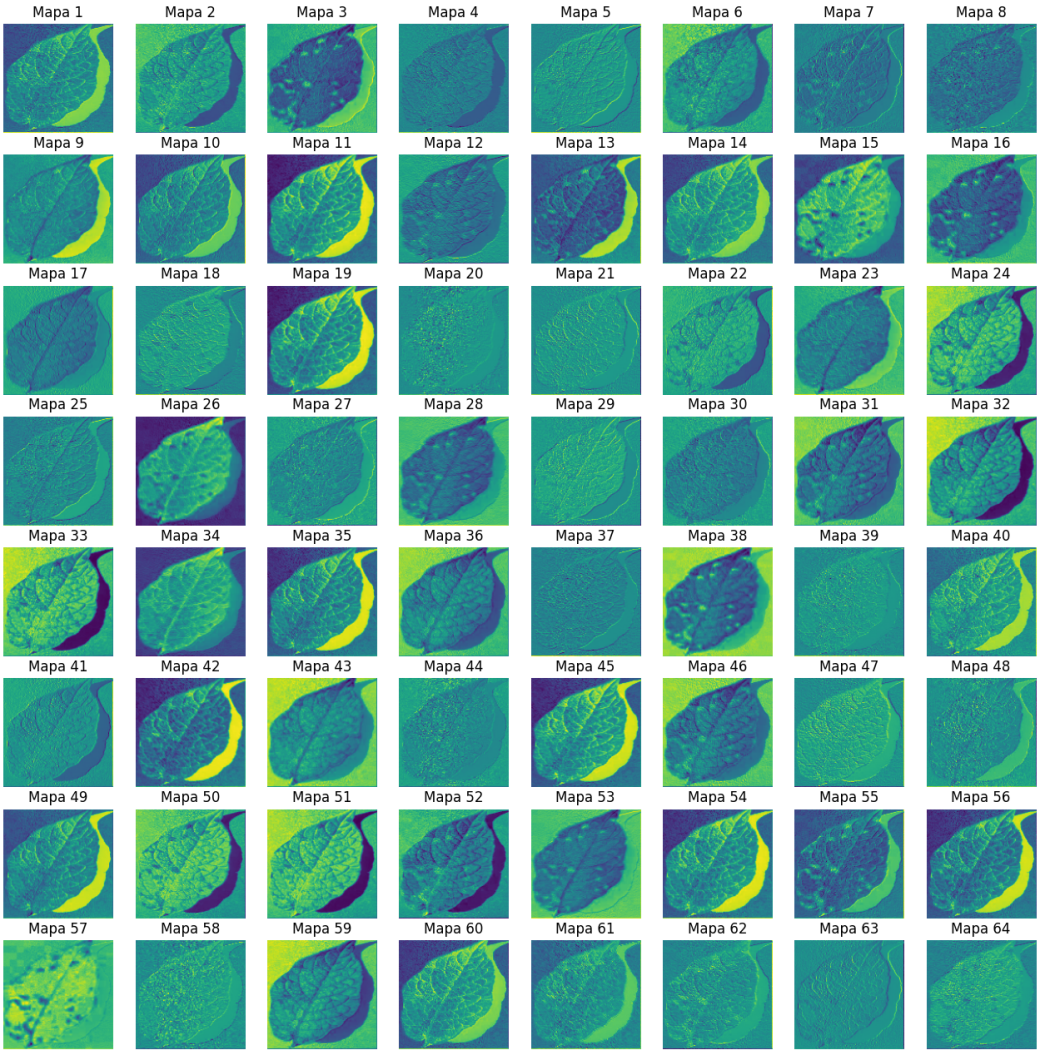


Figura 4.14. Mapas de características para a camada de 64 filtros.

matriz, exceto de que todas as classes tiveram um bom desempenho.

Em suma, por um lado, podemos entender os resultados encontrados como uma prova de que o modelo Xception pode obter resultados muito bons; por outro, surgiu uma dúvida se a divisão do banco (ou a quantidade usada para cada conjunto) em três conjuntos separados pode ter influenciado negativamente os resultados, já que, para algumas classes, a quantidade de exemplos era escassa, como pode ser visto na Tabela 3.2. Como um exemplo de pior caso, como usamos uma divisão de 10% para o conjunto de teste e temos no total 152 exemplos para a classe **Batata saudável**, o conjunto de testes possui apenas 15 imagens para validar o modelo.

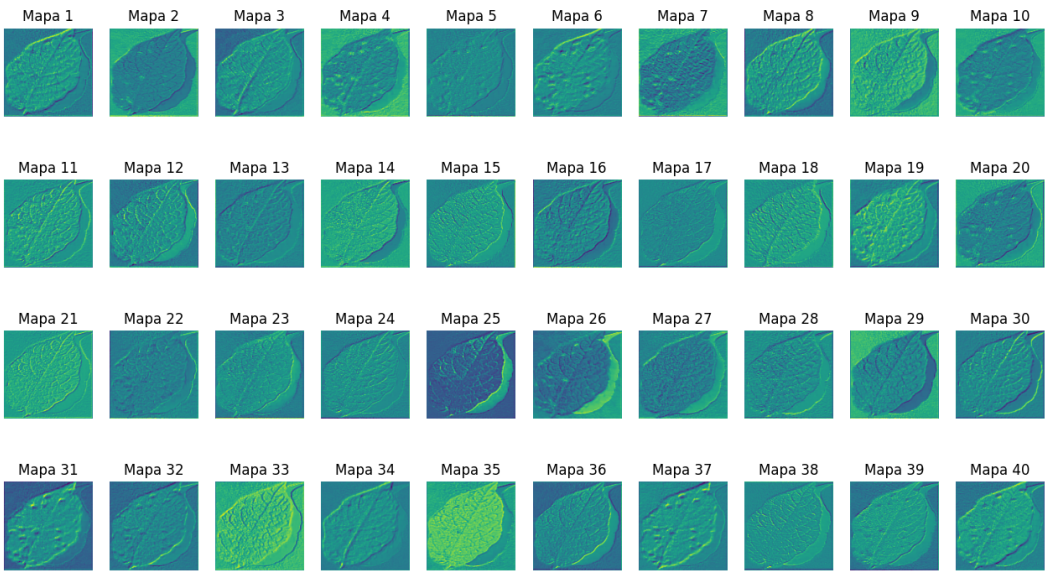


Figura 4.15. Mapas de características para a camada de 128 filtros.

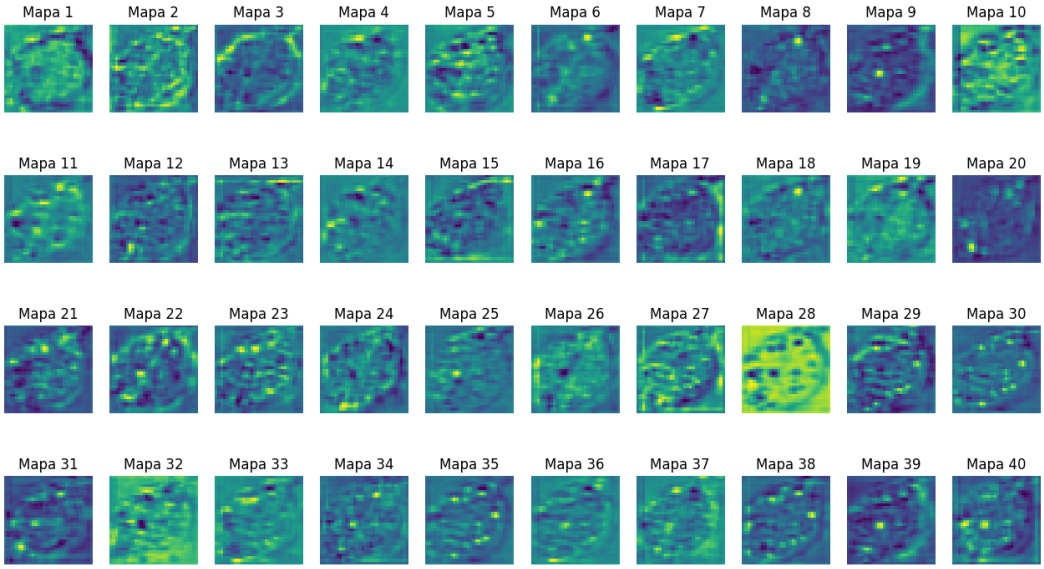


Figura 4.16. Exemplos de mapas de características para a camada de 512 filtros.

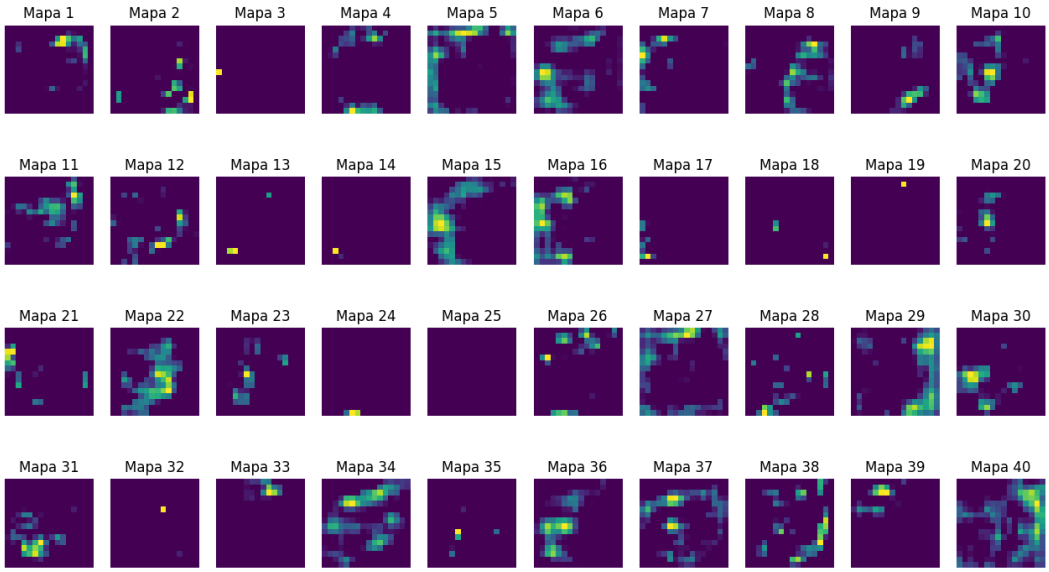


Figura 4.17. Exemplos de mapas de características para a camada de 1024 filtros.

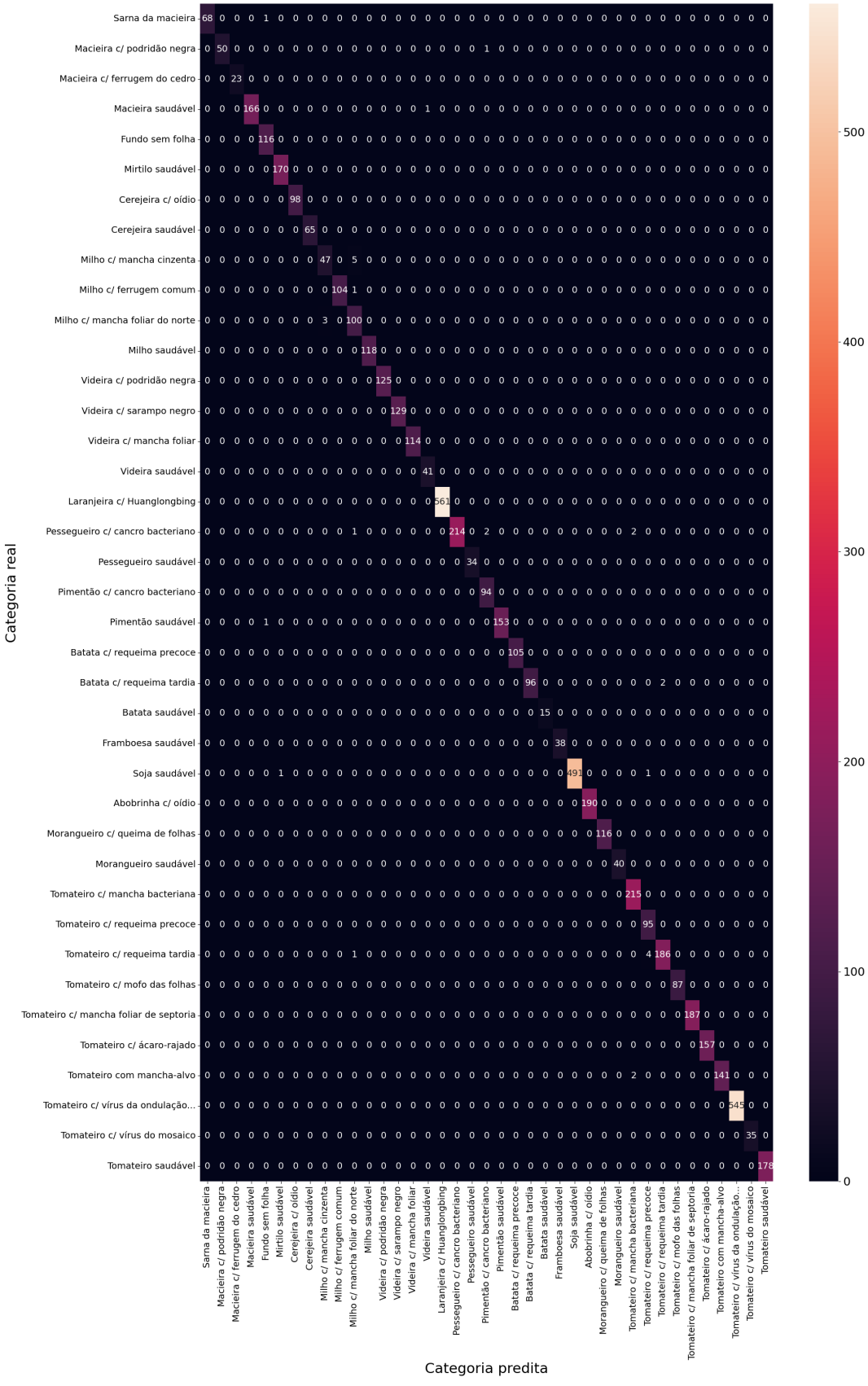


Figura 4.18. Matriz de confusão do Modelo 6.

4.3 ABORDAGEM DA VALIDAÇÃO CRUZADA *K-FOLD*

Decidiu-se, portanto, realizar também uma avaliação de um dos modelos utilizando a validação cruzada *k-fold* para verificação de alguma anomalia. Os resultados encontram-se na Tabela 4.4. As curvas de acurácia e custo, para cada **fold**, para os dados de treinamento e validação podem ser encontrados no Apêndice A.

Como podemos observar pelos resultados, foram observadas algumas alterações nas acurácias de alguns modelos. Aqui, os modelos de melhor desempenho foram os Modelo 1 e 3, que obtiveram acurácias de 99,32% e 99,30%, respectivamente, indicando que a diminuição da taxa de aprendizado de 10^{-3} para 10^{-4} (no Modelo 3) não teve tanto efeito. O Modelo 6, que havia previamente obtido o melhor desempenho, teve sua acurácia diminuída para 98,13%, o que ainda não é um resultado insatisfatório.

Esta abordagem, entretanto, também não é perfeita. Ao analisarmos as curvas de evolução da acurácia e custo ao longo das épocas, no Apêndice A, da Figura 5.1 até a Figura 5.6, observamos que há uma variação considerável, dentro dos dados de validação, até o fim do treinamento, para a maioria dos modelos. Este comportamento indica uma instabilidade dos modelos e uma capacidade de generalização inconsistente.

Este problema pode ter diversas causas. Uma delas é a escolha para o valor de k. Usando um valor de k igual a 4, diminuímos a quantidade de dados para treinamento em 25%, diminuindo a quantidade de dados que a rede pode utilizar para adaptar seus parâmetros. Aumentando-se o valor de k, não só poderíamos observar uma maior estabilidade da rede, mas também com mais um *fold* para adicionarmos no cálculo da média final, menor seria o efeito da instabilidade de cada *fold*. Outra causa é a quantidade de épocas utilizadas: se deixássemos a rede aprender por mais tempo, poderíamos verificar uma maior estabilidade.

Essas soluções, entretanto, não foram estudadas neste trabalho, uma vez que todo o procedimento da abordagem de validação cruzada *k-fold* é muito custosa computacionalmente e as soluções pensadas aumentariam ainda mais este custo e não houve tempo suficiente para isto.

O Modelo 3, no entanto, e diferentemente do Modelo 1, apresentou curvas bem

Modelo	Acurácia
Modelo 1 - *fold 0*	99,48%
Modelo 1 - *fold 1*	99,37%
Modelo 1 - *fold 2*	99,29%
Modelo 1 - *fold 3*	99,12%
Modelo 1 - Final	99,32%
Modelo 2 - *fold 0*	92,59%
Modelo 2 - *fold 1*	98,57%
Modelo 2 - *fold 2*	98,23%
Modelo 2 - *fold 3*	93,79%
Modelo 2 - Final	95,79%
Modelo 3 - *fold 0*	98,87%
Modelo 3 - *fold 1*	99,15%
Modelo 3 - *fold 2*	99,58%
Modelo 3 - *fold 3*	99,59%
Modelo 3 - Final	99,30%
Modelo 4 - *fold 0*	98,73%
Modelo 4 - *fold 1*	99,37%
Modelo 4 - *fold 2*	80,45%
Modelo 4 - *fold 3*	98,15%
Modelo 4 - Final	94,17%
Modelo 5 - *fold 0*	97,62%
Modelo 5 - *fold 1*	99,41%
Modelo 5 - *fold 2*	99,25%
Modelo 5 - *fold 3*	98,73%
Modelo 5 - Final	98,75%
Modelo 6 - *fold 0*	99,29%
Modelo 6 - *fold 1*	98,80%
Modelo 6 - *fold 2*	96,85%
Modelo 6 - *fold 3*	97,60%
Modelo 6 - Final	98,13%
Modelo 7 - *fold 0*	99,11%
Modelo 7 - *fold 1*	98,60%
Modelo 7 - *fold 2*	98,71%
Modelo 7 - *fold 3*	98,68%
Modelo 7 - Final	98,78%

Tabela 4.4. Resultados dos modelos testados, utilizando validação cruzada *k-fold*, com $k = 4$.

estáveis, para todos os *folds*, como se pode observar no Apêndice, pelas Figura 5.3 e 5.1. Como o resultado do Modelo 6, na abordagem anterior, é bem parecido com o do Modelo 3, com acurácias de 99,35% e 99,30%, respectivamente, e como não foi possível identificar nenhuma anomalia presente nos procedimentos, decidimos selecionar o Modelo 6 como o modelo a ser utilizado para testes reais do aplicativo, uma vez que ele foi validado com uma referência bibliográfica.

4.4 TESTE DO APLICATIVO

Terminada a avaliação, carregamos o Modelo 6 na API e realizamos alguns testes. Primeiramente, decidimos validar o aplicativo e o processo de tirar uma foto. Assim, imprimimos as imagens do próprio banco de dados utilizado para treinamento e tiramos fotos delas. Exemplos podem ser encontrados na Figura 4.19.

Como podemos observar, os resultados não foram satifatórios. Apesar de a rede ter acertado alguns casos, ela errou outros. Uma das possíveis razões para esta incosistência está justamente no processo de impressão, que possui suas imperfeições para produzir a mesma imagem que vemos na tela do computador. Entretanto, assim como o ser humano não teria problemas em discernir uma imagem vista de uma tela de uma vista em uma impressão, esperava-se, a princípio, que a rede também não tivesse. Entretanto, o processo de impressão gera imagens com "imperfeições"que são visíveis para um celular, mas não são visíveis para o olho humano. Assim, em um segundo raciocínio, esse resultado é esperado.

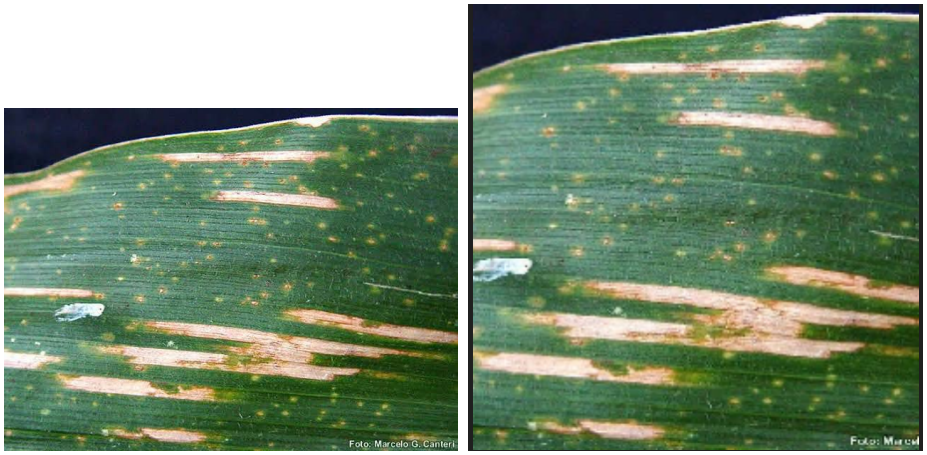
Partimos, então, para uma nova abordagem, para verificação de algum erro na implementação do aplicativo: adicionamos as mesmas imagens (originais, sem tirar foto e sem impressão) diretamente na galeria do celular e realizamos a análise. Os resultados podem ser encontrados na Figura 4.20.

Neste caso, podemos verificar que a rede acertou todos os casos, indicando que a implementação do aplicativo não parece ter problemas. Validado este caso, realizamos o mesmo processo, mas para imagens novas, disponíveis na internet. Os resultados podem ser encontrados na Figura 4.21.

Figura 4.19. Imagens de treinamento e resultados de testes realizados com impressão. Em 4.19(a), o resultado correto seria o de **Sarna da macieira**; em 4.19(b), o resultado correto seria o de **Macieira com ferrugem do cedro**; em 4.19(c), o resultado correto seria o de **Macieira com podridão negra**; em 4.19(d), o resultado foi correto; em 4.19(e), o resultado foi correto; em 4.19(f), o resultado correto seria o de **Mirtilo sadio**.

Figura 4.20. Imagens de treinamento e resultados de testes realizados sem impressão. Todos os resultados estão corretos.

Figura 4.21. Imagens da internet e resultados dos testes realizados. A classificação correta para todos os casos é a de **Milho com mancha cinzenta**.



(a) (b)

Figura 4.22. Exemplo de alteração em uma imagem realizada para que sirva de entrada para a rede.

Neste caso, observamos que os resultados ainda são inconsistentes. Nota-se que a rede acertou para imagens com maior contexto ao redor da planta, com mais detalhes, como nas Figuras 4.21(a), 4.22(a) e 4.21(c), enquanto que a rede errou para algumas imagens que foram tiradas muito próximas da folha, como na Figura 4.21(d) e 4.21(e).

Dado isso, adicionamos um algoritmo para possibilitar verificarmos qual a imagem que a rede realmente está "vendo". Um exemplo pode ser verificado na Figura 4.22. Nela, podemos ver que os cortes realizados podem realmente prejudicar a classificação, já que a imagem cortada parece ter sido tirada de mais próximo ainda, isto é, a imagem cortada suprime alguns detalhes que podem ser importantes para a avaliação. Os cortes acontecem quando a imagem não é quadrada, uma vez que imagens quadradas são apenas redimensionadas para o tamanho 256x256 (necessário para servir de entrada para a rede), enquanto imagens não quadradas não podem ser simplesmente redimensionadas sem distorção e necessitam destes cortes.

Uma outra fonte de erros é a possibilidade de as imagens possuirem classificações originais erradas. As imagens utilizadas foram obtidas por meio de uma pesquisa simples no Google da planta com o nome do agente causador da doença (e não o nome traduzido, uma vez que este pode ser fonte de outros problemas) e, assim, poderiam conter erros. Entretanto, essa validação não foi feita, e decidimos partir para uma outra abordagem, final, com o objetivo de evitar este e o erro do corte.

Na abordagem final, conseguimos, junto à Faculdade de Agronomia e Medicina

Veterinária, alguns exemplos de imagens de plantas com as doenças treinadas. Para isso, fomos até a Fazenda Água Limpa da Universidade de Brasília e tiramos fotos de algumas plantas. Tentamos, aqui, tirar fotos quadradas para evitar o problema observado anteriormente. Os resultados podem ser vistos na Figura 4.23.

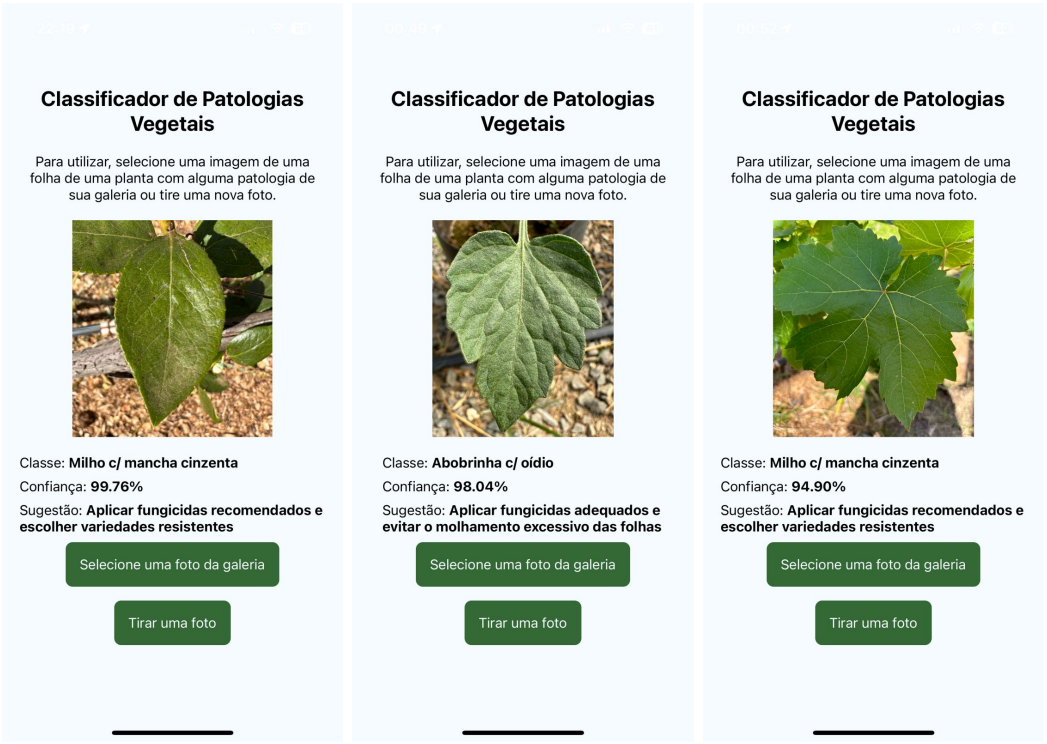
Notamos que os resultados das fotos tiradas também não foram satisfatórios. Na verdade, a rede não conseguiu realizar nenhuma previsão correta. Este resultado não pode ser atribuído a pequenos detalhes, mas sim a algum problema mais fundamental do treino da rede neural.

A explicação mais plausível pode ser entendida ao compararmos alguns exemplos de imagens fornecidas para o treinamento, na Figura 3.2, e as utilizadas para este teste, na Figura 4.23. As imagens do treinamento possuem folhas que foram retiradas das plantas e colocadas em um fundo cinza (algumas tiveram o fundo completamente retirado), enquanto as fotos da fazenda foram tiradas diretamente da planta, com fundos variados. Essa padronização das imagens de teste, portanto, pode não ter fornecido a generalização suficiente para que a rede conseguisse classificar corretamente no uso pretendido para o aplicativo, isto é, com o usuário utilizando-o diretamente no campo, de forma fácil e simples, sem a necessidade de retirar a folha e tirar a foto em um ambiente controlado.

O possível problema no corte das imagens foi testado para essas imagens da fazenda também, mas não foi possível identificar cortes problemáticos. Assim, o problema de generalização provavelmente tem muito mais influência na inconsistência dos resultados anteriores também.

Decidimos, assim, ir à campo mais uma vez, desta vez na Estação Experimental de Biologia da UnB e coletar algumas folhas para que pudéssemos fazer algumas análises com o fundo e as imagens podem ser visualizadas na Figura 4.24. Como podemos observar, os resultados foram inconsistentes, mas indicam o que afirmamos anteriormente: como a alteração de fundo causou uma diferença na avaliação, as imagens do banco de dados não forneceu uma generalização boa o suficiente para o uso do aplicativo.

Por questões de tempo, não foi possível realizar procedimentos com o objetivo de



(a) (b) (c)

(d) (e)

Figura 4.23. Exemplos de resultados para a avaliação do aplicativo na Fazenda Água Limpa. Todos as previsões foram erradas.

Figura 4.24. Exemplos de resultados para a avaliação do aplicativo na Estação Experimental de Biologia da UnB. Todos os exemplos deveriam fornecer a classe **Mirtilo saudável** como correta.

solucionar este problema; contudo, gostaríamos de mencionar duas possiblidades de solução:

- uso de processamento de imanges nas fotos do banco e nas fotos tiradas pelo aplicativo para se retirar o fundo (possivelmente muito difícil, dada a diversidade de fundos);
- criação de um banco de dados próprio ou adição no banco existente de imagens retiradas diretamente no campo, sem nenhum tratamento especial.

A criação de um banco de dados próprio seria a abordagem com maior chance de solucionar o problema. Entretanto, deve-se levar em consideração que a coleta e processamento de dados pode ser um processo bastante demorado e custoso, o que prejudica a ideia de deixar baixos os custos da solução completa.

Por fim, vale notar que, por uma avaliação subjetiva, o desempenho do aplicativo é bastante satisfatório, o que é uma consequência de sua simplicidade, mas favorece a tese de que a tecnologia híbrida utilizada pode ser utilizada.

4.5 LIMITAÇÕES EXPERIMENTAIS

Apesar de ter sido utilizada uma GPU com uma boa nota para treinamento de redes neurais, é comum na literatura o uso de múltiplas GPUs. O Xception do artigo original, por exemplo, foi treinado com 60 GPUs NVIDIA K80 (CHOLLET, 2017).

Consequentemente, alguns hiperparâmetros não puderam assumir quaisquer valores, são eles: a quantidade de camadas, o tamanho dos mini lotes e a quantidade de filtros usadas nas convoluções. Dado que na literatura é consenso que aumentar esses parâmetros percebe-se um aumento da acurácia (até que haja sobreajuste), exceto o tamanho dos mini lotes, tentou-se aumentar esses parâmetros, mas em um determinado ponto, não era possível executar o comando de treinamento, por falta de memória.

Tudo isso se deu pelo fato de que o treinamento foi realizado em uma máquina local e não e um servidor remoto. Tentou-se utilizar o Google Colab, entretanto, os créditos

do plano mensal de R$ 58,00 rapidamente eram consumidos e não seria possível realizar várias tentativas de treinamento, em caso de alteração dos hiperparâmetros e possíveis erros.

Outra limitação experimental foi em relação aos dados. Como pode ser observado pela Tabela 3.2, a quantidade de espécies de plantas é limitada e existem algumas espécies com poucas imagens em comparação com outras: enquanto temos 5090 imagens de **Soja saudável**, temos somente 152 imagens de **Batata saudável**. Isto representa dois problemas: o desbalanceamento dos dados e a pouca quantidade de dados para certas classes.

Tentou-se mitigar esses problemas utilizando uma função de custo (*categorical cross-entropy*), uma métrica (*F1 Score*) e uma técnica (*data augmentation*) apropriadas, mas como não foi realizada uma comparação profunda sem a aplicação dessas técnicas em conjunto, é difícil dizer o quanto que realmente ajudaram.

Uma outra limitação aconteceu com o aplicativo. Inicialmente, o desejo era que o aplicativo pudesse ser utilizado de forma offline, uma vez que o acesso à internet nas zonas rurais é difícil e inconsistente. Entretanto, as tecnologias disponíveis para treinamento de redes neurais estão mais avançadas para utilização na linguagem Python e, como aplicativos para smartphones utilizam outras linguagens, como linguagens específicas para Android (Java, Kotlin) e para iOS (Swift) ou como uma linguagem multiplataforma (JavaScript), não foi possível essa implementação. Uma biblioteca mais recente do próprio TensorFlow, chamada TensorFlow.js, tem feito avanços consideráveis para permitir modelagem e treinamento de redes neurais em um navegador, mas para aplicativos de celular os recursos ainda são escassos.

Um outro desejo inicial era de que pudéssemos criar um aplicativo que tivesse uma maior facilidade de identificação de espécies de vegetações mais presentes no Brasil. Com este objetivo, foi feito contato com a Embrapa e com a Reflora (REFLORA, 2010), um programa criado pelo Governo Federal em 2010 com o objetivo de "resgate, através de imagens em alta resolução, de espécimes da flora brasileira depositados em herbários estrangeiros, para disponibilização ampla e irrestrita no Herbário Virtual Reflora".

A Embrapa está criando um banco de dados com imagens de plantas saudáveis e doentes, porém ainda não está disponível para o público. O banco de dados utilizado neste projeto foi, na verdade, uma sugestão da Embrapa, dada esta falta.

O contato com o programa Reflora, no entanto, não apresentou nenhum resultado. Buscou-se, principalmente, imagens de plantas saudáveis, uma vez que não era objetivo do programa o armazenamento de imagens de plantas doentes. Assim, poderíamos complementar as classes de plantas saudáveis com poucos exemplos no banco original, como a **Batata saudável**. Entretanto, apesar de terem respondido o e-mail inicial de contato, não responderam os e-mails seguintes e, assim, não foi possível complementar o banco.

CAPÍTULO 5

CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, foi apresentado um aplicativo para identificação de patologias vegetais a partir de imagens das folhas dessas plantas, por meio de um modelo moderno de rede neural convolucional. Foram avaliadas as escolhas do modelo e seus parâmetros e realizadas comparações com o modelo de uma referência bibliográfica (G.; J., 2019), que propôs-se a resolver o mesmo problema. Foram também discutidas as escolhas das tecnologias envolvidas na criação do sistema que permitirá as pessoas utilizarem a rede neural, composto pelo aplicativo de celular e pela API.

Foi apresentada a teoria necessária para a compreensão do funcionamento do aplicativo, desde como redes neurais mais simples, como a do perceptron, operam, até as mais complexas e modernas, como as redes neurais convolucionais, em especial o modelo Inception, que motivou a criação do modelo utilizado neste trabaho: o Xception. Explicamos como as convoluções tradicionais e um outro tipo de convolução que este modelo tem como base, as convoluções separáveis por profundidade, funcionam. Explicamos também o papel de cada parâmetro de uma rede neural, como taxa de aprendizado, tamanho dos mini lotes, épocas, função de custo e algortimo de otimização, mostrando possíveis problemas causados por uma escolha ruim, como o sobreajuste e o subajuste. Mostramos também que não só parâmetros podem influenciar o resultado de uma rede, mas também as técnicas utilizadas, como a de *data augmentation* e as abordagens de avaliação final.

Em relação ao aplicativo, explicamos como eles são criados atualmente, detalhando as escolhas de tecnologias, com destaque para as híbridas. Explicamos também o funcionamento de uma API e o papel importante que ela tem no mundo integrado de hoje.

Foi apresentado também o método utilizado, detalhando quais foram as escolhas

dos hiperparâmetros para teste, como foi feita a aquisição do banco de dados e o pré-processamento das imagens e qual foi a configuração experimental para realizar os treinamentos. Detalhamos também a estrutura do modelo do Xception, em sua versão menor, explicando a função de cada camada e as abordagens utilizadas para avaliar o modelo ao final, que foram: a divisão do banco de dados em três conjuntos separados e a validação cruzada *k-fold*, discutindo os resultados e limitações. Apresentamos também como foram construídos o aplicativo e a API, detalhando as ferramentas utilizadas.

Os melhores parâmetros para a rede permitiram obter uma acurácia para os dados de teste de 99,35% e um *F1 Score* de 99,38%. Na abordagem da validação cruzada *k-fold*, utilizada para realizar uma prova real da abordagem anterior, obteve-se uma acurácia de 99,32%, mostrando que realmente foi possível obter uma acurácia melhor que a referência bibliográfica. Entretanto, não foi possível obter resultados satisfatórios para o uso real do aplicativo, dada a falta de capacidade de generalização da rede treinada.

Como foi possível construir:

- um aplicativo para celular, com redução de custos com a utilização de uma tecnologia híbrida, com bom desempenho e que pode ser utilizado por qualquer pessoa com um *smartphone* e acesso à internet;
- uma API que, por possuir poucas funções, é facilmente escalável,

conclui-se que, a depender da complexidade para solução do problema de generalização, é possível construir uma tecnologia de baixo custo e amplo acesso, para auxílio na diminuição de perdas de lavouras devido às doenças que podem atacá-las. Entretanto, gostaríamos de ressaltar que, se a solução para o problema for a criação de um banco de dados próprio, o custo do sistema pode se elevar bastante.

Com base neste trabalho e nos resultados obtidos, sugere-se como trabalhos futuros:

- implementação do modelo projetado no Keras diretamente no Tensorflow para utilização offline do aplicativo;

- criação de um banco de dados próprio ou procurar uma parceria para expansão da quantidade de espécies de plantas e de doenças identificáveis, avaliando se os resultados se mantém satisfatórios com o aumento de classes;

- criação de um banco de dados com imagens mais realistas e específicas considerando o tipo de uso do aplicativo e avaliar a viabilidade como solução de baixo custo;

- melhorar a parte de sugestões de soluções do aplicativo, implementando possivelmente um outro sistema de redes neurais específico capaz de fornecer soluções específicas para cada caso, dado que zonas rurais podem ter clima, relevo e outras características diferentes de outras;

- utilizar uma configuração experimental com mais capacidade para testar redes mais profundas, com parâmetros que exigem mais memória e processamento;

- realizar uma investigação profunda das causas dos resultados diferentes para as abordagens de divisão dos dados em 3 conjuntos e da validação cruzada *k-fold*.

APÊNDICE A

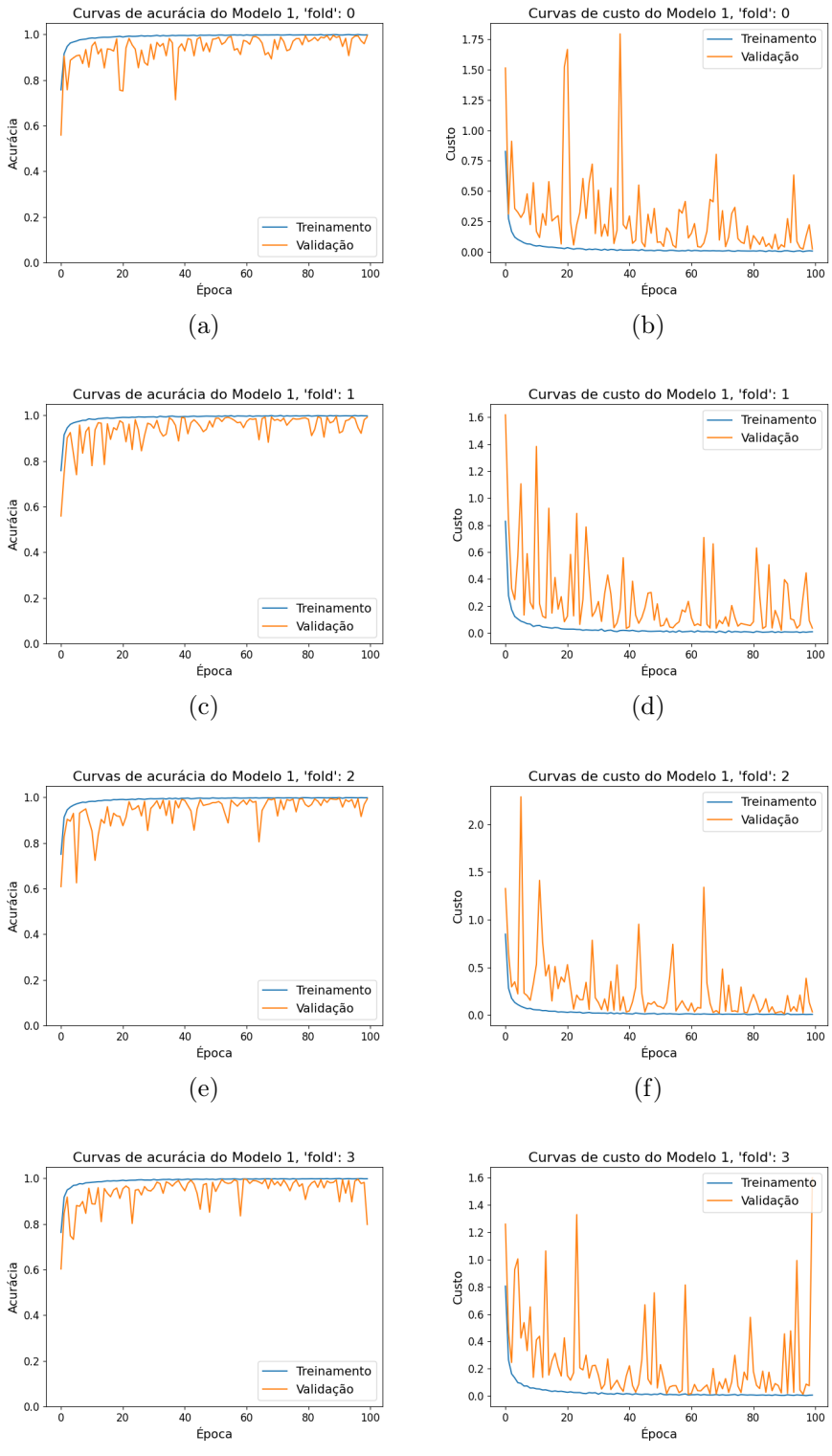
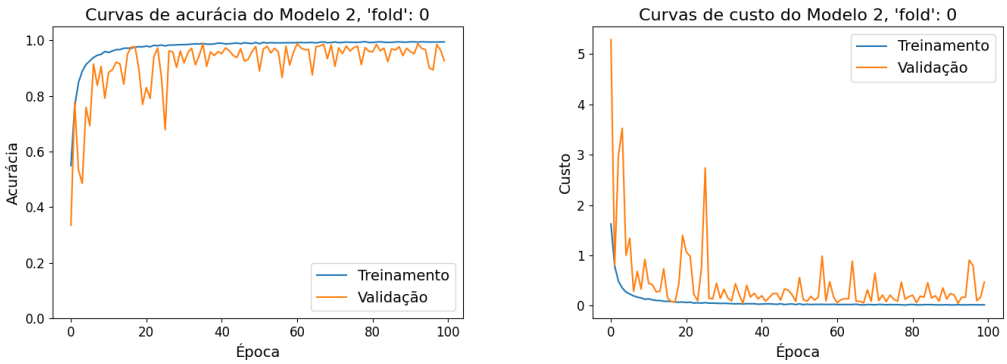
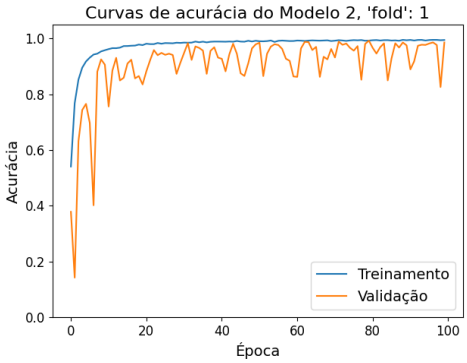


Figura 5.1. Resultados do treinamento e validação do Modelo 1 para validação cruzada *k-fold*.

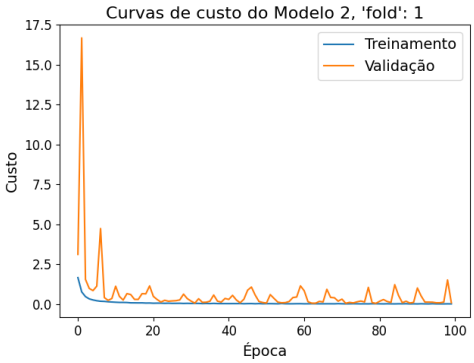


(a)

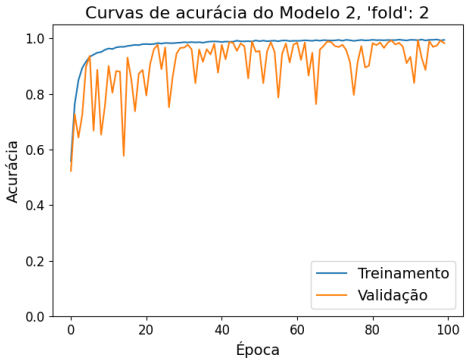
(b)



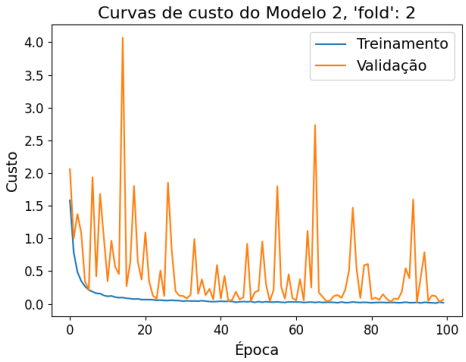
(c)



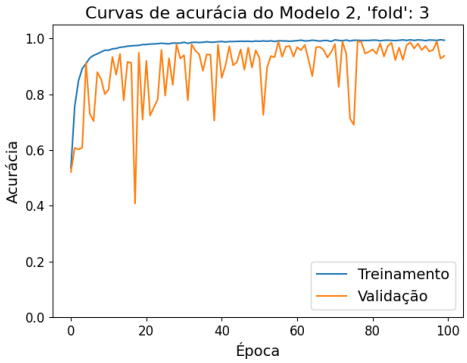
(d)



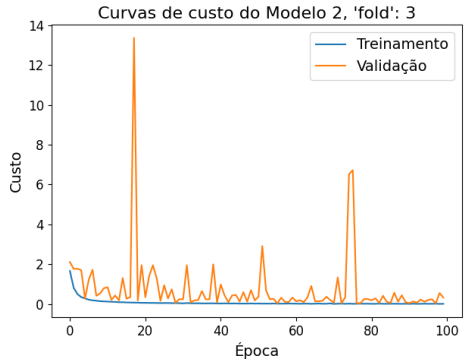
(e)



(f)

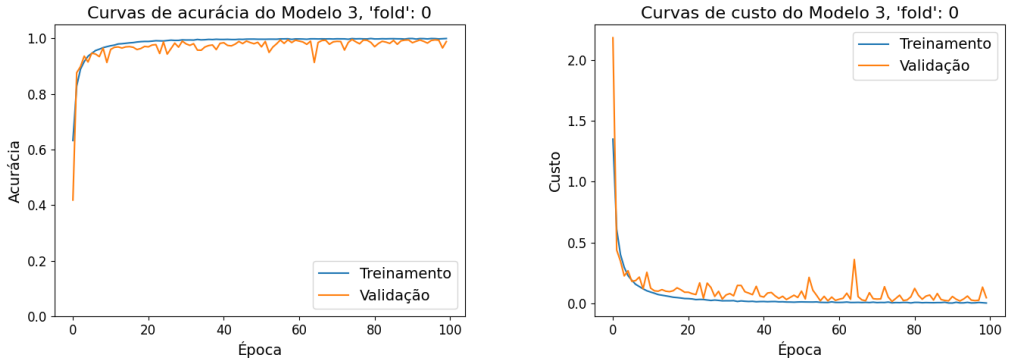


(g)



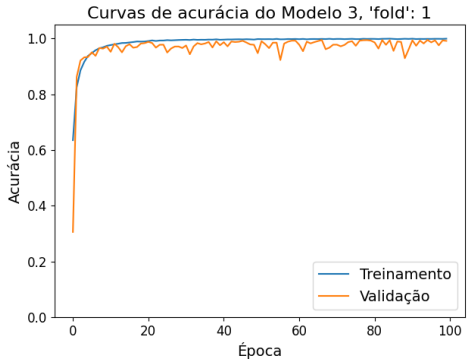
(h)

Figura 5.2. Resultados do treinamento e validação do Modelo 2 para validação cruzada *k-fold*.

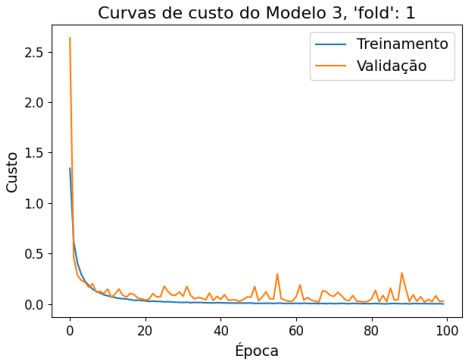


(a)

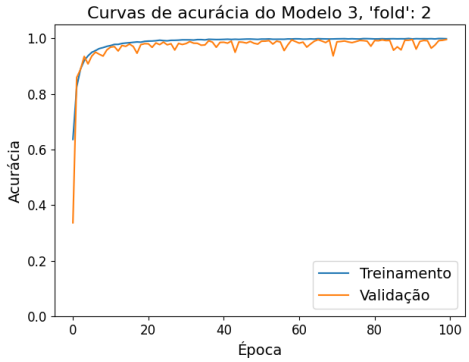
(b)



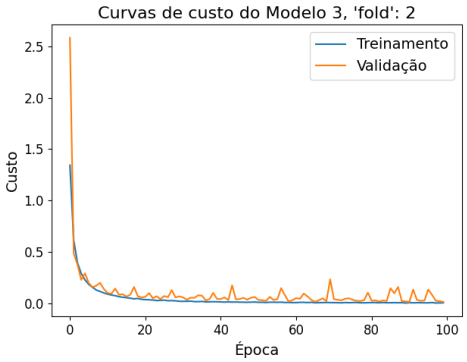
(c)



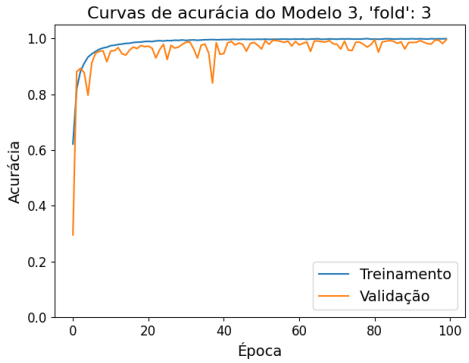
(d)



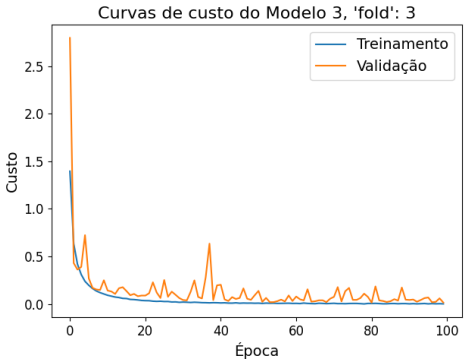
(e)



(f)



(g)



(h)

Figura 5.3. Resultados do treinamento e validação do Modelo 3 para validação cruzada *k-fold*.

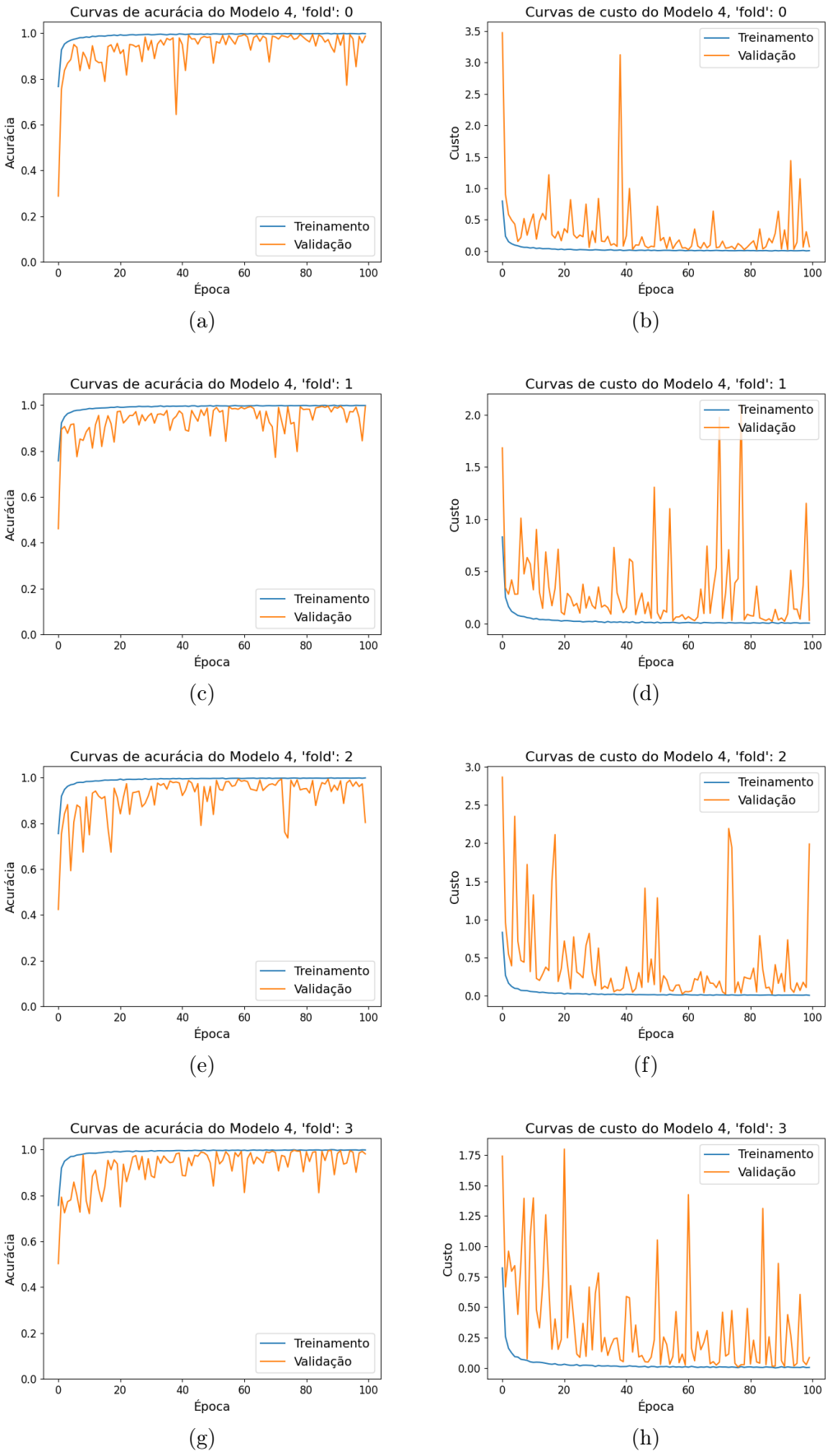
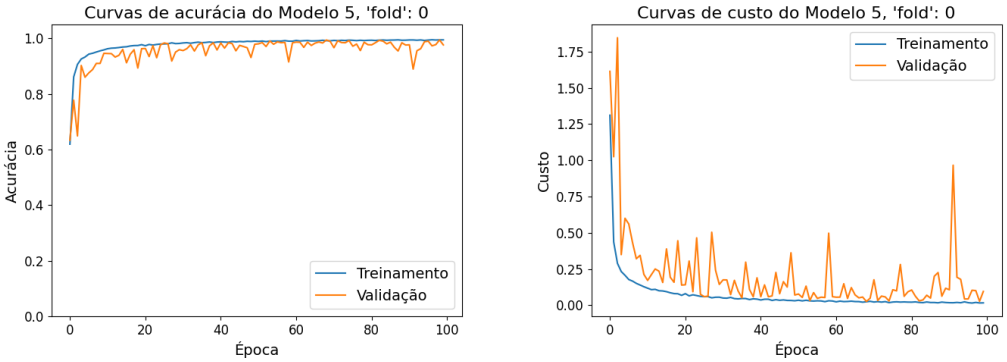
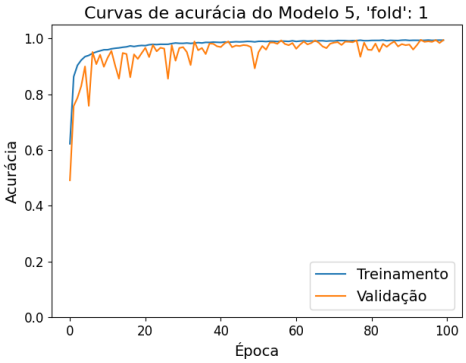


Figura 5.4. Resultados do treinamento e validação do Modelo 4 para validação cruzada *k-fold*.

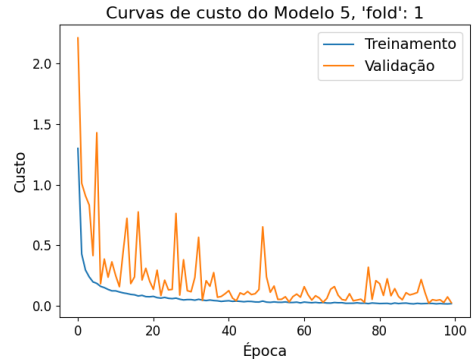


(a)

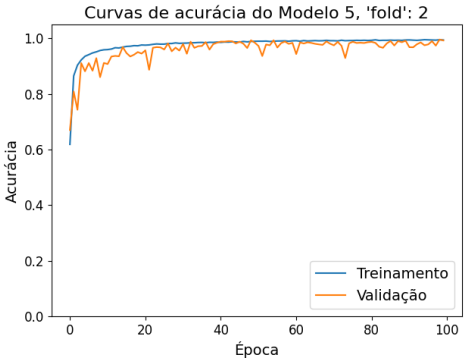
(b)



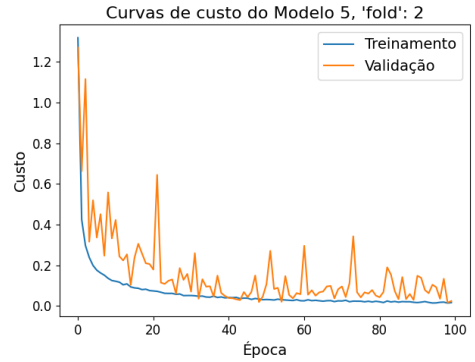
(c)



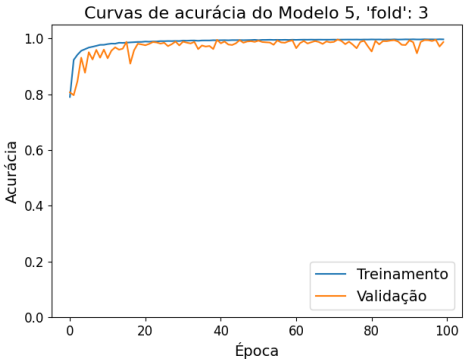
(d)



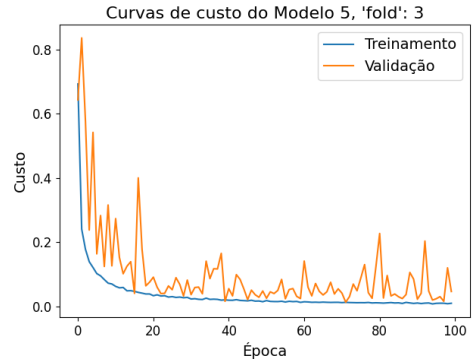
(e)



(f)

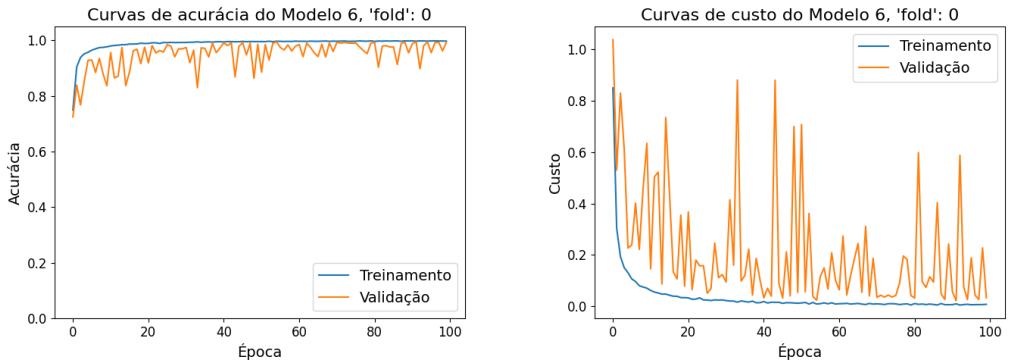


(g)



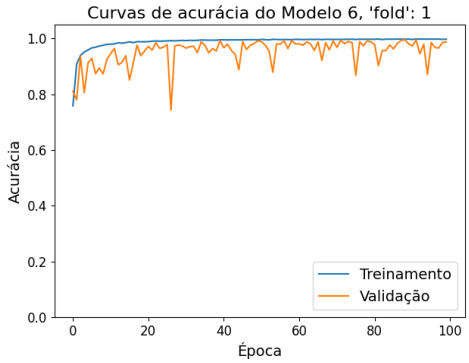
(h)

Figura 5.5. Resultados do treinamento e validação do Modelo 5 para validação cruzada *k-fold*.

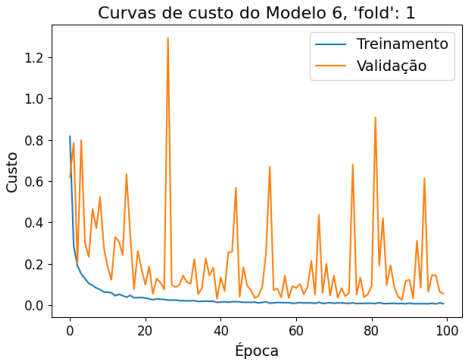


(a)

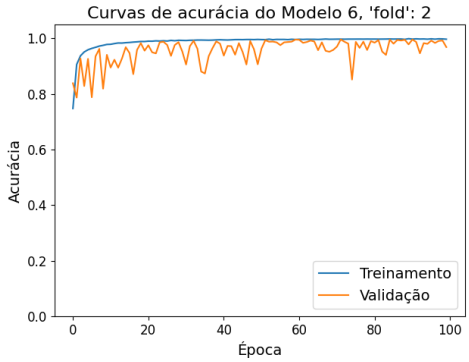
(b)



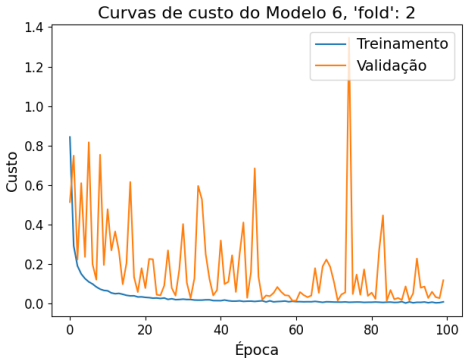
(c)



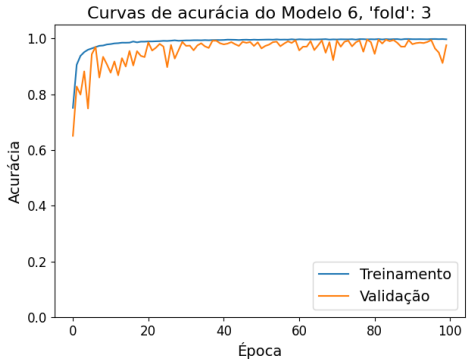
(d)



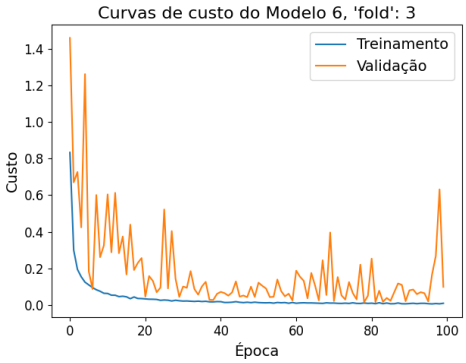
(e)



(f)



(g)



(h)

Figura 5.6. Resultados do treinamento e validação do Modelo 6 para validação cruzada *k-fold*.

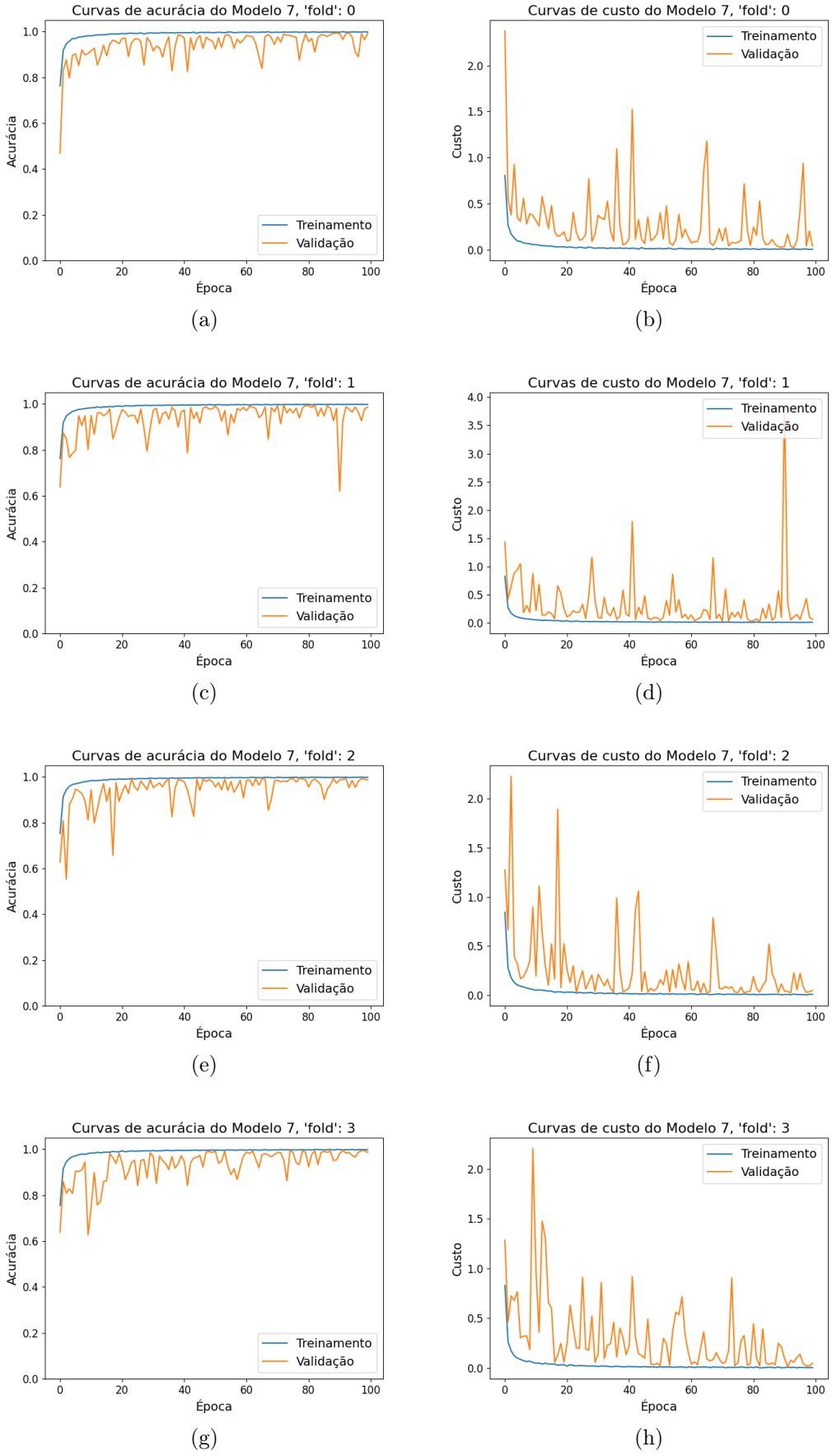


Figura 5.7. Resultados do treinamento e validação do Modelo 7 para validação cruzada *k-fold*.

REFERÊNCIAS BIBLIOGRÁFICAS

ACADEMY, D. S. *Capítulo 4 – O Neurônio, Biológico e Matemático*. 2022. Citado na página 8.

AGROLINK. *Problemas*. 2023. Disponível em: <https://www.agrolink.com.br/problemas> – acesso em 25 jun. 2023. Citado na página 38.

APPLE. *About Objective-C*. 2014. Disponível em: <https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html> – acesso em 25 jun. 2023. Citado na página 34.

BOTELHO, M. *Código da API do TCC*. 2023. Disponível em: <https://github.com/mumispb/tcc-api> – acesso em 25 jun. 2023. Citado na página 51.

BOTELHO, M. *Código do Aplicativo do TCC*. 2023. Disponível em: <https://github.com/mumispb/tcc-app> – acesso em 25 jun. 2023. Citado na página 50.

BOTELHO, M. *Código do Treinamento da CNN do TCC*. 2023. Disponível em: <https://github.com/mumispb/tcc-training> – acesso em 25 jun. 2023. Citado 2 vezes nas páginas 37 and 48.

BRUNO, A.; MORONI, D.; DAINELLI, R.; ROCCHI, L.; MORELLI, S.; FERRARI, E.; TOSCANO, P.; MARTINELLI, M. Improving plant disease classification by adaptive minimal ensembling. *Frontiers in Artificial Intelligence*, v. 5, 2022. ISSN 2624-8212. Disponível em: <https://www.frontiersin.org/articles/10.3389/frai.2022.868926>. Citado na página 1.

BURKOV, A. *The Hundred-Page Machine Learning Book*. Quebec City, Canada: Andriy Burkov, 2019. Citado 6 vezes nas páginas 3, 6, 9, 11, 23, and 31.

CANALRURAL.COM.BR. *Inteligência artificial identifica plantas doentes*. 2023. Disponível em: <https://www.canalrural.com.br/noticias/agricultura/inteligencia-artificial-identifica-plantas-doentes/> – acesso em 22 jun. 2023. Citado na página 3.

CHOLLET, F. *Xception: Deep Learning with Depthwise Separable Convolutions*. 2017. Disponível em: <https://arxiv.org/abs/1610.02357>. Citado 3 vezes nas páginas 26, 31, and 76.

CHOLLET, F. *Image classification from scratch*. 2020. Disponível em: <https://keras.io/examples/vision/image_classification_from_scratch/> – acesso em 25 jun. 2023. Citado na página 41.

DEVELOPER, A. *Desenvolver apps Android com o Kotlin*. 2023. Disponível em: <https://developer.android.com/kotlin?hl=pt-br> – acesso em 25 jun. 2023. Citado na página 34.

EDUCATIVE. *History of Swift*. 2023. Disponível em: <https://www.educative.io/courses/swift-programming-mobile-app/q282KZA1N33> – acesso em 25 jun. 2023. Citado na página 34.

ESALQ/USP, C. de Estudos Avançados em E. A. *PIB-AGRO/CEPEA: APÓS RECORDES EM 2020 E 2021, PIB DO AGRO CAI 4,22% EM 2022*. 2023. Disponível em: <https://www.cepea.esalq.usp.br/br/releases/pib-agro-cepea-apos-recordes-em-2020-e-2021-pib-do-agro-cai-4-22-em-2022.aspx#:~:text=Considerando%2Dse%20os%20desempenhos%20da,pecu%C3%A1rio%20avan%C3%A7ou%202%2C11%25.> – acesso em 22 jun. 2023. Citado na página 2.

FAO. *FAO lista 5 doenças de plantas que a crise climática está agravando*. 2022. Disponível em: <https://brasil.un.org/pt-br/184058-fao-lista-5-doen%C3%A7as-de-plantas-que-crise-clim%C3%A1tica-est%C3%A1-agravando> – acesso em 21 jun. 2023. Citado na página 1.

G., G.; J., A. P. *Data for: Identification of Plant Leaf Diseases Using a 9-layer Deep Convolutional Neural Network*. 2017. Mendeley Data, V1, doi: 10.17632/tywbtsjrjv.1. Citado na página 36.

G., G.; J., A. P. Identification of plant leaf diseases using a nine-layer deep convolutional neural network. *Computers Electrical Engineering*, v. 76, p. 323–338, 2019. ISSN 0045-7906. Disponível em: <https://www.sciencedirect.com/science/article/pii/S0045790619300023>. Citado 5 vezes nas páginas 46, 47, 57, 58, and 79.

GOOGLE. *Redes neurais multiclasse: Softmax*. 2022. Disponível em: <https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax?hl=pt-br> – acesso em 25 jun. 2023. Citado na página 18.

GOV.BR. *Valor Bruto da Produção Agropecuária de 2023 é estimado em R$ 1,216 trilhão*. 2023. Disponível em: <https://www.gov.br/agricultura/pt-br/assuntos/noticias/valor-bruto-da-producao-agropecuaria-de-2023-e-estimado-em-r-1-216-trilhao#:~:text=O%20Valor%20Bruto%20da%20Produ%C3%A7%C3%A3o,foi%20de%20R%24%201%2C161%20trilh%C3%A3o.> – acesso em 21 jun. 2023. Citado na página 2.

HAYKIN, S. *Neural Networks and Learning Machines*. McMaster University Hamilton, Ontario, Canada: PEARSON, 2009. Citado 4 vezes nas páginas 7, 9, 11, and 32.

HE, K.; ZHANG, X.; REN, S.; SUN, J. *Deep Residual Learning for Image Recognition*. 2015. Disponível em: <https://arxiv.org/abs/1512.03385>. Citado na página 29.

HUGHES, D. P.; SALATHE, M. *An open access repository of images on plant health to enable the development of mobile disease diagnostics*. 2016. Disponível em: <https://arxiv.org/abs/1511.08060>. Citado 2 vezes nas páginas 36 and 38.

IBGE. *Censo Agro 2017*. 2017. Disponível em: <https://censoagro2017.ibge.gov.br/templates/censo_agro/resultadosagro/estabelecimentos.html> – acesso em 22 jun. 2023. Citado na página 1.

IBGE. *Informativo Agricultura Familiar - Censo Agro 2017*. 2017. Disponível em: <https://censoagro2017.ibge.gov.br/templates/censo_agro/resultadosagro/pdf/agricultura_familiar.pdf> – acesso em 22 jun. 2023. Citado na página 2.

IMAGENET. *ImageNet Object Localization Challenge*. 2020. Disponível em: <https://www.kaggle.com/competitions/imagenet-object-localization-challenge/data> – acesso em 2 ago. 2023. Citado na página 41.

IOFFE, S.; SZEGEDY, C. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. Disponível em: <https://arxiv.org/pdf/1502.03167.pdf>. Citado na página 29.

KINGMA, D. P.; BA, J. *Adam: A Method for Stochastic Optimization*. 2017. Disponível em: <https://arxiv.org/pdf/1412.6980.pdf>. Citado na página 45.

MCCULLOCH, W. S.; PITTS, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mothemnticnl Biology*, v. 52, p. 99–115, 1990. Disponível em: <https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf>. Citado na página 8.

O'REILLY. *The history of Kotlin*. 2023. Disponível em: <https://www.oreilly.com/library/view/kotlin-for-enterprise/9781788997270/ea4ec584-db64-4026-89a8-2086301eb9c5.xhtml> – acesso em 25 jun. 2023. Citado na página 34.

PAWARA, P.; OKAFOR, E.; SCHOMAKER, L.; WIERING, M. Data augmentation for plant classification. In: BLANC-TALON, J.; PENNE, R.; PHILIPS, W.; POPESCU, D.; SCHEUNDERS, P. (Ed.). *Advanced Concepts for Intelligent Vision Systems*. Cham: Springer International Publishing, 2017. p. 615–626. ISBN 978-3-319-70353-4. Citado na página 24.

POWELL, V. *Image Kernels Explained Visually*. 2015. Disponível em: <https://setosa.io/ev/image-kernels/> – acesso em 25 jun. 2023. Citado 2 vezes nas páginas 20 and 21.

REFLORA. *Programa REFLORA*. 2010. Disponível em: <https://reflora.jbrj.gov.br/reflora/PrincipalUC/PrincipalUC.do> – acesso em 25 jun. 2023. Citado na página 77.

SANDERSON, G. *Backpropagation calculus*. 2017. Disponível em: <https://www.3blue1brown.com/lessons/backpropagation-calculus> – acesso em 25 jun. 2023. Citado na página 14.

SNA. *A importância da conectividade no meio rural*. 2021. Disponível em: <https://www.sna.agr.br/cerca-de-70-de-cinco-milhoes-de-propriedades-rurais-no-brasil-nao-tem-conectividade/> – acesso em 2 ago. 2023. Citado 2 vezes nas páginas 4 and 5.

SZEGEDY, C.; LIU, W.; JIA, Y.; SERMANET, P.; REED, S.; ANGUELOV, D.; ERHAN, D.; VANHOUCKE, V.; RABINOVICH, A. *Going Deeper with Convolutions*. 2014. Disponível em: <https://https://arxiv.org/abs/1409.4842>. Citado 2 vezes nas páginas 24 and 25.

TENSORFLOW. *Mnist Database*. 2023. Disponível em: <https://www.tensorflow.org/datasets/catalog/mnist> – acesso em 25 jun. 2023. Citado na página 18.

WANG, C.-F. *A Basic Introduction to Separable Convolutions*. 2018. Disponível em: <https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728> – acesso em 24 jun. 2023. Citado 3 vezes nas páginas 26, 27, and 28.

ZVORNICANIN, E. *Relation Between Learning Rate and Batch Size*. 2023. Disponível em: <https://www.baeldung.com/cs/learning-rate-batch-size> – acesso em 22 jun. 2023. Citado na página 16.