Matheus Kerber Venturelli

Classificação Semiautomática e Automática de Estruturas Porosas em Perfis de Imagem de Poços

PROJETO FINAL II

DEPARTAMENTO DE ENGENHARIA ELÉTRICA E DEPARTAMENTO DE INFORMÁTICA

Programa de graduação em Engenharia de Computação

Rio de Janeiro
Junho de 2021

Pontifícia Universidade Católica
do Rio de Janeiro

Matheus Kerber Venturelli

Classificação Semiautomática e Automática de Estruturas Porosas em Perfis de Imagem de Poços

Projeto Final II

Projeto Final II, apresentado ao programa de Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do titulo de Engenheiro de Computação.

Orientador : Prof. Marcelo Gattass
Co-orientador: Gabrielle Brandemburg dos Anjos

Rio de Janeiro
Junho de 2021

Resumo

Kerber Venturelli, Matheus; Gattass, Marcelo; Brandemburg dos Anjos, Gabrielle. **Classificação Semiautomática e Automática de Estruturas Porosas em Perfis de Imagem de Poços**. Rio de Janeiro, 2021. 57p. Projeto de Graduação – Departamento de Engenharia Elétrica e Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A classificação de corpos porosos em perfis de imagem de poços é um instrumento importante para diversas aplicações em engenharia. Entretanto, a classificação é feita de maneira majoritariamente manual, o que demanda um tempo considerável de trabalho. Esse projeto tem o objetivo de propor um método automático de fazê-la e também traçar um caminho para que isso seja feito de forma automática, utilizando dados sintéticos e aprendizado profundo.

Palavras-chave

Segmentação de imagens Classificação de objetos Grafos Aprendizado profundo

Sumário

1
Introdução

A indústria de óleo e gás é de extrema importância para a economia mundial. O petróleo é material base para diversas outras indústrias, sendo utilizado em veículos, na produção de plásticos e de tintas, na composição do asfalto e de diversos outros.

Segundo um estudo de mercado feito pela *IBISWorld*, até 2019, a receita do setor de perfuração de óleo de gás chegou a 3.3 trilhões de dólares [1], sendo por volta de 3,8% da economia global.

Dentro dessa indústria, uma das tarefas mais importantes é a simulação de poços de petróleo. Para que a simulação seja possível, é necessário analisar o solo a ser explorado e mapear suas propriedades.

Dentre essas propriedades, a permeabilidade é de especial importância pois ela mede a capacidade de um fluido escoar pelo seu interior, auxiliando a estimativa da quantidade de fluido que pode ser extraído dela. Quanto maiores e mais conectadas forem as cavidades onde os fluidos se acumulam, maior será a permeabilidade da rocha (Rosa et al. 2006).

As estruturas porosas podem ser classificadas em cavernas, vugues e fraturas (Jesus et al. 2016), como representado na Figura 1.1, que também apresenta de forma visual o perfil de imagem, que é um dos tipos de dado mais importantes para esse tipo de análise. Dentre essas estruturas, cada uma contribui de forma diferente para a permeabilidade do meio (Jesus et al. 2016).

[1] `https://www.investopedia.com/ask/answers/030915/what-percentage-global-economy-comprised-oil-gas-drilling-sector.asp`

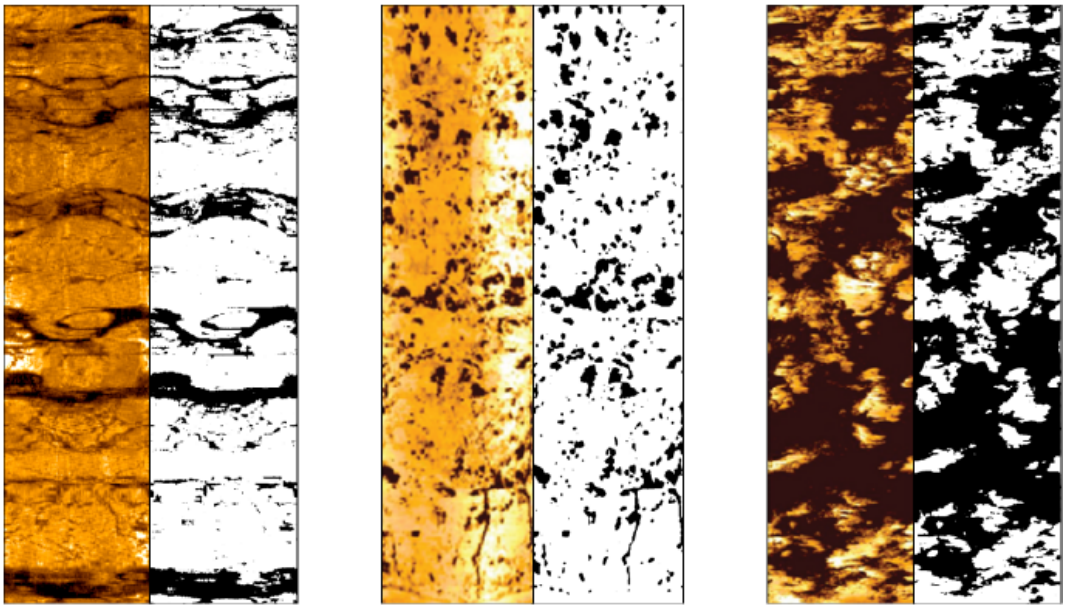

Figura 1.1: Estruturas porosas em perfil de imagem de poço, onde (a) representa fraturas, (b) são os vugues e (c) são as cavernas. Em todas (a), (b) e (c), do lado esquerdo está a imagem original e do lado direito a imagem binarizada. Fontes: (Jesus et al. 2016) e *Division of Marine and Large Programs* [3].

Atualmente, grande parte do trabalho de segmentação e classificação dessas estruturas acontece de forma majoritáriamente manual, que faz o processo se tornar demorado e dispendioso. Portanto, tendo esses aspectos em vista, mostra-se relevante a existência de ferramentas que auxiliem na determinação dessas propriedades de forma automática.

[3] `https://brg.ldeo.columbia.edu/logdb/scientific_ocean_drilling/result/`

2
Situação Atual

2.1
Soluções Disponíveis

Como mencionado no Capítulo 1, o trabalho de segmentação e classificação das estruturas é feito de forma majoritáriamente manual. Entretanto, existem abordagens como a apresentada em (Li et al. 2019), que realiza a classificação de forma automática utilizando operadores morfológicos.

Nela, os autores constroem matrizes de adjacência a partir de diferentes direções nas imagens (Figura 2.1). Depois, aplicam uma junção de operadores morfológicos, criando um algoritmo baseado na morfologia dos caminhos do grafo construído (Figura 2.2).

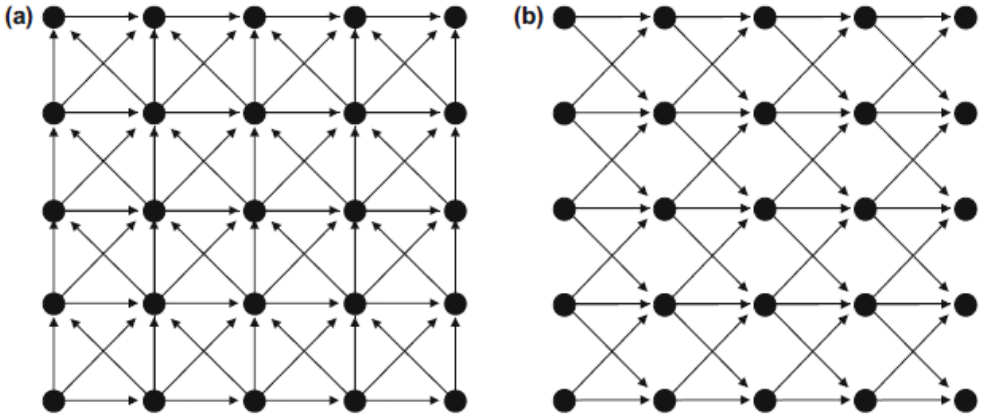

Figura 2.1: Matrizes de adjacência com orientações vertical (a) e horizontal (b). Imagem retirada de (Li et al. 2019).

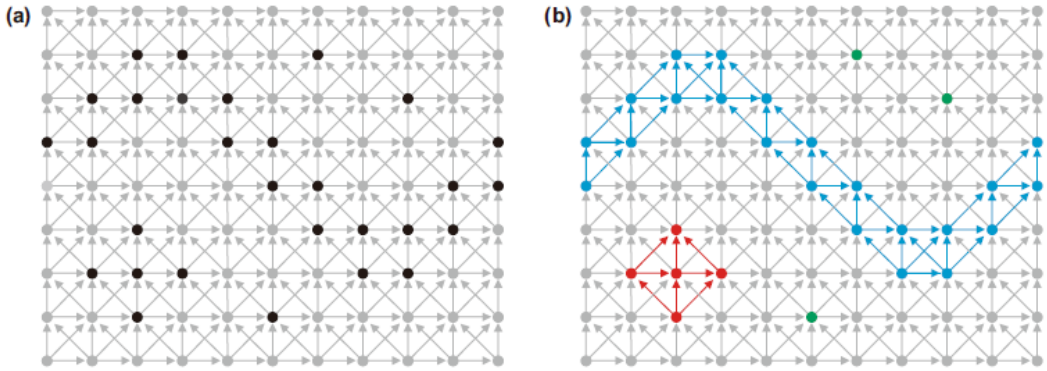

Figura 2.2: Operação de abertura de caminho utilizada na extração de fraturas e vugues. Imagem retirada de (Li et al. 2019).

A abordagem foi reproduzida, posteriormente, de forma semi-automática, por (Gabrielle 2020). Os dados e classificações prévias utilizados nesse projeto foram disponibilizados pela autora.

2.2 Contexto Tecnológico

Para a implementação do projeto, foi utilizada a linguagem *Python*, que conta com diversas bibliotecas que auxiliam no tratamento dos dados e oferecem implementações eficientes, como a *scikit-image* para processamento de imagens, a *NetworkX* para lidar com grafos e as bibliotecas *Keras* e *TensorFlow* para criar redes neurais.

Além disso, também existem bibliotecas como a *matplotlib* que podem ser utilizadas para visualização dos dados de forma versátil. Utilizando essas bibliotecas juntamente com o ambiente de programação *Jupyter Notebook*, é possível desenvolver testes rapidamente e manter uma documentação clara sobre os mesmos.

3
Objetivos

O principal objetivo deste trabalho é propor uma forma semiautomática de classificar estruturas porosas em perfis de imagem binarizados de poços (como os da Figura 3.1) e comparar seus resultados com os obtidos pela reprodução de (Li et al. 2019) feita por (Gabrielle 2020).

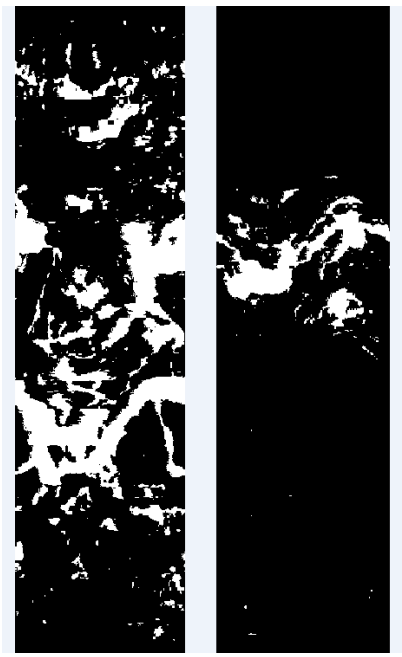

Figura 3.1: Exemplo de perfis de imagem binarizados de poços.

Além disso, um aspecto importante para a criação de algoritmos de classificação desse tipo de dado é a falta de dados anotados, que limita a possibilidade de utilizar algoritmos supervisionados para automatizar o processo.

Portanto, este projeto também visa traçar um caminho para fazer a classificação de forma automática utilizando uma rede neural treinada com dados sintéticos, posteriormente apresentando resultados preliminares da mesma, de forma a auxiliar trabalhos futuros na realização deste tipo de abordagem.

4
Atividades Realizadas

4.1
Estudos Preliminares e Conceituais

Previamente, o aluno já possuía experiência com a linguagem *Python* e com o ambiente de desenvolvimento *Jupyter Notebook*. Já havia desenvolvido diversos projetos, tanto em estudos pessoais quanto em matérias durante o curso de graduação utilizando essas tecnologias. Nestes projetos, foram utilizadas diversas bibliotecas aplicadas no trabalho atual, como *NumPy*, *scikit-image*, *NetworkX* e *Matplotlib*. Entretanto, foi necessário aprofundar os conhecimentos teóricos em relação à visão computacional. Parte da teoria utilizada foi fruto de estudos realizados ao longo do projeto, como a técnica de Esqueletonização, o Ruído de Perlin e a aplicabilidade da teoría de grafos na análise de imagens.

Também foi necessário aprofundar os estudos na área de *Deep Learning*, pesquisando sobre topologias de redes, sobre as diversas camadas utilizadas em visão computacional, como as camadas convolucionais, sobre os algoritmos otimizadores, como o Gradiente Descendente, sobre *loss functions*, como *Categorical Crossentropy* e sobre métodos de avaliação de resultados.

Além disso, também foi realizada a leitura de artigos que abordam o tema de análise de imagens de rochas para entender o contexto e as técnicas normalmente utilizadas nesse tipo de problema, como (Li 2017) e (Li et al. 2019).

4.2
Método

Durante o período do Projeto Final I, foram realizados estudos e implementações com dados de tomografia computadorizada de uma seção de testes cilíndrica de uma rocha. A proposta do projeto se mostrou inviável ao final do período e, portanto, foi readequada à proposta atual. Por esse motivo, o cronograma da Tabela 4.1 tem um contexto diferente do cronograma da Tabela 4.2.

Para a realização das atividades foi utilizado um modelo de trabalho onde o aluno planejava os estudos e implementações das próximas uma ou duas semanas, juntamente com o orientador, com base nos resultados obtidos até o momento e no escopo previsto para o projeto.

Atividades	Ago		Set		Out		Nov		Dez
Projeto									
Escrita da Proposta		■	■	■	■				
Entrega da Proposta					■				
Escrita do Relatório					■	■	■	■	■
Entrega do Relatório									■
Estudo									
Estudar Técnicas de Segmentação			■	■					
Estudar Técnicas de Registro de Imagens					■	■			
Estudar Técnicas de Modelagem de Objetos em Voxels							■	■	
Desenvolvimento									
Testes de Técnicas de Segmentação			■	■					
Testes de Técnicas de Registro de Imagens					■	■			
Testes de Técnicas de Modelagem de Objetos em Voxels							■	■	

Tabela 4.1: Cronograma de atividades de Projeto Final I

Atividades	Mar		Abr		Mai		Jun		Jul
Projeto									
Escrita do Relatório					■	■	■		
Entrega do Relatório								■	
Criação da Apresentação								■	
Defesa do Projeto									■
Estudo									
Estudos Teóricos	■		■						
Estudos de Implementação		■		■					
Desenvolvimento									
Implementação da Detecção de Estruturas Individuais	■								
Implementação da Esqueletonização de Estruturas	■								
Implementação da Extração de Parâmetros		■	■						
Implementação da Classificação de Estruturas			■						
Testes Comparativos da Abordagem Por Maior Caminho em Grafo			■						
Implementação da Geração de Dados Sintéticos				■	■				
Implementação da Rede					■				
Testes e Avaliação da Abordagem Por Aprendizado Profundo					■				

Tabela 4.2: Cronograma de atividades de Projeto Final II

5
Projeto e Especificação do Sistema

5.1
Abordagem por Maior Caminho em Grafo

Os perfis de imagem são obtidos por meio de sondas que percorrem os poços realizando medições, que podem ser acústicas [1] ou elétricas [2].

Após as medições, é feito um tratamento dos dados para gerar imagens. Essas imagens são projeções da parede do poço, que é cilindrica, em planos. Por esse motivo, quando o plano da fratura corta o poço com um ângulo maior que zero, a projeção dela na imagem é uma senoide com amplitude e fase que variam de acordo com a inclinação e direção com que o plano corta o cilindro. Podendo ser uma linha reta, caso o ângulo seja zero (Felix et al. 2013), como representado nas Figuras 5.1 e 5.2.

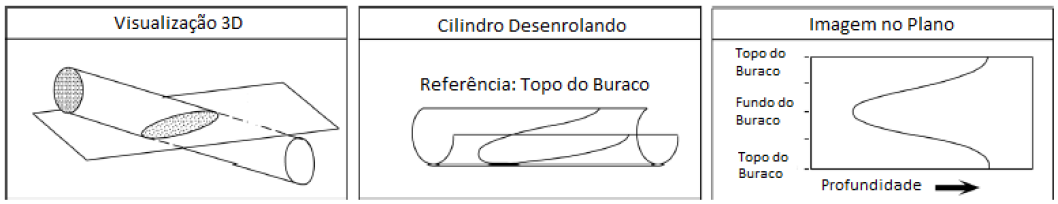

Figura 5.1: Representação da projeção em 2D de um poço cilíndrico horizontal, adaptado de (Asquith et al. 2004).

[1] O perfil de imagem acústica consiste na identificação de propriedades da formação, medindo-se a amplitude e o tempo de trânsito das reflexões de pulsos sonoros emitidos na parede do poço. (Felix et al. 2013)

[2] O perfil de imagem elétrica consiste na identificação de características da formação, medindo-se a resistividade de determinado intervalo rochoso à passagem de corrente elétrica. Com isso, rochas com propriedades diferentes terão valores distintos de resistividade. (Felix et al. 2013)

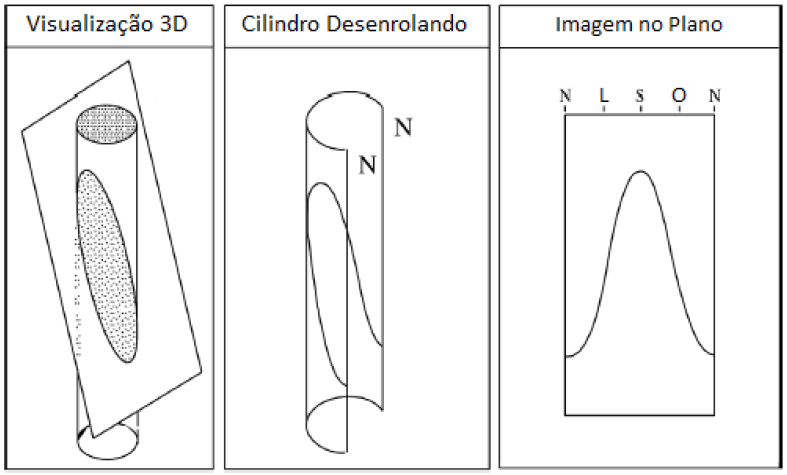

Figura 5.2: Representação da projeção em 2D de um poço cilíndrico vertical, adaptado de (Asquith et al. 2004).

Por esse motivo, as fraturas tendem a ter perfis alongados, com comprimento relativamente grande em relação a sua área, quando comparadas com outras estruturas. A abordagem descrita nessa seção toma proveito dessa característica, utilizanado as etapas descritas a seguir para classificar as estruturas.

5.1.1 Detecção de Estruturas Individuais

O primeiro passo para possibilitar a classificação é identificar, na imagem binarizada, quais pixels pertencem a cada estrutura. Essa informação será uma nova imagem com as mesmas dimensões do dado original, contendo um rótulo, que é um número natural, para cada estrutura. Nessa imagem, os pixels de cada estrutura terão o valor de seu respectivo rótulo.

Para encontrar quais pixels pertencem a qual estrutura, podemos interpretar a imagem como um grafo não direcionado, utilizando a seguinte definição para a criação de arestas:

$$\begin{aligned}&\{\,[\,(\,i = k\,)\ \wedge\ (j = l \pm 1)\,]\\ &\vee\ [\,(\,i = k \pm 1\,) \wedge (\,j = l\,)\,]\\ &\vee\ [\,(\,i = k \pm 1\,) \wedge (\,j = l \pm 1\,)\,]\,\} \wedge (\,x_{i,j} = x_{j,k} = \sigma\,) \leftrightarrow E(x_{i,j}, x_{k,l})\end{aligned} \tag{5-1}$$

Onde $x_{i,j}$ e $x_{k,l}$ são pixels pertencentes à imagem em análise, σ é um número natural que é o valor dos pixels que representam as estruturas na

imagem, e i, j, k e l são os índices dos pixels, tal que $i, j, k, l \in \mathbb{N}$. A operação $E(u, v)$ retorna verdadeiro caso exista aresta entre os vértices u e v e falso caso contrário.

Para obter as estruturas individuais, pode ser usada uma estrutura de união e busca, em que pixels pertencentes à mesma estrutura possuirão o mesmo nó representante na estrutura. Portanto, uma estrutura pode ser definida como:

$$C_k = \{\, x_{i,j} \mid busca(x_{i,j}) = k \,\} \tag{5-2}$$

Onde C_k é a k-ésima estrutura do conjunto de estruturas, com $k \in \mathbb{N}$. Além disso, $x_{i,j}$ é o pixel correspondente à linha i e coluna j da imagem, com $(i, j) \in \mathbb{N}^2$. A operação $busca(v)$ retorna o valor do nó representante do conjunto ao qual o vértice v pertence na estrutura de união e busca, sendo este um número natural.

Para obter os conjuntos, pode ser utilizado o seguinte procedimento:

Algorithm 1: GeraComponentesConexas

```
Atribui a imagem de entrada à variável Dados.
Atribui uma lista vazia à variável Ligados.
Atribui estrutura com as dimensões de Dados à variável
 Componentes, inicializada com valor do fundo da imagem.
Atribui à variável PróximoConjunto com valor 0.
for Linha em Dados do
    for Elemento em Linha do
        if Elemento ≠ Fundo then
            Atribui à lista Vizinhos, os elementos conectados a
             Elemento no grafo.
            if Vizinhos está vazia then
                Atribui ao índice correspondente à variável Elemento,
                 da lista Ligados, a componente que contém
                 PróximoConjunto.
                Atribui PróximoConjunto ao elemento de
                 Componentes correspondente à variável Elemento.
                Incrementa a variável PróximoConjunto.
            else
                Encontra o menor valor de componente entre os
                 vizinhos de Elemento.
                Aplica a operação de de união, gerando um novo
                 representante para os conjuntos dos vizinhos de
                 Elemento e atualiza a variável Ligados com o novo
                 representante.
            end
        end
    end
end
for Linha em Dados do
    for Elemento em Linha do
        Atribui à variável Componente o conjunto equivalente a
         Elemento em Componentes.
        if Componente ≠ Fundo then
            Aplica a operação de busca para encontrar o
             representante de Componente na estrutura de união e
             busca e atualiza a variável Componentes.
        end
    end
end
retorna Componentes
```

5.1.2
Esqueletonização das Estruturas

Para tomar vantagem da estrutura alongada das fraturas, um procedimento que pode ser adotado é o cálculo do comprimento de cada estrutura. Para isso, o procedimento de esqueletonização se mostra útil, pois é um processo que reduz a estrutura a caminhos com um pixel de espessura, ao mesmo tempo que preserva sua conectividade e comprimento (Hu et al. 2011).

Uma forma de definir a operação de esqueletonização é a partir do operador morfológico de afinameto que, por sua vez, pode ser definido a partir da transformada acerto-ou-erro.

A transformada acerto-ou-erro se baseia na utilização de um conjunto, chamado de Elemento Estruturante (ES), que pode ter formas diferentes de acordo com a informação a ser extraída e do tipo de imagem estudada (Facon 2002). Esse conjunto é composto por três tipos de pixels: $foreground$, *background* e *dontcare*. A Transformada acerto-ou-erro alinha cada pixel da imagem a ser analisada com o centro do ES e define como $foreground$ os pixels centrais em que a região de sobreposição entre o ES e a imagem têm correspondencia exata entre $foreground$ e *background*.

De maneira mais formal, sejam C e D dois ES, onde $C \cap D = \varnothing$. Podemos definir o operador B, como o elemento composto pelo par (C, D) e, então temos que a transformada acerto-ou-erro ($\alpha_B(X)$) do conjunto X pelo ES B é:

$$\alpha_B(X) = X \odot B = (X \ominus C) \cap (X' \ominus D) \tag{5-3}$$

Onde X' é o complemento do conjunto X.

A partir da definição da transformada, podemos definir, de maneira genérica, a operação de afinameto ($\epsilon_B(X)$). Portanto, seja $\beta = \{B_1, ..., B_n\}$ um conjunto de ES. Para i entre 1 e n, temos que, para um conjunto X binário:

$$\epsilon_B(X) = X \otimes B_i = X \setminus (X \odot B_i) \tag{5-4}$$

Onde $A \setminus B$ é a operação de complemento relativo de A em B.

A partir de 5-4, podemos obter a operação de esqueletonização de um conjunto X ($\gamma_B(X)$) aplicando a operação de afinameto repetidas vezes, de

maneira ciclica, utilizando o conjunto β, até que haja a convergência:

$$\gamma_B(X) = X \otimes B_1 \otimes B_2 \ ... \otimes B_n \otimes B_1 \ ... \tag{5-5}$$

Com isso, é possível obter o esqueleto de imagens, como exemplificado nas Figuras [3] 5.3 e 5.4.

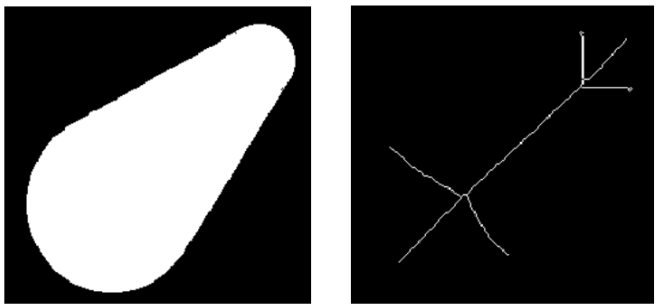

Figura 5.3: Imagem binarizada de um cilindro (a) e o resultado da aplicação do operador de esqueletonização (b).

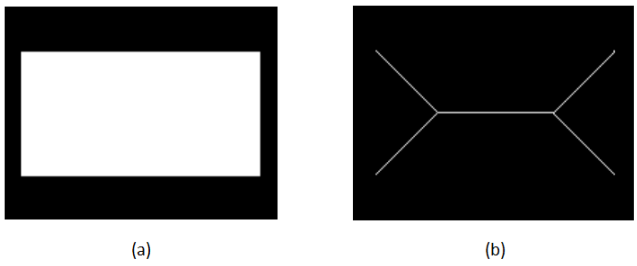

Figura 5.4: Imagem binarizada de um retângulo (a) e o resultado da aplicação do operador de esqueletonização (b).

5.1.3 Extração de Parâmetros

A Partir das imagens esqueletonizadas e do mapa de estruturas individuais obtido com o Algoritmo 1, é possível extrair o comprimento e a área das estruturas presentes nos dados.

Para obter a área de cada estrutura, basta percorrer o mapa de estruturas contando o número de pixels para cada label. Já para o comprimento, uma

[3] Imagens adaptadas de https://homepages.inf.ed.ac.uk/rbf/HIPR2/skeleton.htm.

forma de obtê-lo é calculando o maior caminho no esqueleto da estrutura, já que este preserva sua conectividade e comprimento (Hu et al. 2011).

Por sua vez, uma forma de calcular o maior caminho no esqueleto é primeiramente interpretá-lo como um grafo não direcionado e então calcular o maior caminho no grafo. Para construir o grafo, pode ser utilizada a definição de criação de arestas da Expressão 5-1. Portanto, podemos definir o algoritmo para construção do grafo:

Algorithm 2: GeraGrafo

```
Cria grafo vazio G.
for Linha em Dados do
    for Elemento em Linha do
        if Elemento ≠ Fundo then
            Obtem lista de vizinhos de Elemento de acordo com a
             definição da Expressão 5-1 e atribui à variável Vizinhos.
            for Vizinho em Vizinhos do
                Atribui aresta, em G, entre Elemento e Vizinho.
            end
        end
    end
end
retorna G
```

Com o grafo obtido com o Algoritmo 2, ainda não é possível obter o maior caminho em um tempo razoável, pois o grafo pode ter ciclos e obter o caminho mais longo em um grafo não direcionado ciclico e sem pesos nas arestas é NP-Difícil (Karger et al. 1997). Para tornar esse procedimento possível em tempo polinomial, podemos obter a arvore geradora do grafo e então calcular o maior caminho.

Para obter a arvore geradora, como o grafo não tem pesos nas arestas, podemos utilizar o algoritmo de busca em profundidade (BFS), pois ela checa se um nó já foi acessado antes de visitá-lo, o que remove ciclos. Portanto, a arvore de acessos da BFS pode ser utilizada como árvore geradora.

Só resta obter o maior caminho na arvore, o que é a mesma coisa que obter seu diâmetro. Para isso, podemos definir o seguinte algoritmo:

Algorithm 3: CaminhoMaisLongo

```
Escolhe um nó arbitrário s.
Aplica BFS em s e pega o nó mais distante na árvore de chamadas
 da BFS, u.
Aplica novamente a BFS em u, pegando o nó mais distante na
 árvore de chamadas da BFS, v.
retorna Caminho (u, v)
```

Uma prova formal da corretude do Algoritmo 3 é dada em (Bulterman et al. 2002).

Portanto, o Algoritmo 3 funciona para toda árvore não direcionada e pode ser utilizado para obter o caminho mais longo, que será utilizado para calcular métricas na seção 5.1.4.

5.1.4 Classificação de estruturas

Com o cálculo da área e do maior caminho, é possível criar métricas para classificar as estruturas em vugues ou fraturas. Para isso, as seguintes métricas foram testadas:

$$\alpha_i = \frac{C_i}{A_i} \tag{5-6}$$

$$\lambda_i = \frac{(C_i)^2}{A_i} \tag{5-7}$$

Onde C_i é o comprimento do maior caminho da estrutura i e A_i é a área, também da estrutura i.

Aplicando as métricas nos dados, são obtidos os seguintes histogramas:

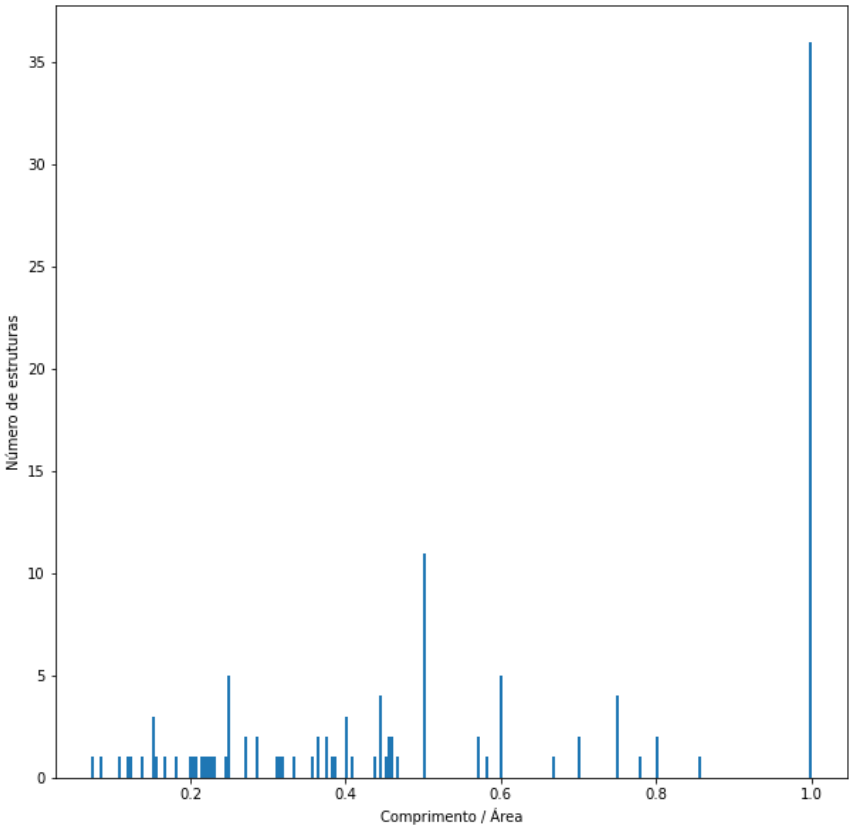

Figura 5.5: Distribuição da Métrica 5-6 aplicada nos dados.

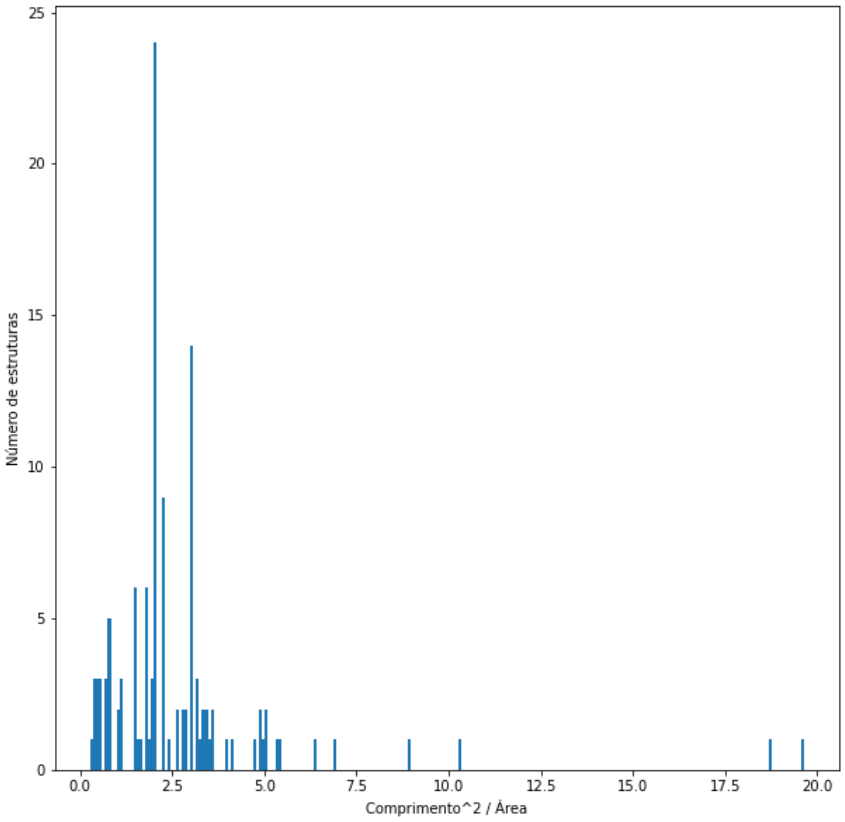

Figura 5.6: Distribuição da Métrica 5-7 aplicada nos dados.

Analisando os histogramas, é possível perceber que a Métrica 5-6 não produz dados separáveis, pois os mesmos aparecem espalhados e com um acúmulo maior em em 1,0. Por outro lado, a Métrica 5-7 produz uma distrubuição mais próxima de 0 e alguns *outliers*. Esses *outliers* são as fraturas, enquanto os dados mais próximos de 0 são vugues. Dessa forma, é possível classificar as estruturas definindo um valor limite para a Métrica 5-7, que separa vugues de fraturas. Mais detalhes dos resultados serão discutidos na no capítulo 6.

5.2 Abordagem por Aprendizado Profundo

Diferentemente da abordagem anterior, o objetivo da seção atual é traçar um caminho para desenvolver uma forma de resolver o problema de maneira completamente automática utilizando aprendizado supervisionado.

Um dos maiores desafios para criar soluções automáticas é a falta de dados anotados. Por esse motivo, este trabalho adota o uso de dados sintéticos que são utilizados para treinar uma rede neural.

5.2.1
Geração de Dados Sintéticos

O primeiro passo para a geração de dados sintéticos, é definir uma forma de gerar fraturas aleatoriamente. Para isso, é necessário definir uma forma de desenhar senoides com parametros que possam ser aleatorizados.

Para isso, podemos utilizar planos cortando cilindros, como representado na Figura 5.2. Além disso, é interessante que o intervalo de valores possíveis dos parâmetros seja finito, permitindo ao algoritmo cobrir todas as possibilidades. Portanto, podemos utilizar coordenadas esféricas para representar a normal ao plano e coordenadas cilíndricas para representar o cilindro, pois ambas podem ser parametrizadas por ângulos que pertencem a intervalos finitos (Figuras 5.7 e 5.8).

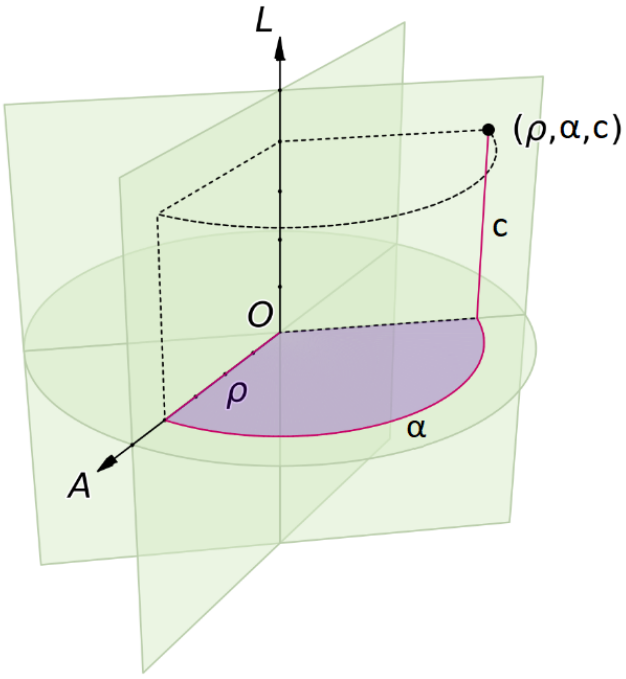

Figura 5.7: Parametrização por coordenadas cilíndricas. Imagem adaptada de *Wikipedia* [5].

[5] https://en.wikipedia.org/wiki/Cylindrical_coordinate_system

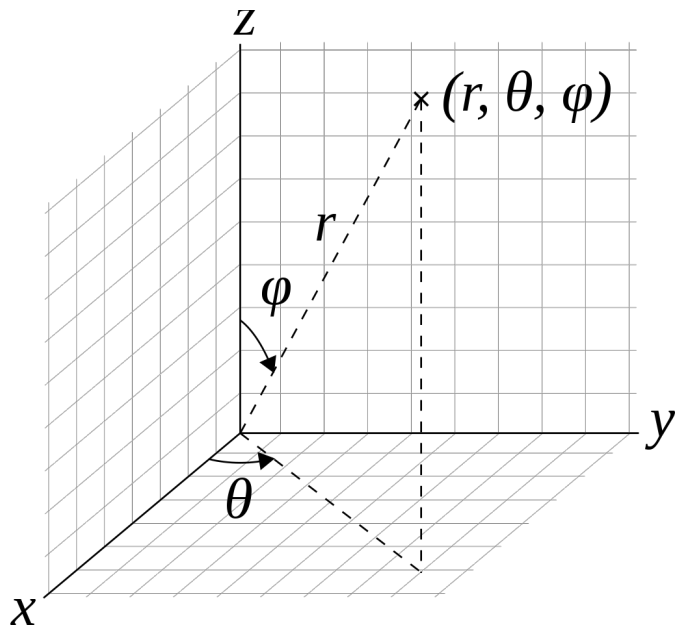

Figura 5.8: Parametrização por coordenadas esféricas. Imagem retirada de *Wikipedia* [7].

Portanto, podemos parametrizar as coordenadas da normal ao plano, com raio r unitário:

$$\begin{aligned} x &= cos(\theta) \cdot sen(\phi) \\ y &= sen(\theta) \cdot sen(\phi) \\ z &= cos(\phi) \end{aligned} \tag{5-8}$$

Onde θ e ϕ são os ângulos da parametrização esférica, como representado na Figura 5.8, e x, y e z são as coordenadas cartesianas correspondentes.

Agora, para o corte no cilindro, também com raio ρ unitário, temos:

$$\begin{aligned} \alpha &= \frac{2 \cdot \pi \cdot j}{w - 1} \\ a &= cos(\alpha) \\ b &= sen(\alpha) \\ c &= \frac{x \cdot a + y \cdot b}{z} \end{aligned} \tag{5-9}$$

[7] `https://en.wikipedia.org/wiki/Spherical_coordinate_system`

Onde j é o índice da coluna da imagem, w é a largura da imagem, α é o ângulo da parametrização cilíndrica, como representado na Figura 5.7, e a, b e c são as coordenadas cartesianas correspondentes.

Para obter a coordenada da altura do pixel na imagem, temos:

$$i = \left\lfloor \frac{(c+1) \cdot (\frac{h}{\tau} - 1)}{2} \right\rfloor + s \tag{5-10}$$

Onde i é o índice da linha da imagem, h é a altura da imagem, τ é um fator de controle do comprimento de pico a pico (C_{pp}) da senoide, representado na Figura 5.9, e s é altura da senoide na imagem.

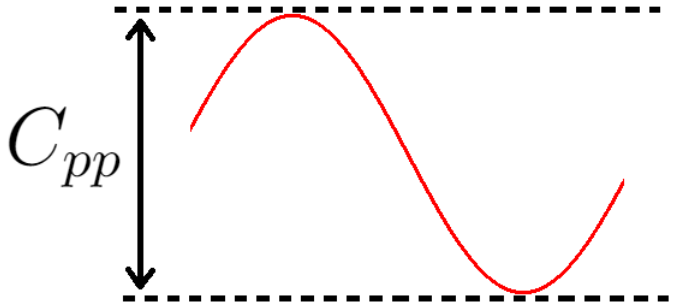

Figura 5.9: Representação visual de C_{pp}.

Portanto, utilizando a Equação 5-10, podemos percorrer a imagem em largura e desenhar a senoide pintando os pixels nas coordenadas (i, j) da mesma.

Também podemos criar uma espessura k para a senoide, como representado na Figura 5.10.

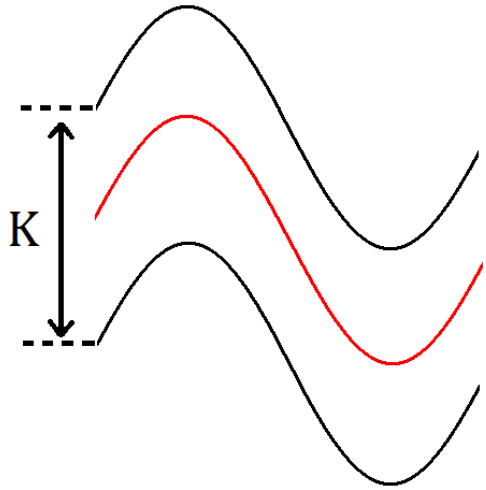

Figura 5.10: Espessura de tamanho k.

Para criar essa espessura, podemos pintar pixels em um intervalo de linhas δ, variando i em cada coordenada (i, j) da curva, definido por:

$$\delta = \left[i - \frac{k}{2},\ i + \frac{k}{2} \right] \tag{5-11}$$

Com isso, é possível obter imagens sintéticas como a Figura 5.11.

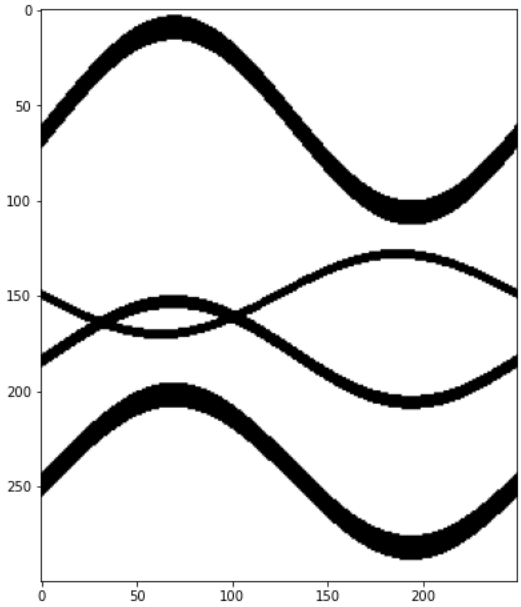

Figura 5.11: Passo inicial da geração de fraturas sintéticas.

Com as Equação 5-10, já é possível desenhar senoides aleatórias, entretanto, esse modelo ainda está distante dos dados reais, pois tem formato muito uniforme. Uma forma de melhorar esse modelo é utilizando do Ruído de Perlin, que pode gerar imperfeições aleatórias no modelo das fraturas.

Para iniciar esse processo, podemos primeiramente criar uma imagem com as mesmas dimensões da imagem original com o ruído puro, como na Figura 5.12.

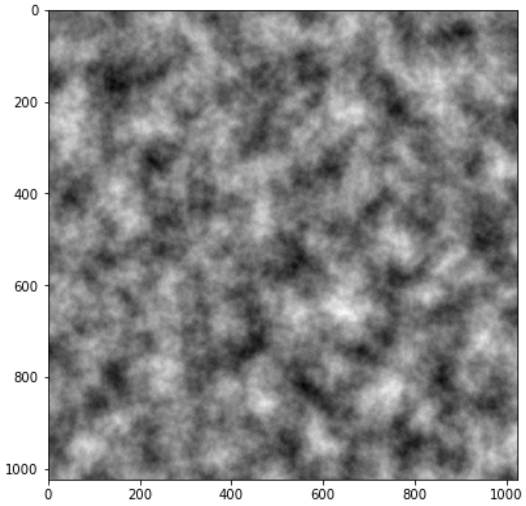

Figura 5.12: Imagem com Ruído de Perlin puro.

A partir da imagem com o ruído puro, podemos aleatorizar a fratura alterando o formato das suas bordas e do seu interior. Para alterar as bordas, podemos alterar o intervalo em que os pixels são pintados ao redor da senoide, definido na Equação 5-11, para incorporar uma componente do ruído. Portanto, sendo o ruído bidimensional ψ e ϵ um fator de intensificação, temos:

$$\Psi = \psi \cdot \epsilon \tag{5-12}$$
$$\delta_{ruído} = \left[i - \frac{k}{2} - ||\lfloor \Psi_{i,j} \rfloor||,\ i + \frac{k}{2} + ||\lfloor \Psi_{i,j} \rfloor|| \right]$$

Além disso, para adicionar mais imperfeições às fraturas, podemos multiplicar a imagem gerada com a Equação 5-12 por outra imagem de Ruído de Perlin, também com as mesmas dimensões da imagem original e, em seguida, estabelecer uma faixa de valores dentro da qual os pixels terão valor de estrutura, enquanto os demais terão valor de fundo. O resultado desse procedimento são imagens como a Figura 5.13.

Figura 5.13: Fraturas sintéticas com imperfeições.

Com o mesmo método do intervalo utilizado para gerar a Figura 5.13, podemos gerar uma nova imagem de ruído com as dimensões da imagem original, aplicar o método do intervalo, obtendo uma imagem com vugues simulados.

Para criar uma imagem com vugues e fraturas, basta que, seja a imagem binária das fraturs f e a imagem binária dos vugues v, crie-se uma nova imagem:

$$fv = f \vee v \tag{5-13}$$

Entretanto, para impedir que as fraturas fiquem encobertas por vugues, é necessário criar uma máscara para remover os vugues que estiverem por cima das fraturas. Para isso, podemos percorrer a senoide nas coordenadas (i, j) da imagem utilizando o intervalo:

$$\delta_{máscara} = \left[i - n \cdot \frac{k}{2} - |\lfloor \Psi_{i,j} \rfloor|,\ i + n \cdot \frac{k}{2} + |\lfloor \Psi_{i,j} \rfloor| \right] \tag{5-14}$$

Onde n é um número natural maior que 1, utilizado para criar um

intervalo ao redor da fratura com o mesmo formato, mas com amplitude maior do que a dela, que foi definida na equação 5-12.

Dessa forma, sendo a máscara binária m, criada utilizando a Equação 5-14,podemos utilizar a seguinte expressão para juntar vugues e fraturas:

$$fv = f \vee [\, v \wedge (\neg\, m\,)\,] \tag{5-15}$$

A Figura 5.14 ilustra o resultado desse processo.

Figura 5.14: Dados sintéticos com vugues e fraturas.

Além disso, como sabemos quais são os vugues e as fraturas, podemos gerar uma imagem que preserva essas informações, para poder utilizar como anotações para o treinamento da rede neural que será descrita na seção 5.2.2.

A Figura 5.15 mostra o resultado final, com os dados sintéticos ao lado das anotações.

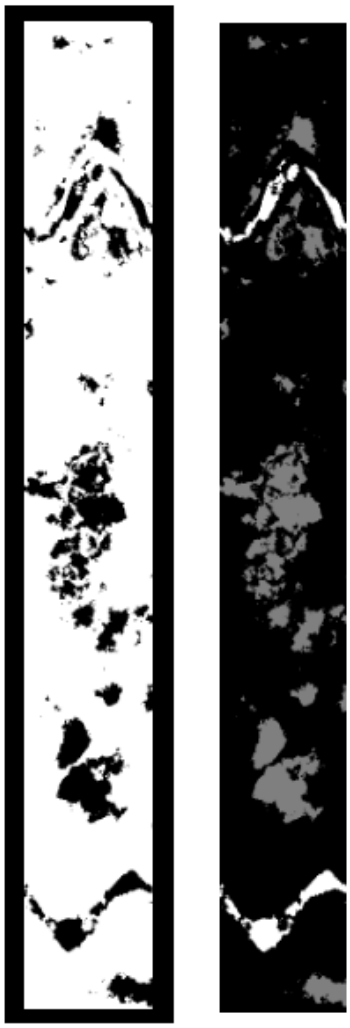

Figura 5.15: Dados sintéticos brutos (a) e dados sintéticos com anotações (b).

Mais detalhes sobre a implementação e aleatorização dos parâmetros serão abordados no capítulo 6.

5.2.2 Estrutura da Rede

Como queremos identificar e classificar estruturas individuais que estão sobre um fundo uniforme e têm todas o mesmo valor de pixel, também uniforme, uma possível abordagem é a utilização de uma rede de detecção e classificação de objetos.

Para isso, foi utilizado o framework *Mask R-CNN (Region Convolutional Neural Network)* (He et al. 2018), que consiste de dois estágios: o primeiro gera propostas, que são regiões com alta probabilidade de conterem objetos, enquanto o segundo, analisa essas propostas, gerando caixas delimitadoras (BBox) ao redor dos objetos e máscaras com o formato dos mesmos.

A abordagem utilizada por esse framework é a de segmentação de instâncias, em que a saída da rede classifica cada pixel da imagem individualmente, gerando uma mascara por objeto detectado, além das BBox, que delimitam a região onde cada objeto está na imagem, como exemplificado na Figura 5.16.

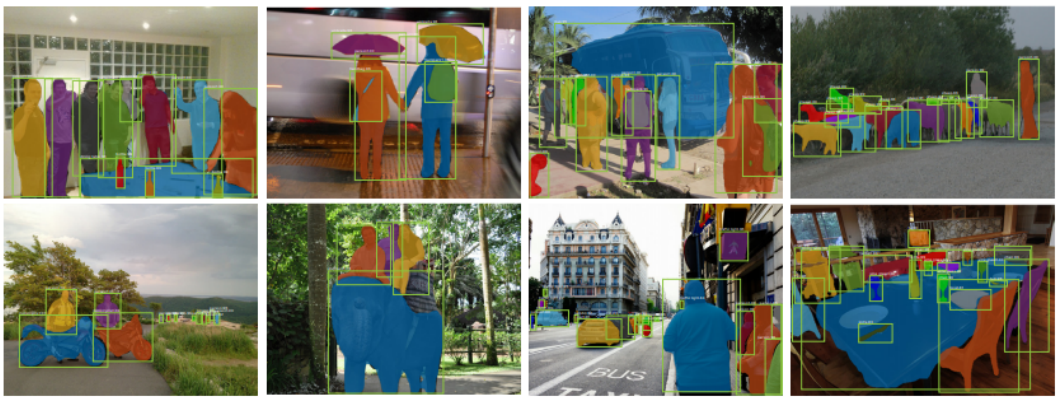

Figura 5.16: Exemplo de segmentação de instâncias usando o framework Mask R-CNN. Imagem retirada de (He et al. 2018).

De maneira mais detalhada, o framework consiste de quatro partes principais: a Rede Base, a *Feature Pyramid Network* (FPN), *Region Proposal Network* (RPN), Classificador de Regiões de Interesse (ROIs), Redimensionamento de ROIs e a Geração de Máscaras de Segmentação.

A **Rede Base** é uma rede neural convolucional convencional, ResNet50 ou ResNet101 (He et al. 2016). Ela é responsável pela extração de $features$ da imagem. As camadas mais externas detectam $features$ mais básicas, como bordas e cantos, enquanto as mais profundas têm a capacidade de detectar objetos mais complexos que dependem dos dados de treino, como carros, pessoas, ou fraturas em rochas.

A **FPN** (He et al. 2017) adiciona uma segunda pirâmide de segmentação à estrutura da Rede Base para melhorar a capacidade de representação de objetos em múltiplas escalas. Ela toma como entrada, os mapas de $features$ da Rede Base e passa a informação da camada superior para as camadas inferiores, juntando com ainformação dos mapas das camadas intermediárias

da Rede Base, como representado na Figura 5.17. Dessa forma, ela possibilita que todas as camadas tenham acesso a $features$ de baixo e alto nível.

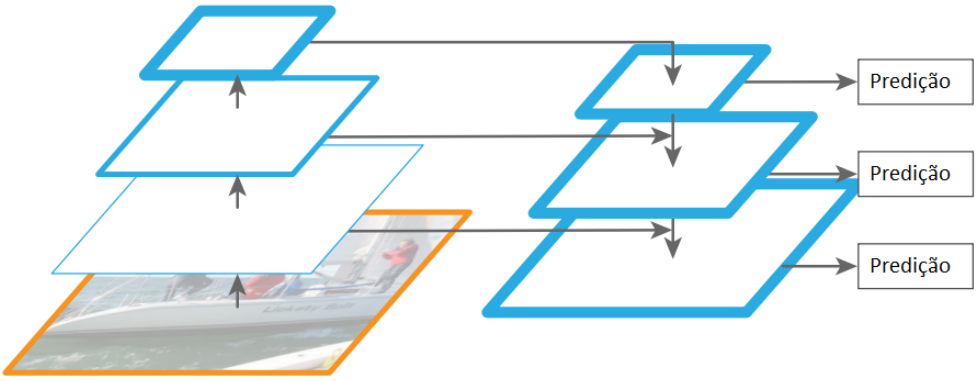

Figura 5.17: Estrutura da FPN. Imagem adaptada de (He et al. 2017).

A **RPN** (Ren et al. 2016) é uma rede que escaneia os mapas de $features$ gerados pela FPN com uma janela deslizante, com o objetivo de encontrar áreas que contenham objetos. As regiões que a RPN escaneia são chamadas de âncoras, que têm dimensões diferentes e se sobrepõem de forma a cobrir a maior parte possível do mapa de $features$.

Ela gera duas saídas para cada âncora: a classe da âncora e um refinamento de BBox. A classe da âncora pode ter valor de plano de fundo ou objeto. Caso tenha valor de objeto, quer dizer que tem uma probabilidade alta de existir um objeto dentro da âncora. O refinamento de BBox é necessário porque a âncora pode não estar bem centrada no objeto. Então, a RPN estima uma porcentagem de mudança para as coordenadas de posição e para as dimensões da âncora, de forma a se adequar melhor ao objeto. Este processo foi representado na Figura 5.18.

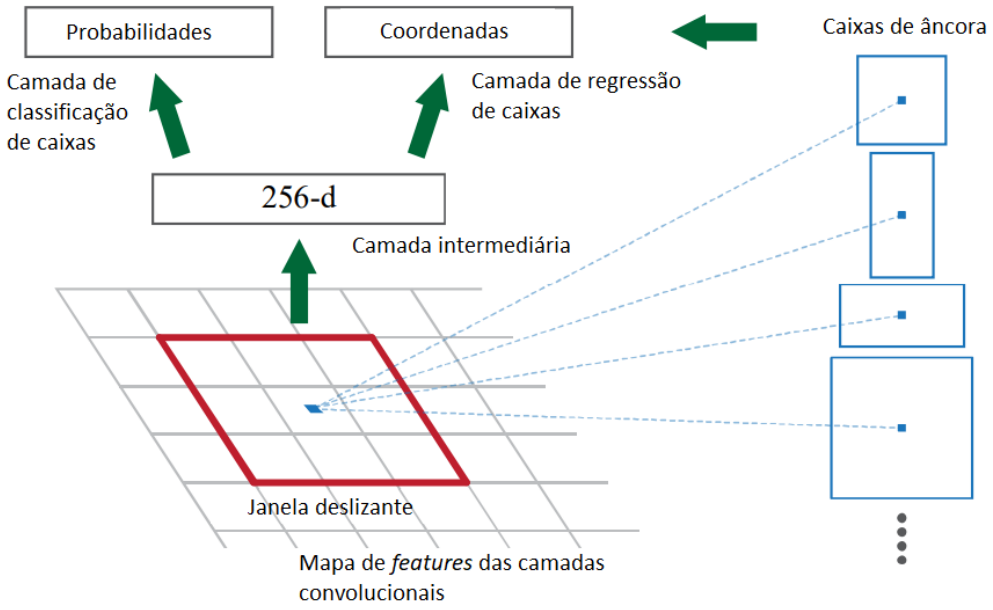

Figura 5.18: Janela deslizante, classificação e refinamento de BBox da RPN. Imagem adaptada de (Ren et al. 2016).

O **Classificador de Regiões de Interesse (ROIs)** (Ren et al. 2016), é uma rede que recebe como entrada a saída da RPN e, assim como a RPN, gera duas saídas: a classe da ROI e um refinamento de BBox. Entretanto, nesse caso, a classificação é mais refinada, podendo classificar o objeto em classes mais complexas, como carro, pessoa, fratura ou vugue. Ela também gera uma classe de plano de fundo, que faz a ROI ser descartada. O Refinamento é parecido com o da RPN e seu objetivo é melhorar a BBox em torno do objeto. Esse processo foi representado na Figura 5.19.

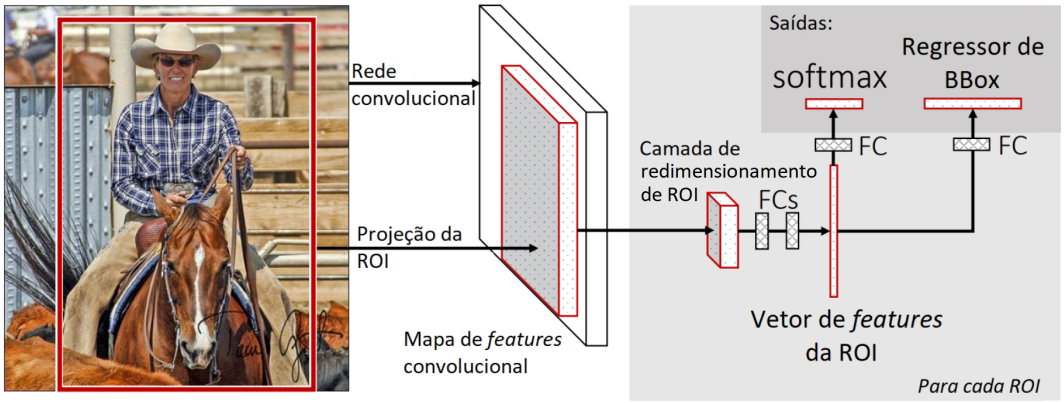

Figura 5.19: Estrutura do Classificador de ROIs. Imagem adaptada de (Girshick 2015).

O **Redimensionamento de ROIs** é um processo realizado antes da classificação, como representado na Figura 5.19, que é necessário porque as camadas de classificação não são capazes de lidar com a variedade de tamanhos e dimensões das ROIs. Por isso, essa camada redimensiona as ROIs para que tenham sempre as mesmas dimensões antes de entrarem na etapa de classificação. Para isso, é utilizado o método *ROIAlign*, que retira amostras em diferentes regiões do mapa de features e aplica uma interpolação bilinear (He et al. 2018).

Por fim, a **Geração de Máscaras de Segmentação** é feita por uma rede convolucional que recebe como entrada as ROIs selecionadas pelo Classificador de ROIs e gera máscaras, inicialmente de resolução baixa (28 x 28 pixels). Essas máscaras de baixa resolução são compostas por *floats*, portanto, elas guardam mais informação doque máscaras binárias. Posteriormente, essas máscaras são redimensionadas para as dimensões das BBox das ROIs. Esse processo gera uma máscara binária por objeto, além da classificação de cada um deles.

6
Implementação e Avaliação

Este capítulo visa apresentar a implementação, os testes e os resultados dos algoritmos descritos no capítulo 4. Todas as implementações foram feitas em python, utlizando bibliotecas e *frameworks* adequados a cada abordagem.

6.1
Abordagem por Maior Caminho em Grafo

6.1.1
Implementação

Para implementar os algoritmos descritos na seção 5.1, foram criadas três classes: *Graph*, *DAO* e *Segmenter*.

A classe ***Graph*** foi construída utilizando a biblioteca *NetworkX*, utilizando a classe *networkx.Graph()* como base para armazenar o grafo, em uma variáel interna chamada *self._graph*. Ela também é responsável pela implementação do algoritmo que encontra a arvore geradora mínima do grafo, implementado como um *wrapper* da função *networkx.minimum_spanning_tree(graph)*, além da implementação da função que calcula o diâmetro da árvore (Algoritmo 3), como representado no Listing 6.1.

```
def longest_path(self):
    # get BFS tree of self._graph,
    # starting on any node N, as tree
    tree = nx.bfs_tree(self._graph,
                       source = list(
                            self._graph.nodes
                                    )[0])

    # call dag_longest_path on tree starting on N
    path = nx.dag_longest_path(tree)

    # get furthest node n_far
    n_far = path[-1]

    # get BFS tree of self._graph,
    # starting on n_far, as tree
    tree = nx.bfs_tree(self._graph,
```

```
18                          source = n_far)
19
20      # call and return dag_longest_path on
21      # tree starting on n_far
22      return nx.dag_longest_path(tree)
```

Listing 6.1: Implementação do Algoritmo 3.

O ***DAO*** *(Data Access Object)* é um *Singleton* responsável por ler e armazenar os dados em variáveis. Ele também guarda as variáveis de estágios intermediários do código, para evitar que hajam cópias desnecessárias, melhorando a performance em velocidade e uso de memória.

O DAO possui cinco variáveis que guardam estágios de processamento intermediários do algoritmo principal, que está na classe *Segmenter*. Essas variáveis são: *data*, que armazena os dados brutos, *hole_map*, que armazena o mapa de estruturas individuais calculado com o Algoritmo 1, *hole_parameters*, que armazena, para cada estrutura, a área e o caminho mais longo calculado com o Algoritmo 3, *skeletonized_data*, que armazena os esqueletos das estruturas e *classified_dataset* que armazena a classificação das estruturas. Todas as variáveis são armazenadas em *numpy.array*'s da biblioteca *numpy*, que implementa diversas operações com vetores e matrizes de maneira eficiente.

A classe ***Segmenter*** possui o código do algoritmo principal, que faz a segmentação semântica dos dados. Ela possui quatro métodos principais:

```
1  def segment_connected_components(self)
2
3  def skeletonize_dataset(self)
4
5  def get_hole_parameters(self)
6
7  def classify_structures(self,
8                          min_structure_area,
9                          cut_value)
```

Listing 6.2: Assinaturas dos métodos da classe *Segmenter*.

O método ***segment_connected_components()*** implementa o Algoritmo 1 utilizando a função *label(image, connectivity=2, return_num=True)* da biblioteca *skimage.measure*, que é uma implementação eficiente do mesmo.

O método ***skeletonize_dataset()*** implementa a esqueletonização dos dados utilizando a função *skeletonize* da biblioteca *skimage.morphology*, que oferece uma implementação eficiente da mesma.

O método ***get_hole_parameters()*** implementa o cálculo da área de cada estrutura, a construção dos grafos dos esqueletos descrita no Algoritmo 2, calcula a árvore de custo mínimo, utilizando o método *wrapper* da classe *Graph*

e utiliza o método *longest_path()* da mesma classe para calcular o comprimento do maior caminho em cada árvore. Além disso, ele também calcula a Métrica 5-7.

Por fim, o método ***classify_structures(min_structure_area, cut_value)*** utiliza o valor de corte *cut_value* e a Métrica 5-7, calculada pelo método *get_hole_parameters()* para classificar cada estrutura, como descrito na Seção 5.1.4. E, por fim, filtra as estruturas, classificando apenas aquelas que têm área significativa, maior que *min_structure_area*.

6.1.2 Testes e Comparações

Nessa seção, será feita a comparação visual dos resultados do algoritmo descrito na seção 5.1 e dos resultados reprodizidos por (Gabrielle 2020), que será analisada na seção 6.1.3. Em todas as imagens, (a.x) se refere aos resultados do algoritmo descrito neste trabalho, enquanto (b.x) se refere aos resultados do trabalho em comparação. Também em todas as imagens, (a.1) e (b.1) são a imagem original, sem classificação, enquanto (a.2) e (b.2) são as estruturas classificadas como fraturas e (a.3) e (b.3) são as estruturas classificadas como vugues. Para todos os testes, foi utilizado 10 como valor de corte para a métrica 5-7 e sem limite mínimo para a área das estruturas.

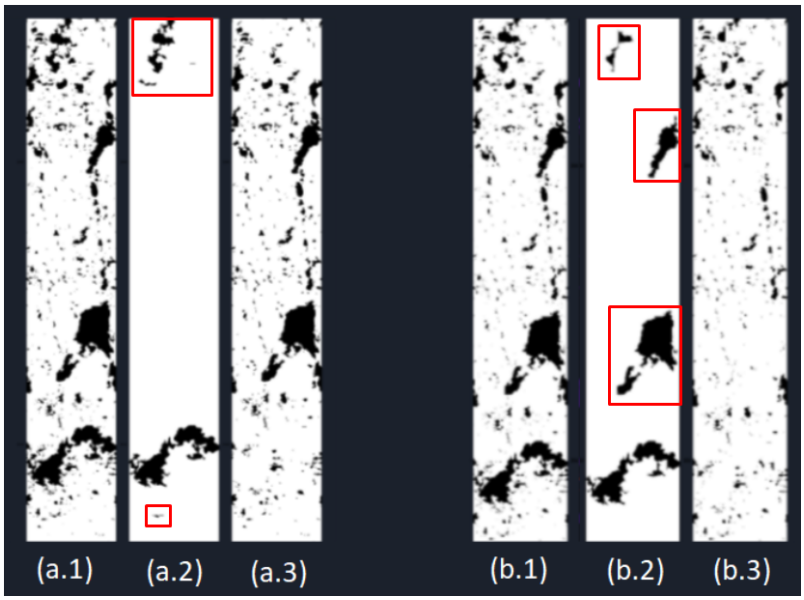

Figura 6.1: Comparação de resultados em imagem com vugues e uma fratura. Estruturas classificadas errado circuladas em vermelho.

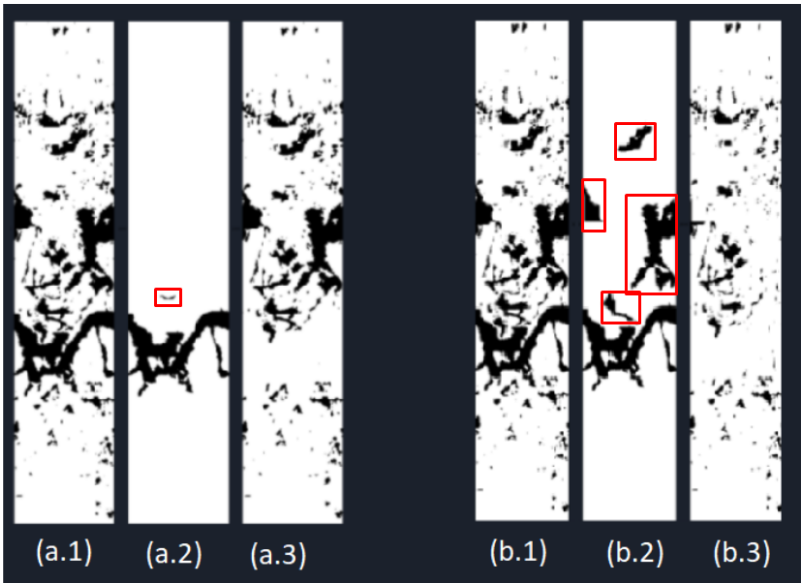

Figura 6.2: Comparação de resultados em imagem com vugues e uma fratura. Estruturas classificadas errado circuladas em vermelho.

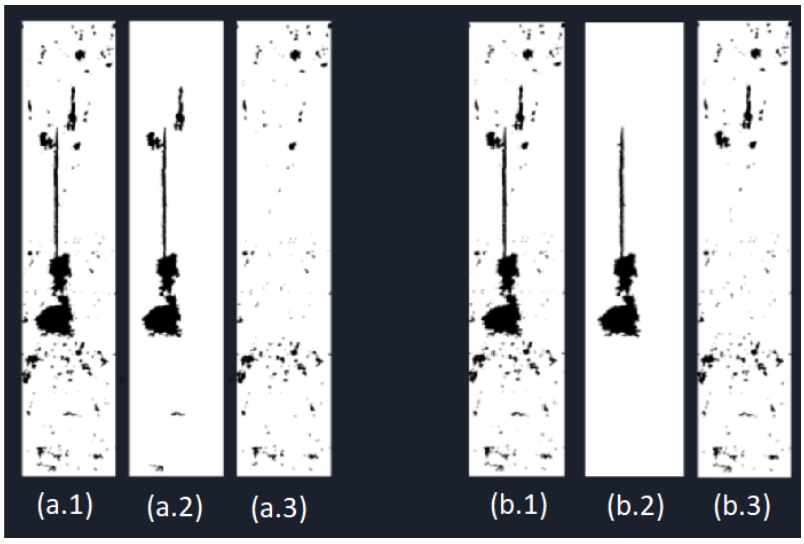

Figura 6.3: Comparação de resultados em imagem apenas com vugues. As estruturas que aparecem em (a.2) e (b.2) foram classificadas de maneira errônea.

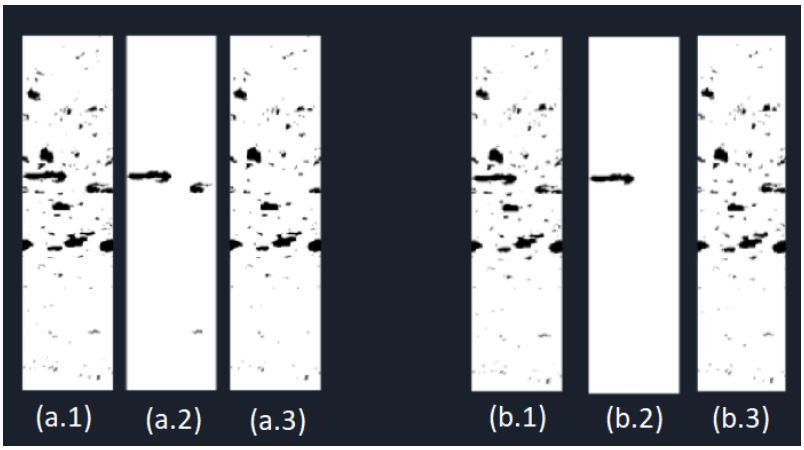

Figura 6.4: Comparação de resultados em imagem apenas com vugues. As estruturas que aparecem em (a.2) e (b.2) foram classificadas de maneira errônea.

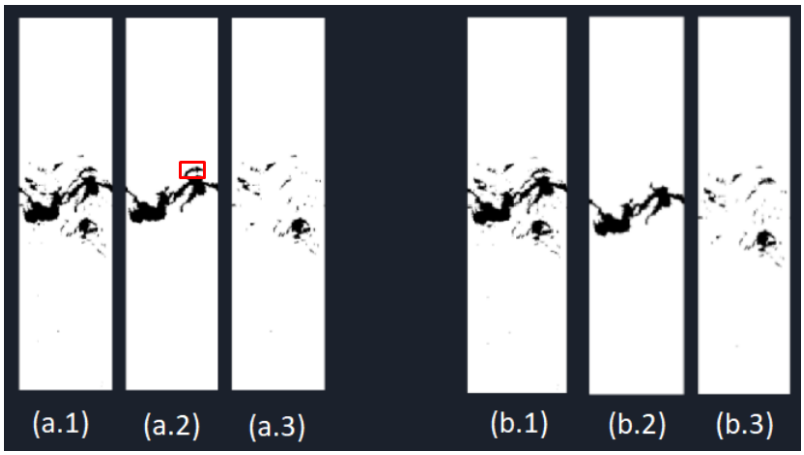

Figura 6.5: Comparação de resultados em imagem com vugues e uma fratura. Estruturas classificadas errado circuladas em vermelho.

6.1.3 Conclusões Parciais

Analisando os resultados da seção anterior, podemos perceber que ambos os algoritmos são eficazes na detecção de fraturas, como evidenciado nas Figuras 6.1, 6.2 e 6.5. Na Figura 6.2, o algoritmo em (a) é capaz de classificar a fratura e os vugues melhor do que o algoritmo (b), pois (b) classificou mais vugues de forma errônea do que (a), que só errou em uma pequena estrutura logo acima da fratura, que deveria ser classificada como vugue.

Na Figura 6.1, ocorre uma situação similar, mas nesse caso, o erro de (a) está na parte de cima da imagem e também logo abaixo da fratura, com uma pequena estrutura que foi classificada errado.

Em ambas as Figuras 6.3 e 6.4, o algoritmo (b) tem uma performance melhor, pois há menos estruturas classificadas como fraturas, já que na imagem classificada só haviam vugues.

Por fim, na Figura 6.5, ambos os algoritmos classificam o dado de maneira simiar, mas o algoritmo (b) consegue identificar um pequeno vugue logo acima da fratura, que (a) classifica de maneira errônea, pois está conectado com a estrutura da fratura.

Portanto, de modo geral, os dois algoritmos têm resultados similares e podem ser utilizados de maneira complementar.

6.2 Abordagem por Aprendizado Profundo

6.2.1 Implementação da Geração de Dados Sintéticos

A implementação da geração de dados sintéticos, descrita na seção 5.2.1, foi feita em uma classe chamada *BoreholeDataGenerator*, que possui seis métodos principais, cujas assinaturas estão representadas no Listing 6.3.

```
1
2  def get_noise_image(self,
3                      width,
4                      height,
5                      scale=100.0,
6                      octaves=6,
7                      persistence=0.5,
8                      lacunarity=2.5,
9                      base=0)
10
11 def draw_fracture(self,
12                     img,
13                     phi,
14                     theta,
15                     color,
16                     aperture=3,
17                     shrinking=5,
18                     shift=0,
19                     scale=100.0,
20                     octaves=6,
21                     persistence=0.5,
22                     lacunarity=2.5,
```

```
                    base=0)

def draw_vugues(self,
                width,
                height,
                threshold=0.3,
                scale=75.0,
                octaves=8,
                persistence=0.5,
                lacunarity=2.5,
                base=0)

def get_random_data(self,
                    width,
                    height,
                    counter,
                    add_padding,
                    threshold_area)

def get_random_dataset(self,
                       size,
                       width,
                       height,
                       seed,
                       threshold_area = 75,
                       mask_rcnn_base_path = None,
                       train_percentage = 0.7,
                       test_percentage = 0.2)
```

Listing 6.3: Assinaturas dos métodos da classe *BoreholeDataGenerator*.

O método ***get_noise_image*** utiliza a função *pnoise2* da biblioteca noise para obter imagens com Ruído de Perlin, como a Figura 5.12. Os parâmetros do método são utilizados pela função *pnoise2* para regular o aspecto do ruído.

O método ***draw_fracture*** implementa o procedimento de geração de fraturas sintéticas descrito na seção 5.2.1, para gerar uma única fratura. Os parâmetros do método foram definidos na Equação 5-10, onde temos as correspondências entre as váriaveis: $\phi = phi$, $\theta = theta$, $k = aperture$ e $s = shift$. O parâmetro *cor* é utilizado para definir o valor dos pixels da fratura, enquanto os outros parâmetros são utilizados para gerar o ruído de perlin.

Durante a geração, todos os parâmetros correspondentes à Equação 5-10 são aleatorizados, com as respectivas faixas de valores: $phi \in \{10, ..., 40\}$ (em graus), $theta \in \{-180, ..., 180\}$ (também em graus), $aperture \in \{3, ..., 11\}$

e $shift \in \{0, ..., (h - 200)\}$, onde h é a altura da imagem. Além disso, $base \in \{0, ..., 300\}$.

O método ***draw_vugues*** implementa a geração de vugues descrita na seção 5.2.1. Para isso, ele utiliza o parâmetro *threshold* como valor limite para o método do intervalo, descrito na mesma seção. Os outros parâmetros são utilizados para regular o ruído de perlin utilizado para a geração, que é obtido com o método *get_noise_image*.

O método ***get_random_data*** utiliza os outros métodos para gerar os dados completos. Primeiramente, ele gera um número aleatório entre três e cinco fraturas, com o método *draw_fracture*, adiciona elas à mesma imagem utilizando a Equação 5-13 e, em seguida, aplica o método do intervalo descrito na seção 5.2.1, com intervalo válido entre -0.1 e 0.0008. Com isso, ja é possível obter imagens como a Figura 5.13.

Além disso, ele também chama o método *draw_vugues* dentre 3 e 5 vezes, aleatóriamente, somando-os à imagem com as fraturas, utilizando a Equação 5-15.

Após gerar uma imagem completa, com dimensões (altura e largura, equivalentes aos parâmetros *height* e *width*, respectivamente) iguais a 800 e 160 pixels, o método começa a prepará-la para o uso no treinamento da rede neural. Como é difícil adequar redes a dimensões de entrada arbitrários no caso de imagens, a imagem gerada é recortada em partes quadradas, com dimensões de 160 pixels de altura e largura. Os recortes são feitos com sobreposições, ou seja, se o primeiro recorte é feito entre os índices de altura 0 e 160, o próximo é feito entre 80 e 240, e o terceiro entre 160 e 320.

Depois de gerar as imagens nesses intervalos, são geradas máscaras binárias, uma para estrutura, além de anotações indicando a classe de cada máscara, para todas as imagens. Entretanto, as máscaras só são geradas caso a área da estrutura seja maior que o valor do parâmetro *threshold_area*, que é igual a 75 pixels.

Ao final do processo, o método retorna os dados gerados na forma de uma lista de dicionários, um para cada imagem, como representado no Listing 6.4.

```
[{
    "image_id": int,
    "raw_image": list,
    "masks": list,
    "class_ids": list
}, ...]
```

Listing 6.4: Formato de retorno do método *get_random_data*

Em que *image_id* é o identificador único da imagem, *raw_image* é a imagem no formato de uma lista 2D com dimensões iguais a (altura, largura), *masks* é a lista de máscaras das estruturas da imagem com dimensões (altura, largura, número de estruturas) e *class_ids* é a lista de identificadores de classe com dimensão igual ao número de estruturas.

Por fim, o método ***get_random_dataset*** utiliza o métdo *get_random_data* para gerar um número de dados total igual ao valor do parâmetro *size*. Desses dados, uma porcentagem igual a *train_percentage* será salva no dataset de treino, enquanto uma porcentagem igual a *test_percentage* será salva no dataset de teste. Além disso, o resto dos dados será salvo no dataset de validação. Todos os três datasets são armazenados no formato *JSON*, como uma lista de dicioários no formato indicado no Listing 6.4.

A *seed* da biblioteca *random* utilizada para a geração dos dados utilizados para o treinamento, validação e teste da rede neste projeto foi 0. Além disso, foram geradas 240 imagens, onde 70% foram destinadas ao conjunto de treino, 20% ao conjunto de teste e o restante ao conjunto de validação.

6.2.2 Implementação da Rede

Para a rede, foi utilizada a implementação de (Mask R-CNN 2017). Este framework disponibiliza duas classes que podem ser extendidas para leitura e armazenamento e preprocessamento dos dados: *Config* e *Dataset*. A classe *Config* é utilizada para configurar os parâmetros do framework quando estiver lidando com os dados em questão. Os parâmetros utilizados para o treinamento da rede, para o dataset deste projeto foram:

```
1
2  class BoreholeConfig(Config):
3      """
4      Configuration for training on the dataset.
5      Derives from the base Config class and overrides some
6      values.
7      """
8      # Give the configuration a recognizable name
9      NAME = "borehole"
10
11     # Number of images on GPU at a time
12     IMAGES_PER_GPU = 1
13
14     # Number of classes (including background)
15     NUM_CLASSES = 3  # Background, fracture, vug
16
17     # Number of training steps per epoch
```

```
18      STEPS_PER_EPOCH = 100
```

Listing 6.5: Implementação da classe que extende *Config*.

Como representado no Listing 6.5, a variável *NAME* indica o nome do *dataset*, *IMAGES_PET_GPU* indica a quantidade de imagens que serão processadas pela GPU por vez, *NUM_CLASSES* define o número de classes que serão identificadas (nesse caso, fundo, fratura e vugue) e *STEPS_PER_EPOCH* indica o número de passos a cada época de treinamento.

A classe ***Dataset*** possui diversos métodos que podem ser sobrescritos para fazer a leitura, armazenamento e preprocessamento dos dados. Para esta implementação, foram sobrescritos os métodos representados no Listing 6.6.

```
1
2  class BoreholeDataset(utils.Dataset):
3
4      def load_borehole(self, dataset_dir, subset=1.0)
5
6      def load_mask(self, image_id)
7
8      def load_image(self, image_id)
```

Listing 6.6: Assinaturas dos métodos da classe que extende *Dataset*.

O método ***load_borehole*** lê o arquivo *JSON* (gerado anteriormente com a implementação descrita na seção 6.2.1) no diretório *dataset_dir*, preprocessa e guarda, em uma variável interna, uma porcentagem igual a *subset* dos dados que estavam no arquivo.

O método ***load_mask*** carrega as máscaras e anotações referentes a um identificador de imagem *image_id*.

Por sua vez, o método ***load_image*** carrega a imagem referentes a um identificador de imagem *image_id*.

6.2.3 Testes

Primeiramente, é necessário testar se o carregamento e preprocessamento dos dados foi feito da maneira correta. Para isso, foi utilizado o método *visualize.display_top_masks* da biblioteca *visualize* do *framework* utilizado. A Figura 6.6 mostra o resultado do preprocessamento.

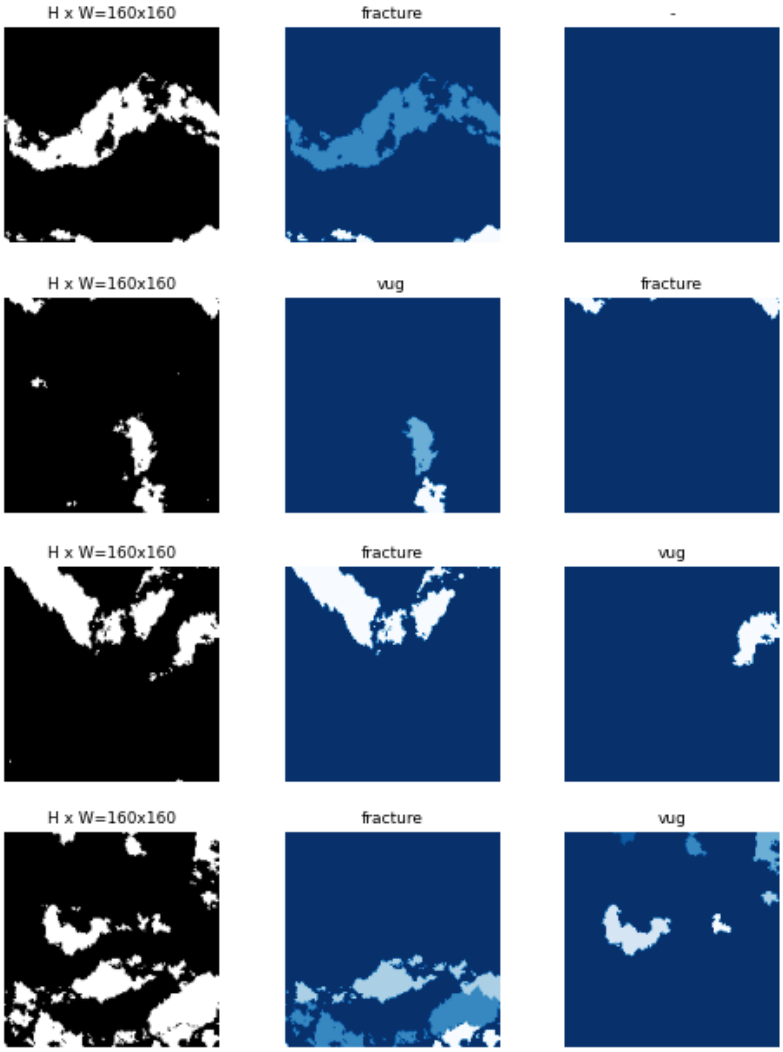

Figura 6.6: Visualização do carregamento e preprocessamento de dados.

Como as máscaras da Figura 6.6 estão com suas posições e classes corretas, o carregamento e preprocessamento também estão corretos.

Em seguida, a rede foi treinada por 20 épocas, utilizando as configurações indicadas no Listing 6.5, realizando *transfer learning*, inicializando a rede com os pesos de uma instância pré-treinada utilizando o *dataset Microsoft COCO: Common Objects in Context* (Lin et al. 2015) disponível no repositório do *framework* (Mask R-CNN 2017). Para o caso de *transfer learning*, o *framework* oferece uma opção para treinar apenas parte das camadas, chamadas internamente de *heads*, como indicado no Listing 6.7.

```
model.train(dataset_train,
            dataset_validation,
            learning_rate = config.LEARNING_RATE,
            epochs=20,
```

```
5                 layers='heads')
```

Listing 6.7: Método de treino do modelo.

Onde *dataset_train* (dados de treino) e *dataset_validation* (dados de validação) são instâncias da classe *BoreholeDataset*, *learning_rate* é a taxa de aprendizado da rede, que é definida na classe *Config*, pois não foi sobrescrita na classe *BoreholeConfig*, *epochs* é o número de épocas e *layers* é o conjunto de camadas da rede a ser treinado.

Após o treinamento, foram realizadas predições no dataset de testes gerado com a implementação da seção 6.2.1. Para avaliar o modelo, foi utilizada a métrica *Mean average precision* (mAP).

Para definir essa métrica é necessário, primeiramente, definir as métricas de *Precision* (P), *Recall* (R), *Intersection over union* (IoU) também e os conceitos de Positivo verdadeiro (TP), Falso positivo (FP) e Falso negativo (FN), para o caso em análise.

Primeiramente, sendo A a máscara predita pela rede e B a máscara correta, temos:

$$IoU = \frac{|A \cap B|}{|A \cup B|} \tag{6-1}$$

A partir do valor do IoU, podemos definir que uma máscara é um Positivo verdadeiro quando seu IoU em relação a máscara verdadeira é maior que 0,5, enquanto um Falso positivo seria o caso em que o IoU é menor que 0,5. Já o Falso negativo ocorre quando a rede não gera máscara para um objeto que estava presente na imagem.

A partir desses conceitos, podemos definir:

$$P = \frac{|TP|}{|TP + FP|} \tag{6-2}$$

$$R = \frac{|TP|}{|TP + FN|} \tag{6-3}$$

Agora, sendo $\{P_1, ..., P_n\}$ e $\{R_1, ..., R_n\}$ os conjuntos de *precisions* e *recalls* para valores crescentes de valor limite do IoU para considerar um TP, onde o índice 1 equivale ao IoU = 0,5 e o índice n, ao IoU = 1. Podemos definir:

$$mAP = \sum_{i=1}^{n} (R_i - R_{i-1}) P_i \tag{6-4}$$

Para avaliar um conjunto de k imagens, podemos realizar a média do mAP de cada imagem e obter o mAP médio do conjunto, portanto:

$$mAP_{conjunto} = \frac{\sum_{i=1}^{k} mAP_i}{k} \tag{6-5}$$

Após realizar a predição no conjunto de testes, o valor da Métrica 6-5 avaliada no conjunto, considerando uma precisão de três casas decimais, foi igual a 0,7. As figuras a seguir demonstram representações visuais das classificações feitas pela rede. Em todas as imagens, (a) é a classificação da rede e (b) é a classificação de referência, em que as fraturas aparecem em branco, os vugues em cinza e o fundo em preto.

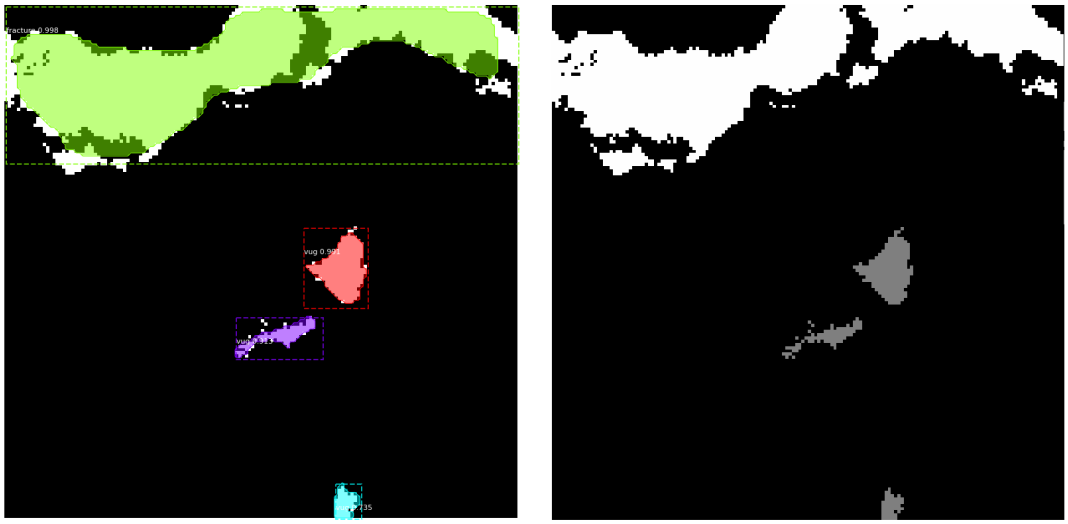

Figura 6.7: Exemplo de classificação com dados sintéticos, onde a rede gerou uma máscara verde, classificada como fratura, e outras classificadas como vugues.

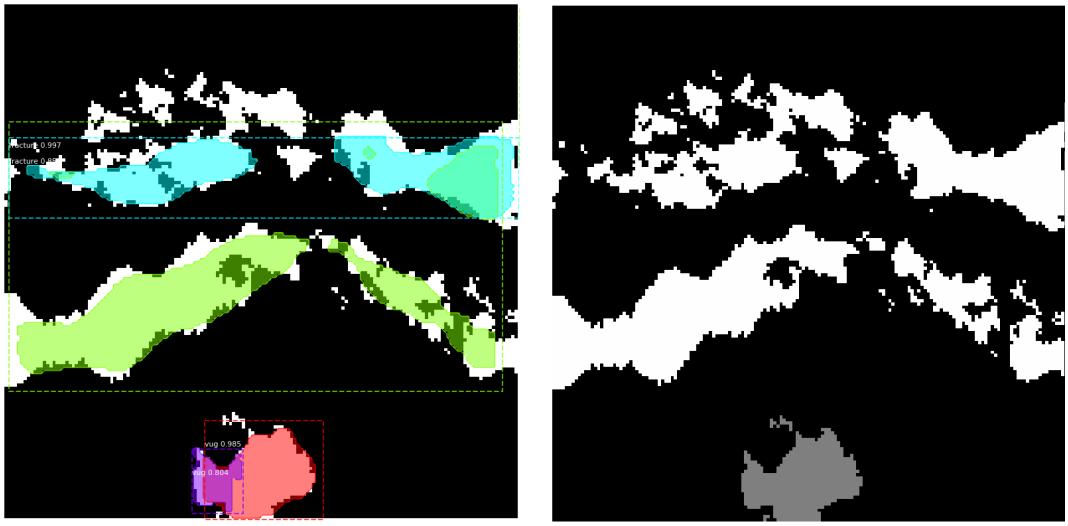

Figura 6.8: Exemplo de classificação com dados sintéticos, onde a rede gerou as máscaras de cores verde e azul, classificadas como fraturas, e as demais classificadas como vugues.

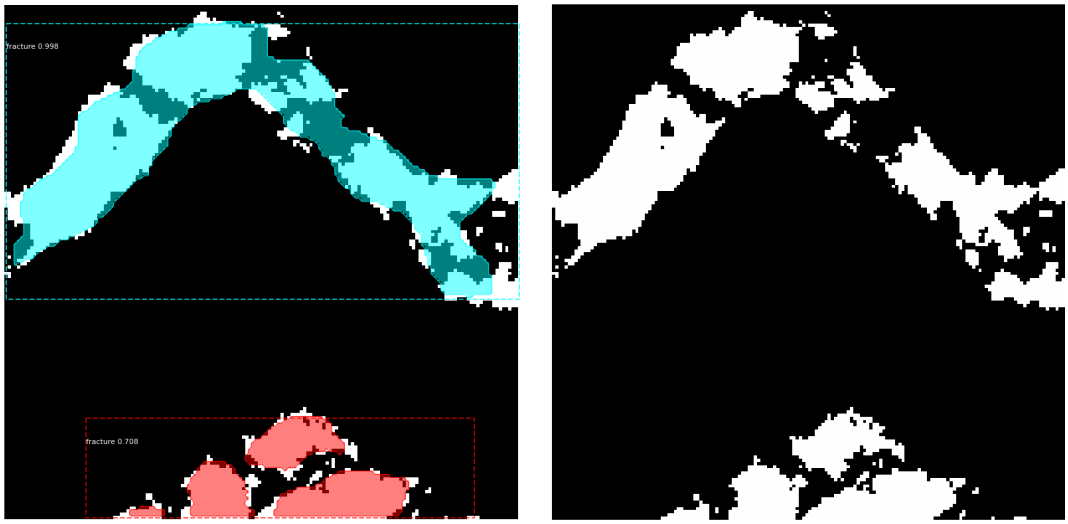

Figura 6.9: Exemplo de classificação com dados sintéticos, onde a rede gerou duas máscaras, uma para cada fratura da imagem, de maneira correta.

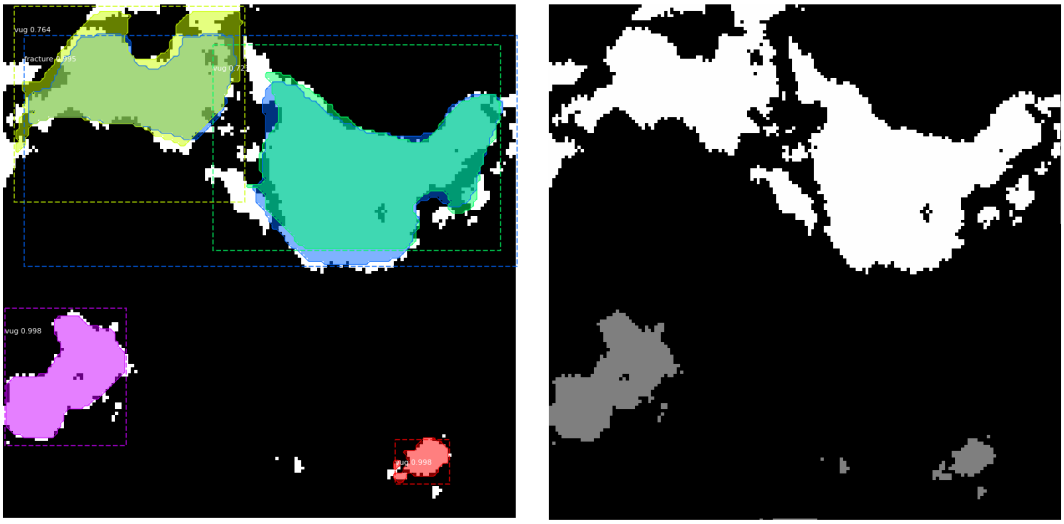

Figura 6.10: Exemplo de classificação com dados sintéticos, onde a rede gerou uma máscara desconexa azul para a fratura e outras quatro de cores variadas, classificadas como vugues.

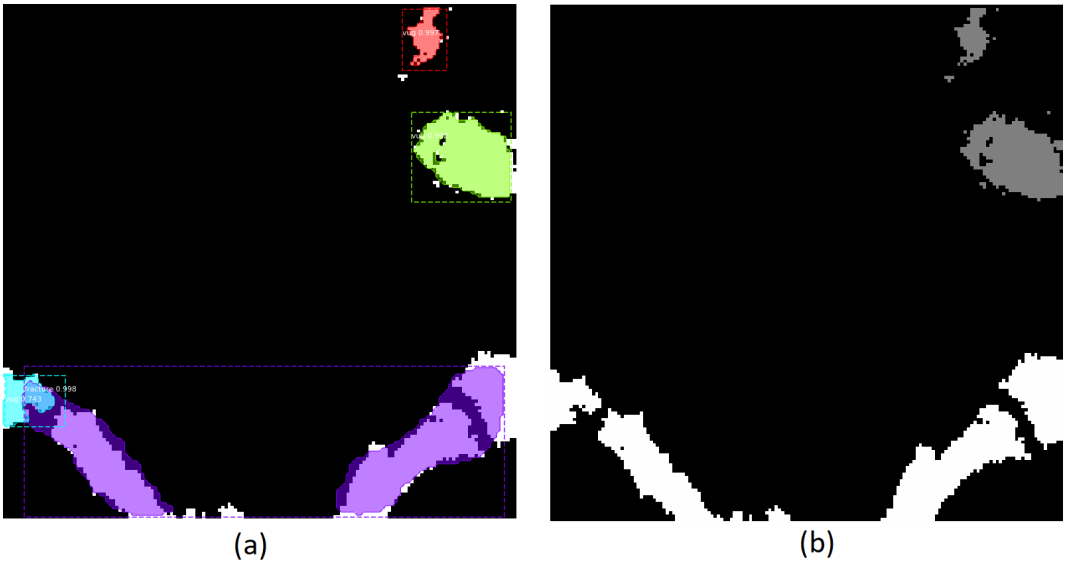

Figura 6.11: Exemplo de classificação com dados sintéticos, onde a rede gerou uma máscara roxa para a fratura e outras três de cores variadas, classificadas como vugues.

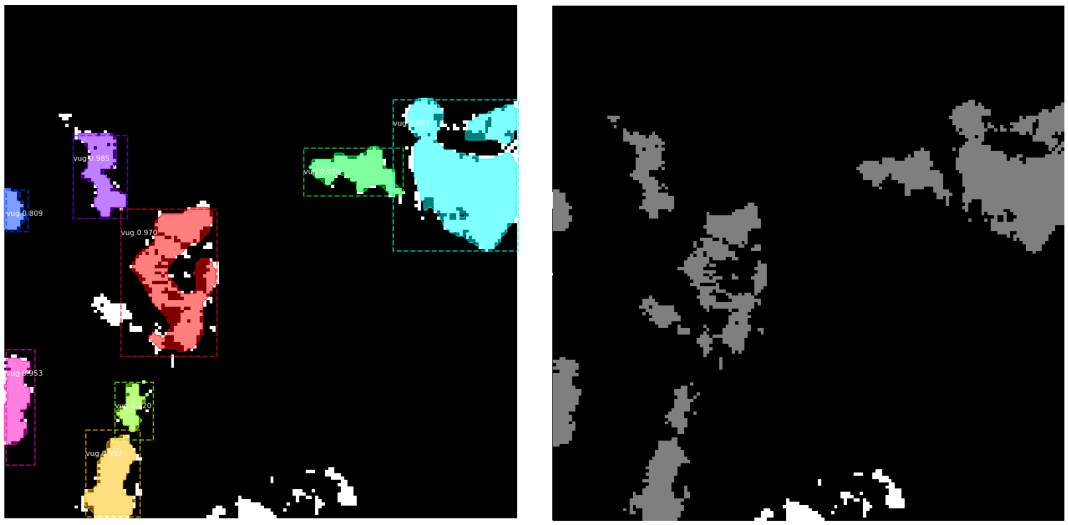

Figura 6.12: Exemplo de classificação com dados sintéticos, onde a rede gerou máscaras para os vugues.

Além disso, também foram feitos testes em dados reais. Não é possível medir a mAP dessas predições, pois os dados não possuem anotações. Entretando, é possível avaliá-los de forma visual. As imagens abaixo foram resultados de testes com dados reais.

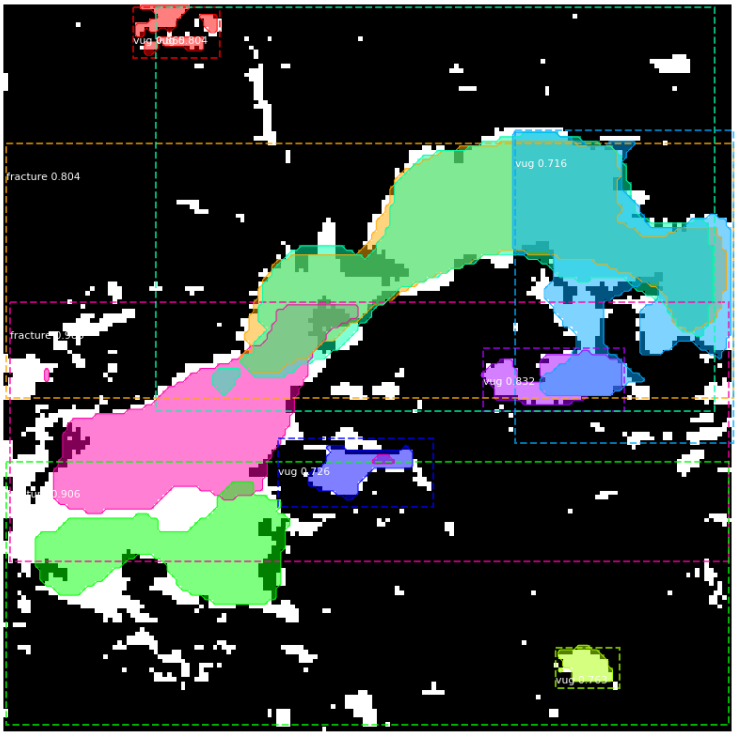

Figura 6.13: Exemplo de classificação com dados reais, onde a rede gerou as máscaras de cores laranja, rosa e verde (abaixo da rosa), classificadas como fraturas, e máscaras de cores roxa, azul, vermelha e os outros dois tons de verde, como vugues.

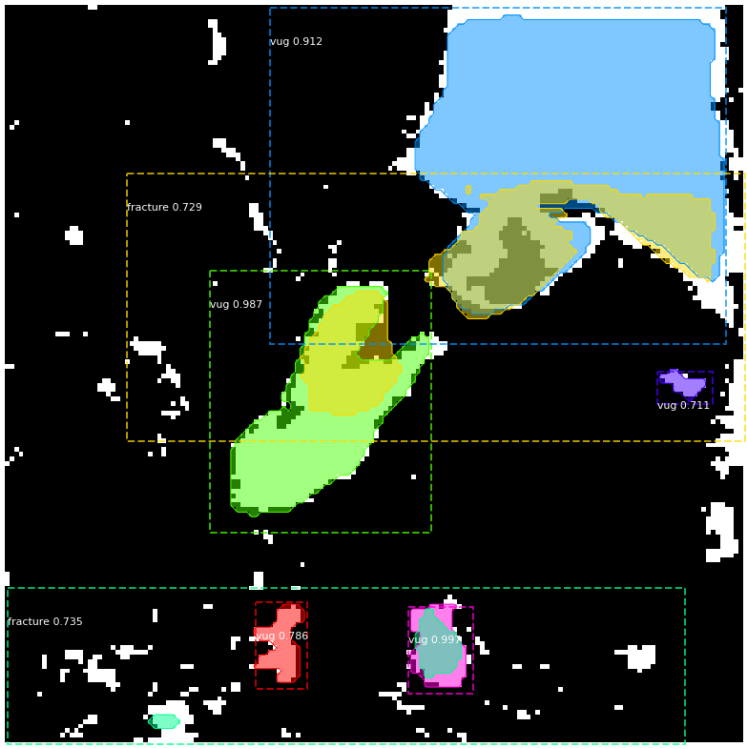

Figura 6.14: Exemplo de classificação com dados reais possuindo apenas vugues, onde a rede gerou as máscaras de cores verde (abaixo da vermelha) e amarela, classificadas como fraturas e as demais classificadas como vugues.

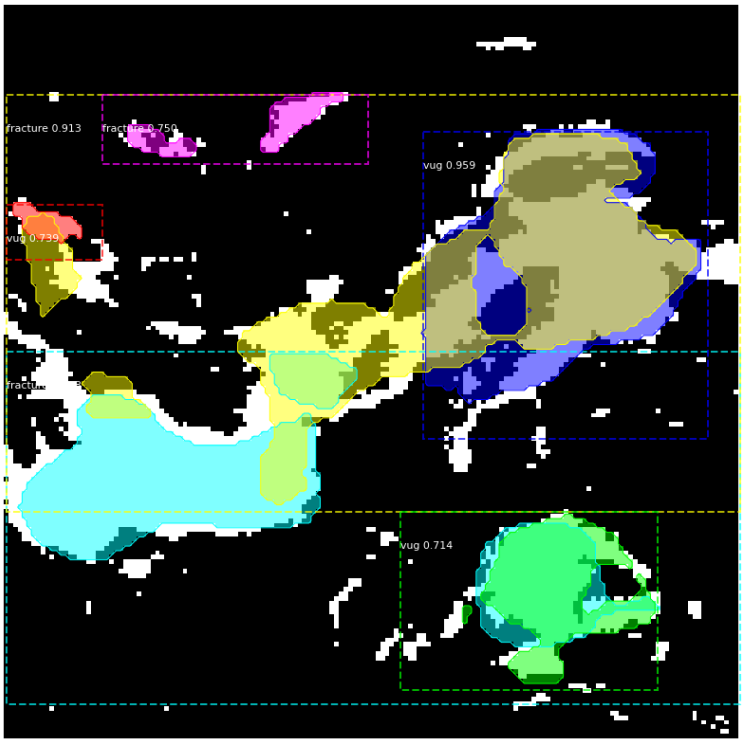

Figura 6.15: Exemplo de classificação com dados reais, onde a rede gerou as máscaras de cores amarela, azul clara e rosa, classificadas como fraturas e as demais como vugues.

6.2.4 Conclusões do Aprendizado Profundo

Analisando os resultados da seção 6.2.3, é possível concluir que, para o caso de dados sintéticos, a rede é capaz de fazer uma segmentação razoável, já que, quanto mais próxima de 1 for a mAP, mais preciso é o modelo e, no caso presente, o mAP foi de 0,7.

Fazendo uma análise visual dos resultados apresentados nas Figuras 6.7 a 6.12, podemos perceber que a rede é capaz de encontrar máscaras que aproximam bem a forma dos vugues, em sua maioria. Entretanto, alguns casos como na figura 6.8, ela prediz dois vugues onde havia apenas um.

No caso das fraturas, a rede se sai melhor quando elas são mais conexas, estreitas e isoladas de outras fraturas. Um exemplo de uma boa segmentação é a Figura 6.7, enquanto há figuras como a 6.10 em que a rede confunde uma fratura com um conjunto de vugues e prediz várias máscaras diferentes. Por outro lado, a Figura 6.9 é um exemplo de uma boa segmentação de fraturas desconexas.

No caso dos dados reais, analisando visualmente, é possível perceber que

a rede consegue identificar melhor os vugues, mas se confunde com as fraturas. Na figura 6.13, por exemplo, ela é capaz de identificar a existencia da fratura, mas identifica três em vez de uma e também identifica vugues sobrepostos a ela. Isso indica que ainda há melhorias a serem feitas na geração de dados sintéticos, principalmente com relação às fraturas.

Portanto, essa modelagem é um indicativo de que a rede de detecção de objetos é capaz de segmentar fraturas e vugues, mesmo que estas estejam desconexas e abre espaço para a criação de modelos mais aprimorados a partir do seu aperfeiçoamento. Alguns detalhes serão discutidos no Capítulo 7.

7
Considerações Finais

Nesse trabalho, foram descritas duas abordagens para resolver o problema proposto. A abordagem por maior caminho em grafo obteve resultados satisfatórios, enquanto a outra ainda tem pontos a serem melhorados, mas mostra um caminho promissor.

A geração de dados sintéticos é uma contribuição que facilita a criação de algoritmos de aprendizado supervisionado, uma vez que torna fácil a criação de *datasets* anotados.

A abordagem por maior caminho em grafo se mostrou uma boa alternativa à solução reproduzida em (Gabrielle 2020), por apresentar resultados semelhantes e poder ser utilizada de forma complementar. Como ela é uma resolução semi-automática, ainda é possível melhorá-la criando um passo de automatização, que não foi realizado devido ao escopo do projeto. Para trabalhos futuros, é possível adaptar a geração de dados sintéticos para gerar fraturas conexas e utilizá-la no treino de algoritmos de aprendizado supervisionado para criar o passo de automatização.

Por sua vez, a abordagem por aprendizado profundo, diferentemente da abordagem reproduzida por (Gabrielle 2020) e da abordagem por maior caminho em grafo, ja é uma forma de resolução automática. Devido ao escopo do projeto, não foi possível aperfeiçoar este método, mas para trabalhos futuros, alguns testes seriam relevantes:

1. Alterar a janela dos recortes dos dados sintéticos adicionando colunas de zeros ao redor da parte central (por exemplo, fazer recortes de altura 512 em vez de 160 e adicionar 176 colunas de altura 512 de cada lado da parte central, que tem largura 160 e altura 512, para manter a imagem quadrada), para que cada predição possa contemplar dados mais completos.

2. Gerar uma quantidade maior de dados e alterar os parâmetros de treinamento.

3. Gerar dados com uma quantidade maior de fraturas conexas.

4. Testar outras redes, tanto de segmentação quanto detecção de objetos.

Caso pudesse começar o projeto agora, focaria na abordagem por aprendizado profundo, que não é limitada a fraturas conexas e intrinsecamente automática, testando mais opções, como as citadas acima, para conseguir valores melhores de mAP nos dados simulados e melhores resultados nos dados reais.

Referências bibliográficas

[Asquith et al. 2004] ASQUITH, G.; KRYGOWSKI, D.; HENDERSON, S. ; HURLEY, N.. **Basic well log analysis**. American Association of Petroleum Geologists, 01 2004.

[Bulterman et al. 2002] BULTERMAN, R. W.; VAN DER SOMMEN, F. W.; ZWAAN, G.; VERHOEFF, T.; VAN GASTEREN, A. J. M. ; FEIJEN, W. H. J.. **On computing a longest path in a tree**. Inf. Process. Lett., 81(2):93–96, Jan. 2002.

[Facon 2002] FACON, J.. **Processamento e análise de imagens**, 2002. [Online; acesso em Maio de 2021].

[Felix et al. 2013] FÉLIX, F. A.; NASCIMENTO, E. S. D. ; BORBA, C.. **Perfis de imagem de poços de petróleo**. Caderno de Graduação - Ciências Exatas e Tecnológicas - UNIT - SERGIPE, 1(2):61–78, fev. 2013.

[Gabrielle 2020] ANJOS, G.. **Classificação de estruturas porosas em perfil de imagem de poço com operadores morfológicos**. 07 2020.

[Girshick 2015] GIRSHICK, R.. **Fast r-cnn**, 2015.

[He et al. 2016] HE, K.; ZHANG, X.; REN, S. ; SUN, J.. **Deep residual learning for image recognition**. In: PROCEEDINGS OF THE IEEE CONFERENCE ON COMPUTER VISION AND PATTERN RECOGNITION (CVPR), June 2016.

[He et al. 2017] LIN, T.-Y.; DOLLÁR, P.; GIRSHICK, R.; HE, K.; HARIHARAN, B. ; BELONGIE, S.. **Feature pyramid networks for object detection**, 2017.

[He et al. 2018] HE, K.; GKIOXARI, G.; DOLLÁR, P. ; GIRSHICK, R.. **Mask r-cnn**, 2018.

[Hu et al. 2011] HU, X.; SUN, B.; ZHAO, H.; XIE, B. ; WU, H.. **Image skeletonization based on curve skeleton extraction**. In: Jacko, J. A., editor, HUMAN-COMPUTER INTERACTION. DESIGN AND DEVELOPMENT APPROACHES, p. 580–587, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Jesus et al. 2016] JESUS, C. M. D.; COMPAN, A. L. M. ; SURMAS, R.. **Permeability estimation using ultrasonic borehole image logs in dual-porosity carbonate reservoirs**. Petrophysics, 57(06):620–637, Dezembro 2016.

[Karger et al. 1997] KARGER, D.; MOTWANI, R. ; RAMKUMAR, G. D. S.. **On approximating the longest path in a graph**. Algorithmica, 18(1):82–98, May 1997.

[Li 2017] LI, B.; TAN, X.; WANG, F.; LIAN, P.; GAO, W. ; LI, Y.. **Fracture and vug characterization and carbonate rock type automatic classification using x-ray ct images**. volumen 153, p. 88–96, 2017.

[Li et al. 2019] LI, X.-N.; SHEN, J.; YANG, W.-Y. ; LI, Z.-L.. **Automatic fracture–vug identification and extraction from electric imaging logging data based on path morphology**. volumen 16, 12 2018.

[Lin et al. 2015] LIN, T.-Y.; MAIRE, M.; BELONGIE, S.; BOURDEV, L.; GIRSHICK, R.; HAYS, J.; PERONA, P.; RAMANAN, D.; ZITNICK, C. L. ; DOLLÁR, P.. **Microsoft coco: Common objects in context**, 2015.

[Mask R-CNN 2017] ABDULLA, W.. **Mask r-cnn for object detection and instance segmentation on keras and tensorflow**. `https://github.com/matterport/Mask_RCNN`, 2017.

[Ren et al. 2016] REN, S.; HE, K.; GIRSHICK, R. ; SUN, J.. **Faster r-cnn: Towards real-time object detection with region proposal networks**, 2016.

[Rosa et al. 2006] ROSA, A. J.; XAVIER, J. A. ; DE SOUZA CARVALHO, R.. **Engenharia de Reservatório de Petróleo**. Interciência, Brasil, 2006.