

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CAMPUS DE SÃO JOÃO DA BOA VISTA

THALES PIMENTEL ZUANAZZI

Aprendizado Profundo para a Segmentação de Regiões Teciduais em Cirurgias Minimamente Invasivas

São João da Boa Vista
2024

Thales Pimentel Zuanazzi

Aprendizado Profundo para a Segmentação de Regiões Teciduais em Cirurgias Minimamente Invasivas

Trabalho de Graduação apresentado ao Conselho de Curso de Graduação em Engenharia Eletrônica e de Telecomunicações do Campus de São João da Boa Vista, Universidade Estadual Paulista, como parte dos requisitos para obtenção do diploma de Graduação em Engenharia Eletrônica e de Telecomunicações .

Orientador: Profº Dr. Marlon Rodrigues Garcia

São João da Boa Vista

2024

Z93a

Zuanazzi, Thales Pimentel

APRENDIZADO PROFUNDO PARA A SEGMENTAÇÃO DE
REGIÕES TECIDUAIS EM CIRURGIAS MINIMAMENTE INVASIVAS /

Thales Pimentel Zuanazzi. -- São João da Boa Vista, 2024

55 p. : il., tabs., fotos

Trabalho de conclusão de curso (Bacharelado - Engenharia de
Telecomunicações) - Universidade Estadual Paulista (UNESP), Faculdade de
Engenharia, São João da Boa Vista

Orientador: Marlon Rodrigues Garcia

1. Inteligência artificial. 2. Python (Linguagem de programação de
computador). 3. Visão por computador. 4. Laparoscopia. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Universidade Estadual Paulista (UNESP), Faculdade de Engenharia, São João da Boa Vista. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

**UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”
FACULDADE DE ENGENHARIA - CÂMPUS DE SÃO JOÃO DA BOA VISTA
GRADUAÇÃO EM ENGENHARIA ELETRÔNICA E DE TELECOMUNICAÇÕES**

TRABALHO DE CONCLUSÃO DE CURSO

**APRENDIZADO PROFUNDO PARA A SEGMENTAÇÃO DE REGIÕES TECIDUAIS
EM CIRURGIAS MINIMAMENTE INVASIVAS**

Aluno: Thales Pimentel Zuanazzi
Orientador: Prof. Dr. Marlon Rodrigues Garcia

Banca Examinadora:

- Marlon Rodrigues Garcia (Orientador)
- Gabriel Pereira Gouveia da Silva (Examinador)
- Lucas Daniel Del Rosso Calache (Examinador)

A ata da defesa com as respectivas assinaturas dos membros encontra-se no prontuário do aluno (Processo nº 28/2024)

São João da Boa Vista, 12 de julho de 2024

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
CAMPUS DE SÃO JOÃO DA BOA VISTA

THALES PIMENTEL ZUANAZZI

**ESTE TRABALHO DE GRADUAÇÃO FOI JULGADO ADEQUADO COMO PARTE DO
REQUISITO PARA A OBTENÇÃO DO DIPLOMA DE "GRADUANDO EM ENGENHARIA
ELETRÔNICA E DE TELECOMUNICAÇÕES "**

**APROVADO EM SUA FORMA FINAL PELO CONSELHO DE CURSO DE GRADUAÇÃO EM
ENGENHARIA ELETRÔNICA E DE TELECOMUNICAÇÕES**

Profº Dr. RAFAEL ABRANTES PENCHEL
Coordenador

BANCA EXAMINADORA:

Profº Dr. Marlon Rodrigues Garcia
Orientador/UNESP-SJBV

Profº Dr. Lucas Daniel Del Rosso Calache
UNESP-SJBV

Profº Dr. Gabriel Pereira Gouveia da Silva
UNESP-SJBV

Julho , 2024

DADOS CURRICULARES

THALES PIMENTEL ZUANAZZI

NASCIMENTO 22/06/1999 - Sorocaba / SP

FILIAÇÃO Wladimir Zuanazzi
Adriana Katia Pimentel Zuanazzi

2017 / 2024 Engenharia Eletrônica e Telecomunica-
ções
UNESP

Dedico este trabalho, primeiramente, à minha família, em especial aos meus pais, que sempre me apoiaram incondicionalmente, proporcionando-me a base e o suporte emocional para alcançar meus objetivos. A vocês, agradeço por todo o amor, paciência e incentivo ao longo de toda minha trajetória acadêmica.

Aos meus amigos, que compartilharam momentos de alegria e desafios, oferecendo apoio e compreensão em todas as fases do curso.

Ao meu orientador, cuja orientação e conhecimento foram fundamentais para a realização deste trabalho. Sua dedicação e disponibilidade foram imprescindíveis para que eu pudesse concluir este projeto com êxito.

E, por fim, a todos aqueles que, de alguma forma, contribuíram para a realização deste trabalho. Agradeço pela inspiração e pelo apoio constante.

“Se você acha que é possível ter uma vida perfeita, viverá em eterna frustração. Altos e baixos, alegria e tristeza, entusiasmo e decepção são partes integrantes da nossa existência. Lute sempre para melhorar e alegre-se com suas conquistas. "Muitas pessoas devem a grandeza de suas vidas aos problemas que tiveram de vencer.”
(Lord Baden-Powell)

RESUMO

Este trabalho de conclusão de curso investigou o uso de aprendizado profundo para a segmentação de imagens em cirurgias laparoscópicas, com o objetivo de desenvolver e aplicar de maneira prática técnicas de visão computacional e Inteligência Artificial em cirurgias de laparoscopia. A pesquisa abordou a aplicação de redes neurais, especificamente a combinação de Redes Neurais Residuais (ResNet) e Autoencoders Convolucionais em Formato de U (U-Net), resultando na utilização da UResNet, uma arquitetura híbrida adaptada para segmentação das imagens em estudo. O estudo utilizou o Dresden Surgical Anatomy Dataset (DSAD), composto por mais de 13.000 imagens segmentadas de cirurgias robóticas assistidas. Ferramentas como a linguagem Python e as bibliotecas PyTorch, Torchmetrics, Torchvision, NumPy, Pandas, Matplotlib, PIL e TQDM foram essenciais para o desenvolvimento e treinamento dos modelos. O Google Colaboratory (Colab) foi utilizado para o processamento devido aos recursos computacionais gratuitos oferecidos. Além disso, o Método Taguchi foi empregado para otimizar os parâmetros da rede neural. Os experimentos realizados mostraram que a UResNet foi capaz de segmentar imagens cirúrgicas com uma acurácia de cerca de 85%, comparável à pesquisa original do DSAD. Os resultados demonstraram a eficácia das técnicas e modelos utilizados, com melhorias pequenas após refinamentos e retreinamento. As redes foram testadas em várias situações, desde imagens simples até complexas presentes no banco de dados, enfrentando desafios como ruídos e variações de iluminação. A pesquisa revelou limitações de processamento que necessitaram do uso do Google Colab, com um tempo de treinamento prolongado devido aos recursos computacionais limitados. Apesar disso, a viabilidade de redes neurais profundas para segmentação de tecidos em imagens médicas foi comprovada. O trabalho destacou a importância de um tratamento mais robusto das imagens e o uso de redes mais profundas em futuras pesquisas. O trabalho atingiu seus objetivos técnicos e acadêmicos, aprofundando o entendimento de ferramentas e técnicas avançadas de aprendizado profundo aplicadas à cirurgia laparoscópica. A tecnologia de segmentação de imagens médicas está em evolução e tem o potencial de se tornar uma ferramenta comum no cotidiano médico, auxiliando na identificação de tecidos e ferramentas cirúrgicas, contribuindo para procedimentos mais precisos e seguros.

PALAVRAS-CHAVE: regiões anatômicas; uresnet; python; torch; Laparoscopia; Segmentação.

ABSTRACT

This work investigated the use of deep learning for image segmentation in laparoscopic surgeries, aiming to assist in the precise identification of anatomical structures during surgical procedures. The research focused on the application of neural networks, specifically the combination of Residual Neural Networks (ResNet) and U-Shaped Convolution Autoencoder (U-Net), resulting in the utilization of UResNet, a hybrid architecture adapted for the segmentation of the studied images. The study utilized the Dresden Surgical Anatomy Dataset (DSAD), composed of over 13,000 segmented images from assisted robotic surgeries. Tools such as Python language and the libraries PyTorch, Torchmetrics, Torchvision, NumPy, Pandas, Matplotlib, PIL, and TQDM were essential for the development and training of the models. Google Colaboratory (Colab) was used for processing due to the free computational resources offered. Additionally, the Taguchi Method was employed to efficiently optimize the neural network parameters. The experiments demonstrated that UResNet was capable of segmenting surgical images with an accuracy of about 85%, comparable to the original DSAD research. The results showed the effectiveness of the techniques and models used, with minor improvements following refinements and retraining. The networks were tested in various situations, from simple to complex images present in the dataset, facing challenges such as noise and lighting variations. The research revealed processing limitations that necessitated the use of Google Colab, with prolonged training times due to limited computational resources. Despite this, the feasibility of deep neural networks for tissue segmentation in medical images was proven. The study highlighted the importance of more robust image processing and the use of deeper networks in future research. The work achieved its technical and academic objectives, deepening the understanding of advanced deep learning tools and techniques applied to laparoscopic surgery. The technology of medical image segmentation is evolving and has the potential to become a common tool in everyday medical practice, assisting in the identification of tissues and surgical tools, contributing to more precise and safer procedures.

KEYWORDS: anatomical regions; uresnet; python; torch; laparoscopy; segmentation.

LISTA DE ILUSTRAÇÕES

Figura 1	Taxa de aprendizado e suas alterações na convergencia da rede neural.	17
Figura 2	Comparação entre (a) os blocos residuais e (b) os blocos convencionais de redes neurais convolucionais.	19
Figura 3	Autoencoder em formato U-Net, composto por 4 camadas de contração/expansão e 1 camada de gargalo, com conexões entres as camadas de contração e expansão, permitindo à rede neural abstrair a imagem e, em seguida, reconstruí-la mantendo detalhes.	20
Figura 4	Camada MaxPooling e AveragePooling, a camada MaxPooling reduz o tamanho da imagens utilizando o maior valor dentre os pixels a serem reduzidos	21
Figura 5	Fluxograma dos Algoritmos e onde cada um é subsequente utilizado.	27
Figura 6	Curvas de acurácia e pontuação dice do treinamento 1, com 8 imagens no lote, taxa de aprendizado de $5e-4$, UResNet de 34 camadas e função de perda CrossEntropyLoss.	42
Figura 7	Curvas de acurácia e pontuação dice do treinamento 2, com 8 imagens no lote, taxa de aprendizado de $1e-3$, UResNet de 18 camadas e função de perda L1Loss.	43
Figura 8	Curvas de acurácia e pontuação dice do treinamento 3, com 8 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 50 camadas e função de perda personalizada.	43
Figura 9	Curvas de acurácia e pontuação dice do treinamento 4, com 6 imagens no lote, taxa de aprendizado de $5e-4$, UResNet de 18 camadas e função de perda customizada.	44
Figura 10	Curvas de acurácia e pontuação dice do treinamento 5, com 6 imagens no lote, taxa de aprendizado de $1e-3$, UResNet de 50 camadas e função de perda CrossEntropyLoss.	44
Figura 11	Curvas de acurácia e pontuação dice do treinamento 6, com 6 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 34 camadas e função de perda L1Loss.	45
Figura 12	Curvas de acurácia e pontuação dice do treinamento 7, com 12 imagens no lote, taxa de aprendizado de $5e-4$, UResNet de 50 camadas e função de perda L1Loss.	45
Figura 13	Curvas de acurácia e pontuação dice do treinamento 8, com 12 imagens no lote, taxa de aprendizado de $1e-3$, UResNet de 34 camadas e função de perda customizada.	46
Figura 14	Curvas de acurácia e pontuação dice do treinamento 9, com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 18 camadas e função de perda CrossEntropyLoss.	46
Figura 15	Fígado ajustado corretamente.	47
Figura 16	Fígado ajustado erroneamente.	47
Figura 17	Mascara do fígado da rede 9.	47

Figura 18	Mascara do fígado da rede 9.	48
Figura 19	Treinamento com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 34 camadas e função de perda CrossEntropyLoss.	49
Figura 20	Treinamento com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 50 camadas e função de perda CrossEntropyLoss.	49
Figura 21	Treinamento com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 50 camadas e função de perda L1Loss.	50
Figura 22	Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, de região tecidual fora de foco, porém com bom contraste e iluminação.	51
Figura 23	Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, de região tecidual em foco e contraste mas com iluminação imperfeita.	51
Figura 24	Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, do fígado um pouco fora de foco com irregularidade no canto superior direito e excesso de iluminação no canto inferior direito.	52
Figura 25	Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, do fígado somente no canto superior esquerdo com má iluminação e foco mas bom contraste.	52
Figura 26	Segmentação apresentada pelo modelo do Treinamento de L1Loss, de região tecidual levemente fora de foco, com bom contraste e iluminação.	53
Figura 27	Segmentação apresentada pelo modelo do Treinamento de L1Loss, de região tecidual fora de foco e com bordas embaçadas.	53
Figura 28	Segmentação apresentada pelo modelo do Treinamento de L1Loss, de região tecidual em foco mas com iluminação imperfeita e contraste ruim nos cantos.	53
Figura 29	Segmentação apresentada pelo modelo do Treinamento de L1Loss, do fígado com má iluminação e contraste.	54

LISTA DE TABELAS

Tabela 1	– Tabela Ortogonal do método Taguchi, de 4 variáveis, com 3 estados e 9 testes. . .	22
Tabela 2	– Tabela Ortogonal de 4 variáveis com 3 parâmetros por variável, mostra um possível distribuição dos parâmetros para teste.	39
Tabela 3	– Tabela Ortogonal com a distribuição das variáveis e parâmetros que foram usados para o treinamento com o método Taguchi.	40
Tabela 4	– Tabela de melhores resultados de acurácia e Dice dos treinamentos.	48
Tabela 5	– Tabela de correlação de valores das macrovariáveis com a acurácia.	48

LISTA DE ABREVIATURAS E SIGLAS

TCC	Trabalho de Conclusão de Curso
UNESP	Universidade Estadual Paulista
IA	Inteligencia Artificial
GPU	Graphics Processing Units
CPU	Central Processing Units
ResNet	Residual neural Networks
UNet	Convolutional neural Networks
UResNet	Residual Convolutional neural Networks
DSAD	Dresden Surgical Anatomy Dataset
API	Application Programming Interface

SUMÁRIO

1	INTRODUÇÃO	14
2	OBJETIVOS	15
3	FUNDAMENTAÇÃO TEÓRICA	16
3.1	Python e Torch	16
3.2	Noções Básicas de Treinamento de Redes Neurais com MNIST	17
3.3	Rede Neural Residual (ResNet) e Rede Neural Convulocional (U-Net)	18
3.3.1	Redes Neurais Residuais (ResNet)	18
3.3.2	U-Net	19
3.4	Método Taguchi, Matriz Ortogonal	21
4	METODOLOGIA	23
4.1	Conjunto de Dados	23
4.2	Bibliotecas Utilizadas	23
4.2.1	PyTorch	24
4.2.2	Torchmetrics	24
4.2.3	Torchvision	25
4.2.4	NumPy	25
4.2.5	Pandas	25
4.2.6	Bibliotecas de Apoio	26
4.3	Ambiente De Execução	26
4.4	Pré-processamento e Treinamento	27
4.4.1	Carregamento e Pré-Processamento das Imagens	27
4.4.2	Estruturação do Modelo	28
4.4.2.1	mySequential	29
4.4.2.2	block_standard	29
4.4.2.3	block_bottleneck	30
4.4.2.4	UResNet	30
4.4.3	Ferramentas e Funções Úteis	32
4.4.3.1	ToTensor	32
4.4.3.2	Rotate	32
4.4.3.3	CenterCrop e Resize	33
4.4.3.4	FlipHorizontal e FlipVertical	33
4.4.3.5	Normalize	33
4.4.3.6	Affine	34
4.4.3.7	save_checkpoint e load_checkpoint	34
4.4.3.8	check_accuracy	35

4.4.3.9	get_loaders	36
4.4.4	Treinamento, Validação e Salvamento da Rede	37
4.4.4.1	train_fn	37
4.4.4.2	CustomCrossEntropyLoss	38
4.4.4.3	main	38
4.5	Método Taguchi para Variação dos Parâmetros da Rede	39
4.6	Treinamento dos Modelos com Maiores Acurácias	41
5	RESULTADOS E DISCUSSÃO	42
5.1	Modelos com Parâmetros do Método Taguchi	42
5.2	Segundo Treinamento	48
5.2.1	Mascaras de Segmentação	50
6	CONCLUSÃO	55
	REFERÊNCIAS	56

1 INTRODUÇÃO

O sucesso de uma operação cirúrgica é diretamente influenciado pela experiência e desempenho da equipe cirúrgica (KOLBINGER et al., 2023a). Uma análise recente revelou que mais da metade dos casos com complicações pós-operatórias estavam associados a erros identificáveis, incluindo enganos na identificação de anatomia (KOLBINGER et al., 2022).

Neste contexto, a visão computacional poderá desempenhar um papel importante, envolvendo a análise automatizada de imagens digitais com o objetivo de auxiliar e aumentar as capacidades visuais humanas (MAIER-HEIN et al., 2022). Esta abordagem poderá ser transformadora na medicina, com aplicações bem-sucedidas como a identificação de pintas e melanomas, retinopatia diabética, risco cardiovascular e encaminhamentos a partir de imagens de fundo de olho e tomografia de coerência óptica (OCT), detecção de lesões mamárias em mamografias e análise da coluna vertebral com ressonância magnética. Um único modelo de aprendizado profundo até mesmo demonstrou ser eficaz no diagnóstico em várias modalidades médicas (por exemplo, radiologia e oftalmologia) (ESTEVA et al., 2019).

Com a laparoscopia consolidada como padrão-ouro para diversos procedimentos cirúrgicos, e o aumento da capacidade computacional, esses princípios têm sido gradualmente aplicados à cirurgia abdominal (MASCAGNI; ALAPATT; SESTINI, 2022). Uma parte das pesquisas em visão computacional cirúrgica concentrou-se na classificação de imagens quanto à presença de instrumentos cirúrgicos ou estruturas anatômicas (KOLBINGER et al., 2023a), além da avaliação da proficiência cirúrgica com base em procedimentos registrados (KOLBINGER et al., 2022). Até o momento, no entanto, nenhum modelo de Inteligência Artificial (IA) baseado em imagens cirúrgicas é utilizado atualmente fora de pesquisa com benefícios clínicos diretos (HARDY et al., 2021).

A utilização da IA em procedimentos cirúrgicos tem o potencial de simplificar procedimentos complicados, melhorando os resultados para os procedimentos no campo médico. A IA poderá auxiliar na identificação precisa de estruturas anatômicas e na tomada de decisões durante as cirurgias (CHOKSI et al., 2023). O treinamento de redes neurais para segmentação de imagens cirúrgicas poderá ser promissor na melhoria da assistência aos cirurgiões, fornecendo informações visuais detalhadas e precisas. Estes avanços representam um passo significativo na integração da IA na prática cirúrgica, podemos ser capazes de fornecer cuidados cirúrgicos eficientes e homogêneos aos pacientes, garantindo segurança (KITAGUCHI; ITO, 2024).

Ao explorar o potencial da IA na cirurgia, este trabalho busca o desenvolvimento de redes neurais artificiais, focadas na segmentação precisa de estruturas anatômicas, aumentando a capacidade de cirurgiões menos experientes em operações mais complexas. Essa abordagem inovadora tem o potencial de enriquecer a experiência tanto dos pacientes quanto dos profissionais de saúde, demonstrando o impacto transformador da IA na cirurgia moderna e abrindo caminho para avanços significativos na medicina.

2 OBJETIVOS

Neste trabalho, propusemos realizar uma investigação focada no desenvolvimento e aplicação prática de técnicas de visão computacional e Inteligência Artificial na cirurgia laparoscópica. Este objetivo é fundamentado na necessidade de avançar nas técnicas de análise e processamento de imagens cirúrgicas, proporcionando um suporte mais preciso e eficiente aos procedimentos laparoscópicos. Com o aumento da capacidade computacional e a evolução contínua da IA, o trabalho visa aplicar algoritmos de segmentação de imagem em regiões anatômicas adquiridas durante cirurgias laparoscópicas robóticas. Os objetivos específicos incluem:

1. Desenvolver as ferramentas e habilidades necessárias em Python, uma linguagem de programação fundamental no campo da ciência de dados e aprendizado de máquina.
2. Desenvolver os conhecimentos necessários para a utilização de bibliotecas, como PyTorch e Torchvision, essenciais para o treinamento e implementação de redes neurais.
3. Compreender o funcionamento e treinamento de Redes Neurais, com foco na criação de uma rede neural específica para segmentação.
4. Entender os desafios atuais para uma Rede de segmentação, explorando os aspectos práticos e teóricos do treinamento de redes neurais aplicadas à cirurgia laparoscópica.

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Python e Torch

Python é uma linguagem de programação de alto nível, conhecida por sua simplicidade e legibilidade, características que a tornam particularmente atraente para iniciantes e profissionais em diversos campos, incluindo ciência de dados, aprendizado de máquina e desenvolvimento web (PEDREGOSA et al., 2011). Sua sintaxe clara e estrutura orientada a objetos permite aos desenvolvedores escreverem códigos limpos e bem estruturados, facilitando a manutenção e a colaboração em projetos complexos. Além disso, Python é uma linguagem interpretada, o que significa que os códigos são executados diretamente, sem a necessidade de compilação prévia, agilizando o ciclo de desenvolvimento e teste (OLIPHANT, 2007).

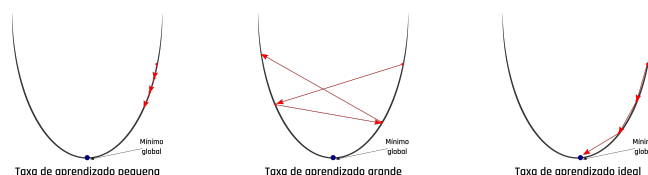
A biblioteca PyTorch, uma das principais ferramentas utilizadas neste trabalho, é um ambiente completo para o desenvolvimento, treinamento de modelos de aprendizado profundo, um tipo de aprendizado de máquina utilizado em aplicações como reconhecimento de imagens e processamento de linguagem. Escrito em Python, é relativamente fácil para a maioria dos desenvolvedores de aprendizado de máquina aprender e usar. PyTorch se destaca pelo seu excelente suporte para GPUs e pelo uso de auto-diferenciação em modo reverso, que permite que os grafos de computação sejam modificados dinamicamente. Isso o torna uma escolha popular para experimentação rápida e prototipagem (PYTORCH, 2024).

O ambiente, chamado de `torch` no código, contém estruturas de dados para tensores multidimensionais e define operações matemáticas sobre esses tensores. Além disso, ele forneceu muitas utilidades para serialização eficiente de tensores e tipos arbitrários, e outras utilidades úteis, funciona como o principal fundamento das outras bibliotecas do ambiente.

Torchvision, uma extensão da biblioteca PyTorch, foi outra ferramenta crucial utilizada neste projeto. Esta biblioteca ofereceu conjuntos de dados populares, arquiteturas de modelos, além de transformações comumente usadas em visão computacional. O torchvision simplifica o processo de carregamento e pré-processamento de imagens, uma etapa fundamental no treinamento de redes neurais para tarefas de visão computacional. Além disso, essa ferramenta também fornece acesso a modelos pré-treinados, que podem ser extremamente úteis para tarefas de transferência de aprendizado, onde um modelo treinado em um grande conjunto de dados é ajustado para uma tarefa específica.

A combinação de Python e Torchvision com a simplicidade e a velocidade de desenvolvimento do Python representam uma combinação valiosa para o desenvolvimento de redes neurais. Python serve como a base, oferecendo uma linguagem de programação robusta e versátil, enquanto o Torch e o torchvision fornecem as funcionalidades específicas necessárias para o treinamento eficiente e a implementação de modelos de aprendizado de máquina. Esta combinação foi essencial para a execução deste projeto, permitindo a criação de redes neurais de segmentação eficazes, além da exploração dos desafios atuais enfrentados por redes de segmentação.

Figura 1 – Taxa de aprendizado e suas alterações na convergência da rede neural.



fonte: (ALMEIDA FELIPE CARVALHO, 2024).

3.2 Noções Básicas de Treinamento de Redes Neurais com MNIST

Para compreender esses conceitos em profundidade, iniciou-se com o treinamento de uma rede utilizando o conjunto de dados MNIST, banco de dados de manuscritos.

Essa abordagem, realizada com o auxílio da linguagem Python e da biblioteca PyTorch, permitiu uma maior compreensão de como cada parâmetro influencia no treinamento de uma rede neural, além de vislumbrar seus impactos nas métricas de treinamento.

No treinamento de um modelo de inteligência artificial, especialmente em redes neurais artificiais, diversos hiperparâmetros são ajustados para controlar o processo de aprendizagem. Os parâmetros de ajuste são essenciais para guiar o modelo em direção à solução de maneira eficaz e eficiente.

O primeiro hiperparâmetro a ser estudado foi a taxa de aprendizado, que é um dos mais importantes. Esse parâmetro controla o tamanho dos passos que o modelo dá no espaço de soluções, a cada vez que aprende durante o processo de aprendizagem. Uma taxa de aprendizado muito alta pode fazer com que o modelo aprenda muito rápido, ultrapassando o ponto ótimo. Já uma taxa muito baixa pode resultar em um processo de treinamento lento e, possivelmente, pode prender o modelo em mínimos locais. Essas diferenças podem ser vistas na Figura 1. A escolha da taxa de aprendizado pode variar significativamente dependendo do problema e do tipo de otimizador utilizado, e muitas vezes é ajustada ao longo do treinamento.

O tamanho do lote, ou *batch size*, é outro macroparâmetro crítico. Ele determina o número de amostras de dados que serão propagadas pelo modelo de inteligência artificial antes de uma atualização dos pesos. Um tamanho de lote grande pode levar a uma estimativa mais estável do gradiente, mas também pode aumentar os requisitos de memória e o tempo de processamento. Por outro lado, um tamanho de lote muito pequeno pode tornar o treinamento mais ruidoso ou instável, embora possa ajudar o modelo a escapar de mínimos locais.

O número de épocas define quantas vezes o conjunto de dados completo passará pelo modelo durante o treinamento. Um número insuficiente de épocas pode resultar no sub-treinamento do modelo, enquanto um número excessivo pode levar ao sobreajuste, onde o modelo aprende padrões específicos do conjunto de treinamento, que não generalizam bem para dados não vistos.

A regularização é uma técnica para prevenir o sobreajuste, adicionando um termo de penalidade à função de perda ou modificando a arquitetura da modelo. O parâmetro de regularização controla a força dessa penalidade. Valores maiores impõem uma regularização mais forte, o que pode prevenir o sobreajuste, mas também pode levar a um modelo sub-treinado se a regularização for excessiva.

O momento é outro parâmetro de ajuste utilizado em certos otimizadores, como o SGD (do termo

em inglês, *Stochastic Gradient Descent*) com momento. Esse parâmetro ajuda a acelerar o treinamento, especialmente em direções consistentes, e a suavizar as oscilações do problema de otimização. Um valor de momento alto pode levar a uma convergência mais rápida, mas também pode causar a ultrapassagem do mínimo.

A inicialização dos pesos é outro aspecto crítico, embora não seja um parâmetro que se ajusta durante o treinamento. A forma como os pesos iniciais são definidos pode afetar significativamente a rapidez e a qualidade da convergência do modelo. Porém, inicializações inadequadas podem levar a problemas como o desaparecimento ou a explosão dos gradientes.

Além desses, existem muitos outros hiperparâmetros específicos para diferentes tipos de modelos e técnicas de treinamento, como a taxa de decaimento do peso, taxa de abandono em camadas de *dropout* (exclusão), e os parâmetros específicos para otimizadores avançados, como Adam ou RMSprop. Todos esses hiperparâmetros podem ser otimizados para melhorar o treinamento e as previsões do modelo.

Ajustar esses hiperparâmetros depende, muitas vezes, de experimentação e da experiência do desenvolvedor. O uso de técnicas de busca de hiperparâmetros pode ajudar a encontrar um conjunto de hiperparâmetros bem ajustados para um dado modelo de IA, como a validação cruzada, a busca em grade, ou métodos baseados em otimização bayesiana.

3.3 Rede Neural Residual (ResNet) e Rede Neural Convulocional (U-Net)

3.3.1 Redes Neurais Residuais (ResNet)

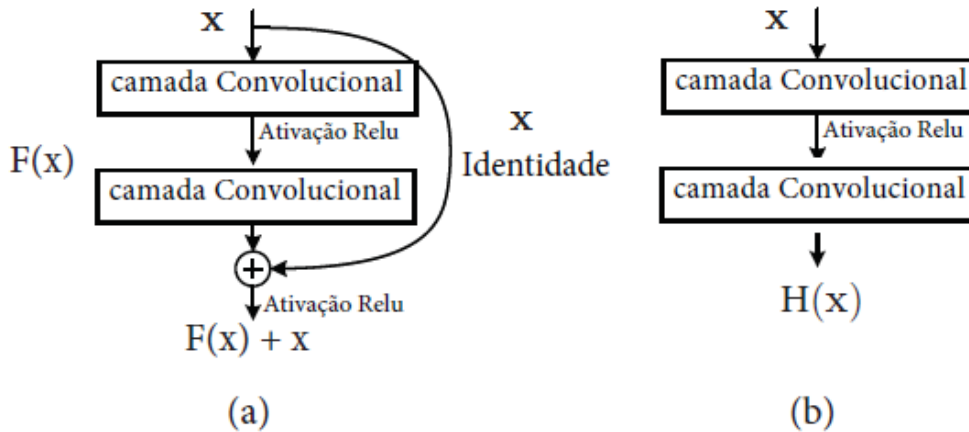
As Redes Neurais Residuais, ou ResNet (do inglês *Residual Network*), foram introduzidas por pesquisadores da Microsoft Research em 2015 (HE et al., 2015). Elas foram projetadas para facilitar o treinamento de redes neurais muito profundas, que anteriormente eram difíceis de serem treinadas, devido ao problema de degradação de modelo em redes muito profundas.

O problema de degradação de modelo consiste na diminuição da taxa de acerto para o conjunto de treino, conforme se aumenta a profundidade da redes neurais artificiais (HE et al., 2015). Esse fenômeno é diferente do sobreajuste, em que a taxa de acerto diminui para o conjunto de validação, e adicionar mais camadas leva a erros de treino ainda maiores.

Para a realização do mapeamento residual foi feita a introdução de conexões residuais ou atalhos, como se pode ver na Figura 2. Essas conexões residuais permitiram que a entrada original de uma camada fosse passada pela própria camada e n camadas subsequentes, facilitando a passagem de informação através de várias camadas sem ser significativamente alterada. Isso foi realizado por meio dos chamados blocos residuais.

A ideia da ResNet foi modificar como o mapeamento de características era feito. Para tanto, os autores reformularam o processo de aprendizagem, de maneira a ajustar o mapeamento residual (Figura 2a), em vez do mapeamento bruto (Figura 2b). Como se pode ver pela Figura 2, a transformação

Figura 2 – Comparação entre (a) os blocos residuais e (b) os blocos convencionais de redes neurais convolucionais.



fonte: editada de (HE et al., 2015).

total feita pelo bloco da ResNet é $\mathcal{F}(x) + x$, que é mais fácil de se otimizar do que somente $\mathcal{H}(x)$ (HE et al., 2015). A presença de x como referência estabiliza o processo de treinamento, prevenindo que o modelo se perca durante a otimização. Podemos ver a ResNet, de maneira alegórica, como a série de Taylor das redes neurais. Essa abordagem facilita a aprendizagem e a propagação dos gradientes (HE et al., 2015).

Os blocos residuais podem ser construídos no modelo de inteligência artificial a partir de camadas mais simples. A ideia é que cada bloco seja composto por uma entrada que será passada pelo bloco e copiada para o fim do bloco (atalho). Essa entrada então passa por n camadas convolucionais. No fim das n camadas, a saída é somada elemento a elemento com a entrada. Esse bloco base pode ser repetido quantas vezes for desejado para gerar redes profundas.

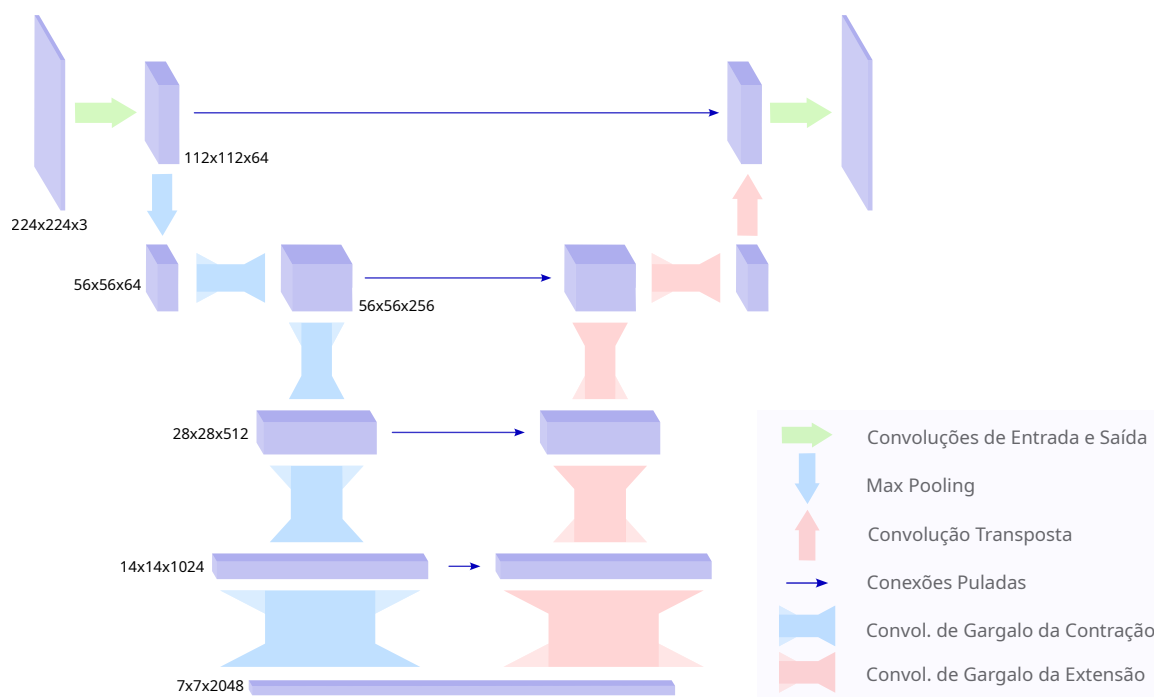
3.3.2 U-Net

A U-Net é uma arquitetura de rede neural projetada para tarefas de segmentação de imagens, com foco originalmente em imagens biomédicas, onde o objetivo é classificar cada pixel de uma imagem em uma categoria. Desenvolvida por Olaf Ronneberger e colegas em 2015 (TOMAR, 2021), a U-Net é notável por sua eficiência em trabalhar com um número limitado de amostras de treinamento e por sua capacidade de produzir segmentações precisas.

A U-Net possui uma arquitetura de *autoencoder*, que aprende duas funções: uma função de codificação que transforma os dados de entrada e uma função de decodificação que recria os dados de entrada a partir da representação codificada. A arquitetura da U-Net é assim chamada devido à sua forma que lembra a letra “U”, conforme mostra o diagrama esquemático da Figura 3.

Essa arquitetura consiste em duas partes distintas: uma parte de contração (representada em azul na Figura 3) e uma parte de expansão (representada em vermelho na Figura 3). Durante a etapa de contração, parte dos tensores são salvos para serem realimentados na etapa de expansão, conforme

Figura 3 – Autoencoder em formato U-Net, composto por 4 camadas de contração/expansão e 1 camada de gargalo, com conexões entre as camadas de contração e expansão, permitindo à rede neural abstrair a imagem e, em seguida, reconstruí-la mantendo detalhes.



fonte: Produção do próprio autor.

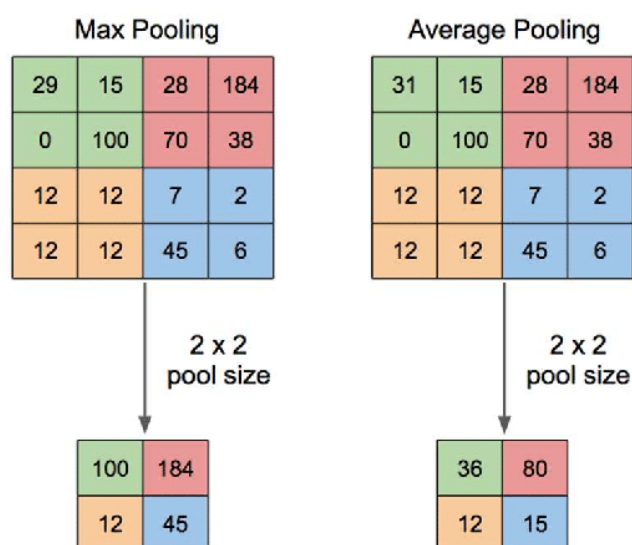
ilustrado pelas setas azuis na Figura 3. Essas partes podem ser simplificadas em blocos, sendo o menor bloco composto por camadas convolucionais aplicadas à imagem. Essas convoluções tratam a imagem para extrair os principais traços, podendo multiplicar ou dividir os canais da imagem. A última parte do bloco é uma camada de ativação ReLU (Rectified Linear Unit), que decide se o valor do neurônio é válido ou não, inserindo uma não linearidade para resolver o problema de desaparecimento de gradiente.

A parte de contração é um bloco de convolução que multiplicará os canais da imagem, precedido por uma camada de `max pooling`, ela funciona como mostrado na Figura 4, resultando em uma redução do tamanho da imagem. O objetivo da contração foi reduzir o tamanho da imagem e em troca capturar mais contexto dessa imagem menor, para poder compreender cada pedaço da imagem.

A parte de expansão consiste na concatenação da saída do bloco convolucional do mesmo nível com uma camada de convolução transposta, que aumenta o tamanho da imagem para o tamanho do nível. Isso tem como objetivo aumentar a resolução da imagem enquanto reduz o número de canais, ou seja, as propriedades da imagem. A concatenação transfere propriedades já encontradas na contração para a expansão, permitindo que a rede utilize simultaneamente informações de contexto e informações espaciais detalhadas.

Com esses 3 blocos foi possível construir uma rede U-Net com quantos níveis fossem desejados, para aproveitar o poder dessa estrutura o aumento dos dados foi realizado, com técnicas de rotação, esticamento e outras multiplicando os dados e tornando o modelo de inteligência artificial mais robusto e mais capaz de segmentar os tecidos.

Figura 4 – Camada MaxPooling e AveragePooling, a camada MaxPooling reduz o tamanho da imagens utilizando o maior valor dentre os pixels a serem reduzidos



fonte: (YANI; IRAWAN; SETIANINGSIH, 2019).

3.4 Método Taguchi, Matriz Ortogonal

O método Taguchi é uma técnica de design de experimentos que se mostra simples e eficaz para aplicação em diversas situações de engenharia. Esse método é particularmente útil para identificar problemas em processos de fabricação ou para delimitar o escopo de um projeto de pesquisa usando dados já existentes. Uma das principais vantagens do método Taguchi é sua capacidade de analisar vários parâmetros sem a necessidade de um número excessivamente alto de experimentos. Por exemplo, um processo com 4 variáveis, cada uma com 3 estados, normalmente exigiria 81 experimentos para testar todas as combinações. No entanto, utilizando os arranjos ortogonais de Taguchi, apenas 9 experimentos são necessários, o que representa menos de 11,1% do número original de experimentos. (LAZIC, 2013).

Esta técnica se baseia nos seguintes conceitos:

- A regra de interação afirma que a maior parte do resultado ocorre devido a um único fator ou à combinação de dois fatores, com o peso do resultado sendo progressivamente menor para cada nova interação de três ou mais fatores.
- Testes em T-way/Testes em pares requerem um número de fatores, com cada par de fatores possuindo no mínimo um teste.
- Matriz de cobertura: é a matriz escolhida que cumpre os conceitos acima.

Uma matriz Taguchi pode ser interpretada com as linhas (séries) representando os testes a serem executados, as colunas representando os fatores a serem testados, os valores da matriz (níveis) representando os valores dos fatores a serem testados e a força, que é uma medida de quantas séries são necessárias para que os níveis apareçam igualmente no fator.

Tabela 1 – Tabela Ortogonal do método Taguchi, de 4 variáveis, com 3 estados e 9 testes.

Treinamento	A	B	C	D
1	1	1	1	1
2	1	2	2	3
3	1	3	3	2
4	2	1	2	2
5	2	2	3	1
6	2	3	1	3
7	3	1	3	3
8	3	2	1	2
9	3	3	2	1

fonte: Produção do próprio autor.

Após a utilização da tabela para os testes, os valores de cada série são anotados. Em seguida, para calcular a relevância de cada nível do fator, somam-se os resultados dos testes que possuíam aquele nível, fazendo isso para todos os fatores de interesse. Esse processo também pode ser aplicado aos pares de fatores. O valor dessa soma é comparado com os outros níveis de cada fator para identificar os níveis que apresentam melhores resultados no teste.

4 METODOLOGIA

4.1 Conjunto de Dados

O Dresden Surgical Anatomy Dataset (DSAD) compreendeu um total de 13.195 imagens distintas, todas com segmentações pixel a pixel de 11 estruturas anatômicas. Estas incluíam parede abdominal, cólon, vasos intestinais (artéria mesentérica inferior e veia mesentérica inferior com seus vasos subsidiários), fígado, pâncreas, intestino delgado, baço, estômago, ureter e glândulas vesiculares. As imagens foram obtidas durante cirurgias robóticas assistidas de cirurgias retais anteriores, abrangendo o período de fevereiro de 2019 a fevereiro de 2021.

Este conjunto de dados foi adquirido no University Hospital and Faculty of Medicine Carl Gustav Carus, localizado na Alemanha. O conjunto de dados foi criado a partir de imagens obtidas através do sistema Da Vinci Xi da Intuitive Surgical, localizada em Sunnyvale, Califórnia, EUA. As cirurgias foram registradas utilizando um endoscópio padrão Da Vinci Xi/X com uma câmera de 8 mm de diâmetro e ângulo de 30 graus, também da Intuitive Surgical. O registro das cirurgias foi realizado com o sistema CAST da Orpheus Medical GmbH, em Frankfurt, Alemanha. Cada gravação foi salva com uma resolução de 1920×1080 pixels em formato MPEG-4.

O conjunto de dados é notável por ser o primeiro com acesso completamente aberto, contendo dezenas de milhares de imagens com segmentação das regiões anatômicas. O principal motivo para isso é o alto número de horas necessárias de profissionais altamente qualificados para segmentarem pixel a pixel todas as imagens.

Os dados incluíam imagens com perturbações como desfoque devido a movimentos da câmera, sujeira na lente e presença de sangue ou fumaça. Isso garantia a robustez dos modelos de aprendizado de máquina treinados neste conjunto de dados. Para assegurar a qualidade das anotações, três anotadores independentes com experiência substancial em cirurgia robótica assistida de reto criaram as anotações pixel a pixel, que foram posteriormente revisadas por um cirurgião com quatro anos de experiência na mesma área.

Este cuidadoso processo de anotação e revisão assegurou a precisão e a confiabilidade dos dados, tornando-os uma fonte relevante para a pesquisa na área de visão computacional aplicada à cirurgia. As imagens são uma representação valiosa da realidade cirúrgica, fornecendo uma base sólida para o treinamento e validação de algoritmos de aprendizado de máquina (KOLBINGER et al., 2023b).

4.2 Bibliotecas Utilizadas

As bibliotecas utilizadas neste projeto formaram a base do desenvolvimento e implementação de modelos de aprendizado de máquina, especialmente em tarefas de visão computacional. A biblioteca `PyTorch`, juntamente com suas extensões, foi a espinha dorsal deste projeto. `Torch` é

conhecida por sua eficiência e flexibilidade na construção de redes neurais artificiais profundas, enquanto `torchvision` complementa-a com conjuntos de dados populares, arquiteturas de modelos e transformações comuns, facilitando o trabalho com imagens.

Além disso, `torchmetrics` foi utilizada, uma biblioteca focada em métricas de aprendizado de máquina, proporcionando uma forma conveniente e padronizada de avaliar e comparar modelos de aprendizado de máquina. Por fim, foi utilizada a biblioteca `NumPy`, uma biblioteca muito utilizada para computação científica em Python. Essa biblioteca ofereceu suporte para grandes matrizes multidimensionais, juntamente com uma ampla coleção de funções matemáticas para operar nessas matrizes. Ainda, também foram usadas as bibliotecas `Pandas`, `matplotlib`, `PIL` e `TQDM`, que serão explicadas nas próximas seções. Todo o código desenvolvido está disponível publicamente em um repositório no GitHub implementado em quatro arquivos distintos: `model.py`, `train.py`, `utils.py` e `dataset.py`.

4.2.1 PyTorch

`PyTorch` foi a principal biblioteca utilizada. Ela é uma biblioteca aberta desenvolvida para a linguagem Python para aprendizado de máquina, atuando no desenvolvimento de modelos de inteligência artificial de aprendizado profundo. A biblioteca foi amplamente utilizada para pesquisa e produção em áreas como visão computacional, processamento de linguagem natural, entre outras.

O `PyTorch` foi construído em torno da biblioteca `Torch`, que utiliza o conceito de tensores, estruturas de dados semelhantes a matrizes multidimensionais. Os tensores no `PyTorch` são similares aos *arrays* do `NumPy`, com suporte para computação paralela em múltiplos núcleos de CPU e GPU simultaneamente, e até em múltiplos computadores, permitindo operações mais rápidas em grandes conjuntos de dados. Isso permite aproveitar o poder de processamento paralelo das GPUs do *Colab*, acelerando o treinamento e inferência de modelos de aprendizado profundo.

O `PyTorch` oferece uma estrutura de baixo nível para operações matemáticas e manipulação de tensores, fornecendo também módulos de alto nível. O `torch.nn` é composto de blocos de construção para redes neurais, usados no arquivo `Model`, `torch.optim` para otimização de modelos e `torch.utils` para manipulação de conjuntos de dados e treinamento de modelos, utilizados nos arquivos `Train` e `Utils`.

4.2.2 Torchmetrics

A biblioteca `torchmetrics` foi uma extensão da biblioteca `PyTorch`, utilizada para facilitar a avaliação de modelos de aprendizado profundo. Esta extensão ofereceu uma variedade de métricas de desempenho pré-implementadas que podiam ser usadas para avaliar a precisão, a revocação, a F1-score, entre outras, em tarefas de classificação, regressão e segmentação. Especificamente, foi utilizada a função `Dice` para a classificação, empregada no arquivo `Utils`. O coeficiente `Dice` é uma

métrica comumente usada para avaliar a sobreposição ou similaridade entre duas amostras. Além disso, a biblioteca é compatível com GPUs, permitindo a aceleração do cálculo de métricas em grandes conjuntos de dados.

4.2.3 Torchvision

Como expansão da biblioteca PyTorch, a biblioteca `torchvision` forneceu transformações de dados comumente usadas em visão computacional. Focada em tarefas de processamento de imagens e visão de máquina, essa biblioteca facilitou o desenvolvimento de modelos de aprendizado profundo.

A biblioteca ofereceu uma ampla variedade de transformações, como redimensionamento, corte, normalização, rotação, espelhamento e transformações afins. Essas transformações foram aplicadas aos conjuntos de dados durante o carregamento para preparar os dados de entrada para o treinamento de modelos, sendo utilizadas nos arquivos `Utils` e `Train`. Além disso, foram utilizadas as ferramentas de conversão de matrizes `numpy` para tensores.

4.2.4 NumPy

A biblioteca `NumPy` foi escolhida por ser a principal biblioteca em Python utilizada para computação numérica e manipulação de matrizes multidimensionais. Ela forneceu uma série de funcionalidades poderosas para realizar operações matemáticas complexas de forma eficiente, especialmente em grandes conjuntos de dados.

O principal uso da `NumPy` foi o `np.array`, uma estrutura de dados que representa matrizes multidimensionais, utilizadas nos arquivos `Dataset` e `Train`. Essas matrizes são eficientes em termos de memória e permitiram a execução rápida de operações matemáticas em massa, além da manipulação, criação e conversão de entradas para matrizes.

4.2.5 Pandas

O `Pandas` possui como principal estrutura o *DataFrame*, semelhante a uma tabela de banco de dados ou planilha do Excel. Essa estrutura foi utilizada para salvar os resultados dos testes de acurácia do treinamento em cada iteração da rede. O `Pandas` permitiu a escrita dos dados no formato CSV, utilizado no arquivo `train.py`. Isso facilitou a importação e exportação de dados entre as iterações e pausas entre treinamentos do modelo.

4.2.6 Bibliotecas de Apoio

As bibliotecas `matplotlib`, `PIL` e `TQDM` foram usadas para funções de apoio como, para plotar gráficos, converter imagens e mostrar o progresso do código. A biblioteca `matplotlib.pyplot` foi uma poderosa ferramenta de visualização de dados em Python. Ela ofereceu uma interface de programação semelhante ao MATLAB, com grande flexibilidade para criar gráficos com múltiplos dados complexos. Foi possível personalizar todos os aspectos visuais dos gráficos gerados, como cores, estilos de linha, legendas, títulos e escalas. Essas opções foram utilizadas para traçar os gráficos com a acurácia da rede neural em relação às iterações, sendo usados no arquivo `train.py`.

`PIL` (do termo em inglês, *Python Imaging Library*) foi uma biblioteca de processamento de imagens em Python. Sua habilidade de conversão de imagens foi utilizada para transformar as imagens BGR (*Blue, Green, Red*) originais para RGB (*Red, Green, Blue*), sendo usada no arquivo `dataset.py`. A transformação de BGR para RGB envolve a troca das posições dos canais de cor, uma vez que o formato BGR armazena a cor azul primeiro, seguido do verde e do vermelho, enquanto o formato RGB armazena a cor vermelha primeiro, seguido do verde e do azul.

Por fim a biblioteca `TQDM` foi usada para criar uma barra de progresso durante iterações em laço ao longo da execução do código. Isso permitiu acompanhar o progresso da operação e estimar o tempo restante de execução, melhorando a experiência do usuário. Essa funcionalidade foi utilizada no arquivo `utils.py`.

4.3 Ambiente De Execução

O ambiente de execução escolhido para este projeto foi o Google Colaboratory (Colab), pertencente ao Google LLC e sediado em Mountain View, Califórnia, Estados Unidos. O Colab é uma plataforma em nuvem que permite escrever, compartilhar, executar e salvar código em Python diretamente no navegador. Ele é especialmente útil para desenvolvimento em aprendizado de máquina, análise de dados e pesquisa científica, pois oferece acesso gratuito a recursos computacionais, como CPUs, GPUs e TPUs.

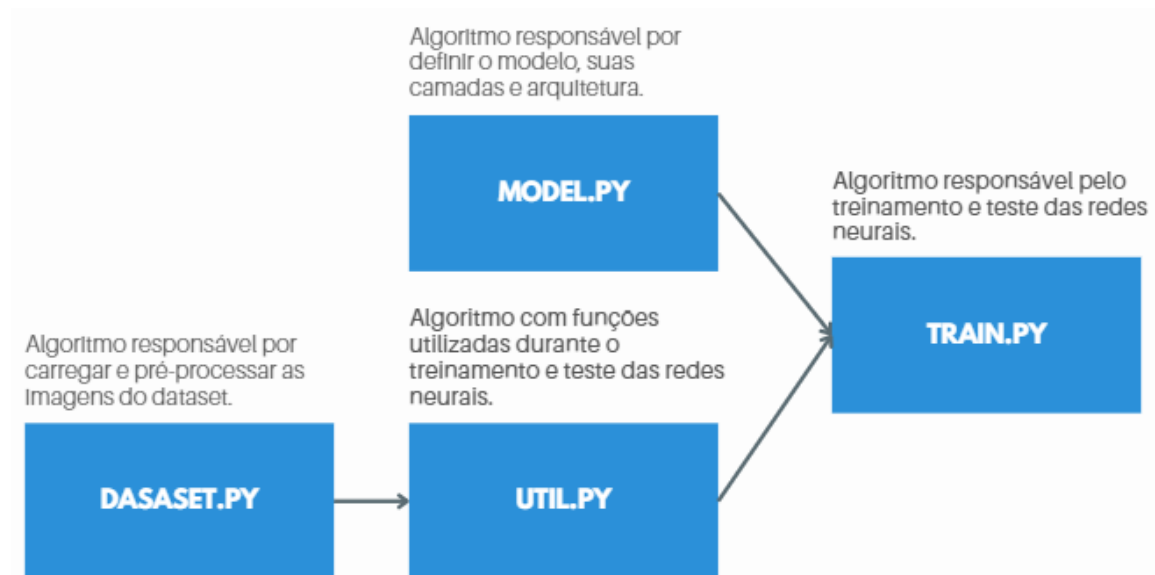
A integração com o Google Drive e a compatibilidade com blocos de anotação Jupyter, um ambiente interativo que permite executar, escrever e editar código Python em tempo real, também contribuíram para a escolha do Colab como ambiente de execução, otimizando o fluxo de trabalho e a gestão de dados.

Utilizamos os recursos de GPU para acelerar o treinamento. No entanto, os recursos de GPU são limitados pelo Colab, então só fomos capazes de treinar 10 gerações por dia. Além disso, utilizamos o modo CPU para gerar as imagens após o treinamento. Mesmo com o recurso de GPU com 15GB e 12,7GB de RAM, com um tamanho de lote de 12 e profundidade da rede de 50 camadas, usamos cerca de 82% da GPU, além de 33GB de 78,2GB usados para armazenar as imagens da biblioteca. Fica evidente que o processo de treinamento é extremamente pesado computacionalmente.

4.4 Pré-processamento e Treinamento

Foram criados 4 algoritmos para a rede neural artificial: `dataset.py`, `model.py`, `util.py` e `train.py`. Cada algoritmo foi responsável por uma parte do treinamento, pode se ver como estão distribuídos na Figura 5. Utilizando das bibliotecas descritas na fundamentação teórica.

Figura 5 – Fluxograma dos Algoritmos e onde cada um é subsequente utilizado.



fonte: Produção do próprio autor.

O algoritmo `dataset.py` foi responsável por preparar as imagens para o treinamento. As imagens foram lidas e convertidas para *arrays* que a rede neural conseguia utilizar. O `model.py` definiu o formato e as capacidades da rede, configurando os blocos do modelo, o número e o tipo das camadas, bem como suas conexões e atalhos, para construir a arquitetura UResNet.

Além disso, o algoritmo `util.py` estabeleceu 12 funções que foram usadas no treinamento, incluindo funções de pré-processamento de imagem para aumentar o número de imagens do DSAD e funções para auxiliar no treinamento. Por fim, o `train.py` foi responsável por treinar o modelo de inteligência artificial, calcular sua acurácia e salvá-la. Esse também foi o algoritmo onde os hiperparâmetros foram definidos e o caminho para o DSAD foi especificado.

4.4.1 Carregamento e Pré-Processamento das Imagens

O carregamento e o pré-processamento das imagens carregadas foi implementado pelo arquivo `dataset.py`. Neste arquivo, é implementada a classe `Dataset`, a qual é responsável por ler as imagens das pastas do DSAD e as transformar em um conjunto de dados que o `PyTorch` pode entender. Para tanto, o código em `dataset.py` deve ter uma classe que herde de `torch.utils.data.Dataset`, a qual será utilizada pelos outros algoritmos. Em nosso caso, essa classe se chamou `RaabinDataset`. O algoritmo também aplicou transformações para a convenção das imagens em

RGB, e para a transformação da imagem de referência (*ground truth*) para a codificação one-hot. A codificação one-hot é uma técnica usada para representar dados binários como vetores, onde imagens de referência são pretas (falso) e brancas (verdadeiro). Cada pixel é convertido em um vetor binário, com 1 representando a presença da característica (branca) e 0 representando a ausência (preta), facilitando a interpretação pelo modelo de aprendizado de máquina.

O algoritmo foi iniciado com a função `__init__`, que foi definida para inicializar a classe. Várias operações foram realizadas para configurá-la. Primeiramente, as entradas foram definidas e o diretório de imagem foi atribuído à variável `image_dir`. A transformação, que pode ser aplicada às imagens, foi atribuída à variável `transform`. Depois, foram criadas listas contendo os nomes dos arquivos de imagem e de máscaras no diretório de imagem, filtrando-os com base em seus prefixos ('image' e 'mask', respectivamente). Além disso, os nomes dos arquivos de imagem e de máscaras foram ordenados numericamente com base nos números extraídos de seus nomes de arquivo, garantindo uma correspondência adequada entre eles.

A definição de `__len__` para a classe foi feita utilizando a quantidade de imagens presentes na pasta do DSAD, definida no `image_dir`. A classe foi definida para retornar um tensor binário, uma vez que a classificação é binária.

Na definição `__getitem__`, a entrada é definida como um índice `idx`. Primeiramente, foi verificado se o `idx` é um tensor PyTorch. Caso seja, ele é convertido em uma lista. Em seguida, foi feita a leitura da imagem correspondente ao índice `idx` a partir do diretório de imagens. Da mesma forma, foi lida a respectiva imagem de máscara, também com base no índice fornecido. Na etapa seguinte, foi realizada a conversão dessas imagens para *arrays* do NumPy, e a conversão para RGB, garantindo uma representação adequada para o processamento posterior.

Para a máscara da imagem, houve uma manipulação adicional: primeiro, foi criada uma nova matriz de máscara preenchida com zeros, com o mesmo formato que a matriz original de máscara. Em seguida, foram aplicadas condições para definir os valores das máscaras com base nos valores dos pixels na imagem original. Esta abordagem converteu a máscara em uma matriz máscara de formato binário simples, onde valores acima do limiar 125, valor máximo de 225 (8 bits) indicando branco absoluto, são mapeados para 1 e valores abaixo desse limiar são mapeados para 0.

Os *arrays* de imagem e máscara processados foram então armazenados em um dicionário Python, onde cada um recebeu uma chave única ('image0' e 'image1', respectivamente). Se houve uma transformação especificada para ser aplicada aos dados, ela foi então aplicada ao dicionário. Finalmente, o dicionário contendo a imagem e sua máscara correspondente, juntamente com qualquer transformação aplicada, foi retornado como saída da definição. Essas etapas permitem recuperar e processar eficientemente os itens do conjunto de dados durante a iteração.

4.4.2 Estruturação do Modelo

O arquivo `Model` é composto por 5 funções que trataram a construção e definição da rede UResNet. São elas: `mySequential`, que trata múltiplas entradas de módulo de rede neural,

`block_standard` e `block_bottleneck`, que são módulos de rede neural artificial configurados especificamente, e `UResNet`, que implementa a rede `UResNet`.

4.4.2.1 `mySequential`

Essa função `mySequential` estendeu a funcionalidade da função `nn.Sequential`, permitindo lidar com múltiplas entradas na geração de camadas. No componente `forward`, itera sobre os módulos contidos na sequência e aplica-os às entradas fornecidas. Se o tipo de entrada for uma tupla, significa que várias entradas foram passadas, então aplica o módulo a cada entrada da tupla. Caso contrário, assume-se que apenas uma entrada foi fornecida e aplica o módulo a essa entrada, retornando a saída do módulo aplicado à entrada.

4.4.2.2 `block_standard`

Essa função `block_standard` foi um bloco convolucional padrão construído para ser usado na rede neural convolucional, possuindo o mesmo número de entradas e saídas. No componente `__init__`, a inicialização ocorre, onde os parâmetros do bloco, como o número de canais de entrada (`in_channels`) e saída (`out_channels`), o redimensionamento de identidade (`identity_downsample`), o passo (`stride`), e uma variável booleana indicando se a camada está realizando uma operação de upsampling (`up`), são configurados, com todos os valores dessas variáveis sendo configurados como `self`.

Depois, se `up` for falso, é inicializada a camada convolucional 2D usando `nn.ConvTranspose2d` com respectivos tamanhos de `kernel` iguais ao passo. Se `up` for verdadeira, `nn.Conv2d` é inicializado com tamanhos de `kernel` igual a 3, seguidas por camadas de normalização e convolução `BatchNorm2d`, `Conv2d`, `BatchNorm2d` e funções de ativação `ReLU`, com cada uma das camadas sendo nomeada em uma variável.

No componente `forward`, a entrada `x` foi primeiro armazenada na variável `identity` para uso posterior. Então, a entrada passou pelas camadas definidas em `__init__` na ordem que foram configuradas. Se houve uma operação de redimensionamento de identidade, ela é aplicada. Se a entrada `long_skip` não for nula, ou seja, se houve conexões de atalho longas, a saída do bloco foi concatenada com `long_skip`. Finalmente, a saída do bloco foi a soma da saída das camadas e da identidade, passada por uma função de ativação `ReLU`. Após isso, a variável `identity` é deletada para economizar memória. Foi retornado a saída da função de ativação, juntamente com `long_skip`.

4.4.2.3 block_bottleneck

A função `block_bottleneck` foi outro tipo de bloco convolucional usado em redes neurais convolucionais, possuindo o dobro de saídas em relação ao número de entradas. No componente `__init__`, os parâmetros do bloco foram inicializados, incluindo os canais de entrada (`in_channels`), os canais de saída (`out_channels`), o redimensionamento de identidade (`identity_scale`), o passo (`stride`) e uma variável booleana indicando se a camada está realizando uma operação de upsampling (`up`).

Se `up` for verdadeiro, mudamos a expansão para retornar o mesmo número de saídas e entradas. Iniciamos as camadas com `nn.ConvTranspose2d`, assim como em `block_standard`, seguida pela camada `BatchNorm2d` e `nn.Conv2d` com tamanhos de `kernel` igual a 3 e passo 1. Se `up` for falso, iniciamos as camadas com `nn.Conv2d` com tamanhos de `kernel` igual a 3 e passo 1, seguida pela camada `BatchNorm2d` e `nn.Conv2d` com tamanhos de `kernel` igual a 3 e passo 1.

Depois, fazemos mais uma normalização com `BatchNorm2d`, outra convolução e outra normalização, ficando com 3 convoluções intercaladas com 3 normalizações. A ativação foi então realizada com `ReLU`. Já a função `forward`, foi executada seguindo a lógica da função `block_standard`, com a diferença sendo as camadas definidas no `__init__`. Foi retornado a saída da função de ativação (resultado de todas as camadas mais a identidade), juntamente com `long_skip`.

4.4.2.4 UResNet

A função `UResNet` constrói as camadas da arquitetura mista de ResNet e UNet, contendo todas as camadas de convoluções, normalizações e ativações, construindo a ordem e quantidade das camadas de maneira configurável. Também foi definida a função `__init__`, na qual os principais componentes do modelo de inteligência artificial foram inicializados. Primeiramente, foi configurado o bloco de expansão entre normal e gargalo, a quantidade de camadas, a quantidade de canais da imagem e o número de classes de saída.

Em seguida, foi definido o canal de entrada como tendo 64 entradas. Foram definidas as camadas convolucionais e de normalização. A primeira camada convolucional `nn.Conv2d` recebeu como entrada o número de canais da imagem de entrada, tendo 64 filtros de convolução, com um tamanho de `kernel` de 7, um `stride` de 2 e um `padding` de 3. Após a convolução, foi aplicada uma camada de normalização `nn.BatchNorm2d`, seguida de uma função de ativação `nn.ReLU`.

Após a primeira camada, foi aplicada uma camada de `nn.MaxPool2d` com um `kernel` de tamanho 3, um `stride` de 2 e um `padding` de 1. Esta camada reduz a resolução espacial da imagem pela metade e diminui a quantidade de parâmetros na rede neural. As camadas de blocos residuais (camada 1 a 8 para expansão) foram criadas utilizando o componente `_make_layer` (será explicado a seguir), que instanciou blocos residuais de acordo com os parâmetros passados (bloco e camada). Essas camadas são organizadas de forma a aumentar gradualmente o número de canais de saída, reduzindo simultaneamente as dimensões espaciais da imagem.

Finalmente, são definidas as últimas camadas convolucionais (convolução 1 e 2 para encolhimento) que processaram a saída das camadas anteriores e produziram a saída final da rede. Elas utilizaram a função `nn.ConvTranspose2d`, que recebeu como entrada o número de canais de entrada da imagem e 128 canais de entrada respectivamente, com 64 canais de saída para ambas convoluções, com um tamanho de `kernel` de 2, um `stride` de 2 e um `padding` de 0. Essas camadas são seguidas por uma camada de normalização `nn.BatchNorm2d` e uma camada de ativação `nn.Softmax`. A função `softmax` é aplicada para retornar um tensor binário com a mesma dimensão do tensor de entrada.

O método `_make_layer` foi responsável por criar camadas de blocos residuais na rede neural artificial. Ele recebeu como entrada o tipo de bloco a ser utilizado (normal ou gargalo), o número de blocos residuais a serem criados, o número de canais de saída das camadas, o `stride` a ser aplicado nas convoluções, e uma variável booleana indicando se a camada está realizando uma operação de *upsampling* (`up`).

Inicialmente, foi definida uma variável `identity_scale` como vazia e uma lista vazia de camadas que armazenará as camadas residuais que serão criadas. Em seguida, foi verificado se `up` era falso. Se fosse, também foi verificado se o `stride` era diferente de 1 ou se o número de canais de entrada não era igual ao número de canais de saída multiplicado pela expansão do bloco. Se uma dessas condições fosse verdadeira, o `mySequential` era chamado e uma operação de convolução era realizada, seguida por uma camada de normalização em lote para atualizar as dimensões do tensor de entrada `identity_scale`. O primeiro bloco foi adicionado à lista de camadas, e o número de canais de entrada foi atualizado para ser igual ao número de canais de saída multiplicado pela expansão do bloco.

Em seguida, um laço foi executado para criar os blocos residuais restantes. Cada bloco foi adicionado à lista de camadas, mantendo o número de canais de entrada constante. Se `up` for verdadeiro, a lógica será semelhante. Primeiro, será verificado se o `stride` é diferente de 1 ou se o número de canais de entrada não é igual ao número de canais de saída multiplicado pela expansão do bloco. Se uma dessas condições for verdadeira, será chamado o `mySequential` e uma operação de convolução transposta será realizada, seguida por uma camada de normalização em lote, com os canais de saída sendo ajustados para o número de canais de saída dividido pelo valor de expansão do bloco. Isso é feito para atualizar as dimensões do tensor de entrada `identity_scale`. O primeiro bloco será adicionado à lista de camadas, e o número de canais de entrada será atualizado para ser igual ao número de canais de saída multiplicado pela expansão do bloco.

Em seguida, se o `stride` for 1 e `up` for verdadeiro, ou se a expansão do bloco for 1 e `up` for verdadeiro, atualizamos o número de canais de entrada para o dobro do número de canais de saída. Para outros casos, atualizamos o número de canais de entrada para o quádruplo do número de canais de saída. Por fim, um laço é executado para criar os blocos residuais restantes com a operação de *upsampling* ativada. Cada bloco é adicionado à lista de camadas, mantendo o número de canais de entrada constante.

Finalmente, a lista de camadas é passada para o construtor de `mySequential` e retornada como saída do método. O método `forward` implementa a passagem direta (*forward pass*) da rede neural.

Inicialmente, o tensor de entrada, chamado de x , é passado através das primeiras camadas da rede definidas em `__init__`, que incluem uma operação de convolução, normalização de lote e ativação ReLU. Em seguida, o resultado das camadas é armazenado na primeira posição de uma lista chamada `long_skip`. Por fim, é realizado um `max pooling` para reduzir as dimensões do tensor x .

Após isso, as camadas residuais (camadas 1 a 4) são chamadas para processar o tensor x . Os resultados intermediários dessas camadas são armazenados na lista `long_skip` em suas posições respectivas, que é invertida logo em seguida. Em seguida, as camadas de `upsampling` (camadas 5 a 8) são chamadas com os tensores intermediários de x armazenados em `long_skip`. O tensor resultante é então processado pelas últimas camadas convolucionais (convolução 1, normalização, concatenação, retificação, convolução 2 e normalização) e passa por uma operação de `nn.Softmax` para obter as probabilidades de classe. Após isso, as variáveis temporárias são deletadas da memória e o tensor resultante é retornado como saída do método.

4.4.3 Ferramentas e Funções Uteis

O *util.py* foi um conjunto de funções utilizadas durante o treinamento e teste do modelo, além de conter unções chamadas de transformações funcionais, controladas pelo usuário. Essa abordagem permitiu controlar de forma determinística as transformações aplicadas nas imagens, garantindo que as mesmas transformações fossem replicadas nas máscaras.

4.4.3.1 ToTensor

A primeira função definida foi denominada `ToTensor`. Trata-se de uma função de transformação funcional projetada para converter uma `ndarray` em um tensor `PyTorch`. Na definição `__init__`, a transformação foi inicializada com o número de imagens (não máscaras) que devem ser convertidas em tensores, sendo 1 o número de imagens padrão (a primeira imagem apenas).

Na definição `__call__`, as imagens foram iteradas e processadas. Se o índice da imagem fosse menor que o número especificado, a imagem foi convertida em um tensor usando a função `to_tensor` da biblioteca `TorchVision`. Caso contrário, a imagem foi convertida em um tensor `PyTorch` usando `torch.from_numpy`, seguida pela permutação das dimensões para que a ordem fosse apropriada para tensores `PyTorch` (de 'HWC' para 'CHW', por exemplo, onde C é canal, H é altura e W é largura). Por fim, as imagens processadas foram retornadas como saída da definição.

4.4.3.2 Rotate

A função `Rotate` foi uma função de transformação que rotacionava uma imagem dentro de um dicionário de imagens, com a possibilidade de controlar a probabilidade de não ocorrência da rotação e

os limites de ângulo de rotação. Na função `__init__`, os parâmetros `limit` e `p` foram configurados. Se nenhum limite fosse especificado, os ângulos de rotação padrão foram definidos como `[0, 360]`. Da mesma forma, se nenhum valor de probabilidade fosse fornecido, o padrão foi definido como `0,5`.

Já na função `__call__`, as imagens foram processadas de acordo com a probabilidade especificada. Se um número aleatório fosse maior do que a probabilidade, a rotação seria aplicada. Um ângulo de rotação foi escolhido aleatoriamente dentro do intervalo de limites definido e, em seguida, todas as imagens no dicionário foram rotacionadas com esse ângulo usando a função `rotate` da biblioteca `TorchVision`. As imagens processadas foram retornadas como saída do componente.

4.4.3.3 CenterCrop e Resize

As funções `CenterCrop` e `Resize` foram funções de transformação que realizavam um recorte central ou um redimensionamento em uma ou várias imagens dentro de um dicionário de imagens. Na função `__init__`, o parâmetro `size` foi especificado para definir as dimensões do recorte central ou redimensionamento desejado, sendo uma lista com as dimensões, por exemplo, `[400, 200]`.

No componente `__call__`, as imagens no dicionário foram iteradas e a função `center_crop` ou `resize` da biblioteca `TorchVision` foi aplicada a cada imagem, que cortou cada imagem centralmente ou a redimensionou de acordo com as dimensões especificadas em `size`. As imagens processadas foram então retornadas como saída da função, mantendo a estrutura do dicionário original.

4.4.3.4 FlipHorizontal e FlipVertical

As funções `FlipHorizontal` e `FlipVertical` foram funções de transformação que realizaram uma inversão horizontal ou vertical em uma ou várias imagens dentro de um dicionário de imagens. Na função `__init__`, o parâmetro `p` foi especificado para definir a probabilidade de inversão horizontal ou vertical, onde `p` era um valor entre `0,0` e `1,0`.

No componente `__call__`, as imagens no dicionário foram iteradas e uma inversão (horizontal ou vertical a depender da função) foi aplicada aleatoriamente a cada imagem com base na probabilidade definida. Isso foi feito utilizando as funções `hflip` e `vflip` da biblioteca `TorchVision`. As imagens processadas foram então retornadas como saída do componente, mantendo a estrutura do dicionário original.

4.4.3.5 Normalize

A função `Normalize` foi uma função de transformação que normalizou um conjunto de imagens dentro de um dicionário de imagens. Com `__init__`, foram especificados os parâmetros `n=1`, `mean=0,5` e `std=0,5` como parâmetros padrão. O parâmetro `n` representou o número de imagens que

foram normalizadas, enquanto `mean` e `std` foram listas que continham as médias e os desvios padrão usados para a operação.

Na definição `__call__`, as imagens no dicionário foram iteradas e a normalização foi aplicada às primeiras `n` imagens do conjunto, utilizando as médias e os desvios padrão especificados. Isso foi realizado utilizando a função `normalize` da biblioteca `TorchVision`. A função aplicou uma transformada seguindo a regra:

$$\text{output[channel]} = \frac{\text{input[channel]} - \text{mean[channel]}}{\text{std[channel]}}, \quad (1)$$

onde o canal é uma cor da imagem. As imagens normalizadas são então retornadas como saída da função, mantendo a estrutura do dicionário original.

4.4.3.6 Affine

Na função `Affine`, é executada uma função chamada de Transformações Afins, que executa várias operações em uma imagem, incluindo rotação, translação, dimensionamento, cisalhamento e interpolação, mantendo o centro da imagem invariante. Na função `__init__`, os parâmetros `size`, `scale` e `p` foram especificados. O parâmetro `size` era uma lista contendo a altura máxima e a largura máxima para a translação da imagem. O parâmetro `scale` foi um valor de escala para realizar o `affine`, variando entre 0 e 1,0. O parâmetro `p` foi a probabilidade de aplicar a transformação.

Já na função `__call__`, as imagens no dicionário foram iteradas e a transformação foi aplicada a elas com a probabilidade definida por `p`. A transformação aplicou a rotação e o cisalhamento com uma probabilidade `aleatoria * scale * 360`, uma translação aleatória dentro dos limites especificados pelo parâmetro `size`. Essas transformações tentam simular variações na perspectiva e na orientação da câmera, e foram realizadas usando a função `affine` da biblioteca `TorchVision`. As imagens transformadas são então retornadas como saída do componente, mantendo a estrutura do dicionário original.

4.4.3.7 save_checkpoint e load_checkpoint

A função `save_checkpoint` recebeu o estado do modelo e salvou esses estados em um arquivo utilizando a função `torch.save`. O nome padrão do arquivo foi `'my_checkpoint.pth.tar'`, onde `pth` era o número da geração, e uma mensagem de aviso foi impressa informando sobre o salvamento.

A função `load_checkpoint` foi utilizada para carregar um `checkpoint` previamente salvo. Ela recebeu o `checkpoint`, o modelo e, opcionalmente, o otimizador. O estado do modelo foi então atualizado com os parâmetros salvos do `checkpoint`. Se um otimizador também foi fornecido, ele foi atualizado com os parâmetros do otimizador salvos no `checkpoint`. Durante a execução, foi impressa uma mensagem indicando que o `checkpoint` estava sendo carregado.

4.4.3.8 check_accuracy

A função `check_accuracy` foi utilizada para verificar a acurácia do modelo de inteligência artificial, recebendo 5 argumentos:

```
def check_accuracy(loader , model , loss_fn ,
                    device='cuda' if torch.cuda.is_available() else 'cpu' ,
                    **kwargs ):
```

Sendo o `Loader` uma lista, que será explicada a seguir, o `model` recebeu o modelo da UResNet, o `loss_fn` recebeu a função usada para o cálculo de perda, como `L1Loss` ou `CrossEntropyLoss`, o `device` definiu se seria utilizada uma GPU ou CPU para os cálculos e `kwargs` recebeu o título do teste.

Essa função começou inicializando três variáveis zeradas que controlam o número de pixels, o número de acertos de pixels e o `dice_score`. Para calcular o escore `Dice`, foi usada a fórmula abaixo, onde VP representa os verdadeiros positivos, FP os falsos positivos e FN os falsos negativos.

$$\text{Dice} = \frac{2 \times \text{VP}}{2 \times \text{VP} + \text{FP} + \text{FN}} \quad (2)$$

Foi utilizada a função `.eval()` para preparar a rede para o teste, desativando ou ativando certas partes da rede neural para poder testá-la, o que diminui o consumo de memória. A biblioteca `tqdm` foi iniciada para mostrar a barra de progresso do teste. Um laço foi iniciado com `torch.no_grad()`, desabilitando o cálculo de gradiente para economizar memória. Com essa configuração, iteramos um loop sobre o dicionário de imagens, extraíndo a imagem e a máscara e as atribuindo às variáveis `x` e `y`, respectivamente, por facilidade, e enviando-as para o `device`. Em seguida, convertemos o `y` para `float` e iniciamos o modelo com `x`. Usando o tamanho obtido na inicialização, foi feito um recorte central em `y` e uma conversão para binário da previsão do modelo.

Com o componente de cálculo de perda, foram calculadas as perdas entre o modelo binário de `x` e a máscara recortada `y`. Em seguida, foi somado o número de pixels em comum de `x` e `y` à variável de controle do número de acertos, e o número de pixels de `x` ao número total de pixels. Foi movido para `torch.int32` dentro do `device` o objeto `Dice`, que foi iniciado e usado para calcular o escore de `x` e `y`, com o resultado sendo somado a `dice_score`. A cada iteração, atualizou-se o `tqdm` com o valor da acurácia média, para fins de visualização.

Para finalizar o laço, salvamos o resultado das perdas em uma variável local e deletamos o restante das variáveis criadas no laço. Em seguida, convertemos a soma do número de acertos, a soma do número de pixels, a soma do `dice_score` e o tamanho do `Loader` para `float` e `int`, com o restante das variáveis sendo apagadas. Imprimimos a acurácia e o escore `Dice`, retornando os valores de acurácia, perda e escore `Dice`.

4.4.3.9 get_loaders

Foi construída uma função para aplicar todas as transformações funcionais ao DSAD, a fim de gerar imagens e máscaras com transformações idênticas, o que não ocorreu ao utilizar as transformações prontas da biblioteca `TorchVision`. Além disso, o estado da rede neural artificial foi testado e retornados três bancos de dados (treino, teste, validação), o número de lotes e a posição dos núcleos de processamento GPU ou CPU.

A função começou recebendo o diretório do DSAD para treino, a porcentagem de imagens a serem atribuídas para validação e teste, o tamanho do lote e das imagens, o número de núcleos de processamento, `pin_memory=True` para acelerar a transferência das imagens da CPU, onde foram carregadas, para a GPU, onde serão utilizadas, o diretório do DSAD para validação e a porcentagem de imagens a serem cortadas do treinamento ou validação.

Definimos duas variáveis que representam as transformadas a serem aplicadas ao primeiro diretório de imagens para treinamento e validação. Utilizamos a função `Compose`, que permite aplicar várias transformadas simultaneamente. Com ela, aplicamos as funções que criamos. Começamos com o `Resize` para o tamanho das imagens recebido pela função, seguido por `FlipVertical` e `FlipHorizontal` com 50% de chance de serem aplicadas, e por fim a `Normalize` com os parâmetros definidos como `n=1`, `mean=[0, 4338, 0, 31936, 0, 312387]`, `std=[0, 1904, 0, 15638, 0, 15657]`.

Determinamos o número de transformadas a serem aplicadas para cada diretório subsequente de imagens do DSAD como 5. Em seguida, concatenamos todos os diretórios de imagens em uma lista. Com a lista, calculamos o tamanho do conjunto de dados a ser usado para treino e teste com base na porcentagem de imagens recebidas pela função. Dividimos o conjunto de dados usando a função `random_split` e o tamanho calculado, gerando uma lista com a porcentagem desejada de dados.

Se um diretório de validação não fosse definido, refaríamos a divisão do conjunto de dados, mas com uma semente aleatória diferente, gerando duas listas de dados, uma para treinamento e outra para validação. Se o diretório fosse definido, concatenaríamos os diretórios definidos em uma lista de validação.

Criamos um laço que passou por todos os diretórios do DASD. Dentro desse laço, um segundo laço foi iniciado para aplicar as transformações numeradas anteriormente, do segundo diretório em diante. Para a primeira transformação, aplicamos o `Rotate` com 72° graus por transformação, o valor sendo escolhido por ser 360° graus divididos por 5 transformações, o `Resize` com o tamanho recebido e o `Normalize` com os mesmos valores anteriores para todos os diretórios. Com (50%) de chance de ocorrer em cada diretório, aplicamos o `FlipVertical` e `FlipHorizontal`.

Para a segunda transformação em diante, aplicamos a mesma lógica das transformações anteriores, com a adição da transformação

$$\text{Affine}(\text{size} = [0,5 \times \text{altura_imagem}, 0,5 \times \text{largura_imagem}]), \quad (3)$$

Para aplicar as transformações, chamamos a função `Dataset` e a passamos o diretório e as transfor-

mações, concatenando o resultado à lista de treinamento.

Se decidirmos recortar a lista de treinamento para acelerar o treinamento, calculamos o quanto precisamos cortar e realizamos o corte usando a função `random_split`, repetindo o processo para as listas de validação e teste. Por fim, a cada vez que a classe é chamada, 3 `DataLoader` são gerados, dependendo do tamanho do lote e do número de núcleos de processamento e a opção de `pin_memory`. O primeiro `DataLoader` foi configurado para o treinamento, com a opção `shuffle=True` para embaralhar os dados a cada época. Juntamente com o `DataLoader` de treinamento, também são gerados os de teste e de validação, sendo todos retornados toda vez que a função é chamada.

4.4.4 Treinamento, Validação e Salvamento da Rede

A *training.py* era o principal arquivo, responsável por utilizar todos os outros arquivos, treinando a rede, salvando e validando. Também podia continuar um treinamento prévio, nele todos os dados do treinamento foram configurados, como os hiperparâmetros e locais dos arquivos da rede neural.

Neste arquivo, são realizadas as definições de todos os caminhos para os arquivos a serem utilizados, definindo desde onde o resultado seria salvo o resultado do treinamento e as imagens resultantes, até onde o modelo seria salvo (como `checkpoint`). Os hiperparâmetros foram definidos individualmente, como o tamanho da imagem em 640x512, assim como as opções de carregamento, salvamento, continuação de treino e testes, entre outras variáveis utilizadas no arquivo.

4.4.4.1 `train_fn`

Foi criada a função `train_fn`, iniciada com um `loop` com `tqdm` para mostrar o progresso do treinamento. O laço foi inicializado com `loop`, iterando sobre o `loader`. Na propagação, foram definidas variáveis `x` e `y`, semelhantes ao `check_accuracy`, com as diferenças sendo a utilização da função `torch.cuda.amp.autocast()` antes de iniciar o modelo em `x` e o cálculo das perdas com um modelo não binário. A função `autocast()` roda automaticamente em um modo de melhor desempenho sem perder precisão, sendo recomendada apenas para avanços no modelo de inteligência artificial.

Para realizar a retropropagação, zeramos os gradientes do otimizador. Se estivermos utilizando uma GPU, invocamos a função `scaler` para aplicar o escalonamento no gradiente em relação à perda calculada. Em seguida, computamos os gradientes dos parâmetros do modelo utilizando a função `backward`. Após isso, avançamos um passo de otimização com `step`, aplicando os gradientes calculados nos pesos do modelo. Para uso futuro, salvamos os valores atuais utilizando `get_scale`. Por fim, atualizamos a escala interna com os novos gradientes utilizando `update`. Esses procedimentos são fundamentais para otimizar o treinamento de modelos de aprendizado profundo, especialmente quando se utiliza hardware acelerado por GPU.

Posteriormente, limpam-se as variáveis que não serão mais usadas para liberar espaço e atualizamos

o loop `tqdm`. Finalizando o laço, deletamos as variáveis do loop `tqdm` e `loader`. Ademais, também atualizamos a taxa de aprendizagem conforme definido no decaimento `optim.lr_scheduler.ExponentialLR(optimizer, gamma=0,9)` e salvamos a última e retornamos da função a última taxa de aprendizado e a perda da rede.

4.4.4.2 CustomCrossEntropyLoss

A função foi criada com o objetivo de explorar a criação de uma nova classe de perda personalizada, além de modificar uma existente para anular a saída se não houver nenhum verdadeiro positivo presente. Na função `__init__`, foi inicializada a função de perda de entropia cruzada padrão (`nn.CrossEntropyLoss()`). Durante a etapa de avanço (`forward`), a função calculou inicialmente a perda padrão utilizando a função `nn.CrossEntropyLoss()`. Em seguida, verificou-se a presença de alvos positivos nos dados. Caso não houvesse alvos positivos, a função retorna uma perda nula; do contrário, ela retorna a perda calculada. Essa abordagem permite uma adaptação flexível da função de perda de acordo com as necessidades específicas do problema em questão.

4.4.4.3 main

A função `Main` foi responsável por treinar e salvar a rede neural artificial. Ela iniciou com a definição do modelo `UResNet` a ser utilizada, a função de cálculo de perda, o otimizador `Adam` e a configuração do decaimento da taxa de aprendizado com `optim.lr_scheduler.ExponentialLR(optimizer, gamma=0,9)`. Além disso, carregou os conjuntos de dados de treinamento, teste e validação utilizando a função `get_loaders`.

Se a função carregasse uma rede para continuar o treinamento ou testá-la, utilizaria a função `load_checkpoint`. Em modo CPU, seria necessário passar um mapeamento para a CPU; em GPU, apenas o local do arquivo e o modelo seriam necessários. Em seguida, prosseguimos com a verificação da acurácia da rede neural.

Se o modelo fosse para treinamento, o algoritmo prepara o dicionário no qual os modelos serão salvos. Se estivessemos continuando o treinamento de um modelo, teríamos iniciado um temporizador e carregado novamente o modelo de inteligência artificial com a função `load_checkpoint`. Em seguida, teríamos lido o arquivo CSV para carregar as informações salvas, como `acc-valid`, `acc-test`, `loss`, `dice score-valid`, `dice score-test` e `time taken`, em um dicionário. Também teríamos atualizado o tempo já decorrido de treinamento.

Se o modelo for para o primeiro período, o algoritmo irá iniciar um temporizador, criando um dicionário com as informações a serem salvas e preenchendo com os resultados da função `check_accuracy` durante o treinamento do modelo. Por fim, também se calcula o tempo tomado salvando-o o resultado de todas essas operações.

Se estivermos usando GPU, teríamos inicializado o `scaler` como `GradScaler()`. Teríamos

definido a última taxa de aprendizado para poder usá-la na função de treinamento. Teríamos começado a iterar pelos períodos, com os valores definidos nos hiperparâmetros. Teríamos chamado a função `train_fn`, adicionando a perda ao dicionário, e salvo o modelo, se escolhido. Em seguida, teríamos verificado a acurácia de validação e teste do modelo, calculado o tempo gasto e adicionado esses valores ao dicionário. Por fim, teríamos salvo os dados do dicionário, se escolhido, e deletado todas as variáveis. Teríamos repetido esse laço até terminar todos os períodos.

4.5 Método Taguchi para Variação dos Parâmetros da Rede

O treinamento de uma rede envolve a escolha de várias macrovariáveis, e os valores dessas variáveis podem influenciar significativamente as taxas de acerto, como a acurácia e o coeficiente de Dice. Esses valores podem ser determinados com base na experiência prévia, em outros modelos existentes ou por meio de experimentação. Quem desenvolve o modelo desempenha um papel fundamental no resultado final, pois ao testar diferentes valores para cada variável, busca-se gerar redes neurais com ótimas taxas de acerto. Para realizar essa busca de forma mais eficaz, utilizamos o método Taguchi com 9 testes, selecionando 4 macrovariáveis e testando 3 valores para cada uma delas, conforme mostrado na Tabela 2.

Tabela 2 – Tabela Ortogonal de 4 variáveis com 3 parâmetros por variável, mostra um possível distribuição dos parâmetros para teste.

Treinamento	A	B	C	D
1	Padrão	Padrão	Padrão	Padrão
2	Padrão	Inferior	Inferior	Superior
3	Padrão	Superior	Superior	Inferior
4	Inferior	Padrão	Inferior	Inferior
5	Inferior	Inferior	Superior	Padrão
6	Inferior	Superior	Padrão	Superior
7	Superior	Padrão	Superior	Superior
8	Superior	Inferior	Padrão	Inferior
9	Superior	Superior	Inferior	Padrão

fonte: Produção do próprio autor.

A taxa de aprendizado foi o primeiro parâmetro escolhido, sendo comumente um dos parâmetros que mais influenciam a velocidade e a acurácia do treinamento de uma rede. Foram selecionados 3 valores para teste: 5×10^{-4} , 1×10^{-3} e 1×10^{-4} . Esses valores representam uma taxa de aprendizagem média, maior e menor, respectivamente. Essa escolha busca compreender o impacto de valores menores ou maiores da taxa de aprendizado no resultado final.

O tamanho do lote determina quantas imagens serão processadas por vez pela rede neural durante cada época do treinamento. Essa escolha não só afeta o desempenho da rede neural artificial, mas também sua capacidade de treinamento, pois lotes maiores exigem mais memória para armazenar os pesos do modelo para cada imagem do lote. Isso impacta diretamente na quantidade de memória RAM utilizada pelo algoritmo. Os valores testados foram 8, 6 e 12, seguindo a mesma abordagem do teste da

taxa de aprendizado, com um valor médio, um menor e um maior, buscando compreender o impacto de diferentes tamanhos de lotes no treinamento.

O usuário pode definir quantas camadas a rede terá, com opções de 18, 34, 50, 101 e 152. Como a UResNet segue uma arquitetura em forma de "U", a adição de cada nova camada na entrada resulta na inclusão de uma nova camada na saída, o que aumenta significativamente o tempo necessário para treinamento e uso da rede neural. Além disso, redes mais profundas podem ter maior propensão ao sobreajuste. Por isso, foram testadas redes neurais artificiais com 18, 34 e 50 camadas.

O último parâmetro escolhido foi a função de cálculo de perda, que geralmente influencia como verdadeiros positivos, falsos positivos, verdadeiros negativos e falsos negativos são tratados durante o treinamento do modelo de inteligência artificial. Foram testados três métodos: CrossEntropy, que é o mais utilizado e documentado na literatura e mede a diferença entre a distribuição de probabilidades previstas pelo modelo e a distribuição verdadeira, penalizando mais fortemente previsões incorretas; L1Loss, que utiliza o erro absoluto médio entre o valor esperado e o valor estimado pelo modelo neural, calculando a diferença absoluta média entre as previsões do modelo e os valores reais, sendo mais robusta a outliers; e um modelo personalizado, que retorna o valor de CrossEntropy normalmente, mas considera o resultado nulo se não houver valores positivos, útil em cenários onde a ausência de exemplos positivos deve ser tratada de maneira diferente. Esses testes foram realizados principalmente para compreender as capacidades desse parâmetro e como cada método influencia o desempenho do modelo durante o treinamento.

Com o método Taguchi, foi possível reduzir o número de testes de 81 para 9, obtendo-se a tabela 3. Essa tabela permitiu uma avaliação mais eficiente, em menos tempo e com menos recursos computacionais. Os resultados dos 9 testes foram avaliados pelas métricas de acurácia e coeficiente de Dice, conforme explicado anteriormente. Alguns modelos foram selecionados para uma inspeção mais detalhada, com mais épocas de treino. Os modelos escolhidos para isso foram as UResNets com maior número de camadas, já que essas geralmente necessitam de mais épocas de treinamento.

Tabela 3 – Tabela Ortogonal com a distribuição das variáveis e parâmetros que foram usados para o treinamento com o método Taguchi.

Treinamento	Lote	Taxa de Aprendizado	Modelo	Funções de Perda
1	8	5e-4	34	CrossEntropyLoss
2	8	1e-3	18	L1Loss
3	8	1e-4	50	CustomLoss
4	6	5e-4	18	CustomLoss
5	6	1e-3	50	CrossEntropyLoss
6	6	1e-4	34	L1Loss
7	12	5e-4	50	L1Loss
8	12	1e-3	34	CustomLoss
9	12	1e-4	18	CrossEntropyLoss

fonte: Produção do próprio autor.

4.6 Treinamento dos Modelos com Maiores Acurácias

Foram treinados mais 3 novos modelos para validar os hiperparâmetros escolhidos. Os parâmetros de taxa de aprendizado foram fixados em 1×10^{-4} , e o tamanho do lote foi fixado em 12 lotes. Os modelos foram testados com 34 e 50 níveis, combinados com as funções de perda CrossEntropy e L1Loss. Os modelos anteriores foram treinados por 20 gerações. Porém, as redes UResNet50 foram treinadas até 60 gerações. O número de gerações foi decidido observando as taxas de acertos, e o treinamento foi encerrado quando essas taxas praticamente deixaram de se modificar.

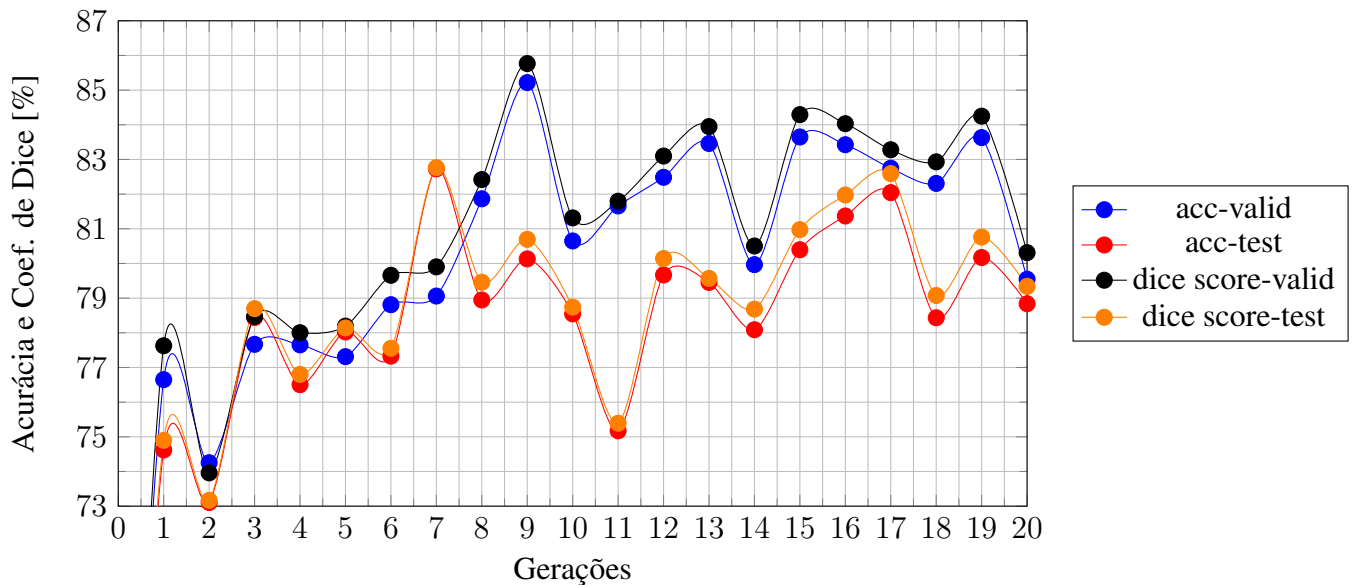
Um modelo ideal deveria possuir um valor de dice comparável ao de 83% apresentado por (KOLBINGER et al., 2022), uma taxa de acerto de validação superior à taxa do teste, indicando uma rede neural artificial bem treinada e com capacidade de generalizar as características abstraídas durante o treinamento, e, por fim, uma curva de acerto que estivesse crescendo ou estabilizada em um valor de próximo a 85%.

5 RESULTADOS E DISCUSSÃO

5.1 Modelos com Parâmetros do Método Taguchi

Com base nos resultados dos modelos treinados com os parâmetros do Método Taguchi, foram obtidos os resultados a serem apresentados nesta seção. Todas as redes foram treinadas por 20 gerações.

Figura 6 – Curvas de acurácia e pontuação dice do treinamento 1, com 8 imagens no lote, taxa de aprendizado de $5e-4$, UResNet de 34 camadas e função de perda CrossEntropyLoss.



fonte: Produção do próprio autor.

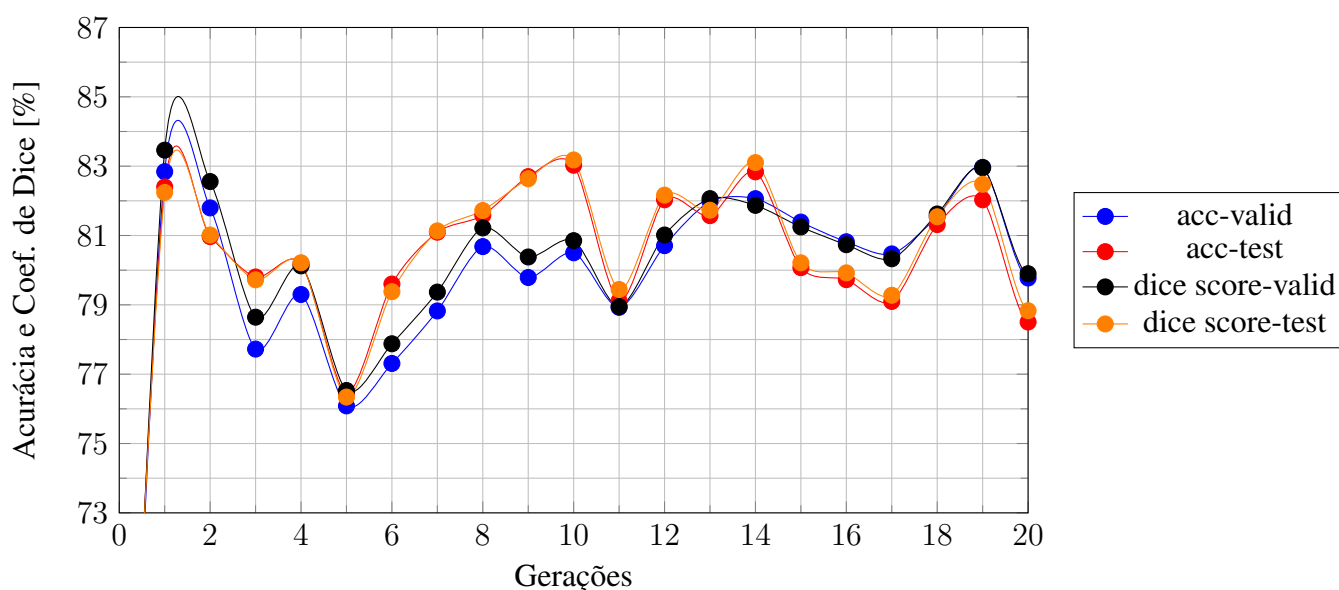
No primeiro treinamento Figura 6, foi possível verificar que o modelo não atinge métricas semelhantes às descritas por Kolbinger et al. (2022). Ademais, os valores de acurácia e coeficiente de Dice também se mostraram menos instáveis (mais flutuantes) durante o treinamento, com relação aos outros modelos testados neste trabalho.

O Segundo treinamento Figura 7 exibiu um comportamento conforme o esperado, com valores de teste acima dos valores de validação e valores de acurácia maiores que os valores de dice. A rede se aproximou da estabilidade, porém ainda com uma acurácia abaixo do desejado. A alta variação entre os pontos pode indicar que a rede neural não está presa em um mínimo local, mas está explorando diferentes áreas em busca do mínimo global.

O terceiro treinamento (Figura 8) foi mais estável que o treinamento anterior, apresentando um comportamento dos coeficientes semelhante nas gerações finais do treinamento, com a rede neural encontrando um mínimo e estabilizando-se ao redor desse mínimo relativamente rápido. O treinamento seguinte (Figura 9) atingiu a estabilidade; porém, a diferença entre o valor de acurácia e o coeficiente de Dice foi relativamente maior, e a sua taxa de acerto foi baixa.

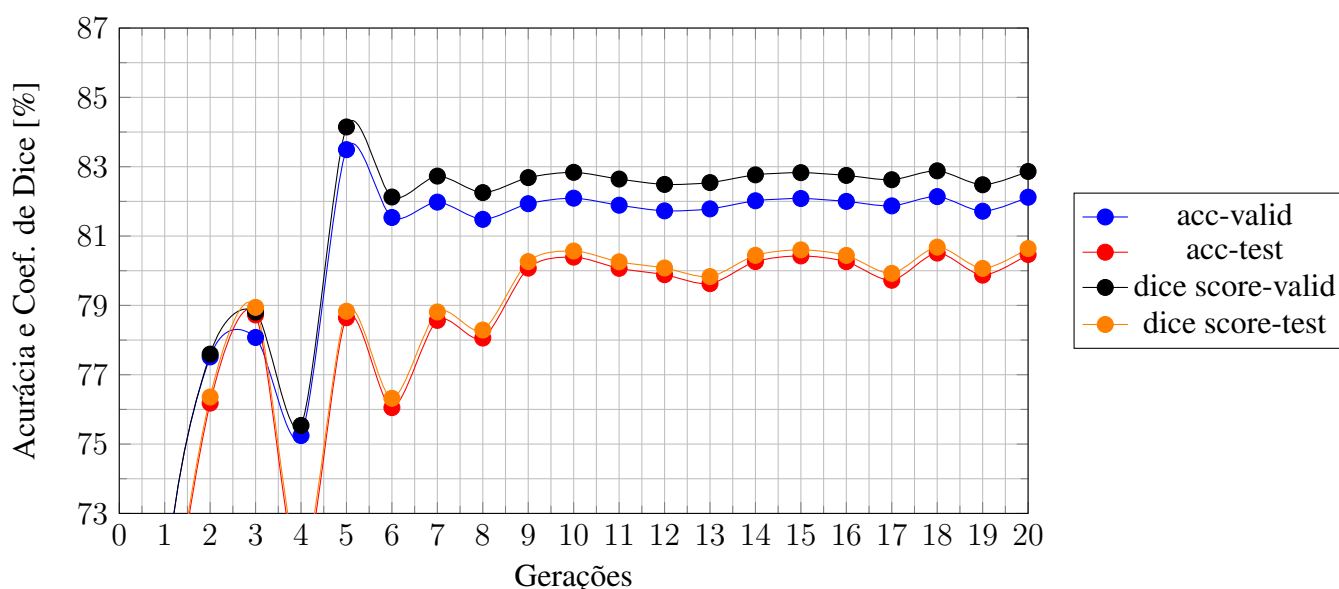
Assim como no terceiro, a rede neural artificial do quinto treinamento Figura 10 apresentou um treinamento estável, identificando um mínimo local, porém o mínimo ficou abaixo da media comparado

Figura 7 – Curvas de acurácia e pontuação dice do treinamento 2, com 8 imagens no lote, taxa de aprendizado de $1e-3$, UResNet de 18 camadas e função de perda L1Loss.



fonte: Produção do próprio autor.

Figura 8 – Curvas de acurácia e pontuação dice do treinamento 3, com 8 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 50 camadas e função de perda personalizada.



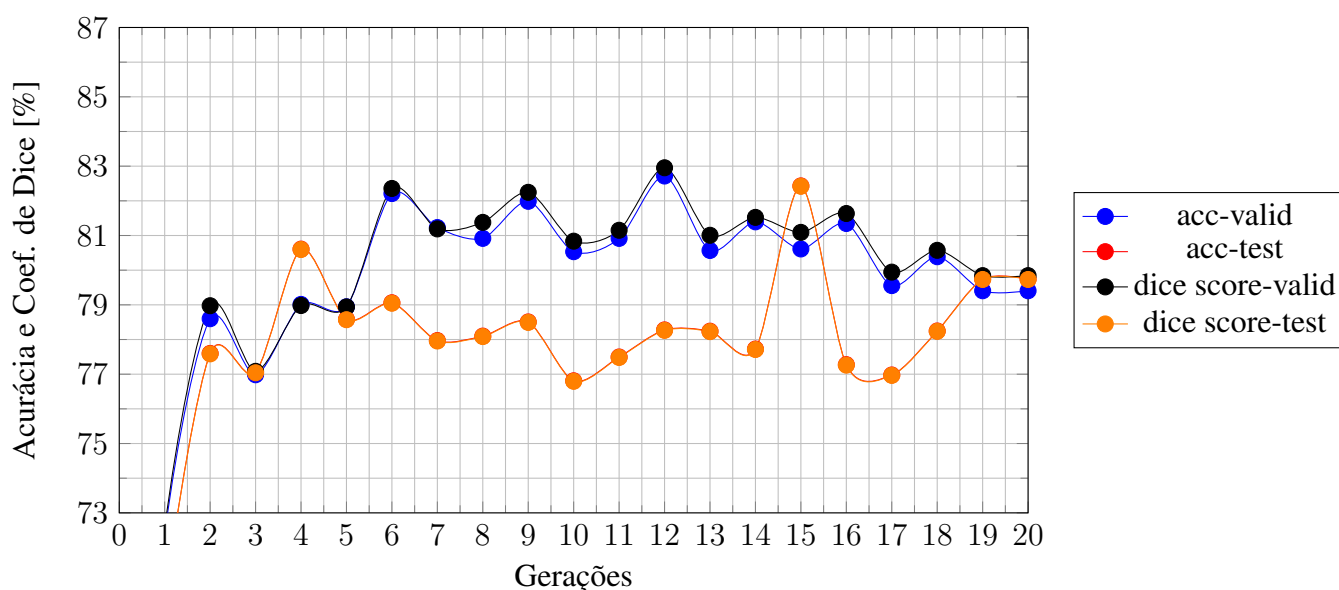
fonte: Produção do próprio autor.

com as outras redes.

O modelo obtido com o Treinamento 6 apresentou altas taxas de acerto, conforme se observa na Figura 11, superando todos os modelos treinados durante o presente trabalho. Porém, os altos valores de acurácia e coeficiente de Dice obtidos durante esse treinamento são provenientes do aprisionamento do modelo em um mínimo local durante o processo de otimização. Com a rede aprendendo bem em situações específicas como na Figura 15 e em outras a obter um valor maior através de falsos negativos, marcando grande parte da imagem como não sendo o tecido de interesse, como na Figura 16.

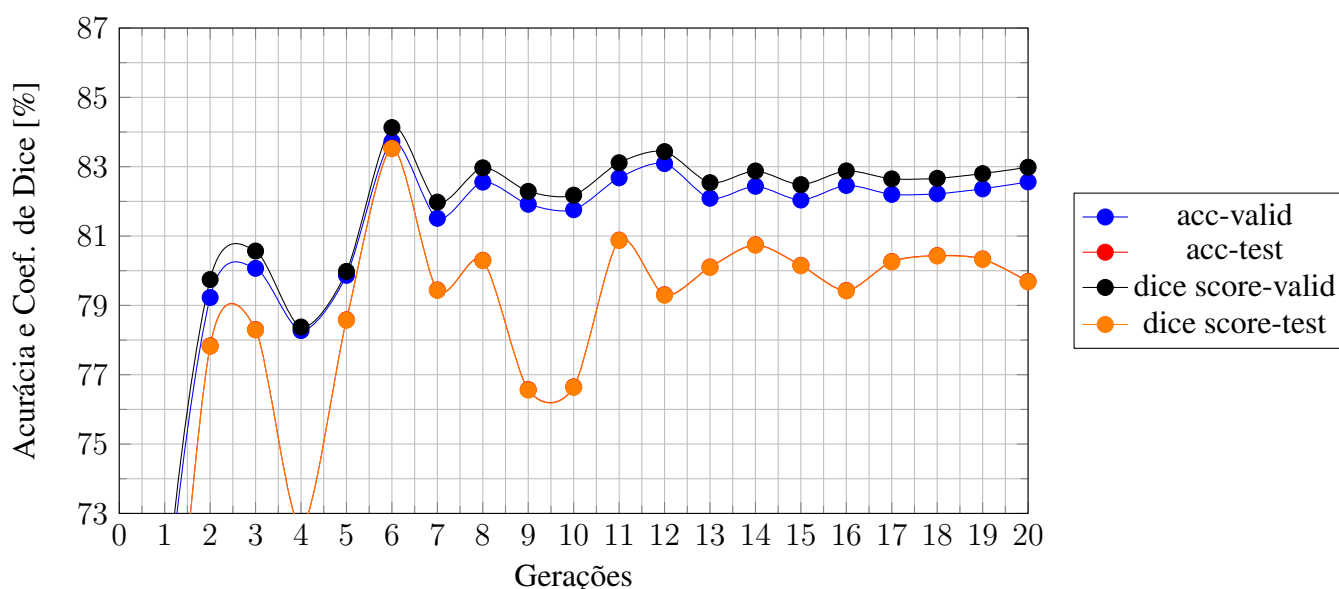
Esse comportamento ocorre porque foi mais fácil para o modelo neural artificial classificar toda a

Figura 9 – Curvas de acurácia e pontuação dice do treinamento 4, com 6 imagens no lote, taxa de aprendizado de $5e-4$, UResNet de 18 camadas e função de perda customizada.



fonte: Produção do próprio autor.

Figura 10 – Curvas de acurácia e pontuação dice do treinamento 5, com 6 imagens no lote, taxa de aprendizado de $1e-3$, UResNet de 50 camadas e função de perda CrossEntropyLoss.



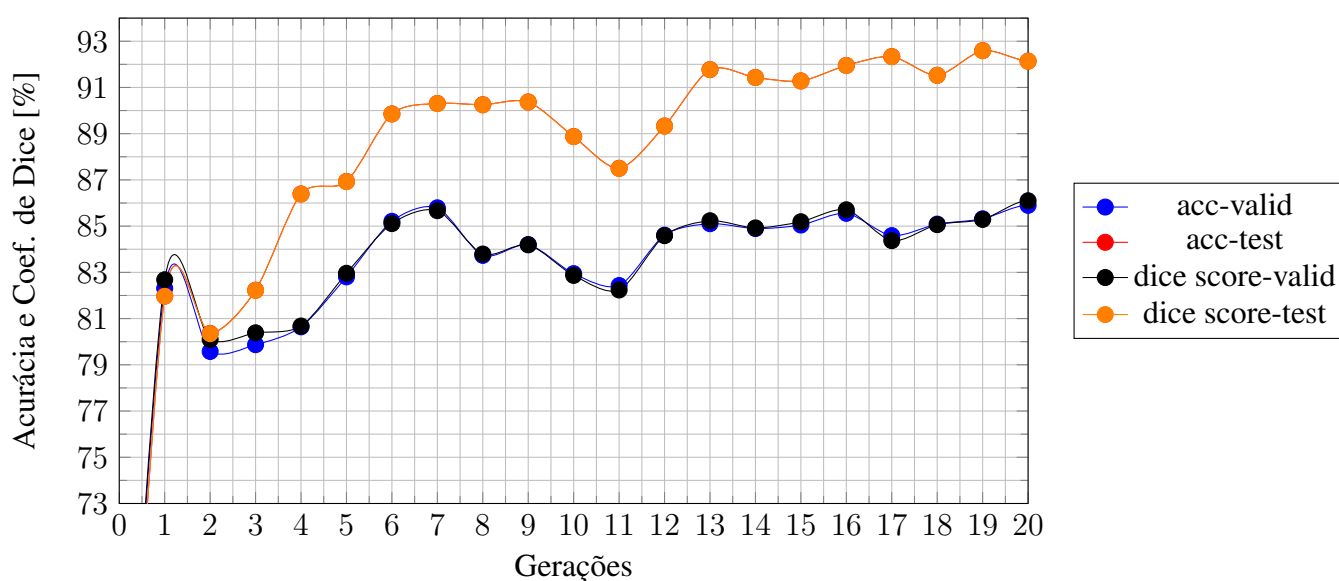
fonte: Produção do próprio autor.

imagem como negativo (preto), ou seja, sem a presença de tecido do fígado, do que tentar identificar corretamente o tecido de interesse. O método de otimização Adam permite que o modelo permaneça preso nesse mínimo local, já que classificar toda a imagem como fundo resulta em uma alta acurácia.

O Treinamento 7 apresentou altas taxas de acurácia e de coeficiente de Dice, conforme se pode observar pela Figura 12, comparado com os outros treinamentos realizados anteriormente. Porém, esse treinamento apresentou menor estabilidade em alguns pontos da rede, 12 e 18 principalmente, os valores de acerto estão próximos do procurado e a rede neural parece estar estabilizando.

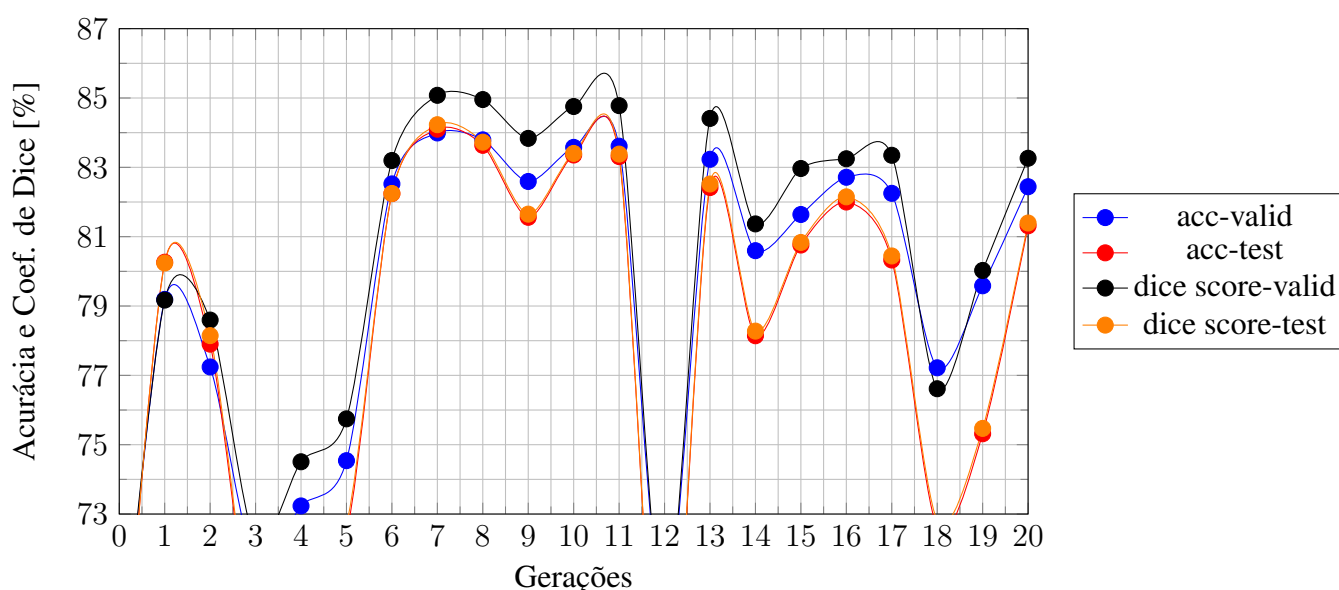
As taxas de acerto e coeficientes Dice calculados durante o treinamento 8 são mostrados na

Figura 11 – Curvas de acurácia e pontuação dice do treinamento 6, com 6 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 34 camadas e função de perda L1Loss.



fonte: Produção do próprio autor.

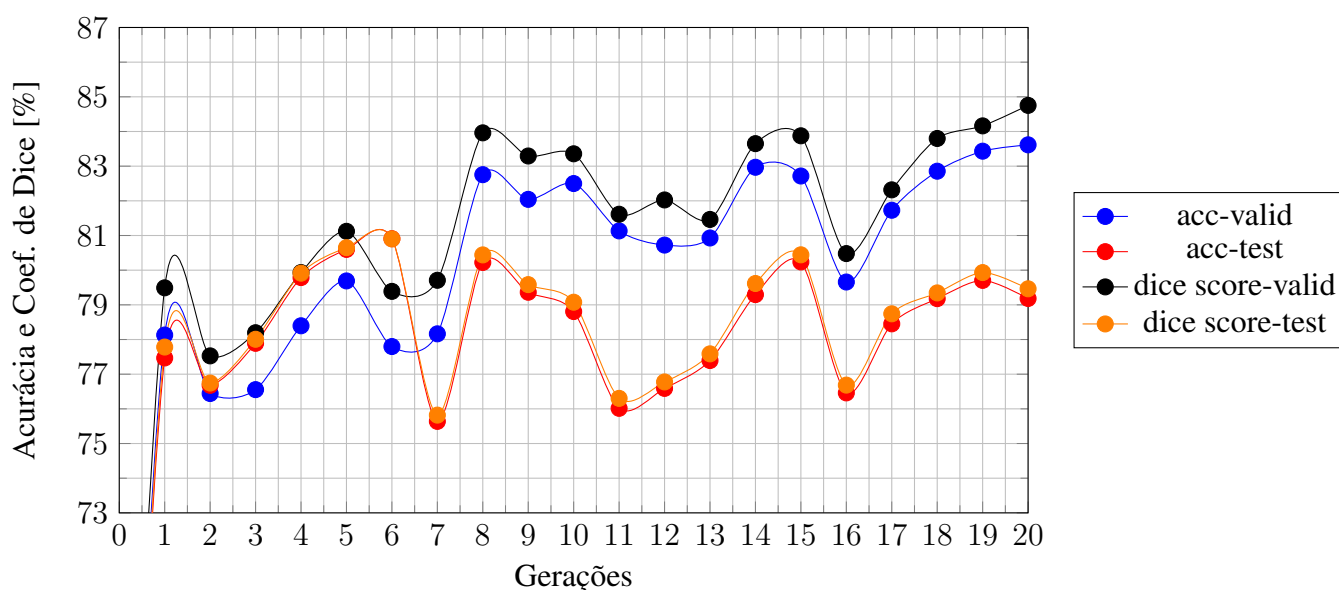
Figura 12 – Curvas de acurácia e pontuação dice do treinamento 7, com 12 imagens no lote, taxa de aprendizado de $5e-4$, UResNet de 50 camadas e função de perda L1Loss.



fonte: Produção do próprio autor.

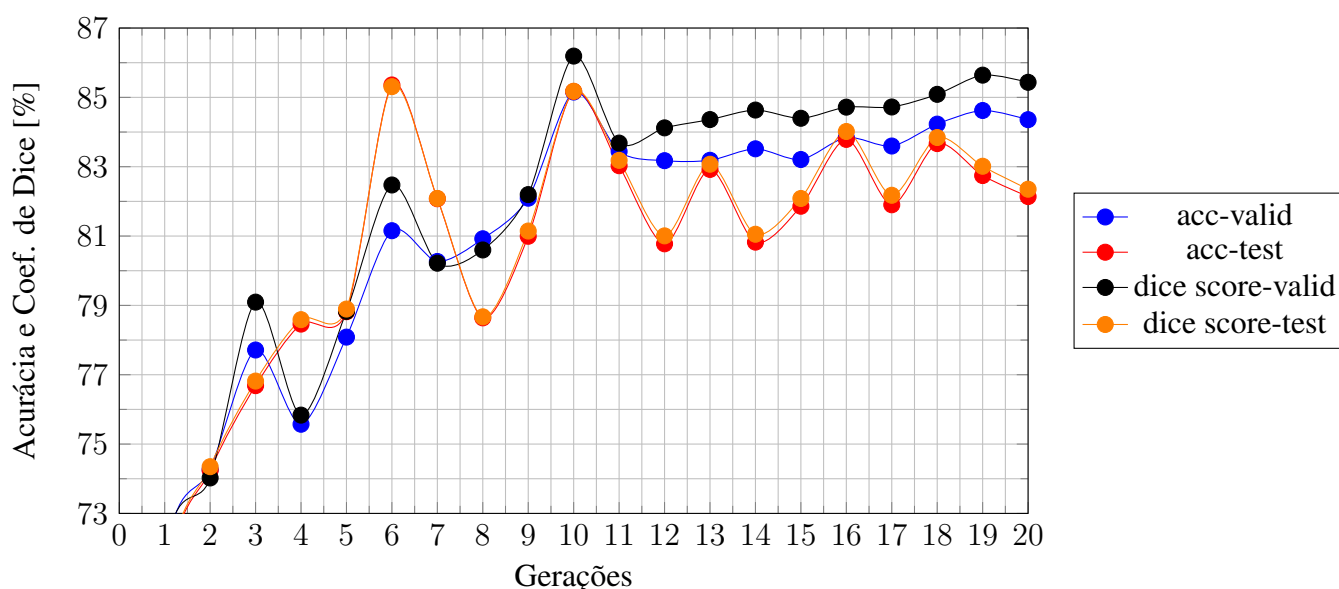
Figura 13, apresentando um comportamento de crescimento da taxa de acerto e um equilíbrio adequado entre os conjuntos de teste e validação, com alguma variação entre os resultados, o que indica que a rede não está presa em mínimos locais. Assim como o treinamento anterior, o treinamento 9 visto na Figura 14 suas curvas, apresentando um comportamento de crescimento da taxa de acerto, possuindo a segunda melhor acurácia, podendo ser vista nas figuras Figura 17 e Figura 18, são as mesmas figuras do treinamento 6 para comparação. No entanto, ainda possui uma leve instabilidade no valor de teste, o que indica que a rede neural ainda não está totalmente estabilizada e que ainda não alcançou um mínimo local.

Figura 13 – Curvas de acurácia e pontuação dice do treinamento 8, com 12 imagens no lote, taxa de aprendizado de $1e-3$, UResNet de 34 camadas e função de perda customizada.



fonte: Produção do próprio autor.

Figura 14 – Curvas de acurácia e pontuação dice do treinamento 9, com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 18 camadas e função de perda CrossEntropyLoss.

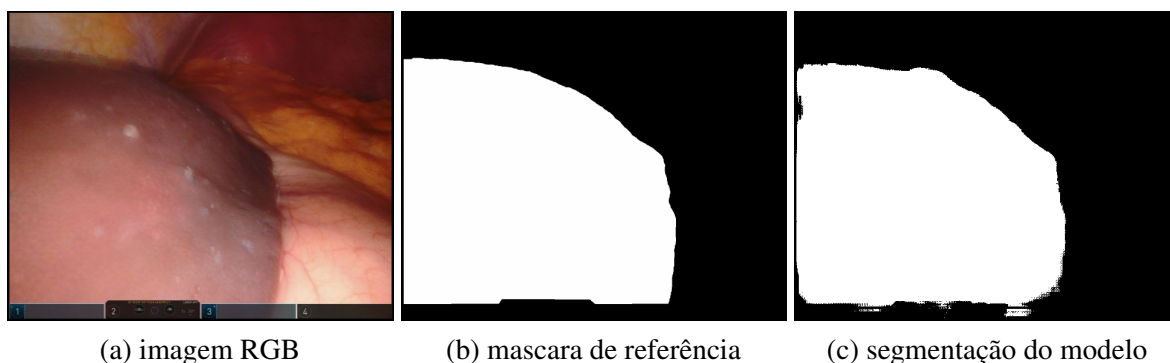


fonte: Produção do próprio autor.

Em geral, os melhores modelos de inteligência artificial alcançaram cerca de 84% de acurácia. Os resultados dos melhores valores de Dice e acurácia de cada modelo estão apresentados na Tabela 4. Em todos os treinamentos, as redes neurais apresentaram um crescimento rápido nas primeiras gerações. Na Tabela 5, os coeficientes de acurácia e Dice são apresentados, respectivamente. Foi possível notar que o modelo de 34 camadas foi superior; porém, o modelo de 50 camadas pode não ter tido gerações suficientes para alcançar seu melhor resultado. As funções de perda `L1Loss` e `CrossEntropyLoss` tiveram os melhores desempenhos. Também foi notável que a taxa de aprendizado menor demonstrou ser superior às taxas maiores, mas taxas de aprendizado menores

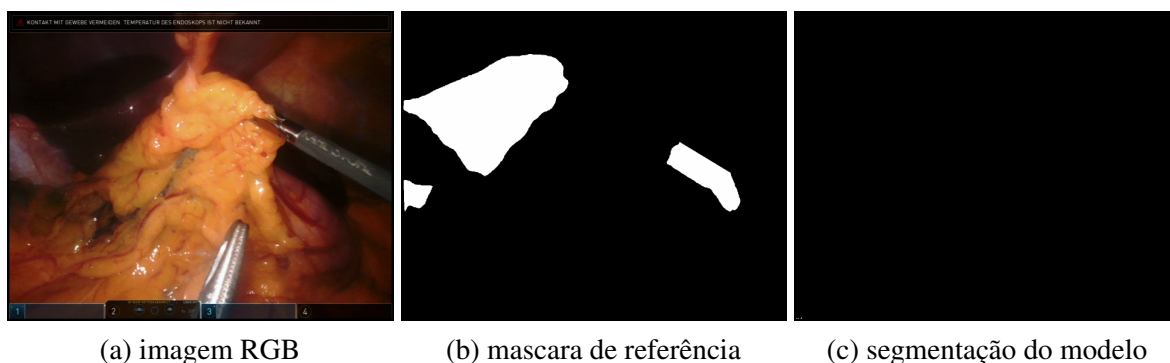
requerem mais épocas para atingir um plateau nos resultados. Os tamanhos de lote 12 e 6 foram melhores que o 8; contudo, foi importante considerar que um lote maior demanda mais recursos computacionais para o treinamento, embora possa ser mais rápido.

Figura 15 – Fígado ajustado corretamente.



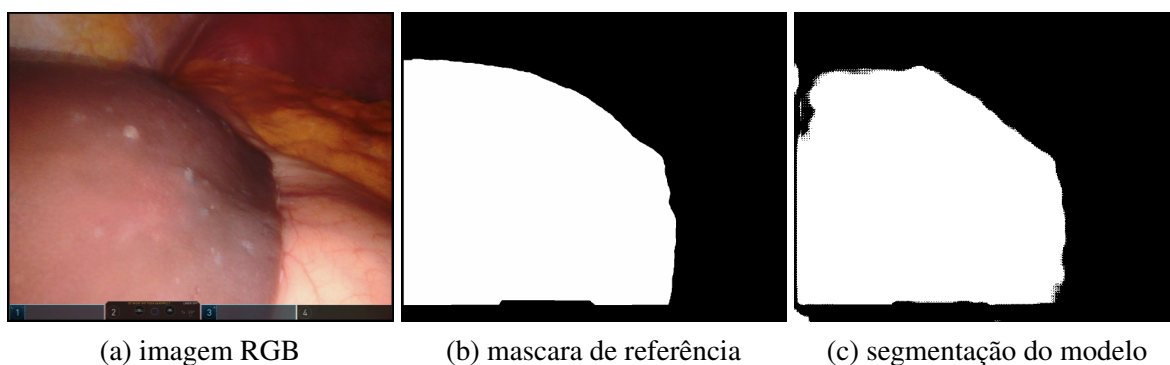
Fonte: Produção do próprio autor.

Figura 16 – Fígado ajustado erroneamente.



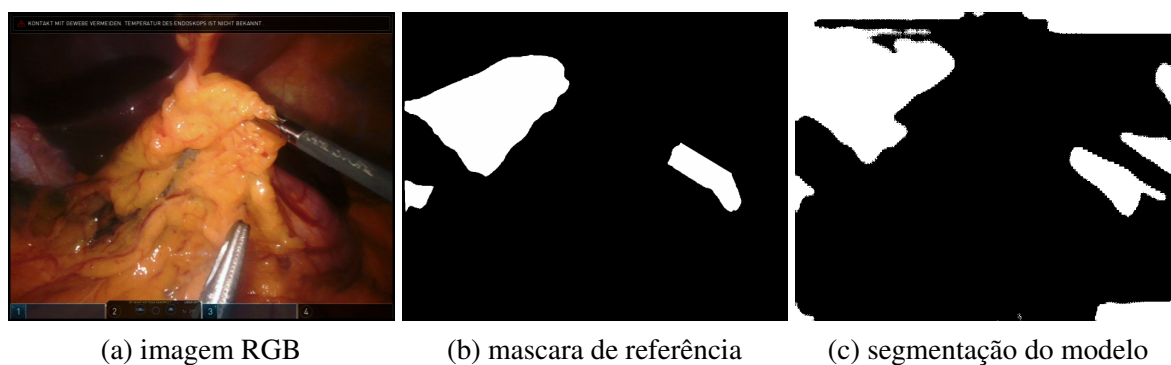
Fonte: Produção do próprio autor.

Figura 17 – Mascara do fígado da rede 9.



Fonte: Produção do próprio autor.

Figura 18 – Mascara do fígado da rede 9.



Fonte: Produção do próprio autor.

Tabela 4 – Tabela de melhores resultados de acurácia e Dice dos treinamentos.

Treinamento	Lote	Taxa de Apr	Modelo	Funções de Perda	Acurácia	Valor Dice
1	8	5e-4	34	CrossEntropyLoss	85,76	82,76
2	8	1e-3	18	L1Loss	83,46	83,17
3	8	1e-4	50	CustomLoss	84,14	80,68
4	6	5e-4	18	CustomLoss	82,95	82,42
5	6	1e-3	50	CrossEntropyLoss	84,12	83,52
6	6	1e-4	34	L1Loss	86,09	92,58
7	12	5e-4	50	L1Loss	85,07	84,22
8	12	1e-3	34	CustomLoss	84,75	80,90
9	12	1e-4	18	CrossEntropyLoss	86,19	85,30

fonte: Produção do próprio autor.

Tabela 5 – Tabela de correlação de valores das macrovariáveis com a acurácia.

	Lote		Taxa de Apr		Modelo		Funções de Perda
8	84,45	5e-4	84,59	34	85,53	Cross	85,36
6	84,39	1e-3	84,11	18	84,20	Custom	83,95
12	85,34	1e-4	85,47	50	84,45	L1	84,87

fonte: Produção do próprio autor.

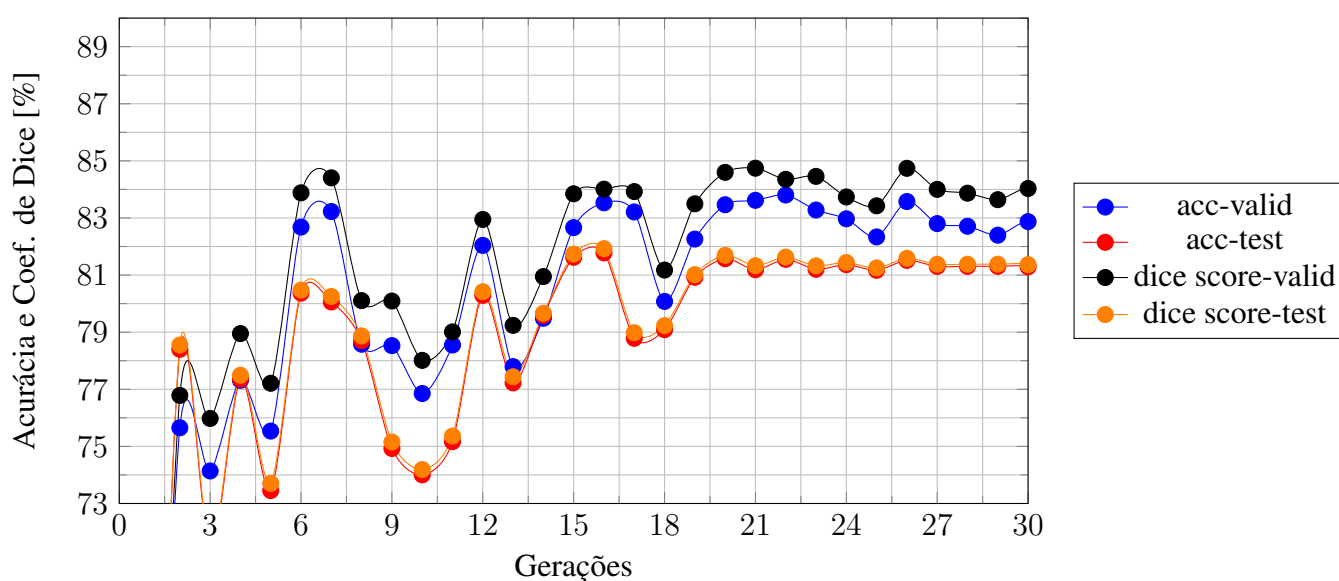
5.2 Segundo Treinamento

Para o segundo treinamento 3 redes foram feitas, os valores das macrovariáveis foram escolhidas com base na Tabela 5, todas com a mesma taxa de treinamento de 1e-4, mesma valor de lote de 12, com `CrossEntropyLoss` sendo feito com 34 e 50 camadas e `L1Loss` sendo feito com 50 camadas apenas. Todas as redes foram treinadas até 30 gerações e em seguida as redes de 50 camadas foram treinadas até 60 gerações.

O primeiro treinamento foi feito com uma rede de 34 camadas e função `CrossEntropyLoss`, pode ser visto na Figura 19, foi um modelo que teve um comportamento bom, porém chegou na estabilidade com um valor de acurácia menor que media das redes do primeiro treinamento, por esse motivo a rede não foi treinada por mais gerações

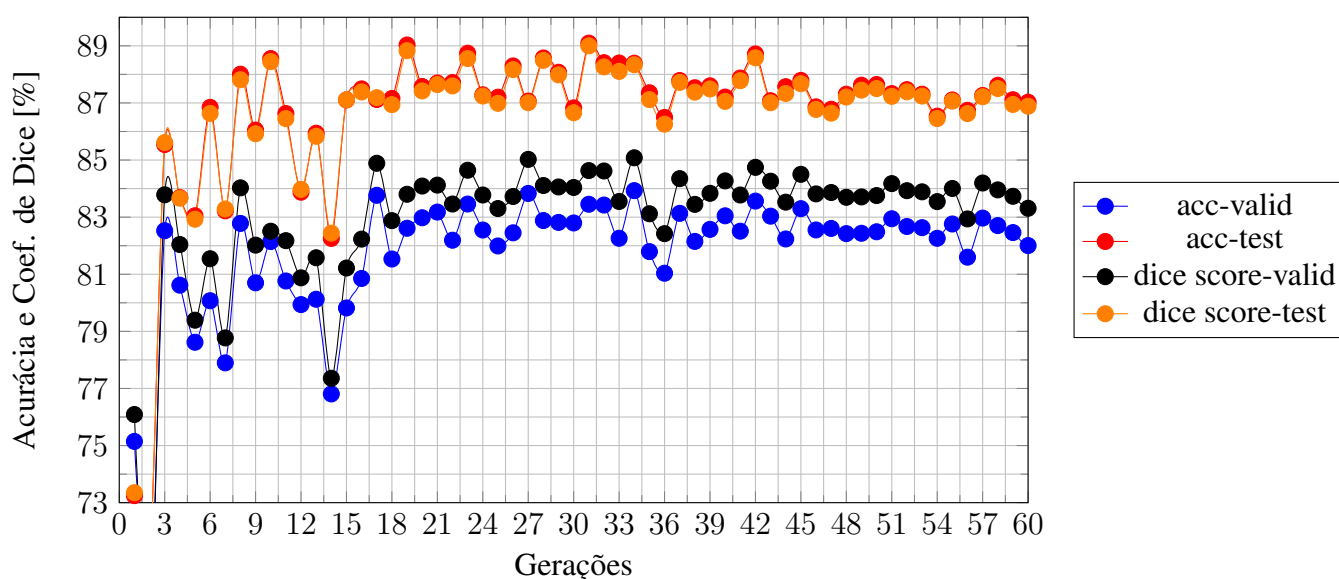
O treinamento Figura 20 teve o mesmo problema da rede 6, mas em um grau menor. Apesar do

Figura 19 – Treinamento com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 34 camadas e função de perda CrossEntropyLoss.



fonte: Produção do próprio autor.

Figura 20 – Treinamento com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 50 camadas e função de perda CrossEntropyLoss.



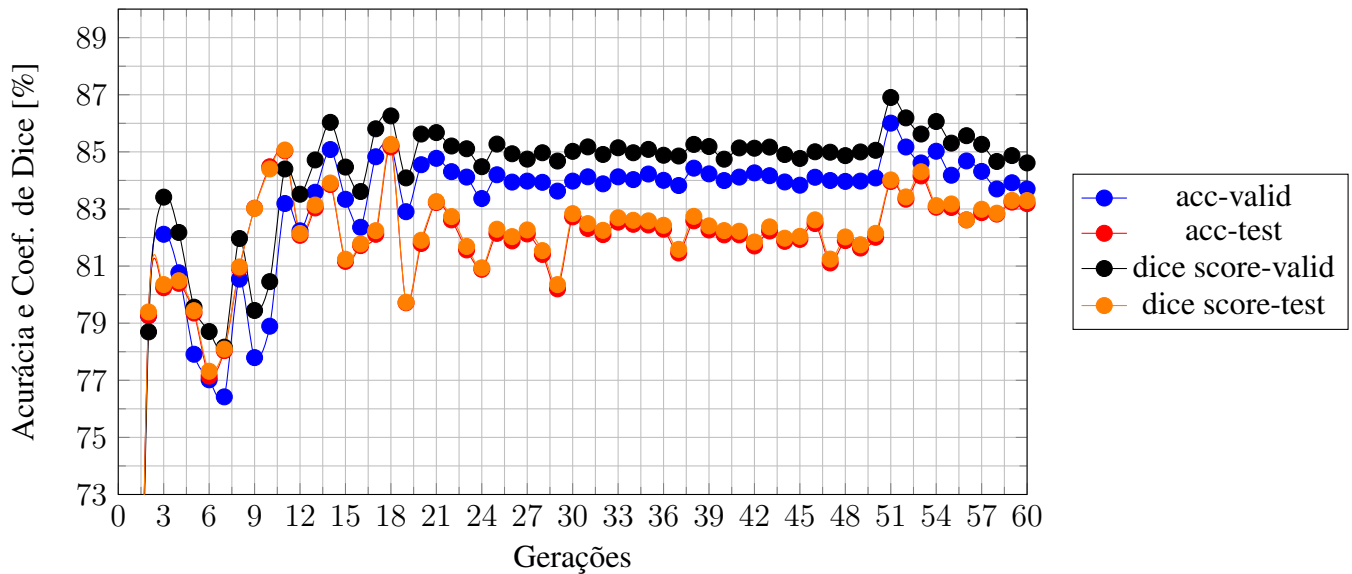
fonte: Produção do próprio autor.

aprisionamento ser inferior ele ainda ocorreu, tornando a rede menos desejável, porem mesmo com isso a rede teve uma boa capacidade de generalização e taxas de acurácia superiores a rede de 34 camadas, por esse motivo a rede teve seu treinamento estendido para 60 gerações, o que não resultou em melhora.

Por fim o treinamento Figura 21 obteve o segundo melhor resultado entre todos, mas não apresentou o problema de aprisionamento, alcançando 85% de acerto com bastante estabilidade, por esse motivo o treinamento foi estendido para 60 gerações, gerando valor máximo de pontuação Dice de 87%.

O treinamento das redes mostrou um aumento de cerca de 2% em relação às redes do primeiro treinamento, com o melhor resultado alcançando 87% de acerto tanto no segundo e terceiro modelo neural

Figura 21 – Treinamento com 12 imagens no lote, taxa de aprendizado de $1e-4$, UResNet de 50 camadas e função de perda L1Loss.



fonte: Produção do próprio autor.

artificial. Portanto, foi possível obter resultados comparáveis aos apresentados por (KOLBINGER et al., 2022).

5.2.1 Máscaras de Segmentação

Com os resultados do treinamento das redes de 50 camadas sendo superiores aos da rede neural de 34 camadas, as máscaras dos modelos de 50 camadas foram geradas e suas imagens obtidas. As imagens escolhidas representam diferentes situações e momentos da laparoscopia, como tecido bem definido, fora de foco, obstruído, mal iluminado e outras condições, demonstrando as dificuldades enfrentadas em encontrar modelos de redes neurais capazes de manter alta acurácia em todas as situações.

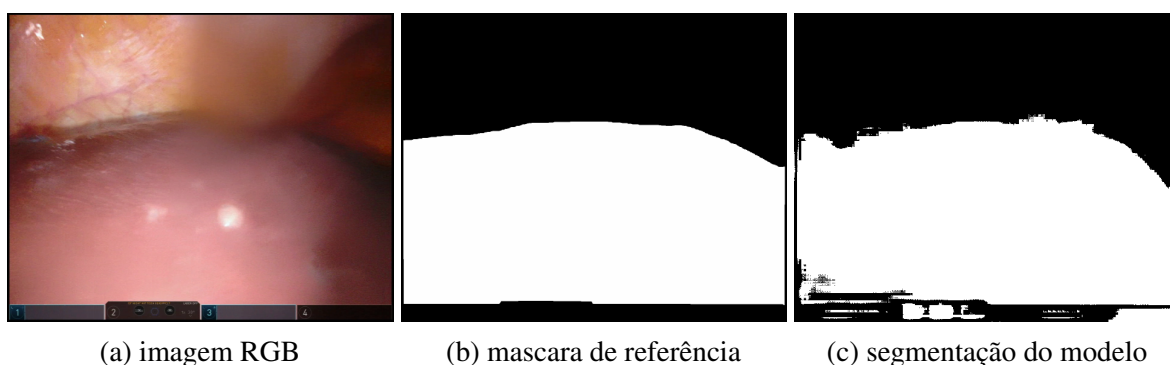
Cada figura foi composta primeiramente pela máscara do DSAD, a máscara da rede neural artificial e a imagem real. Todas as imagens encontram-se na terceira pasta de fígados, com o número da imagem sendo igual ao encontrado na pasta do DSAD.

Para o modelo neural artificial com a função de perda `CrossEntropyLoss`, a máscara resultante gerou as figuras: Figura 22, Figura 23, Figura 24 e Figura 25. As imagens mostram que apesar do aprisionamento ainda foi capaz de certa generalização, com as figuras em geral conseguindo mostrar o contorno do tecido em interesse ou grande parte do mesmo.

Com a rede utilizando o método L1 para o cálculo de acurácia, foram obtidas as figuras: Figura 26, Figura 27, Figura 28 e Figura 29. As figuras demonstram uma boa generalização, conseguindo contornar o tecido de maneira precisa e tendendo a ser conservativa na segmentação, preferindo marcar a imagem como não contendo o tecido quando há incerteza. As implicações disso para a qualidade e segurança do procedimento cirúrgico são significativas. Uma segmentação conservativa

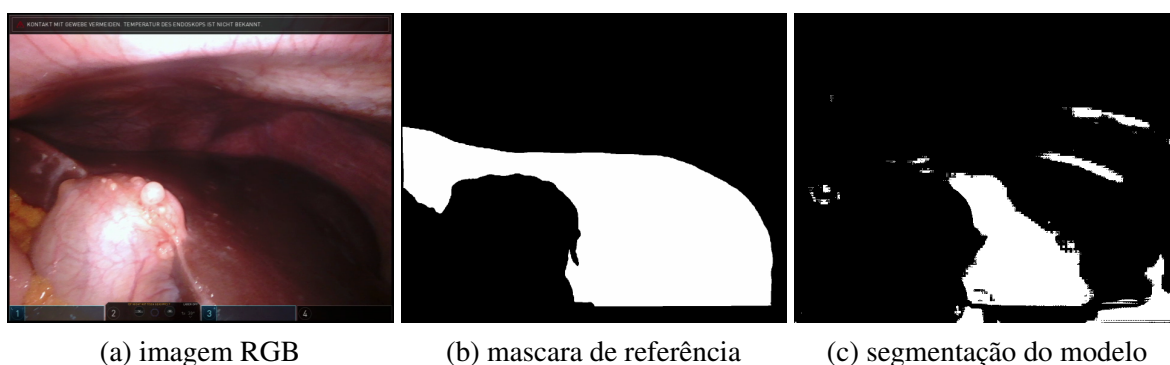
reduz a probabilidade de falsos positivos, minimizando o risco de intervenção em áreas saudáveis e potencialmente evitando danos desnecessários aos tecidos circundantes.

Figura 22 – Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, de região tecidual fora de foco, porém com bom contraste e iluminação.



Fonte: Produção do próprio autor.

Figura 23 – Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, de região tecidual em foco e contraste mas com iluminação imperfeita.



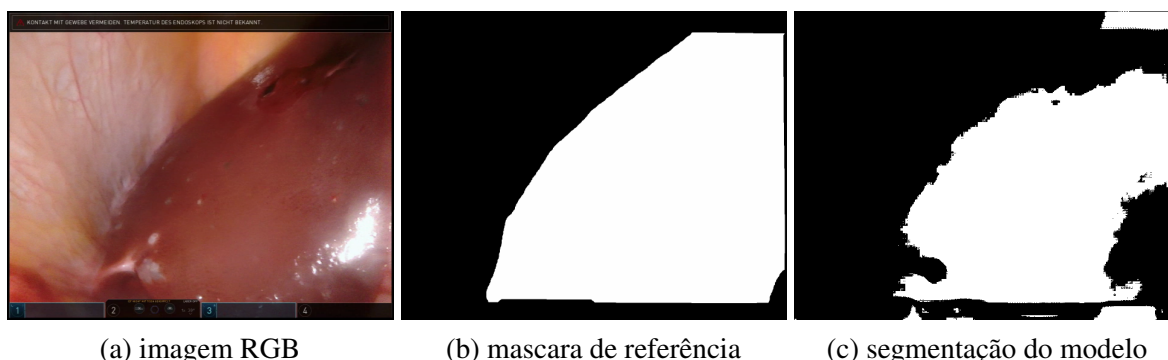
Fonte: Produção do próprio autor.

A Figura 22 mostra a capacidade do modelo neural artificial identificar o fígado onde o mesmo esteja fora do foco, possuindo pouco ruído na borda do tecido, possivelmente causado pela sombra e falta de foco. Na Figura 23 o tecido está em foco porém está na sombra, a rede neural foi capaz de encontrar parte das bordas do fígado, mas sofreu com a imperfeição da iluminação, o que pode ocorrer com frequência durante endoscopias, confundindo uma parte mais escura da imagem com o objeto de estudo.

Em casos com muita luz como na Figura 24 a máscara pode confundir uma parte com excesso de luz (branca) como não sendo parte do tecido, assim como partes onde o contraste muda muito, como os pontos pretos na parte superior da imagem ou na parte inferior com o tecido branco na frente do fígado. O último exemplo foi a Figura 25 onde havia pouco tecido de interesse e uma ferramenta cirúrgica, mesmo assim a rede conseguiu identificar a presença do fígado no canto superior esquerdo.

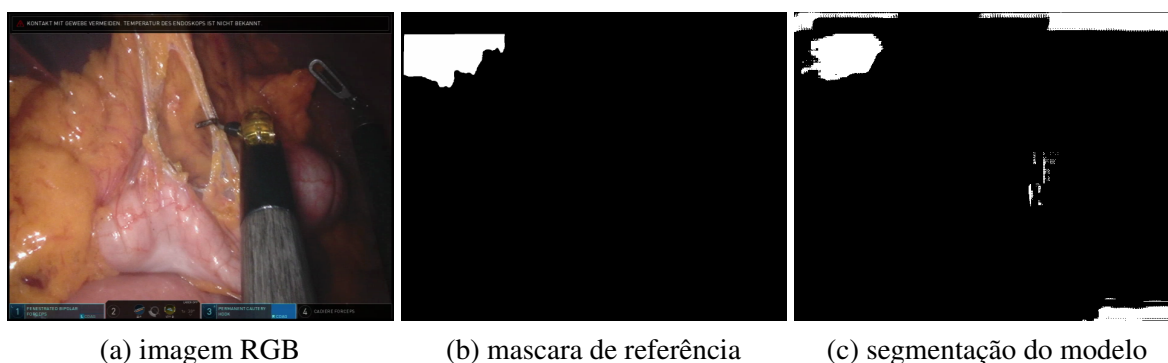
Na Figura 26, foi possível perceber ruído na identificação do fígado, possivelmente causado pela imagem possuir sombras e não estar focada perfeitamente no tecido, resultando em uma imagem com bordas pouco definidas e com variação de cor e luminosidade. A rede possivelmente escolheu

Figura 24 – Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, do fígado um pouco fora de foco com irregularidade no canto superior direito e excesso de iluminação no canto inferior direito.



Fonte: Produção do próprio autor.

Figura 25 – Segmentação apresentada pelo modelo do Treinamento de CrossEntropyLoss, do fígado somente no canto superior esquerdo com má iluminação e foco mas bom contraste.



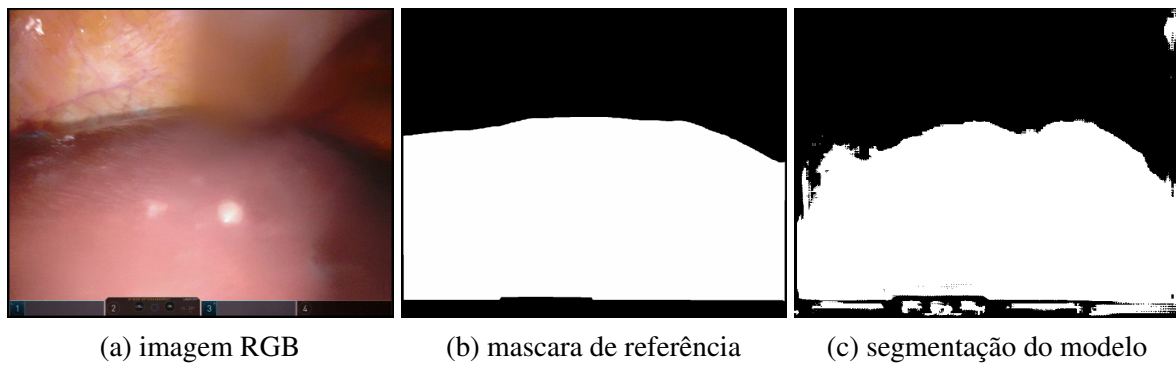
Fonte: Produção do próprio autor.

segmentar as partes com mais certeza e excluir as partes com dúvida. Na Figura 27, as dificuldades são parecidas, mas a garra aparece por cima do tecido. O modelo neural foi capaz de notar a garra, mas se perdeu com a coloração do fígado mudando devido ao sombreamento e também identificou erroneamente a borda superior direita provavelmente por conta da sombra mudando o tom nesse local.

A Figura 28 mostra o fígado com posição, formato e cor diferentes devido a iluminação diferente, possuindo o omento passando no meio do fígado. Já a Figura 29 representa um caso em que não é possível definir facilmente a borda do órgão por estar na sombra. A Figura 29 teve uma boa segmentação, apesar da dificuldade em definir onde o tecido termina e a sombra começa, além das mudanças de cor causadas pela iluminação no fígado. Já a Figura 28 enfrentou mais dificuldades devido à qualidade da imagem, conseguindo segmentar a borda do fígado, mas não o fígado inteiro.

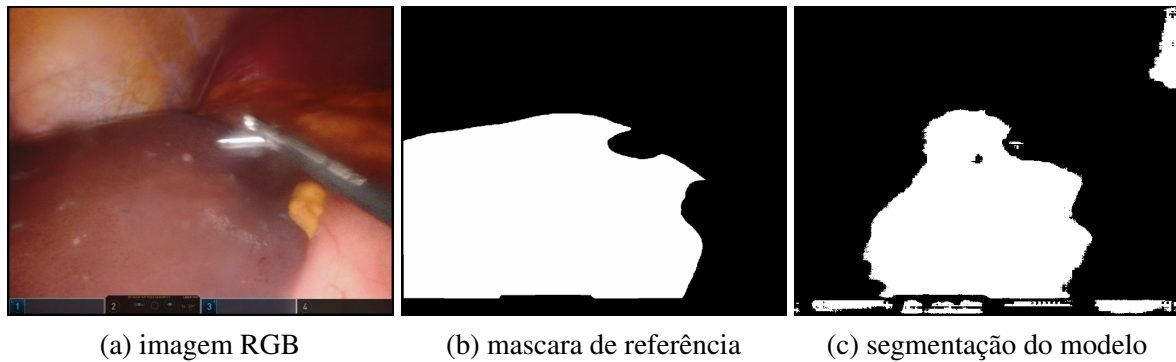
Em todas as imagens, ocorreram ruídos. Nas imagens menos definidas, como mostrado na Figura 22, Figura 26, Figura 24 e Figura 27, o ruído ocorreu na borda. Em imagens mais complexas e bem definidas, como as figuras Figura 28, Figura 25, Figura 23 e Figura 29, o ruído ocorreu no interior do tecido e nas bordas, onde houve mudanças de tons devido à iluminação. Em todos os casos, as barras inferior e superior, criadas pelo equipamento de operação, foram confundidas. Isso pode ser porque elas aparecem em todas as fotos e a rede aprendeu a reconhecê-las, mesmo que não seja o objetivo da

Figura 26 – Segmentação apresentada pelo modelo do Treinamento de L1Loss, de região tecidual levemente fora de foco, com bom contraste e iluminação.



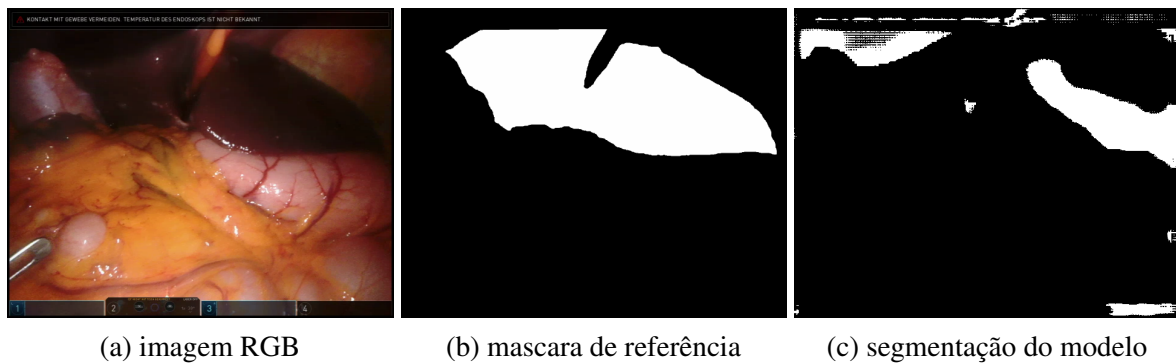
Fonte: Produção do próprio autor.

Figura 27 – Segmentação apresentada pelo modelo do Treinamento de L1Loss, de região tecidual fora de foco e com bordas embaçadas.



Fonte: Produção do próprio autor.

Figura 28 – Segmentação apresentada pelo modelo do Treinamento de L1Loss, de região tecidual em foco mas com iluminação imperfeita e contraste ruim nos cantos.



Fonte: Produção do próprio autor.

segmentação.

Figura 29 – Segmentação apresentada pelo modelo do Treinamento de L1Loss, do fígado com má iluminação e contraste.



(a) imagem RGB

(b) mascara de referência

(c) segmentação do modelo

Fonte: Produção do próprio autor.

6 CONCLUSÃO

Neste trabalho, foram alcançados resultados comparáveis à pesquisa original do banco de dados DSAD. Todos os modelos treinados apresentaram um crescimento rápido na acurácia nas primeiras gerações, obtendo uma acurácia de pelo menos 82% para a segmentação de imagens do fígado. Esse resultado demonstra que os modelos não tiveram dificuldades no treinamento, mesmo com o número relativamente baixo de imagens do conjunto de dados em comparação a outras aplicações que utilizam centenas de milhares de imagens. Ainda há poucas pesquisas sobre segmentação dos tecidos e suas regiões, com Kolbinger et al. (2022) sendo atualmente a única pesquisa encontrada na revisão de literatura.

Os objetivos principais do trabalho foram atingidos, aprofundando o entendimento de Python e das bibliotecas PyTorch e Torchvision, bem como do funcionamento e da construção de redes neurais. O estudo proporcionou uma compreensão detalhada dos desafios associados ao treinamento de redes neurais artificiais para a segmentação em geral e, especificamente, a de tecidos humanos em cirurgias de laparoscopia.

As limitações de processamento enfrentadas levaram ao uso do Google Colab e à necessidade de um mês para treinar certas redes. Além disso, a dificuldade de treinar segmentações em redes profundas com menos de mil imagens de ótima qualidade disponíveis, com várias delas apresentando defeitos como fumaça, desfoque ou mau contraste, foi um grande obstáculo. O DSAD foi o maior banco de dados disponível no momento do trabalho. Para o futuro, um tratamento mais robusto das imagens e o uso de redes mais profundas, treinadas em um computador mais potente, podem trazer resultados superiores. Alternativamente, a exploração de novas técnicas de modelo de inteligência artificial pode oferecer avanços.

A tecnologia de segmentação de imagens médicas está em evolução e, em poucos anos, poderá se tornar uma ferramenta comum no cotidiano médico. Essa tecnologia tem o potencial de auxiliar médicos na identificação de tecidos e ferramentas cirúrgicas durante exames e cirurgias, contribuindo para procedimentos mais precisos e seguros.

Em suma, este trabalho não apenas atingiu seus objetivos técnicos e acadêmicos, mas também destacou o potencial futuro da segmentação automatizada em imagens médicas, abrindo caminho para inovações que podem revolucionar o campo da medicina.

Todo o código desenvolvido está disponível publicamente em um repositório no GitHub implementado em quatro arquivos distintos: `model.py`, `train.py`, `utils.py` e `dataset.py`.

REFERÊNCIAS

- ALMEIDA FELIPE CARVALHO, F. M. A. **Introdução ao Machine Learning**. 2024. Disponível em: <<https://dataat.github.io/introducao-ao-machine-learning/regressÃo.html#regressÃo-linear>>.
- CHOKSI, S. et al. Bringing artificial intelligence to the operating room: edge computing for real-time surgical phase recognition. **Surgical Endoscopy**, v. 37, p. 8778–8784, November 2023. Received: 10 April 2023; Accepted: 19 July 2023; Published: 14 August 2023.
- ESTEVA, A. et al. A guide to deep learning in healthcare. **Nature Medicine**, v. 25, n. 24, p. 24–29, 2019.
- HARDY, N. P. et al. Intraprocedural artificial intelligence for colorectal cancer detection and characterisation in endoscopy and laparoscopy. **Surgical Innovation**, v. 26, n. 8, p. 768–775, 2021.
- HE, K. et al. Deep residual learning for image recognition. **2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)**, p. 770–778, 2015. Disponível em: <<https://api.semanticscholar.org/CorpusID:206594692>>.
- KITAGUCHI, D.; ITO, M. Computer vision in colorectal surgery: Current status and future challenges. **Seminars in Colon and Rectal Surgery**, p. 101008, 2024. ISSN 1043-1489. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S1043148924000071>>.
- KOLBINGER, F. R. et al. Artificial intelligence for context-aware surgical guidance in complex robot-assisted oncological procedures: An exploratory feasibility study. **European Journal of Surgical Oncology (EJSO)**, July 2023. Article info: Published online: July 27, 2023; Accepted: July 26, 2023; Received in revised form: June 20, 2023; Received: March 13, 2023; Publication stage: In Press Journal Pre-Proof. Copyright: © 2023 The Authors. Published by Elsevier Ltd.
- KOLBINGER, F. R. et al. Artificial intelligence for context-aware surgical guidance in complex robot-assisted oncological procedures: An exploratory feasibility study. **European Journal of Surgical Oncology (EJSO)**, [Publisher Name], [Volume Number], n. [Issue Number], p. [Page Numbers], 2023. Affiliations: Department of Visceral, Thoracic and Vascular Surgery, University Hospital and Faculty of Medicine Carl Gustav Carus, Technische Universität Dresden, and others.
- KOLBINGER, F. R. et al. Anatomy segmentation in laparoscopic surgery: Comparison of machine learning and human expertise – an experimental study. **International Journal of Surgery**, 2022. Disponível em: <<https://doi.org/10.1097/JS9.0000000000000595>>.
- LAZIC, L. Use of orthogonal arrays and design of experiments via taguchi methods in software testing. In: . [s.n.], 2013. Disponível em: <<https://api.semanticscholar.org/CorpusID:14219992>>.
- MAIER-HEIN, L. et al. Surgical data science – from concepts toward clinical translation. **Medical Image Analysis**, v. 76, p. 102306, 2022. ISSN 1361-8415. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S1361841521003510>>.
- MASCAGNI, P.; ALAPATT, D.; SESTINI, L. e. a. Computer vision in surgery: from potential to clinical value. **npj Digit. Med.**, Nature Portfolio, v. 5, p. 163, 2022. Disponível em: <<https://doi.org/10.1038/s41746-022-00707-5>>.
- OLIPHANT, T. E. Python for scientific computing. **Computing in science & engineering**, IEEE, v. 9, n. 3, p. 10–20, 2007.

PEDREGOSA, F. et al. Scikit-learn: Machine learning in python. **the Journal of machine Learning research**, JMLR. org, v. 12, p. 2825–2830, 2011.

PYTORCH. 2024. Disponível em: <<https://www.nvidia.com/en-us/glossary/pytorch/>>.

TOMAR, N. **What is U-Net?** 2021. Disponível em: <<https://medium.com/analytics-vidhya/what-is-unet-157314c87634>>.

YANI, M.; IRAWAN, S.; SETIANINGSIH, C. Application of transfer learning using convolutional neural network method for early detection of terry's nail. **Journal of Physics: Conference Series**, v. 1201, p. 012052, 05 2019.