Universidade Federal do ABC

Trabalho de Graduação em
Engenharia de Informação

Processamento de Imagens para AUV: Segmentação de Imagem para Identificação de Contornos

Robson Costa Santiago

Santo André, SP
2021

Robson Costa Santiago

Processamento de Imagens para AUV: Segmentação de Imagem para Identificação de Contornos

Monografia apresentada ao curso de Engenharia de Informação da Universidade Federal do ABC como parte dos requisitos para a obtenção do grau de Engenheiro de Informação.

UNIVERSIDADE FEDERAL DO ABC

Orientador:
Prof. Dr. Ricardo Suyama

Co-Orientador:
Prof. Dr. Magno Enrique Mendoza Meza

Santo André, SP

2021

Sumário

Lista de Figuras IV

Lista de Tabelas VII

Lista de Abreviações VIII

1 Introdução 3

1.1 Visão Computacional . . . 3

1.2 ROVs e AUVs . . . 6

1.3 Processamento de Imagens Subaquáticas . . . 7

1.4 Objetivo Geral . . . 10

1.4.1 Objetivos Específicos . . . 10

2 Fundamentação Teórica 11

2.1 Segmentação de Imagem . . . 11

2.2 Rede Neural Convolucional . . . 16

2.2.1 Convolução . . . 17

2.2.2 ReLU (*Rectified Linear Unit*) . . . 19

2.2.3 *Pooling* . . . 19

2.2.4 Treinamento . . . 20

2.2.5 CNN em Segmentação de Imagens . . . 23

2.2.6 *Data Augmentation* . . . 26

2.3 Segmentação por Clusterização *K-Means* . . . 27

3 Metodologia e Implementação 29

3.1 Implementação . . . 31

3.1.1 ENet 31
3.1.2 K-Means 40

4 Resultados e Discussão 42

4.1 Rede Neural 42
4.1.1 Tempo de Processamento 51
4.2 K-Means 51
4.2.1 Tempo de Processamento 53

5 Conclusão 54

Lista de Figuras

1.1 Detecção de bordas para segmentação de imagem. 5
1.2 Exemplo de aplicação da visão computacional, o reconhecimento facial, capaz de identificar rostos nas imagens (SZELISKI, 2011). 5
1.3 Exemplos de *designs* para ROVs e AUVs. Fonte: ⟨https://www.researchgate.net/figure/Different-kinds-of-AUVs-and-ROVs-106_fig3_332889842⟩. 6
1.4 Metodologia de segmentação de imagem subaquática proposta por Yu et al. (2019). 7
1.5 Diagrama representativo dos diferentes fenômenos atuantes na propagação da luz no oceano. Fonte: Adaptado de Drews-Jr et al. (2016). 8
1.6 Conjunto de imagens antes e após a aplicação de métodos propostos por diversos autores para restauração de imagens subaquáticas. Fonte: Zhang et al. (2019). . 9

2.1 Exemplo de aplicação da segmentação semântica e da segmentação de instância. Nota-se que o segundo método se preocupa em separar objetos dentro dos grupos pré-definidos, nesse caso, de cadeiras. Fonte: ⟨https://towardsdatascience.com/review-deepmask-instance-segmentation-30327a072339⟩. 12
2.2 Desenho ilustrando o princípio que governa a segmentação *Watershed*. Fonte: (BALA, 2012). 14
2.3 Diagrama de detetor de borda por rede neural. Fonte: (TERRY; VU, 1993). . . . 14
2.4 Rede Neural Profunda, demonstra a ideia básica do *Deep Learning*. Fonte: ⟨https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351⟩. 16
2.5 Exemplo de processamento de imagem utilizando CNN. Fonte: ⟨https://towardsdatascience.com/introducing-convolutional-neural-networks-in-deep-learning-400f9c3ad5e9⟩. . 17

2.6 Representação da estrutura matricial para uma imagem em RGB. Fonte: ⟨https://www.researchgate.net/figure/A-three-dimensional-RGB-matrix-Each-layer-of-the-matrix-is-a-two-dimensional-matrix_fig6_267210444⟩. 18
2.7 Exemplo de convolução entre uma camada de entrada com dimensões $32 \times 32 \times 3$ e um filtro de dimensões $5 \times 5 \times 3$. Fonte: (AGGARWAL, 2018). 18
2.8 Exemplo de aplicação de uma operação de *max-pooling* com passos 1 e 2, demonstrando a diferença de tamanho entre as camadas resultantes de tal processo. Fonte: (AGGARWAL, 2018). 20
2.9 Diagrama que sintetiza as ideias dos diferentes tipos de aprendizado. Fonte: ⟨https://br.pinterest.com/pin/434597432787064636/⟩. 21
2.10 Representação topográfica do gradiente descendente. Fonte: (ZHOU, 2018). . . 22
2.11 Processo de *Backpropagation* simplificado. Fonte: ⟨https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199⟩. 23
2.12 Blocos estruturais da rede ENet. (a) o bloco inicial com *MaxPooling* de janelas não sobrepostas de 2×2 e convolução com 13 filtros, somando 16 mapas de atributos após concatenação. (b) "módulo de gargalo", segundo Paszke et al. (2016), com bloco `conv` podendo ser uma convolução regular, dilatada ou total com filtros 3×3 ou, ainda, convolução 5×5 decomposta em duas assimétricas. Fonte: Adaptado de (PASZKE et al., 2016). 25
2.13 Aplicação do ENet para segmentação de imagens de ruas. Fonte: Adaptado de (PASZKE et al., 2016). 26
2.14 Curva típica de comportamento de *overfitting*. Repare que o erro de treinamento tende a convergir em uma tendência de queda, enquanto que o erro de validação passa a subir. Fonte: Autoria própria. 27
2.15 Exemplo simples de clusterização por *K-Means*. Fonte: Autoria própria. 28

3.1 Exemplos de imagens e suas máscaras de segmentação *ground truth* em RGB. Fonte: ⟨http://irvlab.cs.umn.edu/resources/suim-dataset⟩. 30
3.2 Exemplo de par de imagem/máscara disponível no *SUIM Dataset*. Fonte: http://irvlab.cs.umn.edu/resources/suim-dataset. 31
3.3 Aproximação das máscaras pré (esquerda) e pós (direita) processamento de *threshold*. Pode-se notar a diferença de nitidez entre as bordas das máscaras, devido a interpolação dos *pixels* na vizinhança entre as classes. Fonte: Autoria própria. 32
3.4 Representação simplificada do conceito de taxa de aprendizado. Fonte: Autoria própria. 37

4.1 Curvas de perda e *MIoU* de treinamento e validação com *set* de validação enviesado. Fonte: Autoria própria. 43

4.2 Curvas de perda e *MIoU* de treinamento e validação, por duzentas épocas. Fonte: Autoria própria. 44

4.3 Curvas de perda e *MIoU* de treinamento e validação, por trezentas épocas. Fonte: Autoria própria. 44

4.4 Predição das imagens de teste pelo modelo treinado com duzentas épocas. Fonte: Autoria própria. 45

4.5 Predição das imagens de teste pelo modelo treinado com trezentas épocas. Fonte: Autoria própria. 45

4.6 Curvas de perda e *MIoU* de treinamento e validação, por duzentas épocas, para um processo de treinamento de predição de quatro classes. Fonte: Autoria própria. 49

4.7 Curvas de perda e *MIoU* de treinamento e validação, por duzentas épocas, para um processo de treinamento de predição de três classes. Fonte: Autoria própria. 49

4.8 Predição das imagens de teste pelo modelo treinado considerando quatro classes. Fonte: Autoria própria. 50

4.9 Predição das imagens de teste pelo modelo treinado considerando três classes. Fonte: Autoria própria. 50

4.10 Comparação das técnicas de segmentação por *K-Means* e CNN. Fonte: Autoria própria. 52

Lista de Tabelas

2.1 Comparação das principais técnicas de segmentação de imagem. 15
2.2 Tabela síntese adaptada do trabalho original que descreve a arquitetura ENet, fornecendo as dimensões das saídas considerando uma entrada de 512×512. Fonte: Adaptado de (PASZKE et al., 2016). 24

3.1 Codificação RGB das classes do *SUIM Dataset*. 31
3.2 Exemplo de matriz de confusão 2x2. 40

4.1 Mapeamento de classes após redução para quatro classes. 48
4.2 Mapeamento de classes após redução para três classes. 48
4.3 Métrica *MIoU* para as imagens testadas por ambas as técnicas. 53

Lista de Abreviações

AUV	Autonomous Underwater Vehicle (Veículo Autônomo Subaquático)
B/W	Black/White
CNN	Convolutional Neural Network
CUDA	Compute Unified Device Architecture (Arquitetura de Dispositivo de Computação Unificada)
DL	Deep Learning (Aprendizagem Profunda)
EDP	Equação Diferencial Parcial
ENet	Efficient Neural Network (Rede Neural Eficiente)
GPU	Graphics Processing Unit (Unidade de Processamento Gráfico)
HDR	High Dynamic Range (Grande Alcance Dinâmico)
HROV	Hybrid Remotely Operated Vehicle (Veículo Híbrido Operado Remotamente)
INS	Inertial Navigation System (Sistema de Navigação Inercial)
IoU	Intersect Over Union (Interseção sobre União)
MIoU	Mean Intersect Over Union (Interseção sobre União média)
MRF	Markov Random Field (Campo Aleatório de Markov)
MS-DOS	Microsoft Disk Operating System (Sistema Operacional em Disco da Microsoft)
PReLU	Parametric Rectified Linear Unit (Unidade de Retificação Linear Paramétrica)
RAM	Random Access Memory (Memória de Acesso Aleatório)
ReLU	Rectified Linear Unit (Unidade de Retificação Linear)
RGB	Red, Green, Blue (Vermelho, Verde, Azul)
ROV	Remotely Operated Vehicle (Veículo Operado Remotamente)
RTO	Real Time Operation (Operação em Tempo Real)
SGD	Stochastic Gradient Descent (Gradiente Descendente Estocástico)
SSS	Side Scan Sonar (Sonar de Varredura Lateral)

TPU	Tensor Processing Unit (Unidade de Processamento de Tensor)
UUV	Unmanned Underwater Vehicle (Veículo Subaquático não-tripulado)
VeRSTAPi2	Veículos Robóticos Subaquáticos Teleoperados/Autônomos para Inspeção e Intervenção

Resumo

Visão computacional é um campo amplamente estudado dentro das ciências da computação, surgindo a partir da ideia de se obter ou reconstruir características específicas de imagens como, por exemplo, formato, coloração e intensidade luminosa, obtidas por sensores óticos, como câmeras digitais, comumente utilizadas. Em constante aprimoramento, o esforço computacional necessário para efetuar tais operações é cada vez mais otimizado, reduzindo e viabilizando o *hardware* embarcado a ser aplicado em processamento de imagem em tempo real, característica essa requerida por sistemas autônomos, como Veículos Autônomos Subaquáticos (AUV, *Autonomous Underwater Vehicle*). Tal autonomia requer a habilidade de se identificar possíveis obstáculos e reconhecer de maneira satisfatória o ambiente em que se encontra, sendo de vital importância para garantir a autonomia destes equipamentos.

Em face destas considerações, estudou-se a visão computacional aplicada a ambientes subaquáticos, mais especificamente a operação de segmentação de imagem, uma das ferramentas utilizadas neste campo, empregada para distinção de formas específicas, como obstáculos e seres vivos, debaixo d'água. Uma breve revisão histórica desta ciência é aqui revisitada, com estudo das implementações mais utilizadas e um maior aprofundamento no âmbito de redes neurais. Foi aplicada a rede neural ENet (*Efficient Neural Network*) em um *dataset* aquático disponível publicamente para análise de sua performance, obtendo-se uma Interseção sobre União Média de 71,57% para as imagens de teste selecionadas, para três classes de predição. Comparou-se esta performance com uma técnica de segmentação mais tradicional de clusterização, *K-Means*, obtendo-se 39,00% para o mesmo conjunto de imagens. Estes resultados evidenciam que a tarefa de se segmentar imagens subaquáticas é desafiadora, e um pré-processamento destas imagens juntamente com um *dataset* bastante expressivo e representativo se fazem necessários.

Palavras-chave: Visão computacional. Segmentação de Imagem. Autônomo. Redes Neurais.

Abstract

Computer vision is a field widely studied within the computer sciences, arising from the idea of obtaining or reconstructing specific characteristics of images, such as format, color and light intensity, obtained by optical sensors, such as digital cameras, commonly used. In constant improvement, the computational effort required to carry out such operations is increasingly optimized, reducing and enabling embedded hardware to be applied in real-time image processing, a characteristic required by autonomous systems, such as Autonomous Underwater Vehicles (AUV). Such autonomy requires the ability to identify possible obstacles and to recognize in a satisfactory way the environment in which it finds itself, being of vital importance to guarantee the autonomy of this equipment.

In view of these considerations, computer vision applied to underwater environments was studied, more specifically the image segmentation operation, one of the tools applied by this field, used to distinguish specific forms, such as obstacles and living beings, under water. A brief historical review of this science is revisited here, with a study of the most widely used implementations and a deeper understanding of neural networks. The neural network ENet (Efficient Neural Network) was applied to a publicly available aquatic dataset to analyze its performance, obtaining a Mean Intersection over Union of 71,57% for the test images selected, for three classes. This performance was compared with a more traditional clustering segmentation technique, K-Means, obtaining 39,00% for the same set of images. These results show that the task of segmenting underwater images is challenging, and a pre-processing of these images together with a very expressive and representative dataset are necessary.

Keywords: Computer Vision. Image Segmentation. Autonomous. Neural Networks.

CAPÍTULO 1

Introdução

1.1 Visão Computacional

A primeira palavra que compõe a nomenclatura desta área é também um dos cinco sentidos humanos: a visão. É a capacidade de se captar energia luminosa refletida pelos objetos e interpretá-la de forma a se conceber a percepção do universo a sua volta. Como traduzir esta obra da evolução biológica em instruções de máquina, para então capacitá-la a possuir tal sentido? É uma pergunta complexa, tanto em semântica, quanto no objetivo em si. Em parte, este desafio se deve por se tratar de um problema inverso, onde se faz necessário recuperar informações desconhecidas ou perdidas para se chegar à solução. Levando-se em conta, neste sentido, a complexidade de se modelar a realidade visível de maneira fiel, inúmeras dificuldades emergem e configura-se este desafio complicado (SZELISKI, 2011).

A visão computacional é, portanto, a ciência e tecnologia responsáveis pelo desenvolvimento de sistemas artificias capazes de simular o sentido da visão humana, podendo extrair determinadas informações de imagens. Além disso, pode-se considerar que este campo da ciência é uma subárea da inteligência artificial, esta que busca aproximar o poder e capacidade de processamento das máquinas à inteligência humana. De maneira mais técnica, a inteligência artificial busca construir e entender os chamados agentes inteligentes, compreendidos como entidades que podem sentir e extrair informações do ambiente em que se encontra, e tomar ações que lhes sejam mais benéficas e eficientes (RUSSEL; NORVIG, 1995). De certa forma, é possível se traçar paralelos entre agentes inteligentes e seres humanos, quanto a tomadas de decisões.

Os primeiros esforços dentro da visão computacional datam da década de 60. Inicialmente,

a metodologia proposta era a conexão de uma câmera ao computador e fazê-lo descrever as imagens capturadas (SZELISKI, 2011 apud BODEN, 2006). Posteriormente, tentava-se extrair bordas e limites nas imagens buscando inferir objetos. Ainda durante esta década, de acordo com Szeliski (2011 apud MARR, 1982), uma primeira noção de como se descrever um sistema de processamento de informação visual foi sugerido e posteriormente publicado de maneira póstuma. Marr dividiu esta noção em três diferentes níveis de abstração: o primeiro nível, denominado teoria computacional, busca explicar o objetivo da aplicação e quais restrições podem ser inferidas acerca do problema; o segundo nível, de representação e algoritmo, descreve como são representadas a entrada, saída e demais parâmetros que compõem a etapa de processamento, e quais algoritmos serão utilizados para se alcançar o resultado desejado; e o terceiro nível, de implementação de *hardware*, que mapeia o segundo nível ao meio físico do computador. Estes conceitos foram e ainda são utilizados em algumas formulações da visão computacional.

Já na década de 80, pode-se destacar principalmente a introdução de modelos discretos de Campos Aleatórios de Markov (MRF, *Markov Random Fields*) como forma de se formular determinados problemas de detecção de bordas, e a utilização de filtros de Kalman para maior refinamento de variantes dos MRF. Na década de 90, algoritmos de multivisão estéreo eram bastante estudados (sendo tópico de pesquisa até hoje) e um maior aprofundamento dentro da área de segmentação de imagem, introduzindo técnicas como os de mínima energia, cortes normalizados e o chamado *mean-shift*. A Figura 1.1 mostra uma aplicação simples da segmentação de imagem. Vale destacar ainda dentro desta década os estudos de visão computacional em conjunto com a computação gráfica, produzindo técnicas como transformação de imagem (ou *image morphing*) e formação de modelos 3D a partir de coleções de imagens. Por fim, nos anos 2000 em diante, houve um maior aprofundamento nos estudos das técnicas de HDR (*High Dynamic Range*), bastante utilizadas atualmente, surgimento de técnicas de reconhecimento de objetos, desenvolvimento de algoritmos mais eficientes para problemas de otimização global e a aplicação de técnicas sofisticadas de *machine learning* em problemas de visão computacional (SZELISKI, 2011).

Quanto às aplicações da visão computacional, pode-se citar alguns casos bastante utilizados atualmente:

- Inspeção de máquinas: capaz de verificar tolerâncias de peças e detecção de falhas, por exemplo (SZELISKI, 2011);
- Reconhecimento facial: analisa a imagem de tal forma a se identificar rostos e posicionar limites que explicitem essas partes, como se mostra na Figura 1.2;
- Biometria: amplamente utilizada no ramo de segurança, realiza controle de acesso com base na análise dos padrões que identificam unicamente cada impressão digital das pessoas;
- Aplicações médicas: como apresenta Chouhan, Kaul e Singh (2019 apud CHAIRA,

Figura 1.1: Detecção de bordas para segmentação de imagem.

2010), uma das incontáveis aplicações da visão computacional é a segmentação de vasos sanguíneos e células sanguíneas em imagens médicas.

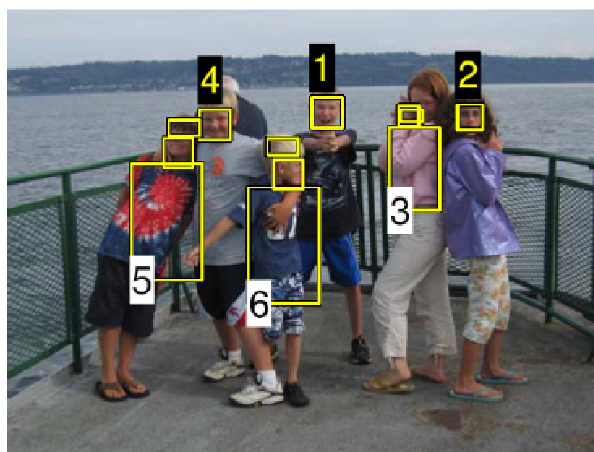

Figura 1.2: Exemplo de aplicação da visão computacional, o reconhecimento facial, capaz de identificar rostos nas imagens (SZELISKI, 2011).

Como mostra a última aplicação supracitada, é possível segmentar imagens de acordo com padrões identificáveis e previamente analisados e treinados em algoritmos específicos. Destaca-se, aqui, mais uma aplicação desta área e objeto de estudo deste trabalho: a análise de imagens subaquáticas. Propõe-se então a utilização de diferentes algoritmos de segmentação de imagem em fotografias debaixo d'água obtidas por meio de *datasets* que estejam disponíveis publicamente. Serão levantados o grau de confiança dos resultados obtidos e a performance de cada algoritmo em termos de impacto computacional, de tal forma que possa ser embarcado em um veículo do tipo AUV por meio de um *hardware* compacto.

1.2 ROVs e AUVs

ROVs são veículos subaquáticos não tripulados controlados remotamente, pela superfície, através de pilotos capacitados para operar tais equipamentos. Possui inúmeras aplicações, atreladas principalmente às áreas petrolífera, exploração marítima e educacional (SANTIAGO; MEZA; TITOTTO, 2017). São robôs submarinos capazes de prospectar, monitorar e até realizar determinadas intervenções por meio de ferramentas, remotamente. Fazem parte de um grupo maior de equipamentos denominados Veículos Subaquáticos não-tripulados (UUV, *Unmanned Underwater Vehicle*), que também inclui uma subclasse que engloba os chamados AUVs. Este último apresenta como grande diferencial a capacidade de operação autônoma, isto é, sem intervenção humana. A Figura 1.3 apresenta alguns modelos de ROVs e AUVs. De modo geral, ROVs possuem um corpo mais cúbico visando maior estabilidade de operação, evitando momentos provocados por movimentos de braços robóticos ou perturbações na água. AUVs possuem um invólucro externo em formato normalmente cilíndrico, diminuindo o arrasto frontal quando em navegação no sentido de seu eixo e compactação do tamanho total (normalmente não possuem braços robóticos para intervenções de manutenção como os ROVs). O grupo de pesquisa VeRSTAPi2 (2017) busca aprimorar o Veículo Híbrido Operado Remotamente (HROV, *Hybrid Remotely Operated Vehicle*) Proteo que se encontra na UFABC de Santo André, podendo futuramente ser desenvolvida uma plataforma autônoma a ser embarcada neste equipamento, demandando, assim, um algoritmo de segmentação de imagem, objeto de estudo deste trabalho, servindo como um ponto de partida para futuros trabalhos e estudos posteriores para possível implementação no HROV Proteo.

Figura 1.3: Exemplos de *designs* para ROVs e AUVs. Fonte: ⟨https://www.researchgate.net/figure/Different-kinds-of-AUVs-and-ROVs-106_fig3_332889842⟩.

Uma das características fundamentais para se alcançar a autonomia e independência do piloto humano é a habilidade de se localizar e identificar o ambiente que o(a) circunda. Suponha

um agente fictício (neste caso, nosso AUV) que deve realizar uma determinada missão, por exemplo, monitoramento subaquático), e deve ser capaz de completar tal objetivo sem intervenção de um agente externo. Uma das primeiras características que se sobressaem como indispensáveis para este agente é a capacidade de se situar e distinguir, de maneira minimamente satisfatória para tal missão, os diferentes objetos que compõem o universo a sua volta. Neste exemplo, identificar possíveis obstáculos, desde irregularidades que compõem o leito marítimo e fauna subaquática, até objetos submersos decorrentes de abandono ou acidente em alto mar, como cascos de navios afundados. Para tanto, além de um sistema de posicionamento geográfico (como o Sistema de Navegação Inercial - INS, *Inertial Navigation System*), deve-se incluir um dispositivo capaz de executar este discernimento entre obstáculos e caminho livre, como sonares ou câmeras associadas a algoritmos de visão computacional.

É importante, portanto, utilizar algoritmos que efetuem este processamento da imagem capturada (ou *frames* de vídeo) em tempo real, para a operação autônoma. Investiga-se adiante uma das etapas ou subáreas da visão computacional, a segmentação de imagem. A Figura 1.4 apresenta um diagrama como exemplo de segmentação de imagem subaquática.

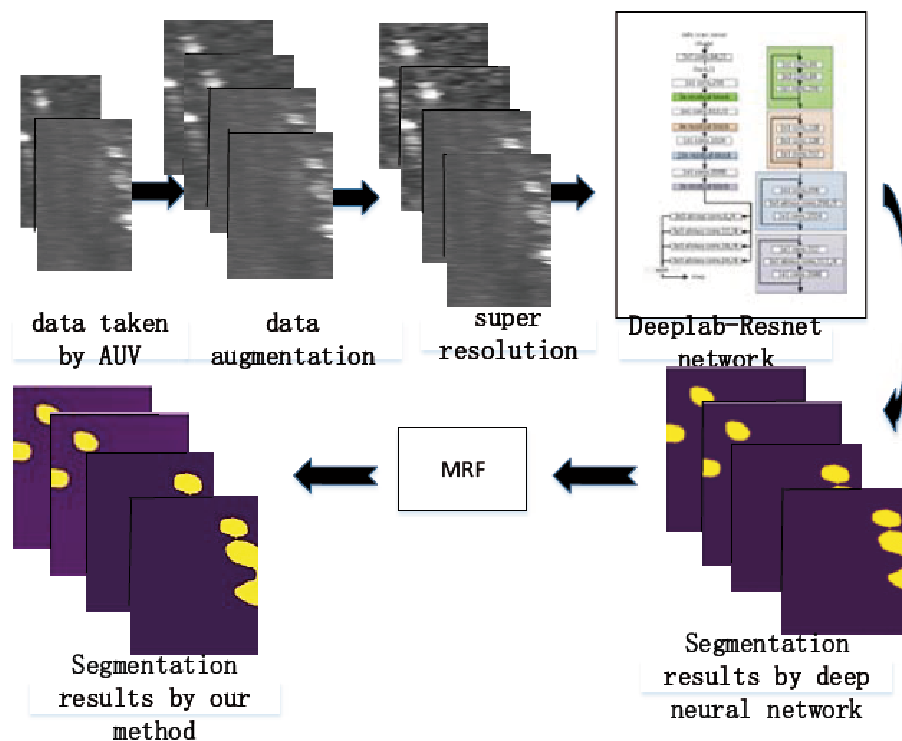

Figura 1.4: Metodologia de segmentação de imagem subaquática proposta por Yu et al. (2019).

1.3 Processamento de Imagens Subaquáticas

Quando se introduziu neste trabalho o conceito de visão computacional, utilizou-se o significado físico do sentido visão. Revisitando-o, "a visão é a capacidade de se captar energia

luminosa refletida pelos objetos e interpretá-la de forma a se conceber a percepção do universo a sua volta". Um ponto chave desta expressão se encontra no trecho que diz respeito à captação da luz refletida pelos objetos. Pode-se, ainda, extrair a informação mais valiosa que resume o princípio fundamental da visão: a luz. Por meio desta, e de suas consequentes reflexões, objetos podem ser interpretados por diferentes sistemas (orgânicos ou não).

Quando se fala em imagens subaquáticas, este ponto-chave se torna ainda mais relevante e determinante. A propagação da luz na água (ainda mais no oceano) sofre ação de fenômenos físicos de maneira mais intensa quando comparada à sua propagação no ar. No primeiro caso, a intensidade luminosa decai exponencialmente durante seu percurso através da água e, portanto, limita a distância de visibilidade. Esse fator é ainda mais pronunciado quando se leva em conta aplicações subaquáticas em maiores profundidades, demandando muitas vezes iluminação artificial que, por sua vez, apresenta também outras peculiaridades, como iluminação não uniforme, prejudicada principalmente nas áreas periféricas da imagem (SCHETTINI; CORCHS, 2010). Além disso, quanto maior a profundidade, maior a dificuldade de comprimentos de onda maiores (tendendo ao vermelho) penetrar na água. Assim, a luz azul (dentre as visíveis) alcança a maior profundidade em comparação às outras. Isso significa que, em grandes profundidades, as cores começam a se esvair. (HAVAí, 20–?). Os fenômenos que induzem estes eventos são absorção e espalhamento da luz, não somente pela água em si, mas também pela heterogeneidade do meio, que apresenta matéria orgânica dissolvida e minúsculas partículas suspensas na água (SCHETTINI; CORCHS, 2010). Um diagrama representativo destes efeitos pode ser conferido na Figura 1.5.

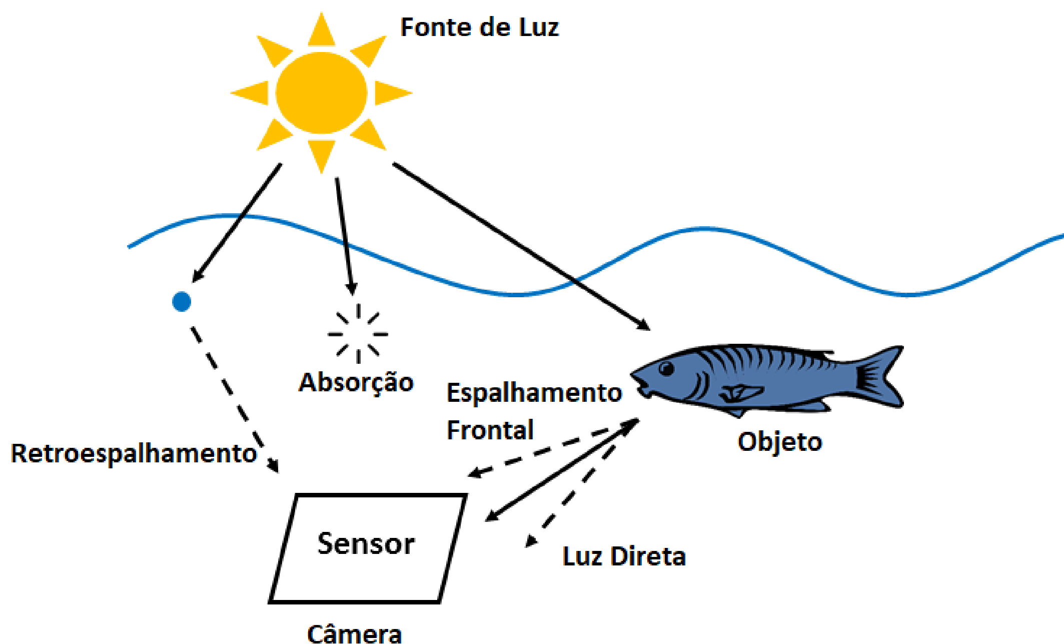

Figura 1.5: Diagrama representativo dos diferentes fenômenos atuantes na propagação da luz no oceano. Fonte: Adaptado de Drews-Jr et al. (2016).

Em virtude destes fenômenos que afetam a propagação da luz na água, as imagens capturadas, principalmente em grandes profundidades, apresentam baixa qualidade, apresentam alta distorção e turbidez, que dificultam a distinção de objetos (principalmente para algoritmos de visão computacional), baixa fidelidade de cores e redução de brilho e contraste. Existem técnicas de processamento de imagem que buscam restaurar determinados parâmetros da imagem e um outro conjunto de técnicas especializadas no aprimoramento da imagem, focando não no processo de degradação da imagem em si, provocado pelos diferentes fenômenos físicos, mas sim na imagem degradada propriamente dita, buscando uma melhora no contraste e definição (ZHANG et al., 2019). Resultados da aplicação de diferentes métodos de restauração de imagem podem ser conferidos na Figura 1.6.

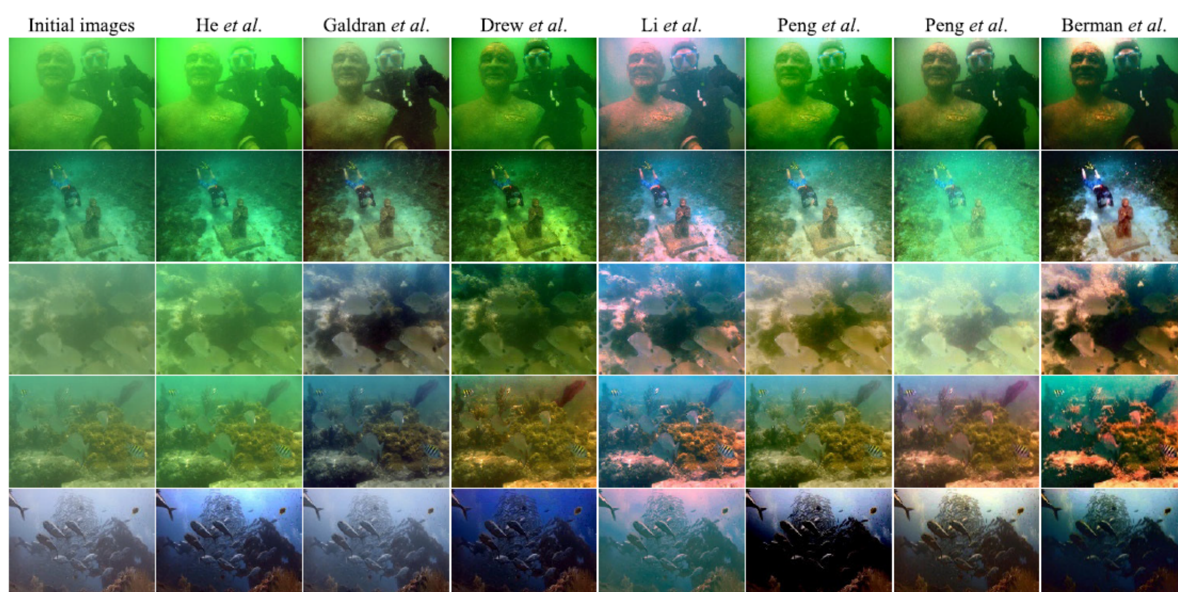

Figura 1.6: Conjunto de imagens antes e após a aplicação de métodos propostos por diversos autores para restauração de imagens subaquáticas. Fonte: Zhang et al. (2019).

Tratando-se de segmentação de imagem e algumas de suas técnicas, com maior detalhamento para a técnica de segmentação de imagem via redes neurais, é realizada uma fundamentação teórica no Capítulo 2, que também apresenta de maneira concisa o conceito de expansão artificial de dados (*Data Augmentation*). Já no Capítulo 3 são apresentadas as ferramentas utilizadas para a implementação prática do algoritmo de rede neural selecionado (ENet), além de elucidar de maneira didática alguns dos desafios encontrados durante este processo, apresentando as decisões tomadas ao longo da experimentação. Por fim, no Capítulo 4, são apresentados os resultados obtidos pela implementação proposta por este trabalho, procedido de uma breve discussão dos dados colhidos e por um breve comparativo com um método de segmentação mais tradicional (*K-Means*).

1.4 Objetivo Geral

Este trabalho tem por objetivo levantar as principais técnicas utilizadas no ramo de segmentação de imagem e aplicar a segmentação por Redes Neurais Convolucionais, (sendo a *ENet*, por Paszke et al. (2016), a arquitetura aqui escolhida) em um *dataset* de imagens subaquáticas, analisando seu desempenho e comportamento. Métricas como perda por iteração e Interseção sobre União Média (MIoU, *Mean Intersection over Union*) serão utilizadas para avaliar os resultados obtidos. Um breve comparativo com um método mais tradicional também é realizado, sendo o *K-Means* o algoritmo escolhido.

1.4.1 Objetivos Específicos

- Levantamento bibliográfico do estado-da-arte de técnicas para segmentação de imagens subaquáticas;
- Levantamento de dados para análise - coleta de imagens subaquáticas disponíveis por *datasets* públicos para análise dos algoritmos propostos;
- Aplicação da CNN proposta e avaliação dos resultados obtidos;
- Aplicação do algoritmo *K-Means* para breve comparativo com a arquitetura de CNN selecionada;
- Organização dos resultados obtidos dos itens precedidos de maneira concisa e didática visando instruir futuros trabalhos que venham a implementar estratégias semelhantes.

CAPÍTULO 2

Fundamentação Teórica

2.1 Segmentação de Imagem

O princípio fundamental de qualquer algoritmo de segmentação de imagem é, como o próprio nome sugere, dividir a imagem em setores onde cada partição possui determinadas características em comum. É bastante utilizada como um processo suporte em aplicações de compressão de imagem e reconhecimento de objetos, pois auxilia a processar apenas as partes de interesse da imagem, evitando o dispêndio de processamento computacional em porções sem informação útil (KAUR; KAUR, 2019).

É possível categorizar os algoritmos de segmentação de imagem em dois grandes grupos, de acordo com seu objetivo principal: algoritmos de segmentação semântica e algoritmos de segmentação de instância.

- **Segmentação Semântica (*Semantic Segmentation*)**: detecta grupos de *pixels* com características comuns e os agrupa baseado em categorias pré-definidas, por exemplo, supondo uma imagem de uma rua, separa calçada, pessoas, bicicletas, carros, *etc.*;
- **Segmentação de Instância (*Instance Segmentation*)**: pode-se dizer que estes algoritmos vão um passo além dos algoritmos de segmentação semântica, pois difere dentro dos grupos pré-definidos, diferentes instâncias/objetos. Portanto, é capaz de distinguir, por exemplo, crianças de adultos entre pessoas, carros de ônibus entre automóveis, *etc.*

A Figura 2.1 mostra uma aplicação simples de segmentação de imagem, colocando em contraste as duas metodologias acima citadas.

Pode-se ainda categorizar os algoritmos considerando como eles abordam o problema, isto é, de acordo com a estratégia básica a ser aplicada na separação dos segmentos. Elencam-se

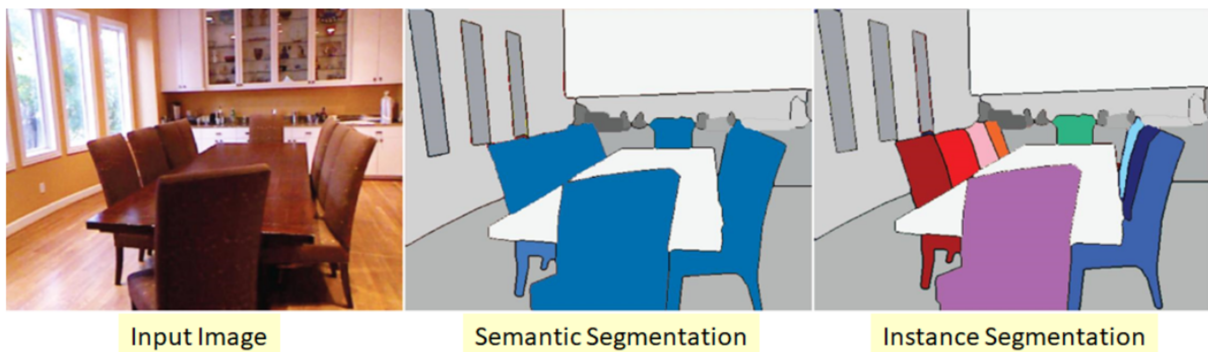

Figura 2.1: Exemplo de aplicação da segmentação semântica e da segmentação de instância. Nota-se que o segundo método se preocupa em separar objetos dentro dos grupos pré-definidos, nesse caso, de cadeiras. Fonte: ⟨https://towardsdatascience.com/review-deepmask-instance-segmentation-30327a072339⟩.

a seguir as topologias mais utilizadas a fim de ilustrar possíveis abordagens em problemas de segmentação de imagem (KAUR; KAUR, 2019):

- **Método do Limiar (*Thresholding Method*)**: constituem os algoritmos mais básicos, separando os *pixels* da imagem de acordo com sua intensidade. Portanto, é importante haver certo contraste entre os objetos e o fundo da imagem que se queira processar. Apresenta ótima performance para imagens que sejam mais fáceis de se distinguir os objetos a serem destacados, podendo-se ainda aplicar uma transformação Preto/Branco (B/W, *Black/White*) a fim de se ter apenas uma matriz de intensidades. Existem três tipos de métodos de limiar:

    1. *Limiar Global*: atribui um valor constante T para toda a imagem, produzindo dois grupos;
    2. *Limiar Variável*: atribui um valor de limiar T que pode variar, por exemplo, de acordo com a posição x e y de cada *pixel*; e
    3. *Limiar Múltiplo*: atribui n valores T(n) separando a imagem em n grupos.

    Normalmente utiliza-se histogramas de intensidade de *pixel* para se estabelecer os possíveis valores dos limiares;

- **Método de Segmentação Baseado em Contornos (*Edge Based Segmentation Method*)**: entende-se por contorno como o limite entre duas regiões homogêneas. Algoritmos baseados neste método tendem a detectar tais limites entre possíveis objetos e o fundo da imagem. É um processo de detecção de descontinuidades, onde aplicam-se máscaras/filtros (matrizes) que podem identificar pontos isolados (pela diferença de nível entre vizinhos), linhas (pelas máximas respostas de cada máscara) e contornos (pela aplicação de operadores específicos, como o gradiente) (AL-AMRI; KALYANKAR; KHAMITKAR, 2010);

- **Método de Segmentação Baseado em Regiões (*Region Based Segmentation Method*)**: segmenta a imagem em regiões com características parecidas. Existem duas formas básicas de implementação deste método (KAUR; KAUR, 2019):
    1. *Regiões Crescentes*: iniciam com pequenos aglomerados de *pixels* (as sementes, *seeds*) que expandem de acordo com condições pré-determinadas agrupando outros *pixels* que se enquadram em tais condições;
    2. *Separação e Fusão de Regiões*: separa inicialmente a imagem em pequenas regiões e agrega posteriormente regiões adjacentes com características semelhantes.
- **Método de Segmentação Baseado em Clusterização (*Clustering Based Segmentation Method*)**: são coleções de elementos similares, onde *clusters* cumprem requisitos específicos na classificação de objetos no processo. Desta forma, esses *clusters* agregam elementos que são mais parecidos entre si do que com outros elementos agrupados nos demais *clusters*. O algoritmo *K-means* é um dos algoritmos de *clusterização* mais utilizados, consistindo no agrupamento de diferentes *clusters* de acordo com suas distâncias. Basicamente, é implementado da seguinte forma (YUHENG; HAO, 2017):
    1. Seleciona-se aleatoriamente K centros de clusterização iniciais;
    2. Calcula a distância de cada amostra para cada centro de *cluster*, associando cada amostra ao centro de clusterização mais próximo;
    3. Para cada *cluster*, atualiza-se seus centros somando-se as distâncias de cada ponto com o seu respectivo centro associado e dividindo-se a soma pelo número de pontos deste *cluster*;
    4. Repete-se os passos 2 e 3 até atingir a convergência.
- **Método de Segmentação *Watershed* (*Watershed Segmentation Method*)**: é um método de segmentação idealizado pelo processo de formação de ”bacias hidrográficas”em uma paisagem ou relevo topográfico, oriundas de uma inundação, sendo estas as linhas que dividem os domínios de atração da chuva que cai sobre a região. Na prática, a técnica *Watershed* é aplicada ao gradiente da imagem e as linhas divisivas das ”bacias”separam regiões homogêneas. O gradiente de imagem para a transformação *Watershed* é normalmente encontrado através do gradiente morfológico (BALA, 2012). Uma abstração deste conceito pode ser vista na Figura 2.2.
- **Método de Segmentação Baseado em Equações Diferenciais Parciais (*Partial Differential Equation Based Segmentation Method*)**: método rápido, sendo valioso para aplicações em tempo real. Há dois tipos básicos de métodos de Equações Diferenciais Parciais (EDPs): detecção de bordas por filtro de difusão isotrópico não-linear, normalmente utilizando modelos de difusão, procurando por máximos e mínimos da função;

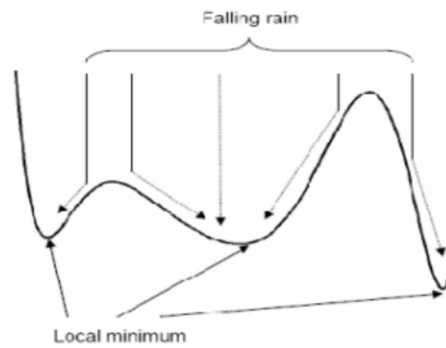

Figura 2.2: Desenho ilustrando o princípio que governa a segmentação *Watershed*. Fonte: (BALA, 2012).

e restauração por variação convexa não-quadrática, utilizada para remoção de ruído. Assim, normalmente a EDP de quarta ordem é utilizada para atenuação do ruído da imagem, enquanto a EDP de segunda ordem é aplicada para melhor detecção de limites e bordas (KAUR; KAUR, 2019).

- **Método de Segmentação Baseado em Rede Neural Artificial (*Artificial Neural Network Based Segmentation Method*)**: baseia-se na aplicação de redes neurais artificiais para reconhecimento de padrões, mais especificamente, neste caso, de contornos. É bastante utilizada para esta aplicação devida a introdução de um limiar não-linear pelos pesos e pelo uso da função de ativação sigmoidal na saída dos nós. Além disso, pode-se treinar a rede com dados que se aproximem da aplicação que deseja-se implementar (TERRY; VU, 1993). Apesar do tempo dispendido para o treinamento da rede, por se tratar de uma estrutura do tipo "caixa-preta", isto é, não se conhece as minúcias por dentro do código resultante final, garante que a implementação seja basicamente a configuração e ajuste da rede em si, não necessitando de técnicas avançadas de programação ou de fórmulas matemáticas complexas. Uma representação bastante simplificada do processo de detecção de borda por rede neural é mostrada na Figura 2.3.

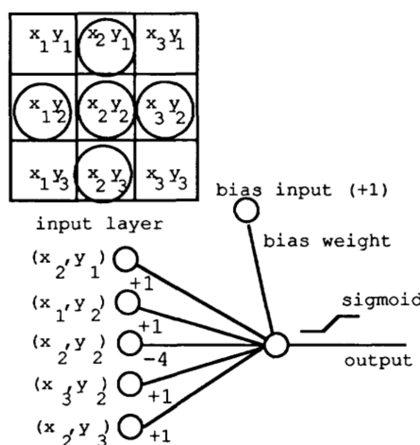

Figura 2.3: Diagrama de detetor de borda por rede neural. Fonte: (TERRY; VU, 1993).

A Tabela 2.1 apresenta de maneira concisa os prós e contras das metodologias mais utilizadas em segmentação de imagem, adaptada de Kaur e Kaur (2019 apud NARKHEDE, 2013).

Tabela 2.1: Comparação das principais técnicas de segmentação de imagem.

Técnica de Segmentação	**Estratégia**	**Prós**	**Contras**
Método do Limiar	Separa por valores limiares de intensidade	Sem necessidade de informações prévias, implementação simples	Altamente dependentes de picos (alto contraste), detalhes espaciais não são considerados
Método Baseado em Contornos	Detecção de descontinuidades	Melhor performance em imagens com bom contraste entre objetos	Não adequado para muitos contornos ou falsos positivos
Método Baseado em Regiões	Separa em regiões homogêneas	Maior imunidade a ruído e útil quando um critério de similaridade é fácil de ser encontrado	Alto consumo de tempo e memória
Método de Clusterização	Separa em clusters homogêneos	Mais indicado para problemas reais por utilizar lógica fuzzy (KAUR; KAUR, 2019)	Difícil levantamento da função de associação
Método Watershed	Estratégia traçada como uma interpretação topológica	Resultados mais estáveis e limites detectados são contínuos	Cálculos complexos de gradientes
Método Baseado em Equações Diferenciais Parciais	Baseada em aplicações de equações diferenciais	Método mais rápido e portanto adequado para aplicações onde o fator tempo é crítico	Maior complexidade computacional
Método Baseado em Rede Neural Artificial	Processos de treinamento para tomadas de decisão	Programação menos complexa	Tempo gasto para treinamento da rede

2.2 Rede Neural Convolucional

As chamadas Redes Neurais Convolucionais (CNN, *Convolutional Neural Networks*) são objetos de estudo e ferramentas importantes no campo da computação *Deep Learning*. Este, por sua vez, é um ramo de *Machine Learning*, ou aprendizado de máquina, que consiste em desenvolver algoritmos capazes de se aperfeiçoarem com base em experiências sem programação explícita. Quanto ao *Deep Learning*, sua ideia básica consiste em compor algoritmos baseados na repetição de funções específicas em múltiplas camadas, aumentando a generalização do aprendizado (AGGARWAL, 2018). A Figura 2.4 esquematiza esta repetição em camadas que compõe uma rede neural profunda.

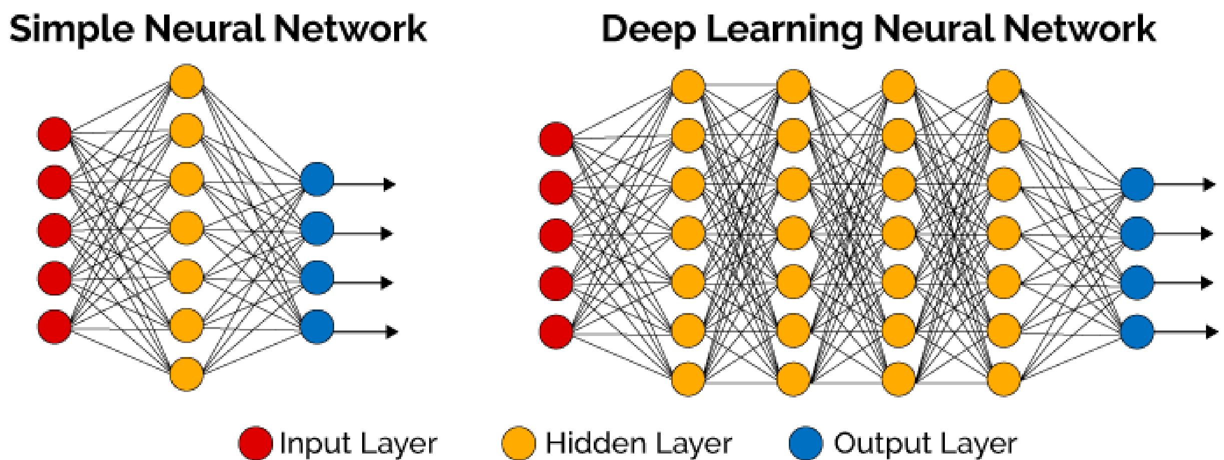

Figura 2.4: Rede Neural Profunda, demonstra a ideia básica do *Deep Learning*. Fonte: ⟨https://becominghuman.ai/deep-learning-made-easy-with-deep-cognition-403fbe445351⟩.

Se tratando de redes neurais, estas foram concebidas de modo a simular o sistema nervoso humano no processo de aprendizagem de máquinas, de maneira similar aos neurônios. Redes neurais artificiais são, teoricamente, capazes de aprender qualquer função matemática desde que sejam fornecidos dados de treinamento suficientes. A abstração deste conceito consiste numa abordagem de algoritmos de aprendizado que são continuamente otimizados por meio de um grafo computacional de dependências entre entrada e saída. As unidades computacionais que compõem a rede artificial também são chamadas de neurônios e são interconectados por meio de linhas que ponderam a função recebida por cada neurônio, através de pesos. Dessa forma, a rede gerada propaga os sinais de entrada dos primeiros neurônios até o(s) neurônio(s) de saída, sendo estes pesos parâmetros intermediários deste processo. Através do constante ajuste destes pesos que é possível o aprendizado do sistema (AGGARWAL, 2018).

Uma rede neural convolucional é tipicamente utilizada em aplicações de processamento de imagens. Foi concebida justamente para trabalhar com entradas matriciais que possuam forte dependência espacial. As imagens previamente citadas se encaixam bem nesta ideia, pois geralmente locais adjacentes de uma imagem possuem cores similares por uma questão de

continuidade. Uma característica importantíssima dessa rede neural é a aplicação da operação de *convolução* em uma ou mais camadas, como matrizes de pesos multiplicadas pelas matrizes de dados de entrada (AGGARWAL, 2018). A Figura 2.5 apresenta algumas das camadas que compõem uma CNN. Um maior detalhamento das camadas mais utilizadas será feito adiante (a saber: convolução, *pooling* e Unidade de Retificação Linear (ReLU, *Rectified Linear Unit*).

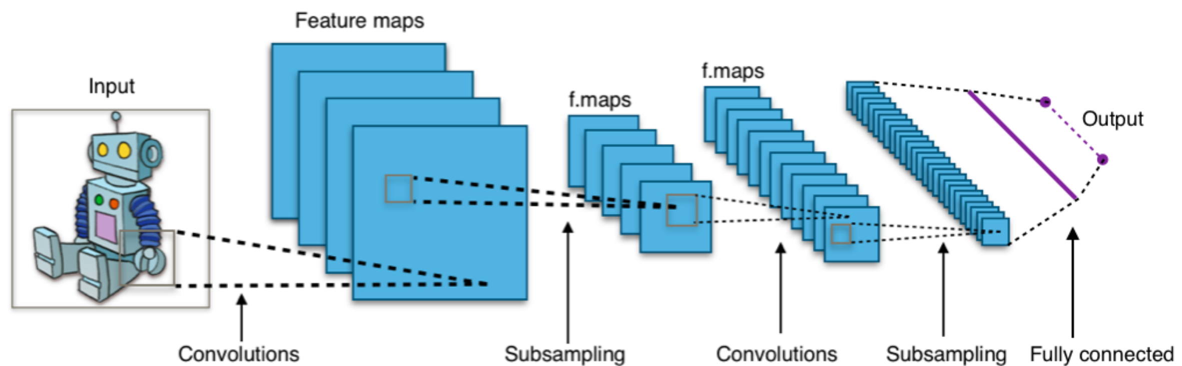

Figura 2.5: Exemplo de processamento de imagem utilizando CNN. Fonte: ⟨https://towardsdatascience.com/introducing-convolutional-neural-networks-in-deep-learning-400f9c3ad5e9⟩.

2.2.1 Convolução

Como o propósito principal deste trabalho é a aplicação da segmentação de imagem em uma foto subaquática, tratam-se aqui os dados de entrada como imagens. Sabe-se que imagens são conjuntos de dados que podem ser representados por matrizes (ou, fundamentalmente, tensores) de dimensões $L \times W \times d$, sendo L a altura da imagem, W a largura da imagem e d a quantidade de canais de cores que representam a imagem. Por exemplo, uma imagem em escala de cinza de 32 *pixels* de altura por 32 *pixels* de largura, pode ser representada por uma matriz $32 \times 32 \times 1$, sendo a última dimensão responsável por indicar o nível de intensidade de branco onde, numa representação de 8-bits, 255 corresponde ao branco total, 0 ao preto total e valores intermediários representando possíveis escalas de cinza. Se esta imagem fosse descrita em RGB (*Red, Green, Blue*), teríamos, portanto, dimensões equivalentes a $32 \times 32 \times 3$, sendo então três níveis de intensidade para cada uma das cores que compõe o espectro RGB. A Figura 2.6 mostra uma representação de uma matriz $7 \times 5 \times 3$.

Numa rede neural convolucional, seus parâmetros são organizados em estruturas tridimensionais, normalmente com comprimento e largura iguais, porém muito menores que a camada à qual são aplicadas. Estas estruturas são comumente chamadas de **filtros** ou ***kernels***. Nas camadas convolucionais, onde ocorrem as operações de convolução, estes filtros são “varridos” ao longo da imagem de modo que toda a imagem seja percorrida pelos mesmos, onde a cada iteração é executada uma operação de multiplicação escalar. Ao final da aplicação do filtro por

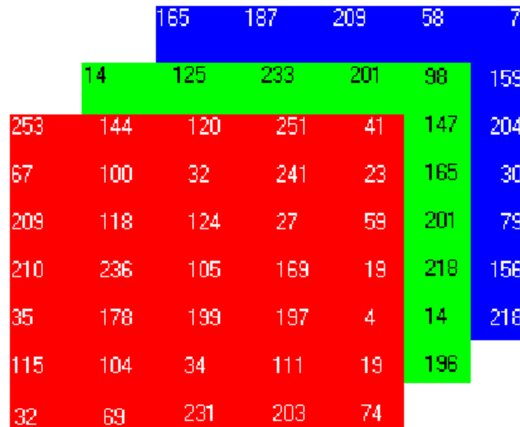

Figura 2.6: Representação da estrutura matricial para uma imagem em RGB. Fonte: ⟨https://www.researchgate.net/figure/A-three-dimensional-RGB-matrix-Each-layer-of-the-matrix-is-a-two-dimensional-matrix_fig6_267210444⟩.

cada posição da imagem (vide Figura 2.7 exemplificando tal processo), considerando um filtro de dimensões $F \times F \times d$ (lembrando que são usualmente quadrados e possuem a última dimensão igual à da imagem), a camada resultante possuirá dimensões $(L - F + 1) \times (W - F + 1) \times z$, onde z dependerá do número de filtros diferentes a serem aplicados (AGGARWAL, 2018).

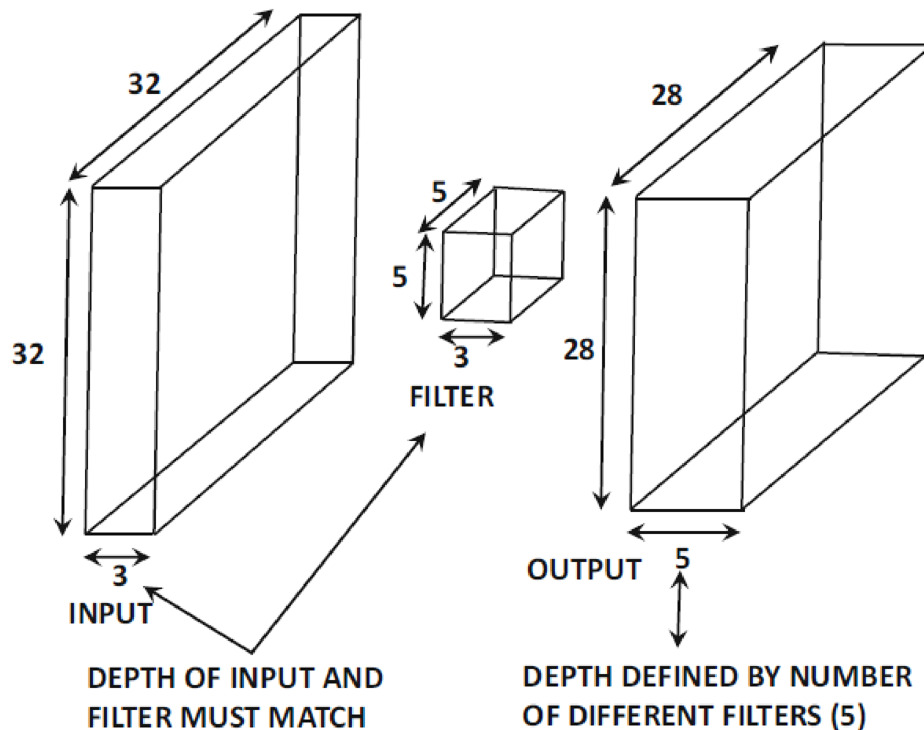

Figura 2.7: Exemplo de convolução entre uma camada de entrada com dimensões $32 \times 32 \times 3$ e um filtro de dimensões $5 \times 5 \times 3$. Fonte: (AGGARWAL, 2018).

Com estes conceitos em mente, pode se formalizar a operação de convolução. Por meio de

tensores, é possível escrever que o p-ésimo filtro na q-ésima camada possui parâmetros descritos pelo tensor tridimensional $W^{(p,q)} = [w_{ijk}^{(p,q)}]$, onde os índices i, j e k denotam as posições ao longo da altura, largura e profundidade do filtro. O mapa de atributos da q-ésima camada é descrito pelo tensor tridimensional $H^{(q)} = [h_{ijk}^{(q)}]$. Define-se, enfim, que a operação de convolução da q-ésima camada com a *(q+1)*-ésima camada é (AGGARWAL, 2018):

$$h_{ijp}^{(q+1)} = \sum_{r=1}^{F_q} \sum_{s=1}^{F_q} \sum_{k=1}^{d_q} w_{rsk}^{(p,q)} h_{i+r-1,j+s-1,k}^{(q)} \qquad \begin{array}{l} \forall i \in \{1...,L_q - F_q + 1\} \\ \forall j \in \{1...,B_q - F_q + 1\} \\ \forall i \in \{1...,d_{q+1}\} \end{array} \tag{2.1}$$

É possível interpretar a equação 2.1 como sendo um produto escalar sobre todo o volume do filtro repetido sobre todas as possíveis posições espaciais (i, j) e p filtros. Vale citar também que em todas as redes neurais é possível se adicionar um termo denominado *bias*, que é basicamente um valor $b^{(p,q)}$ a ser adicionado numa operação de convolução entre o p-ésimo filtro com a q-ésima camada, sendo então adicionado ao produto escalar da operação (AGGARWAL, 2018).

2.2.2 ReLU (*Rectified Linear Unit*)

A camada de ReLU representa a função de ativação de mesmo nome bastante utilizada em redes neurais. De maneira simplificada, retorna o valor máximo entre 0 e x, ou seja, assume valor nulo para valores negativos e x para valores positivos. Formalizando esta operação (SKANSI, 2018):

$$\rho(x) = max(x, 0) \tag{2.2}$$

2.2.3 *Pooling*

A operação de *Pooling* trabalha em matrizes menores, de tamanho $P_q \times P_q$, tendo como saída uma camada de mesma profundidade. A operação de *Pooling* pode reduzir drasticamente as dimensões de cada mapa de ativação a depender do passo a ser utilizado: se S_q = 1 (*stride* = 1), a nova camada terá dimensões $(L_q - P_q + 1) \times (B_q - P_q + 1) \times d_q$ e, se $S_q > 1$, então as dimensões serão $(L_q - P_q)/S_q + 1 \times (B_q - P_q)/S_q + 1 \times d_q$. Outra importante característica da aplicação da operação de *Pooling* é o aumento do campo receptivo em conjunto com a redução do espaço da camada. Campos receptivos maiores permitem capturar regiões maiores da imagem em um atributo complexo nas camadas seguintes (AGGARWAL, 2018).

Duas operações comumente utilizadas do *Pooling* são:

1. *Max-Pooling*: para cada quadrado de região $P_q \times P_q$ de cada um dos d_q mapas de ativação, o valor máximo é retornado;
2. *Average-Pooling*: bastante simular ao *max-pooling*, porém retorna a média dos valores da região aplicada (BROWNLEE, 2019).

A Figura 2.8 demonstra a aplicação de uma operação de *max-pooling* com passos $S_q | \forall q \in \{1,2\}$.

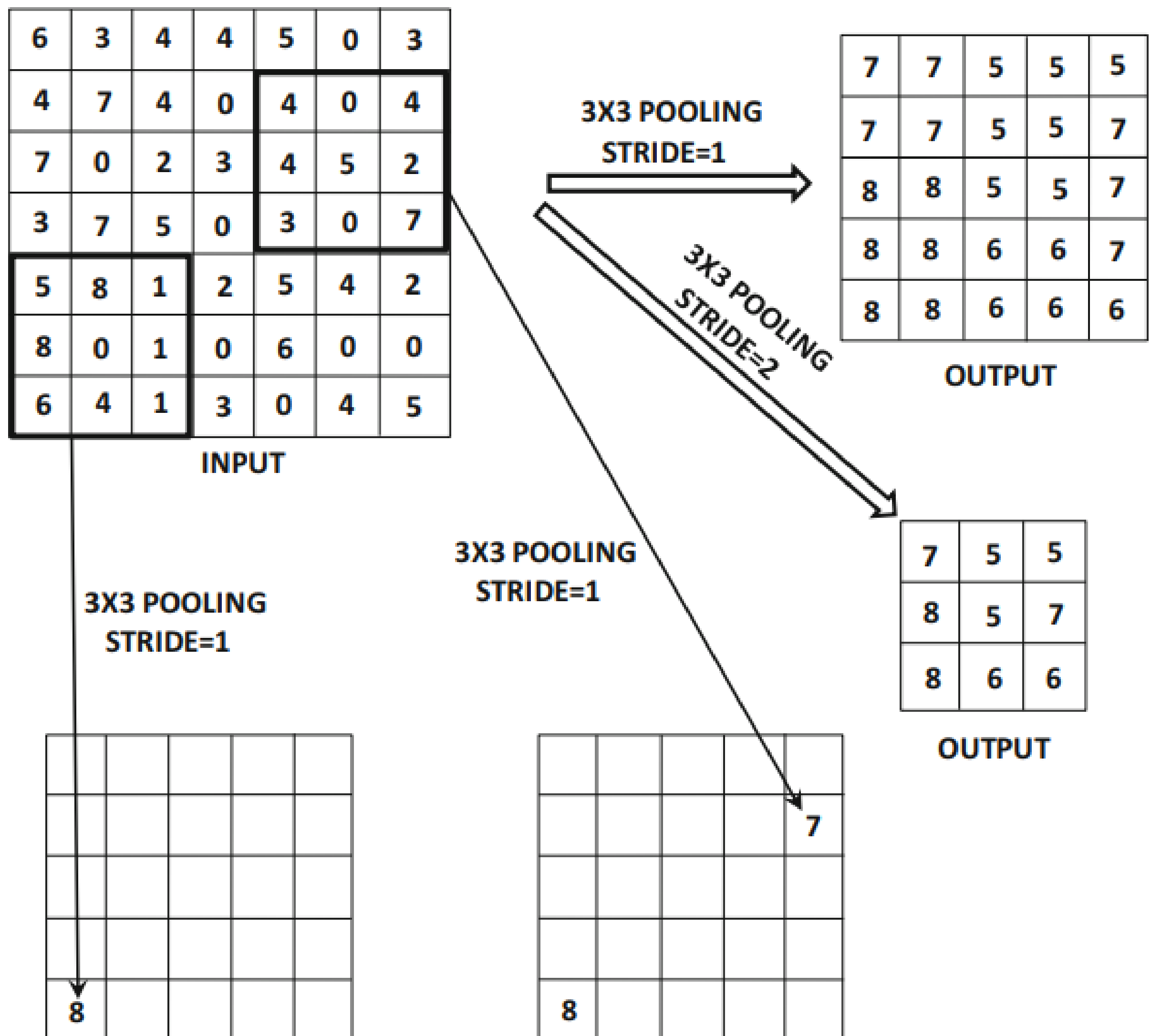

Figura 2.8: Exemplo de aplicação de uma operação de *max-pooling* com passos 1 e 2, demonstrando a diferença de tamanho entre as camadas resultantes de tal processo. Fonte: (AGGARWAL, 2018).

2.2.4 Treinamento

O processo de treinamento de redes neurais consiste na aplicação de algoritmos que, de forma iterativa, realiza a modificação dos pesos das conexões de cada neurônio artificial, de

modo a produzir o menor erro possível na saída. Existem basicamente três abordagens para se executar o processo de treinamento (BODEN, 2006):

- *Aprendizado Supervisionado*: através de uma gama de dados de entrada e saída, o programador alimenta a rede neural com as entradas indicando os resultados desejados de saída, informando ao algoritmo se o resultado é satisfatório ou não. Em caso negativo, outra iteração é executada, reajustando-se os pesos da rede até atingir o sucesso;
- *Aprendizado Não-Supervisionado*: não apresenta os resultados desejados de saída e nem fornece penalidades/recompensas (utilizados no Aprendizado por Reforço) para a rede, sendo esta a responsável por se reajustar de acordo com os resultados alcançados;
- *Aprendizado por Reforço*: fornece "recompensas/punições" de acordo com as decisões tomadas, esperando-se ajustar seus parâmetros com base na assertividade.

A Figura 2.9 apresenta um diagrama que resume os diferentes tipos de aprendizado citados acima.

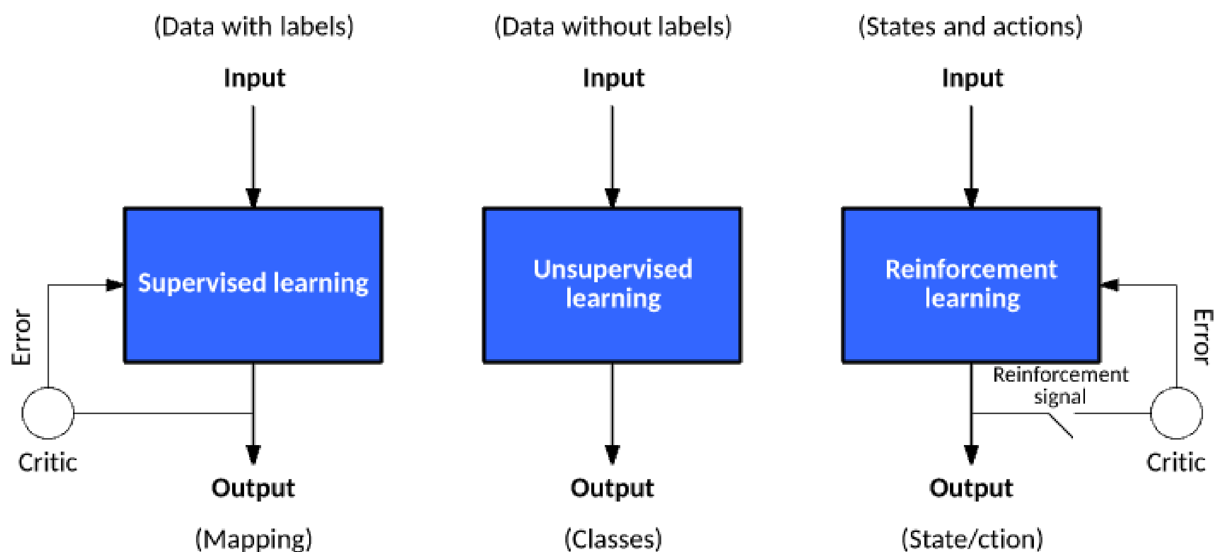

Figura 2.9: Diagrama que sintetiza as ideias dos diferentes tipos de aprendizado. Fonte: ⟨https://br.pinterest.com/pin/434597432787064636/⟩.

Duas das técnicas mais utilizadas nos treinamentos de redes neurais convolucionais, ambas normalmente aplicadas como aprendizado supervisionado, são as chamadas **Gradiente Descendente (*Gradient Descent*)** e Retropropagação (***Backpropagation***).

No campo da matemática, o gradiente é definido como um vetor que aponta para a direção de maior (ou menor) crescimento da função. Utiliza-se este conceito para diferenciação entre dois valores durante o treinamento de redes neurais. O gradiente descendente baseia-se nos seguintes passos (ZHOU, 2018):

1. Atribua um valor aleatório (ou zero) para θ;

2. Modifique o valor de θ de tal forma que a função de erro $J(\theta)$ decresça na direção de diminuição do gradiente.

Como se observa na Figura 2.10, ajusta-se o valor de θ fazendo com que $J(\theta)$ decresça até um ponto de mínimo, atingindo a convergência quando θ assume um valor extremo. Formalizando matematicamente (ZHOU, 2018):

$$\frac{\partial}{\partial \theta}J(\theta) = \frac{\partial}{\partial \theta}\frac{1}{2}\sum_{i=1}^{m}(h_{\theta}(x)-y)^2 = (h_{\theta}(x)-y)x^{(i)} \tag{2.3}$$

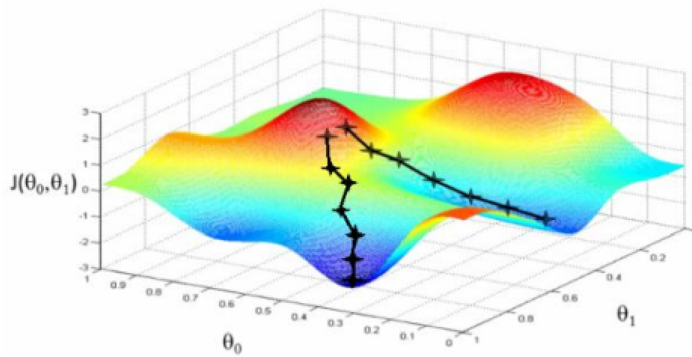

Figura 2.10: Representação topográfica do gradiente descendente. Fonte: (ZHOU, 2018).

O *Backpropagation* consiste em um processo iterativo onde se aplicam derivadas para se aprender os pesos dos neurônios, repetindo-se diversas vezes de tal forma a se “propagar do fim ao começo” os erros através das camadas, tal que (SKANSI, 2018):

$$w_{n+1} = w_n - \eta \nabla E \tag{2.4}$$

onde:

w é o peso;

η é a taxa de aprendizagem; e

E representa a função custo que mede o desempenho geral.

A taxa de aprendizagem η é responsável por ponderar o quanto se atualiza os valores a cada iteração, podendo, a depender da aplicação, ser implementada de três modos (SKANSI, 2018):

1. Taxa de Aprendizagem fixa;
2. Taxa de Aprendizagem adaptável global;
3. Taxa de Aprendizagem adaptável para cada conexão.

No final das contas, *Backpropagation* nada mais é do que a aplicação do Gradiente Descendente através da regra da cadeia, propagando-se do fim da rede ao início, para cada camada, de modo a se atualizar os erros ao longo das camadas. A Figura 2.11 mostra um diagrama simplificado deste conceito.

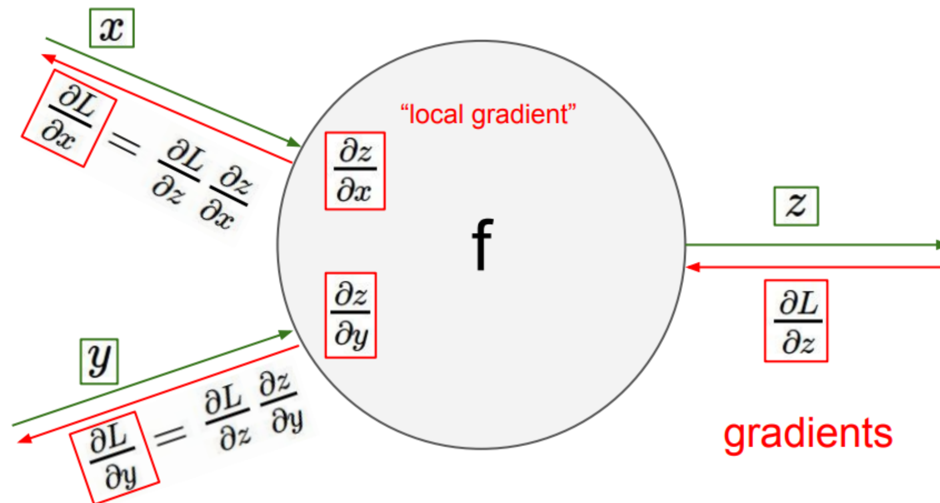

Figura 2.11: Processo de *Backpropagation* simplificado. Fonte: ⟨https://becominghuman.ai/back-propagation-in-convolutional-neural-networks-intuition-and-code-714ef1c38199⟩.

2.2.5 CNN em Segmentação de Imagens

Se tratando de aplicações de Operação em Tempo Real (RTO, *Real Time Operation*), um fator crucial a ser levado em conta é o seu tempo de processamento. É fácil imaginar que, por exemplo, em uma missão autônoma de um agente, a habilidade de interpretar o ambiente ao seu redor é imperativo. Algoritmos que demandem elevado tempo de trabalho para operar impedem o correto funcionamento da autonomia a ser embarcada em tal agente.

Seleciona-se, aqui, a arquitetura ENet desenvolvida por Paszke et al. (2016) e implementada, em conjunto com MRF, por Li et al. (2019) para segmentação de imagens de Sonar de Varredura Lateral (SSS, *Side Scan Sonar*). Esta arquitetura de CNN apresenta uma boa performance em tempo de processamento, sendo apropriada para aplicações em tempo real.

A arquitetura da ENet pode ser sintetizada conforme a Tabela 2.2 que mostra a tabela de estágios da arquitetura adaptada do trabalho original de Paszke et al. (2016). Os blocos principais são sintetizados conforme a Figura 2.12 também adaptada de Paszke et al. (2016) para fins de introdução da arquitetura neste trabalho.

O estágio inicial da rede contém um "bloco inicial" conforme mostrado na Figura 2.12a. Os estágios 1, 2 e 3 (observados na Figura 2.2, onde cada estágio é delimitado por linhas horizontais) pertencem ao chamado *Encoder*, sendo compostos por diversos "módulos de gargalo" com operações de *downsampling* (subamostragem, com exceção do estágio 3). Os estágios 4 e 5 formam o chamado *Decoder*, também compostos por "módulos de gargalo", porém com

Tabela 2.2: Tabela síntese adaptada do trabalho original que descreve a arquitetura ENet, fornecendo as dimensões das saídas considerando uma entrada de 512×512. Fonte: Adaptado de (PASZKE et al., 2016).

Nome	Tipo	Tamanho de saída
Inicial		16 x 256 x 256
Gargalo1.0	Subamostragem	64 x 128 x 128
4 x Gargalo 1.x		64 x 128 x 128
Gargalo2.0	Subamostragem	128 x 64 x 64
Gargalo2.1		128 x 64 x 64
Gargalo2.2	Dilatada 2	128 x 64 x 64
Gargalo2.3	Assimétrica 5	128 x 64 x 64
Gargalo2.4	Dilatada 4	128 x 64 x 64
Gargalo2.5		128 x 64 x 64
Gargalo2.6	Dilatada 8	128 x 64 x 64
Gargalo2.7	Assimétrica 5	128 x 64 x 64
Gargalo2.8	Dilatada 16	128 x 64 x 64
Repetir seção 2, sem Gargalo2.0		
Gargalo4.0	Sobreamostragem	64 x 128 x 128
Gargalo4.1		64 x 128 x 128
Gargalo4.2		64 x 128 x 128
Gargalo5.0	Sobreamostragem	16 x 256 x 256
Gargalo5.1		16 x 256 x 256
Convolução Completa		C x 512 x 512

operações de *upsampling* (sobreamostragem). Estes "módulos de gargalo", conforme se observa na Figura 2.12b, são compostos por (PASZKE et al., 2016):

- Projeção 1×1 que reduz dimensionalidade;
- Camada convolucional principal;
- Expansão 1×1;
- Bloco regularizador com *SpatialDropout* de Tompson et al. (2015) com parâmetros $p = 0,01$ e $p = 0,1$ a depender do estágio. *Dropout* é um recurso computacional que consiste em remover *inputs* de uma camada probabilisticamente, podendo ser variáveis de entrada da amostragem dos dados ou ativações, produzindo um efeito de simulação de várias redes de estruturas altamente heterogêneas que normalmente tornam os nós da rede mais robustas às entradas (BROWNLEE, 2018). Já o *SpatialDropout* se diferencia por remover mapas de atributos inteiros da camada convolucional, não sendo usados, então, em operações de *pooling* (TOMPSON et al., 2015).

Entre todas as operações de convolução, são inseridos blocos de normalização de lotes (*Batch Normalization* (IOFFE; SZEGEDY, 2015)) e Unidades de Retificação Linear Paramétrica (PReLU, *Parametric Rectified Linear Unit*) (HE et al., 2015; PASZKE et al., 2016). *Batch Normalization* é uma técnica que consiste em normalizar a saída de uma camada de ativação anterior

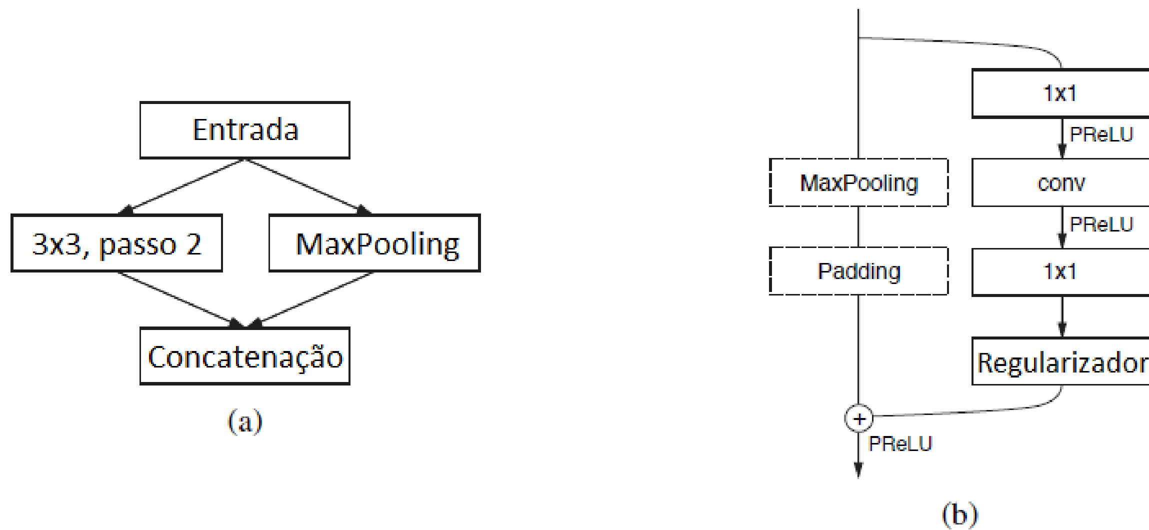

Figura 2.12: Blocos estruturais da rede ENet. (a) o bloco inicial com *MaxPooling* de janelas não sobrepostas de 2×2 e convolução com 13 filtros, somando 16 mapas de atributos após concatenação. (b) "módulo de gargalo", segundo Paszke et al. (2016), com bloco `conv` podendo ser uma convolução regular, dilatada ou total com filtros 3×3 ou, ainda, convolução 5×5 decomposta em duas assimétricas. Fonte: Adaptado de (PASZKE et al., 2016).

subtraindo a média e dividindo pelo desvio padrão do *batch*. Garante maior estabilidade da rede neural e acelera o treinamento, permitindo maior faixa da taxa de treinamento e minimizando o prejuízo na convergência do aprendizado (HUBER, 2020). Já o PReLU basicamente é uma função de ativação que aprende adaptativamente os parâmetros dos retificadores, melhorando a acurácia dos resultados por baixo custo computacional, podendo ser treinado também por *backpropagation* (HE et al., 2015).

Para os gargalos de *downsampling*, a projeção 1×1 é substituída por uma camada de convolução 2×2 com passo 2 e é feito o *zero padding* (adição de zeros nas extremidades das dimensões dos dados, ou seja, nas bordas das imagens ou mapa de atributos) das ativações para casamento do número de mapas de atributos, além da adição de uma camada de *Max Pooling*. Também são feitas alterações no *Decoder*, com substituição das camadas de *Max Pooling* por camadas de *Max Unpooling*, basicamente a operação reversa, além da substituição do *Padding* por convolução no domínio do espaço sem *bias* (PASZKE et al., 2016).

A Figura 2.13 apresenta alguns dos resultados da aplicação desta arquitetura. A principal vantagem desta topologia é a grande diminuição de esforço computacional e, consequentemente, tempo de processamento.

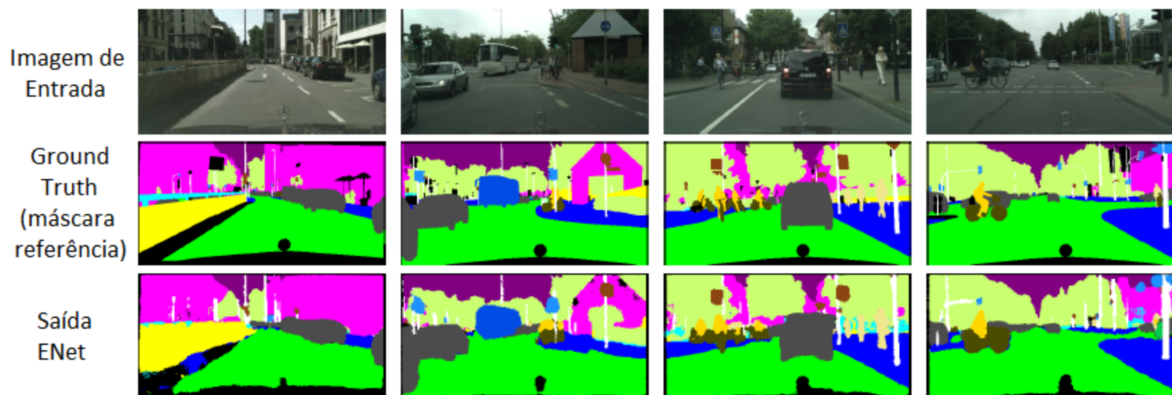

Figura 2.13: Aplicação do ENet para segmentação de imagens de ruas. Fonte: Adaptado de (PASZKE et al., 2016).

2.2.6 *Data Augmentation*

Para o treinamento de redes neurais, é imprescindível a coleta de uma base de dados que reflita informações e características que sejam desejadas ao aprendizado da rede neural. Tal conjunto de dados, o *dataset*, pode ser um desafio a depender da aplicação final. Imagens médicas, por exemplo, é uma das áreas onde o Aprendizado Profundo (DL, *Deep Learning*) está se desenvolvendo para criação de redes neurais capazes de reconhecer possíveis indicadores de doenças em seus estágios iniciais. Um grande obstáculo, entretanto, está na dificuldade em se obter uma amostra significativa de dados para o treinamento. CNNs são altamente dependentes de grandes *datasets* para que o aprendizado possa ocorrer apropriadamente (levando-se em conta, claro, que a arquitetura da rede neural esteja corretamente construída e seja profunda o suficiente tal a ser capaz de abstrair as características necessárias). Dois problemas intrinsecamente relacionados, mas não exclusivos à qualidade do conjunto de dados de treinamento são o *overfitting* (sobre-ajuste) e o *underfitting* (sub-ajuste):

1. *Overfitting*: o sobre-ajuste pode ser entendido como um ajuste tão bem adaptado ao conjunto de dados de treinamento que, ao receber novos dados que são alheios a este, não consegue performar tão bem quanto com os utilizados para o treino. Em outras palavras, a generalização do modelo é baixa. O modelo se ajustou tanto aos dados de treinamento que este não aprendeu características importantes que distinguem o conjunto de treinamento de outros dados de validação. Uma maneira de se identificar esta tendência é observar que o erro de treinamento é baixo, enquanto que o erro de validação e teste é alto. A Figura 2.14 exemplifica uma situação de *overfitting*. Este comportamento pode ser causado, por exemplo, por um conjunto de dados insuficiente (*dataset* pequeno) ou um algoritmo extremamente sofisticado para os dados;
2. *Underfitting*: o sub-ajuste ocorre quando o modelo não é capaz de se ajustar sequer aos dados de treinamento. Neste caso, tanto o erro de treinamento quanto os erros de

validação e teste são altos, podendo indicar que o algoritmo é inadequado (rede neural muito simples para o conjunto de dados), o conjunto de dados não é grande o suficiente e/ou as características do *dataset* são fracamente relacionadas entre si.

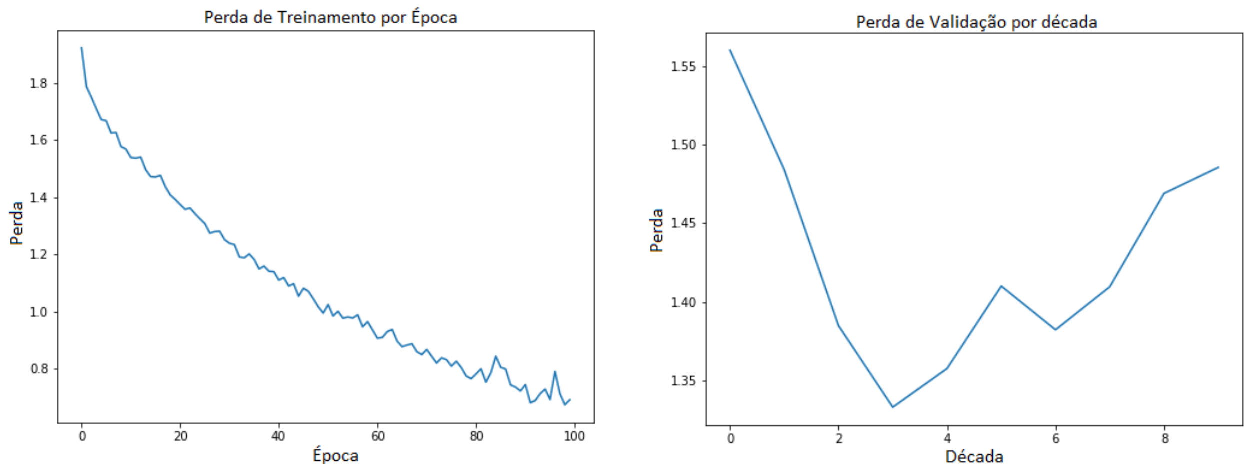

Figura 2.14: Curva típica de comportamento de *overfitting*. Repare que o erro de treinamento tende a convergir em uma tendência de queda, enquanto que o erro de validação passa a subir. Fonte: Autoria própria.

Data Augmentation é o nome dado a um conjunto de técnicas que buscam aprimorar o tamanho e qualidade do conjunto de dados disponível para treinamento (SHORTEN; KHOSHGOFTAAR, 2019). Algumas destas técnicas foram brevemente mencionadas na descrição da arquitetura da *ENet*, na seção 2.2.5, como *dropout* e *batch normalization*. Outras estratégias que podem ser implementadas também são *flips* aleatórios, recortes aleatórios e processamentos de imagem como transformações no espaço de cores.

Neste trabalho, algumas destas estratégias foram utilizadas para se minimizar a tendência ao *overfitting*, como forma de se expandir artificialmente o *dataset* utilizado. Para tanto, inversões horizontais e verticais aleatórias, bem como *crops* aleatórios foram escolhidos como técnicas simples de se prover algum nível de expansão no conjunto de dados.

2.3 Segmentação por Clusterização *K-Means*

Dentre os métodos de clusterização, o *K-Means* é um dos mais utilizados, muito por conta de sua simplicidade de implementação e por ser computacionalmente rápido. Essencialmente, consiste em dividir um conjunto de dados (imagens, para o tema de segmentação aqui abordado) que possuam semelhanças (cor) computando suas respectivas distâncias (normalmente euclidiana) de K centróides dispostos aleatoriamente ou sob uma determinada regra (manualmente, por exemplo). Depois, busca agrupar cada ponto ao *cluster* cujo centróide seja mais próximo. Iterativamente, o algoritmo recalcula um novo centróide para cada *cluster* e contabiliza novamente a

distância entre os pontos e o respectivo novo centróide, reagrupando-os aos *clusters* de menor distância. Assim, pode-se dizer que os centróides são os pontos de cada *cluster* onde a soma das distâncias entre os objetos deste *cluster* são mínimas, e o algoritmo trabalha de maneira iterativa para esta minimização (DHANACHANDRA; MANGLEM; CHANU, 2015).

O algoritmo pode então ser resumido como segue:

1. K *clusters* são inicializados;
2. Os centróides dos K *clusters* são selecionados (de maneira aleatória ou não);
3. Os dados são agrupados conforme os centróides mais próximos. Para uma distância euclidiana, tem-se:

$$d = ||p(x,y) - c_k|| \tag{2.5}$$

   onde *p(x,y)* representa um ponto (dado) e c_k o centróide k;
4. Quando todos os dados são atribuídos aos seus centróides mais próximos, estes centros são recalculados conforme:

$$c_k = \frac{1}{k} \sum_{y \in c_k} \sum_{x \in c_k} p(x,y) \tag{2.6}$$

5. O processo é repetido até que o critério de parada seja satisfeito (por tolerância ou por erro).

A Figura 2.15 apresenta uma simplificação do processo do algoritmo de clusterização *K-Means* para um pequeno conjunto de pontos.

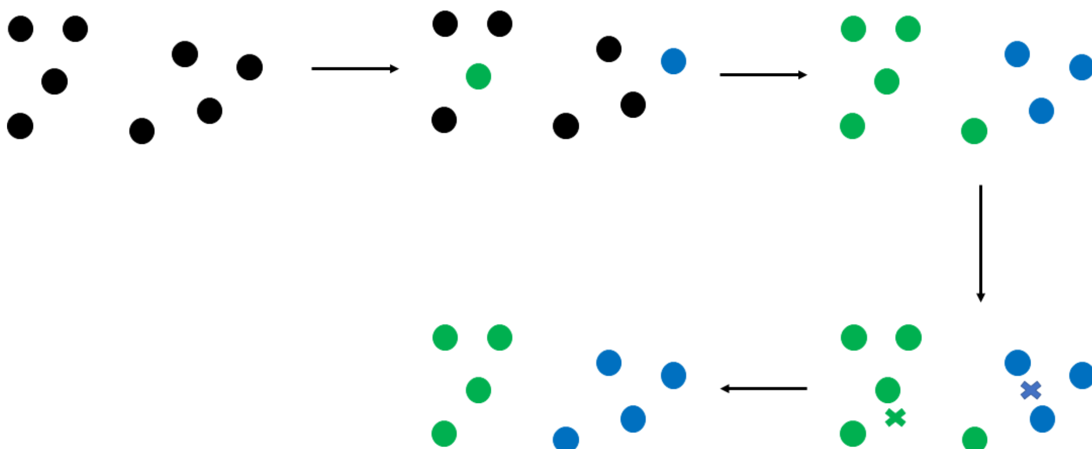

Figura 2.15: Exemplo simples de clusterização por *K-Means*. Fonte: Autoria própria.

Embora de fácil implementação, uma desvantagem que vale ser mencionada é a sua alta dependência de uma boa seleção dos centróides iniciais. Se estes forem aleatoriamente selecionados, resultados diferentes são gerados. Desta maneira, para segmentação de imagens, é interessante a implementação de uma estratégia de seleção destes centróides, o que pode se tornar desafiador para operações que demandam processamento em tempo real.

CAPÍTULO 3

Metodologia e Implementação

Através do estudo e análise da fundamentação bibliográfica levantada neste documento e a categorização do algoritmo a ser implementado, ENet (CNN), foram efetuadas as seguintes etapas a fim de se atingir ao final do processo de Trabalho de Graduação:

- Levantamento de imagens através de *dataset(s)* disponível(is) publicamente que contenham diferentes aspectos contribuintes à caracterização de imagens subaquáticas (turbidez, distorção de cores, etc.);
- Codificação de uma classe *Custom Dataset* que integre todas as funcionalidades já implementadas por um código base com adição de funções que possam compatibilizar um *dataset* externo ao código pré-existente. Deve-se levar em conta a codificação correta da máscara RGB de *ground truth* para correta ingestão do novo *dataset* implementado;
- Definição de funções auxiliares que efetuem transformações nas imagens do *dataset* para providenciar uma expansão artificial de dados (*data augmentation*) a fim de se minimizar os efeitos de *overfitting* da rede neural;
- Verificação empírica de alguns *hiper*-parâmetros procurando melhores indicadores de perda, Interseção sobre União média (MIoU, *Mean Intersect over Union*) e estabilidade de curvas de aprendizado;
- Medição da performance da rede analisando perda média e MIoU;
- Comparação simples com algoritmo de clusterização *K-Means*.

Os seguintes materiais/ferramentas foram utilizados(as) para as etapas supracitadas:

- *PyTorch*[1]: é uma biblioteca *open-source* utilizada para aplicações de aprendizado de

[1] https://pytorch.org/

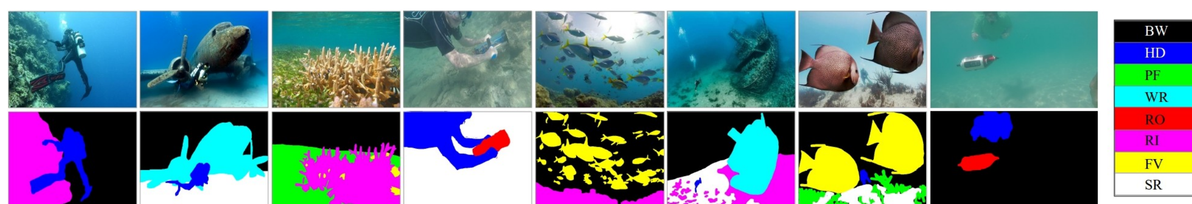

Figura 3.1: Exemplos de imagens e suas máscaras de segmentação *ground truth* em RGB. Fonte: ⟨http://irvlab.cs.umn.edu/resources/suim-dataset⟩.

máquina e *Deep Learning* em *Python*. Fornece funções, por exemplo, para cálculos de tensores com aceleração via Unidade de Processamento Gráfico (GPU, *Graphics Processing Unit*) (e via *CUDA*®, *Compute Unified Device Architecture*), funções para construção de redes neurais, transformações de imagens, *etc.*;

- *Google Colab*[2]: ambiente colaborativo do *Google* em nuvem que permite codificar e rodar códigos em *Python* com interface bastante similar ao *Jupyter Notebook*. Além disso, fornece gratuitamente ambientes de execução remotos, disponibilizando recursos computacionais como Disco, RAM (*Random Access Memory*) e GPU para implementação do projeto proposto;
- *SUIM Dataset*[3]: *dataset* disponibilizado pela Universidade de Minnesota destinado a estudos de segmentação de imagens subaquáticas. Contém 1525 imagens de treinamento e validação, e 110 imagens de teste para avaliação do modelo levantado, sendo composto de pares de imagens subaquáticas com suas respectivas máscaras (*ground truth*) codificadas em RGB representando as classes do *dataset*. A Figura 3.1 apresenta alguns pares de imagem/máscara do *dataset* em questão;
- *OpenCV*[4]: biblioteca para aplicações de Visão Computacional, *open source* e de fácil utilização em conjunto com o *Google Colab*. Foi escolhido para implementação do algoritmo *K-Means*.

O ambiente de execução do *Google Colab* foi configurado para um ambiente com GPU dedicada alocada dinamicamente, podendo variar entre Nvidia® K80, T4, P4 e P100. Esta disponibilidade de GPUs em nuvem justifica a utilização do *Google Colab* neste trabalho, acelerando significativamente os tempos de treinamento da rede neural. É importante salientar também que o *PyTorch* possibilita efetuar as operações entre tensores utilizando a tecnologia CUDA®, que otimiza a computação paralela melhorando ainda mais o tempo de treinamento.

[2]https://colab.research.google.com/
[3]http://irvlab.cs.umn.edu/resources/suim-dataset
[4]https://opencv.org/

3.1 Implementação

3.1.1 ENet

A primeira etapa para se construir o *pipeline* do algoritmo para treinamento da rede neural (assumindo que a rede esteja pronta, como é o caso) é o levantamento de dados para treinamento. Há diversos *datasets* disponíveis publicamente para fins de estudo e propósitos acadêmicos. O *dataset* escolhido é o *SUIM Dataset*, da Universidade de Minnesota. A Figura 3.2 mostra um exemplo de par de imagem/máscara contido neste *dataset*. Note que a máscara (o *ground truth*, sendo a resposta que o algoritmo deve buscar se aproximar) é composta por segmentos de *pixels* de mesma cor, que apontam para a mesma classe.

Figura 3.2: Exemplo de par de imagem/máscara disponível no *SUIM Dataset*. Fonte: http://irvlab.cs.umn.edu/resources/suim-dataset.

Para este *dataset* em específico, pode-se distinguir sem alterações na base de dados até 8 classes, detalhadas na Tabela 3.1.

Tabela 3.1: Codificação RGB das classes do *SUIM Dataset*.

Código RGB	**Cor**	**Descrição da classe**	**Num. da classe**
(0, 0, 0)	Preto	Fundo (corpo d’água)	0
(0, 0, 255)	Azul	Mergulhadores	1
(0, 255, 0)	Verde	Plantas aquáticas e ervas marinhas	2
(0, 255, 255)	Azul claro	Destroços e ruínas	3
(255, 0, 0)	Vermelho	Robôs (AUVs, ROVs e instrumentos)	4
(255, 0, 255)	Rosa	Corais e invertebrados	5
(255, 255, 0)	Amarelo	Peixes e vertebrados	6
(255, 255, 255)	Branco	Fundo do mar e rochas	7

Estes dados são processados por meio de uma classe denominada *CustomDataset*, onde alguns métodos variam de acordo com o *dataset* utilizado. A maior parte do código foi aproveitado do repositório do github da implementação do algoritmo ENet por Silva (2018), com

algumas alterações para adequar o código às necessidades deste trabalho, e para adequar a classe responsável pelo gerenciamento do *dataset* ao conjunto de dados levantados.

Como será melhor descrito adiante, a máscara codificada por *pixels* RGB deve apresentar um número de possibilidades de codificação que seja compatível ao número de classes que se deseja identificar. Um dos entraves enfrentados pela utilização do *dataset* "*as is*" (cujas imagens possuem formato *jpeg* e máscaras possuem formato *bitmap*) foi o fato de algumas máscaras apresentarem um número de *pixels* com tons RGB muito maior que oito, sendo este número o número total de classes a serem treinadas e indicadas pelo conjunto de dados sem qualquer modificação. Conforme se observa na Figura 3.3, que apresenta um *zoom* de duas máscaras do mesmo *dataset*, temos à direita um padrão bem definido de classes, onde *pixels* pretos representam o fundo (corpo d'água), vermelho representa um robô (AUV) e azul representa um mergulhador. Logo, esta máscara apresenta três classes de segmentação. À esquerda observa-se que, provavelmente devido a alguma compactação realizada durante o *upload* ou *download* do *dataset*, houve uma interpolação dos *pixels* onde diversos valores intermediários entre preto, azul e vermelho foram gerados. Desta maneira, durante a codificação da máscara de imagem para um tensor contendo *pixels* pertencentes ao intervalo [0, nclasses] (sendo *nclasses* o número de classes para segmentação), valores maiores que **nclasses** são passados a esse tensor, gerando incompatibilidade entre as etapas do *pipeline* do treinamento.

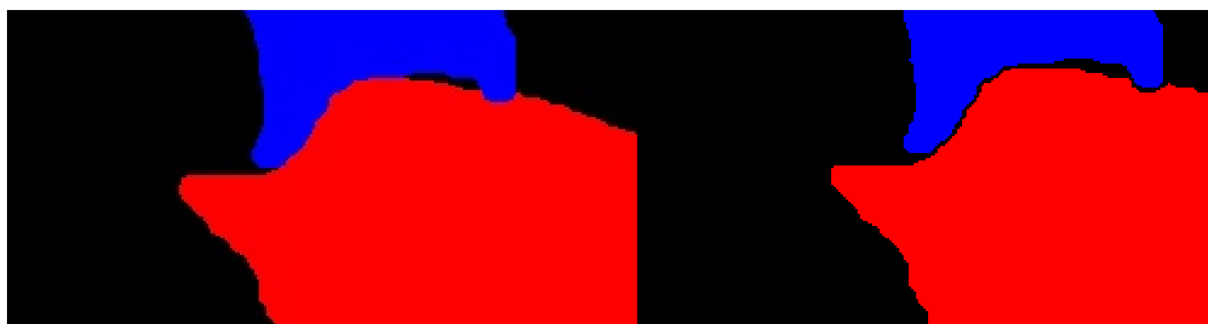

Figura 3.3: Aproximação das máscaras pré (esquerda) e pós (direita) processamento de *threshold*. Pode-se notar a diferença de nitidez entre as bordas das máscaras, devido a interpolação dos *pixels* na vizinhança entre as classes. Fonte: Autoria própria.

Um algoritmo auxiliar em *Python* foi criado para percorrer a máscara e definir um *pixel* dentre os possíveis pela codificação de classes para cada *pixel* que esteja fora do intervalo [[0, 255]][5], através de um limiar definido. Desta maneira, garante-se que todas as máscaras apresentem *pixels* somente dentro do intervalo esperado definido por [0, nclasses]. Um dicionário em *Python* é responsável por codificar as classes (de 0 a 7) com as respectivas cores (sendo essas a "chave" do dicionário).

O *dataset* foi transferido para o *Google Drive* a fim de se realizar o processamento dos dados no *Google Colab*. Embora este ambiente possa disponibilizar GPUs/TPUs (*Tensor*

[5]Zero ou duzentos e cinquenta e cinco.

Processing Unit) para uso temporário, é importante notar que a leitura de dados através do *Google Drive* é notoriamente lenta. Em todo o *pipeline* de treinamento da rede neural, este é o maior gargalo. Uma maneira de contornar este entrave é gravar arquivos `.zip` contendo os conjuntos de dados de treinamento, validação e teste, descompactando-os no ambiente de execução ("!" é utilizado para comandos externos ao *Python*, como comandos *Unix* e *MS-DOS - Microsoft Disk Operating System*), conforme o código 1 apresentado na página seguinte. Separou-se, dentro das 1525 imagens disponíveis, aproximadamente 25% para dados de validação (executada a cada dez épocas de treinamento). A diferença entre o conjunto de dados de treinamento e o conjunto de dados de validação é que o primeiro é utilizado diretamente no ajuste dos pesos da rede neural durante o processo de treinamento, enquanto que o segundo é utilizado principalmente para fornecer indícios de *overfitting* no modelo, visto que este conjunto não influi nos pesos da rede e, portanto, são "dados novos" para a rede neural. Portanto, a análise deste grupo de dados permite um melhor direcionamento no ajuste dos hiper-parâmetros (parâmetros que são ajustados antes do treinamento, através de análises empíricas ou heurísticas, e que muitas vezes são ajustados ao longo da experimentação) da rede e do algoritmo. Estes parâmetros serão melhor explicados na seção 3.1.1.3.

```
!unzip -q -n "/content/drive/My Drive/Colab Notebooks/train.zip"
!unzip -q -n "/content/drive/My Drive/Colab Notebooks/val.zip"
!unzip -q -n "/content/drive/My Drive/Colab Notebooks/test.zip"
```

Código 1: Código para descompressão dos dados do *dataset* dentro do ambiente de execução.

3.1.1.1 *Custom Dataset*

Para a classe *CustomDataset*, é necessário sobrescrever dois métodos se herdarmos a classe base *dataset* do *PyTorch* (`CustomDataset(Dataset)`):

1. `__len__`: deve ser codificado de modo a retornar o tamanho do *dataset*; e
2. `__getitem__`: dado um índice `index`, deve ser codificado de modo a retornar uma amostra do *dataset*. Neste caso, retorna um par imagem/máscara.

Há ainda um terceiro método construtor, `__init__`, que pode ser flexibilizado passando-se atributos para instanciação do objeto. Neste caso, como argumentos, utilizou-se uma função `transform`, responsável por aplicar transformações na imagem/máscara de modo a se expandir artificialmente o *dataset* (*data augmentation*), procurando evitar o problema de *overfitting*, duas *strings* contendo os caminhos para os diretórios que contém os dados de treinamento/validação/teste (imagem e máscara), e uma função auxiliar `pil_loader`, responsável por armazenar um par imagem/máscara em formato `PIL Image` (da biblioteca *Pillow* - `import PIL`).

Algumas das transformações escolhidas para a aplicação de *data augmentation* podem ser conferidas no trecho de código a seguir (adaptado do fórum da comunidade de *PyTorch*):

```
def transform(img, label, resize, crop_size):
    img = transforms.Resize((resize, resize))(img)
    label = transforms.Resize((resize, resize), Image.NEAREST)(label)

    i, j, h, w = transforms.RandomCrop.get_params(
        img, output_size=(crop_size, crop_size))
    img = TF.crop(img, i, j, h, w)
    label = TF.crop(label, i, j, h, w)

    if random.random() > 0.5:
        img = TF.hflip(img)
        label = TF.hflip(label)

    if random.random() > 0.5:
        img = TF.vflip(img)
        label = TF.vflip(label)

    img = transforms.ToTensor()(img)

    return img, label
```

Código 2: Código para transformações de imagem e máscara. Adaptado de: https://discuss.pytorch.org/t/torchvision-transfors-how-to-perform-identical-transform-on-both-image-and-target/10606/30.

Para unificar a entrada de dados, um redimensionamento do tamanho da imagem e máscara é realizado, além de se diminuir substancialmente o tempo de treinamento (a depender dos tipos de imagens a serem utilizados, normalmente se sugere iniciar com tamanhos pequenos de *input size*, como 224x224). Como filtro de *resize* para a máscara, o método *NEAREST* é escolhido por não gerar *pixels* através de uma interpolação linear, por exemplo, escolhendo-se apenas um existente dentre a vizinhança. Deste modo, preserva-se a codificação de classes da máscara.

Como dito anteriormente, as outras funções foram incluídas com o propósito de se minimizar o *overfitting* do modelo, introduzindo certa aleatoriedade nas transformações das imagens e máscaras para evitar que o mesmo se atenha a características não desejáveis e que desfavoreçam a generalização do modelo. `RandomCrop` retorna parâmetros de recorte para realizar o mesmo *crop* (neste caso, de 256x256, com redimensionamento prévio para 512x512) tanto na imagem

quanto na máscara. É importante garantir que o conjunto de transformações (especialmente o `RandomCrop`) se aplique de maneira igual em ambos. Após o *crop*, com probabilidade de 0.5, são efetuados *flips* horizontais e verticais, novamente para se introduzir aleatoriedade no conjunto de dados. Por fim, a imagem é convertida para tensor, a fim de ser ingerida propriamente pela rede neural posteriormente. A máscara será convertida para tensor após a execução do método que codifica os *pixels* RGB de acordo com as classes, descrita pelo código a seguir:

```
def mask_to_class_rgb(mask, mapping):
    mask = torch.from_numpy(np.array(mask))
    mask = torch.squeeze(mask)

    class_mask = mask
    class_mask = class_mask.permute(2, 0, 1).contiguous()
    h, w = class_mask.shape[1], class_mask.shape[2]
    mask_out = torch.empty(h, w, dtype=torch.long)

    for key, color in mapping.items():
        idx = (class_mask == torch.tensor(color, dtype=torch.uint8)
               .unsqueeze(1).unsqueeze(2))
        validx = (idx.sum(0) == 3)
        mask_out[validx] = torch.tensor(key, dtype=torch.long)

    return mask_out
```

Código 3: Função de conversão dos *pixels* RGB de acordo com a respectiva classe. Adaptado de: https://discuss.pytorch.org/t/semantic-segmentation-how-to-map-rgb-mask-in-data-loader/72130.

Esta função transforma a máscara, que até então possuía dimensão [H, W, C], onde H é a altura da imagem, W é a largura da imagem e C é o número de canais de cores que, no caso, $C = 3$, para uma máscara codificada em classes de dimensões [H, W], onde cada valor que representa um *pixel* neste tensor apresenta a classe codificada pela respectiva cor, onde estes valores pertencem ao intervalo [0, **nclasses**]. Este tensor será posteriormente passado como argumento para funções do treinamento.

3.1.1.2 *Dataloader*

O *DataLoader* é um iterador disponível pelo *PyTorch* que permite percorrer as imagens e máscaras dos conjuntos de treinamento, validação e teste e transportá-las em *batches* para os

pipelines de treinamento e predição (quando a rede é testada). São passados quatro parâmetros:

1. *transformed dataset*: são os dados do *dataset* pós-transformações, do tipo *Dataset*. Neste caso, passa-se, para o treinamento, o endereço de diretório que contém os dados de treinamento;
2. *batch size*: número de pares imagem/máscara fornecidos a cada iteração para a rede, do tipo *inteiro*. Será melhor elucidado adiante;
3. *shuffle*: *booleano* que determina se a ordem de formação dos *batches* será aleatória dentro do conjunto; e
4. *number of workers*: inteiro que determina o número de processos para paralelização (*multi-process*).

O *batch size*, como dito anteriormente, é uma amostra retirada do conjunto de dados, seja de treinamento, seja de validação e teste. A rede neural é treinada para minimizar a função de perda através do gradiente descendente. O chamado Gradiente Descendente Estocástico calcula o gradiente em uma amostra (*batch size*) do conjunto de dados. Esta amostragem dos dados permite uma convergência mais rápida do algoritmo de treinamento, visto que o gradiente destes *batches* representam uma aproximação do gradiente global do *dataset* e permite que os pesos da rede possam ser ajustados pouco a pouco, em detrimento de se atualizar estes valores somente após percorrer todo o conjunto de dados.

O *batch size* é um dos hiper-parâmetros do algoritmo. É um valor escolhido através de análises de resultados prévios, análises experimentais ou através de heurísticas. O tamanho da amostra influi diretamente na velocidade de treinamento, visto que pode ser paralelizado através dos *workers*, como também afeta na capacidade de generalização do modelo para o conjunto de dados de teste (onde, normalmente, lotes menores garantem melhor generalização). Além disso, grandes *batches* levam mais tempo para convergir para o mínimo global da função de perda. Já lotes menores podem não garantir a convergência para o mínimo global, buscando muitas vezes soluções não-ótimas. Portanto, é necessário ponderar o tamanho do *batch size* de acordo com os objetivos e necessidades da aplicação da rede.

3.1.1.3 *Hiper-parâmetros* do algoritmo

Como dito anteriormente, são parâmetros ajustados através de resultados prévios ou através da experimentação. Os outros hiper-parâmetros da rede são:

- *Weight Decay*: define uma penalidade que é aplicada aos pesos a cada atualização dos mesmos, multiplicando-os por um número pouco menor que um. Pode contribuir para reduzir o *overfitting* e mantém os pesos da rede em níveis pequenos. Manteve-se, aqui, o valor deste parâmetro igual ao recomendado pelo trabalho da ENet: **2e-4**;

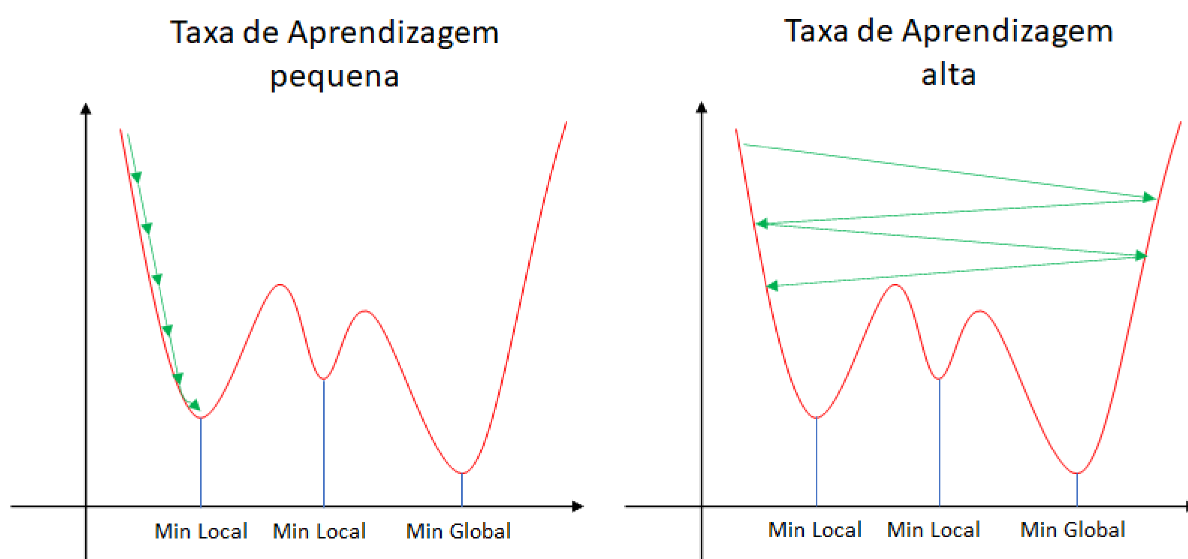

Figura 3.4: Representação simplificada do conceito de taxa de aprendizado. Fonte: Autoria própria.

- *Learning rate*: é a taxa de aprendizado como parâmetro para aplicação do Gradiente Descendente, responsável por ponderar o quanto se atualiza os valores da rede a cada iteração. Uma fácil explicação que tende a guiar um bom ajuste do *learning rate* é: suponha que a função custo que deva ser minimizada possua uma superfície extremamente irregular, com diversos mínimos locais. Uma taxa de aprendizagem pequena faz com que o gradiente se mova lentamente na direção de maior variação (no caso, na direção do mínimo). Isto significa que a função pode ficar facilmente "presa" a um desses mínimos locais, não tendo "energia" o suficiente para escapar destes e buscar um mínimo global. Isto provoca a impossibilidade de atingir o resultado ótimo, além de contribuir para o *overfitting*. Já uma taxa de aprendizagem alta, a função pode saltar de uma maneira caótica que também impossibilita a localização do mínimo global, o que requere um ajuste mais fino perto do seu menor valor. Isso significa que deve haver um compromisso na escolha do *learning rate*. A Figura 3.4 demonstra uma representação simples desta ideia. Uma boa estratégia de se escolher o valor é começar com uma taxa relativamente alta (como 0.001 ou até 0.005) e ajustá-la ao longo das épocas de treinamento. Neste trabalho, os resultados mais consistentes foram obtidos com *learning rate* de **0.001**;
- Épocas (*epochs*): é o número de vezes que o algoritmo irá percorrer o conjunto de dados de treinamento durante o processo de aprendizagem. Portanto, x épocas garantem que cada *batch* de dados contribua x vezes para o ajuste de peso da rede neural. Lembrando que o *dataset* é divido em um dado número de *batches* para o processo de iteração. Ocorre, portanto, outro processo iterativo a cada época que percorre o conjunto de

dados um número de vezes tal que `i = len(dataset) / batch_size`. Durante a experimentação, variou-se este parâmetro entre 100 e 300 para análise dos resultados, onde treinamentos com 200 e 300 épocas apresentaram os resultados mais consistentes.

3.1.1.4 Pesos de classes, critérios de perda e otimizador

O *dataset* utilizado possui até oito classes para treinamento. Dependendo do *dataset* utilizado no treinamento (como é o caso deste), pode haver certo desbalanceamento na quantidade de aparições de cada classe. Para a maioria dos algoritmos de aprendizado de máquina, assume-se que a quantidade de dados esteja igualmente distribuída dentro de suas classificações. Nestes casos, um *dataset* desbalanceado tende a provocar um certo enviesamento (*bias*) do classificador na direção das classes mais abundantes. Desta maneira, tanto o treinamento quanto a predição do algoritmo ficam prejudicados.

Aqui, usou-se a mesma estratégia descrita por Paszke et al. (2016) para atribuição dos pesos de balanceamento, conforme segue:

$$w_{class} = \frac{1}{ln(c + p_{class})} \tag{3.1}$$

onde:

c é um hiper-parâmetro adicional fixado em 1.02;

p_{classe} é a probabilidade da classe.

Como critério de perda para o treinamento, tratando-se de uma classificação multi-classe, escolheu-se o *cross entropy loss* disponível pelo *PyTorch*. A função de custo de entropia cruzada é definida conforme equação 3.2, observando-se a presença do termo de peso de classes (*class weight) (CROSSENTROPYLOSS, 2020)*:

$$loss(x, class) = weight[class](-x[class] + log(\sum_j exp(x[j]))) \tag{3.2}$$

onde:

x é a saída da predição da rede;

$class$ é a classe verdadeira;

$weight$ é o peso atribuído à classe.

Como o treinamento é executado em *batches*, uma média das perdas é feita a cada iteração.

O otimizador escolhido foi o *Adam*, também disponibilizado pelo *PyTorch*. Os otimizadores são algoritmos e métodos utilizados para a alteração dos atributos da rede neural (como a taxa de aprendizado e os pesos da rede) de modo a se minimizar a função custo. O otimizador *Adam* pode ser considerado uma extensão do Gradiente Descendente Estocástico (SGD, *Stochastic*

Gradient Descent, que também pode ser escolhido como um otimizador pelo *PyTorch*). Um destaque deste algoritmo é o ajuste individual da taxa de aprendizado para cada peso da rede, enquanto que o SGD mantém um valor fixo para toda a rede. De acordo com Brownlee (2017), este algoritmo calcula uma média móvel exponencial do gradiente e do gradiente ao quadrado controlando a variação destes através de parâmetros pré-estabelecidos β_1 e β_2. Esta estratégia e outras implementadas dentro do *Adam* garante que ele seja um otimizador bastante eficiente e de rápida convergência, frequentemente utilizado em problemas de visão computacional.

O *Adam* aceita os seguintes parâmetros:

- *Alpha*: é a própria taxa de aprendizado;
- β_1: é a taxa de decaimento das estimativas de primeiro-momento;
- β_2: é a taxa de decaimento das estimativas de segundo-momento;
- ε: é um número quase nulo que previne divisões por zero durante a implementação.

Para os testes conduzidos neste trabalho, manteve-se os valores de β_1, β_2 e ε padrões da implementação do *PyTorch*, isto é, $\beta_1 = 0.9$, $\beta_2 = 0.999$ e $\varepsilon = 10^{-8}$.

3.1.1.5 *Intersection over Union*

A métrica utilizada por Paszke et al. (2016) para seleção dos melhores modelos a cada etapa de validação (e também adotada neste trabalho) é a razão interseção/união (*Intersection over Union*, também conhecido como Índice de Jaccard). É uma métrica onde as predições são acumuladas em uma matriz de confusão e o IoU é calculado como segue:

$$IoU = \frac{Verdadeiro\,Positivo}{Verdadeiro\,Positivo + Falso\,Positivo + Falso\,Negativo} \tag{3.3}$$

onde:

Verdadeiro Positivo: é uma predição X do classificador onde o resultado é realmente X;

Falso Positivo: é uma predição X do classificador onde o resultado é, na verdade, Y, sendo Y qualquer resultado que não X;

Falso Verdadeiro: é uma predição do classificador tal que considera um ponto sendo não-X e, de fato, não é X; e

Falso Negativo: é uma predição do classificador tal que considera um ponto sendo não-X, porém, o resultado é X (SKANSI, 2018).

Estes valores, como dito, são tabelados na matriz de confusão. Esta, por sua vez, é uma matriz que mapeia as frequências de classificação para cada classe que o modelo deve predizer. A Tabela 3.2 exemplifica o formato geral de uma matriz de confusão 2x2 (utilizada em classificações binárias):

Tabela 3.2: Exemplo de matriz de confusão 2x2.

	Predito: SIM	**Predito: NÃO**
Verdadeiro: SIM	Número de Verdadeiros Positivos	Número de Falsos Negativos
Verdadeiro: NÃO	Número de Falsos Negativos	Número de Falsos Verdadeiros

Por meio desta matriz, diversas métricas podem ser extraídas, como *recall* e o *F-Score*, além do próprio IoU, que servem como indicadores de performance do modelo experimentado.

3.1.2 K-Means

O algoritmo de *K-Means* utilizado para a comparação aqui proposta foi implementado por meio do pacote *OpenCV* e o seguinte *pipeline* foi implantado:

1. A imagem a ser segmentada é carregada por meio da função `cv2.imread` e posteriormente convertida para o espaço de cores RGB (a função `imread` carrega a imagem em BGR);
2. Para um melhor contraste da imagem e melhor definição dos contornos, o canal α é alterado para 2 através de `cv2.addWeighted`, utilizando a mesma imagem para a composição e com β e γ setados para 0. Este valor foi testado empiricamente e pode não ser ótimo, onde uma análise mais aprofundada deve ser conduzida para se levantar tal otimização;
3. A imagem é transformada em um *array* 2D RGB por meio de `reshape((-1,3))` e posteriormente convertida para `float32`;
4. O critério de parada adotado foi tanto por acurácia quanto por número máximo de iterações, com `cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER`, sendo *max_iter* igual a 5 e *Epsilon* igual a 0.1;
5. Número de *clusters* foi setado para 3 (a comparação foi realizada utilizando apenas três classes dentre as oito possíveis, sendo melhor explicado no capítulo 4);
6. Número de tentativas selecionado para 10;
7. Localização dos centróides iniciais aleatórios (`cv2.KMEANS_RANDOM_CENTERS`);
8. Os centros `(centers)` retornados pela função `cv2.kmeans` são então convertidos para `uint8` e os *labels* retornados são transformados para um *array* 1D através de `np.flatten()`;
9. A imagem segmentada é lida por meio de `centers[labels]` e transformada para o formato da imagem original através de `reshape(img.shape)`.

O código utilizado pode ser conferido integralmente a seguir:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

imgname = 'd_r_166_.jpg'
img = cv2.imread(imgname)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
img = cv2.addWeighted(img, 2, img, 0, 0)
plt.imshow(img)

pixel_values = img.reshape((-1, 3))
pixel_values = np.float32(pixel_values)
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 5, 0.1)
_, labels, (centers) = cv2.kmeans(pixel_values, 3, None, criteria, 10,
                                  cv2.KMEANS_RANDOM_CENTERS)

centers = np.uint8(centers)
labels = labels.flatten()
segmented_image = centers[labels]
segmented_image = segmented_image.reshape(img.shape)

plt.imshow(segmented_image)
plt.show()
```

Código 4: Algoritmo para implementação da técnica de clusterização por *K-Means*, via OpenCV.

CAPÍTULO 4

Resultados e Discussão

Neste capítulo são apresentados os resultados obtidos com as experimentações descritas no Capítulo 3, juntamente com as discussões pertinentes. Primeiramente, são descritas as tentativas iniciais de treinamento, onde evidenciou-se uma falha na separação dos conjuntos de treinamento e validação, através da análise das curvas de aprendizado. Posteriormente, são apresentados os resultados dos processos de treinamento com duzentas e trezentas épocas, ambas utilizando todas as oito classes disponíveis para treinamento. Por fim, os resultados dos processos de treinamento para quatro e três classes por duzentas épocas são apresentados, seguidos dos resultados e discussão acerca da implementação do algoritmo *K-Means*.

4.1 Rede Neural

Os primeiros testes de treinamento funcionais para a rede ENet mostraram uma clara tendência de aprendizado, conforme Figura 4.1 nos gráficos de *Train Epoch Loss* e *Train Epoch MIoU*, através do decréscimo e do aumento, respectivamente, da perda e do *MIoU*.

Porém, é importante notar que, com base nas curvas de validação (a validação ocorre a cada dez épocas, correspondendo a uma década), havia uma tendência contrária de perda e uma instabilidade na qualidade da predição, evidenciando uma falha no processo. Uma análise mais cuidadosa permite propor certas hipóteses que levariam a possíveis soluções face o problema detectado.

Considerando que a rede neural executa a etapa de *backpropagation* somente no conjunto de dados de treinamento e, portanto, não aprende com o *set* de validação (este é utilizado apenas para verificação da performance da rede para posterior refinamento dos parâmetros de

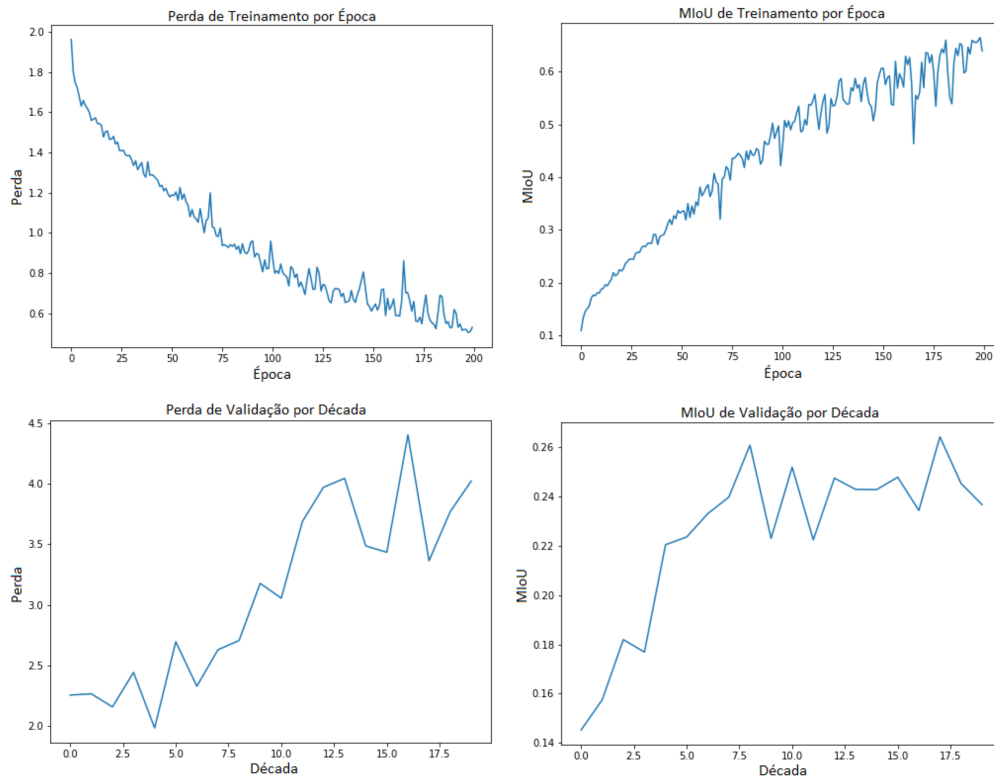

Figura 4.1: Curvas de perda e *MIoU* de treinamento e validação com *set* de validação enviesado. Fonte: Autoria própria.

treinamento), e observando uma tendência de alta na perda do conjunto de dados de validação, evidenciou-se uma tendência de *overfitting*. Outro fator encontrado foi o *IoU* de uma classe específica zerada. Uma hipótese levantada foi o desbalanceamento de classes, visto que a rede teria se adaptado bem aos dados de treinamento, porém não conseguia generalizar para dados nunca vistos. Embora o *data augmentation* possa prevenir isso, a classe com *IoU* zerada indica uma insuficiência de dados de aprendizado para esta. Por fim, verificou-se que a separação dos dados do *dataset* entre treinamento e validação fora feito com grande desbalanceamento de classes: em especial, no conjunto de validação, notou-se um número extremamente alto de aparições da classe problemática. Como havia um “viés” acentuado nesta separação, poucos dados haviam sido disponibilizados para treinamento desta classe, enquanto que os dados de teste apresentavam muito mais repetições da mesma. Conforme a rede aprende a identificar as outras classes, ajustando pesos da rede para detectá-las, pouco peso é atribuído à classe problemática. Há, então, uma diminuição da perda de treinamento em detrimento ao aumento da perda de validação.

Após a melhor separação dos dados entre os conjuntos, obteve-se curvas mais condizentes com o processo de aprendizado esperado, onde ambas as curvas de perda tendem a diminuir, enquanto que as curvas de *MIoU* apresentam melhora. As Figuras 4.2 e 4.3 apresentam dois processos de treinamento, onde variou-se o número de épocas, somente, para verificação da

influência deste hiper-parâmetro na eficiência do aprendizado.

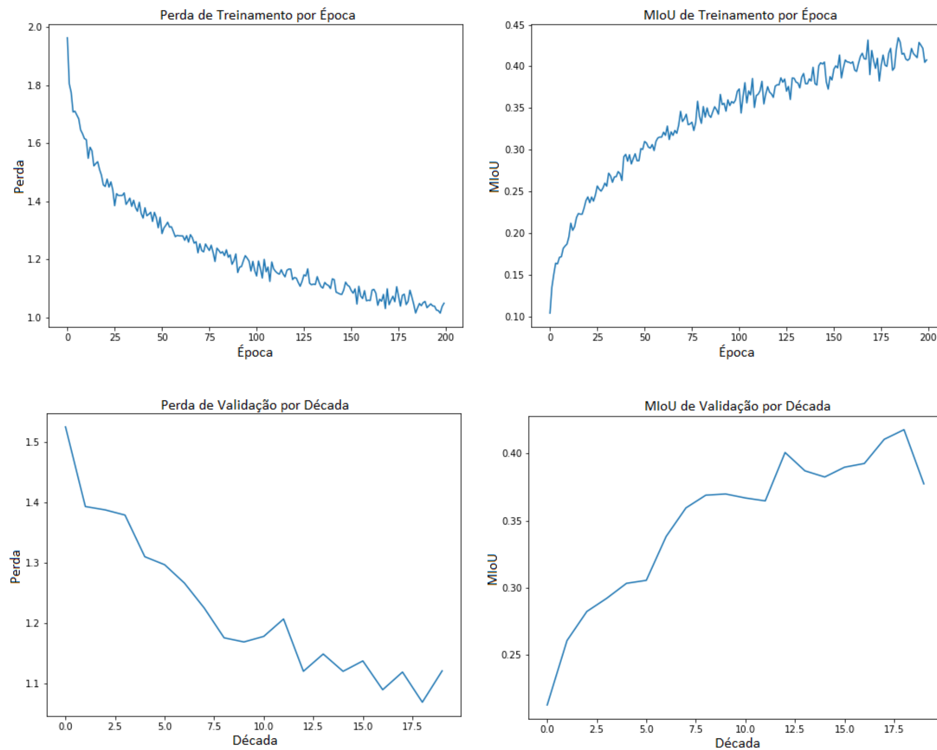

Figura 4.2: Curvas de perda e *MIoU* de treinamento e validação, por duzentas épocas. Fonte: Autoria própria.

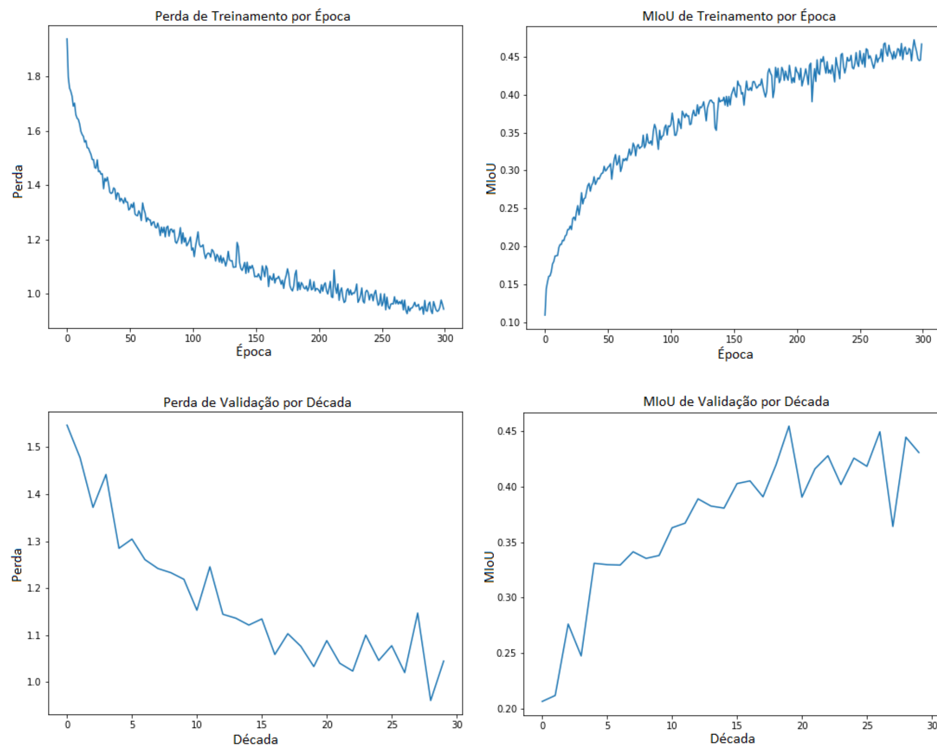

Figura 4.3: Curvas de perda e *MIoU* de treinamento e validação, por trezentas épocas. Fonte: Autoria própria.

Com base nos modelos levantados, foi fornecido, para cada, um *batch* de dez imagens para predição e verificação da performance obtida. As Figuras 4.4 e 4.5 apresentam, respectivamente, a predição dos modelos treinados depois de duzentas e trezentas épocas.

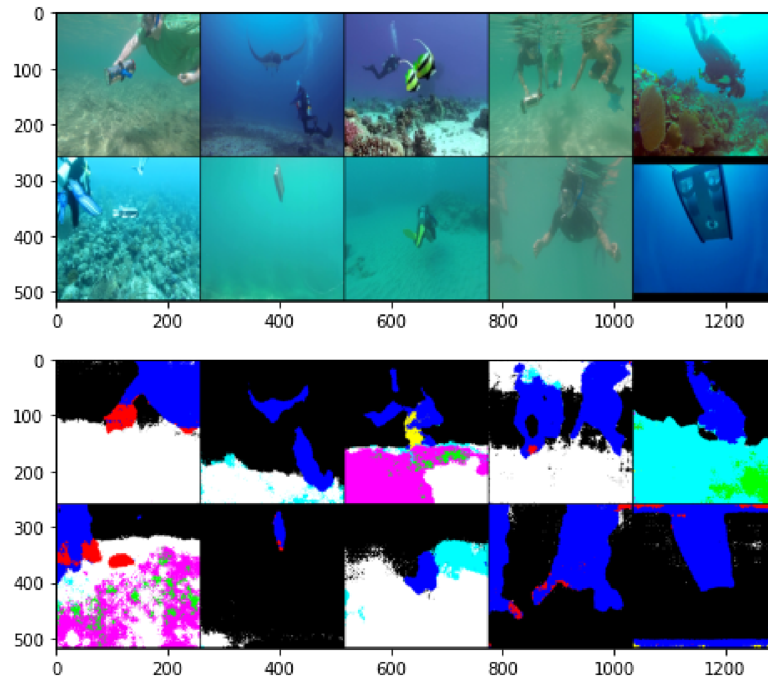

Figura 4.4: Predição das imagens de teste pelo modelo treinado com duzentas épocas. Fonte: Autoria própria.

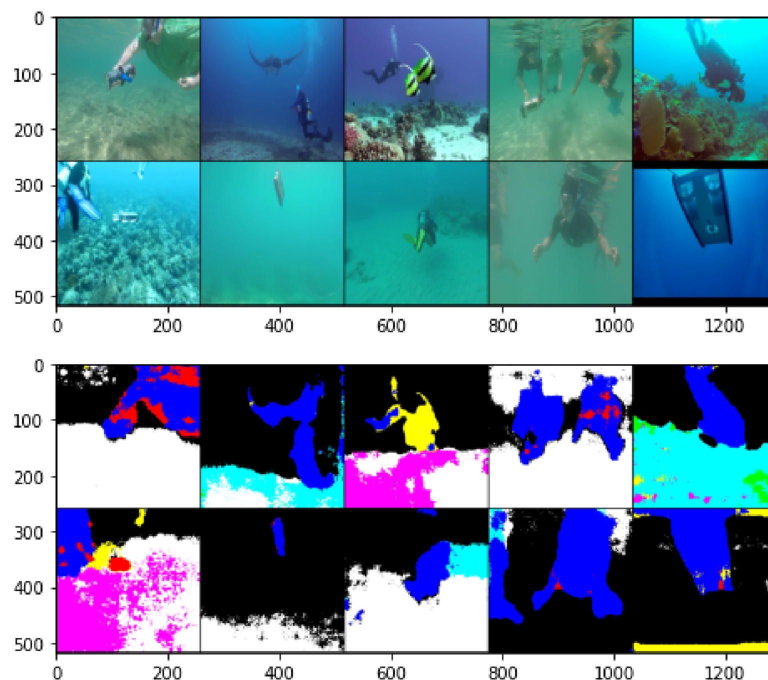

Figura 4.5: Predição das imagens de teste pelo modelo treinado com trezentas épocas. Fonte: Autoria própria.

É possível notar que os treinamentos apresentam certa tendência de convergência ao fim do número de épocas, com perdas de treinamento próximas a um e o *score* de *MIoU* atingindo até 45%, com 39% de *MIoU* para o conjunto de teste (110 imagens). Para o caso da aplicação por Paszke et al. (2016), no *dataset Cityscapes*, que possui 2975 imagens de treinamento e 19 classes de predição, foi obtida uma média de 68,3%. Face estes números, os resultados podem ser considerados não tão satisfatórios. Uma hipótese levantada é o número relativamente elevado de classes para um *dataset* não extenso o suficiente. Embora a técnica de *data augmentation* tenha sido utilizada ainda que de forma básica, a quantidade de dados de treinamento precisa ser relevante para um processo adequado de aprendizado.

Com base nesta hipótese, e voltando-se para uma aplicação que priorizaria a prevenção de colisões durante navegação de um ROV/AUV, foram aglutinadas classes de treinamento que apresentam semelhanças tais que justifiquem esta ação. Desta maneira, dois outros processos de treinamento foram realizados para o mesmo conjunto de dados de aprendizado, diminuindo-se para quatro e, posteriormente, três classes a serem treinadas. O seguinte algoritmo foi utilizado para aglutinar cores presentes nas máscaras a fim de se diminuir o número de classes.

Este algoritmo lê as imagens das máscaras e as converte em *arrays* do *numpy*. Então, é criada uma sub-máscara binária de mesmas dimensões das máscaras onde, para cada *pixel* da imagem, é atribuído `True` para os *pixels* que se encaixem dentro de determinada condição (neste caso, que sejam das cores correspondentes às classes que queremos aglutinar). Assim, atribui-se uma outra cor cuja codificação seja referente à classe a qual queremos reclassificar. Por exemplo, *pixels* que identificam a classe de “peixes e vertebrados”, de cor [255, 255, 0], são remapeadas para a classe de “mergulhadores humanos”, de cor [0, 0, 255], através da atribuição da cor azul aos *pixels* anteriormente pertencentes ao amarelo.

```
import glob
from PIL import Image
import numpy as np
import os

masks_dir = glob.glob(os.getcwd()+"\\test\\masks\\*.bmp")
lim = 200

for img_path in masks_dir:
    img = Image.open(img_path)
    arr = np.array(np.asarray(img))

    valid_yellow_range = np.logical_and(np.logical_and(arr[:,:,0] > \
        lim, arr[:,:,1] > lim), arr[:,:,2] <= lim)
    valid_magenta_range = np.logical_and(np.logical_and(arr[:,:,0] > \
        lim, arr[:,:,2] > lim), arr[:,:,1] <= lim)
    valid_cyan_range = np.logical_and(np.logical_and(arr[:,:,1] > \
        lim, arr[:,:,2] > lim), arr[:,:,0] <= lim)
    valid_red_range = np.logical_and(np.logical_and(arr[:,:,0] > \
        lim, arr[:,:,1] <= lim), arr[:,:,2] <= lim)
    valid_green_range = np.logical_and(np.logical_and(arr[:,:,0] <= \
        lim, arr[:,:,1] > lim), arr[:,:,2] <= lim)

    arr[valid_yellow_range] = [0, 0, 255]
    arr[valid_red_range] = [0, 0, 255]
    arr[valid_green_range] = [255, 255, 255]
    arr[valid_magenta_range] = [255, 255, 255]
    arr[valid_cyan_range] = [255, 255, 255]

    outputimg = Image.fromarray(arr)
    outputimg.save(img_path)
```

Código 5: Algoritmo para agrupamento de classes de máscaras

Os agrupamentos de classes realizados puderam ser sintetizados conforme descrevem as Tabelas 4.1 e 4.2.

Tabela 4.1: Mapeamento de classes após redução para quatro classes.

Classificação original			Classificação com 4 classes	
Descrição da classe	**Código RGB**	**Índice**	**Código RGB Remapeado**	**Índice**
Fundo (corpo d'água)	(0, 0, 0)	0	(0, 0, 0)	0
Mergulhadores	(0, 0, 255)	1	(0, 0, 255)	1
Plantas aquáticas e ervas marinhas	(0, 255, 0)	2	(255, 255, 255)	7
Destroços e ruínas	(0, 255, 255)	3	(0, 255, 255)	3
Robôs (AUVs, ROVs e instrumentos)	(255, 0, 0)	4	(0, 0, 255)	1
Corais e invertebrados	(255, 0, 255)	5	(255, 255, 255)	7
Peixes e vertebrados	(255, 255, 0)	6	(0, 0, 255)	1
Fundo do mar e rochas	(255, 255, 255)	7	(255, 255, 255)	7

Tabela 4.2: Mapeamento de classes após redução para três classes.

Classificação original			Classificação com 4 classes	
Descrição da classe	**Código RGB**	**Índice**	**Código RGB Remapeado**	**Índice**
Fundo (corpo d'água)	(0, 0, 0)	0	(0, 0, 0)	0
Mergulhadores	(0, 0, 255)	1	(0, 0, 255)	1
Plantas aquáticas e ervas marinhas	(0, 255, 0)	2	(255, 255, 255)	7
Destroços e ruínas	(0, 255, 255)	3	(255, 255, 255)	7
Robôs (AUVs, ROVs e instrumentos)	(255, 0, 0)	4	(0, 0, 255)	1
Corais e invertebrados	(255, 0, 255)	5	(255, 255, 255)	7
Peixes e vertebrados	(255, 255, 0)	6	(0, 0, 255)	1
Fundo do mar e rochas	(255, 255, 255)	7	(255, 255, 255)	7

Estas reduções garantiram uma melhora nos índices de performance da rede neural, apresentando, para um treinamento considerando quatro classes, uma perda de treinamento próxima a 0,60 e uma perda de validação entre 0,60 e 0,70, enquanto que o índice *MIoU* se aproximou de 0,60 para as melhores décadas de treinamento (a validação ocorre a cada dez épocas - uma década) e 0,54 para o conjunto de teste. Já para o treinamento considerando três classes, as perdas de treinamento e validação se aproximaram de 0,45, enquanto que o *MIoU* se aproximou de 0,70 para as melhores décadas de treinamento (o algoritmo salva o melhor ajuste de pesos a cada década, mantendo a melhor performance obtida até então). Já o *MIoU* para as 110 imagens de teste ficou em 0,65. As Figuras 4.6 e 4.7 demonstram os resultados alcançados através destes processos de treinamento.

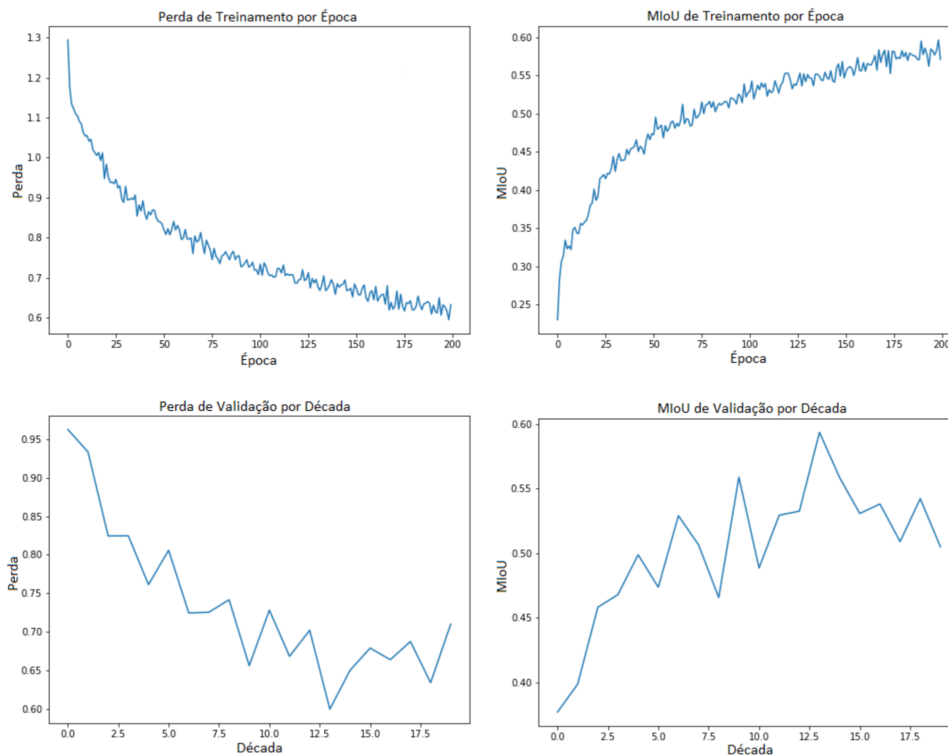

Figura 4.6: Curvas de perda e *MIoU* de treinamento e validação, por duzentas épocas, para um processo de treinamento de predição de quatro classes. Fonte: Autoria própria.

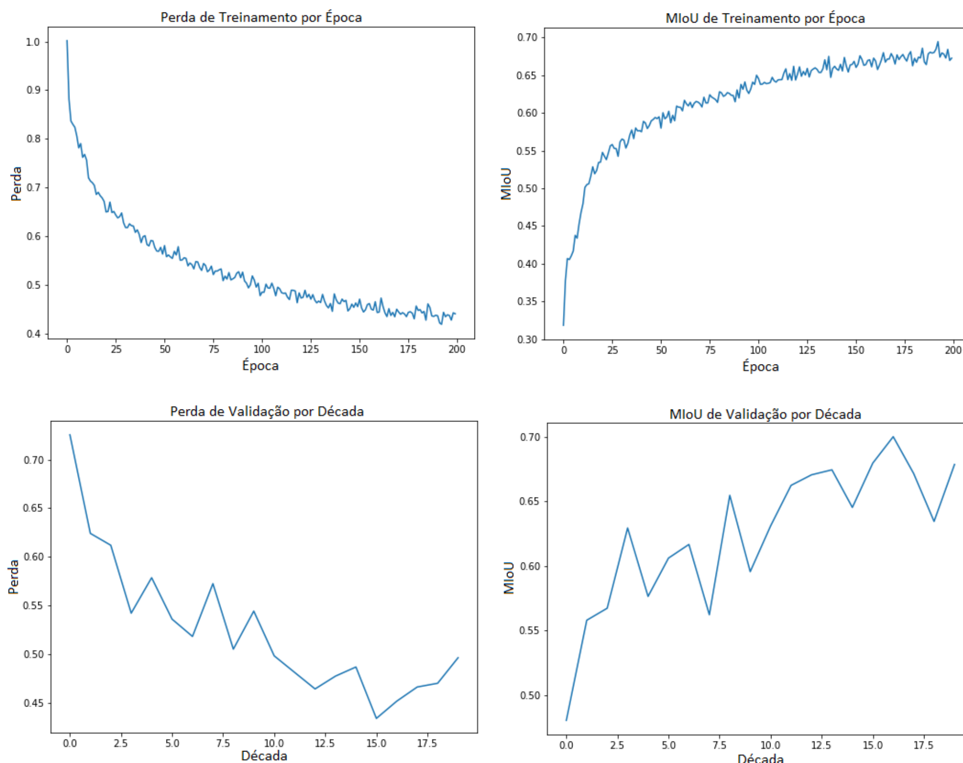

Figura 4.7: Curvas de perda e *MIoU* de treinamento e validação, por duzentas épocas, para um processo de treinamento de predição de três classes. Fonte: Autoria própria.

Já as Figuras 4.8 e 4.9 apresentam as predições de classes para um conjunto de dez imagens

de teste, considerando, respectivamente, quatro classes e três classes de treinamento.

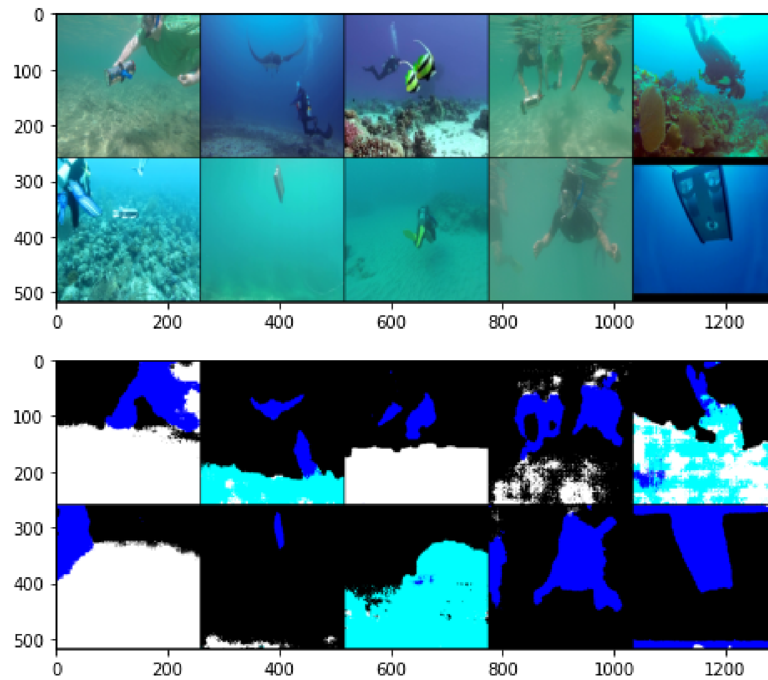

Figura 4.8: Predição das imagens de teste pelo modelo treinado considerando quatro classes. Fonte: Autoria própria.

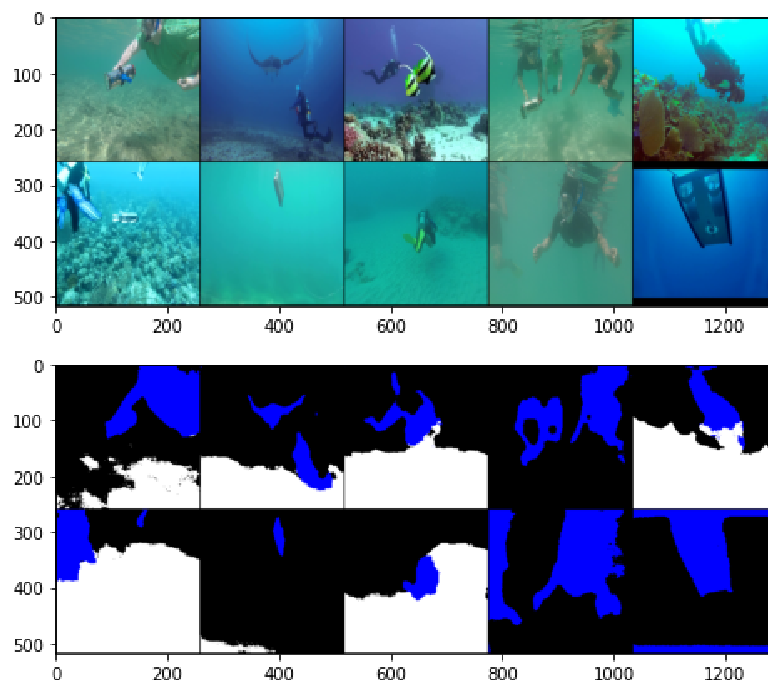

Figura 4.9: Predição das imagens de teste pelo modelo treinado considerando três classes. Fonte: Autoria própria.

Estes resultados podem ser melhorados em futuros trabalhos através do aprimoramento da rede neural, após uma análise de sua estrutura e otimização de camadas que agreguem

maior capacidade de extração das *features* necessárias para o reconhecimento dos contornos em imagens subaquáticas. Um outro ponto de melhoria também pertinente seria a aplicação de estratégias de pré-processamento a fim de se reduzir o ruído das imagens e buscar um maior contraste de cores, ou outras técnicas que procurem acentuar os contornos dos objetos a serem identificados. Também vale citar que os resultados podem ser melhor aprimorados a partir do enriquecimento do *dataset*, desde que o conjunto seja extenso e representativo o suficiente para uma maior acurácia. Assim, uma rede neural mais adequada (mais profunda?) e compatível com o tamanho do *dataset* construído (maior), aliado a estratégias de pré-processamento que permitam uma melhor identificação dos *features* desejados das imagens se apresenta como uma proposta de melhoria para eventuais futuros trabalhos de temas convergentes a este.

4.1.1 Tempo de Processamento

Para medir o tempo de processamento da rede neural ENet na aplicação deste trabalho, optou-se por utilizar a metodologia descrita por Geifman (2020). Nela, uma entrada *dummy*, isto é, um tensor que simula um *input* de dados (imagem), é utilizada para realizar um "aquecimento" da GPU e fazer com que esta esteja pronta e disponível para o processamento real dos dados desejados, em um estado de utilização pleno que não esteja conservando energia, e sim pronto para utilizar todo o potencial de processamento gráfico disponível. Depois desta pré-inicialização, por meio dos métodos `torch.cuda.Event` e `torch.cuda.synchronize`, é medido o tempo de inferência da rede neural. Neste caso, um conjunto de *batch* de dez imagens foi alimentado na rede neural e uma média foi realizada. O tempo de inferência médio obtido foi de 17,74 milissegundos, o que garante uma taxa de 56 quadros por segundo aproximadamente. Vale lembrar que a biblioteca *PyTorch* é otimizada para trabalhar com GPUs e também para operar com a tecnologia CUDA®, acelerando significativamente os tempos de processamento do algoritmo.

4.2 K-Means

Para fins comparativos, executou-se o algoritmo *K-Means*, disponível no pacote *OpenCV* para *Python*, para três imagens escolhidas aleatoriamente dentre o subconjunto de dados de teste do *dataset* utilizado. Os parâmetros do algoritmo podem ser conferidos na seção 3.1.2.

A Figura 4.10 mostra, na linha superior, as imagens escolhidas para a comparação; na linha seguinte, o *ground truth* das respectivas imagens; na terceira linha, o resultado obtido pela implementação do algoritmo de clusterização *K-Means*; e, na última linha, o resultado obtido pela implementação do algoritmo de CNN.

Vale notar que o algoritmo de clusterização baseia-se tão somente na distância dos pontos aos seus respectivos centróides (aleatoriamente dispostos conforme o número de centróides K

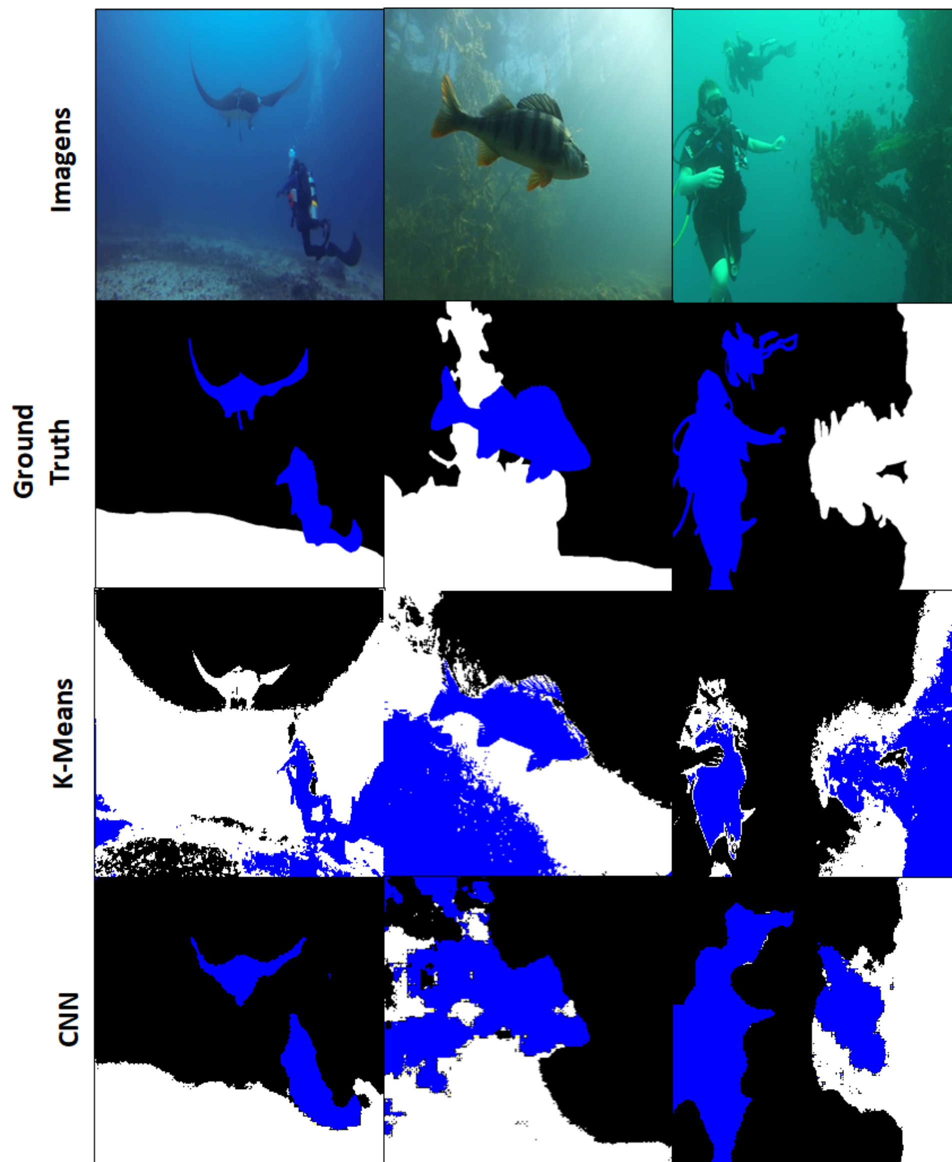

Figura 4.10: Comparação das técnicas de segmentação por *K-Means* e CNN. Fonte: Autoria própria.

escolhido). Partindo deste fato, é notório que uma desvantagem deste método é justamente a incapacidade de se distinguir as classes na imagem, também por se tratar de um algoritmo não supervisionado. As imagens mostradas acima foram processadas posteriormente atribuindo-se as classes que mais faziam sentido face a distribuição dos *pixels* resultante da aplicação do *K-Means*. Como a rede neural busca extrair características que não restringem-se apenas na posição relativa dos *pixels*, é um algoritmo supervisionado e utilizado para segmentação semântica, podendo discernir entre classes.

Em uma análise qualitativa, o algoritmo *K-Means* pôde identificar os contornos dos objetos de interesse quando há uma certa nitidez e contraste com relação aos seus entornos, embora não consiga separar entre classes. Percebe-se, assim, que as plantas aquáticas de fundo da segunda imagem e o mergulhador ao fundo da terceira imagem não foram segmentados (a primeira

situação também ocorre no caso da rede neural). Assim, a aplicação deste algoritmo de maneira isolada, em um primeiro momento, não se mostra tão vantajosa a ponto de justificar seu uso, sem qualquer tipo de pré-processamento. Porém, é possível e plausível que a associação do *K-Means* com outras técnicas ou com imagens de maior nitidez (requerendo, portanto, um processamento prévio para tal) possa apresentar bons resultados. Já para o algoritmo de CNN, para a primeira imagem, a rede neural pôde identificar de maneira satisfatória os contornos das classes. Já na segunda imagem, embora o peixe tenha sido corretamente identificado, os contornos não foram precisamente definidos, apresentando certa distorção entre classes. Na terceira imagem, os mergulhadores puderam ser identificados e corretamente classificados, porém parte dos corais foram segmentados como pertencentes à uma classe errada. As observações feitas no fim da seção anterior podem ser futuramente investigadas de tal forma a se melhorar este desempenho.

Quantitativamente, utilizando-se a métrica *MIoU* das classes, apenas para as três imagens selecionadas, a técnica por CNN demonstra vantagem, com uma média de de 71,57%, frente a 39,00% da técnica por *K-Means*, conforme demonstra a tabela 4.3.

Tabela 4.3: Métrica *MIoU* para as imagens testadas por ambas as técnicas.

Técnica	**MIoU**			
	1ª Imagem	**2ª Imagem**	**3ª Imagem**	**Média**
CNN (ENet)	81,00%	62,10%	71,61%	71,57%
K-Means	33,46%	39,24%	44,30%	39,00%

4.2.1 Tempo de Processamento

Novamente, para fins comparativos, mediu-se o tempo de execução do algoritmo *K-Means* para um *input* de uma imagem, conforme os parâmetros ajustados e descritos anteriormente. Para as três imagens de teste utilizadas, obteve-se um tempo médio de 100,66 milissegundos. Vale salientar aqui que a versão disponível para uso do *OpenCV* no *Google Colab* é a 4.1.2, não apresentando suporte à utilização das funções da biblioteca pela GPU, além do fato de que as imagens foram redimensionadas para 256×256, a mesma dimensão das imagens disponibilizadas para treinamento da rede neural. Desta forma, este tempo pode ser consideravelmente aprimorado pela utilização de uma versão do *OpenCV* mais atualizada e que faça uso de uma placa gráfica dedicada, sobretudo para imagens maiores.

CAPÍTULO 5

Conclusão

A habilidade de se mapear o entorno de um determinado agente, sendo capaz de interagir com a vizinhança e extrair informações, é um dos requisitos para um agente ser autônomo. Para os AUVs, radares ou câmeras digitais constituem os sensores que reproduzem a capacidade de visão do equipamento, e a segmentação de imagem se mostra como uma das possíveis técnicas para a identificação de contornos de sua vizinhança. Após a fundamentação teórica descrita no capítulo 2, onde se abordaram alguns conceitos desta área científica e principais técnicas aplicadas, com maior aprofundamento para o campo de redes neurais, utilizou-se o *pipeline* disponível publicamente da rede neural ENet e adaptação do *dataset SUIM* de imagens subaquáticas para investigação de sua performance, explicitando os principais desafios encontrados durante a experimentação. Embora os resultados não tenham sido plenamente satisfatórios (principalmente para o treinamento com oito classes), o conceito de segmentação de imagens pôde ser apresentado e estudado ao longo dos capítulos 3 e 4, com resultados consistentes para treinamentos por duzentas épocas, *batch size* de dez imagens e otimizador *Adam* para predições com três classes.

Uma breve comparação com uma técnica de segmentação mais simples, de clusterização por *K-Means*, com base em três imagens do conjunto de testes, também foi realizada no capítulo 4, onde notou-se a superioridade do método por redes neurais, tendo como performance de *MIoU* igual a 71,57% para três classes de predição, frente a 39,00% de *MIoU* para o *K-Means*. Destaca-se, todavia, que aprimoramentos tanto na rede neural, com melhorias na arquitetura da rede buscando melhorar a otimização e adequação à aplicação aqui tratada, quanto no *dataset*, levantado de maneira suficientemente expressiva e representativa, aliados a estratégias de pré-processamento de imagens, com melhoria da definição de contornos e correção de turbidez e distorção de cores, se apresentam como pontos promissores para se obter um melhor desempenho

final da rede (sobretudo para treinamentos com mais classes de predição) sendo este último ponto de melhoria também aplicável à técnica de segmentação por K-Means. Pode-se, ainda, investigar outras topologias de redes neurais que atendam às demandas de segmentação de imagens subaquáticas e operação em tempo real, com boa eficiência de identificação das classes e tempo de processamento hábil.

Referências Bibliográficas

AGGARWAL, C. C. **Neural Networks and Deep Learning**. [S.l.]: Springer, 2018. 524 p.

AL-AMRI, S. S.; KALYANKAR, N. V.; KHAMITKAR, S. D. Image segmentation by using edge detection. **International Journal on Computer Science and Engineering**, IJCSE, v. 02, n. 03, p. 804–807, 2010.

BALA, A. An improved watershed image segmentation technique using matlab. **International Journal of Scientific & Engineering Research**, v. 3, 2012.

BODEN, M. **Mind as Machine: A History of Cognitive Science**. [S.l.]: Oxford University Press, 2006. 781 p.

BROWNLEE, J. **Gentle Introduction to the Adam Optimization Algorithm for Deep Learning**. 2017.

BROWNLEE, J. **How to Reduce Overfitting With Dropout Regularization in Keras**. 2018. Disponível em: ⟨https://machinelearningmastery.com/how-to-reduce-overfitting-with-dropout-regularization-in-keras/⟩.

BROWNLEE, J. **A Gentle Introduction to Pooling Layers for Convolutional Neural Networks**. 2019. Disponível em: ⟨https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/⟩.

CHAIRA, T. Intuitionistic fuzzy segmentation of medical images. **IEEE Transactions on Biomedical Engineering**, IEEE, v. 57, n. 6, p. 1430–1436, 2010.

CHOUHAN, S. S.; KAUL, A.; SINGH, U. P. Image segmentation using computational intelligence techniques: Review. **Archives of Computational Methods in Engineering**, Springer, v. 26, p. 533–596, 2019.

CROSSENTROPYLOSS. 2020. Disponível em: ⟨https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html⟩.

DHANACHANDRA, N.; MANGLEM, K.; CHANU, Y. J. Image segmentation using k -means clustering algorithm and subtractive clustering algorithm. **Procedia Computer Science**, v. 54, p. 764–771, 2015.

DREWS-JR, P. et al. Underwater depth estimation and image restoration based on single images. **IEEE Computer Graphics and Applications**, v. 36, p. 24–35, 03 2016.

GEIFMAN, A. **The Correct Way to Measure Inference Time of Neural Networks**. 2020. Disponível em: ⟨https://deci.ai/the-correct-way-to-measure-inference-time-of-deep-neural-networks⟩.

HAVAí, U. do. **Light in the Ocean**. 20–? Disponível em: ⟨https://manoa.hawaii.edu/exploringourfluidearth/physical/ocean-depths/light-ocean⟩.

HE, K. et al. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. **IEEE International Conference on Computer Vision (ICCV 2015)**, v. 1502, 02 2015.

HUBER, J. **Batch normalization in 3 levels of understanding**. 2020. Disponível em: ⟨https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338⟩.

IOFFE, S.; SZEGEDY, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. 02 2015.

KAUR, D.; KAUR, Y. Various image segmentation techniques: A review. **International Journal of Computer Science and Mobile Computing**, IJCSMC, v. 3, n. 5, p. 809–814, 2019.

LI, K. et al. Real-time segmentation of side scan sonar imagery for auvs. In: . [S.l.: s.n.], 2019. p. 1–5.

MARR, D. **Vision: A Computational Investigation Into the Human Representation and Processing of Visual Information**. [S.l.]: MIT Presser, 1982. 403 p.

NARKHEDE, H. P. Review on image segmentation techniques. **International Journal of Science and Modern Engineering**, IJSME, v. 01, n. 08, 2013.

PASZKE, A. et al. Enet: A deep neural network architecture for real-time semantic segmentation. **ArXiv**, abs/1606.02147, 06 2016.

RUSSEL, S. J.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. [S.l.]: Prentice Hall, 1995.

SANTIAGO, R. C.; MEZA, M. E. M.; TITOTTO, S. L. M. C. Desenvolvimento de design modular para ROV bioinspirado. In: BLUCHER PROCEEDINGS. **11º Congresso Brasileiro de Inovação e Gestão de Desenvolvimento do Produto**. [S.l.], 2017. v. 3, n. 12.

SCHETTINI, R.; CORCHS, S. E. Underwater image processing: State of the art of restoration and image enhancement methods. **EURASIP Journal on Advances in Signal Processing**, v. 2010, 2010.

SHORTEN, C.; KHOSHGOFTAAR, T. M. A survey on image data augmentation for deep learning. **Journal of Big Data**, v. 6, 2019.

SILVA, D. **PyTorch-ENet**. 2018. Disponível em: ⟨https://github.com/davidtvs/PyTorch-ENet⟩.

SKANSI, S. **Introduction to Deep Learning**. [S.l.]: Springer, 2018. 196 p.

SZELISKI, R. **Computer Vision**. Cham, Switzerland: Springer, 2011.

TERRY, P. J.; VU, D. Edge detection using neural networks. p. 391–395, 1993.

TOMPSON, J. et al. Efficient object localization using convolutional networks. In: . [S.l.: s.n.], 2015. p. 648–656.

VeRSTAPi2. **Research Group: Teleoperated / Autonomous Underwater Robotic Vehicles for Inspection and Intervention (VeRSTAPi2)**. 2017. Disponível em: ⟨http://pesquisa.ufabc.edu.br/verstapi2/⟩.

YU, F. et al. Segmentation of side scan sonar images on auv. In: IEEE. **IEEE Underwater Technology (UT)**. [S.l.], 2019. p. 1–4.

YUHENG, S.; HAO, Y. Image segmentation algorithms overview. 07 2017.

ZHANG, W. et al. A survey of restoration and enhancement for underwater images. **IEEE Access**, v. 7, p. 182259 – 182279, 12 2019.

ZHOU, X. Understanding the convolutional neural networks with gradient descent and backpropagation. **Journal of Physics: Conference Series**, v. 1004, p. 012028, 04 2018.