UNICAMP

UNIVERSIDADE ESTADUAL DE CAMPINAS

Instituto de Matemática, Estatística e Computação Científica

ANA SILVIA MORETTO BRAGA

Um Estudo Sobre As redes Neurais Aplicadas na Detecção de Comportamento Humano

Campinas

2022

Ana Silvia Moretto Braga

Um Estudo Sobre As redes Neurais Aplicadas na Detecção de Comportamento Humano

Dissertação apresentada ao Instituto de Matemática, Estatística e Computação Científica da Universidade Estadual de Campinas como parte dos requisitos exigidos para a obtenção do título de Mestra em Matemática Aplicada e Computacional.

Orientadora: Juliana Verga Shirabayashi

Coorientador: Jair da Silva

Este trabalho corresponde à versão final da Dissertação defendida pela aluna Ana Silvia Moretto Braga e orientada pela Profa. Dra. Juliana Verga Shirabayashi.

Campinas

2022

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

B73e

Braga, Ana Silvia Moretto, 1971-
Um estudo sobre as redes neurais aplicadas na detecção de comportamento humano / Ana Silvia Moretto Braga. – Campinas, SP : [s.n.], 2022.

Orientador: Juliana Verga Shirabayashi.
Coorientador: Jair da Silva.
Dissertação (mestrado profissional) – Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica.

1. Redes neurais (Computação). 2. Aprendizado profundo. 3. Funções de custo. 4. Gradiente descendente. 5. Reconhecimento de imagem. I. Verga, Juliana, 1984-. II. Silva, Jair da. III. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Computação Científica. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Study on neural networks applied to detection of human behavior
Palavras-chave em inglês:
Neural networks (Computer science)
Deep learning
Cost functions
Gradient descending
Image recognition
Área de concentração: Matemática Aplicada e Computacional
Titulação: Mestra em Matemática Aplicada e Computacional
Banca examinadora:
Juliana Verga Shirabayashi [Orientador]
Carla Taviane Lucke da Silva Ghidini
Emerson Vitor Castelani
Data de defesa: 28-04-2022
Programa de Pós-Graduação: Matemática Aplicada e Computacional

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-2979-0654
- Currículo Lattes do autor: http://lattes.cnpq.br/0559616415501875

Dissertação de Mestrado Profissional defendida em 28 de abril de 2022 e aprovada pela banca examinadora composta pelos Profs. Drs.

Prof(a). Dr(a). JULIANA VERGA SHIRABAYASHI

Prof(a). Dr(a). EMERSON VITOR CASTELANI

Prof(a). Dr(a). CARLA TAVIANE LUCKE DA SILVA GHIDINI

A Ata da Defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria de Pós-Graduação do Instituto de Matemática, Estatística e Computação Científica.

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

A todos os professores e funcionários do IMECC, pela oportunidade incrível de aprendizado neste trabalho em conjunto e pela confiança em mim depositada, apesar do meu longo tempo afastada da vida acadêmica.

Aos meus orientadores *Profa. Dra. Juliana Verga Shirabayashi* e *Prof. Dr. Jair da Silva* pelo estímulo, apoio e confiança em meu trabalho.

Aos meus gerentes da IBM, *Mauricio Akio Nishioka* e *Raombanarivo Dina Ramilijaona* por toda compreensão e incentivo durante este longo desafio.

Aos professores do Instituto de Computação da Unicamp, *Prof. Dr. Zanoni Dias, Profa. Dra. Sandra Eliza Fontes de Avila, Profa. Dra. Esther Luna Colombini, Prof. Dr. Anderson de Rezende Rocha e Prof. Dr. Hélio Pedrini* que me abriram as portas do conhecimento de redes profundas e que me permitiram desenvolver parte deste trabalho.

Aos Colegas do Curso de Mineração de Dados Complexos: *Rodrigo Rodarte, Érico Luiz Encarnação Rocha, Helena Guimarães de Menezes Viana, Roberto Navas Pinheiro* pela parceria e todo esforço em conjunto que nos levaram aos resultados iniciais do estudo de caso aqui apresentado.

Ao meu marido *Ruy Carlos Herrera Braga*, pelo carinho, companheirismo e, principalmente, pela compreensão aos muitos finais de semana em que renunciamos a nosso descanso para que eu pudesse mergulhar em livros e pesquisas.

A todos os colegas de Mestrado que, embora não nominalmente citados, foram extremamente importantes nesta jornada de conhecimento e aprendizado.

Resumo

O objetivo deste trabalho, além de discutir os aspectos matemáticos envolvidos na formação das redes neurais e a importância de se entender os mesmos para os ajustes em algoritmos, é mostrar o desempenho de algoritmos de redes neurais para reconhecimento de imagem. Para este experimento foi utilizado um banco de imagens disponibilizado em uma plataforma de modelagem preditiva e de competições analíticas. O objetivo da competição era identificar um entre 10 diferentes comportamentos de motoristas ao volante para cada imagem. A ferramenta utilizada para execução dos algoritmos foi o *Google Colaboratory*, o qual disponibiliza a utilização da linguagem Python combinada com a biblioteca *TensorFlow Keras*, bem como o uso de arquiteturas pré treinadas de redes neurais em que é possível aproveitar pesos aprendidos anteriormente em ambientes computacionais mais robustos. Todos os recursos utilizados são gratuitos e disponíveis na internet a qualquer usuário. Foram treinados 14 modelos, tendo o desempenho de cada um deles avaliado pela acurácia balanceada, à qual chegou a alcançar desempenho acima de 81% em alguns destes modelos. Considerando o fato de que estes modelos foram construídos com redes pré treinadas, onde o desempenho máximo atingido por tais redes em competições ficam em torno de 97% é realmente extraordinário verificar os resultados obtidos com a utilização de recursos simples e que estão à disposição da maioria das pessoas. Também foi mensurada a capacidade de generalização dos modelo utilizando-se a perda logarítmica. Neste caso, o melhor resultado obtido foi de 0.49606. Comparando este resultado, com os resultados obtidos pelo vencedor desta competição que foi de 0.08, podemos constatar que nosso modelo é razoavelmente bom e, tomando por base a acurácia balanceada, é possível concluir que ainda há espaço para melhorar, só que para isto, um dos caminhos mais prováveis seria adicionar recursos computacionais mais robustos, mas que não é o propósito deste trabalho.

Palavras-chave: redes neurais. redes convolucionais. função de custo. gradiente descendente. aprendizado de máquina. aprendizado profundo. otimização. reconhecimento de imagem.

Abstract

The objective of this work, in addition to discussing the mathematical aspects involved in the formation of neural networks and the importance of understanding them for adjustments in algorithms, is to demonstrate the performance of neural network algorithms for image recognition. For this experiment, a database of images available in a predictive modeling and analytical competition platform was used, where the objective is to identify one of 10 different behaviors of drivers behind the wheel for each image. The tool used to execute the algorithms was Google Collaborative, which used the Python language combined with the TensorFlow Keras library, in addition to using three pre-trained neural network architectures where it was possible to take advantage of weights previously learned in more robust computational environments. All resources used are free and available on the internet to any user. Fourteen models were trained, with the performance of each one being evaluated by balanced accuracy, where it was possible to achieve performance above 81% in some of these models. Considering that these models were built with pre-trained networks, where their maximum performance achieved in competitions is around 97% it is extraordinary to verify such results obtained with the use of simple resources that are available to the most people. The generalization capacity of the models was also measured using the logarithmic loss. In this case, the best result obtained was 0.49606. Comparing this result with the results obtained by the winner of this competition, which was 0.08, we can see that our model is reasonably good and, based on balanced accuracy, it is possible to conclude that there is still room for improvement, but for this, a one of the most likely paths would be to add payable computational resources, but that is not the purpose of this work.

Keywords: neural networks. convolutional networks. cost function. descending gradient. machine learning. deep learning. optimization. image recognition.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ACC	Acurácia
AccB	Acurácia Balanceada
AdaLinE	*Adaptive Linear Element*
ADAM	*Adaptive Moment*
API	*Application Programming Interface*
CFM	Conselho Federal de Medicina
CNN	*Convolutional Neural Network*
EUA	Estados Unidos da América
FC	*Fully Connected*
FF	*Feed Forward*
FN	Falso Negativo
FP	Falso Positivo
GPU	*Graphic Processing Unit*
IBM	*International Business Machine*
ILSVRC	*ImageNet Large Scale Visual Recognition Challenge*
IMECC	Instituto de Matemática, Estatística e Computação Científica
LTU	*Linear Threshold Unit*
MLP	*Multi Layer Perceptron*
MPG	Milhas por galão
MSE	*Mean Squared Error*
ReLU	*Rectified Linear Unit*
ResNet	*Residual Network*
SDG	*Stochastic Descending Gradient*

SVM	*Support Vector Machine*
SUS	Sistema Único de Saúde
UNICAMP	Universidade Estadual de Campinas
VGG	*Visual Geometry Group*
VN	Verdadeiro Negativo
VP	Verdadeiro Positivo

Sumário

Introdução

Um dos principais aspectos que diferenciam os humanos de outras criaturas do reino animal é seu talento em lidar com situações de maneira criativa, bem como a sua capacidade cognitiva de relacionar fatos e buscar na natureza a inspiração para soluções dos seus problemas. Muitas invenções humanas foram inspiradas em mecanismos que vem da natureza e isto também ocorre com as redes neurais que tem sua inspiração na arquitetura do cérebro humano. Estudos relacionados à arquitetura cerebral, como ele lida com sobrecarga de informação e múltiplos dados sensoriais, como funciona o processo de tomada de decisão e sua capacidade de desenvolver regras baseadas em experiências anteriores têm sido muito relevantes dentro da computação, especialmente dentro da inteligência artificial e mais especificamente, dentro das redes neurais onde, graças ao disparo de neurônios ao longo de uma cadeia e, baseado nos disparos de neurônios anteriores, o próximo neurônio decide gerar ou não gerar sinais de acordo com estas entradas. Os sinais, então ativados, são transmitidos para os próximos neurônios criando assim um sistema complexo e poderoso na predição de resultados.

Em outra frente, de acordo com as análises do Professor Adjunto em Ciências da Computação da Universidade de Standford e Co-Fundador do Coursera, Andrew Ng, (NG, 2022d), a evolução tecnológica, bem como a capacidade atual de geração de dados, tem contribuído substancialmente para a formação de redes neurais computacionais extremamente eficientes, aplicadas aos problemas cada vez mais complexos, já que uma das premissas básicas da utilização de redes neurais é que quanto maior o volume de dados de entrada, melhor é o desempenho dos algoritmos de redes neurais, seja em problemas de predição, seja em problemas de classificação.

No entanto, apesar de *softwares* e outros recursos computacionais tais como memórias e processadores disponibilizados, até certo ponto de forma gratuita na internet, os ajustes entre dados, *software*, *hardware* requerem um entendimento razoável dos conceitos matemáticos que conectam todos estes elementos a fim de que se possa extrair o melhor resultado com os recursos disponíveis.

As redes neurais artificiais estão contidas no campo da inteligência artificial (CHOLLET, 2019). Apesar das pesquisas em torno da Inteligência Artificial existirem desde os anos 30, o termo surgiu pela primeira vez na *Conferência de Dartmouth* em 1956 e pode ser associado a máquinas que imitam as funções cognitivas humanas, tais como a capacidade de reconhecer e classificar padrões em imagens, textos, sons, ou de prever eventos futuros baseados em fatos presentes. A inteligência artificial tem tido um papel fundamental na transformação das diversas áreas da nossa existência, seja na saúde

como ferramenta de suporte a médicos em seus diagnósticos, seja na engenharia, tanto na produção de carros de direção autônoma, como na construção de casas inteligentes capazes de executar diversas tarefas através de comando de voz. Também tem forte influência na publicidade com o *marketing* direcionado, nos transportes com a otimização da cadeia de suprimentos, nas transações financeiras com operações remotas e dispositivos de segurança capazes de identificar possíveis fraudes de forma bastante eficiente. Estes são apenas alguns dos muitos exemplos de transformação promovidos pela inteligência artificial. Dentro do campo da inteligência artificial existe o aprendizado de máquina, que se dedica à construção e estudo de programas que não são explicitamente programados, mas que são capazes de aprender padrões conforme eles são expostos a um conjunto de dados. Na prática, por questões matemáticas bem simples que veremos adiante, quanto mais dados, melhor será o aprendizado destes padrões.

Segundo (GÉRON, 2019), existem 4 tipos de aprendizado de máquina: supervisionado, não supervisionado, semi supervisionado e aprendizado por reforço.

O objetivo do aprendizado supervisionado é fazer uma predição baseada em fatos conhecidos e que são informados ao algoritmo. Quando a predição é baseada em variáveis discretas, chamamos este processo de classificação. Neste caso, o aprendizado advém de uma base dados já rotulada com os valores verdadeiros. A partir deste conjunto de dados, o algoritmo deve aprender os padrões e passar a classificar dados desconhecidos e sem rótulos. Já no caso de variáveis contínuas utilizamos o processo de regressão, ou seja, baseados num histórico de preço de imóveis e algumas características relevantes, o algoritmo deve fazer uma previsão futura para o preço de qualquer imóvel dentro daquelas características pré-determinadas.

Já no método não supervisionado, o objetivo do algoritmo é procurar por padrões na estrutura dos dados e agrupá-los da maneira mais adequada, ao invés de predizer algum evento. Neste caso, não são passados rótulos para o algoritmo, o esperado deste método é descobrir padrões e tendências e agrupar ou associar os dados de acordo com suas semelhanças.

No aprendizado semi supervisionado, os algoritmos conseguem lidar com dados treinamento rotulados e não rotulados. Um bom exemplo de utilização deste método é o Google Fotos. Ao carregar fotos de família na ferramenta, ela é capaz de reconhecer uma determinada pessoa em várias fotos. Esta é a parte não supervisionada do algoritmo. Já a parte supervisionada, fica por conta do usuário que deverá identificar quem são as pessoas das fotos, que é extremamente útil para pesquisa de fotos.

Por último, o aprendizado por reforço é um tipo de treinamento que tem por objetivo a tomada de uma série de decisões. O sistema de aprendizagem, chamado de agente neste contexto, pode observar o ambiente, selecionar e executar ações e obter recompensas (em caso de acertos) ou penalidades (em casos de erros). Esta categoria de

aprendizado é muito utilizada em *games* e robótica.

O fundamento do aprendizado de máquina é ver e rever repetidamente um conjunto de dados, ao invés de ser um conjunto de regras explicitamente programado por humanos afim de tomar uma decisão. Trata-se de uma técnica extremamente eficiente para dados estruturados. Entretanto, para dados não estruturados, que é o caso de sons, imagens e textos, ela apresenta algumas limitações.

Um exemplo destas limitações (IBM, 2022b) é a utilização de *pixels* de imagens como características de entrada de dados. Supondo a análise de uma imagem pequena com 256×256 *pixels*, quando transformada em um vetor, isto resultaria em mais de 65 mil características de entrada, o que é um montante considerável de dados de entrada para se trabalhar. Um outro ponto importante, é que somente o valor de um pixel não é uma informação suficiente a respeito de uma imagem. A relação espacial entre todos os pixels da imagem é essencial para compreender formatos, contornos, além de outras características relevantes para uma correta classificação.

É dentro deste contexto que foi criado um segmento conhecido como *Deep Learning* dentro de *Machine Learning*. *Deep Learning* nada mais é que um *machine learning* feito com a utilização de redes neurais profundas (*deep neural networks*), ou redes neurais com várias camadas, uma vez que o conceito de profundidade é diretamente ligado ao número de camadas em uma arquitetura de redes neurais. Trata-se de um tema de vanguarda, onde a maioria de investimentos e pesquisas em aprendizado de máquina estão concentradas devido ao incrível desempenho alcançado ultimamente por vários algoritmos em grandes bancos de dados. Um aspecto bastante interessante do aprendizado profundo é que o próprio algoritmo escolhe quais características (*features*) deverão ser utilizadas nos modelos, enquanto num modelo clássico de aprendizado de máquina, estes dados precisam ser prioritariamente definidos por quem está treinando o modelo. Em outras palavras, no caso de classificação de imagens, as redes neurais irão receber os *pixels* de uma imagem como entrada e processá-las para extrair as características (*features*) relevantes através de diferentes e complexas combinações. Frequentemente, apesar destas combinações não serem claramente compreensíveis aos olhos humanos ao tentar interpretá-las em suas camadas intermediárias, elas são extremamente úteis para completar as tarefas de reconhecimento destas imagens.

As redes neurais são definidas por dois elementos básicos: algoritmo de aprendizado e topologia (ou arquitetura) (COLOMBINI, 2020).

Os algoritmos de aprendizado são capazes de aproximar funções contínuas arbitrárias, ou seja, ao contrário de outros métodos de aprendizado de máquina tradicionais onde os modelos são baseados em funções pré-determinadas (Regressão Linear, Regressão Logística), nas redes neurais a função de aproximação é definida pela própria rede, de acordo com sua topologia e baseada nos pesos constituídos no treinamento. Os parâmetros

de cada modelo devem ser ajustados de acordo com o arranjo da arquitetura e com o paradigma de cada rede (se ela é treinada de modo supervisionado, não supervisionado ou reforço).

Já na topologia das redes neurais é possível identificar as unidades de processamento, ou seja, o tipo de neurônio que está sendo utilizado, como são as conexões entre estes neurônios (se eles se ligam de forma inibitória – sinal do peso negativo, ou de forma excitatória – sinal do peso positivo), como é o arranjo entre os neurônios, quais estão ligados ou desligados, como suas camadas estão organizadas e como ocorre a propagação da informação.

As redes neurais possuem 3 tipos básicos de camadas: entrada, intermediárias (ou escondidas) e saída. A camada de entrada é obrigatória e cada neurônio desta camada representa uma variável da entrada de dados. Já as camadas intermedárias são camadas opcionais, mas podem ser múltiplas. São elas quem determinam a complexidade da rede. Em algumas arquiteturas é possível inclusive desativar algumas destas camadas para otimizar o processamento dos dados. Por último, a camada de saída é responsável por exibir os resultados gerados. O número de neurônios desta camada vai depender do formato de dados esperado.

A proposta deste trabalho é discutir os métodos de aprendizado de máquina onde as redes neurais estão inseridas, destacando os aspectos matemáticos em que as principais topologias de redes neurais se baseam, demonstrar de forma prática o modo como as redes neurais computam seus cálculos, e também, entender a razão pelo qual o volume de dados é relevante para seu desempenho.

Além disto, discorremos sobre as redes pré-treinadas, mostramos seus benefícios e as soluções que driblaram as limitações técnicas da época em que foram apresentadas.

Finalmente, apresentamos alguns resultados de testes computacionais onde o objetivo principal foi demonstrar como determinados ajustes em algoritmos podem trazer resultados expressivos, dependendo do problema que se propõe resolver, e até onde se pode chegar com os recursos gratuitos atualmente disponíveis.

Este trabalho está organizado da seguinte forma:

- No capítulo 1, apresentamos a revisão bibliográfica de alguns trabalhos já existentes na literatura, bem como apresentamos alguns conceitos matemáticos sob os quais os processos de Regressão Linear, Regressão Logística e Redes Neurais foram desenvolvidas;
- No capítulo 2, apresentamos a metodologia aplicada aos processos de *Machine Learning*;

- No capítulo 3, fazemos a análise dos dados e a apresentação de alguns detalhes do algoritmo proposto para cada um dos 14 modelos propostos;
- No capítulo 4, são apresentados os resultados e as análises dos testes computacionais realizados para cada modelo proposto;
- No capítulo 5, são feitas as considerações finais deste trabalho e algumas propostas para trabalhos futuros.

1 Revisão Bibliográfica

É fascinante o momento histórico que estamos vivenciando com a Inteligência Artificial (IA). Apesar de seus vários ciclos em que se alternaram momentos de grandes investimentos e expectativas, com momentos de frustração e arrefecimento nos investimentos ao longo de sua história, a IA tem provocado uma revolução em nossas vidas atualmente, e isto parece ser só o começo.

Assim sendo vamos voltar no ano de 1943.(ACADEMY, 2022a) Nesta época em que basicamente existiam apenas máquinas gigantes computando trajetórias balísticas, o neurofisiologista *Warren McCulloch* e o matemático *Walter Pitts* inauguraram a teoria das redes neurais propondo um modelo matemático simples para o neurônio biológico. Apesar de alguns percalços iniciais, este modelo iria embasar toda a teoria subsequente.

Em 1952, *Hodgkin* e *Huxley* propuseram um modelo baseado em equações diferenciais não-lineares para explicar o processo de disparo de um neurônio através da ativação de seus canais de sódio e potássio. Entretanto, este modelo era bastante complexo para o ambiente computacional da época, uma vez que modelar cada neurônio em particular, bem como sua dinâmica em conjunto, era extremamente custoso, além de não existirem recursos computacionais suficientes.

A conferência de Verão em *Dartmouth College*, *New Hampshire*, EUA em 1956 deu a largada oficial à criação de um campo de estudos oficialmente conhecido como Inteligência Artificial (IA). Desde esta época, já existia muita polêmica em torno da IA, seja pela suposta presunção de seu nome, seja pelos limites práticos da capacidade de processamento dos computadores que impediam o progresso das pesquisas, o que levava periodicamente, a promessas exageradas e às correspondentes decepções. Baseado nas linhas de pensamento de *McCulloch*, entre os anos de 1957 e 1962, *Frank Rosenblatt* da *Universidade de Cornell (New York)* desenvolveu uma rede de múltiplos neurônios e a denominou como *Rede Perceptron*, o que estabeleceu uma primeira geração de redes neurais artificiais. Basicamente, o *perceptron* era um modelo matemático para um neurônio computacional, mas com a habilidade de trabalhar em conjunto (ou seja, em rede) funcionando como um classificador de padrões de dados linearmente separáveis e saída booleana, mas ainda sem capacidade de aprendizado. Este modelo se baseia no cálculo de uma soma ponderada de entradas, de onde se subtrai um limite e passa um dos dois valores possíveis como resultado.

Nesta mesma época, em paralelo aos trabalhos de *Rosenblatt, Bernard Widrow* da *Universidade de Stanford (EUA)* propôs um algoritmo de aprendizado para um neurônio com ativação diferencial, batizado como *AdaLinE (Adaptive Linear Element)* mas que

também ficou conhecido como *regra Delta*. Novamente, os escassos recursos computacionais da época inviabilizavam este modelo, porém mais tarde, este princípio de treinamento acabou sendo generalizado para redes com modelos neurais mais sofisticados.

Esta discussão científica, com suposições de computadores com um poder de raciocínio e processamento igual ou superior ao do cérebro humano, gerou fortes expectativas à época. Acreditava-se que em pouco tempo estaríamos interagindo com máquinas extremamente inteligentes, utilizando a capacidade de um neurônio computacional em resolver problemas, aplicando-se inclusive algumas regras de aprendizado. Entretanto, em 1969, *Marvin Minsky* e *Seymour Papert* publicaram um artigo questionando a capacidade do perceptron em resolver problemas que envolvem padrões não linearmente separáveis (o problema do *XOR – OU exclusivo*). Esta publicação, além de várias discussões éticas sobre o efeito da IA sobre a vida humana, acabou comprometendo a credibilidade dos estudos realizados em torno das redes neurais à época, o que arrefeceu os ânimos dos investimentos nesta área e fez com que até 1986 surgisse uma época conhecida como o inverno da IA.

Na década de 80 as discussões sobre as redes neurais se reacenderam. Surgiram novas arquiteturas com múltiplas camadas – as MLPs – *Multi Layers Perceptrons* – que aliadas a melhores técnicas de treinamento, como *Backpropagation* – ou a retro propagação de erros – permitiam ao algoritmo computar o erro na saída de um sistema supervisionado e propagar este erro de volta ao longo de toda arquitetura da rede. Apesar de um aprendizado mais lento, uma vez que muitas iterações eram necessárias para este aprendizado, os resultados gerados eram bem mais precisos.

Nos anos 90 outras técnicas de aprendizado de máquina foram adotadas pelos pesquisadores (por exemplo: SVM – *Support Vector Machine*) uma vez que estas técnicas apresentavam bons resultados e eram suportadas por bases teóricas muito sólidas. Este fato fez com que soluções em IA começassem a decolar com sucesso em áreas como reconhecimento de fala, diagnóstico médico, robótica e outras. Em 1996 o *Deep Blue* da IBM conseguiu derrotar o grande campeão de xadrez russo *Garry Kasparov*. Além disto, a Google revolucionou sua ferramenta de pesquisa com a implementação do seu algoritmo de classificação de páginas para exibição de resultados de pesquisa.

(IBM, 2022a) Em 2006, se cunhou pela primeira vez o nome das redes neurais profundas (*Deep Learning*), após a publicação de um artigo (HINTON; SALAKHUTDINOV, 2006) abordando como uma rede neural de várias camadas poderia ser pré-treinada em uma camada por vez. Em seguida, com a descoberta de que com uma base de dados suficientemente grande as redes neurais não precisam de pré-treinamento e as taxas de erro caem significativamente, faz com que, a partir de 2010, ressurja uma onda de interesse pelo uso das arquiteturas de redes neurais e, com isto inicia-se uma explosão na utilização das redes neurais artificiais.

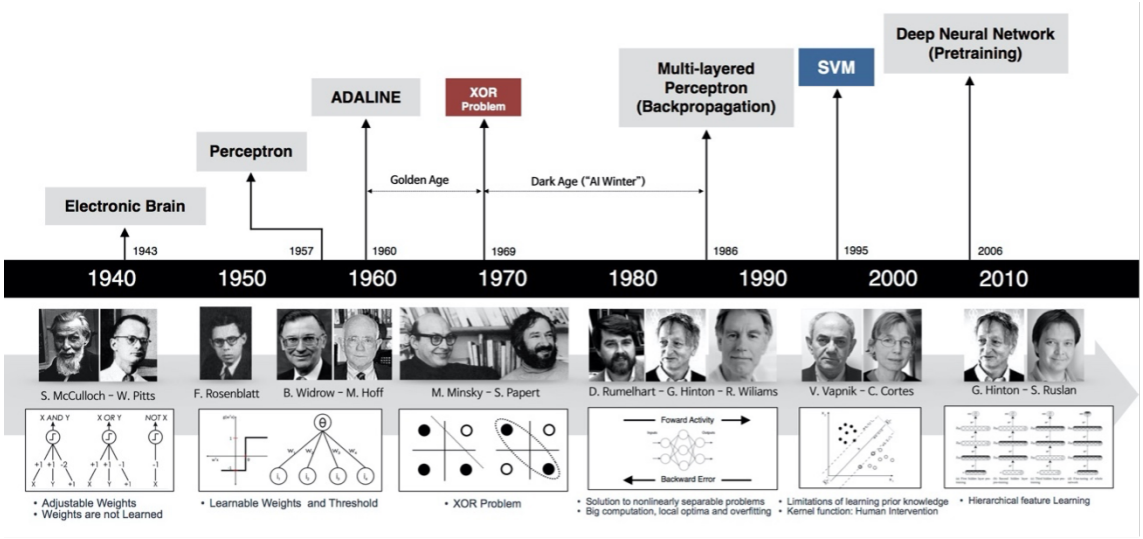
Em 2009, surgiu o banco de dados *ImageNet* composto por um conjunto de

dados inovador que forneceu milhões e milhões de imagens rotuladas, que até então não estavam disponíveis para os profissionais. A partir daí, algoritmos começam a competir em várias tarefas de reconhecimento visual e isso aumenta muito a oportunidade de um algoritmo tentar e aprender com muitos exemplos.

Em 2015, uma das bibliotecas mais populares no aprendizado profundo, a *TensorFlow* foi construída, tornando-a mais poderosa e acessível em 2019.

Em 2018 a Google lança seu primeiro serviço de Taxi sem motorista na cidade de *Phoenix (Arizona) - EUA* e, em 2019, acontece um debate histórico entre a nova IA da IBM, a *Project Debater* contra o campeão mundial de debates em 2016, *Harish Nataranja*.

Figura 1 – Linha do tempo das redes neurais.



Fonte: Imagem extraída da internet – https://www.deeplearningbook.com.br/uma-breve-historia-das-redes-neurais-artificiais.

Apesar da existência de modelos básicos de redes neurais a mais de 30 anos, a utilização prática das mesmas só se tornou efetivamente possível nos anos recentes devido ao progresso tecnológico, uma vez que parte significativa dos resultados obtidos por uma rede neural profunda dependem de poder computacional robusto já que seus algoritmos são baseados em uma quantidade gigantesca de cálculos diferenciais realizados para cada neurônio em toda a rede. Também, os algoritmos utilizados para treino de redes neurais executam operações tipicamente gráficas que foram muito beneficiadas pela evolução das placas de vídeo, que proporcionaram a otimização de parte destas operações gráficas em *hardwares* que permitem o processamento de alto desempenho, as chamadas GPU – *Graphic Processing Unit*. Um outro fato relevante é que o surgimento das redes sociais promoveu um volume gigantesco de dados disponíveis para utilização. Portanto, a união de algoritmos sofisticados com processadores rápidos e um conjunto de dados suficientemente grandes tem tornado possível a realização de tarefas, antes realizadas apenas por seres humanos, como análise de imagens e voz, processamento de linguagem natural, previsão

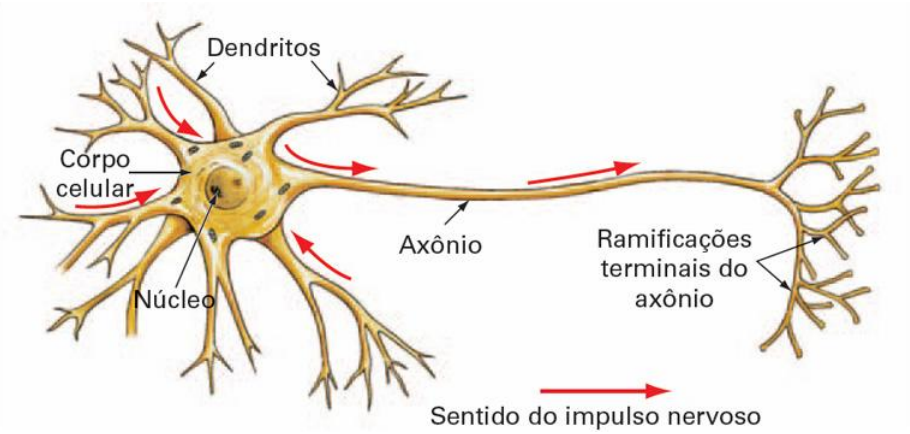
de valores e tomada de decisões.

Apesar da existência de uma gama gigantesca de arquiteturas de redes neurais, existe uma base fundamental utilizada na criação delas. Mesmo com a evolução destas estruturas, com a adição de elementos cada vez mais complexos, a grande maioria delas utiliza o princípio que discutiremos neste trabalho.

Um neurônio artificial é "*uma abstração matemática, baseada em um neurônio biológico, capaz de capturar elementos relevantes para a representação do conhecimento*". Já os "*fenômenos mentais podem ser descritos como uma rede de unidades simples (neurônios) interconectadas*" (COLOMBINI, 2020). A forma como estes neurônios são dispostos, as suas várias camadas, bem como suas entradas e saídas, definem as arquiteturas das redes neurais.

Do campo da biologia, sabemos que sistema nervoso biológico é formado por arquiteturas extremamente complexas compostas por neurônios que carregam informações vitais sobre suas características e responsáveis por desempenhar diferentes funções (ACADEMY, 2022b). Estas células nervosas possuem um corpo celular com dois tipos de ramos: os dendritos e os axônios, conforme representados na Figura 2.

Figura 2 – Neurônio biológico.



Fonte: Imagem extraída da internet – https://www.deeplearningbook.com.br/o-neuronio-biologico-e-matematico.

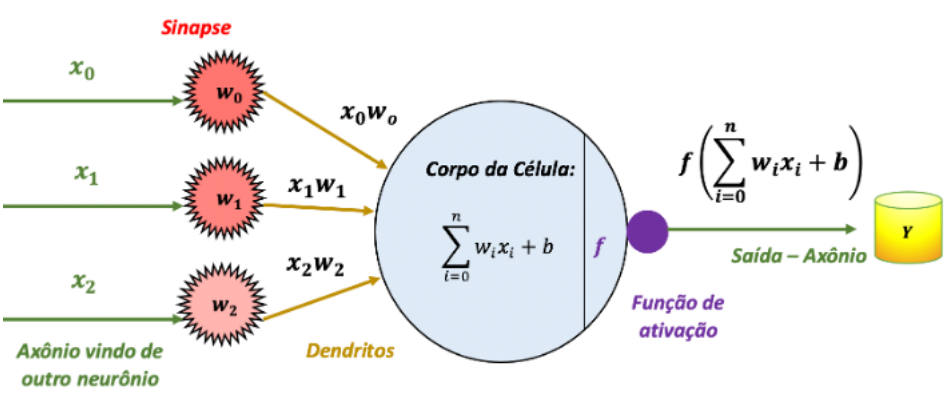
A comunicação entre os neurônios ocorre através de impulsos captados pelos dendritos, responsáveis por receber a informação e repassar para o corpo da célula através do axônio. As ações geradas à partir desta comunicação são as conhecidas sinapses. O axônio recebe sinais a partir do corpo da célula e os transporta para os dendritos que vão repassar para os dendritos de neurônios vizinhos através de sinapses, que podem ser tanto excitatórias quanto inibitórias.

De acordo com (MITCHELL, 1997), o cérebro humano possui em média 10^{11} neurônios. Uma característica interessante sobre eles é que apesar de termos bilhões de

neurônios, o tempo de execução de um neurônio biológico é muito inferior ao tempo de execução de uma unidade lógica num computador. A ordem do tempo de resposta de um neurônio biológico é da ordem de 10^{-3} segundos, enquanto uma porta lógica tem um tempo de resposta da ordem de 10^{-9} segundos. Entretanto, apesar de sua inferioridade em relação ao tempo de reposta, a eficiência do cérebro humano se dá pela quantidade gigantesca de conexões. As interconexões entre os neurônios formam redes neurais complexas, não lineares e paralelas, o que permite uma divisão de trabalho, criando assim um grande diferencial computacional capaz de superar sua baixa velocidade em relação a uma porta lógica.

O modelo matemático gerado a partir do conceito de um neurônio biológico em redes neurais pode ser visto na Figura 3.

Figura 3 – O Perceptron de Rosenblatt.

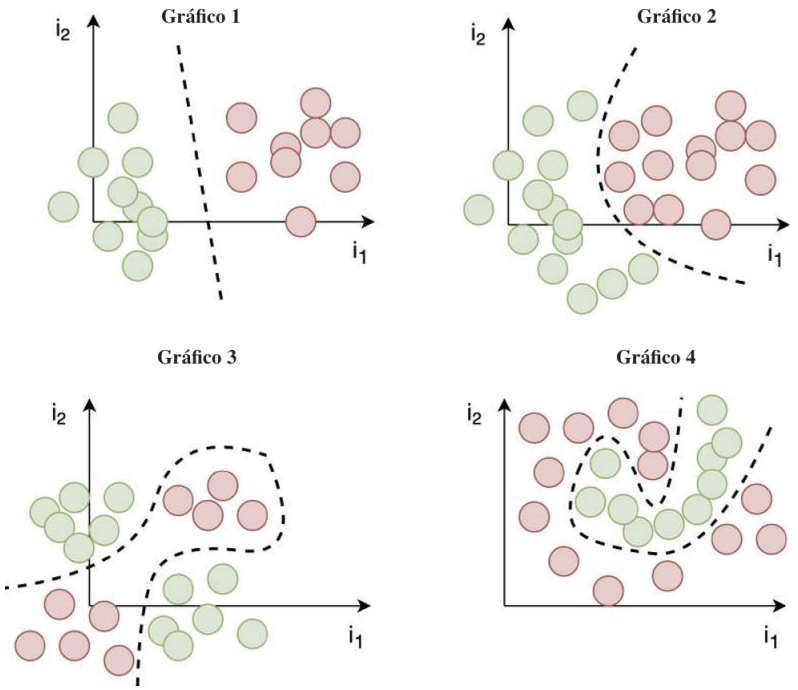


Fonte: imagem adaptada extraída do site https://cs231n.github.io/convolutional-networks.

A entrada de um neurônio (dendritos) se conecta com a saída de outros neurônios (axônios). Os sinais recebidos de outros neurônios são representados por x_i, enquanto as forças de conexão entre os neurônios, denominadas pesos sinápticos, representam o peso de cada sinal para aquele neurônio e são simbolizadas por w_i. O corpo celular elabora uma soma ponderada destas entradas, que é o potencial de ativação, e adiciona uma constante de polarização, também conhecida por *bias*, que representa um limiar de ativação de cada neurônio. Uma vez que esta operação matemática foi realizada, aplica-se uma função de ativação que irá determinar o formato de saída dos dados. Já a saída da função de ativação será o resultado y, que é a saída de cada neurônio. Não existe uma estrutura fixa de um neurônio artificial, apesar de todas se utilizarem do mesmo princípio básico. Existem diferentes formações de neurônios e elas dependem basicamente da função de ativação utilizada, que é o que define a saída dos dados (binária, valor contínuo, pulsos elétricos) bem como a normalização de seus resultados dentro do domínio da função utilizada. Uma configuração de neurônios em rede, ou seja, uma rede neural artificial é capaz de aproximar qualquer função, desde que ela tenha uma topologia e dados suficientes

para seu aprendizado. Na Figura 4, temos 4 classificadores, cada um com uma superfície de separação de dados entre as classes verde e vermelha. A topologia de rede a ser treinada, ou seja, o número de neurônios a serem conectados em rede vai depender da complexidade do problema a ser resolvido. Veja que no Gráfico 1 da Figura 4 temos uma situação bem simples de regressão linear que provavelmente poderia ser resolvido com apenas um neurônio. Já os Gráficos 3 e 4 são separações mais complexas que irão precisar de redes mais robustas, uma vez que a capacidade de aprendizado das redes neurais está representada nos pesos (w_i). Portanto, a topologia da rede e os pesos aprendidos na fase de treinamento é o que capacitam a rede a fazer separações bem mais complexas.

Figura 4 – Diferentes superfícies de separação para classificação.



Fonte: Notas de Aula do Curso Mineração de Dados Complexos - IC Unicamp - 2020 (COLOMBINI, 2020).

1.1 *Arquitetura Multi-Layer Perceptron (MLP)* ou *Feed Forward (FF)*

De acordo com (GOODFELLOW; BENGIO; COURVILLE, 2016a), as MLPs surgiram com o intuito de ser resolver problemas não linearmente separáveis. O primeiro modelo de perceptron, criado em 1958 por *Frank Rosenblatt*, era baseado em uma *Linear Treshold Unit (LTU)*, ou seja, um único neurônio que recebia diversos valores de entrada (x_i), os quais eram multiplicados pelos pesos da sinapse (w_i). O resultado desta soma ponderada somado a um *bias* (viés) era submetido a uma função de ativação. Esta função

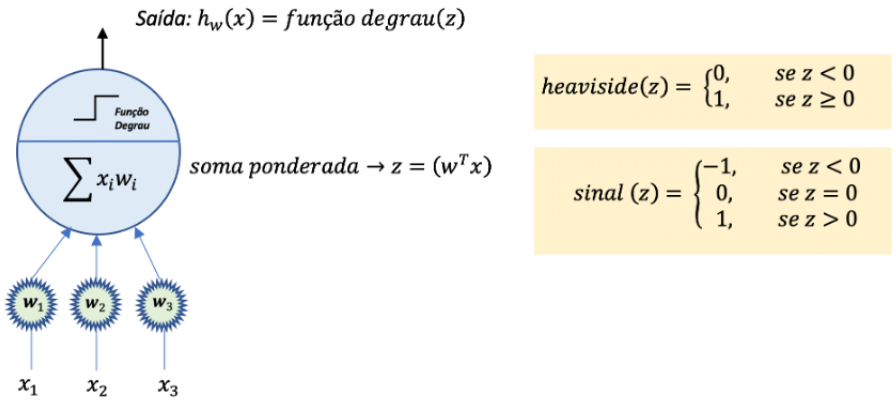
exibia o resultado da classificação, dependendo do limiar atingido (Sim ou Não, Verdadeiro ou Falso) – veja ilustração na Figura 3.

Entretanto, esta limitação em resolver apenas problemas linearmente separáveis frustrou as altas expectativas em termos de revolução tecnológica da época, fazendo com que investimentos em pesquisas de redes neurais permanecessem congelados por um certo tempo. A solução proposta anos mais tarde, que iria permitir soluções de problemas não linearmente separáveis, foi adicionar camadas intermediárias no modelo de Rosenblatt. Neste processo, os neurônios intermediários recebem os valores padrões da entrada, mas são modificados pelos pesos multiplicados a estes valores. Além disto, as saídas dos neurônios das camadas intermediárias são transformadas pela função de ativação, que pode ser linear ou não linear, e este processo ocorre até se chegar aos neurônios de saída da rede, que são os resultados calculados pela rede como um todo. Assim, surgiu a rede neural artificial *Multi Layer Perceptron (MLP)*.

1.2 Função de Ativação

O resultado gerado pela soma ponderada dos neurônios passa para um próximo cálculo chamado de função de ativação. Seu objetivo é normalizar o resultado desta soma ponderada dentro de um intervalo fechado, ou também ser interpretado como a probabilidade do resultado. Além disto, através da função de ativação é possível introduzir um componente de não linearidade, conferindo às redes neurais uma complexidade maior, já que ela se torna capaz de aprender além das relações lineares entre as variáveis dependentes e independentes (COLOMBINI, 2020). Existem várias funções de ativação utilizadas pelas redes neurais, que podem solucionar problemas lineares ou não lineares.

Figura 5 – *Perceptron – Linear Threshold Unit (LTU)* ativada por uma função escada.



Fonte: A Autora (2022).

1.2.1 Funções para problemas lineares

Dentre as funções para problemas lineares mais comuns, discutidas em (COLOMBINI, 2020), pode-se destacar a função degrau, que foi a primeira função de ativação utilizada em operações com neurônios. Sua principal característica é que ela possui um limite de ativação igual a zero. Logo, se a somatória das entradas multiplicadas pelo peso for um valor maior que zero, então, sua saída é ativada, ou seja, a função irá exibir o resultado igual a 1. Esta função é muito utilizada para problemas de classificação binário (sim ou não, verdadeiro ou falso). Sua principal desvantagem é que, por não ser uma função diferenciável, ela impede o treinamento de um neurônio. Existem dois tipos de funções degrau utilizadas: função degrau de *Heaviside* e função degrau sinal. A diferença entre elas é relacionada a ativação do neurônio no caso em que a soma ponderada resulta em zero.

A função degrau de *Heaviside* é definida conforme 1.1:

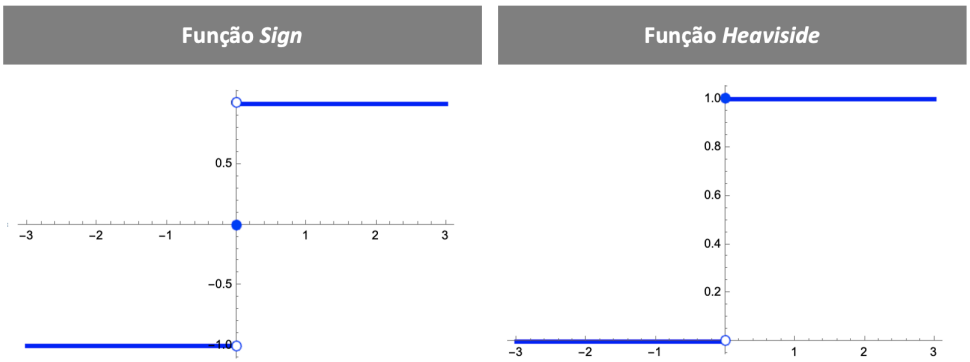
$$Heaviside(z) = \begin{cases} 0, & \text{se } z < 0 \\ 1, & \text{se } z \geqslant 0. \end{cases} \tag{1.1}$$

Já a função sinal é definida conforme 1.2:

$$Sign(z) = \begin{cases} -1, & \text{se } z < 0 \\ 0, & \text{se } z = 0 \\ 1, & \text{se } z > 0. \end{cases} \tag{1.2}$$

Veja que na função degrau sinal se $z = 0$, então a saída deve ser 0, ou seja, nestas condições o neurônio está desligado. Na Figura 6 é possível visualizar os Gráficos de cada uma destas funções degrau.

Figura 6 – Gráficos das funções Sinal e *Heaviside*.



Fonte: A Autora (2022).

1.2.2 Funções para problemas não lineares

Problemas de classificação mais complexos, tais como, identificação de probabilidade entre múltiplas classes, exigem a utilização de funções de características não lineares e diferenciáveis. O fato de uma função ser diferenciável, como veremos logo adiante, é o que confere os atributos de aprendizado das redes neurais. Conforme (NG, 2022e), as principais funções atualmente utilizadas com estas características são:

1.2.2.1 Função logística sigmoidal:

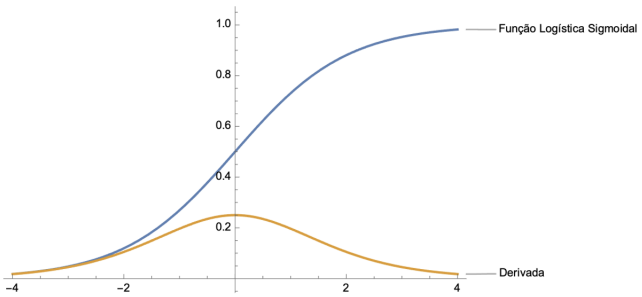
A função logística sigmoidal é uma função real S: R $\rightarrow (0,1)$, definida conforme 1.3:

$$S(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}, \tag{1.3}$$

onde $\theta^T x$ representa o valor obtido na soma ponderada do neurônio.

Trata-se de uma função diferenciável com derivada positiva em cada ponto. Apesar deste atributo permitir o treinamento da rede, já que é a derivada da função que permite o retorno de ajuste dos pesos, podemos observar na plotagem da função e sua respectiva derivada (ver Figura 7) que para os extremos da função, (ou seja, quando ela se aproxima de 0 ou 1) a sua derivada está muito próxima de 0. Isto significa que quando as saídas desta função estiverem próximas das classes correspondentes a rede deixa de aprender, ou então seu aprendizado começa a ficar muito lento. Em outras palavras, o formato gaussiano da função derivada da função sigmoidal, mostra que o aprendizado da rede somente é significativo na região intermediária. Já nos casos de saída perto dos extremos $(0,1)$, o aprendizado torna-se pouco significativo, uma vez que um gradiente atinge valores próximos a zero e o retorno da correção dos pesos torna-se nulo. Sem dúvida alguma, isto gera impactos no aprendizado de redes muito profundas ou com características específicas. Logo, dependendo do cenário, seu uso não é recomendável.

Figura 7 – Representação gráfica da função logística sigmoidal e sua derivada.



Fonte: A Autora (2022).

1.2.2.2 Função Tangente Hiperbólica:

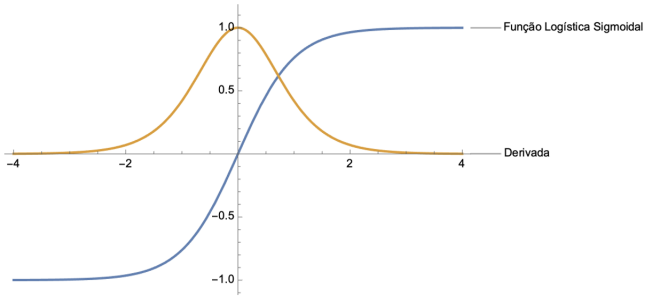
A função Tanh: R $\rightarrow (-1, 1)$ é definida conforme 1.4

$$Tanh(\theta^T x) = \frac{2}{1 + e^{-2(\theta^T x)}} - 1, \tag{1.4}$$

onde $\theta^T x$ representa o valor obtido na soma ponderada do neurônio.

Trata-se de uma função bem similar à função sigmoidal. Sua diferença básica é que ela pode assumir valores positivos e negativos. Na representação gráfica da Figura 8 é possível observar que ela apresenta problemas semelhantes aos da função sigmoidal.

Figura 8 – Representação gráfica da função tangente hiperbólica e sua derivada.

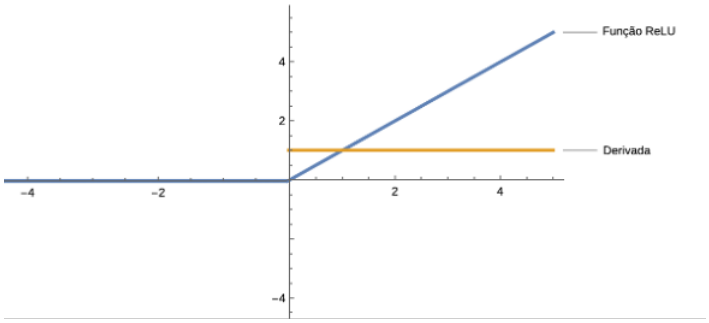


Fonte: A Autora (2022).

1.2.2.3 Função *ReLU* – (*Rectified Linear Unit*):

De acordo com (GÉRON, 2019), a principal característica da função $ReLU$ é que toda soma ponderada menor que zero é zero, e todo valor acima de zero é igual a x. Portanto, trata-se de uma função linear em um domínio onde $x > 0$, logo sua imagem está contida no intervalo $(0, +\infty]$. Por se tratar de uma função linear, sua derivada é uma constante. Apesar de ser uma função parcialmente linear, a sua natureza dentro do aprendizado da rede é não linear, graças ao tratamento dado por ela para todo $x \geqslant 0$. A sua derivada constante para todo $x > 0$ faz com que ela não tenha os problemas descritos na função sigmoidal quanto à perda de aprendizado quando o gradiente se torna muito próximo de zero. Ou seja, a taxa de aprendizado acima de zero irá representar um ajuste significativo nos pesos da rede. A grande desvantagem da $ReLU$ é para os resultados da soma ponderada serem menores que zero. Para esta parcela de resultados não haverá aprendizado na rede, uma vez que não existe derivada para este domínio da função. Dependendo do cenário, isto pode fazer com que uma parte substancial da rede se torne passiva em termos de aprendizado. A figura 9 representa o gráfico da função $ReLU$, bem como sua derivada.

Figura 9 – Representação gráfica da função *ReLU* e sua derivada.



Fonte: A Autora (2022).

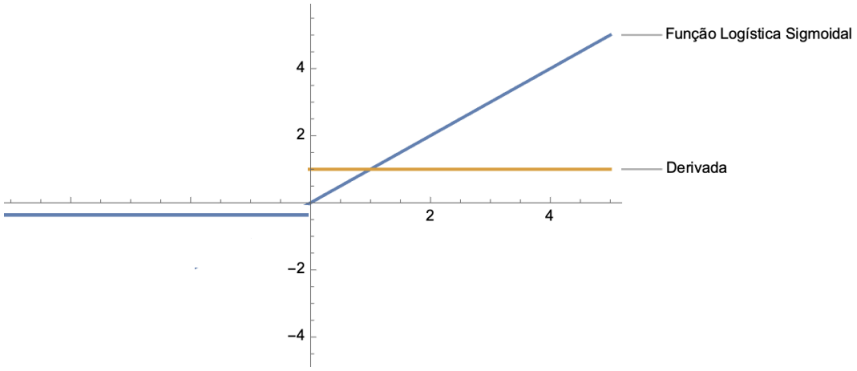
1.2.2.4 Função *Leaky ReLU*:

Esta função surgiu para resolver o problema da *ReLU* no domínio de $x < 0$. A *Leaky ReLU* é definida conforme 1.5:

$$LeakReLU(x) = \begin{cases} -0,01, & \text{se } x < 0 \\ x, & \text{se } x \geqslant 0. \end{cases} \tag{1.5}$$

Segundo (COLOMBINI, 2020), a grande vantagem da *Leaky ReLU* é que, computacionalmente, ela é mais barata em relação à sigmoidal ou tangente hiperbólica, uma vez que ela executa operações matemáticas mais simples. A Figura 10 representa o gráfico da função *Leaky ReLU* e sua derivada.

Figura 10 – Representação gráfica da função *Leaky ReLU* e sua derivada.



Fonte: A Autora (2022).

1.2.2.5 Função Softmax:

Esta função é muito utilizada na última camada de problemas de classificação, uma vez que ela é capaz de formatar o resultado como uma probabilidade da amostra analisada ser de cada uma das classes definidas, por exemplo: 0.1 de chance de ser da classe 1, 0.2 de chance de ser da classe 2, 0.7 de chance de ser da classe 3 (GÉRON, 2019).

A Softmax é definida conforme 1.6:

$$S_i = \frac{e^{z_i}}{\sum e^{z_j}}, \tag{1.6}$$

onde:

- i = índice do neurônio sendo calculado;
- j = todos os neurônios de um nível;
- z representa o vetor dos neurônios de saída.

Portanto, a função de ativação além de formatar o disparo do neurônio, ela também determina a contribuição deste disparo para a correção dos pesos da rede durante o aprendizado, uma vez que a derivada da função de ativação é a responsável por tais correções.

Segundo, (COLOMBINI, 2020), a escolha da função de ativação tem a ver com o problema a ser resolvido e com a profundidade da rede. Em problemas de robótica, especialmente na área de aprendizado de movimento e controle, usa-se bastante a função tangente hiperbólica, uma vez que estas redes não são muito profundas. Já em problemas de análise de imagem, onde a complexidade das redes é bem maior, normalmente usa-se funções do tipo *ReLU* ou *ReLU Leaky*. Um outro ponto relevante para a escolha da função de ativação é se a faixa de valores do resultado deve estar entre 0 e 1, entre –1 e 1 ou entre $-\infty$ e ∞. É importante ressaltar que os algoritmos mais comuns utilizados em aprendizado de redes neurais permitem o uso de diferentes funções de ativação em diferentes camadas da rede.

1.3 Como operam os neurônios dentro de uma rede neural

As primeiras redes neurais que trabalhavam com função degrau não possuíam capacidade de aprendizado. Estes neurônios tinham capacidade de operação, mas não tinham capacidade de aprendizado uma vez que não tinham capacidade de propagar esta informação. Com o uso de funções de ativação diferenciável (ao invés do uso de uma função degrau, utilizar uma função linear, por exemplo) os neurônios tornaram-se capazes de aprender.

Usando uma estrutura simples, igual ao *Perceptron de Rosenblatt*, vamos ilustrar uma operação básica que um neurônio poderia resolver. Para isto, vamos utilizar a função *AND*, onde o resultado é verdadeiro (1) se, e somente se, todas as entradas forem verdadeiras, conforme Figura 11.

Sejam duas entradas $x_1, x_2 \in \{0, 1\}$. Vamos representar a função *AND* para que ela possa nos dar as saídas para cada combinação x_1, x_2. Iniciaremos a operação com pesos

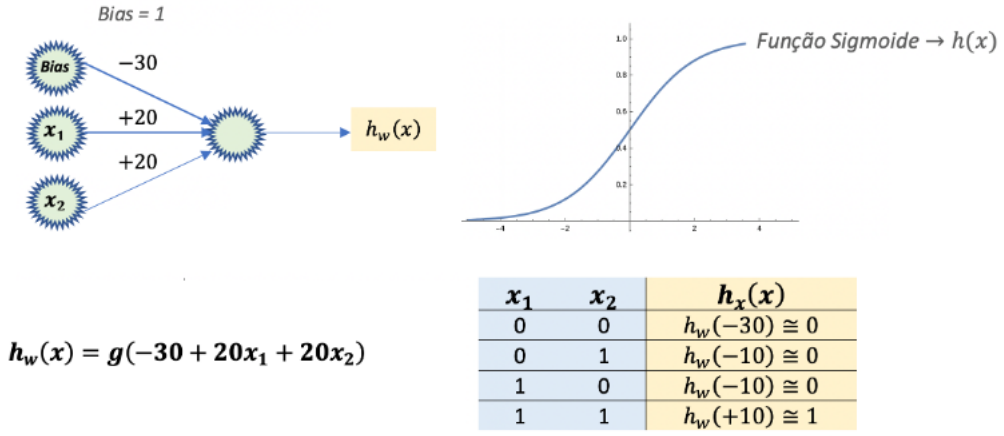
Figura 11 – Tabela Verdade da função AND.

Tabela Verdade And		
x_1	x_2	Resultado
0	0	0
0	1	0
1	0	0
1	1	1

Fonte: A Autora (2022).

pré-determinados $\{-30, 20, 20\}$. Na Figura 12 é possível verificar que com estes pesos é possível representar a função *AND*.

Figura 12 – Operação de um neurônio na representação da função *AND* - 1.



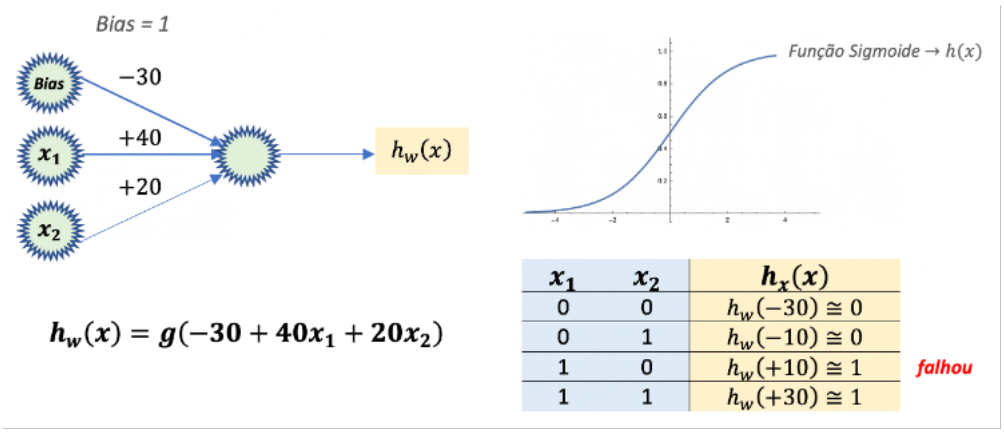
Fonte: A Autora (2022).

Entretanto, podemos ver na Figura 13 que uma simples alteração no peso da entrada x_1 e a hipótese da função *AND* deixa de ser representada por este neurônio, o que significa que os pesos (pré determinados ou aprendidos) é que irão determinar os resultados esperados para cada função.

É interessante notar nos dois exemplos representados pelas Figuras 12 e 13, que o *bias* representa o limiar do neurônio, ou seja, o neurônio dispara sempre que a soma ponderada entre x_1, x_2 é maior que o próprio *bias*. Também, existem infinitas combinações de pesos para x_1, x_2 que poderão fazer com que este neurônio dispare ou não. Portanto, o trabalho da rede neural é aprender quais são os pesos de cada entrada e, o aprendizado somente é possível quando a rede consegue computar a saída e, caso a hipótese esteja errada, ela possa propagar a correção afim de reajustar os pesos e acertar da próxima vez.

Um outro fato relevante já discutido aqui é que o neurônio sozinho só é capaz de separar o que for linearmente separável. Em algumas outras funções lógicas, em particular

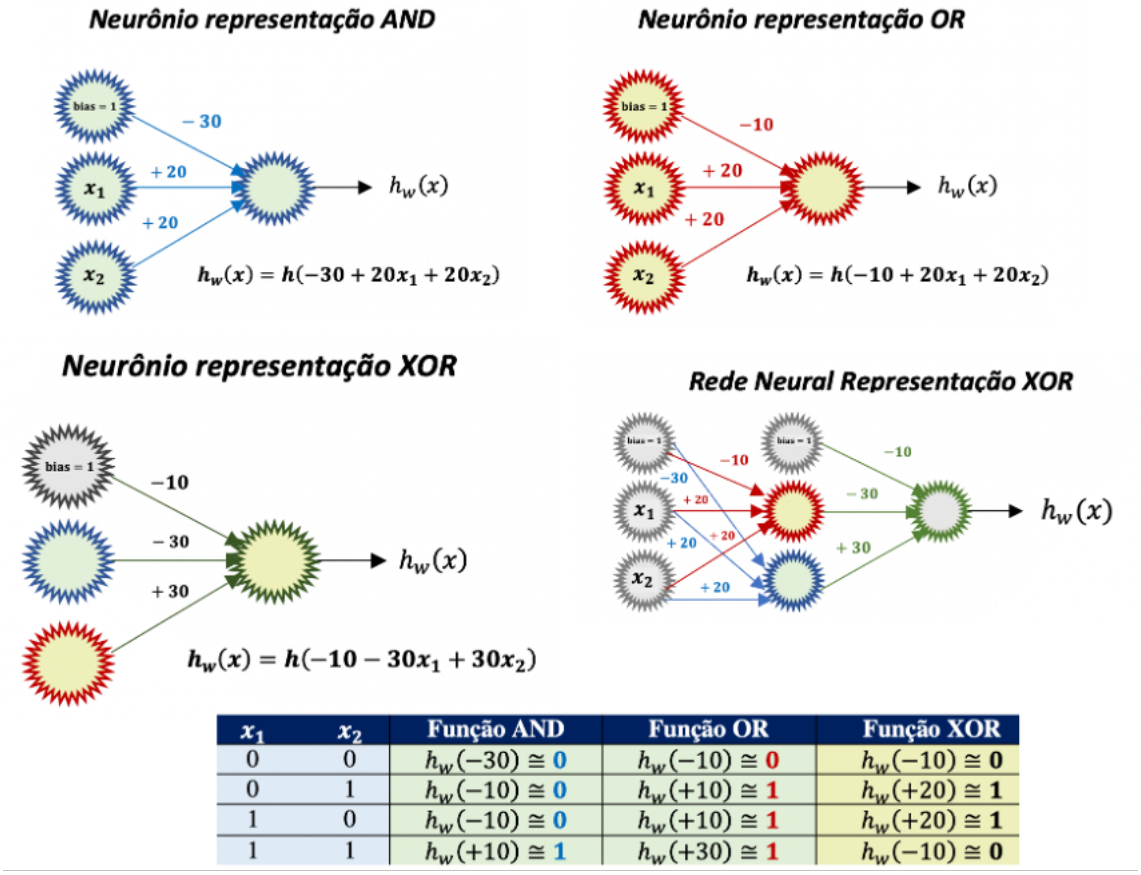
Figura 13 – Operação de um neurônio que não representa a função *AND* - 2.



Fonte: A Autora (2022).

a função *XOR* que não é linearmente separável, não é possível encontrar um conjunto de *bias* e pesos capazes de representar tal função em um único neurônio.

Figura 14 – Combinação de neurônios em rede para representar a função XOR.



x_1	x_2	Função AND	Função OR	Função XOR
0	0	$h_w(-30) \cong 0$	$h_w(-10) \cong 0$	$h_w(-10) \cong 0$
0	1	$h_w(-10) \cong 0$	$h_w(+10) \cong 1$	$h_w(+20) \cong 1$
1	0	$h_w(-10) \cong 0$	$h_w(+10) \cong 1$	$h_w(+20) \cong 1$
1	1	$h_w(+10) \cong 1$	$h_w(+30) \cong 1$	$h_w(-10) \cong 0$

Fonte: A Autora (2022).

A solução para a função *XOR* se dá apenas com uma rede de neurônios onde é possível combinar mais de uma função.

Na Figura 14 é possível visualizar a combinação de 2 neurônios, um representando a função *AND*, outro representando a função *OR* que combinados em rede conseguem representar a função *XOR*.

Este passo de rede neural para a função *XOR* representado na Figura 14 é o passo conhecido por *forward propagation* (propagação para frente) de uma rede neural de arquitetura *Feed Forward* (FF) – ou seja, uma rede que passa a informação adiante no sentido da esquerda para direita, além do fato que os elementos da mesma camada não se conectam.

Dada a propagação dos dados de entrada, para que uma rede neural consiga fornecer uma saída, ou seja, uma hipótese para um determinado problema, é necessário avaliar o quão próxima esta hipótese está do resultado verdadeiro. A partir desta comparação é necessário reajustar os pesos destas entradas, para que da próxima vez a hipótese esteja mais próxima do valor verdadeiro esperado na saída da rede. Uma das primeiras redes que surgiu com capacidade de aprendizado em problemas linearmente separáveis foi a rede *Adaline (Adaptative Linear Entity)*. Era uma rede simples composta apenas de entrada e saída (não existiam camadas intermediárias). Esta rede tinha uma função de ativação linear na saída e seu algoritmo de treinamento lhe permitia considerar o erro de sua hipótese e fazer o reajuste necessário dos pesos. Este algoritmo ficou conhecido com *regra Delta*, mas tinha a limitação de resolver somente problemas linearmente separáveis, já que as redes daquela época ainda não estavam preparadas para problemas não linearmente separáveis. Com a descoberta de que era possível resolver problemas não lineares adicionando camadas intermediárias nas arquiteturas das redes, novos algoritmos de treinamento surgiram, dentre eles, o *backpropagation*, que usa como base a *regra Delta*, mas que é capaz de percorrer as várias camadas intermediárias da arquitetura da rede.

1.4 O embasamento matemático do aprendizado de máquina

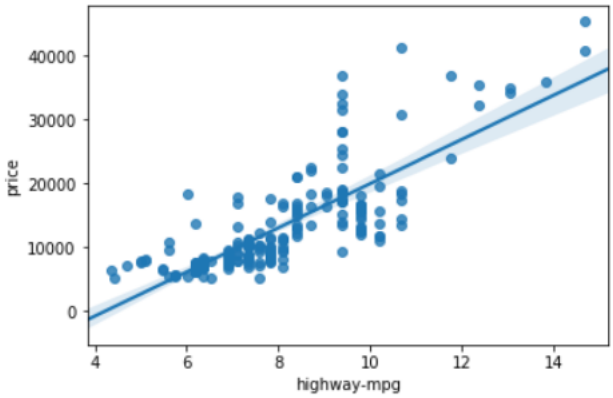
Pode-se dizer que as redes neurais são apenas modelos estatísticos não lineares, muito parecidos com o modelo de regressão linear ou logística. Além das redes neurais, existem inúmeros modelos de aprendizado de máquina. A definição de qual modelo utilizar vai depender basicamente dos dados disponíveis e da resposta que se está buscando.

1.4.1 Regressão Linear

O modelo de aprendizado de máquina mais simples utilizado atualmente é a regressão linear em que, com base em uma entrada x, representada na Figura 15 como o consumo de combustível de um carro medido em *mpg* (milhas por galão), tenta-se definir os parâmetros (a, b) que melhor representam a distribuição dos dados e, a partir disto, traçar uma reta do tipo $f(x) = ax + b$ que tem por objetivo maximizar a probabilidade entre

os dados, especialmente em casos de distribuições normais (GOODFELLOW; BENGIO; COURVILLE, 2016b).

Figura 15 – Visualização gráfica da relação entre consumo de combustível (mpg) e preço (US$) de carros americanos.



Fonte: A Autora (2022).

Assim, o resultado de uma função linear gerado por um algoritmo de aprendizado de máquina tem sua imagem em R^1 e é chamado de hipótese. Ou seja, a hipótese (que é uma variável dependente) é formulada como $h_\theta(x) = \theta_0 + \theta_1 x_1$, onde x (também conhecida por variável independente) representa o valor de entrada dos dados, e θ são os parâmetros que o algoritmo deve encontrar. De acordo com (ROCHA, 2020), existem diversos métodos para se otimizar os valores de θ, entretanto uma das técnicas mais utilizadas é adotar uma função de custo adequada ao problema que está se tentando resolver. Esta função deverá calcular a distância entre as previsões do modelo linear e os exemplos de treinamento; o objetivo é minimizar essa distância. Um dos métodos mais utilizados para o cálculo destas distâncias é computar o erro quadrático médio (GÉRON, 2019), que pode ser definido pela Fórmula 1.7:

$$J(\theta_0, \theta_1) = \sum_{i=1}^{m} (\hat{y}_i - y_i)^2. \tag{1.7}$$

onde:

- m corresponde ao número de amostras submetidas ao algoritmo;
- y é o valor real da amostra que será comparado com a hipótese prevista;
- y é o valor previsto pelo algoritmo, ou seja, trata-se da hipótese;

portanto:

$$\hat{y} = h_\theta(x) = \theta_0 + \theta_1 x_1. \tag{1.8}$$

Então, a fórmula do erro quadrático médio, que a partir de agora vamos chamar de função de custo pode ser reescrita como na Fórmula 1.9:

$$J(\theta_0, \theta_1) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x)_i - y_i)^2. \tag{1.9}$$

A partir do cálculo das perdas é possível obter o gradiente da função de custo, que depende apenas dos parâmetros θ_0 e θ_1 e seu objetivo principal é ajustar os parâmetros na direção oposta ao gradiente calculado até alcançar seu mínimo global. Desta forma, a função de custo é minimizada gradativamente. A velocidade da descida do gradiente também pode ser controlada por um parâmetro α denominado taxa de aprendizado. Esta taxa de aprendizado funciona como um escape em casos de funções com vários mínimos locais, o que as vezes dificulta a minimização da função custo. Trata-se de uma constante, com valor bem baixo com objetivo de fazer a atualização de pesos de forma lenta e suave, evitando assim, grandes passos e comportamento caótico.

O gradiente da função de custo $\nabla J(\theta_0, \theta_1)$ é definido por:

$$\frac{\partial J}{\partial \theta_0} = \frac{2}{m}\sum_{i=1}^{m}(h_\theta(x)_i - y_i). \tag{1.10}$$

$$\frac{\partial J}{\partial \theta_1} = \frac{2}{m}\sum_{i=1}^{m}(h_\theta(x)_i - y_i)x_i. \tag{1.11}$$

onde, 1.10 e 1.11 são as derivadas parciais da função custo em relação aos parâmetros θ_0, θ_1.

O Gráfico da Figura 16 mostra a direção do aumento do gradiente da função de custo definida na Fórmula 1.9. O algoritmo de aprendizado da regressão linear deverá computar o sentido oposto, ou seja, ele deve procurar o caminho para diminuir o gradiente e, com isto, ajustar os parâmetros da reta de forma que esta represente da melhor forma possível os dados disponíveis, de acordo com o modelo que está sendo treinado.

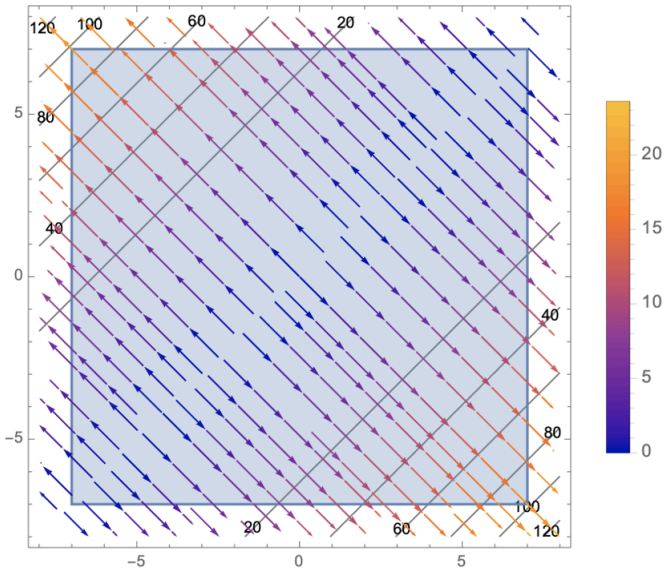
Este modelo de regressão linear com uma única variável independente pode ser expandido para problemas com múltiplas variáveis de entrada. Neste caso, a hipótese pode ser escrita como na Expressão 1.12:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + ... + \theta_n x_n. \tag{1.12}$$

E a função de custo é definida em 1.13:

$$J(\theta_0, \theta_1, ..., \theta_n) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x_n)_i - y_i)^2. \tag{1.13}$$

Figura 16 – Visualização Gráfica do aumento do gradiente da função de custo da regressão linear.



Fonte: A Autora (2022).

Quanto ao gradiente da função de custo para múltiplas variáveis irá depender de quantas entradas x_n irão compor a função de custo. Deve existir uma derivada parcial correspondente a cada característica x_n que componha a função de custo.

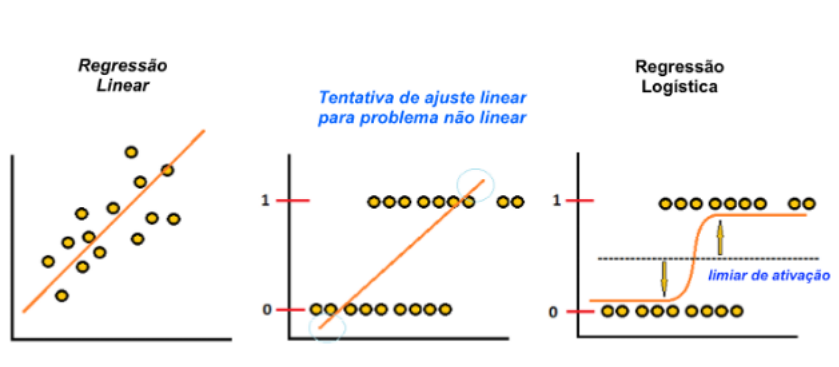
1.4.2 Regressão Logística

Conforme, (ROCHA, 2020), dependendo do que se busca como resposta, o modelo de regressão linear pode não ser o mais adequado, por exemplo, o caso dos dados categóricos onde as respostas precisam estar num intervalo definido entre 0 e 1. Neste caso, a melhor saída é a regressão logística onde o objetivo central é encontrar uma curva em formato *'S'* que consiga se ajustar aos dados, conforme pode ser visto na Figura 17.

De acordo com (NG, 2022b), regressão logística, que é a unidade base das redes neurais, é um classificador extremamente eficiente em que a modelagem da classe (saída) deve ser 0 ou 1. Ou seja, a hipótese da regressão logística é definida como $0 \leqslant h_0(x) \leqslant 1$. Isto porque, como veremos adiante, nas previsões do algoritmo no momento da descida do gradiente, ele sempre vai zerar uma parte da equação e ativar a outra parte.

Assim como na regressão linear, a hipótese da regressão logística é formulada com base em $\theta^T x$, ou seja, dado um conjunto de treinamento com m exemplos (representados em linhas) e n características (representadas em colunas), conforme vemos na Matriz 1.14:

Figura 17 – Visualização gráfica de um problema de classificação binária ajustado por uma reta ou por uma curva.



Fonte: Figura adaptada extraída do site
https://ichi.pro/pt/regressao-logistica-totalmente-explicada-com-python-268974005935565.

$$\begin{cases} \theta_{10}x_{10}, \theta_{11}x_{11}, \theta_{12}x_{12}, ..., \theta_{1n}x_{1n}y_1 \\ \theta_{20}x_{20}, \theta_{21}x_{21}, \theta_{22}x_{22}, ..., \theta_{2n}x_{2n}y_2 \\ \vdots \\ \theta_{m0}x_{m0}, \theta_{m1}x_{m1}, \theta_{m2}x_{m2}, ..., \theta_{mn}x_{mn}y_m, \end{cases} \tag{1.14}$$

e sendo x os valores de entrada e θ os parâmetros que o algoritmo precisa aprender, a representação vetorial deste sistema é conforme 1.15:

$$\theta^T x = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 & ... & \theta_n \end{bmatrix} * \begin{bmatrix} x_0 = 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in R^{n+1}. \tag{1.15}$$

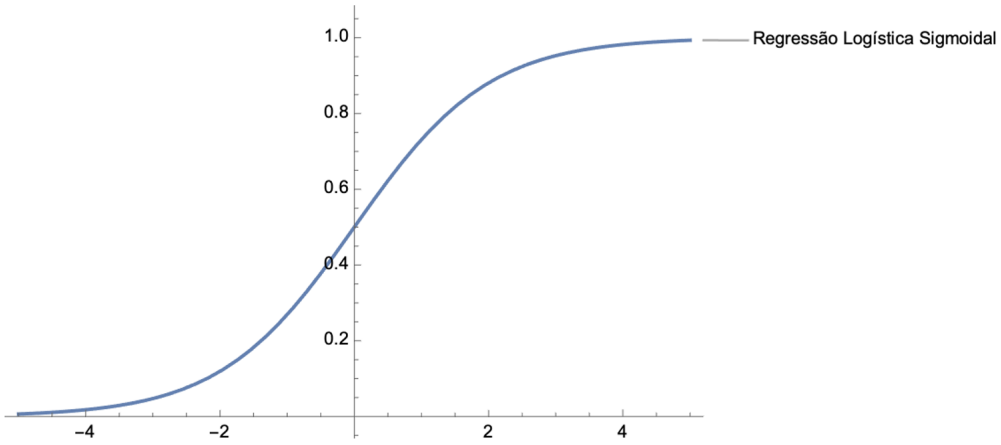
Em seguida, a expressão $\theta^T x$ é submetida a uma função sigmoide g, também conhecida como função logística, que irá limitar sua imagem entre 0 e 1. Esta função é representada em 1.16:

$$g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}. \tag{1.16}$$

Olhando o gráfico da função logística (Equação 1.16), na Figura 18, fica claro perceber que toda vez que a hipótese tiver um valor negativo, a previsão será 0, e toda vez que a hipótese tiver um valor positivo, a previsão será 1. Ou seja, se $\theta^T x \geqslant 0$, então, $g(\theta^T x) \geqslant 0.5$, o que significa que $\hat{y} = 1$. Mas, se $\theta^T x < 0$, então, $g(\theta^T x) < 0.5$, o que significa que $\hat{y} = 0$.

O resultado da hipótese é interpretado da seguinte forma: supondo $g(\theta^T x) = 0.8$, então, $g(\theta^T x) = P(y = 1|x; \theta)$, ou seja, a probabilidade que dado x parametrizado por θ

Figura 18 – Gráfico da função logística.



Fonte: A Autora (2022).

seja positivo (= 1) é de 80%. Portanto, este resultado baseado na teoria das probabilidades é quem irá definir se a previsão será 0 ou 1.

O limiar entre o que é 0 ou 1 é chamado fronteira de decisão. Este limiar é definido na modelagem da função e é o que permite ao algoritmo dar mais ou menos prioridade a uma das classes. Por exemplo, se $g(\theta^T x) \geqslant 0.7$, então a saída deve ser 1 (positiva), caso contrário é 0 (negativa).

1.4.2.1 Função de Custo

Assim como na regressão linear, em regressão logística a otimização dos parâmetros θ se dá pela função de custo. Então, dado um conjunto de treinamento com m exemplos e n características, tem-se a representação matricial $m \times n$, conforme 1.14.

Sendo x os valores de entrada e θ os parâmetros que o algoritmo precisa aprender, a representação vetorial desta soma ponderada está representada em 1.15

A partir da hipótese, baseada nos parâmetros de θ, que é definida conforme Equação 1.16, define-se a função de custo, que nada mais é que o cálculo do erro quadrático médio, conforme expresso na Equação 1.17:

$$Custo(h_\theta(x_i), y_i) = \frac{1}{m}(h_\theta(x_i) - y_i)^2. \tag{1.17}$$

Mas agora, o problema que estamos resolvendo é não linear, portanto, se aplicarmos uma função de regressão linear a estes dados, o resultado será um uma função não convexa e, conforme discutido em (NG, 2022a), isto não é muito adequado para a execução da descida do gradiente, uma vez que o algoritmo terá dificuldades em encontrar o mínimo global de uma função deste tipo. Ao contrário de uma função convexa, onde existe a certeza de convergência para o mínimo global.

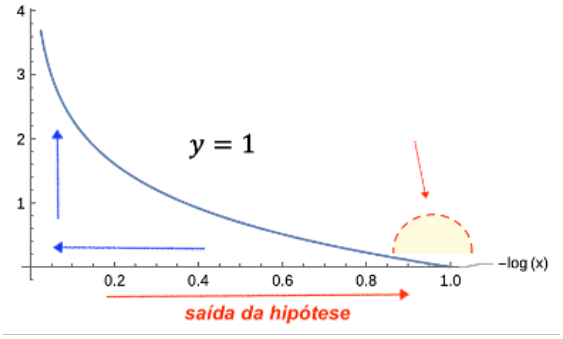
Para contornarmos este problema, podemos definir a função de custo conforme a Expressão 1.18:

$$Custo(h_\theta(x_i), y_i) = \begin{cases} -log(h_\theta(x_i)), & \text{se } y = 1 \\ -log(1 - h_\theta(x_i)), & \text{se } y = 0. \end{cases} \tag{1.18}$$

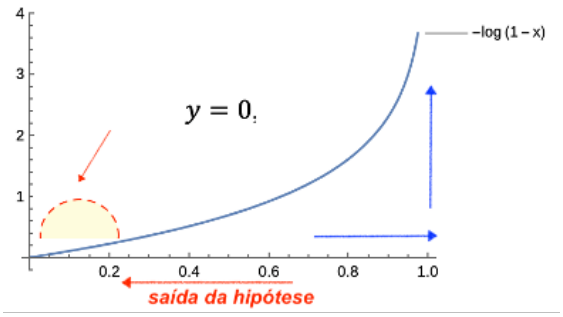
A Figura 19 representa as duas possibilidades desta função de custo, também conhecida como entropia cruzada. (ACADEMY, 2022c).

Figura 19 – Representação Gráfica da Entropia Cruzada.

(a) Gráfico $f(x) = -log(x)$



(b) Gráfico $f(x) = -log(1 - x)$



Fonte: A Autora (2022).

Veja no gráfico da Figura 19a que o valor real é $y = 1$, então, à medida que a hipótese se aproxima de 1, o valor de custo do aprendizado é praticamente zero, pois a curva vai achatando no eixo x à medida que ela se aproxima de 1. Ao passo que se $\hat{y} = h_0(x) = 0$, então a função custo tende a infinito.

Da mesma forma, é possível observar o gráfico da Figura 19b onde o valor real é $y = 0$. Se a hipótese $\hat{y} = h_0(x) = 0$, então o valor do custo será praticamente zero, uma vez que a curva se achata na origem do gráfico. Porém, se $\hat{y} = h_0(x) = 1$, então o custo tende a infinito. É este alto custo que fará com que o algoritmo corrija os seus parâmetros adquirindo assim o aprendizado.

Assim, encontrada uma função adequada à hipótese, é possível compactar as duas linhas em que se define $y = 0$ e $y = 1$ e obter a função de custo, conforme Expressão 1.19:

$$Custo(h_\theta(x_i), y_i) = [-y \cdot log(h_\theta(x_i)) - (1 - y)log(1 - h_\theta(x_i)]. \tag{1.19}$$

Veja que a Equação 1.19 expressa exatamente o problema de classificação, uma vez que a função garante a separação entre as classes 0 e 1, ou seja, se o valor real de y é 1, então a outra parte será desativada e vice-versa e, desta forma, a equação de custo consegue estabelecer exatamente o tamanho do erro da classificação:

Se $y = 0$, então, temos a Expressão 1.20:

$$[-0 \cdot log(h_\theta(x_i)) - (1-0)log(1-h_\theta(x_i)] = -log(1-h_\theta(x_i). \tag{1.20}$$

Se $y = 1$, então, temos a Expressão 1.21:

$$[-1 \cdot log(h_\theta(x_i)) - (1-1)log(1-h_\theta(x_i)] = -log(h_\theta(x_i)). \tag{1.21}$$

Portanto, além do fato desta função de custo utilizar o princípio da estimativa de máxima verossimilhança, ela tem a propriedade de ser convexa, o que é muito eficiente para a descida do gradiente. Isto faz com que esta função seja muito utilizada na modelagem de problemas de regressão logística.

Expandindo esta definição para o caso geral com múltiplos exemplos, a função de custo para regressão logística passa a ser definida conforme Expressão 1.22:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \cdot log(h_\theta(x_i)) + (1-y_i)log(1-h_\theta(x_i))]. \tag{1.22}$$

Uma vez definida a função de custo $J(\theta)$ é hora de otimizar os parâmetros θ. Para isto, utilizaremos o método da descida do gradiente para encontrar o mínimo global da função de custo.

1.4.2.2 Descida do Gradiente

A descida do gradiente é um dos componentes cruciais que tornam o aprendizado dos parâmetros de nossas redes neurais realmente possível (KATHURIA, 2018). Como já visto na regressão linear, o gradiente se dá pelo cálculo das derivadas parciais da função de custo. No caso de n características (ou seja, x_n valores de entrada), será necessário executar um loop para calcular todos os θ_n relacionados as n características para cada um dos m exemplos de treinamento. Portanto, o *loop* da derivada parcial será:

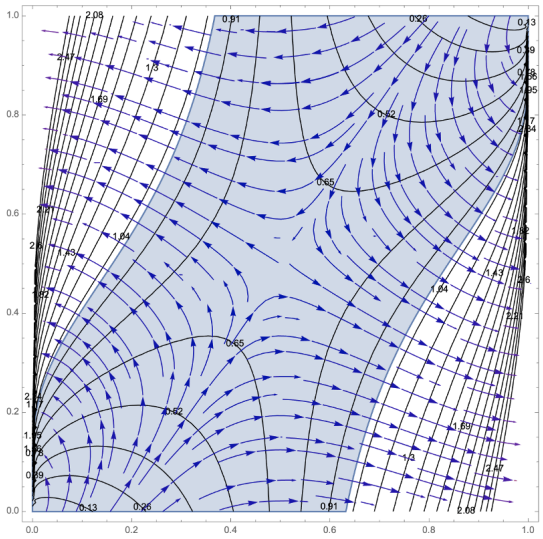
$$\theta_n = \theta_n - \frac{\partial J(\theta)}{\partial \theta_n}, \tag{1.23}$$

sendo $n = 0, 1, ..., n$

Encontrada as derivadas parciais da função de custo, o ajuste dos parâmetros da função de custo se dá no sentido oposto ao crescimento do gradiente.

Na Figura 20, plotamos no *Wolfram Mathematica* as curvas de nível da função de custo, destacando-a dentro de uma região plana de domínio um pouco menor do que será utilizado, ou seja, $[0.1, 0.9]$. É possível observar que os pontos de máximo da função

Figura 20 – Gráfico do fluxo do gradiente da função de custo da regressão logística.



Fonte: A Autora (2022).

se encontram nos extremos laterais, na parte central da região plana traçada. Também é possível perceber os pontos de mínimo nos extremos $[0, 0]$ e $[1, 1]$, que são os pontos de acerto do algoritmo de aprendizagem, e que, portanto, a função deverá ter custo zero. O Gráfico de fluxo demonstra as direções para onde o gradiente aumenta. Ou seja, as correções dos parâmetros devem ser feitas em sentido contrário a este fluxo do gradiente, de forma a minimizar as perdas da função de custo. Portanto, o gráfico da Figura 20 nos dá a intuição de que a descida do gradiente deve ir em sentido as direções extremas, ou seja, $[0, 0]$ para classe negativa, e $[1, 1]$ para classe positiva.

O gradiente descendente é um dos algoritmos mais populares para otimização de redes neurais. De acordo com a literatura, (HINTON; SRIVASTAVA; SWERSKY, 2012), (RUDER, 2018), (AVILA, 2020) existem 3 variações distintas do gradiente descendente que diferem apenas na forma de atualização dos parâmetros da função de custo. Vejamos:

1.4.2.2.1 Descida do gradiente estocástico (SDG)

Nesta variação os parâmetros da rede são ajustados a cada exemplo passado pelo algoritmo. O grande inconveniente deste tipo de otimização são as atualizações de pesos frequentes e com alta variância, causando assim, grande flutuação da função de custo. O reajuste dos parâmetros θ com o método de descida do gradiente estocástico se dá pela Expressão 1.24:

$$\theta = \theta - \alpha(\theta, x_i; y_i). \tag{1.24}$$

1.4.2.2.2 Descida do gradiente *batch (Vanilla)*

Neste caso, os ajustes dos parâmetros são feitos após o processamento de todo o conjunto de dados a cada época. Uma época é definida como uma passagem completa (ida e volta, ou propagação e retro-propagação) por todos os exemplos do conjunto de treinamento. Este processo é extremamente lento e, portanto, não aplicável em base de dados muito grandes. Nesta variação, o reajuste dos parâmetros θ se dá pela Expressão 1.25:

$$\theta = \theta - \alpha(\theta). \tag{1.25}$$

1.4.2.2.3 Descida do gradiente *mini batch*

Aqui os ajustes dos parâmetros são feitos após o processamento de subconjuntos (mini lotes) de dados. O tamanho dos mini batchs são configurados no algoritmo. Este é o método mais utilizado no treinamento de redes convolucionais, uma vez que ele reduz a variância da atualização levando a uma convergência mais estável. O reajuste dos parâmetros θ se dá pela Expressão 1.26:

$$c : \theta = \theta - \alpha(\theta, x_i : x_{i+c}; y_i : y_{i+c}). \tag{1.26}$$

Apesar da descida do gradiente ser um método de otimização poderoso, ele enfrenta alguns desafios em processos de base de dados reais onde a representação espacial dos dados não tem uma curva perfeita com um mínimo global facilmente identificado. Em muitos processos de treinamento, o gradiente pode ficar preso em mínimos locais limitando assim, o aprendizado do modelo. Muitas vezes, este problema pode ser resolvido pela escolha de uma taxa de aprendizado adequada (α), mas nem sempre isto é suficiente.

1.4.2.3 Métodos de otimização da descida do gradiente

Existem, na literatura, diversas propostas para contornar este desafio, ou seja, estas propostas tentam otimizar o processo de descida do gradiente. Destacamos aqui algumas destas propostas discutidas em várias publicações pesquisadas, tais como: (CHOLLET, 2019), (RUDER, 2018), (AVILA, 2020)

1.4.2.3.1 *Momentum*

Este método foi proposto para acelerar o gradiente descendente estocástico nas direções relevantes, e amortizar possíveis oscilações em direções não relevantes. O momentum é definido pela Expressão 1.27:

$$V_t = \gamma V_{t-1} + \alpha \nabla J(\theta). \tag{1.27}$$

$\theta = \theta - V_t$, ou seja, uma fração γ da atualização anterior é adicionada à atualização atual. Empiricamente, constatou-se que $\gamma = 0.9$ é um valor bem aceitável na maioria dos problemas a serem resolvidos. (HINTON; SRIVASTAVA; SWERSKY, 2012).

1.4.2.3.2 *Adagrad*

Este otimizador proposto por (DUCHI; HAZAN; SINGER, 2011) permite a utilização de diferentes taxas de aprendizado (α) para cada parâmetro θ_i a cada passo t (parâmetro θ_t, i). Portanto,

$$\theta_{t+1}, i = \theta_t, i - \frac{\alpha}{\sqrt{G_{t,ii} + \epsilon}} \cdot \nabla J(\theta_{t,i}), \tag{1.28}$$

onde:

- ϵ é uma constante adicionada para evitar que o denominador seja igual a zero.
- $G_{t,ii}$ é a matriz onde cada elemento da diagonal é a soma dos quadrados dos gradientes em relação a θ_t até o passo t.

A grande vantagem do otimizador *Adagrad* é que não é necessário configurar a taxa de aprendizado (α), uma vez que ele vai ser automaticamente ajustado pelo algoritmo no decorrer das necessidades do treinamento. A desvantagem é que a soma positiva ocorrida no denominador desta equação, pode fazer a taxa de aprendizado encolher até chegar a zero e, a partir deste momento, a rede não irá continuar aprendendo. Outros otimizadores, tais como, *Adadelta* ou *RMSprop* surgiram com a proposta de corrigir alguns inconvenientes do *Adagrad*.

1.4.2.3.3 ADAM (*Adaptive Moment Estimation*)

Um dos otimizadores mais utilizados é o ADAM (*Adaptive Moment Estimation*). Este método também se baseia no cálculo de taxa de aprendizado adaptada para cada parâmetro, só que agora, considerando a média e a variância dos gradientes até o passo t (KINGMA; BA, 2015). Este otimizador é definido por:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t, \tag{1.29}$$

onde:

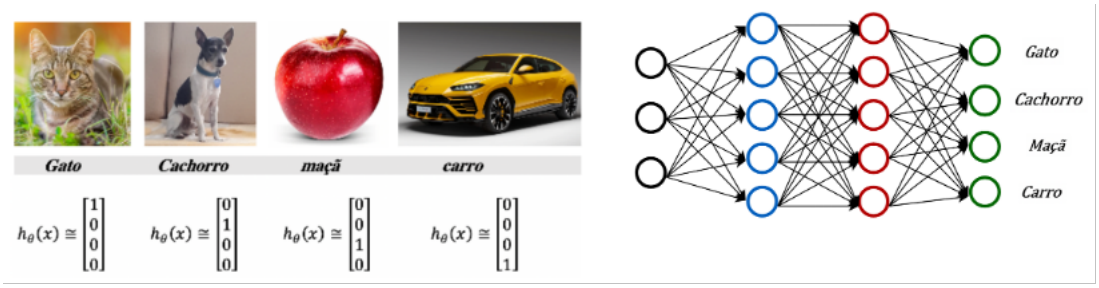
- $\hat{m}_t$ é a média dos gradientes até o passo t;
- $\hat{v}_t$ é a variância dos gradientes até o passo t.

Existe uma gama enorme de otimizadores para a descida do gradiente e, regularmente, novas idéias e novas propostas surgem na literatura.

1.5 Treinamento das Redes Neurais

De uma forma bem resumida, podemos dizer que um neurônio é, na verdade, uma unidade de regressão logística. Portanto, as redes neurais é um emaranhado complexo de milhares de unidades de regressão logística funcionando numa certa ordem determinada por sua arquitetura (NG, 2022c). Logo, as redes neurais também usam o método de calcular os parâmetros θ relacionados a cada uma das entradas x_i e chegam a uma previsão $\hat{y}$ através de uma função de custo cuidadosamente escolhida para o problema que se quer resolver. Comparando este resultado $\hat{y}$ ao y real é possível computar um erro. Este erro deve ser retro propagado pela rede na proporção e direção contrária determinada pelo gradiente da função que gerou o resultado $\hat{y}$. Entretanto, a regressão logística se propõe a resolver apenas problemas de classificação binária, ou seja, duas classes apenas. Nas redes neurais, este problema se torna mais complexo, uma vez que o objetivo agora é trabalhar com K classes. Além disto, *um dos fatos mais impressionantes sobre redes neurais é que elas podem computar qualquer função* (ACADEMY, 2022d). Para provar tal afirmação teríamos que abordar os teoremas da universalidade, mas isto foge à proposta deste trabalho. Contudo, podemos ter esta intuição analisando a Figura 14.

Figura 21 – Problema de classificação de múltiplas classes em redes neurais.



Fonte: Imagem adaptada extraída das Notas de Aula do Curso Mineração de Dados Complexos - IC Unicamp - 2020 (COLOMBINI, 2020).

A Figura 21 representa um exemplo do método conhecido como *One Hot Encoding*. Neste exemplo, temos 4 vetores de comprimento igual ao número de classes, e cada um destes vetores é representado por 1 na posição onde a imagem é verdadeira. Por exemplo: o segundo vetor representa o cachorro, sendo 1 a resposta verdadeira, nosso vetor é formado por 1 na segunda posição e zero nas outras, o mesmo se dá com os outros vetores que representam as outras categorias.

Observe que na camada de saída da rede existem 4 neurônios, sendo que cada um deles indica uma classe para o problema, e o objetivo é que aqueles neurônios com classe diferente do resultado real tenha um erro associado a ele (NG, 2022c).

Já a generalização da função de custo da regressão logística para esta nova situação de múltiplas classes se dá através da Equação 1.30:

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m}\sum_{i=1}^{k} y_k^{(i)} log(h_\theta x^{(i)})_k + (1 - y_k^{(i)}) log(1 - (h_\theta x^{(i)})_k)]. \tag{1.30}$$

Novamente, o aprendizado da rede se dá pela retro-propagação do erro gerado nesta previsão. A retro propagação acontece na proporção e direção oposta ao aumento do gradiente calculado sobre a função de custo.

Já em casos de problemas de regressão, onde o objetivo é aprender um valor real (similar a regressão linear), a computação do erro se dá através da Fórmula 1.31:

$$erro(MSE) = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2, \tag{1.31}$$

onde:

- m é o número de exemplos a serem treinados;
- $\hat{y}$ é o valor previsto;
- y é o valor real esperado.

Neste caso, o erro retro propagado também se dá pelo gradiente da função de custo, que agora é uma função linear.

1.5.1 Exemplo de treinamento de uma função de custo

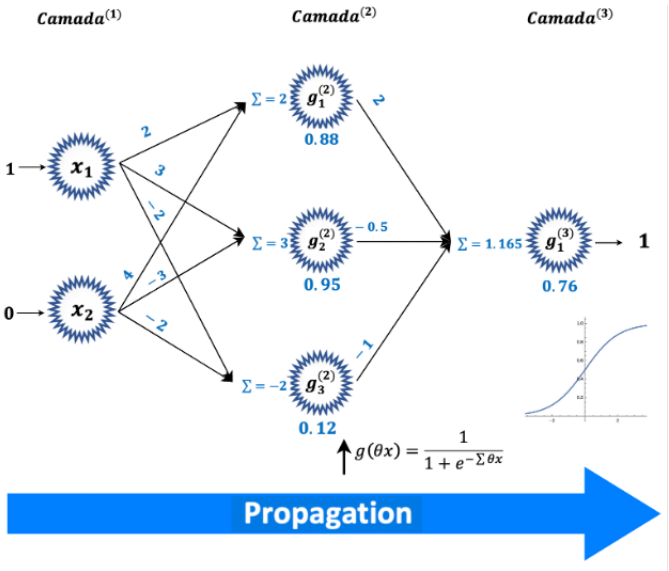
Na Figura 22, tem-se uma ilustração bem simples dos cálculos de uma rede neural criada à partir das explicações de (COLOMBINI, 2020). Obviamente, uma rede neural costuma ser bem mais complexa, mas para efeitos ilustrativos vamos analisar apenas uma ramificação desta rede para facilitar os cálculos.

No diagrama da Figura 22 temos as entradas $x_1 = 1, x_2 = 0$. Cada conexão de um neurônio com a camada seguinte existe um peso associado. A soma ponderada de cada neurônio é submetida a função de ativação, neste caso, por se tratar de um problema de classificação, vamos utilizar a função sigmoide, conforme Equação 1.32:

$$Z_\theta^{(L)} = \frac{1}{1 + e^{-\sum \theta x}}. \tag{1.32}$$

O resultado do valor da função de ativação corresponde à entrada da camada seguinte. Como estamos utilizando uma função sigmoide na função de ativação é esperado que o resultado transmitido para a próxima camada esteja dentro do intervalo $[0, 1]$. Neste caso, é necessário definir um limiar de ativação para a camada final, ou seja, à partir de

Figura 22 – Cálculo de propagação de uma rede (esquerda para direita).



Fonte: A Autora (2022).

qual resultado da última função de ativação, o resultado apresentado pela rede deve ser considerado 0 ou 1 (falso ou verdadeiro). No exemplo da Figura 22, adotamos um limiar de ativação padrão de = 0.5. Como o resultado da última função de ativação foi de 0.76, o resultado final calculado pela rede foi 1, ou seja, $\hat{y} = 1$.

Agora, supondo-se que o resultado esperado fosse 0, ou seja, $y = 0$, então é necessário calcular o erro e retro propagá-lo na rede. Neste caso, temos que ajustar os valores dos parâmetros da saída para a entrada.

De acordo com (LIMA, 2016), dada a função sigmoide $z(x) = \dfrac{1}{1+e^{-x}}$, utilizamos a regra do quociente para calcular sua derivada, ou seja:

$$\frac{d}{dx} \cdot \frac{f(x)}{g(x)} = \frac{f'(x) \cdot g(x) - f(x) \cdot g'(x)}{g(x)^2}, \tag{1.33}$$

portanto:

$$z'(x) = \frac{0 - (-e^{-x})}{(1+e^{-x})^2} = -e^{-x}(1+e^{-x})^2. \tag{1.34}$$

$$z'(x) = \frac{1+e^{-x}-1}{(1+e^{-x})^2} = \frac{1+e^{-x}}{(1+e^{-x})^2} - \frac{1}{(1+e^{-x})^2}. \tag{1.35}$$

$$z'(x) = \frac{1}{(1+e^{-x})} - \frac{1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\left(1 - \frac{1}{1+e^{-x}}\right), \tag{1.36}$$

que também pode ser escrita conforme Expressão 2.2:

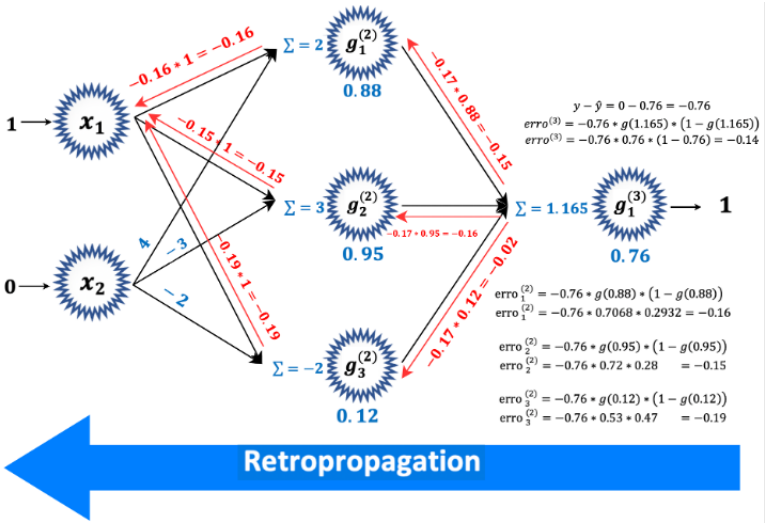
$$z'(x) = z(x) \cdot (1 - z(x)). \tag{1.37}$$

Então, o erro da camada de saída deve ser calculado como a diferença entre o y real e o $\hat{y}$ previsto multiplicado pela derivada.

$$\text{erro}^{(3)} = y - \hat{y} = 0 - 0.76 = -0.76.$$
$$z'^{(3)}(x) = z(0.76) \cdot (1 - z(0.76)) = 0.22.$$
$$\text{erro} \cdot z'(x) = -0.76 \cdot 0.22 = -0.17.$$

Com o erro calculado da camada 3, agora é necessário retro propagar este erro para a camada 2, multiplicando o valor deste erro (-0.17) pelo valor de ativação de cada neurônio da camada 2.

Figura 23 – Cálculo de retro propagação de uma rede (direita para esquerda).



Fonte: A Autora (2022).

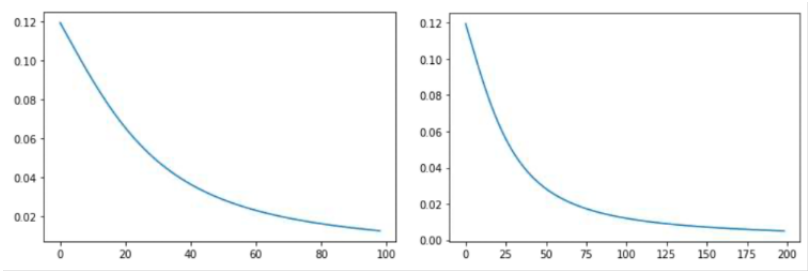
Na sequência, temos que retro propagar o erro da camada 2 para a camada 1. Novamente, vamos calcular a derivada $z'(x)$ para o valor de ativação de cada neurônio da camada 2 e multiplicar pelo erro encontrado no final da rede (0.76).

Em seguida, vamos distribuir este erro para a camada 1 de acordo com sua contribuição no erro total determinada por sua derivada. Estes cálculos estão representados na Figura 23.

Se entrarmos na rede com o mesmo exemplo, agora com os parâmetros ajustados para os valores encontrados no cálculo da Figura 23, chegaremos à uma saída $\hat{y} = 0.46$ e, a cada entrada, este valor será refinado um pouco mais. Entretanto, a taxa de correção tende a cair a cada iteração.

Na Figura 24, é possível visualizar graficamente que depois de 100 ou 200 iterações o ganho é pouco significativo.

Figura 24 – Gráfico de performance da rede depois de 100 e 200 iterações.



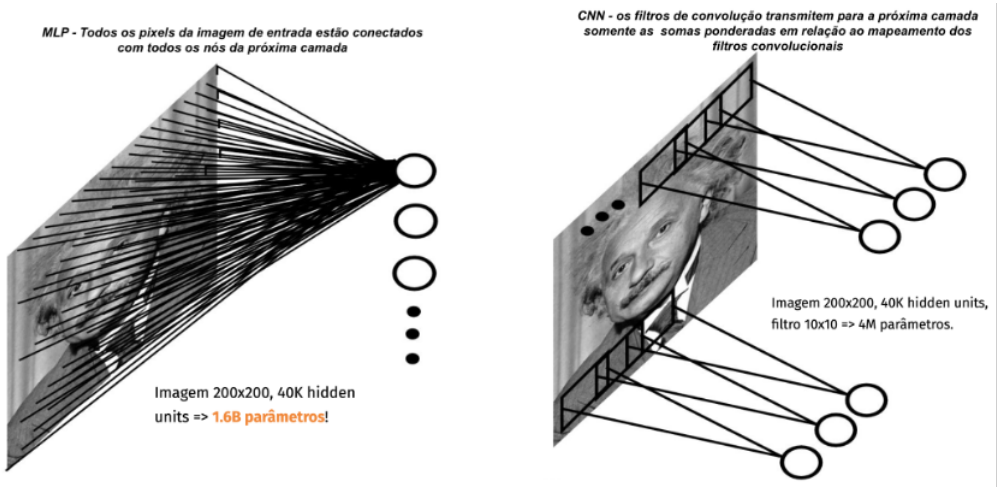
Fonte: Notas de Aula do Curso Mineração de Dados Complexos - IC Unicamp - 2020 (COLOMBINI, 2020).

1.6 A arquitetura das Redes Neurais Convolucionais (CNN)

Conforme vimos anteriormente, as redes neurais MLP (*Multi Layer Perceptron*) são totalmente conectadas, ou seja, todos os neurônios de uma camada, estão conectados a todos os neurônios das camadas seguintes. Em muitos casos esta configuração é muito benéfica à solução do problema sendo capaz de resolver problemas não lineares e de extrema complexidade.

Entretanto, conforme (AVILA, 2020), em casos como por exemplo, classificação de imagens, este processo pode ser extremamente custoso mas de desempenho limitado, uma vez que a modelagem de uma imagem de entrada na arquitetura MLP, transforma a entrada em um dado unidimensional, ou seja, se a entrada for uma imagem de $32 \times 32 \times 3$ (32×32 de dimensão com 3 camadas de cores RGB), esta entrada é transformada em um vetor de 3072×1 e isto, em casos de imagens pode ser um problema, já que a informação espacial pode ser perdida neste enfileiramento unidimensional de pixels e, muitas vezes, o contexto da imagem onde ela está representada também é importante para sua classificação.

Figura 25 – Conexão entre neurônios: MLP x CNN.



Fonte: Iimagem extraída da internet
https://www.cs.toronto.edu/~ranzato/files/ranzato_CNN_stanford2015.pdf.

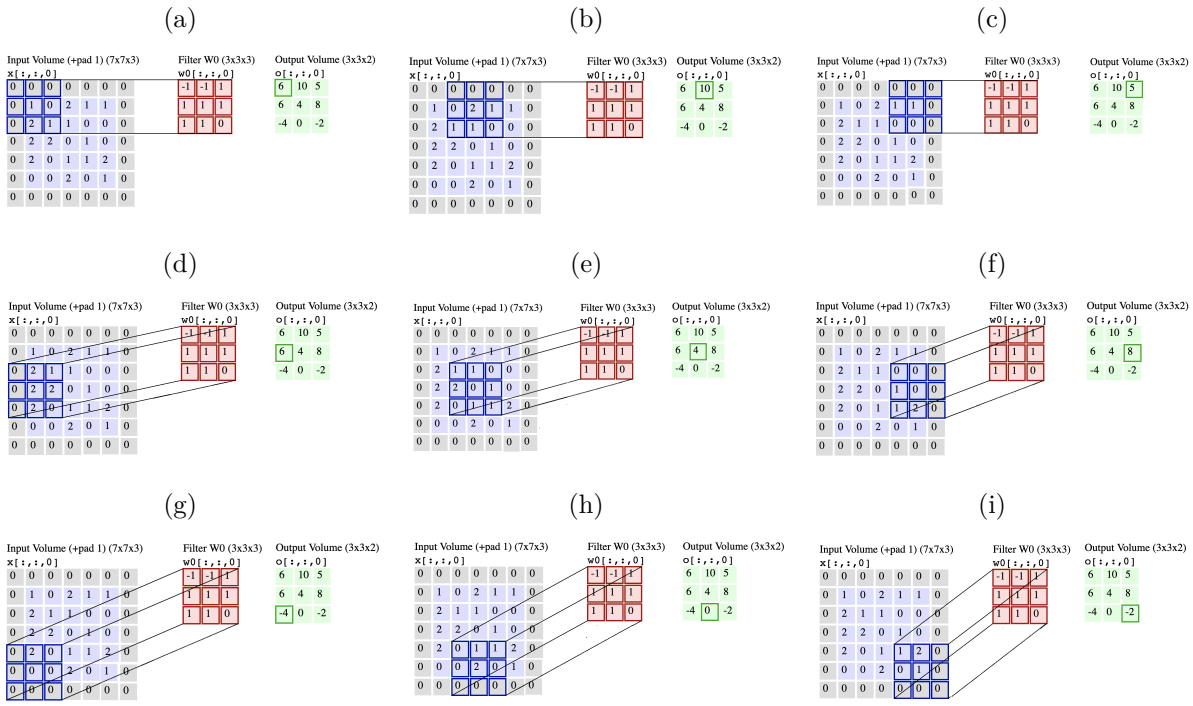
Com o intuito de resolver problemas como este, surgiram as redes convolucionais (CNN – *Convolutional Neural Network*). Tais redes efetuam uma operação de filtragem no processo de leitura (ou seja, é feita uma operação de convolução) em todas as camadas fazendo com que, ao invés de produzir uma região esticada, seja produzida uma estrutura parcial, mas preservando as características espaciais da imagem. A Figura 25 mostra a diferença de conexão entre os neurônios nas redes CNN e MLP.

Tecnicamente falando, convolução é um operador linear que aplica um filtro (*kernel*, ou máscara de convolução) a uma imagem gerando uma imagem filtrada. O objetivo principal destes filtros é identificar diferentes padrões dentro da imagem, tais como: existência de formas, bordas ou diferentes texturas, ou ainda, se existem objetos na imagem, ou se a densidade de pixels variam devido à iluminação e contraste (ANZAI, 2012). Estas características são aprendidas nas camadas intermediárias da arquitetura desta rede. E assim, a rede pode, por exemplo, começar aprendendo bordas, que evoluem para formas e, em seguida, capturam as texturas da imagem. À medida que cada uma destas camadas se conectam entre si, a relação entre os pixels destas camadas, torna-se uma ligação muito mais apurada entre cada um destes detalhes, compondo assim uma imagem muito mais real do que uma imagem totalmente esticada em um único vetor. Em termos bem simples, podemos dizer que os filtros de convolução são, na verdade, detectores de recursos dentro das imagens e que, as diversas camadas da rede identificam a intensidade de conexão entre estes recursos preservando posicionamento de cada detalhe, bem como garantindo que ele seja invariável em qualquer escala.

Na Figura 26 é possível visualizar a representação de uma imagem de entrada (quadro em azul) e uma matriz menor denominada *matriz de convolução* (quadro em vermelho). Após escolher um ponto de centralização entre a imagem de entrada e o filtro de convolução, este filtro é passado por toda a imagem, gerando uma soma ponderada de cada área convolucionada. E o resultado desta soma ponderada torna-se um elemento da matriz de saída (quadro em verde).

O processo de aprendizado da CNN se inicia assim que a imagem de entrada é capturada. Neste momento, inicia-se o pré-processamento da imagem através da utilização dos filtros de convolução, que são capazes de atribuir pesos sobre vários aspectos distintos da imagem, diferenciando uns dos outros. Enquanto nos métodos anteriores os filtros são feitos à mão, as redes convolucionais, com treinamento suficiente, são capazes de aprender estes filtros e extrair estas características que diferenciam as imagens, tornando-se uma ferramenta extremamente poderosa em processos de reconhecimento de imagens. Além disto, uma outra grande vantagem das CNN é que enquanto as MLP conectam cada pixel da camada de entrada com todos os neurônios da camada seguinte, as CNN irão transmitir somente a soma ponderada em relação ao filtro de cada região da imagem esquadrinhada, o que torna esta arquitetura bem menos custosa em termos de recursos computacionais,

Figura 26 – Processo de convolução de uma imagem.



Fonte: imagem adaptada extraída do site https://cs231n.github.io/convolutional-networks.

uma vez que ela reduz significativamente o número de parâmetros a serem treinados, especialmente em casos de imagens de alta resolução.

Ainda sobre as redes convolucionais, vamos entender alguns aspectos de como ela é constituída, já que isto é fundamental no momento de determinar algumas variáveis antes de rodar seus algoritmos.

Tipicamente, uma CNN é composta por:

- Camadas convolucionais, seguidas por uma função de ativação.
- Camadas de *pooling*.
- Camadas de *Fully Connected* (FC).

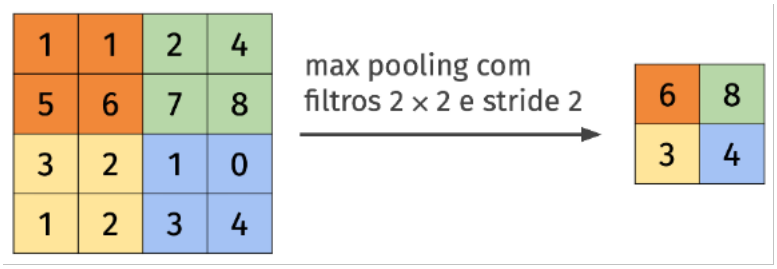
As camadas convolucionais, conforme já vimos, são as camadas nas quais os filtros de convolução esquadrinham a imagem e computam o produto interno entre o filtro e a imagem gerando como resultado um mapa de ativação (GÉRON, 2019). Ao montar uma rede neural utilizando a biblioteca *TensorFlow*, é necessário determinar a quantidade de camadas convolucionais, quantos filtros a rede deve ter, bem como suas dimensões. Estes filtros são elementos fundamentais no aprendizado da rede, já que agora eles irão determinar os parâmetros que a rede deverá aprender.

De acordo com (AVILA, 2020), existem alguns mecanismos que podem ser utilizados para reduzir as dimensões da saída (mapas de ativação), como por exemplo, é

possível determinar o preenchimento (*padding*) deste filtro, ou seja, ao centralizar o filtro na imagem, caso seja necessário passar este filtro por toda a borda, então será necessário inserir pixels nas bordas da imagem e, com isto, manter as dimensões da imagem no mapa de ativação. Da mesma forma, é possível definir se este filtro vai percorrer a imagem passando por toda linha x coluna, ou se isto será feito num passo maior (*stride*) gerando assim uma saída de dimensão menor. Dependendo da situação, manter a resolução da imagem original não seja tão relevante. Então, neste caso, consegue-se um ganho computacional neste processamento. Além disto é possível executar um processo de dilatação da imagem. Neste caso, o filtro é expandido e a imagem a ser esquadrinhada torna-se maior do que num processo sem dilatação.

Conforme (GÉRON, 2019), as camadas de *pooling* (ou camadas de agrupamento) costumam ser agregadas sempre após as camadas de convolução. Estas camadas têm por objetivo condensar a saída das camadas de convolução. Elas não possuem peso. Seu papel principal é selecionar as informações mais relevantes dos mapas de ativação da camada anterior e transmiti-las para a camada seguinte. Esta relevância de informação pode ser definida por vários critérios, como por exemplo, resumir uma região de 2×2 neurônios da camada convolucional gerando apenas a ativação máxima desta pequena região (*Max-Pooling*); ou então, tomar a raiz quadrada da soma dos quadrados das ativações na região 2×2 (*Pooling L2*). Estas abordagens (*Max-Pooling, Pooling L2* etc.) podem ser configuradas no algoritmo e, ao medir o desempenho do modelo, elas podem ser comparadas e ajustadas ao que for mais interessante para a solução do problema. Esta técnica reduz o tamanho da saída, e, portanto, o número de parâmetros a serem computados tornando seu processamento mais barato. Um exemplo desta redução pode ser verificado na Figura 27. Observe que, ao passar um filtro 2×2 com um *stride* $= 2$, o que significa um salto de tamanho 2, é possível reduzir uma matriz de 4×4, para 2×2. Como se trata de uma operação de *max pooling*, o filtro 2×2 está esquadrinhando a matriz de dados 4×4 (ou seja, o mapa de ativação) e pegando o maior valor de cada região, gerando uma matriz de resultados de tamanho 2×2.

Figura 27 – Exemplo de uma saída após uma operação em uma camada de *Pooling*.



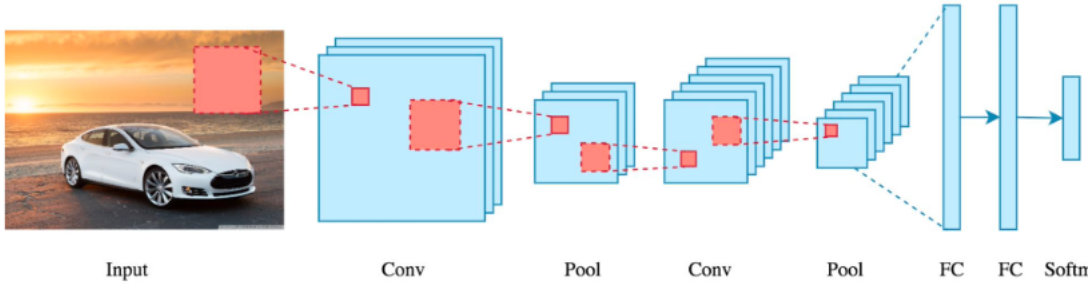
Fonte: Notas de Aula do Curso Mineração de Dados Complexos - IC Unicamp - 2020 (AVILA, 2020).

Por último, a camada final de conexões da rede (*fully connected*) é uma camada

igual a outras arquiteturas, que irá conectar os neurônios da camada de *pooling* com os neurônios de saída de cada uma das classes.

A Figura 28, mostra de forma bem resumida a estrutura básica da arquitetura de uma rede convolucional. Ou seja, temos uma imagem que passa por uma camada de convolução, onde as características desta imagem são codificadas, em seguida temos uma camada de *pooling*, que tem por objetivo agregar os resultados da(s) camada(s) anterior(es). Segue assim até chegar ao final da rede, onde existirão as camadas *fully connected*, que correspondem a vetores de tamanho correspondente ao número de classes que se está tentando aprender, até chegar a uma função de ativação *Softmax* que irá emitir uma única saída para cada classe.

Figura 28 – Resumo da arquitetura de uma rede convolucional.



Fonte: Notas de Aula do Curso Mineração de Dados Complexos - IC Unicamp - 2020 (AVILA, 2020).

1.7 Redes pré-treinadas com transferência de conhecimento (*transfer learning*)

De acordo com (AVILA, 2020), nos últimos anos, a comunidade de pesquisa propôs várias arquiteturas, métodos de otimização e técnicas de regularização da CNN. Além de focar na melhoria da generalização e eficácia em tarefas particulares, alguns trabalhos também visaram aumentar a eficiência dessas redes, permitindo-lhes treinar mais rápido e operar em dispositivos de baixa potência.

As redes pré-treinadas se baseiam no argumento de que um modelo pré-treinado em uma tarefa diferente do problema atual pode fornecer um ponto de partida muito útil para diferentes tarefas, ou seja, os recursos aprendidos durante o treinamento na tarefa antiga são úteis para a nova tarefa.

A grande motivação para a utilização de uma rede pré-treinada surge do fato que as camadas iniciais de uma rede neural são muito difíceis de treinar. Isto se dá por conta da forma como os pesos são otimizados. Ou seja, à medida em que vamos retro propagando o erro calculado através da derivada parcial, é muito provável que, ao

chegarmos às camadas iniciais (que são as últimas neste processo), as atualizações de peso sejam insignificantes, a ponto de comprometer o aprendizado da rede. Por outro lado, considerando a forma como as redes convolucionais funcionam, sabemos que as camadas iniciais destinam-se a representar apenas os elementos mais primitivos, como a identificação de bordas, por exemplo. Já as camadas posteriores, irão capturar os elementos particulares específicos do nosso conjunto de dados. Portanto, considerando estes dois aspectos, podemos concluir que as camadas finais, além de serem mais relevantes por conta das particularidades especificas ao banco de dados em treinamento, são mais fáceis e rápidas de treinar, o que gera um impacto imediato nos resultados finais. Além disto, treinar uma rede neural a partir do zero é uma tarefa complicada que requer tempo e recursos computacionais robustos, dependendo do tamanho e da complexidade do banco de dados.

Portanto, o conceito das redes pré-treinadas se baseia na possibilidade de se treinar apenas as camadas posteriores e, para as camadas iniciais, pode-se utilizar os pesos já aprendidos por estas redes em trabalhos anteriores. Obviamente, quanto mais semelhantes seus dados e problemas forem aos dados de origem da rede pré-treinada, melhor será o resultado final para os dados específicos que se está tentando classificar.

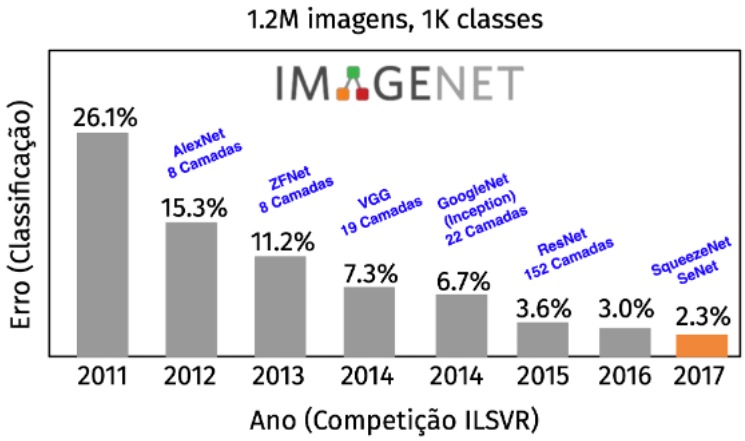
Ainda de acordo com (GÉRON, 2019), tendo em vista este novo conceito, desde 2012, a competição anual '*ImageNet Large Scale Visual Recognition Challenge*' (ILSVRC) promovida pela plataforma *ImageNet* https://image-net.org tem trazido contribuições expressivas em termos de inovação e performance de modelos de redes convolucionais para reconhecimento de imagens. Ainda que hoje em dia a plataforma com milhões de imagens separadas em 1000 classes distintas não represente um volume de dados tão grandioso quanto era em 2012, esta competição virou referência para pesquisadores do mundo inteiro na área de IA.

Algumas arquiteturas vencedoras destas competições trouxeram grandes resultados ao processo de treinamento de redes neurais. Na Figura 29, temos um gráfico com as arquiteturas vencedoras em cada ano, bem como a taxa de erro máxima que elas apresentaram em seus experimentos de suas divulgações. Dentre estas arquiteturas, algumas merecem destaque. Vejamos:

1.7.1 *AlexNet* – 2012

A primeira grande vencedora que merece destaque foi a *AlexNet* em 2012, desenvolvida por *Alex Krizhevsky*. A sua principal contribuição, de acordo com seu artigo original (KRIZHEVSKY; SUTSKEVER; HINTON, 2012), é que sua profundidade de 8 camadas era responsável pelo seu alto desempenho (sua taxa de acerto era de 84,7%), e que isto se tornou viável graças a utilização de recursos computacionais bem mais eficientes (unidades de processamento gráfico – GPU). Além disto, esta rede introduziu a utilização

Figura 29 – Arquiteturas vencedoras do ILSVRC e suas respectivas taxas de erro – de 2012 e 2017.



Fonte: Notas de Aula do Curso Mineração de Dados Complexos - IC Unicamp - 2020 (AVILA, 2020).

da função de ativação *ReLU*, que resolvia alguns problemas de fuga do gradiente quando se utilizava a função sigmoide. Além de adicionar camadas de dropout (conforme veremos na Seção 2.2) e combinar camadas convolucionais com camadas densas no final da arquitetura antes da previsão dos resultados, a *AlexNet* também trouxe a possiblidade de aplicar transformações aleatórias ao banco de imagens, criando assim imagens sintéticas que tinham como objetivo equilibrar a quantidade de classes e aumentar a quantidade de amostras e tornar o treinamento das redes neurais um processo mais robusto.

1.7.2 *VGGNet* – 2014

(SIMONYAN; ZISSERMAN, 2014) Em 2014, foi a vez da *VGGNet* desenvolvida pelo denominado Grupo de Geometria Visual da Universidade de *Oxford* – Inglaterra. O diferencial da *VGGNet* era sua profundidade. Foram apresentadas 6 arquiteturas diferentes, sendo que as duas com melhor desempenho foram a VGG16, com 16 camadas e a VGG19, com 19 camadas de profundidade. A arquitetura básica da VGG era composta por 5 blocos, cada bloco possui de 2 a 3 convoluções consecutivas seguidas por uma camada de *Max-Pooling*, além de 3 camadas densas no final. Todas as camadas convolucionais e *pooling* utilizam o mesmo *padding* (descrito acima). Já o *stride* é igual a 1 para todas as camadas de convoluções e a função de ativação utilizada é a *ReLU*. Uma contribuição relevante da VGGNet em relação as arquiteturas anteriores é que, apesar da VGG apresentar mais camadas de convolução, ela se utiliza de filtros menores (3×3, em sua maioria), o que diminui o número de parâmetros a serem treinados, e por outro lado o número maior de camadas de convoluções acabam aumentando a capacidade de aprender recursos complexos (ou seja, combinando mais operações não lineares).

1.7.3 *ResNet* - 2015

(HE XIANGYU ZHANG, 2015) Essencialmente, até 2014 sabia-se empiricamente que o aprendizado de uma rede neural se dava pela adição de camadas ocultas à sua arquitetura, uma vez que isto criava o poder de solução para problemas não lineares. Entretanto, esta técnica tinha um limite máximo de 30 camadas. Após este limite, começava a surgir um problema conhecido como explosão ou desaparecimento do gradiente, o que limitava a capacidade de aprendizado da rede(GÉRON, 2019).

Conforme visto anteriormente, o treinamento de uma rede neural profunda se dá pela retro-propagação do erro calculado através das derivadas parciais da função de custo. Desta forma, em uma rede de n camadas ocultas, terão milhares de derivadas sendo multiplicadas juntas. Então, para derivadas com valores suficientemente grandes, o gradiente irá aumentar exponencialmente à medida que retro propagamos o erro até chegar um momento em que o valor do gradiente explode. Por outro lado, caso as derivadas sejam muito pequenas, o gradiente então irá diminuir exponencialmente até chegar a zero (desaparecimento do gradiente). Além disto, um outro fator relevante começava a ser observado: a acurácia das redes convolucionais não aumentam na mesma proporção em que se aumentam as camadas em sua arquitetura, em alguns casos, a precisão pode até degradar.

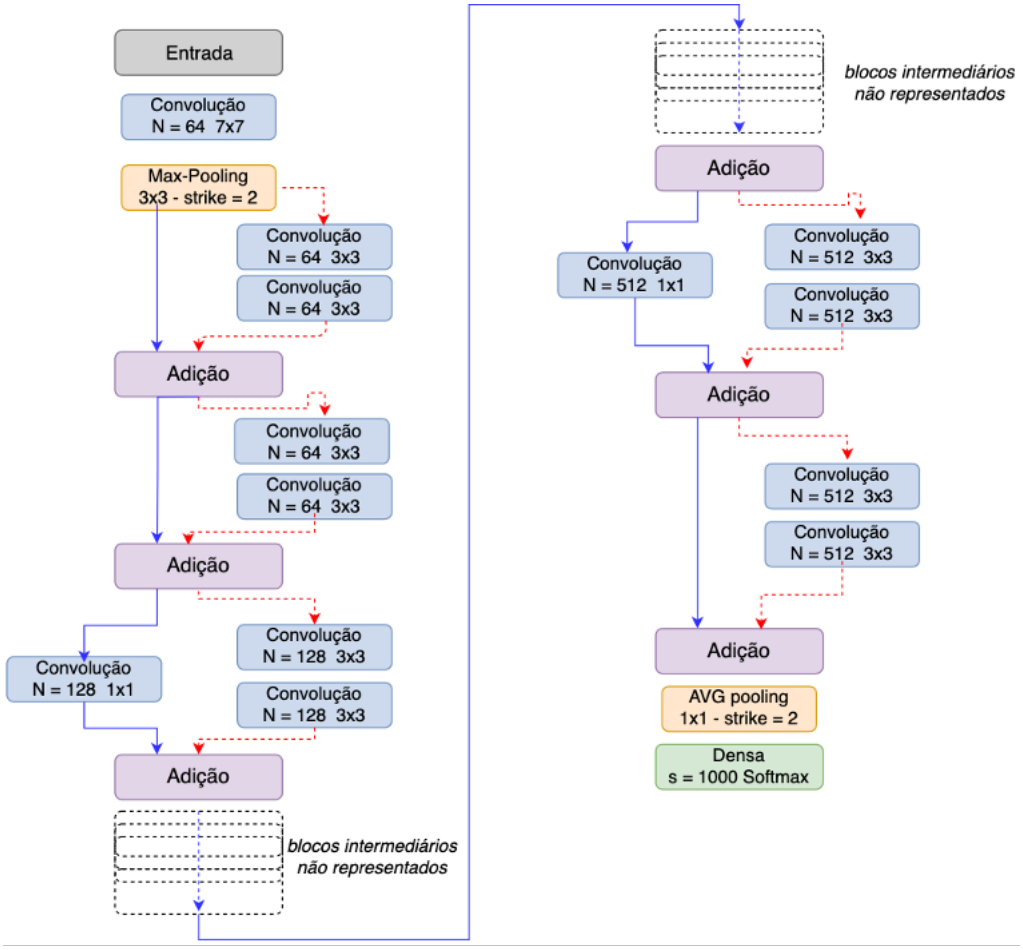
Entretanto, em 2015 surgiu uma nova solução com uma abordagem muito eficiente para a criação de redes profundas trazendo um desempenho muito superior em relação aos modelos anteriores – a arquitetura *ResNet* (*Residual Network*).

Assim como a *VGGNet*, a *ResNet* possui diferentes versões com diferentes quantidades de camadas (*ResNet18, ResNet50, ResNet101, ResNet152*) e sua grande contribuição foi demonstrar ser possível empilhar até 152 camadas implementando um mapa residual criado a partir dos resultados de uma série de camadas de convolução.

Observando a topologia da rede na Figura 30 é possível visualizar uma arquitetura composta de blocos de camadas com operações paralelas, sendo que de um lado segue um caminho não linear (representado por setas pontilhadas) onde são aplicadas algumas convoluções com normalização de lote (*batch*) e ativação *ReLU* para os mapas residuais de entrada. Já o outro lado (representado por setas azuis contínuas) segue um caminho de identidade que simplesmente encaminha os resultados da camada anterior sem qualquer alteração.

Perceba que as camadas nomeadas como Adição recebem o valor x da camada anterior, e, recebe um valor $f(x)$ processado pelos blocos residuais. Com isto, caso o valor de $f(x)$ seja zero, pelo menos, o que foi aprendido anteriormente será transmitido adiante. É desta forma, que blocos residuais, ao invés de camadas de redes simples, não comprometem o desempenho geral da rede. Evidentemente, a entrada e saída de cada

Figura 30 – Topologia da rede ResNet.



Fonte: imagem adaptada – extraída do livro *Hands-On Computer Vision with TensorFlow 2 – Section 2: State-of-the-Art Solutions for Classic Recognition Problems – Influential Classification Tools.*

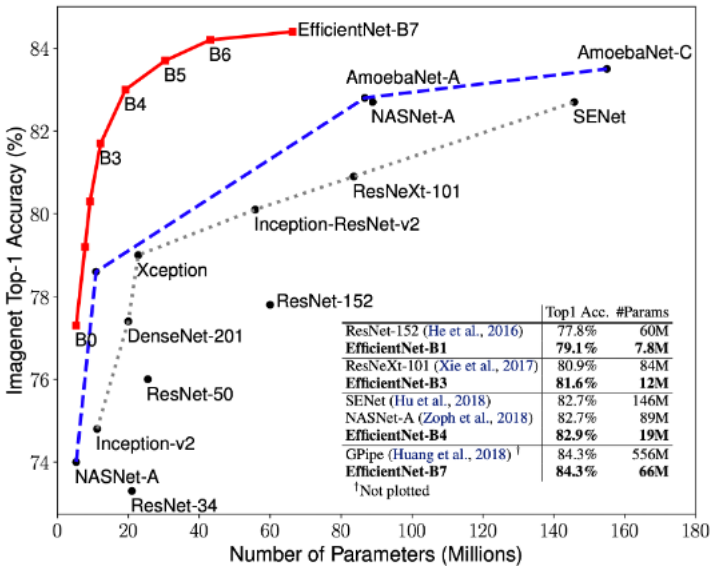
etapa da rede precisam ter dimensões compatíveis. Daí a existência de convoluções 1×1. Ou seja, se existe um mapa de características de tamanho $64 \times 64 \times 128$, mas a saída é de $64 \times 64 \times 256$, então será necessário um bloco de convolução de $1 \times 1 \times 256$ para que a saída seja de tamanho de $64 \times 64 \times 256$. Em arquiteturas mais profundas é utilizado o que se chama de *estrutura de gargalo*, ou seja, primeiramente aplica-se uma convolução 1×1 com objetivo de reduzir a dimensionalidade, em seguida, aplica-se convoluções 3×3 e, finalmente, utiliza-se a dimensionalidade 1×1 para restaurar a dimensão original. Isto equivale a diminuir o número de parâmetros em arquiteturas mais profundas.

Conforme (AVILA, 2020), a grande contribuição da *ResNet* em resolver os problemas de degradação do gradiente no processo de otimização, com taxas de erros bem pequenas e baixo custo computacional fez desta arquitetura uma das ferramentas mais eficazes atualmente em termos de processamento de redes neurais.

1.7.4 *EfficientNet* – 2019

Lançada pela Google em 2019 a família *EfficientNet* veio com uma proposta de equilíbrio entre profundidade, largura e resolução da rede, o que pode levar a um melhor desempenho. Ao contrário de outras redes onde se leva em conta principalmente a profundidade das redes, a proposta desta arquitetura é dimensionar uniformemente estes 3 fatores (profundidade, largura e resolução) utilizando coeficientes de escala fixos.

Figura 31 – Desempenho da EfficientNet.



Fonte: Imagem extraída de (TAN, 2019).

Desde 2012 as redes convolucionais tem se tornado cada vez mais precisas, à medida que se tornam cada vez maiores. Enquanto a *GoogleNet*, vencedora de 2014, utilizava em torno de 6.8 milhões de parâmetros, a *ResNet*, vencedora de 2015, na sua versão 50 utilizava em torno 23.6 milhões de parâmetros. Já a rede *SeNet*, vencedora de 2017 conseguiu atingir a impressionante taxa de erro de 2.3% utilizando 557 milhões de parâmetros. A *EfficientNet*, na sua versão mais robusta (B7), utiliza 66.6 milhões de parâmetros e seu desempenho é apresentado na Figura 31 conforme artigo de seu inventor (TAN, 2019).

2 Metodologia

Uma vez discutido o embasamento matemático em que se sustentam as redes neurais e tendo conhecido as soluções inovadoras propostas pelas redes neurais pré-treinadas, o próximo passo será avaliar a eficiência das redes neurais através de uma solução de aprendizado de máquina utilizando algoritmos em linguagem computacional *Python*.

O problema a ser resolvido é o de identificar o comportamento de motoristas ao volante através de fotos tiradas do painel de seus carros. Para tal desafio utilizaremos uma base de dados provida pela plataforma *Kaggle* https://www.kaggle.com, que é uma plataforma de modelagem preditiva e de competições analíticas.

De acordo com o Conselho Federal de Medicina (CFM), um balanço feito entre 2009 e 2018 aponta que os acidentes de trânsito deixaram mais de 1,6 milhão de feridos no Brasil, implicando um custo de quase 3 bilhões de reais ao Sistema Único de Saúde (SUS). Parte desses acidentes acontecem por imprudência e distrações dos motoristas, como responder mensagens ou falar ao celular.

Nos EUA, a companhia de seguros *State Farm*, também preocupada com as estatísticas alarmantes de que 1 em cada 5 acidentes de carro é provocado por um motorista distraído lançou um desafio na plataforma *Kaggle* para testar se câmeras do painel de um carro podem detectar automaticamente os motoristas que se envolvem em comportamentos distraídos. Para isto, a empresa forneceu um conjunto de imagens composto por 102.150 imagens capturadas por câmeras instaladas no interior de veículos. Deste total 22.424 imagens são destinadas para treino e validação do modelo e 79.726 imagens deverão ser utilizadas para testes. Estas fotos mostravam os motoristas em diversas atividades tais como: dirigindo, falando ao celular, ajeitando o visual, ou enviando mensagens de texto. Cada uma destas imagens era rotulada com sua classe verdadeira e o objetivo principal do desafio era classificar qual a atividade desenvolvida pelo motorista, dentro de um conjunto de dez possíveis atividades, conforme definidas na Figura 32.

Diante do exposto, nossa proposta é resolver um problema típico de aprendizado de máquina supervisionado, onde aplicaremos técnicas de processamento de imagens com redes convolucionais e classificação multi classes. Nosso objetivo será definir, treinar e comparar diferentes métodos de classificação, bem como mostrar como determinados ajustes nos algoritmos influenciam fortemente nos resultados.

O primeiro passo é a definição dos recursos a serem utilizados neste processo de treinamento dos nossos modelos de classificação.

Devido ao grande volume de dados a serem processados, optamos por utilizar

Figura 32 – Classes de comportamento dos motoristas.

(a) dirigindo em segurança

(b) digitando mão direita

(c) falando ao celular – direita

(d) digitando – mão esquerda

(e) falando ao celular – esquerda

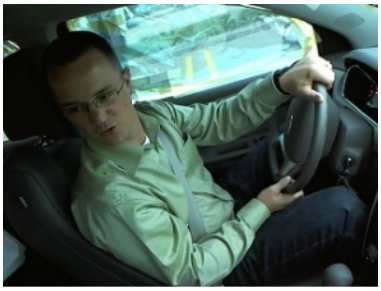
(f) operando dispositivo painel

(g) bebendo

(h) mexendo no banco de trás

(i) ajeitando o visual

(j) conversando com passageiro



Fonte: Imagens obtidas no banco de dados da plataforma *Kaggle*.

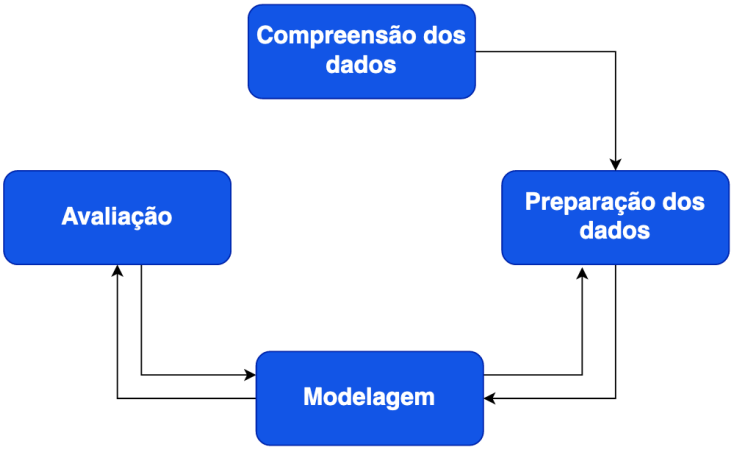
a ferramenta *Google Colab* (https://colab.research.google.com), que é uma ferramenta colaborativa que permite desenvolver código em *Python* em seu navegador sem a necessidade de instalação ou configuração de qualquer aplicativo. Para acessá-la, basta criar uma conta Google. Além disto, o *Google Colab* disponibiliza acesso limitado e gratuito a GPU.

Através da utilização da ferramenta *Python*, a proposta é combinar o uso

da biblioteca *TensorFlow Keras*, que é um *framework* de código aberto adequado para implementação de solução de aprendizado profundo, e que deverá executar automaticamente a otimização do modelo baseado na técnica do gradiente descendente. Além disto, este *framework* também possibilita a utilização das arquiteturas pré-treinadas escolhidas para nossos modelos: *VGG16, Resnet50* e família *EfficientNet*.

Com base em *workflows* utilizados em processos de *machine learning* descritos na literatura (CHOLLET, 2019), em nossos processo de avaliação dos modelos seguiremos as etapas descritas na Figura 33.

Figura 33 – Metodologia em Ciência de Dados.



Fonte: A Autora (2022).

Na primeira etapa, que é a compreensão dos dados, iremos avaliar quais os dados disponíveis, a qualidade das imagens, a necessidade de redimensionamento, bem como o balanceamento entre as classes. O balanceamento entre as classes é muito importante para garantir que o algoritmo seja capaz de produzir regras de classificação igualitária entre as classes, evitando assim, uma menor taxa de reconhecimento para a classe com número de amostras em menor quantidade. Em termos éticos, este é um preceito básico da IA para se evitar quaisquer resultados tendenciosos ou discriminatórios (GÉRON, 2019). Lembrando que as redes neurais são modelos probabilísticos, logo, é necessário garantir uma representação equilibrada de todas as possibilidades para garantir uma igualdade de chances para todas as classes.

Em seguida, a fase de preparação dos dados, será executada utilizando a função *ImageDataGenerator* da biblioteca *TensorFow Keras*(CHOLLET, 2019). O objetivo desta função é pré-processar as imagens baixadas para que elas possam caber no formato de entrada da rede. Além disto, esta função permite reajustar a escala das imagens, além de gerar dois conjuntos distintos de imagens que serão utilizados para treino e validação. A plataforma *Kaggle* fornece um total de 22.424 imagens que devem ser utilizadas para treino e validação. Estas imagens possuem rótulo identificando a classe a que cada imagem

pertence. Vamos dividir este conjunto de imagens numa proporção de 20 a 25% para treino e 75 a 80% para validação. Já as 79.726 imagens destinadas a teste são disponibilizadas em um conjunto separado. Estas imagens não possuem rótulos e não devem ser submetidas a nenhum pré processamento.

Na fase de modelagem, além de uma rede neural convolucional treinada a partir do zero, foram exploradas 3 arquiteturas pré-treinadas propostas nas competições promovidas pela plataforma *ImageNet*: *VGG*, *ResNet50* e família *EfficientNet*. Para todas as arquiteturas pré-treinadas utilizamos a otimização de pesos do *ImageNet*, uma vez que desejamos explorar a capacidade das redes neurais em aprender conceitos equivalentes em suas camadas inicial e intermediárias. Ou seja, vamos absorver o aprendizado adquirido nas competições do *ImageNet* e fazermos algumas adaptações *(fine tunning)* nas arquiteturas para que possamos adequá-las ao nosso objetivo de classificar o comportamento de motoristas ao volante. Tais adaptações consistem em removermos a última camada totalmente conectada, responsável por classificar em uma das 1000 classes do *ImageNet*, e adicionarmos uma nova camada totalmente conectada com 10 neurônios de saída que deverão representar cada uma das classes do nosso problema. Usaremos ativação Softmax para esta camada de saída. Por último, vamos testar o congelamento e descongelamento de camadas pré-treinadas, o que significa manter ou atualizar os pesos iniciais das camadas pré-treinadas durante o processo de treinamento do modelo. Além disto, adotamos algumas práticas para evitar que nossos modelos atinjam o *overfit*. A definição de *underfit* e *overfit*, bem como as práticas utilizadas nos nossos modelos estão descritas na Seção 2.2. Na Tabela 1 apresentamos um resumo dos 14 modelos, dos quais pretendemos discutir os resultados.

Por último, avaliação dos nossos modelos será feita por duas métricas diferentes em duas etapas distintas do processo. Na fase de treino e validação utilizaremos a acurácia balanceada, que é calculada com resultados providos pela matriz de confusão, que será discutida em 2.1.1. Inicialmente, esta métrica será fundamental para indicar problemas de *underfit*. Quanto maior a acurácia balanceada, melhor será nosso modelo. Já na fase de testes, seguindo os requisitos propostos pela plataforma *Kaggle*, para medirmos a capacidade de generalização dos modelos utilizaremos a perda logarítmica (*logloss*). Neste caso, quanto menor esta medida, melhor será a capacidade de generalização do nosso modelo, uma vez que a *logloss* aumenta à medida que a probabilidade prevista diverge do valor de y real. Na Seção 2.1 explicamos em detalhes o processo de cálculo de cada uma destas métricas.

Mod	Arquitetura	Camadas Congeladas	Resolução Imagem	Otimizador	Taxa de Aprendizado	Dados Sintéticos
01	CNN do zero com: 3 camadas de convolução, ativação ReLU, 2 camadas max$_p$*ooling*, 2 camadas dropout.	N/A	299 × 299	Adam	0.001	N/A
02	*VGG16* *somente com as 13* *camadas convolucionais*	Não	299 × 299	SGD+momentum	0. 01	Cisalhamento
03	*VGG16* *com 13 camadas* *convolucionais + adição* *3 camadas densas no final da rede*	Não	299 × 299	SGD+Decay	0.001	Cisalhamento
04	*Resnet50*	Sim	224 × 224	SGD	0.0001	N/A
05	*ResNet50*	Sim	224 × 224	SGD	0.01	N/A
06	*ResNet50*	Sim	224 × 224	SGD+Decay	0.01	N/A
07	*ResNet50*	Sim	224 × 224	SGD+momentum	0.01	N/A
08	*ResNet50*	Sim	224 × 224	SGD+momentum	0.01	cisalh+ zoom
09	*ResNet50*	Sim	224 × 224	SGD+momentum	0.01	cisalhamento
10	*ResNet50*	Não	224 × 224	SGD+momentum	0.01	cisalhamento
11	*EfficientNetB0*	Não	224 × 224	SGD+momentum	0.01	cisalhamento
12	*EfficientNetB1*	Não	240 × 240	SGD+momentum	0.01	cisalhamento
13	*EfficientNetB2*	Não	260 × 260	SGD+momentum	0.01	cisalhamento
14	*EfficientNetB3*	Não	300 × 300	SGD+momentum	0.01	cisalhamento

Tabela 1 – Modelos propostos para discussão de resultados.

2.1 Métricas de Desempenho

Conforme discutido anteriormente, optamos pelas métricas de acurácia balanceada e perda logarítmica para medir e comparar o desempenho dos nossos modelos.

2.1.1 Acurácia balanceada

Tanto a acurácia, quanto a acurácia balanceada são métricas calculadas a partir dos resultados gerados na matriz de confusão (GÉRON, 2019).

Muito útil para análise de desempenho de modelos de classificação, a matriz de confusão é uma matriz quadrada que apresenta o número de linhas e colunas equivalentes ao número de classes do problema que está sendo analisado. A quantidade de acertos de cada classe é exibida na diagonal da matriz. Sua estrutura básica é constituída conforme Tabela 2, e seus valores são definidos como:

- **Verdadeiros positivos (VP)**: valores positivos que o sistema julgou positivos (acerto).
- **Falsos negativos (FN)**: valores positivos que o sistema julgou negativos (erro).
- **Verdadeiros negativos (VN)**: valores negativos que o sistema julgou como negativos (acerto).
- **Falsos positivos (FP)**: valores negativos que o sistema julgou positivos (erro).

		Classe Real	
		Positiva	**Negativa**
Classe Predita	**Positiva**	Verdadeiro Positivo (VP)	Falso Positivo (FP)
	Negativa	Falso Negativo (FN)	Verdadeiro Negativo (VN)

Tabela 2 – Matriz de Confusão.

A acurácia é a razão entre o total de acertos e o total de amostras do conjunto. Esta medida é altamente suscetível a desbalanceamento do conjunto de dados e pode facilmente induzir a erros de interpretação sobre o desempenho do modelo. Isto acontece, pois, conforme é possivel visualizar em sua fórmula, os valores de classificados VN (verdadeiro negativo) podem mascarar as classificações baixas de VP (verdadeiro positivo) gerando a falsa impressão de que o modelo está fazendo classificações corretas. Sua expressão matemática é dada por:

$$Acc = \frac{VP + VN}{VP + FN + FP + VN}. \tag{2.1}$$

Para evitar tais equívocos, e alcançar um valor mais apurado em relação aos acertos do modelo em relação às classes, optamos por utilizar a acurácia balanceada, uma vez que esta métrica não é afetada pelo desbalanceamento das classes, já que os cálculos são baseados em taxas de VP (verdadeiros positivos) e VN (verdadeiros negativos), conforme podemos visualizar na Fórmula 2.2:

$$AccB = \frac{1}{2}\Big(\frac{VP}{VP + FN} + \frac{VN}{VN + FP}\Big). \tag{2.2}$$

2.1.2 Perda Logarítmica

A perda logarítmica (logloss) é definida pela Fórmula 2.3:

$$logloss = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij} log(\hat{y}_{ij}), \tag{2.3}$$

onde:

- N é o número de imagens no conjunto de teste;
- M é o número de rótulos de classe de imagem;
- $y_{ij} = 1$ se a observação i pertence à classe j. Caso contrário, $y_{ij} = 0$;
- $\hat{y}_{ij}$ é a probabilidade prevista de que a observação i pertence à classe j.

Em última análise, a avaliação da acurácia balanceada e perda logarítmica irá indicar se nossos modelos tem capacidade de continuar melhorando, ou se ele já começa a indicar *overfit*.

2.2 Técnicas para lidar com *Underfit* ou *Overfit*

O grande desafio do aprendizado de máquina é treinar um algoritmo de forma que ele consiga generalizar o problema, ou seja, que ele seja capaz de fazer predições ou associações com qualquer base de dados, e não somente com as utilizadas no seu processo de treinamento (CHOLLET, 2019). Normalmente, num processo de treinamento é necessário lidar com dois tipos de problemas:

- *Underfit*: é quando o modelo é muito simples para aprender a complexidade dos dados. Neste caso, as taxas de aprendizado são muito baixas, ou então, dentro das probabilidades genéricas da matemática (exemplo: se temos como opção cara ou coroa, então a probabilidade de obter cada uma é de 50%).
- *Overfit*: ao contrário do *underfit*, neste caso o algoritmo se ajusta perfeitamente ao conjunto de dados alcançando uma taxa muito alta de acertos. Porém, ao submetê-lo a problemas reais, sua performance cai consideravelmente. Neste caso, pode-se afirmar que o modelo aprendeu um problema específico, mas não consegue generalizar para qualquer outro caso.

Existem diversas técnicas para lidar com cada um destes problemas. No caso dos nossos modelos que apresentaram *underfit*, a estratégia adotada foi tentar outros modelos utilizando arquiteturas pré-treinadas tais como: *VGG16, ResNet50, EfficientNetB0, EfficientNetB1, EfficientNetB2 e EfficientNetB3*.

Já para evitarmos os casos de *overfit*, adotamos as seguintes técnicas de regularização bastante recomendadas para generalização do modelo:

1. **Decaimento do peso *(Weight Decay)*** : também conhecida como regularização L2, esta técnica permite adicionar um termo regularizador na função de custo a ser minimizada, ou seja, dada a função de custo (ou entropia cruzada):

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y_i \cdot log(h_\theta(x_i)) + (1-y_i)log(1-h_\theta(x_i))], \tag{2.4}$$

   podemos acrescentar o termo regularizador:

$$+\frac{\lambda}{2m}\sum w^2, \tag{2.5}$$

   onde $\lambda > 0$ é conhecido como parâmetro de regularização e m é a quantidade de amostras do conjunto de treinamento. Já $\sum \mathrm{w}^2$, representa a somatória de todos os pesos da rede.

De forma muito prática, este tipo de regularização acaba fazendo com que a rede aprenda pesos pequenos. Pesos grandes são admitidos apenas se isto conseguir melhorar significativamente a primeira parte da função de custo. Agora, esta relação entre a função de custo e o termo de regularização é determinada por λ, ou seja, quando λ é pequeno, preferimos minimizar a função de custo original, mas quando λ é grande, preferimos pesos pequenos.

2. **Parada Antecipada *(Early Stopping)***: o principal objetivo de treinamento de uma rede neural é alcançar um bom grau de generalização, ou seja, ao ser submetida aos dados de testes, a rede precisa manter um desempenho bem próximo aos resultados de treino/validação. A definição da quantidade de épocas no treinamento de um modelo de redes neurais nem sempre é uma tarefa fácil, já que poucas épocas podem impedir que o modelo aprenda tudo o que for possível com os dados de treinamento (underfit), e muitas épocas podem fazer com que o modelo se ajuste demais aos dados de treinamento e tenha queda de desempenho com dados de teste (overfit). A técnica de *early stopping* é utilizada justamente para interromper o processo de treinamento quando o erro de validação começa a ficar maior que o erro de treinamento (GÉRON, 2019). Isto ajuda a evitar que o modelo "aprenda demais", ou não aprenda o suficiente.

3. ***Dropout***: esta técnica de regularização permite 'apagar' alguns neurônios dentro da rede, e com isto são apagadas também as conexões relacionadas aos mesmos. Este processo evita que a rede memorize caminhos realizados no treinamento (GÉRON, 2019). Se, por exemplo, em uma época de treinamento existem determinadas conexões, e estas são apagadas para uma época seguinte, significa que houve uma pequena modificação da arquitetura e, portanto, o modelo vai evitar o aprendizado deste caminho que pode ou não existir. Com isto, esta técnica auxilia na minimização de riscos de overfit. Apesar desta técnica ter sido recentemente patenteada, ela é oferecida para diversas arquiteturas pré-treinadas na biblioteca Keras.

4. **Expansão de dados *(Data Augmentation)***: segundo (CHOLLET, 2019), esta técnica permite aplicar transformações (rotações, cortes, alterações de escala, tonalidade de cores, mistura aleatória de imagens) aos dados existentes de forma a expandir o conjunto de dados e fornecer mais exemplos para treinamento.

5. **Inicialização dos pesos**: em casos de redes pré-treinadas, elas já possuem seus pesos de inicialização, uma vez que estas redes já passaram por um processo de aprendizado. Nos nossos experimentos com redes pré-treinadas utilizaremos os pesos da *ImageNet*. Porém, ao construir uma rede neural desde seu início, é necessário definir critérios de inicialização de pesos. Caso não seja definido, na utilização da biblioteca *Keras*, a inicialização pré-definida é a inicialização *Xavier (glorot_uniform)*.

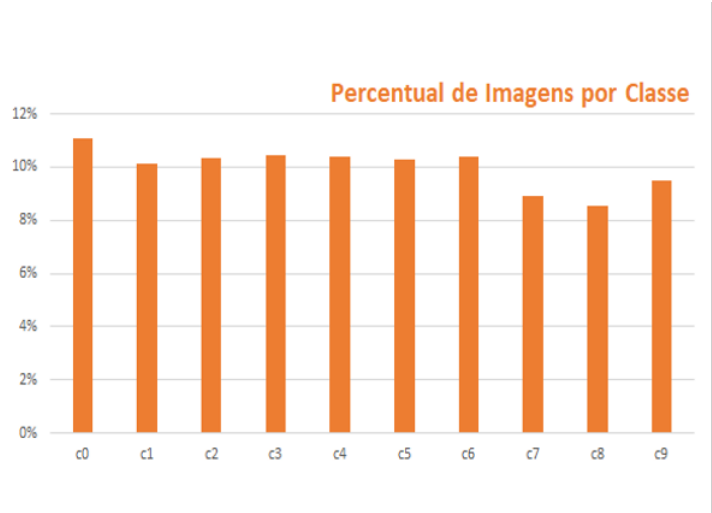
3 Análise dos Dados

3.1 Compreensão dos dados

Iniciamos nosso processo fazendo uma análise do banco de dados para entendermos o balanceamento entre as classes.

O gráfico da Figura 34 demonstra um certo equilíbrio na quantidade de imagens por classes dentro do conjunto de dados. Ainda assim, é recomendável garantir um equilíbrio adicionando um determinado peso a cada classe. Isto eliminará qualquer relevância de uma classe em detrimento de outra.

Figura 34 – Gráfico da distribuição de imagens pelas 10 Classes.



Fonte: (BRAGA et al., 2020).

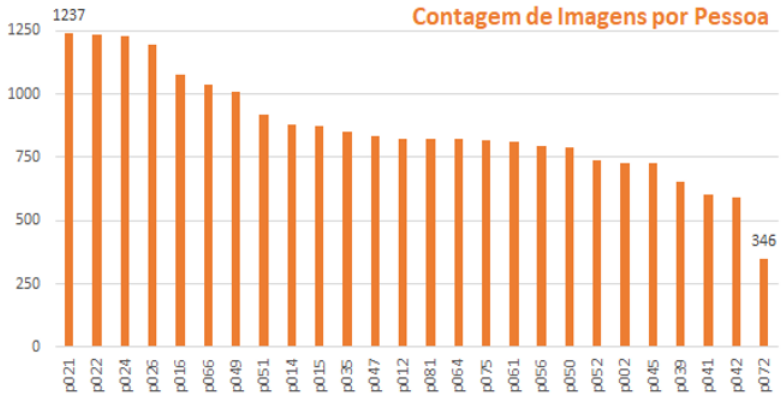
Um aspecto relevante deste conjunto de dados é que estas fotos são apenas simulações geradas em um ambiente controlado onde um caminhão estava arrastando os carros pelas ruas, e que, portanto, os motoristas não estavam realmente dirigindo. Nestas circunstâncias, um único motorista pode estar representando mais de uma classe de comportamento dentro do nosso conjunto de dados.

No gráfico da Figura 35 é possível analisar o número de fotos capturadas de cada participante. Podemos verificar que existe uma variação alta entre o participante com mais de 1.200 fotos e outro com 340 fotos.

Uma outra avaliação sobre a base de dados de imagens para treino, foi verificar a distribuição de pessoas em relação às classes e como cada classe era representada pelas múltiplas pessoas, conforme pode ser visualizado no gráfico da Figura 36

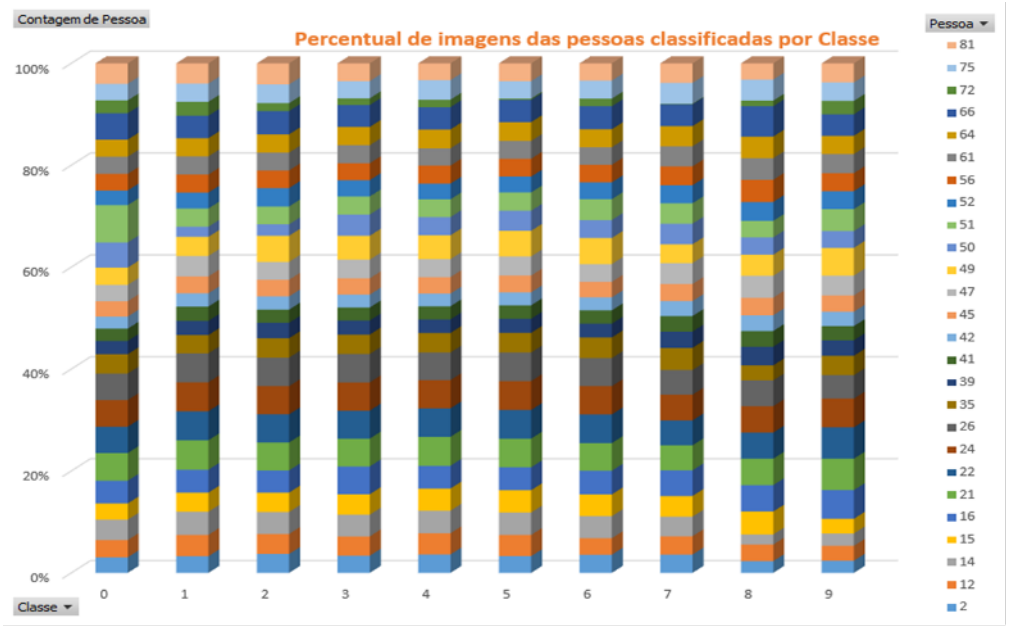
Após compreender as particularidades do conjunto de dados, fizemos o *down-*

Figura 35 – Gráfico da Contagem de imagens por pessoa.



Fonte: (BRAGA et al., 2020).

Figura 36 – Gráfico representando a distribuição das pessoas por cada uma das classes.



Fonte: (BRAGA et al., 2020).

load das imagens da plataforma Kaggle para nosso ambiente *Google Colab*. Em seguida, executamos um pré processamento inicial para definirmos um modelo base para nossa tarefa de classificação.

3.2 Modelo 1 - Arquitetura treinada a partir do zero

Em nosso primeiro modelo, testamos uma rede neural convolucional treinada a partir do zero. Pré processamos as imagens e definimos como dimensão de entrada na rede o tamanho de 128×128. Separamos os dados de treino e validação na proporção de 25%

Classe: Peso	Classe: Peso
classe 0: 1.0000000000000000;	classe 1: 1.0975896531452087;
classe 2: 1.0742232451093210;	classe 3: 1.0607954545454545;
classe 4: 1.0699140401146132;	classe 5: 1.0767012687427913;
classe 6: 1.0705275229357798;	classe 7: 1.2430093209054593;
classe 8: 1.3019525801952580;	classe 9: 1.1690670006261740;

Tabela 3 – Atribuição de pesos às classes.

e 75% respectivamente, ou seja, designamos 5602 imagens para treino e 16822 imagens para validação. Apesar de não identificarmos um desbalanceamento expressivo entre as classes na etapa de compreensão dos dados, adotamos uma fórmula para inserir pesos compensatórios e garantir um perfeito balanceamento entre as classes. A fórmula adotada fazia a contagem de imagens por classe. Selecionamos o maior total dentre as classes e calculamos a razão entre este valor máximo e a quantidade total de cada classe. Assim sendo, o peso de cada classe ficou conforme descrito na Tabela 3 e o mesmo foi adotado para todos os outros 13 modelos subsequentes.

Em seguida, definimos a nossa arquitetura com camada convolucional com 50 filtros de tamanho 9×9 e ativação $ReLU$, uma outra camada com 100 filtros de tamanho 5×5 e ativação $ReLU$, e mais uma camada com 120 filtros de tamanho 3×3 e ativação $ReLU$. Também utilizamos camadas de *maxpooling* e *dropout*, o que gerou um total de 1.186.230 parâmetros, conforme o mapa da arquitetura descrito na Figura 37. Para o treinamento do modelo escolhemos um otimizador $ADAM$ e uma taxa de aprendizado $\alpha = 0.001$. Após rodarmos 10 épocas, chegamos aos resultados conforme matriz de confusão expressa na Tabela 4. Também podemos observar os mesmos resultados no mapa de calor da Figura 38.

Tendo definida a arquitetura, passamos para o treinamento do modelo. Escolhemos um otimizador $ADAM$ e uma taxa de aprendizado $\alpha = 0.001$. Após rodarmos 10 épocas, chegamos aos resultados conforme matriz de confusão expressa na Tabela 4. Também podemos observar os mesmos resultados no mapa de calor da Figura 38.

Conforme os resultados mostrados na Tabela 4, a acurácia balanceada atingida por este modelo nos mostrou que o mesmo não era capaz de capturar a complexidade do conjunto de imagens oferecido, uma vez que nossos objetos possuem uma grande variação em aparência, formas e cores. No mapa de calor da Figura 38, é possível verificar que o maior índice de desempenho não chegou a 13%. Assim, tecnicamente podemos afirmar que nosso modelo está em *underfit*. Considerando que para um treinamento a partir do zero precisaríamos de um volume gigantesco de dados e, portanto, recursos bem mais robustos do que o que temos disponível, o próximo passo seria testar uma arquitetura pré-treinada.

Além disto, outros testes preliminares nos mostraram que o fato das fotos serem

Figura 37 – Descritivo da arquitetura do Modelo 1.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 120, 120, 50)      12200
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 116, 116, 100)     125100
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 58, 58, 100)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 56, 56, 120)       108120
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 28, 28, 120)       0
_________________________________________________________________
dropout (Dropout)            (None, 28, 28, 120)       0
_________________________________________________________________
flatten (Flatten)            (None, 94080)             0
_________________________________________________________________
dropout_1 (Dropout)          (None, 94080)             0
_________________________________________________________________
dense (Dense)                (None, 10)                940810
=================================================================
Total params: 1,186,230
Trainable params: 1,186,230
Non-trainable params: 0
_________________________________________________________________
```

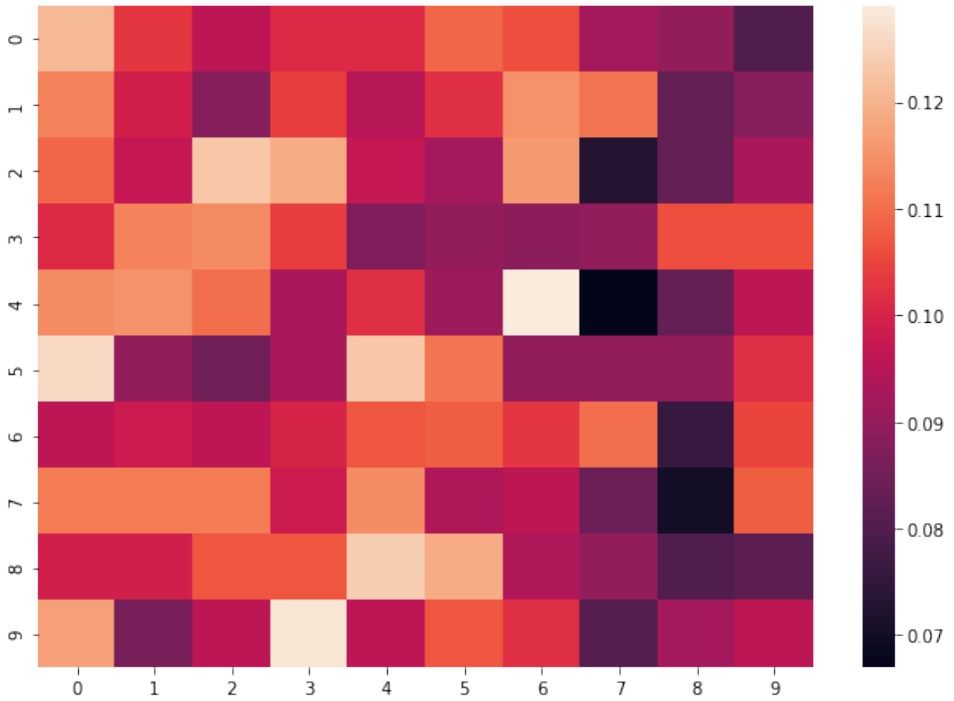
Fonte: A Autora (2022).

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.121	0.103	0.096	0.101	0.101	0.109	0.106	0.092	0.09	0.08
c1	0.113	0.099	0.088	0.104	0.095	0.102	0.115	0.111	0.083	0.088
c2	0.109	0.097	0.123	0.119	0.097	0.092	0.116	0.073	0.083	0.093
c3	0.101	0.113	0.114	0.104	0.087	0.09	0.089	0.09	0.106	0.106
c4	0.114	0.115	0.11	0.093	0.102	0.091	0.129	0.067	0.083	0.096
c5	0.126	0.09	0.085	0.093	0.123	0.111	0.09	0.09	0.09	0.102
c6	0.096	0.098	0.096	0.1	0.107	0.108	0.103	0.11	0.076	0.105
c7	0.112	0.112	0.112	0.098	0.114	0.094	0.096	0.084	0.07	0.108
c8	0.099	0.099	0.107	0.107	0.124	0.119	0.094	0.09	0.08	0.082
c9	0.117	0.086	0.096	0.128	0.096	0.107	0.102	0.081	0.092	0.096
	Acurácia Balanceada de Validação: 0.102									

Tabela 4 – Matriz de confusão dos dados de Validação - Modelo 1.

produzidas por atores, e que um mesmo ator podia representar comportamentos (classes) distintos, estava influenciando nossos resultados. Em alguns destes modelos preliminares chegamos a ter um resultado de acurácia balanceada de 98% e uma perda logarítmica da ordem de 30 (quando esperávamos um resultado menor que 1), o que era um indício claro de *overfit*, ou seja, um desempenho excelente na validação, só que uma baixa capacidade de generalização. A conclusão aqui foi que, possivelmente, os modelos estavam também

Figura 38 – Mapa de calor do Modelo 1.



Fonte: A Autora (2022).

fazendo reconhecimento facial, ao invés de reconhecer somente o comportamento dos motoristas. Isto explicaria o excelente desempenho no ambiente de treino e validação. Entretanto, quando exposto ao ambiente real (testes) os rostos eram diferentes e o aprendizado não era relevante. A solução encontrada para contornar este problema foi incluir mais um critério de separação de dados para garantir que motoristas do grupo de treino, não estivessem no conjunto de validação e vice-versa. Portanto, nos próximos modelos apresentados aqui, todos eles contemplam este critério de separação de dados.

3.3 Modelo 2 - Arquitetura pré-treinada *VGG*

Para este novo modelo decidimos testar a arquitetura pré-treinada *VGG*. Seguindo o fluxo descrito na Figura 33 retornamos à fase de preparação dos dados e fizemos algumas alterações no pré processamento das imagens, tais como: resolução de entrada de imagem de 299×299; separação dos dados de treino e validação na proporção de 30% e 70%, respectivamente, ou seja, designamos 4.646 imagens para treino e 17.778 imagens para validação. Além disto, incluímos uma transformação de dados por cisalhamento.

Tendo concluído o pré processamento das imagens, optamos por utilizar a arquitetura *VGG*, que possui 13 camadas convolucionais, conforme descrita na Figura 40. Além destas 13 camadas, a VGG possui as camadas não treináveis que são 5 camadas de *Max Pooling* e 2 camadas de *dropout*.

Além destas camadas básicas da arquitetura *VGG*, também é possível adicionar mais 3 camadas densas. No entanto, para o Modelo 2 não adicionamos as camadas densas, totalizando assim 14.719.818 parâmetros.

Para o nosso modelo, num primeiro momento optamos por utilizar um otimizador SGD, o que gerou o efeito de perda do gradiente, conforme visualizado na Figura 39.

Figura 39 – Resultado de perda do gradiente.

```
Epoch 1/10
556/556 [==========] - 426s 723ms/step - loss: nan - accuracy: 0.1096 -
val_loss: nan - val_accuracy: 0.1162

Epoch 00001: saving model to ds4a/cp3.ckpt
Epoch 2/10
556/556 [==========] - 391s 703ms/step - loss: nan - accuracy: 0.1096 -
val_loss: nan - val_accuracy: 0.1162

Epoch 00002: saving model to ds4a/cp3.ckpt
Epoch 3/10
556/556 [==========] - 385s 692ms/step - loss: nan - accuracy: 0.1096 -
val_loss: nan - val_accuracy: 0.1162

Epoch 00003: saving model to ds4a/cp3.ckpt
```

Fonte: A Autora (2022).

Para tentarmos resolver o problema, inicialmente, aplicamos um otimizador SDG com uma programação de taxa de aprendizado *(learning rate schedule)*, onde a proposta é ajustar a taxa de aprendizado durante o treinamento, de acordo com uma programação definida utilizando a função *optmizers.schedule.ExponentialDecay* da biblioteca *Keras*. Na programação implementada para este modelo, estamos definindo uma taxa de aprendizado inicial de 0.001, um $decay\_steps = 10000$ e um $decay\_rate = 0.9$, ou seja, a taxa de aprendizado inicial irá diminuir em 0.9 a cada 10000 passos do processo. Entretanto, continuamos com o problema de fuga do gradiente. Somente mudando para o otimizador *ADAM* o problema foi resolvido, conforme podemos observar na Figura 41.

Entretanto, mesmo com a mudança do otimizador, o modelo não apresentou qualquer desempenho, uma vez que ele conseguia aprender somente uma única classe, conforme podemos ver na matriz de confusão da Figura 42.

3.4 Modelo 3 - Arquitetura VGG16 com acréscimo de camadas densas

Novamente, seguindo o fluxo descrito na Figura 33 retornamos agora à fase de modelagem para fazermos alguns ajustes da rede, adicionando as camadas densas ao final da mesma e mantendo descongeladas as camadas pré-treinadas. Este ajuste aumentou

Figura 40 – Descritivo da arquitetura *VGG*.

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 299, 299, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 299, 299, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 299, 299, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 149, 149, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 149, 149, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 149, 149, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 74, 74, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 74, 74, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 74, 74, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 74, 74, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 37, 37, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 37, 37, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 37, 37, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 37, 37, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 18, 18, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 18, 18, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 9, 9, 512)         0
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 9, 9, 512)         14714688
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 10)                5130
=================================================================
Total params: 14,719,818
Trainable params: 14,719,818
Non-trainable params: 0
_________________________________________________________________
```

Fonte: A Autora (2022).

para 33.638.218 o número de parâmetros, conforme detalhamento na Figura 43.

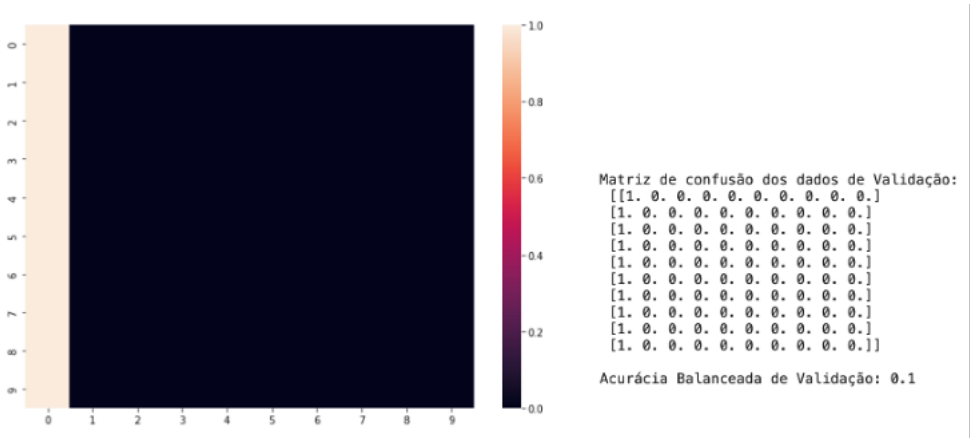
Todos os outros ajustes da modelagem anterior foram mantidos, ou seja, mantivemos o otimizador e a taxa de aprendizado. Após a execução de 10 épocas chegamos a uma acurácia balanceada de 73.7%. Portanto, o ajuste fino na rede incluindo as camadas densas ao final da rede fizeram grande diferença nos resultados, conforme verificado na

Figura 41 – Problema de fuga do gradiente resolvido.

```
Epoch 1/10
556/556 [============] - 767s 1s/step - loss: 2.8575 - accuracy: 0.1069 -
val_loss: 2.3004 - val_accuracy: 0.1162
Epoch 2/10
556/556 [============] - 728s 1s/step - loss: 2.2963 - accuracy: 0.1070 -
val_loss: 2.3026 - val_accuracy: 0.0997
Epoch 3/10
556/556 [============] - 723s 1s/step - loss: 2.2904 - accuracy: 0.1113 -
val_loss: 2.3031 - val_accuracy: 0.0997
Epoch 4/10
556/556 [============] - 718s 1s/step - loss: 2.3001 - accuracy: 0.1024 -
val_loss: 2.3022 - val_accuracy: 0.0997
Epoch 5/10
556/556 [============] - 718s 1s/step - loss: 2.2996 - accuracy: 0.1084 -
val_loss: 2.3022 - val_accuracy: 0.1162
Epoch 6/10
556/556 [============] - 718s 1s/step - loss: 2.2995 - accuracy: 0.1096 -
val_loss: 2.3024 - val_accuracy: 0.1162
Epoch 7/10
556/556 [============] - 718s 1s/step - loss: 2.2995 - accuracy: 0.1096 -
val_loss: 2.3024 - val_accuracy: 0.1162
Epoch 8/10
556/556 [============] - 718s 1s/step - loss: 2.2995 - accuracy: 0.1096 -
val_loss: 2.3024 - val_accuracy: 0.1162
Epoch 9/10
556/556 [============] - 718s 1s/step - loss: 2.2995 - accuracy: 0.1096 -
val_loss: 2.3024 - val_accuracy: 0.1162
Epoch 10/10
556/556 [============] - 719s 1s/step - loss: 2.2995 - accuracy: 0.1096 -
val_loss: 2.3023 - val_accuracy: 0.1162
```

Fonte: A Autora (2022).

Figura 42 – Mapa de calor e a matriz de confusão do Modelo 2.



Fonte: A Autora (2022).

matriz de confusão representada pela Tabela 5 e pelo mapa de calor da Figura 44.

Este foi nosso primeiro modelo com bom desempenho que iremos utilizar como modelo base para os próximos. Submetemos nossos resultados ao *Kaggle* para avaliar nosso modelo com dados reais e o valor da perda logarítmica foi 2.42887. Considerando que o modelo vitorioso tinha uma perda logarítmica menor que 0.1, decidimos avaliar outras possibilidades para tentar melhorar nosso modelo.

Figura 43 – Detalhamento das camadas densas adicionadas ao final da rede.

```
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 9, 9, 512)         14714688
_________________________________________________________________
global_average_pooling2d (Gl (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 4096)              2101248
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dense_2 (Dense)              (None, 10)                40970
=================================================================
Total params: 33,638,218
Trainable params: 33,638,218
Non-trainable params: 0
_________________________________________________________________
```

Fonte: A Autora (2022).

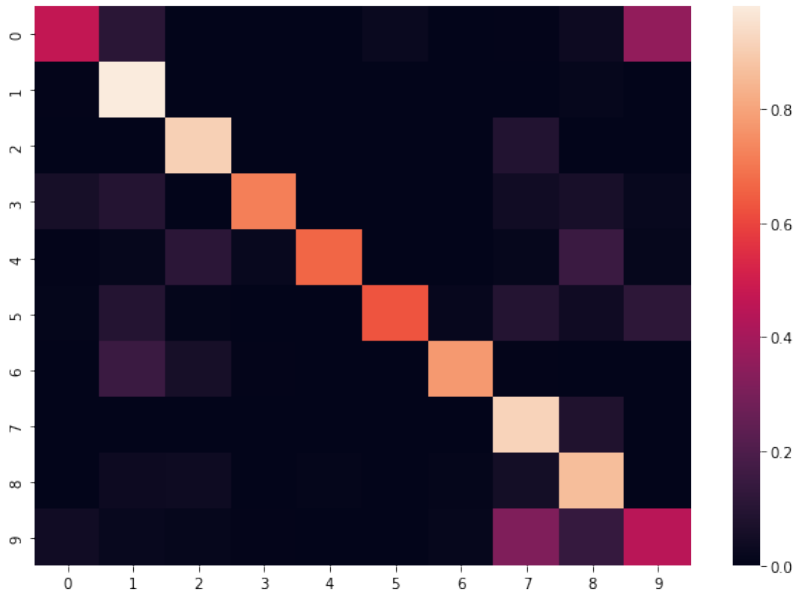
	c0	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**
c0	0.469	0.111	0	0	0	0.025	0	0.006	0.031	0.358
c1	0.007	0.98	0	0	0	0	0	0	0.014	0
c2	0	0	0.911	0	0	0	0	0.089	0	0
c3	0.058	0.094	0	0.717	0	0	0	0.043	0.065	0.022
c4	0.007	0.014	0.114	0.021	0.664	0	0	0.014	0.15	0.014
c5	0.008	0.093	0.008	0	0	0.628	0.016	0.093	0.039	0.116
c6	0	0.152	0.058	0.007	0	0	0.775	0.007	0	0
c7	0	0	0	0	0	0	0	0.917	0.083	0
c8	0	0.031	0.038	0	0.008	0	0.008	0.053	0.863	0
c9	0.044	0.022	0.015	0.007	0	0	0.015	0.314	0.139	0.44
	Acurácia Balanceada de Validação: 0.737									

Tabela 5 – Matriz de confusão dos dados de Validação - Modelo 3.

3.5 Modelos 4 à 10 - arquitetura *ResNet50*

Para os próximos modelos (de 4 até 10) optamos por utilizar a arquitetura *ResNet50* e avaliar suas qualidades. Basicamente, para todos estes modelos, separamos os dados de treino e validação na proporção aproximada de 20% e 80%, respectivamente, ou seja, 4.646 imagens foram destinadas para treino e 17.778 imagens foram destinadas para validação. Também pré processamos todas as imagens para que tivessem resolução 224×224 de entrada, uma vez que este é o padrão da rede. Exceto para o Modelo 10, cada um destes modelos terão um total aproximado de 23,5 milhões de parâmetros, entretanto, somente 20.490 parâmetros serão realmente treinados. Isto ocorre porque inicialmente decidimos congelar as camadas já treinadas da rede e treinar apenas a camada extratora das classes que irá realizar um refinamento da rede. Além disto, incluímos uma camada

Figura 44 – Mapa de calor do Modelo 3.



Fonte: A Autora (2022).

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.506	0.054	0	0.02	0.002	0.004	0.009	0.017	0.306	0.083
c1	0.101	0.489	0.002	0.006	0.01	0	0.012	0.002	0.372	0.006
c2	0	0	0.517	0	0	0	0	0.002	0.468	0.012
c3	0.23	0.007	0	0.246	0	0.002	0.004	0.009	0.463	0.039
c4	0.259	0.009	0.034	0.041	0.009	0.009	0.019	0.004	0.497	0.12
c5	0.136	0.06	0.002	0.012	0.002	0.58	0.005	0.002	0.194	0.007
c6	0.002	0.16	0.017	0	0.002	0	0.343	0	0.471	0.004
c7	0.057	0.077	0.02	0.002	0	0	0.007	0.212	0.556	0.069
c8	0.057	0.016	0.025	0.005	0	0.002	0.078	0.018	0.789	0.009
c9	0.205	0.115	0.009	0.022	0	0.004	0.026	0.028	0.214	0.377
	Acurácia Balanceada de Validação: 0.407									

Tabela 6 – Matriz de confusão dos dados de Validação - Modelo 4.

densa de 10 classes com ativação softmax, que será a camada de saída dos resultados. Finalmente, todos estes modelos deverão rodar até um limite de 10 ou 20 épocas, mas com a regularização de Parada Antecipada *(Early Stopping)*. Por último, para cada um destes 7 modelos fizemos uma série de combinações, tanto na fase de pré processamento, quanto na fase de geração do modelo. A descrição da arquitetura *ResNet*, com o detalhamento das camadas e parâmetros gerados podem ser visto no Apêndice A, pág. 100.

3.5.1 Modelo 4 - otimizador SGD e taxa de aprendizado = 0.0001

Inicialmente, o Modelo 4 utiliza otimizador para gradiente SGD e taxa de aprendizado $\alpha = 0.0001$. Os resultados da validação estão expressos na Tabela 6.

	c0	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**
c0	0.652	0.144	0.004	0.002	0	0.002	0	0.002	0.119	0.076
c1	0.012	0.846	0.02	0	0	0	0	0	0.121	0
c2	0.031	0.006	0.764	0	0	0	0	0.006	0.193	0
c3	0.243	0.011	0	0.683	0	0.002	0	0	0.061	0
c4	0.435	0.013	0.051	0.019	0.336	0.002	0.002	0	0.124	0.017
c5	0.032	0.007	0	0	0	0.956	0	0.005	0	0
c6	0.009	0.149	0.019	0.002	0	0	0.734	0	0.086	0
c7	0.01	0.022	0	0	0	0	0	0.889	0.074	0.005
c8	0.064	0.03	0.039	0	0.007	0	0.014	0.101	0.746	0
c9	0.279	0.089	0.004	0.002	0	0	0.007	0.081	0.333	0.205
	Acurácia Balanceada de Validação: 0.681									

Tabela 7 – Matriz de confusão dos dados de Validação - Modelo 5.

3.5.2 Modelo 5 - otimizador SGD e taxa de aprendizado = 0.01

No Modelo 5, preservamos todas as características do Modelo 4, ou seja, mesma arquitetura, sem transformação de dados, otimizador SGD, mas alteramos a taxa de aprendizado para 0.01. A melhora do desempenho foi significativa, alcançando uma acurácia balanceada de 68.1%, conforme podemos ver na Tabela 7.

3.5.3 Modelo 6 - otimizador SGD e taxa de aprendizado com queda programada

Para o Modelo 6, mantivemos as características do Modelo 4, exceto para a taxa de aprendizado, onde agora utilizaremos uma programação da queda de taxa de aprendizado similar ao aplicado no Modelo 2, só que com taxa de aprendizado inicial igual a 0.01, já que esta mostrou o melhor desempenho até agora. Entretanto, o resultado da acurácia balanceada (67.7%) demonstra que não obtivemos resultados significativos com tal regularização, conforme podemos visualizar nos resultados da Tabela 8.

3.5.4 Modelo 7 - otimizador SGD+momentum e taxa de aprendizado = 0.01

No Modelo 7, optamos por uma combinação de otimizadores SGD + *momentum* com taxa de aprendizado $\alpha = 0.01$. E, como podemos observar na Tabela 9, tivemos uma boa melhora na acurácia balanceada alcançando os 74.9%, o que supera o nosso Modelo 3 tomado como base.

Um outro aspecto interessante quando comparamos o Modelo 7 ao Modelo 3 é que apesar da acurácia balanceada de ambos serem bem próximas (73.7% para o Modelo 3 e 74.9% para o Modelo 7) quando comparamos a perda logarítmica (Modelo 3 = 2.42887; Modelo 7 = 0.83849) vemos que o Modelo 7 é muito mais interessante, já que a sua perda

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.73	0.081	0.002	0.037	0	0	0	0.006	0.111	0.033
c1	0.03	0.844	0.006	0.002	0.004	0	0	0	0.109	0.004
c2	0	0.006	0.786	0	0	0	0.004	0.004	0.191	0.008
c3	0.183	0.028	0	0.698	0	0.002	0	0.004	0.057	0.028
c4	0.379	0.002	0.013	0.019	0.42	0.004	0.002	0.011	0.141	0.009
c5	0.021	0.081	0	0	0	0.885	0	0.009	0.005	0
c6	0	0.259	0.006	0.002	0	0	0.646	0.002	0.084	0
c7	0.007	0.059	0	0.002	0	0	0	0.654	0.259	0.017
c8	0.018	0.039	0.03	0	0.005	0.005	0.037	0.121	0.744	0.002
c9	0.196	0.092	0.015	0.007	0	0.028	0	0.1	0.203	0.359
	Acurácia Balanceada de Validação: 0.677									

Tabela 8 – Matriz de confusão dos dados de Validação - Modelo 6.

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.461	0.011	0	0.072	0	0	0	0.046	0.02	0.389
c1	0.016	0.923	0.006	0.036	0	0	0	0.01	0	0.008
c2	0.006	0	0.887	0	0	0	0	0.039	0.049	0.018
c3	0.052	0	0	0.872	0	0	0	0.017	0.048	0.011
c4	0.028	0	0	0.073	0.548	0	0.002	0.171	0.069	0.109
c5	0.007	0	0	0.166	0	0.79	0	0.018	0.002	0.016
c6	0	0.121	0.015	0	0	0	0.793	0.024	0.048	0
c7	0	0	0	0	0	0	0	0.975	0	0.025
c8	0.005	0.002	0.002	0	0	0	0.007	0.304	0.675	0.005
c9	0.039	0.002	0.004	0.004	0.002	0	0	0.277	0.105	0.566
	Acurácia Balanceada de Validação: 0.749									

Tabela 9 – Matriz de confusão dos dados de Validação - Modelo 7.

logarítmica menor mostra que ele obtém melhor resultado em dados reais, ou seja, sua capacidade de generalização é muito maior do que a do Modelo 3.

Nos Modelos 4, 5, 6 e 7 testamos alguns otimizadores e pudemos observar que o modelo que deu melhor resultado utilizou otimizador SGD+ *momentum* com taxa de aprendizado = 0.01. Nos próximos experimentos iremos manter esta combinação de otimizador e taxa de aprendizado.

3.5.5 Modelo 8 - Transformação de Dados - cisalhamento e *zoom*

Os resultados dos Modelos 4, 5, 6, 7 mostraram que o otimizador SGD + *Momentum* com taxa de aprendizado $\alpha = 0.01$ se adequaram muito bem aos nossos dados. Portanto, vamos adotar esta configuração para nossos próximos modelos.

Para o Modelo 8, voltaremos ao processo de preparação dos dados, de acordo

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.726	0.002	0	0	0	0	0	0.004	0.161	0.107
c1	0.004	0.98	0.002	0.002	0	0	0	0	0.006	0.006
c2	0.012	0.012	0.737	0	0	0	0.014	0.021	0.191	0.012
c3	0.135	0.002	0	0.776	0	0	0	0	0.08	0.007
c4	0.156	0.002	0.002	0.013	0.518	0	0	0	0.18	0.128
c5	0.032	0	0	0.055	0	0.91	0	0	0	0.002
c6	0.009	0.043	0.002	0	0.002	0	0.896	0.002	0.043	0.002
c7	0.002	0.015	0	0	0	0	0.007	0.775	0.151	0.049
c8	0.009	0.03	0.005	0	0.002	0	0.011	0.03	0.908	0.005
c9	0.163	0.013	0.002	0	0.002	0	0	0.02	0.187	0.612
	Acurácia Balanceada de Validação: 0.784									

Tabela 10 – Matriz de confusão dos dados de Validação - Modelo 8.

com a Figura 33, e vamos avaliar algumas técnicas de inserção de dados sintéticos aleatórios dentro do nosso banco de dados *(data augmentation)*. Basicamente, iremos criar outras imagens a partir das imagens originais usando transformações como: translação, rotação, modificação de perspectiva, cisalhamento, espelhamento, ampliação ou redução de regiões da imagem. A biblioteca *Keras* sendo executada com o *TensorFlow* nos permite criar um objeto chamado *ImageDataGenerator* onde é possível especificar uma faixa (em graus) do quanto uma imagem pode ser rotacionada, ou uma faixa (em porcentagem) para deslocamentos verticais ou horizontais, ou ainda se a imagem pode ser espelhada em relação ao eixo x (vertical) ou eixo y (horizontal).

Para nosso banco de dados temos que escolher cuidadosamente estes argumentos, uma vez que as imagens que estamos classificando diferenciam o que é direita e esquerda e isto pode confundir o classificador. Assim, decidimos excluir transformações de rotação, translação e espelhamento. Vamos utilizar apenas *zoom* e cisalhamento. Usaremos a seguinte configuração:

- *Shear_range* = 0.2: transformação de cisalhamento em imagens aleatórias numa taxa de 20%;
- *Zoom_range* = 0.2: ampliação aleatória nas imagens numa taxa de 20%.

Conforme dados da Tabela 10, a acurácia balanceada subiu para 78.4%. Entretanto, a perda logarítmica deste modelo ficou em 0.97819, portanto, maior do que a do modelo anterior. Como o objetivo desta técnica é melhorar a capacidade de generalização do modelo, consideramos que a aplicação de *zoom* e cisalhamento, aplicadas ao mesmo tempo, não trouxe melhoras em relação à generalização do modelo.

	c0	**c1**	**c2**	**c3**	**c4**	**c5**	**c6**	**c7**	**c8**	**c9**
c0	0.474	0.052	0.004	0.106	0	0.004	0	0.087	0.054	0.22
c1	0.004	0.964	0	0.01	0	0	0	0.01	0	0.012
c2	0	0	0.875	0	0	0	0	0.014	0.088	0.023
c3	0.028	0	0	0.9	0	0	0	0.004	0.05	0.017
c4	0.021	0	0	0.056	0.595	0	0	0.064	0.24	0.024
c5	0.009	0	0.002	0.014	0	0.968	0	0.002	0	0.005
c6	0	0.035	0.006	0.004	0	0	0.911	0.015	0.028	0
c7	0	0.002	0	0	0	0	0	0.933	0.052	0.012
c8	0	0.021	0.018	0	0	0	0.009	0.135	0.805	0.011
c9	0.092	0.02	0.022	0.007	0.002	0	0.007	0.161	0.089	0.601
	Acurácia Balanceada de Validação: 0.803									

Tabela 11 – Matriz de confusão dos dados de Validação - Modelo 9.

3.5.6 Modelo 9 - Transformação de Dados - cisalhamento apenas

Assim sendo, para o Modelo 9 utilizamos apenas a transformação de cisalhamento.

Podemos perceber que em nosso Modelo 9, conforme Tabela 11, tivemos uma melhora tanto na acurácia balanceada, que alcançou a marca de 80.3%, quanto na perda logarítmica, que atingiu o valor de 0.75436, que é nossa menor marca até agora. Este resultado nos leva a concluir que a transformação por cisalhamento é a mais adequada para nosso conjunto de dados.

3.5.7 Modelo 10 - Descongelamento das camadas pré-treinadas

Agora que já temos uma noção de qual o melhor otimizador e qual a melhor taxa de aprendizado para nosso modelo, que já identificamos quais técnicas de regularização se ajustam melhor aos nossos dados, queremos verificar se a utilização de camadas descongeladas traz algum ganho para o desempenho do nosso modelo. Nosso objetivo será observar se, ao permitir que o algoritmo atualize os pesos em todas as camadas da rede durante o *backpropagation*, nosso modelo obterá alguma melhora no desempenho.

Feito os devidos ajustes na arquitetura (ou seja, após alterar o hiper parâmetro *layer.trainable* para *True*), executamos o treinamento do nosso Modelo 10.Este levou em torno de 50% de tempo a mais para ser concluído em relação aos modelos anteriores, ou seja, enquanto que os modelos 4 à 9 executaram o treinamento em média em torno de 1 hora, o Modelo 10 levou 1 hora e 35 minutos.

Com relação ao desempenho, vemos na Tabela 12 que a acurácia balanceada subiu para 82.6%. Já a capacidade de generalização piorou em relação ao modelo anterior, uma vez que, ao submetermos os resultados na plataforma *Kaggle*, a perda logarítmica

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.68	0.026	0	0.017	0	0.004	0.002	0.067	0.107	0.098
c1	0.002	0.913	0.034	0.018	0	0	0	0	0.014	0.018
c2	0	0	0.926	0	0	0	0	0.037	0.037	0
c3	0.111	0.009	0	0.852	0	0	0	0.015	0.013	0
c4	0.041	0	0.004	0.015	0.839	0	0.004	0.002	0.088	0.006
c5	0.016	0.002	0	0.039	0	0.94	0	0.002	0	0
c6	0.002	0.05	0.006	0.002	0	0	0.87	0.004	0.065	0
c7	0	0.002	0	0	0	0	0	0.995	0.002	0
c8	0	0.03	0.037	0	0.018	0	0.027	0.098	0.789	0
c9	0.139	0.035	0.013	0.002	0.002	0.035	0.007	0.159	0.157	0.451
	Acurácia Balanceada de Validação: 0.826									

Tabela 12 – Matriz de confusão dos dados de Validação - Modelo 10.

Modelo Base	Resolução
*EfficientNetB*0	224
*EfficientNetB*1	240
*EfficientNetB*2	260
*EfficientNetB*3	300
*EfficientNetB*4	380
*EfficientNetB*5	456
*EfficientNetB*6	528
*EfficientNetB*7	600

Tabela 13 – Resolução de imagem recomendada para cada modelo da família *EfficientNet*.

ficou em 0.91708.

3.6 Modelos 11 à 14 - arquitetura *EfficientNet*

A família *EfficientNet*, apresentada em 2019, é considerada atualmente um dos modelos mais eficientes e de última geração, uma vez que ela tenta associar bons resultados com máxima eficiência de memória e eficiência de tempo de execução.

Ao todo, são 8 modelos da *EfficientNet (B0 a B7)* fornecidas pela biblioteca *Keras*, que representa uma boa combinação de eficiência e precisão em uma variedade de escalas, sem necessariamente alterar a arquitetura de um modelo para outro.

Nossos recursos disponíveis nos permitiram avaliar até o modelo *B3*. Para versões acima desta, são requeridas maiores capacidade de memória e processadores GPU mais robustos, e nossa proposta é tentar fazer o máximo com os recursos gratuitos.

É recomendado que, para cada modelo de arquitetura da família *EfficientNet*, se utilize uma resolução de imagem distinta, adequada a cada versão (detalhes na Tabela 13).

Evidentemente, à medida que aumentamos o modelo base e a resolução da imagem de entrada, também deve aumentar o número de parâmetros a serem treinados. Para todos os experimentos a seguir usaremos apenas os argumentos bem-sucedidos nos modelos anteriores, ou seja: otimizador SGD + *momentum* com taxa de aprendizado 0.01, inserção de dados sintéticos aleatórios de cisalhamento e camadas descongeladas.

Como nos modelos anteriores gerados a partir da arquitetura *ResNet*, vamos inicializar com os pesos da *ImageNet*, e ajustar a camada de saída adicionando uma camada densa para 10 classes com ativação *softmax* que irá garantir que a saída seja interpretada como uma probabilidade em porcentagem de se pertencer a uma determinada classe. Também adicionamos uma camada de *MaxPooling*, uma outra de *BatchNormalization*, e uma última de *Dropout* na razão de 20%.

3.6.1 Modelo 11 - arquitetura *EfficientNetB0*

Como em todos os outros modelos, fizemos o pré processamento das imagens para adaptá-las a entrada da rede, que é uma resolução de 224×224. Em seguida, geramos o resumo da rede, conforme Figura 45.

Figura 45 – Detalhamento da arquitetura *EfficientNet B0*.

```
Model: "model_B0"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
efficientnetb0 (Functional)  (None, None, None, 1280)  4049571
_________________________________________________________________
global_average_pooling2d (Gl (None, 1280)              0
_________________________________________________________________
batch_normalization (BatchNo (None, 1280)              5120
_________________________________________________________________
dropout (Dropout)            (None, 1280)              0
_________________________________________________________________
dense (Dense)                (None, 10)                12810
=================================================================
Total params: 4,067,501
Trainable params: 4,022,918
Non-trainable params: 44,583
_________________________________________________________________
```

Fonte: A Autora (2022).

Veja que esta arquitetura B0 é bem mais enxuta que a *ResNet*, com aproximadamente 4 milhões de parâmetros para treinar apenas. Após o treinamento e validação, o modelo atingiu uma acurácia balanceada de 72.7% e uma perda logarítmica de 0.98190.

Apesar dos resultados da *ResNet* ainda serem melhores, ainda podemos explorar um pouco mais da arquitetura para tentarmos melhorar o desempenho do nosso modelo.

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.522	0.019	0	0.017	0	0	0	0.056	0.13	0.257
c1	0.002	0.865	0	0	0	0	0	0.006	0.125	0.002
c2	0	0.012	0.86	0	0	0	0	0.021	0.101	0.006
c3	0.054	0	0	0.872	0.002	0	0	0.011	0.059	0.002
c4	0.006	0	0	0.017	0.818	0	0	0.064	0.092	0.002
c5	0.014	0.012	0.002	0.185	0	0.751	0	0.009	0.012	0.016
c6	0	0.255	0.05	0	0.006	0	0.514	0.052	0.123	0
c7	0	0	0	0	0	0	0	0.926	0.074	0
c8	0.005	0.014	0.005	0	0.009	0	0.005	0.188	0.776	0
c9	0.068	0.035	0.002	0.004	0.002	0	0	0.192	0.329	0.368
	Acurácia Balanceada de Validação: 0.727									

Tabela 14 – Matriz de confusão dos dados de Validação - Modelo 11.

3.6.2 Modelo 12 - arquitetura *EfficientNetB1*

Para o Modelo 12, utilizamos a *EfficientNetB1* e ajustamos a resolução de entrada para 240×240, conforme o recomendado para esta arquitetura. Fizemos os ajustes necessários na camada de saída e geramos o resumo da rede, conforme Figura 46

Figura 46 – Detalhamento da arquitetura *EfficientNet B1*.

```
Model: "model_B1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 240, 240, 3)]     0
_________________________________________________________________
efficientnetb1 (Functional)  (None, None, None, 1280)  6575239
_________________________________________________________________
global_average_pooling2d (Gl (None, 1280)              0
_________________________________________________________________
batch_normalization (BatchNo (None, 1280)              5120
_________________________________________________________________
dropout (Dropout)            (None, 1280)              0
_________________________________________________________________
dense (Dense)                (None, 10)                12810
=================================================================
Total params: 6,593,169
Trainable params: 6,528,554
Non-trainable params: 64,615
_________________________________________________________________
```

Fonte: A Autora (2022).

Conforme podemos perceber, esta arquitetura, além de permitir uma resolução maior de entrada, ela também tem uma quantidade maior de parâmetros (6.528.554 parâmetros treináveis).

Os resultados apresentados pelo Modelo 12 foram melhores que o anterior, atingindo uma acurácia balanceada de 74.5% e uma perda logarítmica de 0.92868. Entretanto, ele ainda não consegue superar nosso melhor modelo utilizando a *ResNet*.

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.383	0.009	0	0.002	0.02	0.017	0	0.159	0.137	0.272
c1	0	0.84	0	0.004	0	0	0	0.002	0.115	0.038
c2	0	0	0.838	0	0.004	0	0	0.004	0.08	0.074
c3	0.041	0	0	0.872	0	0	0	0.007	0.074	0.007
c4	0.002	0	0.002	0.017	0.657	0	0	0.013	0.101	0.208
c5	0.005	0.016	0	0	0	0.958	0	0.009	0.002	0.009
c6	0	0.11	0.013	0	0.002	0	0.758	0.015	0.091	0.011
c7	0	0.057	0	0	0	0	0	0.768	0.091	0.084
c8	0	0.023	0.021	0	0.032	0	0.002	0.16	0.741	0.021
c9	0.109	0.009	0.002	0	0.002	0.129	0.002	0.07	0.048	0.63
	Acurácia Balanceada de Validação: 0.745									

Tabela 15 – Matriz de confusão dos dados de Validação - Modelo 12.

3.6.3 Modelo 13 - arquitetura *EfficientNetB2*

Para o Modelo 13, utilizamos a *EfficientNetB2* e ajustamos a resolução de entrada para 260×260, conforme o recomendado para esta arquitetura. Fizemos os ajustes necessários na camada de saída e no resumo da rede podemos observar que o número total de parâmetros subiu para 7.788.291, conforme ilustrado na Figura 47.

Figura 47 – Detalhamento da arquitetura *EfficientNet B2*.

```
Model: "model B2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 260, 260, 3)]     0
_________________________________________________________________
efficientnetb2 (Functional)  (None, None, None, 1408)  7768569
_________________________________________________________________
global_average_pooling2d (Gl (None, 1408)              0
_________________________________________________________________
batch_normalization (BatchNo (None, 1408)              5632
_________________________________________________________________
dropout (Dropout)            (None, 1408)              0
_________________________________________________________________
dense (Dense)                (None, 10)                14090
=================================================================
Total params: 7,788,291
Trainable params: 7,717,900
Non-trainable params: 70,391
_________________________________________________________________
```

Fonte: A Autora (2022).

Treinamos o Modelo 13 e pudemos observar que a acurácia balanceada aumentou para 75.3% e a perda logarítmica caiu para 0.52889. Apesar de não atingirmos ainda nosso melhor desempenho em termos de acurácia, a perda logarítmica nos mostra que este é o nosso melhor modelo em termos de generalização até agora.

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.515	0	0	0.011	0	0	0.002	0.111	0.022	0.339
c1	0	0.907	0	0	0	0	0	0.02	0.004	0.069
c2	0.008	0	0.743	0	0	0	0	0	0.021	0.228
c3	0.057	0.002	0	0.893	0	0	0	0.009	0.028	0.011
c4	0.066	0	0	0.013	0.704	0	0.004	0.017	0.069	0.126
c5	0.009	0	0	0	0	0.963	0	0.005	0	0.023
c6	0.002	0.164	0.006	0	0	0	0.693	0.026	0.048	0.06
c7	0.022	0.005	0	0	0	0.005	0	0.654	0.114	0.2
c8	0.032	0.032	0.014	0	0	0	0.014	0.014	0.796	0.098
c9	0.205	0.009	0.007	0	0.004	0	0.009	0.039	0.072	0.656
	Acurácia Balanceada de Validação: 0.753									

Tabela 16 – Matriz de confusão dos dados de Validação - Modelo 13.

3.6.4 Modelo 14 - arquitetura *EfficientNetB3*

Para o Modelo 14, utilizamos a *EfficientNetB3* e ajustamos a resolução de entrada para 300×300, conforme o recomendado para esta arquitetura. Fizemos os ajustes necessários na camada de saída e geramos o resumo da rede. Assim como nos modelos anteriores, o número de parâmetros treináveis agora subiu para 10,7 milhões, conforme podemos visualizar na Figura 48.

Figura 48 – Detalhamento da arquitetura *EfficientNet B3*.

```
Model: "model_B3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 300, 300, 3)]     0
_________________________________________________________________
efficientnetb3 (Functional)  (None, None, None, 1536)  10783535
_________________________________________________________________
global_average_pooling2d (Gl (None, 1536)              0
_________________________________________________________________
batch_normalization (BatchNo (None, 1536)              6144
_________________________________________________________________
dropout (Dropout)            (None, 1536)              0
_________________________________________________________________
dense (Dense)                (None, 10)                15370
=================================================================
Total params: 10,805,049
Trainable params: 10,714,674
Non-trainable params: 90,375
_________________________________________________________________
```

Fonte: A Autora (2022).

Treinamos o Modelo 14 e o resultado foi realmente muito bom. Obtivemos uma acurácia balanceada de 81.2% e a perda logarítmica caiu para 0.49606. Tanto em acurácia balanceada, quanto em generalização do modelo, este foi nosso melhor resultado.

Na Tabela 18 apresentamos um resumo dos resultados de todos os modelos

	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
c0	0.615	0.015	0.002	0	0.002	0.013	0.011	0.015	0.011	0.317
c1	0.01	0.909	0	0.002	0	0	0.006	0.006	0.004	0.063
c2	0.004	0	0.834	0	0	0.006	0	0.023	0.01	0.123
c3	0.07	0	0	0.867	0	0	0.004	0	0.011	0.048
c4	0.021	0	0.004	0.009	0.739	0	0	0.004	0.069	0.154
c5	0.009	0.002	0	0	0	0.975	0	0.007	0	0.007
c6	0.002	0.054	0.006	0	0	0	0.896	0.039	0.002	0
c7	0	0	0	0	0	0	0	0.872	0.121	0.007
c8	0.005	0.023	0.005	0	0.007	0.009	0.014	0.176	0.757	0.005
c9	0.115	0.002	0.002	0	0.002	0	0.004	0.065	0.15	0.658
	Acurácia Balanceada de Validação: 0.812									

Tabela 17 – Matriz de confusão dos dados de Validação - Modelo 14.

Mod	Arquitetura	Camadas Congeladas	Resolução Imagem	Otimizador	Taxa de Aprendizado	Dados Sintéticos	Acurácia Balanceada	Perda Logaritmica
01	CNN do zero	N/A	299 × 299	Adam	0.001	N/A	10.2	-
02	*VGG16*	Não	299 × 299	SGD+momentum	0. 01	Cisalhamento	NaN	-
03	*VGG16*	Não	299 × 299	SGD+Decay	0.001	Cisalhamento	73.7	2.42887
04	*Resnet50*	Sim	224 × 224	SGD	0.0001	N/A	40.7	1.72566
05	*ResNet50*	Sim	224 × 224	SGD	0.01	N/A	68.1	1.16730
06	*ResNet50*	Sim	224 × 224	SGD+Decay	0.01	N/A	67.0	1.34406
07	*ResNet50*	Sim	224 × 224	SGD+momentum	0.01	N/A	74.9	0.83849
08	*ResNet50*	Sim	224 × 224	SGD+momentum	0.01	cisalh+ zoom	78.4	0.97819
09	*ResNet50*	Sim	224 × 224	SGD+momentum	0.01	cisalhamento	80.3	0.75436
10	*ResNet50*	Não	224 × 224	SGD+momentum	0.01	cisalhamento	82.6	0.91708
11	*EfficientNetB0*	Não	224 × 224	SGD+momentum	0.01	cisalhamento	72.7	0.98190
12	*EfficientNetB1*	Não	240 × 240	SGD+momentum	0.01	cisalhamento	74.5	0.92868
13	*EfficientNetB2*	Não	260 × 260	SGD+momentum	0.01	cisalhamento	75.3	0.52889
14	*EfficientNetB3*	Não	300 × 300	SGD+momentum	0.01	cisalhamento	81.2	0.49606

Tabela 18 – Resumo dos modelos treinados.

treinados. Destacamos em amarelo a melhor acurácia balanceada - arquitetura *ResNet50* (Modelo 10) - medida com os dados de validação.

Em verde, destacamos o modelo que demonstrou melhor capacidade de generalização quando exposto aos dados de testes, ou seja, o menor resultado da perda logarítmica - *EfficientNetB3* (Modelo 14) medido à partir do Modelo 3.

4 Resultados e Discussões

Neste trabalho, além da tentativa de treinarmos um modelo de redes neurais para classificação de imagens à partir do zero, avaliamos várias arquiteturas pré-treinadas de redes neurais convolucionais publicadas na literatura para classificar imagens que representavam comportamento de motoristas ao volante.

Os experimentos mostraram que as técnicas de aprendizagem por transferência de conhecimento podem alcançar bons resultados ao alavancar a generalização das camadas iniciais em um domínio diferente. Também pudemos avaliar que os ajustes de hiper parâmetros, tais como, a utilização do otimizador combinado com uma taxa de aprendizado adequada podem trazer resultados significativos aos modelos e, para isto, é importante aliar o entendimento dos dados que estão sendo analisados com o conceito matemático que embasam os algoritmos utilizados.

Um outro aspecto significativo, especialmente nos casos de generalização dos modelos, é a utilização de estratégias de *data augmentation* que tem por objetivo aumentar artificialmente os dados disponíveis. Nos nossos experimentos, provavelmente pelo fato de recebermos uma base de dados apresentando um certo balanceamento de classes, esta técnica não precisou ser muito explorada, mas em casos onde este balanceamento de dados reais nem sempre é possível, esta técnica pode adicionar bons resultados aos experimentos. Ainda assim, nossos experimentos foram suficientes para mostrar que os métodos utilizados (cisalhamento, aplicação de zoom, rotação, etc) precisam ser cuidadosamente escolhidos, pois, caso contrário, podem confundir o classificador.

Pela progressão de resultados demonstrados em nossos modelos, não temos a menor dúvida que se experimentássemos as versões da *EfficientNetB4* ou acima estes números continuariam melhorando, no entanto, uma de nossas propostas aqui era descobrir até onde poderíamos chegar utilizando recursos gratuitos. Até a *EfficientNetB3* conseguimos rodar nossos modelos usando a versão sem custo do *Google Colab*. A partir da versão 4, recursos de memória mais potentes são requeridos e, portanto, não conseguimos continuar o processo.

Para uma análise mais detalhada, na Tabela 19 temos um demonstrativo de quanto foi o acerto em cada classe em cada modelo.

Uma observação bem interessante nesta tabela é que, na média, as classes **C0 - dirigindo em segurança** e **C9 - conversando com o passageiro** foram as classes mais difíceis de acertar. A classe **C4 - falando ao celular - esquerda** também não teve um bom desempenho em determinados modelos (especialmente os modelos com a *ResNet*, porém sua taxa de acertos melhorou nos modelos da *EfficientNet*. Já as classes **C1**

Mod	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
03	46.90	98.00	91.10	71.70	66.40	62.80	77.50	91.70	86.30	44.50
04	50.60	48.90	51.70	24.60	0.90	58.00	34.30	21.20	78.90	37.70
05	65.20	84.60	76.40	68.30	33.60	95.60	73.40	88.90	74.60	20.50
06	56.70	93.70	81.50	69.30	38.80	82.90	59.00	88.60	65.90	33.60
07	46.10	92.30	88.70	87.20	54.80	79.00	79.30	97.50	67.50	56.60
08	72.60	98.00	73.70	77.60	51.80	91.00	89.60	77.50	90.80	61.20
09	47.40	96.40	87.50	90.00	59.50	96.80	91.10	93.30	80.50	60.10
10	68.00	91.30	92.60	85.20	83.90	94.00	87.00	99.50	78.90	45.10
11	52.20	86.50	86.00	87.20	81.80	75.10	51.40	92.60	77.60	36.80
12	38.30	84.00	83.80	87.20	65.70	95.80	75.80	76.80	74.10	63.00
13	51.50	90.70	74.30	89.30	70.40	96.30	69.30	65.40	79.60	65.60
14	61.50	90.90	83.40	86.70	73.90	97.50	89.60	87.20	75.70	65.80
Méd (%)	**54.75**	**87.94**	**80.89**	**77.03**	**56.79**	**85.40**	**73.11**	**81.68**	**77.53**	**49.21**

Tabela 19 – Demonstrativo de acertos por classe.

- digitando mão direita e **C5 - Operando dispositivo no painel** tiveram desempenho acima da média. Isto nos mostra que existem níveis de complexidade diferentes entre as classes, ou seja, determinados comportamentos podem ser tão discretos, como por exemplo, conversar com o passageiro ao lado, que pode ser confundido com dirigir em segurança. Já um movimento de digitação de um determinado lado, ou um braço esticado tentando alcançar o painel pode ser mais facilmente identificado.

Portanto, trabalhos futuros, além de mais recurso computacional, será necessário investigar formas para tratar as particularidades destas classes com menor taxa de acerto.

5 Considerações Finais

É inegável a relevância das redes neurais nos aplicativos e ferramentas que moldam nossas vidas cotidianas. Desde recursos inteligentes em nossos celulares, passando pelo reconhecimento facial ou de voz, ou mecanismos de buscas na internet, até casas inteligentes ou carros autônomos. Neste trabalho focamos apenas em um exemplo de classificação de imagens com objetivo final de identificar comportamentos de motoristas ao dirigir, mas as possibilidades de infusão da inteligência artificial e redes neurais em nosso cotidiano são inúmeras e, certamente terão impacto profundo em nossas vidas num futuro bem próximo.

Um dos aspectos que tornam as redes neurais tão interessantes é que elas conseguem se ajustar a qualquer função matemática contínua, permitindo uma aproximação tão boa quanto desejável. Em outras palavras, não importa qual seja a função, sempre deve existir uma rede neural onde, para cada característica x, o resultado previsto $f(x)$ possa ser transmitido pela rede com uma aproximação suficientemente próxima do resultado real. Esta intuição vem do fato de que, algebricamente, as redes neurais podem ser consideradas um sistema possível e indeterminado. E, este embasamento matemático, aliado a uma quantidade significativa de dados e recursos computacionais cada vez mais rápidos devem tornar as redes neurais cada vez mais precisas.

Um dos focos deste trabalho foi discutir o funcionamento das redes neurais iniciando pela demonstração de seu bloco básico de construção, que é o neurônio artificial, que nada mais é que uma variação do perceptron de *Rosenblatt*. A partir deste ponto, dependendo das características do problema e dos resultados que se pretende alcançar, existe uma função de ativação. Aqui nós discutimos as funções sigmoidal, tangente hiperbólica, $ReLU$ e *Softmax*. Em seguida, apresentamos através de um exemplo prático como é o aprendizado das redes neurais, e discutimos seu componente essencial neste processo que é a descida do gradiente e alguns métodos de otimização, que são essenciais para contornar problemas como a fuga do gradiente, por exemplo.

Como as redes neurais possuem arquiteturas distintas e, cada tipo de arquitetura é adequado para resolver um conjunto específico de problemas, escolhemos focar nas redes neurais convolucionais, que são altamente recomendadas para reconhecimento de imagens. Além disto, discutimos sobre o conceito de transferência de conhecimento, que tem contribuído muito para tornar as redes neurais cada vez mais acessíveis. E aqui, vale destacar a contribuição contínua e contundente da comunidade científica, que organiza e participa de competições, que compartilha resultados de suas técnicas e experimentos, que compartilha seus códigos e contribuem com excelentes artigos, livros e materiais diversos,

o que tem transformado o cenário tecnológico e colocado estes avanços cada vez mais ao alcance de todos.

Finalmente, para ilustrar a nossa discussão, desenvolvemos 14 modelos que tinham por objetivo classificar imagens que representavam o comportamento de motoristas ao volante. A progressão destes modelos, medidas pela acurácia balanceada e pela perda logarítmica, mostravam a importância de cada um dos elementos discutidos anteriormente.

Exceto o primeiro modelo, em que utilizamos o otimizador Adam, todos os outros foram treinados usando gradiente descendente estocástico (SGD), no qual os gradientes são calculados usando o algoritmo de retro propagação, mas associados a diferentes tipos de otimização *(ADAM, Momentum, Decay)* e combinados com diferentes taxas de aprendizado. Além disto, passamos por diferentes arquiteturas de transferência de conhecimento *(VGG, ResNet, EfficientNet)*, onde pudemos evidenciar o quanto é possível usufruir da precisão e eficiência das redes neurais com recursos relativamente simples e, cada vez mais ao alcance de todos.

Por último, mas não menos importante, apesar de não termos abordado diretamente neste trabalho a questão da ética decorrentes de modelos de treinamento em conjuntos de dados que não são suficientemente diversos ou contêm preconceitos humanos, queremos ressaltar a importância da responsabilidade e comprometimento ético dos profissionais que trabalham ou desenvolvem estes recursos, uma vez que estas ferramentas também podem trazer um forte componente de manipulação de escolhas e opiniões. Portanto, cabe aos profissionais que se utilizam delas, a constante vigilância e validação dos resultados obtidos.

Referências

ACADEMY, D. S. *Deep Learning Book*. Online: https://www.deeplearningbook.com.br/uma-breve-historia-das-redes-neurais-artificiais, 2022. Acesso em: 18 set 2021. Citado na página 21.

_____. *Deep Learning Book*. Online: https://www.deeplearningbook.com.br/o-neuronio-biologico-e-matematico, 2022. Acesso em: 25 set 2021. Citado na página 24.

_____. *Deep Learning Book*. Online: https://www.deeplearningbook.com.br/cross-entropy-cost-function, 2022. Acesso em: 18 ago 2021. Citado na página 41.

_____. *Deep Learning Book*. Online: https://www.deeplearningbook.com.br/as-redes-neurais-artificiais-podem-computar-qualquer-funcao, 2022. Acesso em: 18 set 2021. Citado na página 46.

ANZAI, Y. *Pattern recognition and machine learning*. Capítulos 3 e 10: "Morgan Kaufmann", 2012. Citado na página 51.

AVILA, S. *Curso: Mineração de Dados Complexos - Deep Learning*. 2020. Unicamp Moddle. Disponível em: https://moodle.lab.ic.unicamp.br/moodle/course/view.php?id=409. Acesso em: 31 out 2020. Citado 8 vezes nas páginas 43, 44, 50, 52, 53, 54, 56 e 58.

BRAGA, A. S. M.; RODARTE, R.; ROCHA, L. E.; VIANA, H. G. d. M.; PINHEIRO, R. N. *Projeto Final 2020.2 do Curso de Mineração de Dados Complexos; Distracted Driver Detection*. 2020. YouTube. Disponível em: https://www.youtube.com/playlist?list=PLwNGlbMOyZP1kfg5ypE6yd7GkPogp_qQH. Acesso em: 16 dez 2020. Citado 3 vezes nas páginas 68, 69 e 97.

CHOLLET, F. *Deep Learning with Python*. 2. ed. Capítulos: 1 à 8: "Manning Publications", 2019. Citado 5 vezes nas páginas 16, 44, 62, 66 e 67.

COLOMBINI, E. *Curso: Mineração de Dados Complexos - Aprendizado de Máquina Supervisionado II*. 2020. Unicamp Moddle. Disponível em: https://moodle.lab.ic.unicamp.br/moodle/course/view.php?id=406. Acesso em: 03 out 2020. Citado 10 vezes nas páginas 18, 24, 26, 27, 28, 31, 32, 46, 47 e 50.

DUCHI, J.; HAZAN, E.; SINGER, Y. Adaptive subgradient methods for online learning and stochastic optimization. *http://jmlr.org/papers/v12/duchi11a.html*, 2011. Citado na página 45.

GÉRON, A. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. Capítulos: 1-4, 10, 11, 13: "O'Reilly Media, Inc.", 2019. Citado 11 vezes nas páginas 17, 30, 31, 36, 52, 53, 55, 57, 62, 64 e 67.

GOODFELLOW, I.; BENGIO, Y.; COURVILLE, A. *Deep Learning*. Cap.6: MIT Press, 2016. https://www.deeplearningbook.org/contents/mlp.html. Citado na página 26.

_____. *Deep Learning*. Cap.5: MIT Press, 2016. https://www.deeplearningbook.org/contents/ml.html. Citado na página 36.

HE XIANGYU ZHANG, S. R. J. S. K. ImageNet Deep Residual Learning for Image Recognition ResNet. *https://arxiv.org/abs/1512.03385*, 2015. Citado na página 57.

HINTON, G.; SRIVASTAVA, N.; SWERSKY, K. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf*, 2012. Citado 2 vezes nas páginas 43 e 45.

HINTON, G. E.; SALAKHUTDINOV, R. R. Reducing the dimensionality of data with neural networks. *science*, American Association for the Advancement of Science, v. 313, n. 5786, 2006. Citado na página 22.

IBM. *Exploratory Data Analysis for Machine Learning - History of Machine Learning and Deep Learning*. IBM Training: https://learn.ibm.com/mod/video/view.php?id=166092&forceview=1, 2022. Acesso em: 20 nov 2021. Citado na página 22.

_____. *Exploratory Data Analysis for Machine Learning - Machine Learning and Deep Learning*. IBM Training: https://learn.ibm.com/mod/video/view.php?id=166090, 2022. Acesso em: 20 nov 2021. Citado na página 18.

KATHURIA, A. *Intro to optimization in deep learning: Gradient Descent*. 2018. Disponível em: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/amp. Acesso em: 15 out 2021. Citado na página 42.

KINGMA, D. P.; BA, J. Adam: A method for stochastic optimization. *https://arxiv.org/pdf/1412.6980*, 2015. Citado na página 45.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks (AlexNet). *https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html*, 2012. Citado na página 55.

LIMA, E. L. *Análise real*. 12. ed. [S.l.]: Impa Rio de Janeiro, 2016. v. 1. Citado na página 48.

MASOLO, C. Understanding ontological levels. In: LIN, F.; SATTLER, U. (Ed.). *Proceedings of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010)*. AAAI Press, 2010. p. 258–268. Disponível em: http://wiki.loa-cnr.it/Papers/kr10v0.7.pdf. Acesso em: 2 jan. 2012. Nenhuma citação no texto.

MITCHELL, T. M. *Machine Learning*. Capítulo 4: "McGraw-Hill Science/Engineering/-Math", 1997. Citado na página 24.

NG, A. *Machine Learning - week 1: Linear Regression with One Variable*. Coursera: https://www.coursera.org/learn/machine-learning/home/week/1, 2022. Acesso em: 31 out 2021. Citado na página 40.

_____. *Machine Learning - week 3: Logistic Regression*. Coursera: https://www.coursera.org/learn/machine-learning/home/week/3, 2022. Acesso em: 31 out 2021. Citado na página 38.

_____. *Machine Learning - weeks 4 5: Neural Networks*. Coursera: https://www.coursera.org/learn/machine-learning/home, 2022. Acesso em: 31 out 2021. Citado na página 46.

_____. *Neural Networks and Deep Learning - week 1: Introduction to Deep Learning*. Coursera: https://www.coursera.org/learn/neural-networks-deep-learning/lecture/praGm/why-is-deep-learning-taking-off, 2022. Acesso em: 31 ago 2021. Citado na página 16.

_____. *Neural Networks and Deep Learning - week 3: Shallow Neural Network*. Coursera: https://www.coursera.org/learn/neural-networks-deep-learning/lecture/4dDC1/activation-functions, 2022. Acesso em: 30 set 2021. Citado na página 29.

ROCHA, A. *Curso: Mineração de Dados Complexos - Aprendizado de Máquina Supervisionado I*. 2020. Unicamp Moddle. Disponível em: https://moodle.lab.ic.unicamp.br/moodle/course/view.php?id=405. Acesso em: 05 set 2020. Citado 2 vezes nas páginas 36 e 38.

RUDER, S. *An overview of gradient descent optimization algorithms*. 2018. Internet. Disponível em: https://ruder.io/optimizing-gradient-descent. Acesso em: 15 out 2021. Citado 2 vezes nas páginas 43 e 44.

SIMONYAN, K.; ZISSERMAN, A. ImageNet Very Deep Convolutional Networks for Large-Scale Image Recognition VGG-16. *https://arxiv.org/abs/1409.1556*, 2014. Citado na página 56.

TAN, Q. V. L. M. ImageNet - Thinking Model Scaling for Convolutional Neural Networks - EfficientNet. *https://arxiv.org/abs/1905.11946*, 2019. Citado na página 59.

Apêndices

APÊNDICE A – Tutorial do algoritmo base de rede pré-treinada utilizado neste estudo

Os algoritmos deste estudo foram criados utilizando a ferramenta *Google Colaboratory*. Para utilizara ferramenta basta logar no Google Colab https://colab.research.google.com/ através de uma conta Google que pode ser criada gratuitamente, caso o usuário ainda não a tenha. Em seguida, basta criar um novo *notebook* para iniciar seu próprio código, ou então, fazer o upload de um código previamente criado.

Neste trabalho, accessamos as imagens fornecidas pela plataforma *Kaggle* através de sua API. Para isto, é necessário obter suas credenciais na plataforma *Kaggle* contidas em um arquivo *.JSON*. Após o *download* das credenciais em seu computador, será necessário fazer o *upload* das mesmas no *Google Colab* utilizando as duas linhas de comando abaixo:

```
from google.colab import files
files.upload()
```

Um ícone de busca de arquivo deve ser exibido. Basta escolher o arquivo *.JSON* anteriormente baixado da plataforma *Kaggle*. Em seguida, temos que instalar a ferramenta que nos permitirá utilizar a API e, para isto, utilizamos o comando abaixo:

```
!pip install kaggle==1.5.6
```

Também é necessária a criação de um diretório de nome *.kaggle*, copiar o arquivo *kaggle.json* para este novo diretório e ajustar as permissões com o comando *chmod 600*

```
! mkdir ~/.kaggle
! cp kaggle.json ~/.kaggle/
! chmod 600 ~/.kaggle/kaggle.json
```

Na sequência, criamos um diretório local chamado *dataset*, fizemos o *download* das imagens a serem utilizadas neste diretório e descompactamos o arquivo através dos comandos:

```
! mkdir dataset
! kaggle competitions download -c state-farm-distracted-driver-
      detection -p '/content/dataset'
! unzip  '/content/dataset/state-farm-distracted-driver-
      detection.zip' -d '/content/dataset'
```

Antes de continuarmos, é importante fazer um comentário. Este algoritmo foi inicialmente apresentado em (BRAGA et al., 2020). Na época, entendeu-se a primeira versão deste algoritmo atingiu *overfit* uma vez que sua acurácia balanceada nas fases de treino e validação chegava a atingir uma taxa acima de 97%. Entretanto, ao submeter o algoritmo aos dados de testes, a perda logarítmica ficava em torno de 0.4, mas nossa referência, que era o vendedor da competição, deveria ser abaixo de 0.1. Depois de avaliarmos os nossos resultados e pesquisarmos as instruções e comentários na própria plataforma *Kaggle* percebemos que uma particularidade do banco de dados poderia estar nos levando ao resultado de *overfit*. E esta particularidade é o fato de que as imagens não são situações reais, mas sim, são imagens produzidas por atores simulando distrações de motoristas ao volante e, um único ator, estaria representando mais de uma situação, ou seja, um mesmo ator estaria contido em várias classes e, pelo visto, parte do aprendizado do nosso algoritmo tinha a ver com o reconhecimento facial de cada ator. Entretanto, nosso propósito era apenas o reconhecimento do comportamento (classes). Este entendimento aconteceu nas etapas finais do trabalho e, portanto, não houve tempo hábil para maiores explorações. Entretanto, nesta releitura deste algoritmo, este aprendizado foi incorporado. Portanto, o próximo passo é incluir alguns comandos que irão garantir que na separação dos dados de treino e validação, os motoristas do conjunto de treino não sejam os mesmos dos conjunto de validação e vice versa.

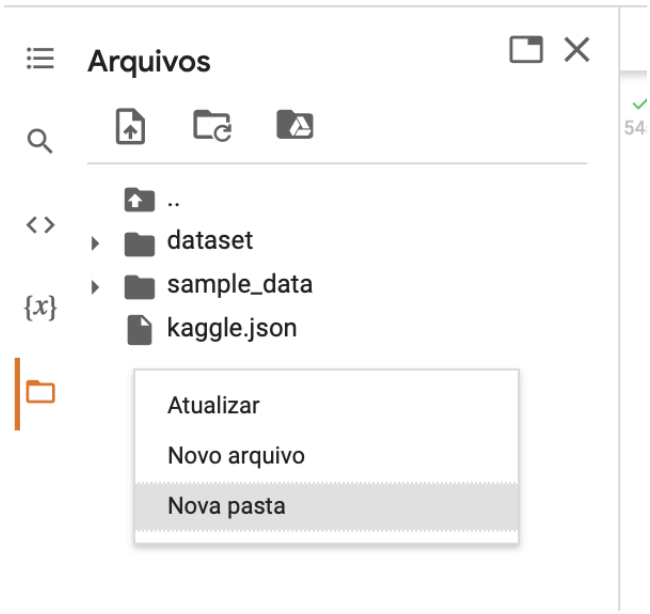
Assim sendo, com as imagens em nosso ambiente virtual local descompactadas, iremos garantir que atores do grupo de treino não poderão entrar no grupo de validação. Para isto, criamos 2 arquivos *.csv* distintos (**trainamento.csv** e **validacao.csv**) identificando quais atores estariam em cada grupo. No *Google Colab*, à esquerda na secão *arquivos*, conforme Figura 49 criamos a pasta *split*

Em seguida, fizemos o upload dos arquivos **trainamento.csv** e **validacao.csv** na pasta *split*. Na sequência, o próximo bloco de comandos irão garantir esta separação das imagens que já estão em diretório local do *Google Colab*:

```
import os
import pandas as pd
import shutil

print (os.getcwd())
list_train = pd.read_csv(os.getcwd()+'/split/
      trainamento.csv',sep=";")
list_valid = pd.read_csv(os.getcwd()+'/split/
      validacao.csv',sep=";")
final_split = os.getcwd()
os.listdir('/content/dataset/imgs/train')
```

Figura 49 – Tela do *Google colab*.




```
labels=os.listdir('/content/dataset/imgs/train')
for key in labels:
    path= final_split + '/Train/'+key
    path_valid= final_split + '/Valid/'+key
    if(os.path.isdir(path)):
        shutil.rmtree(path)
    if(os.path.isdir(path_valid)):
        shutil.rmtree(path_valid)
    os.makedirs(path,exist_ok=True)
    os.makedirs(path_valid,exist_ok=True)

path = '/content/dataset/imgs/train'
path_dst = final_split + '/Train/'
for i,line in list_train.iterrows():
    folder = line['pasta']
    file = line['imagem']
    print(line['pasta'] + '/' + line['imagem'])
    shutil.copy(path +'/'+folder+ '/'+file ,
          path_dst+'/'+folder+ '/'+file)

path = '/content/dataset/imgs/train'
path_dst = final_split + '/Valid/'

for i,line in list_valid.iterrows():
    folder = line['pasta']
    file = line['imagem']
    print(line['pasta'] + '/' + line['imagem'])
    shutil.copy(path +'/'+folder+ '/'+file ,
          path_dst+'/'+folder+ '/'+file)
```

O próximo passo é o pré-processamento das imagens. Para isto, importamos para o *Python* as bibliotecas necessárias:

```
from keras.applications.resnet import preprocess_input
import tensorflow as tf
from PIL import Image
import matplotlib.pyplot as plt
```

Em seguida, utilizamos a classe *ImageDataGenerator()* da biblioteca *Keras* que nos permitirá utilizar algumas técnicas de *data augmentation*, ou seja, vamos promover algumas transformações nas imagens com o objetivo de aumentar sua diversidade dos dados, sem ter que coletar novas imagens. No exemplo abaixo, vamos fazer uma transformação de cisalhamento na imagem (*shear_range=0.2*) com ângulo de cisalhamento = 0.2 no sentido anti-horário em radianos. Entretanto, em tutoriais da biblioteca *Keras* como deste link : https://faroit.com/keras-docs/1.0.6/preprocessing/image, é possível encontrar todos os argumentos que podem ser usados.

```
train_datagen = tf.keras.preprocessing.image.ImageDataGenerator
      (preprocessing_function=tf.keras.applications.resnet50.
       preprocess_input, shear_range=0.2)
```

Com o *dataset* pré-processado gerado, o próximo passo é fazer uma separação entre os dados que serão destinados a treino e validação. Para isto, vamos utilizar a função *image_dataset_from_directory*, que irá gerar os subdiretórios de imagens de validação e treino. Além disto, iremos redimensionar o tamanho das imagens para o tamanho adequado de cada rede pré-treinada (neste exemplo: 224 × 224, uma vez que este é o padrão recomendado pela rede pré-treinada *ResNet*). Existem outros argumentos que podem ser utilizados dentro desta função. Maiores detalhes, podem ser vistos na documentação oficial da biblioteca *Keras* no link: https://keras.io/api/preprocessing/image/#image-data-preprocessing

```
train_generator = train_datagen.flow_from_directory('/content/
      Train', target_size=(224, 224),
      batch_size=32, shuffle=True,
      class_mode='categorical', subset='training')

validation_generator = train_datagen.flow_from_directory
      ('/content/Valid', target_size=(224, 224),
      batch_size=32, class_mode='categorical', shuffle=False)
```

Em muitos casos de *machine learning*, os dados disponíveis nunca estão balanceados, ou seja, não existe uma quantidade de amostras igual entre as classes. Isto quase sempre é um problema, uma vez que este desbalanceamento pode fazer com que seu

algoritmo de classificação, ao tentar adivinhar uma classe, ele dê preferência àquelas com maior número de amostras, criando assim, um viés para estes dados. No banco de dados utilizado para este trabalho, excepcionalmente, as classes até possuíam um balanceamento aceitável, mas de qualquer forma, inserimos o código abaixo, para adicionar um peso a cada classe, de acordo com sua quantidade de amostras, como forma de corrigir um eventual desbalanceamento.

```
from collections import Counter
counter = Counter(train_generator.classes)
max_val = float(max(counter.values()))
class_weights = {class_id : max_val/num_images for class_id,
      num_images in counter.items()}
print(class_weights)
```

Agora, vamos criar o modelo de treinamento utilizando o pacote *Model* da biblioteca *Keras*. Neste modelo, vamos utilizar a rede pré treinada *ResNet* e utilizar os pesos aprendidos na competição *ImageNet*

```
from tensorflow.keras.models import Model
model = None
full_model = None
model = tf.keras.applications.ResNet50(weights='imagenet',
      include_top=True)
model = Model(inputs=model.input, outputs=model.
      get_layer('avg_pool').output)
model.summary()
```

O resultado deste comando acima será exibir a arquitetura da rede *ResNet* onde o usuário poderá analisar em detalhes como suas camadas são compostas. Veja no final que temos 23.534.592 parâmetros treináveis, ou seja, aqueles que devem ser atualizados (via gradiente descendente) para minimizar a perda durante o treinamento. E temos 53.120 parâmetros não treináveis, que normalmente, são atualizados pelo modelo durante a passagem direta.

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/
      resnet/resnet50_weights_tf_dim_ordering_tf_kernels.h5

102973440/102967424 [==============================] - 7s 0us/step
102981632/102967424 [==============================] - 7s 0us/step

Model: "model"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 224, 224, 3) 0
__________________________________________________________________________________________________
conv1_pad (ZeroPadding2D)       (None, 230, 230, 3)  0           input_1[0][0]
__________________________________________________________________________________________________
```

```
conv1_conv (Conv2D)             (None, 112, 112, 64) 9472        conv1_pad[0][0]
__________________________________________________________________________________________________
conv1_bn (BatchNormalization)   (None, 112, 112, 64) 256         conv1_conv[0][0]
__________________________________________________________________________________________________
conv1_relu (Activation)         (None, 112, 112, 64) 0           conv1_bn[0][0]
__________________________________________________________________________________________________
pool1_pad (ZeroPadding2D)       (None, 114, 114, 64) 0           conv1_relu[0][0]
__________________________________________________________________________________________________
pool1_pool (MaxPooling2D)       (None, 56, 56, 64)   0           pool1_pad[0][0]
__________________________________________________________________________________________________
conv2_block1_1_conv (Conv2D)    (None, 56, 56, 64)   4160        pool1_pool[0][0]
__________________________________________________________________________________________________
conv2_block1_1_bn (BatchNormali (None, 56, 56, 64)   256         conv2_block1_1_conv[0][0]
__________________________________________________________________________________________________
conv2_block1_1_relu (Activation (None, 56, 56, 64)   0           conv2_block1_1_bn[0][0]
__________________________________________________________________________________________________
conv2_block1_2_conv (Conv2D)    (None, 56, 56, 64)   36928       conv2_block1_1_relu[0][0]
__________________________________________________________________________________________________
conv2_block1_2_bn (BatchNormali (None, 56, 56, 64)   256         conv2_block1_2_conv[0][0]
__________________________________________________________________________________________________
conv2_block1_2_relu (Activation (None, 56, 56, 64)   0           conv2_block1_2_bn[0][0]
__________________________________________________________________________________________________
conv2_block1_0_conv (Conv2D)    (None, 56, 56, 256)  16640       pool1_pool[0][0]
__________________________________________________________________________________________________
conv2_block1_3_conv (Conv2D)    (None, 56, 56, 256)  16640       conv2_block1_2_relu[0][0]
__________________________________________________________________________________________________
conv2_block1_0_bn (BatchNormali (None, 56, 56, 256)  1024        conv2_block1_0_conv[0][0]
__________________________________________________________________________________________________
conv2_block1_3_bn (BatchNormali (None, 56, 56, 256)  1024        conv2_block1_3_conv[0][0]
__________________________________________________________________________________________________
conv2_block1_add (Add)          (None, 56, 56, 256)  0           conv2_block1_0_bn[0][0]
                                                                 conv2_block1_3_bn[0][0]
__________________________________________________________________________________________________
conv2_block1_out (Activation)   (None, 56, 56, 256)  0           conv2_block1_add[0][0]
__________________________________________________________________________________________________
conv2_block2_1_conv (Conv2D)    (None, 56, 56, 64)   16448       conv2_block1_out[0][0]
__________________________________________________________________________________________________
conv2_block2_1_bn (BatchNormali (None, 56, 56, 64)   256         conv2_block2_1_conv[0][0]
__________________________________________________________________________________________________
conv2_block2_1_relu (Activation (None, 56, 56, 64)   0           conv2_block2_1_bn[0][0]
__________________________________________________________________________________________________
conv2_block2_2_conv (Conv2D)    (None, 56, 56, 64)   36928       conv2_block2_1_relu[0][0]
__________________________________________________________________________________________________
conv2_block2_2_bn (BatchNormali (None, 56, 56, 64)   256         conv2_block2_2_conv[0][0]
__________________________________________________________________________________________________
conv2_block2_2_relu (Activation (None, 56, 56, 64)   0           conv2_block2_2_bn[0][0]
__________________________________________________________________________________________________
conv2_block2_3_conv (Conv2D)    (None, 56, 56, 256)  16640       conv2_block2_2_relu[0][0]
__________________________________________________________________________________________________
conv2_block2_3_bn (BatchNormali (None, 56, 56, 256)  1024        conv2_block2_3_conv[0][0]
__________________________________________________________________________________________________
conv2_block2_add (Add)          (None, 56, 56, 256)  0           conv2_block1_out[0][0]
                                                                 conv2_block2_3_bn[0][0]
__________________________________________________________________________________________________
conv2_block2_out (Activation)   (None, 56, 56, 256)  0           conv2_block2_add[0][0]
__________________________________________________________________________________________________
conv2_block3_1_conv (Conv2D)    (None, 56, 56, 64)   16448       conv2_block2_out[0][0]
__________________________________________________________________________________________________
conv2_block3_1_bn (BatchNormali (None, 56, 56, 64)   256         conv2_block3_1_conv[0][0]
__________________________________________________________________________________________________
conv2_block3_1_relu (Activation (None, 56, 56, 64)   0           conv2_block3_1_bn[0][0]
__________________________________________________________________________________________________
conv2_block3_2_conv (Conv2D)    (None, 56, 56, 64)   36928       conv2_block3_1_relu[0][0]
__________________________________________________________________________________________________
conv2_block3_2_bn (BatchNormali (None, 56, 56, 64)   256         conv2_block3_2_conv[0][0]
__________________________________________________________________________________________________
conv2_block3_2_relu (Activation (None, 56, 56, 64)   0           conv2_block3_2_bn[0][0]
__________________________________________________________________________________________________
conv2_block3_3_conv (Conv2D)    (None, 56, 56, 256)  16640       conv2_block3_2_relu[0][0]
__________________________________________________________________________________________________
conv2_block3_3_bn (BatchNormali (None, 56, 56, 256)  1024        conv2_block3_3_conv[0][0]
```

```
conv2_block3_add (Add)          (None, 56, 56, 256)  0           conv2_block2_out[0][0]
                                                                 conv2_block3_3_bn[0][0]
conv2_block3_out (Activation)   (None, 56, 56, 256)  0           conv2_block3_add[0][0]
conv3_block1_1_conv (Conv2D)    (None, 28, 28, 128)  32896       conv2_block3_out[0][0]
conv3_block1_1_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block1_1_conv[0][0]
conv3_block1_1_relu (Activation (None, 28, 28, 128)  0           conv3_block1_1_bn[0][0]
conv3_block1_2_conv (Conv2D)    (None, 28, 28, 128)  147584      conv3_block1_1_relu[0][0]
conv3_block1_2_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block1_2_conv[0][0]
conv3_block1_2_relu (Activation (None, 28, 28, 128)  0           conv3_block1_2_bn[0][0]
conv3_block1_0_conv (Conv2D)    (None, 28, 28, 512)  131584      conv2_block3_out[0][0]
conv3_block1_3_conv (Conv2D)    (None, 28, 28, 512)  66048       conv3_block1_2_relu[0][0]
conv3_block1_0_bn (BatchNormali (None, 28, 28, 512)  2048        conv3_block1_0_conv[0][0]
conv3_block1_3_bn (BatchNormali (None, 28, 28, 512)  2048        conv3_block1_3_conv[0][0]
conv3_block1_add (Add)          (None, 28, 28, 512)  0           conv3_block1_0_bn[0][0]
                                                                 conv3_block1_3_bn[0][0]
conv3_block1_out (Activation)   (None, 28, 28, 512)  0           conv3_block1_add[0][0]
conv3_block2_1_conv (Conv2D)    (None, 28, 28, 128)  65664       conv3_block1_out[0][0]
conv3_block2_1_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block2_1_conv[0][0]
conv3_block2_1_relu (Activation (None, 28, 28, 128)  0           conv3_block2_1_bn[0][0]
conv3_block2_2_conv (Conv2D)    (None, 28, 28, 128)  147584      conv3_block2_1_relu[0][0]
conv3_block2_2_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block2_2_conv[0][0]
conv3_block2_2_relu (Activation (None, 28, 28, 128)  0           conv3_block2_2_bn[0][0]
conv3_block2_3_conv (Conv2D)    (None, 28, 28, 512)  66048       conv3_block2_2_relu[0][0]
conv3_block2_3_bn (BatchNormali (None, 28, 28, 512)  2048        conv3_block2_3_conv[0][0]
conv3_block2_add (Add)          (None, 28, 28, 512)  0           conv3_block1_out[0][0]
                                                                 conv3_block2_3_bn[0][0]
conv3_block2_out (Activation)   (None, 28, 28, 512)  0           conv3_block2_add[0][0]
conv3_block3_1_conv (Conv2D)    (None, 28, 28, 128)  65664       conv3_block2_out[0][0]
conv3_block3_1_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block3_1_conv[0][0]
conv3_block3_1_relu (Activation (None, 28, 28, 128)  0           conv3_block3_1_bn[0][0]
conv3_block3_2_conv (Conv2D)    (None, 28, 28, 128)  147584      conv3_block3_1_relu[0][0]
conv3_block3_2_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block3_2_conv[0][0]
conv3_block3_2_relu (Activation (None, 28, 28, 128)  0           conv3_block3_2_bn[0][0]
conv3_block3_3_conv (Conv2D)    (None, 28, 28, 512)  66048       conv3_block3_2_relu[0][0]
conv3_block3_3_bn (BatchNormali (None, 28, 28, 512)  2048        conv3_block3_3_conv[0][0]
conv3_block3_add (Add)          (None, 28, 28, 512)  0           conv3_block2_out[0][0]
                                                                 conv3_block3_3_bn[0][0]
```

```
conv3_block3_out (Activation)   (None, 28, 28, 512)  0           conv3_block3_add[0][0]
__________________________________________________________________________________________________
conv3_block4_1_conv (Conv2D)    (None, 28, 28, 128)  65664       conv3_block3_out[0][0]
__________________________________________________________________________________________________
conv3_block4_1_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block4_1_conv[0][0]
__________________________________________________________________________________________________
conv3_block4_1_relu (Activation (None, 28, 28, 128)  0           conv3_block4_1_bn[0][0]
__________________________________________________________________________________________________
conv3_block4_2_conv (Conv2D)    (None, 28, 28, 128)  147584      conv3_block4_1_relu[0][0]
__________________________________________________________________________________________________
conv3_block4_2_bn (BatchNormali (None, 28, 28, 128)  512         conv3_block4_2_conv[0][0]
__________________________________________________________________________________________________
conv3_block4_2_relu (Activation (None, 28, 28, 128)  0           conv3_block4_2_bn[0][0]
__________________________________________________________________________________________________
conv3_block4_3_conv (Conv2D)    (None, 28, 28, 512)  66048       conv3_block4_2_relu[0][0]
__________________________________________________________________________________________________
conv3_block4_3_bn (BatchNormali (None, 28, 28, 512)  2048        conv3_block4_3_conv[0][0]
__________________________________________________________________________________________________
conv3_block4_add (Add)          (None, 28, 28, 512)  0           conv3_block3_out[0][0]
                                                                 conv3_block4_3_bn[0][0]
__________________________________________________________________________________________________
conv3_block4_out (Activation)   (None, 28, 28, 512)  0           conv3_block4_add[0][0]
__________________________________________________________________________________________________
conv4_block1_1_conv (Conv2D)    (None, 14, 14, 256)  131328      conv3_block4_out[0][0]
__________________________________________________________________________________________________
conv4_block1_1_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block1_1_conv[0][0]
__________________________________________________________________________________________________
conv4_block1_1_relu (Activation (None, 14, 14, 256)  0           conv4_block1_1_bn[0][0]
__________________________________________________________________________________________________
conv4_block1_2_conv (Conv2D)    (None, 14, 14, 256)  590080      conv4_block1_1_relu[0][0]
__________________________________________________________________________________________________
conv4_block1_2_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block1_2_conv[0][0]
__________________________________________________________________________________________________
conv4_block1_2_relu (Activation (None, 14, 14, 256)  0           conv4_block1_2_bn[0][0]
__________________________________________________________________________________________________
conv4_block1_0_conv (Conv2D)    (None, 14, 14, 1024) 525312      conv3_block4_out[0][0]
__________________________________________________________________________________________________
conv4_block1_3_conv (Conv2D)    (None, 14, 14, 1024) 263168      conv4_block1_2_relu[0][0]
__________________________________________________________________________________________________
conv4_block1_0_bn (BatchNormali (None, 14, 14, 1024) 4096        conv4_block1_0_conv[0][0]
__________________________________________________________________________________________________
conv4_block1_3_bn (BatchNormali (None, 14, 14, 1024) 4096        conv4_block1_3_conv[0][0]
__________________________________________________________________________________________________
conv4_block1_add (Add)          (None, 14, 14, 1024) 0           conv4_block1_0_bn[0][0]
                                                                 conv4_block1_3_bn[0][0]
__________________________________________________________________________________________________
conv4_block1_out (Activation)   (None, 14, 14, 1024) 0           conv4_block1_add[0][0]
__________________________________________________________________________________________________
conv4_block2_1_conv (Conv2D)    (None, 14, 14, 256)  262400      conv4_block1_out[0][0]
__________________________________________________________________________________________________
conv4_block2_1_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block2_1_conv[0][0]
__________________________________________________________________________________________________
conv4_block2_1_relu (Activation (None, 14, 14, 256)  0           conv4_block2_1_bn[0][0]
__________________________________________________________________________________________________
conv4_block2_2_conv (Conv2D)    (None, 14, 14, 256)  590080      conv4_block2_1_relu[0][0]
__________________________________________________________________________________________________
conv4_block2_2_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block2_2_conv[0][0]
__________________________________________________________________________________________________
conv4_block2_2_relu (Activation (None, 14, 14, 256)  0           conv4_block2_2_bn[0][0]
__________________________________________________________________________________________________
conv4_block2_3_conv (Conv2D)    (None, 14, 14, 1024) 263168      conv4_block2_2_relu[0][0]
__________________________________________________________________________________________________
conv4_block2_3_bn (BatchNormali (None, 14, 14, 1024) 4096        conv4_block2_3_conv[0][0]
__________________________________________________________________________________________________
conv4_block2_add (Add)          (None, 14, 14, 1024) 0           conv4_block1_out[0][0]
                                                                 conv4_block2_3_bn[0][0]
__________________________________________________________________________________________________
conv4_block2_out (Activation)   (None, 14, 14, 1024) 0           conv4_block2_add[0][0]
__________________________________________________________________________________________________
conv4_block3_1_conv (Conv2D)    (None, 14, 14, 256)  262400      conv4_block2_out[0][0]
__________________________________________________________________________________________________
```

```
conv4_block3_1_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block3_1_conv[0][0]
__________________________________________________________________________________________________
conv4_block3_1_relu (Activation (None, 14, 14, 256)  0           conv4_block3_1_bn[0][0]
__________________________________________________________________________________________________
conv4_block3_2_conv (Conv2D)    (None, 14, 14, 256)  590080      conv4_block3_1_relu[0][0]
__________________________________________________________________________________________________
conv4_block3_2_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block3_2_conv[0][0]
__________________________________________________________________________________________________
conv4_block3_2_relu (Activation (None, 14, 14, 256)  0           conv4_block3_2_bn[0][0]
__________________________________________________________________________________________________
conv4_block3_3_conv (Conv2D)    (None, 14, 14, 1024) 263168      conv4_block3_2_relu[0][0]
__________________________________________________________________________________________________
conv4_block3_3_bn (BatchNormali (None, 14, 14, 1024) 4096        conv4_block3_3_conv[0][0]
__________________________________________________________________________________________________
conv4_block3_add (Add)          (None, 14, 14, 1024) 0           conv4_block2_out[0][0]
                                                                 conv4_block3_3_bn[0][0]
__________________________________________________________________________________________________
conv4_block3_out (Activation)   (None, 14, 14, 1024) 0           conv4_block3_add[0][0]
__________________________________________________________________________________________________
conv4_block4_1_conv (Conv2D)    (None, 14, 14, 256)  262400      conv4_block3_out[0][0]
__________________________________________________________________________________________________
conv4_block4_1_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block4_1_conv[0][0]
__________________________________________________________________________________________________
conv4_block4_1_relu (Activation (None, 14, 14, 256)  0           conv4_block4_1_bn[0][0]
__________________________________________________________________________________________________
conv4_block4_2_conv (Conv2D)    (None, 14, 14, 256)  590080      conv4_block4_1_relu[0][0]
__________________________________________________________________________________________________
conv4_block4_2_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block4_2_conv[0][0]
__________________________________________________________________________________________________
conv4_block4_2_relu (Activation (None, 14, 14, 256)  0           conv4_block4_2_bn[0][0]
__________________________________________________________________________________________________
conv4_block4_3_conv (Conv2D)    (None, 14, 14, 1024) 263168      conv4_block4_2_relu[0][0]
__________________________________________________________________________________________________
conv4_block4_3_bn (BatchNormali (None, 14, 14, 1024) 4096        conv4_block4_3_conv[0][0]
__________________________________________________________________________________________________
conv4_block4_add (Add)          (None, 14, 14, 1024) 0           conv4_block3_out[0][0]
                                                                 conv4_block4_3_bn[0][0]
__________________________________________________________________________________________________
conv4_block4_out (Activation)   (None, 14, 14, 1024) 0           conv4_block4_add[0][0]
__________________________________________________________________________________________________
conv4_block5_1_conv (Conv2D)    (None, 14, 14, 256)  262400      conv4_block4_out[0][0]
__________________________________________________________________________________________________
conv4_block5_1_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block5_1_conv[0][0]
__________________________________________________________________________________________________
conv4_block5_1_relu (Activation (None, 14, 14, 256)  0           conv4_block5_1_bn[0][0]
__________________________________________________________________________________________________
conv4_block5_2_conv (Conv2D)    (None, 14, 14, 256)  590080      conv4_block5_1_relu[0][0]
__________________________________________________________________________________________________
conv4_block5_2_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block5_2_conv[0][0]
__________________________________________________________________________________________________
conv4_block5_2_relu (Activation (None, 14, 14, 256)  0           conv4_block5_2_bn[0][0]
__________________________________________________________________________________________________
conv4_block5_3_conv (Conv2D)    (None, 14, 14, 1024) 263168      conv4_block5_2_relu[0][0]
__________________________________________________________________________________________________
conv4_block5_3_bn (BatchNormali (None, 14, 14, 1024) 4096        conv4_block5_3_conv[0][0]
__________________________________________________________________________________________________

conv4_block5_add (Add)          (None, 14, 14, 1024) 0           conv4_block4_out[0][0]
                                                                 conv4_block5_3_bn[0][0]
__________________________________________________________________________________________________
conv4_block5_out (Activation)   (None, 14, 14, 1024) 0           conv4_block5_add[0][0]
__________________________________________________________________________________________________
conv4_block6_1_conv (Conv2D)    (None, 14, 14, 256)  262400      conv4_block5_out[0][0]
__________________________________________________________________________________________________
conv4_block6_1_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block6_1_conv[0][0]
__________________________________________________________________________________________________
conv4_block6_1_relu (Activation (None, 14, 14, 256)  0           conv4_block6_1_bn[0][0]
__________________________________________________________________________________________________
conv4_block6_2_conv (Conv2D)    (None, 14, 14, 256)  590080      conv4_block6_1_relu[0][0]
__________________________________________________________________________________________________
conv4_block6_2_bn (BatchNormali (None, 14, 14, 256)  1024        conv4_block6_2_conv[0][0]
```

```
conv4_block6_2_relu (Activation (None, 14, 14, 256)  0           conv4_block6_2_bn[0][0]
__________________________________________________________________________________________________
conv4_block6_3_conv (Conv2D)    (None, 14, 14, 1024) 263168      conv4_block6_2_relu[0][0]
__________________________________________________________________________________________________
conv4_block6_3_bn (BatchNormali (None, 14, 14, 1024) 4096        conv4_block6_3_conv[0][0]
__________________________________________________________________________________________________
conv4_block6_add (Add)          (None, 14, 14, 1024) 0           conv4_block5_out[0][0]
                                                                 conv4_block6_3_bn[0][0]
__________________________________________________________________________________________________
conv4_block6_out (Activation)   (None, 14, 14, 1024) 0           conv4_block6_add[0][0]
__________________________________________________________________________________________________
conv5_block1_1_conv (Conv2D)    (None, 7, 7, 512)    524800      conv4_block6_out[0][0]
__________________________________________________________________________________________________
conv5_block1_1_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block1_1_conv[0][0]
__________________________________________________________________________________________________
conv5_block1_1_relu (Activation (None, 7, 7, 512)    0           conv5_block1_1_bn[0][0]
__________________________________________________________________________________________________
conv5_block1_2_conv (Conv2D)    (None, 7, 7, 512)    2359808     conv5_block1_1_relu[0][0]
__________________________________________________________________________________________________
conv5_block1_2_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block1_2_conv[0][0]
__________________________________________________________________________________________________
conv5_block1_2_relu (Activation (None, 7, 7, 512)    0           conv5_block1_2_bn[0][0]
__________________________________________________________________________________________________
conv5_block1_0_conv (Conv2D)    (None, 7, 7, 2048)   2099200     conv4_block6_out[0][0]
__________________________________________________________________________________________________
conv5_block1_3_conv (Conv2D)    (None, 7, 7, 2048)   1050624     conv5_block1_2_relu[0][0]
__________________________________________________________________________________________________
conv5_block1_0_bn (BatchNormali (None, 7, 7, 2048)   8192        conv5_block1_0_conv[0][0]
__________________________________________________________________________________________________
conv5_block1_3_bn (BatchNormali (None, 7, 7, 2048)   8192        conv5_block1_3_conv[0][0]
__________________________________________________________________________________________________
conv5_block1_add (Add)          (None, 7, 7, 2048)   0           conv5_block1_0_bn[0][0]
                                                                 conv5_block1_3_bn[0][0]
__________________________________________________________________________________________________
conv5_block1_out (Activation)   (None, 7, 7, 2048)   0           conv5_block1_add[0][0]
__________________________________________________________________________________________________
conv5_block2_1_conv (Conv2D)    (None, 7, 7, 512)    1049088     conv5_block1_out[0][0]
__________________________________________________________________________________________________
conv5_block2_1_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block2_1_conv[0][0]
__________________________________________________________________________________________________
conv5_block2_1_relu (Activation (None, 7, 7, 512)    0           conv5_block2_1_bn[0][0]
__________________________________________________________________________________________________
conv5_block2_2_conv (Conv2D)    (None, 7, 7, 512)    2359808     conv5_block2_1_relu[0][0]
__________________________________________________________________________________________________
conv5_block2_2_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block2_2_conv[0][0]
__________________________________________________________________________________________________
conv5_block2_2_relu (Activation (None, 7, 7, 512)    0           conv5_block2_2_bn[0][0]
__________________________________________________________________________________________________
conv5_block2_3_conv (Conv2D)    (None, 7, 7, 2048)   1050624     conv5_block2_2_relu[0][0]
__________________________________________________________________________________________________
conv5_block2_3_bn (BatchNormali (None, 7, 7, 2048)   8192        conv5_block2_3_conv[0][0]
__________________________________________________________________________________________________
conv5_block2_add (Add)          (None, 7, 7, 2048)   0           conv5_block1_out[0][0]
                                                                 conv5_block2_3_bn[0][0]
__________________________________________________________________________________________________
conv5_block2_out (Activation)   (None, 7, 7, 2048)   0           conv5_block2_add[0][0]
__________________________________________________________________________________________________
conv5_block3_1_conv (Conv2D)    (None, 7, 7, 512)    1049088     conv5_block2_out[0][0]
__________________________________________________________________________________________________
conv5_block3_1_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block3_1_conv[0][0]
__________________________________________________________________________________________________
conv5_block3_1_relu (Activation (None, 7, 7, 512)    0           conv5_block3_1_bn[0][0]
__________________________________________________________________________________________________
conv5_block3_2_conv (Conv2D)    (None, 7, 7, 512)    2359808     conv5_block3_1_relu[0][0]
__________________________________________________________________________________________________
conv5_block3_2_bn (BatchNormali (None, 7, 7, 512)    2048        conv5_block3_2_conv[0][0]
__________________________________________________________________________________________________
conv5_block3_2_relu (Activation (None, 7, 7, 512)    0           conv5_block3_2_bn[0][0]
__________________________________________________________________________________________________
conv5_block3_3_conv (Conv2D)    (None, 7, 7, 2048)   1050624     conv5_block3_2_relu[0][0]
```

```
__________________________________________________________________________________________________
conv5_block3_3_bn (BatchNormali (None, 7, 7, 2048)   8192        conv5_block3_3_conv[0][0]
__________________________________________________________________________________________________
conv5_block3_add (Add)          (None, 7, 7, 2048)   0           conv5_block2_out[0][0]
                                                                 conv5_block3_3_bn[0][0]
__________________________________________________________________________________________________
conv5_block3_out (Activation)   (None, 7, 7, 2048)   0           conv5_block3_add[0][0]
__________________________________________________________________________________________________
avg_pool (GlobalAveragePooling2 (None, 2048)         0           conv5_block3_out[0][0]
==================================================================================================
Total params: 23,587,712
Trainable params: 23,534,592
Non-trainable params: 53,120
__________________________________________________________________________________________________
```

Tendo definido o modelo da rede, o próximo passo será pegar as camadas do modelo previamente treinado e congelá-las para evitar destruir qualquer informação que eles contenham durante as próximas rodadas de treinamento. Na sequência, vamos adicionar uma nova camada no final da rede para fazer o ajuste ao número de classes, que no nosso trabalho, são 10, e utilizar a função de ativação *Softmax*.

```
for layer in model.layers:
    layer.trainable = False

full_model = tf.keras.Sequential([model, tf.keras.layers.Dense
      (10, activation='softmax')])

full_model.summary()
```

Observe que para este novo modelo definido com o nome de *full_model* apenas 20.490 parâmetros serão treinados.

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
model (Functional)           (None, 2048)              23587712
_________________________________________________________________
dense_5 (Dense)              (None, 10)                20490
=================================================================
Total params: 23,608,202
Trainable params: 20,490
Non-trainable params: 23,587,712
_________________________________________________________________
```

Criamos algumas ações que serão necessárias durante o processo de treinamento, uma delas é o *checkpoint_callback*, que deverá salvar os pesos (em um arquivo nomeado como *best_model.hdf5*) para que os mesmos possam ser carregados posteriormente para continuar o treinamento a partir do estado salvo. Uma outra ação é o *Early Stopping*, que

deverá monitorar a variável *val_loss* e, ao detectar uma sequência de 3 valores constantes, o treinamento será finalizado.

```
from keras.callbacks import ModelCheckpoint, EarlyStopping
checkpoint_callback = ModelCheckpoint('best_model.hdf5',
      save_best_only=True, monitor='val_loss', mode='min')
es = EarlyStopping(monitor='val_loss', patience=3)
```

Na sequência, vamos definir o otimizador. Escolhemos o SGD, definimos a taxa de aprendizado = 0.01, também escolhemos um momentum = 0.9

```
sgd = tf.keras.optimizers.SGD(learning_rate=0.01,
      momentum=0.9, nesterov=True)
```

Depois de criado o modelo, é necessário compilá-lo. Neste processo, é informamos qual será a função de custo (categorical_crossentropy), o otimizador(sgd, definido no passo anterior), e a métrica de treino/validação (*categorical_accuracy*), que é a acurácia balanceada.

```
full_model.compile(loss='categorical_crossentropy',
      optimizer=sgd, metrics=['categorical_accuracy'])

history_tr = full_model.fit(train_generator,epochs=10,
                  class_weight=class_weights,
                  validation_data=validation_generator,
                  callbacks=[es, checkpoint_callback])
```

À medida que o treinamento é executado, é possível acompanhar os resultados da execução do modelo para cada época. O tempo de execução pode variar, dependendo dos recursos que estão sendo utilizados, ou das operações que estão sendo feitas pelo algoritmo.

```
Epoch 1/10
556/556 [==============================] - 304s 511ms/step - loss: 0.1241 -
        categorical_accuracy: 0.9679 - val_loss: 0.9350 - val_categorical_accuracy: 0.7243
Epoch 2/10
556/556 [==============================] - 284s 510ms/step - loss: 0.0061 -
        categorical_accuracy: 0.9981 - val_loss: 0.8090 - val_categorical_accuracy: 0.7781
Epoch 3/10
556/556 [==============================] - 282s 508ms/step - loss: 0.0057 -
        categorical_accuracy: 0.9984 - val_loss: 1.1485 - val_categorical_accuracy: 0.7215
```

Após o processo de treinamento e validação ser completado, é hora de avaliar os resultados. Primeiramente, vamos gerar a matriz de confusão dos dados de validação, utilizando os comandos:

```
from sklearn import metrics
from sklearn.metrics import confusion_matrix
import numpy as np

true_labels = validation_generator.classes
predictions = full_model.predict(validation_generator)
y_true = true_labels
y_pred = np.argmax(predictions, axis=1)
results =  metrics.confusion_matrix(y_true, y_pred,
      normalize="true").round(3)
balanceada_validacao = metrics.balanced_accuracy_score
      (y_true, y_pred).round(3)

print("Matriz de confusão dos dados de Validação:
      \n", results, "\n")
print("Acurácia Balanceada de Validação:", balanceada_validacao)
```

O resultado exibido pelo algoritmo é:

```
Matriz de confusão dos dados de Validação:

[0.68  0.026 0.    0.017 0.    0.004 0.002 0.067 0.107 0.098]
[0.002 0.913 0.034 0.018 0.    0.    0.    0.    0.014 0.018]
[0.    0.    0.926 0.    0.    0.    0.    0.037 0.037 0.   ]
[0.111 0.009 0.    0.852 0.    0.    0.    0.015 0.013 0.   ]
[0.041 0.    0.004 0.015 0.839 0.    0.004 0.002 0.088 0.006]
[0.016 0.002 0.    0.039 0.    0.94  0.    0.002 0.    0.   ]
[0.002 0.05  0.006 0.002 0.    0.    0.87  0.004 0.065 0.   ]
[0.    0.002 0.    0.    0.    0.    0.    0.995 0.002 0.   ]
[0.    0.03  0.037 0.    0.018 0.    0.027 0.098 0.789 0.   ]
[0.139 0.035 0.013 0.002 0.002 0.035 0.007 0.159 0.157 0.451]

Acurácia Balanceada de Validação: 0.826
```

Os valores verdadeiros (rótulos) de cada classe estão armazenados na variável *y_true*. Já Os valores preditos (calculados pelo algoritmo e depois comparados com os valores verdadeiros) estão armazenados na variável *y_predictions*. E a variável *y_pred* armazena a classe de maior probabilidade de ser a verdadeira. Também é possível plotar um mapa de calor com os dados da matriz de confusão e verificar os resultados graficamente:

```
import seaborn as sn
import pandas as pd
import matplotlib.pyplot as plt
array =results
df_cm = pd.DataFrame(array, range(10), range(10))
plt.figure(figsize=(10,7))
sn.heatmap(df_cm, annot=False, annot_kws={"size": 16})
plt.show()
```

Agora que sabemos o desempenho de nosso algoritmo em termos de treino e validação, é hora de verificar os resultados junto aos dados de testes. Para conhecermos a capacidade de generalização do nosso algoritmo, temos que submeter os resultados na plataforma *Kaggle*, que irá nos informar o valor da perda logarítmica calculada pela plataforma. Estes valores não podem ser calculados pelo algoritmo, uma vez que trata-se de uma competição e, portanto, não sabemos os rótulos das classes verdadeiras dos dados de testes. Os dados submetidos ao *Kaggle* devem ser formatados em uma planilha fornecida por eles, a qual fazemos o *download* em máquina local, subimos na pasta */content/dataset/* do *Google Colab* e depois carregamos no *python* com o seguinte código:

```
import pandas as pd
submission = pd.read_csv('/content/dataset/
      sample_submission.csv')
submission
```

A planilha para submissão dos dados de testes tem a seguinte formatação:

	img	c0	c1	c2	c3	c4	c5	c6	c7	c8	c9
0	img_001.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
1	img_010.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
2	img_100.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
...	...	...	...	...	...	...	...	...	...	...	...
79721	img_99994.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
79722	img_99995.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
79723	img_99996.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
79724	img_99998.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
79725	img_99999.jpg	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1

Agora, precisamos submeter as imagens de testes ao algoritmo e fazer a previsão para cada uma delas, em seguida, popular a planilha que iremos submeter ao *Kaggle*. Este processamento levou em torno de 12 minutos.

```
from tqdm import tqdm
import cv2
from keras.applications.resnet import preprocess_input

preds_test = []
contador = 0
maximo = 64
X = np.zeros((maximo, 224, 224, 3), dtype=np.uint8)

for im in tqdm(imgs_test):
    X[contador] = cv2.resize(cv2.imread('/content/dataset/
          imgs/test/%s' % im), (224, 224))

    contador += 1
```

```
        if contador == maximo:
            contador = 0
            X = tf.keras.applications.resnet50.preprocess_input(X)
            prediction = full_model.predict(X)
            for each_pred in prediction:
                preds_test.append(each_pred)
            X = np.zeros((maximo, 224, 224, 3), dtype=np.uint8)

    prediction = full_model.predict(X[:contador])

    for each_pred in prediction:
        preds_test.append(each_pred)

    ## criação de uma matriz com os resultados:
    sub = []
    for i in range(len(imgs_test)):
        aux = [imgs_test[i]]
        for j in preds_test[i]:
            aux.append(j)
        sub.append(aux)
```

Com a matriz de resultados das previsões dos dados de testes, vamos converter esta matriz em um *dataframe* e salvar este *dataframe* em um arquivo de extensão *.csv*.

```
    submission = pd.DataFrame(data=sub,
          columns=submission.columns, index=None)

    submission.to_csv('/content/Resnet_rev1_descon_shear.csv',
          index=False)
```

Por último, basta pegar o arquivo *Resnet_rev1_descon_shear.csv* na pasta *content* do *Google Colab* e fazer o *upload* do mesmo na plataforma *Kaggle:* no link: *https://www.kaggle.com/c/state-farm-distracted-driver-detection/submit*