Roteiro Avaliação Parcial 3

Fundamentos de Informática em Imagens Médicas

Elaborado por: João Pedro Timossi Rozette
Supervisionado por: Paulo Mazzoncini de Azevedo Marques

---

1. Introdução

Este roteiro tem como objetivo apresentar um guia aos alunos para a abordagem e resolução do problema proposto: a classificação automatizada corpos vertebrais quanto ao tipo de fratura por compressão vertebral (FCVs), benignas ou malignas, a partir de imagens de ressonância magnética (RM) ponderadas em T1 da coluna lombar.

Recomenda-se que o problema seja resolvido utilizando a linguagem Python e o ambiente Google Colab (ou outros IDEs baseados em notebooks python).

1.1. O Problema

O desafio consiste em diferenciar FCVs benignas (secundárias à osteoporose) de FCVs malignas (secundárias à infiltração neoplásica). Para isso, utilizaremos um conjunto de dados de RM que inclui imagens originais e suas respectivas segmentações das vértebras fraturadas.

1.2. Objetivos Principais

Este roteiro será dividido nas seguintes seções:

- 1. Configuração do Ambiente: Preparação do ambiente de desenvolvimento.
- 2. Explorar e Organizar: Entender e organizar os arquivos de imagem e de segmentação.
- 3. Processar as Imagens: Alinhar as máscaras de segmentação com as imagens originais para extrair com precisão a área de uma vértebra específica.
- 4. Criar Patches: Gerar imagens 2D recortadas centradas na região de interesse.
- 5. Dividir a Base: Separar os *patches* em diretórios de treino, teste e validação, garantindo que todas as imagens de um mesmo paciente permaneçam no mesmo conjunto para evitar "vazamento de dados".
- 6. Modelagem e Classificação: Construção, treinamento e avaliação de modelos de aprendizado profundo de máquina.

2. Configuração do Ambiente

Para esta atividade, recomenda-se a utilização do Google Colab ou Jupyter Notebook, que oferecem um ambiente interativo para desenvolvimento em Python.

2.1. Instalação e Importação das Bibliotecas

As principais bibliotecas necessárias para este roteiro são: os (para manipulação de sistema de arquivos), shutil (para operações de alto nível em arquivos), pandas (para manipulação de dados tabulares), nrrd (para leitura de arquivos .nrrd), numpy (para operações numéricas), matplotlib.pyplot (para visualização), PIL (Pillow, para processamento de imagens), math (para funções matemáticas) e tensorflow (para aprendizado de máquina).

A maioria das bibliotecas que utilizaremos já são pré-instaladas em ambientes como o Google Colab. Caso a importação direta de uma biblioteca apresente erro, poderá ser instalada utilizando o gerenciador de pacotes pip.

Exemplo:

```
pip install pynrrd
```

2.2. Montagem do Google Drive (Opcional, para Google Colab)

Se você estiver utilizando o Google Colab e seus arquivos de dados estiverem armazenados no Google Drive, será necessário montar o Drive para que seu ambiente de execução possa acessá-los.
Isso pode ser feito diretamente na interface do Colab, ou rodando o script abaixo.

```
from google.colab import drive
drive.mount('/content/drive')
```

3. Carregamento e Exploração dos Dados

Nesta seção, você aprenderá a carregar as imagens originais e as máscaras de segmentação no ambiente Python e a realizar uma exploração inicial dos dados.

Baixe a base de dados completa diretamente do *Zenodo* (https://zenodo.org/records/13274445) e faça o upload dela no Drive. Em seguida, use a ferramenta *ZIP Extractor* para descompactar a base.

3.1. Estrutura dos Dados

O conjunto de dados é composto por:

- *1-Original_Anon_Dicom*: Arquivos no formato .dcm correspondentes aos exames anonimizados dos pacientes.
- *2-Rescaled_256_NRRD:* Arquivos no formato .nrrd contendo as imagens de RM ponderadas em T1
- *3-Segmentation_NRRD:* Arquivos no formato .seg.nrrd contendo as segmentações das vértebras fraturadas.
- Patients.xlsx (planilha excel) com informações sobre ID dos pacientes, tipo e localização das fraturas, idade e gênero.

3.2: Exploração dos Arquivos

Nesta etapa, vamos entender a estrutura dos nossos dados e organizá-los em pastas separadas por classe (benigna ou maligna).

3.2.1 Explorando arquivos .nrrd

Ao final desta etapa teremos dois dicionários, um com as máscaras e outro com as imagens obtidas por ressonância magnética. As chaves (Keys) dos dicionários das máscaras serão o índice do paciente e a respectiva vértebra (por exemplo a segmentação da vértebra L1 do paciente 1 terá a chave P1L1). Enquanto as chaves do dicionário de imagens serão somente o índice do paciente (P1, P2, P3, ..., P91)

- Crie uma variável *string* chamada *path_mask* e atribua a ela o caminho do diretório onde estão localizadas as segmentações das imagens
- Acesse esse diretório usando o comando *chdir* da biblioteca *os* (*os.chdir(path_mask)*)
- Crie um dicionário vazio chamado *masks* e outro chamado *header_mask*
- Crie duas strings *sub1* e *sub2* e atribua à primeira o nome do diretório das segmentações e à segunda o nome da extensão do arquivo. Por exemplo, no meu caso usando Google Colab eu fiz *sub1 = '3-Segmentation_NRRD/'* e *sub2= '.seg.nrrd'* sendo que salvei no caminho *path_mask = "/content/drive/MyDrive/Fundamentos de Informática em Imagens Médicas 2025/HCFMRP_Complete_Anon_VCF_Database/3-Segmentation_NRRD"*

- Agora criamos um loop *for* para ler todos os arquivos neste diretório de uma única vez. Use o comando *for file in os.listdir():* sendo que *os.listdir* gera uma lista com o nome de todos os arquivos do diretório
    - *o* Queremos ler somente os arquivos que terminam em .nrrd, portanto tem que criar um condicional *if file.endswith('.nrrd'):*
        - Atribua a uma variável *string* chamada *file_path* a concatenação de *path_mask* com *file* (este *file* está armazenando o nome do arquivo no loop *for*)
        - Salve em *idx1* e *idx2* o index de *sub1* e *sub2* dentro de *file_path* usando a função *string.index(substring)*
        - Atribua à uma variável *string* chamada *res* uma string vazia (*res=‘’*). Essa variável vai salvar o nome do arquivo que será a chave do dicionário
        - Crie as chaves para o dicionário com o loop

```
for idx in range(idx1 + len(sub1), idx2):
    res = res + file_path[idx]
```

        - Leia a imagem e o cabeçalho usando o comando *read* da biblioteca *nrrd* do respectivo *file_path*

Para ver se deu certo, dê o comando *matplotlib.pyplot.imshow(masks['P1L1'][:,:,0], cmap='gray')* e você deve obter uma imagem como a mostrada na Figura 1.

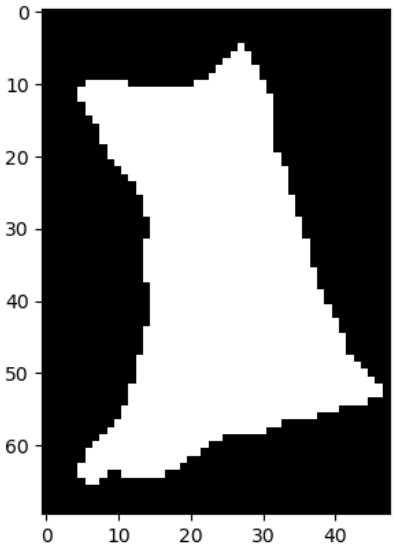


Figura 1: Seção da vértebra L1 do paciente 1

Repita as mesmas etapas, mas agora para as imagens originais. Faça as adaptações que forem necessárias. Dicas: Substitua *path_mask* por *path_images*, o nome do dicionário vazio que armazenará as imagens deve ser diferente não sendo necessário salvar o cabeçalho, e o nome de *sub1* deve ser diferente se você colocou em outro diretório e o de sub2 não será .seg.nrrd, mas somente .nrrd. O corte número 5 do paciente P1 deve ser algo parecido como o mostrado na Figura 2 depois do comando *imshow(images['P1'][:,:,5], cmap='gray')*.

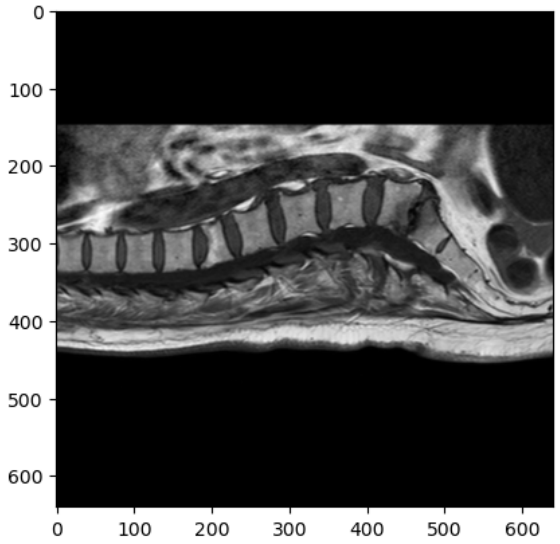


Figura 2: Fatia 5 do paciente P1.

3.2.3. Entendendo a Relação entre as Máscaras e as Imagens Originais

Dê o comando *print(masks['P1L1'][:,:].shape)* e verifique as dimensões das máscaras, que para o caso do P1L1 é (70,48,5). Como vimos em aula, imagens são matrizes cujos valores são relacionados à uma escala de cinza, portanto temos uma imagem de 70 pixels por 48 com 5 camadas.

Dê o comando *print(images['P1'][:,:].shape)* e verifique as dimensões das imagens originais, que para o caso do P1 é (640, 640, 14).

Explore os cabeçalhos das imagens, principalmente os das segmentações. Existem várias informações sobre como a segmentação foi feita. O que importa para nós é a informação contida em *'Segmentation_ReferenceImageExtentOffset'* que guarda a referência da segmentação em relação à imagem original. Dando o comando *masks_header['P1L1']*, vemos que a referência é uma string ‘267 182 5’, que está mostrada no ponto vermelho da Figura 3. Ou seja, a localização da segmentação dentro da imagem original está (x,y,z)=(182,267,182). Essa inversão de x e y aconteceu porque estamos trabalhando com uma imagem da matriz transposta, mas não vai mudar a construção das etapas a seguir

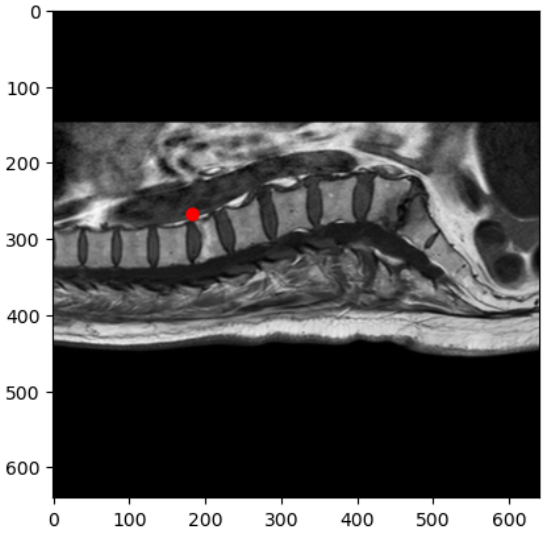


Figura 3: Localização de referência da segmentação em relação à imagem original.

4. Organizando os Arquivos por Classe

Para facilitar o trabalho, vamos copiar (ou mover) os arquivos de imagem (presentes na pasta *2-Rescaled_256_NRRD*) e de máscara (presentes na pasta *3-Segmentation_NRRD*) para subpastas *BENIGN* e *MALIGNANT*. A informação de qual paciente pertence a cada classe está no arquivo *Patients.xlsx*.

Sendo assim, o código desta etapa deve:

- Ler o arquivo Patients.xlsx usando a biblioteca pandas.
- Cria as pastas de destino: BENIGN e MALIGNANT.
- Percorrer a lista de pacientes e, para cada um, copiar (ou mover) o arquivo de exame (.nrrd) e os arquivos de máscara (.seg.nrrd) para a pasta da sua respectiva classe.

Resultado Esperado: Após executar o código, as pastas 2-Rescaled_256_NRRD e 3-Segmentation_NRRD terão subpastas BENIGN e MALIGNANT com os arquivos correspondentes.

5. Processamento das Imagens e Extração de Patches

Esta é etapa central do processamento das imagens, nosso objetivo é extrair patches de 150×150 pixels (esse tamanho pode ser ajustado conforme a necessidade) diretamente sobre cada vértebra previamente segmentada. Esse recorte deve garantir que a região de interesse (ROI) esteja centralizada na vértebra, preservando suas características estruturais. A partir desses patches, realizamos a extração de informações de textura das imagens de ressonância magnética, utilizando as máscaras de segmentação como guia para delimitar precisamente os limites da vértebra.

Abaixo estão duas maneiras de realizar esta tarefa.

5.1 Método 1

5.1.1 Exigências para o uso deste método

Antes de implementar o Método 1, você vai precisar converter as imagens originais e suas máscaras para PNG.
Obs: ao final deste tópico, tem um bloco de código que pode facilitar essa atividade

5.1.2 Objetivo do pipeline

Parear cada imagem de paciente (P7.png, P11.png, …) com todas as máscaras do mesmo paciente (P7L1_*.png, P7L2_*.png, …).

Para cada máscara: aplicar na imagem, centralizar pelo bounding box da máscara, redimensionar mantendo proporção e preencher para 150×150.

5.1.3 Configurações

Caminhos: onde buscar imagens e máscaras e a saída.

Padrões de nome: regex simples para extrair P<n> (imagem) e P<n>L<k> (máscara).

Patch: tamanho fixo (150).

Morfologia: USE_MORPHOLOGY permite ligar/desligar; KERNEL_SIZE controla o “tamanho do pincel” (3×3, 5×5, …).

```
import os, re
from collections import defaultdict
import cv2
import numpy as np

# ===== Caminhos =====
image_dir  = '/content/extracted/ImagensEMascaras/original_images'
mask_dir   = '/content/extracted/ImagensEMascaras/masks'
output_dir = '/content/extracted/patients_gf/output/'
os.makedirs(output_dir, exist_ok=True)

# ===== Parâmetros =====
PATCH_SIZE       = 150              # patch final (quadrado)
USE_MORPHOLOGY   = True             # << ligar/desligar morfologia
KERNEL_SIZE      = 5                # 3 ou 5 são escolhas comuns
KERNEL_SHAPE     = cv2.MORPH_RECT   # MORPH_RECT, MORPH_ELLIPSE, MORPH_CROSS

# ===== Regex de pareamento =====
```

```
RX_IMG  = re.compile(r'\bP(\d+)\b', re.IGNORECASE)       # captura P<n> nas imagens
RX_MASK = re.compile(r'\bP(\d+)L(\d+)\b', re.IGNORECASE) # captura P<n>L<k> nas
máscaras
```

Por que oferecer USE_MORPHOLOGY?

O fechamento morfológico (dilatação seguida de erosão) preenche buracos pequenos e suaviza falhas nas máscaras. Quando a máscara já está “limpa”, você pode desligar. Se notar bordas recortadas demais ao desligar, ligue novamente.

Para este estudo, a princípio, utilizaremos este parâmetro como *False*, mas é importante entender seu funcionamento para trabalhar com outras bases de dados.

5.1.4 Listagem dos arquivos

Filtramos apenas .png e forçamos .lower() para evitar que um eventual .PNG quebre o fluxo de funcionamento do codigo.

sorted dá uma ordem determinística (bom para depuração).

```
image_files = sorted([f for f in os.listdir(image_dir) if
f.lower().endswith('.png')])

mask_files  = sorted([f for f in os.listdir(mask_dir)  if
f.lower().endswith('.png')])
```

5.1.5 Pareamento imagem → lista de máscaras

Indexamos máscaras por paciente.

Ordenamos por “nível” L (L1, L2, L3, …) apenas por conveniência.

```
# Indexa máscaras por paciente

masks_by_patient = defaultdict(list)

for mf in mask_files:

    m = RX_MASK.search(mf)

    if not m:

        continue

    p, L = m.group(1), int(m.group(2))

    masks_by_patient[p].append((L, mf))



# Ordena por nível vertebral

for p in list(masks_by_patient.keys()):

    masks_by_patient[p].sort(key=lambda x: x[0])
```

```
# Mapa final: imagem -> [máscaras...]

images_and_masks = {}

for image_file in image_files:

    mi = RX_IMG.search(image_file)

    if not mi:

        print(f"[aviso] imagem sem P<n>: {image_file} — pulando")

        continue

    p_img = mi.group(1)

    pairs = masks_by_patient.get(p_img, [])

    images_and_masks[image_file] = [m for _, m in pairs]
```

Detalhes importantes:

Funciona com P007 ou P7 do mesmo jeito (regex pega o número).

Se você quiser restringir níveis (quantidade de máscaras por imagem) (ex.: só L1–L5), troque (\d+) por ([1-5]) em RX_MASK.

5.1.6 Centralizar e recortar para 150×150 (mantendo proporção)

Usamos o contorno externo da máscara para pegar um bounding box e cortar a região.

Redimensionamos mantendo a proporção para caber em 150×150.

Preenchemos (padding) com preto até fechar 150×150.

```
def centralize_and_crop(masked_image, mask_clean, target_size=PATCH_SIZE):

    contours, _ = cv2.findContours(mask_clean, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

    if len(contours) == 0:

        return cv2.resize(masked_image, (target_size, target_size),
interpolation=cv2.INTER_LANCZOS4)



    x, y, w, h = cv2.boundingRect(contours[0])

    cropped = masked_image[y:y+h, x:x+w]



    scale = min(target_size / h, target_size / w)
```

```
    resized = cv2.resize(cropped, (max(1, int(w * scale)), max(1, int(h * scale))),
interpolation=cv2.INTER_LANCZOS4)



    h_new, w_new = resized.shape[:2]

    top    = (target_size - h_new) // 2

    bottom = (target_size - h_new + 1) // 2

    left   = (target_size - w_new) // 2

    right  = (target_size - w_new + 1) // 2



    return cv2.copyMakeBorder(resized, top, bottom, left, right,
cv2.BORDER_CONSTANT, value=[0, 0, 0])
```

5.1.7 Loop de processamento

Imagem: carregada em BGR (padrão do OpenCV).

Máscara: carregada em grayscale.

Sem threshold (binarização) e sem normalização

Morfologia apenas se USE_MORPHOLOGY=True.

```
 # Kernel (só será usado se USE_MORPHOLOGY for True)

kernel = cv2.getStructuringElement(KERNEL_SHAPE, (KERNEL_SIZE, KERNEL_SIZE)) if
USE_MORPHOLOGY else None



for image_file, mask_list in images_and_masks.items():

    print(f'Processando: {image_file}')

    if not mask_list:

        print(f'  [aviso] nenhuma máscara para {image_file}')

        continue



    img_path = os.path.join(image_dir, image_file)

    img = cv2.imread(img_path)  # BGR

    if img is None:

        print(f'  [erro] não carregou imagem: {image_file}')

        continue
```

```
    for mask_file in mask_list:
        print(f'  > máscara: {mask_file}')
        mask_path = os.path.join(mask_dir, mask_file)
        mask = cv2.imread(mask_path, cv2.IMREAD_GRAYSCALE)
        if mask is None:
            print(f'  [erro] não carregou máscara: {mask_file}')
            continue

        # Opcional: morfologia para limpar a máscara (NÃO binariza)
        if USE_MORPHOLOGY:
            try:
                mask_clean = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)
            except cv2.error as e:
                print(f'  [erro] morfologia em {mask_file}: {e}')
                mask_clean = mask
        else:
            mask_clean = mask

        # Aplica a máscara na imagem (regiões não-máscara ficam 0)
        masked_image = cv2.bitwise_and(img, img, mask=mask_clean)

        # Centraliza e recorta para 150×150
        final_image  = centralize_and_crop(masked_image, mask_clean,
target_size=PATCH_SIZE)

        # Extrai Lk do nome da máscara
        m = RX_MASK.search(mask_file)
        vertebra_name = f"L{m.group(2)}" if m else "Lx"

        # Salva saída
        base_img = os.path.splitext(image_file)[0]
        out_name = f'processed_{base_img}_{vertebra_name}.jpg'
```

```
        out_path = os.path.join(output_dir, out_name)

        cv2.imwrite(out_path, final_image)

        print(f'  ✔ salvo: {out_path}')
```

5.1.8 Sobre o kernel (quando usar e alternativas)

Serve para: fechamento morfológico → preencher buracos pequenos e suavizar cantinhos na máscara

Tamanho:

3×3: efeito mais suave; bom se sua máscara já é precisa e só tem ruído mínimo.

5×5: preenche melhor buracos pequenos; pode engrossar bordas um pouco.

Formato:

MORPH_RECT: quadrado (padrão).

MORPH_ELLIPSE: efeito mais arredondado, às vezes mais natural para anatomia.

MORPH_CROSS: altera menos.

Se suas máscaras já são binárias/limpas, pode setar:

USE_MORPHOLOGY = False

Se notar contornos “picotados” ou pequenos vazios dentro do osso, ligue e teste:

USE_MORPHOLOGY = True

KERNEL_SIZE = 3 # experimente 3 e 5

5.2 Método 2

Esse método preserva o exame completo em 3D e garante que as máscaras sejam registradas e alinhadas ao volume original, usando SimpleITK como biblioteca central, junto com NumPy e Pillow (PIL) para salvar patches. As vantagens são a robustez e a precisão: permite escolher automaticamente a fatia mais representativa (maior área de máscara), corrigir desalinhamentos via resample, e manter o fluxo compatível com dados médicos em formato padrão. A desvantagem é a complexidade maior — exige lidar com volumes, metadados de espaçamento e transformações — e também maior custo computacional e armazenamento, já que processar arquivos NRRD é mais pesado que manipular PNGs.

5.2.1 Visão geral

Entrada: exames 3D em .nrrd e máscaras 3D em .nrrd (idealmente *.seg.nrrd).

Passos-chave:

Inventariar exames por classe e indexar máscaras por ID.

Alinhar cada máscara ao exame (mesma grade) com SimpleITK.

Escolher a fatia com maior área de máscara (fatia representativa).

Gerar patch 150×150 centralizado no centróide da máscara (com padding se necessário).

(Opcional) Salvar overlay para validar o encaixe (imagem + máscara em vermelho).

Saída:

SAIDA/<CLASSE>/*__150x150.png

SAIDA/<CLASSE>/_overlay/*__150x150_overlay.png (se SAVE_OVERLAY=True)

Bibliotecas centrais: SimpleITK (leitura NRRD + resample), NumPy (operações), Pillow (salvar PNG).

5.2.2 Configuração

Defina caminhos, extensões, parâmetros de patch e flags de saída.

```
# ================== CONFIG ==================
import os, re
from pathlib import Path
import numpy as np
from PIL import Image

ROOT_EXAMS = "/content/drive/MyDrive/.../2-Rescaled_256_NRRD"
ROOT_MASKS = "/content/drive/MyDrive/.../3-Segmentation_NRRD"
OUTPUT_ROOT  = "/content/drive/MyDrive/.../SAIDA"

CLASSES      = ["BENIGN", "MALIGNANT"]
EXT_EXAM     = ".nrrd"
EXT_MASK_PREF= ".seg.nrrd"   # preferir máscaras rotuladas

PATCH_HW   = (150, 150)      # (H, W)
BBOX_MARGIN= 0               # margem extra ao redor do centróide (px)
SAVE_OVERLAY = True          # salva figura com overlay p/ validação
```

Dicas rápidas:

Se mudar a organização de pastas ou nomes de classes, ajuste ROOT_EXAMS/CLASSES.

BBOX_MARGIN > 0 adiciona contexto ao redor da lesão.

5.2.3 Dependência (SimpleITK)

Leitura/registro de NRRD e resample de máscara para a grade do exame.

```
try:

    import SimpleITK as sitk

except Exception:

    import sys, subprocess

    subprocess.check_call([sys.executable, "-m", "pip", "install", "-q",
"SimpleITK"])

    import SimpleITK as sitk

os.makedirs(OUTPUT_ROOT, exist_ok=True)
```

obs: neste caso, subprocess funciona como uma alternativa ao comando pip apresentado anteriormente, permitindo instalar bibliotecas diretamente do bloco de código que está rodando (caso ainda não tenha sido instalada), sem precisar rodar outros comandos/blocos ou interromper a execução atual.

5.2.4 Utilitários (funções usadas no pipeline)

5.2.4.1 Helpers gerais

```
# ================= UTILS =================

_start_digits = re.compile(r"^0*(\d+)")

_any_digits   = re.compile(r"(\d+)")


def ensure_dir(p): os.makedirs(p, exist_ok=True)


def norm_id_from_name(fname: str, ext_hint: str) -> str:

    """

    Normaliza ID numérico do nome do arquivo (tolera zeros à esquerda).

    Ex.: 'P007.seg.nrrd' -> '7'; 'Exam_12.nrrd' -> '12'.

    """

    stem = fname[:-len(ext_hint)] if fname.lower().endswith(ext_hint.lower()) else
os.path.splitext(fname)[0]

    m = _start_digits.match(stem)

    if m: return m.group(1)

    m2 = _any_digits.search(stem)

    return m2.group(1) if m2 else stem
```

5.2.4.2 Conversão p/ 8 bits (apenas p/ salvar PNG/overlay)

```
def sitk_to_u8(arr: np.ndarray) -> np.ndarray:

    """

    Converte slice 2D para uint8 via min-max local (0..255).

    Usado apenas para salvar/visualizar (não altera o exame original).

    """

    arr = arr.astype(np.float32)

    amin, amax = float(arr.min()), float(arr.max())

    if amax > amin:

        arr = (arr - amin) / (amax - amin) * 255.0
```

```
    else:
        arr = np.zeros_like(arr)
    return arr.astype(np.uint8)
```

5.2.4.3 Resample da máscara para geometria do exame

```
def resample_mask_to_exam(mask_img: sitk.Image, exam_img: sitk.Image) -> sitk.Image:
    """
    Alinha máscara à grade do exame (tamanho, origem, direção, espaçamento).
    """
    res = sitk.ResampleImageFilter()
    res.SetReferenceImage(exam_img)
    res.SetInterpolator(sitk.sitkNearestNeighbor)
    res.SetTransform(sitk.Transform(3, sitk.sitkIdentity))
    res.SetDefaultPixelValue(0)
    return res.Execute(mask_img)
```

5.2.4.4 Escolher fatia mais representativa

```
def best_slice_axis_index(mask_np3d: np.ndarray):
    """
    Seleciona fatia com maior área de máscara ao longo do eixo z (0).
    Retorna (eixo=0, idx_z).
    """
    areas = mask_np3d.reshape(mask_np3d.shape[0], -1).sum(axis=1)
    idx_z = int(np.argmax(areas))
    return 0, idx_z
```

Note que este bloco de código seleciona a fatia de maior área, tendo em vista que os arquivos .NRRD tem vários cortes. Para evitar possíveis problemas com a baixa quantidade de arquivos, você pode substituir essa função por uma que considere cada corte (do exame e da máscara) como uma imagem independente, o que aumentará sua base, pois cada máscara possui mais de um nível.

Se optar por essa outra abordagem, será excencial parear corretamente cada máscara com sua imagem original, tendo em vista que, em todos os casos que eu olhei, o exame tem mais cortes do que suas máscaras.

Exemplo: Para o paciente 1, seu exame possui 14 cortes; enquanto sua respectiva máscara para a vértebra L1, apenas 8

5.2.4.5 Overlay (validação visual)

```
def overlay_rgb(gray_u8: np.ndarray, mask2d: np.ndarray, alpha=0.35):
    """
    Aplica máscara (vermelho) sobre grayscale (uint8). Retorna RGB (H,W,3).
    """
    h, w = gray_u8.shape
    rgb = np.stack([gray_u8]*3, axis=-1).astype(np.float32)
    red = np.zeros_like(rgb); red[...,0] = 255
    rgb[mask2d] = (1-alpha)*rgb[mask2d] + alpha*red[mask2d]
    return np.clip(rgb,0,255).astype(np.uint8)
```

5.2.4.6 Janela centrada no centróide + recorte com padding

```
def centered_window_from_centroid(mask2d: np.ndarray, H: int, W: int, ph: int, pw:
int, margin: int=0):
    """
    Calcula a janela (ph,pw) centrada no centróide da máscara (ou no centro da
imagem se vazia).
    """
    ys, xs = np.where(mask2d)
    if ys.size == 0:
        yc, xc = H//2, W//2
    else:
        yc = int(round(ys.mean()))
        xc = int(round(xs.mean()))
    ph = ph + 2*margin
    pw = pw + 2*margin
    y0 = yc - ph//2
```

```
    x0 = xc - pw//2
    y1 = y0 + ph
    x1 = x0 + pw
    return y0, y1, x0, x1


def crop_with_padding(img: np.ndarray, y0: int, y1: int, x0: int, x1: int):
    """
    Recorta [y0:y1, x0:x1]; se sair da borda, preenche com zeros.
    Funciona para 2D (H,W) e 3D (H,W,C).
    """
    H, W = img.shape[:2]
    ph, pw = y1 - y0, x1 - x0
    if img.ndim == 2:
        out = np.zeros((ph, pw), dtype=img.dtype)
    else:
        C = img.shape[2]
        out = np.zeros((ph, pw, C), dtype=img.dtype)

    src_y0 = max(0, y0); src_y1 = min(H, y1)
    src_x0 = max(0, x0); src_x1 = min(W, x1)

    dst_y0 = src_y0 - y0
    dst_x0 = src_x0 - x0
    dst_y1 = dst_y0 + (src_y1 - src_y0)
    dst_x1 = dst_x0 + (src_x1 - src_x0)

    if src_y0 < src_y1 and src_x0 < src_x1:
        out[dst_y0:dst_y1, dst_x0:dst_x1] = img[src_y0:src_y1, src_x0:src_x1]
    return out
```

5.2.4.7 Detectar classe pelo caminho (opcional)

```
def normalize_class_from_path(path: str):
    parts = {p.lower() for p in Path(path).parts}
    if "benign" in parts: return "BENIGN"
    if "malignant" in parts: return "MALIGNANT"
    return None
```

5.2.5 Inventário (exames por classe + index de máscaras)

Coletamos exames *.nrrd em ROOT_EXAMS/<CLASSE>/.

Varremos recursivamente ROOT_MASKS por todas as máscaras *.nrrd.

Construímos um índice por ID, priorizando *.seg.nrrd.

```
# ================= INVENTÁRIO =================
# Exames por classe
exams_by_class = {cls: [] for cls in CLASSES}
for cls in CLASSES:
    p = os.path.join(ROOT_EXAMS, cls)
    if os.path.isdir(p):
        exams_by_class[cls] = sorted([os.path.join(p, f)
                                      for f in os.listdir(p) if
f.endswith(EXT_EXAM)])

# Máscaras (scan recursivo)
all_masks = []
for root, _, files in os.walk(ROOT_MASKS):
    for f in files:
        if f.endswith(".nrrd"):
            all_masks.append(os.path.join(root, f))

# Index por ID normalizado (preferindo .seg.nrrd)
masks_index = {}
for mp in all_masks:
    name = Path(mp).name
    if name.lower().endswith(EXT_MASK_PREF.lower()):
        mid = norm_id_from_name(name, EXT_MASK_PREF); prio = 0
    else:
        mid = norm_id_from_name(name, ".nrrd"); prio = 1
    masks_index.setdefault(mid, []).append((prio, mp))
for k in list(masks_index.keys()):
    masks_index[k] = [p for _, p in sorted(masks_index[k], key=lambda x: x[0])]
```

Atenção:

Se seus IDs não forem puramente numéricos, adapte norm_id_from_name.

Se houver várias máscaras por exame, o índice manterá todas; processamos cada uma.

5.2.6 Processamento (loop principal)

Para cada exame: casar imagens e máscaras pelo ID, alinhar, escolher fatia, gerar patches e salvar.

```
# ================== PROCESSAMENTO ==================
totals = {"processed":0, "saved":0, "no_mask":0, "empty_mask":0}
PH, PW = PATCH_HW

for cls in CLASSES:
    out_dir = os.path.join(OUTPUT_ROOT, cls)
    out_overlay = os.path.join(out_dir, "_overlay")
    ensure_dir(out_dir)
    if SAVE_OVERLAY: ensure_dir(out_overlay)

    processed=saved=no_mask=empty_mask=0

    for exam_path in exams_by_class.get(cls, []):
        exam_name = Path(exam_path).name
        exam_id   = norm_id_from_name(exam_name, EXT_EXAM)

        cand = masks_index.get(exam_id, [])
        same_class = [m for m in cand if normalize_class_from_path(m)==cls]
        picks = same_class if same_class else cand
        if not picks:
            no_mask += 1
            continue

        # Carrega EXAME (3D esperado: (z,y,x))
        try:
            exam_img = sitk.ReadImage(exam_path)
            exam_np  = sitk.GetArrayFromImage(exam_img)
        except Exception as e:
            print(f"[{cls}] ERRO lendo exame {exam_path}: {e}")
            continue

        for mask_path in picks:
            try:
                mask_img = sitk.ReadImage(mask_path)
            except Exception as e:
                print(f"[{cls}] ERRO lendo máscara {mask_path}: {e}")
                continue

            # Alinha máscara à geometria do exame
            try:
                mask_res = resample_mask_to_exam(mask_img, exam_img)
            except Exception as e:
                print(f"[{cls}] ERRO resamplando máscara:\n  EXAME: {exam_path}\n
MASK:  {mask_path}\n  -> {e}")
                continue

            mask_np = sitk.GetArrayFromImage(mask_res) > 0  # (z,y,x) bool
            if mask_np.sum() == 0:
                empty_mask += 1
                continue

            # Seleciona fatia de maior área da máscara
            _, idx = best_slice_axis_index(mask_np)  # eixo 0 (z)
            mask2d = mask_np[idx, :, :]
            exam2d = exam_np[idx, :, :]

            # Converte p/ 8 bits e aplica máscara (fora=0)
            exam2d_u8 = sitk_to_u8(exam2d)
            masked = exam2d_u8.copy()
            masked[~mask2d] = 0

            # Janela 150x150 centrada no centróide (com BBOX_MARGIN opcional)
```

```
            H, W = masked.shape
            y0, y1, x0, x1 = centered_window_from_centroid(mask2d, H, W, PH, PW,
margin=BBOX_MARGIN)
            patch = crop_with_padding(masked, y0, y1, x0, x1)
            # garante 150x150 (sanidade)
            if patch.shape != (PH, PW):
                pad = np.zeros((PH, PW), dtype=patch.dtype)
                h, w = patch.shape
                yb = max(0, (PH - h)//2); xb = max(0, (PW - w)//2)
                ye = min(PH, yb + h);      xe = min(PW, xb + w)
                pad[yb:ye, xb:xe] = patch[:ye-yb, :xe-xb]
                patch = pad

            # Overlay (RGB) com a MESMA janela
            if SAVE_OVERLAY:
                ov_full = overlay_rgb(exam2d_u8, mask2d)  # (H,W,3)
                ov_patch = crop_with_padding(ov_full, y0, y1, x0, x1)
                if ov_patch.shape[:2] != (PH, PW):
                    pad = np.zeros((PH, PW, 3), dtype=ov_patch.dtype)
                    h, w = ov_patch.shape[:2]
                    yb = max(0, (PH - h)//2); xb = max(0, (PW - w)//2)
                    ye = min(PH, yb + h);      xe = min(PW, xb + w)
                    pad[yb:ye, xb:xe] = ov_patch[:ye-yb, :xe-xb]
                    ov_patch = pad

            # Salva
            mask_stem = Path(mask_path).stem
            out_png   = os.path.join(out_dir,
f"{exam_id}__slice{idx:03d}__{mask_stem}__150x150.png")
            Image.fromarray(patch).save(out_png)
            if SAVE_OVERLAY:
                out_ov = os.path.join(out_overlay,
f"{exam_id}__slice{idx:03d}__{mask_stem}__150x150_overlay.png")
                Image.fromarray(ov_patch).save(out_ov)

            saved += 1
            processed += 1

    print(f"[{cls}] Processados: {processed} | Salvos: {saved} | Sem máscara:
{no_mask} | Máscara vazia: {empty_mask}")
    totals["processed"] += processed
    totals["saved"]     += saved
    totals["no_mask"]   += no_mask
    totals["empty_mask"]+= empty_mask

print("\n==== RESUMO GERAL ====")
print(f"Processados: {totals['processed']}")
print(f"Salvos: {totals['saved']}")
print(f"Sem máscara: {totals['no_mask']}")
print(f"Máscara vazia: {totals['empty_mask']}")
print(f"Pasta de saída: {OUTPUT_ROOT}")
print("Verifique também as imagens em SAIDA/<CLASSE>/_overlay para validar o
encaixe.")
```

5.2.7 Pontos de atenção e diagnóstico rápido

- Alinhamento (overlay não casa): verifique resample_mask_to_exam e IDs; confirme se a máscara realmente corresponde ao exame.
- Máscara “sumiu” após resample: checar FOV/spacing/direction; pode ser máscara de outro volume.
- Comparabilidade de intensidades entre pacientes: troque sitk_to_u8 (min–max local) por janela fixa adequada ao protocolo.
- Múltiplas regiões desconexas na máscara: para isolar cada componente, use connected components (opcional)..

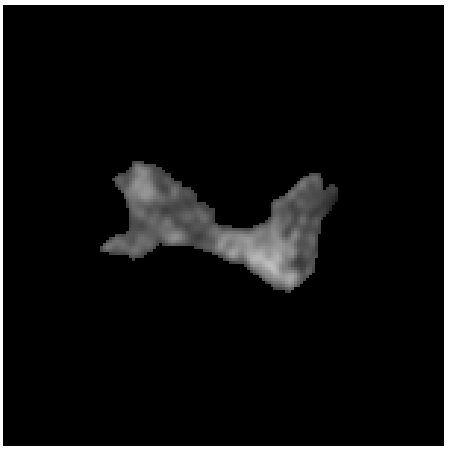
5.3 Extraindo uma imagem NRRD layer a layer (opcional)

```
# ===== CONFIGURAÇÕES =====
input_nrrd = "meu_arquivo.nrrd"   # caminho do arquivo .nrrd
output_dir = "slices_png"         # pasta de saída

# cria pasta de saída, se não existir
os.makedirs(output_dir, exist_ok=True)

# ===== LEITURA DO NRRD =====
volume, header = nrrd.read(input_nrrd)  # volume 3D (array numpy)

print("Dimensões do volume:", volume.shape)  # Ex: (altura, largura, cortes)

# ===== SALVAR TODOS OS SLICES =====
for i in range(volume.shape[2]):  # último eixo são os cortes
    slice_img = volume[:, :, i]   # pega corte i

    # normaliza valores para 0-255 (8 bits)
    min_val, max_val = slice_img.min(), slice_img.max()
    if max_val > min_val:  # evita divisão por zero
        slice_norm = (255 * (slice_img - min_val) /
np.ptp(slice_img)).astype(np.uint8)
    else:
        slice_norm = np.zeros_like(slice_img, dtype=np.uint8)

    # cria imagem e salva
    img = Image.fromarray(slice_norm)
    img.save(os.path.join(output_dir, f"slice_{i:03d}.png"))

print(f"Total de cortes salvos: {volume.shape[2]}")
```

Note que, a extração do conteúdo dos arquivos dessa maneira não é o único passo. Ao trabalhar com cada corte individualmente, você precisará garantir que o pareamento layer a layer está sendo feito corretamente.
Você pode utilizar a função auxiliar de Overlay do Método 2 como ajuda para visualizar isso e fazer o pareamento correto para cada caso.

5.4 Patches

Ao final deste tópico, você deverá ter pastas contendo imagens semelhantes a esta:



Todas as imagens resultantes devem ter dimensão uniforme, apresentando a vértebra posicionada no centro e mantendo a textura da área de interesse.

6. Divisão dos Datasets de Treino Teste e Validação

6.1 Objetivo

- Separar a base em train/validation/test mantendo todas as imagens/patches do mesmo paciente no mesmo split (evita data leakage).

- Preservar o balanceamento por classe o melhor possível, fazendo a divisão dentro de cada classe (BENIGN/MALIGNANT) e depois unindo.

- Garantir reprodutibilidade com semente fixa (SEED).

- Organização de pastas
    - Origem (SOURCE_ROOT): contém BENIGN/ e MALIGNANT/ com as imagens (ignorar os overlays).
    - Destino (SPLIT_ROOT): será criado com a estrutura:
        - train/benign, train/malignant
        - validation/benign, validation/malignant
        - test/benign, test/malignant

- Renomeação de classe: mapeiar BENIGN → benign e MALIGNANT → malignant para padronizar nomes no destino.

6.2 Identificação de paciente

O patient_id deve ser extraído do nome do arquivo (ex.: tudo antes de __), com fallback para a primeira sequência numérica.

Regra de ouro: o que define o split é o paciente, não o arquivo. Ou seja, todos os arquivos/patches daquele paciente seguem para o mesmo conjunto.

6.3 Estratégia de divisão

- Realiza a divisão por classe, a partir do conjunto de pacientes únicos daquela classe.

- Usa RATIOS (ex.: 80/10/10) e SEED para embaralhar e particionar (reprodutível)
    - Se você for usar a estratégia de validação cruzada no modelo, aqui você não definirá um diretório de validação

- Depois combina os splits das classes, checando que nenhum paciente aparece em mais de um conjunto.

6.4 O que é ignorado

Overlays (arquivos e a pasta _overlay) não entram nos splits (caso você os tenha gerado).

Arquivos duplicados no destino devem ser pulados ou sobrescritos conforme o OVERWRITE.

6.5 Validação importante

Consistência de classe por paciente: alerta se algum paciente aparece em mais de uma classe (isso indica problema na base).

Exclusividade por split: assertiva para garantir que nenhum paciente esteja em train e validation, por exemplo.
Isso também deve ser verdade se você fez a extração layer a layer dos arquivos anteriormente. Um paciente deve ter todas as suas imagens em um mesmo diretório.

6.6 Dicas rápidas (boas práticas)

- Pathlib + shutil: ótimos para percorrer diretórios e copiar arquivos com segurança e legibilidade.
- random.Random(SEED): assegura a mesma partição a cada execução (todos do grupo terá o dataset dividido de forma igual)
- pandas (opcional): gere um manifest.csv com colunas (filepath, split, class, patient_id) — facilita auditoria e debug.
- tqdm (opcional): barra de progresso na cópia.
- logging (opcional): logs mais limpos que print.
- scikit-learn (opcional):
    - GroupShuffleSplit ou StratifiedGroupKFold quando precisar estratificar por classe com grupos (= pacientes) — útil em bases maiores e com imbalance.
- Integridade dos nomes: padronize extensões (.png vs .PNG) e verifique a extração de patient_id para evitar colisões (ex.: 007 vs 7).
- Revisão manual: inspecione alguns casos limítrofes (p. ex., pacientes com poucas imagens) e ajuste RATIOS se o número total de pacientes for pequeno.

7. Modelagem e Classificação

Com os dicionários de treino, teste e validação devidamente montados, contendo as regiões de interesse para classificação, você está pronto para os próximos estágios do projeto de classificação de imagens médicas. Esta seção abordará a construção e/ou escolha de modelos adequados para a classificação das vértebras quanto ao tipo de fratura presente (benigna ou maligna).

7.1. Construção e Treinamento de Classificadores

Esta é a etapa central onde você irá construir e/ou treinar um modelo para classificar as imagens. A escolha do modelo é flexível, e é altamente recomendado experimentar diferentes arquiteturas e abordagens. Para problemas de classificação de imagens, Redes Neurais Convolucionais (CNNs) são uma escolha comum e poderosa, e a biblioteca tensorflow.keras é uma excelente ferramenta para construí-las.

Ao construir seu modelo, considere os seguintes pontos:

- Arquitetura do Modelo: Experimente diferentes configurações de camadas, como camadas convolucionais (Conv2D), camadas de pooling (MaxPooling2D), camadas densas (Dense) e camadas de achatamento (Flatten). A profundidade e largura da rede podem impactar significativamente o desempenho.

- Função de Ativação: Escolha funções de ativação apropriadas para as camadas, como relu para camadas ocultas e sigmoid (para classificação binária) ou softmax (para classificação multiclasse) para a camada de saída.

- Otimizador e Função de Perda: Selecione um otimizador (ex: Adam, SGD) e uma função de perda (ex: binary_crossentropy para classificação binária) que sejam adequados para o seu problema.

- Regularização: Para evitar o overfitting, considere a adição de técnicas de regularização:
    - Dropout: Adicione camadas de Dropout entre as camadas do modelo. O dropout desativa aleatoriamente uma porcentagem de neurônios durante o treinamento, forçando a rede a aprender representações mais robustas.
    - Regularização L1/L2: Adicione regularização L1 ou L2 aos pesos das camadas densas.
    - Parada Antecipada (Early Stopping): Implemente um callback de parada antecipada durante o treinamento. Isso monitora uma métrica de desempenho (por exemplo, perda de validação) e interrompe o treinamento quando a métrica para de melhorar, evitando o overfitting e economizando tempo computacional.
    - Experimentação de Hiperparâmetros: Os hiperparâmetros (taxa de aprendizado, tamanho do lote, número de épocas, número de neurônios nas camadas, etc.) têm um grande impacto no desempenho do modelo. Recomenda-se testar diferentes combinações de hiperparâmetros para encontrar a configuração ideal. Técnicas como busca em grade (Grid Search) ou busca aleatória (Random Search) podem ser úteis.

7.1.1. Construção de Modelos do Zero

Para construir um modelo de Rede Neural Convolucional (CNN) do zero, você pode utilizar a API Sequencial ou a API Funcional do Keras. A ideia é empilhar camadas que aprendam a extrair características das imagens e, em seguida, classificar essas características. Comece com uma arquitetura simples e, gradualmente, adicione complexidade conforme a necessidade. Explore diferentes números de camadas convolucionais, filtros, tamanhos de kernel e configurações de pooling. A experimentação é fundamental para encontrar a melhor arquitetura para o seu problema específico.

7.2. Utilização de Modelos Pré-treinados e Transfer Learning

Uma abordagem poderosa para problemas de classificação de imagens, especialmente com conjuntos de dados menores, é o uso de modelos pré-treinados e a técnica de Transfer Learning. Modelos pré-treinados são redes neurais convolucionais que já foram treinadas em grandes conjuntos de dados (como o ImageNet) para tarefas de classificação de imagens genéricas. Eles aprenderam a extrair características visuais de baixo nível (bordas, texturas) e de alto nível (formas, objetos) que podem ser úteis para o seu problema específico.

Alguns modelos pré-treinados populares e eficazes disponíveis no tensorflow.keras.applications incluem:

•VGG16/VGG19: Modelos mais antigos, mas ainda eficazes e fáceis de entender.

•ResNet50/ResNet101/ResNet152: Redes mais profundas que utilizam conexões residuais para mitigar o problema do gradiente evanescente.

•InceptionV3/InceptionResNetV2: Modelos que utilizam módulos Inception para capturar características em múltiplas escalas.

•MobileNetV2: Projetado para ser eficiente em dispositivos móveis, mas ainda com bom desempenho.

Existem diversas outras opções de modelos a serem utilizados, nem todos no keras.applications, recomendo que busquem na internet modelos que achem interessantes.

7.2.1 Como Importar e Aplicar Transfer Learning:

Para utilizar um modelo pré-treinado, você pode importá-lo diretamente utilizando `from keras.applications import SeuModelo`. A ideia é carregar o modelo sem as camadas finais de classificação (definindo include_top=False) e, em seguida, adicionar suas próprias camadas densas para a tarefa de classificação específica. Você pode congelar as camadas convolucionais do modelo pré-treinado (model.trainable = False) para que seus pesos não sejam atualizados durante o treinamento, aproveitando as características já aprendidas. Alternativamente, você pode descongelar algumas das últimas camadas convolucionais para um fine-tuning, permitindo que o modelo se adapte melhor aos seus dados.

```
from keras.applications import SEUMODELO
from keras import layers
from keras import models
from keras import optimizers

base_model = SEUMODELO(weights='imagenet', include_top=False, input_shape=(150, 150,
3))

# Congelar as camadas originais
base_model.trainable = False

# Adicionar novas camadas
model = models.Sequential()
model.add(base_model)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

7.2.1.1 Configuração de Hiperparâmetros para Transfer Learning:

Ao usar transfer learning, a experimentação de hiperparâmetros continua sendo crucial. Considere:

- Taxa de Aprendizado: Geralmente, uma taxa de aprendizado menor é recomendada para o fine-tuning das camadas pré-treinadas, para evitar que os pesos pré-existentes sejam alterados drasticamente.

- Número de Épocas: Pode ser necessário um número menor de épocas se as camadas pré-treinadas estiverem congeladas, pois apenas as novas camadas estão sendo treinadas. Para fine-tuning, mais épocas podem ser necessárias.

- Tamanho do Lote: Experimente diferentes tamanhos de lote para otimizar o uso da memória e a estabilidade do treinamento.

- Dropout e Regularização: Continue aplicando dropout e outras técnicas de regularização nas novas camadas densas que você adicionar para evitar o overfitting no seu conjunto de dados.

7.3 Avaliação do Modelo

Após o treinamento, é crucial avaliar o desempenho do seu modelo no conjunto de testes (dados não vistos durante o treinamento). Utilize métricas apropriadas para problemas de classificação, como:

- Acurácia: Proporção de previsões corretas.

- Precisão (Precision): Proporção de verdadeiros positivos entre todos os positivos previstos.

- Recall (Sensibilidade): Proporção de verdadeiros positivos entre todos os positivos reais.

- Sensibilidade (ou recall, taxa de verdadeiros positivos): Mede a capacidade do teste de identificar corretamente os casos positivos

- Especificidade (taxa de verdadeiros negativos): Mede a capacidade do teste de identificar corretamente os casos negativos.

- F1-score: Média harmônica da precisão e do recall.

- Curva ROC e AUC (Area Under the Curve): A Curva Característica de Operação do Receptor (ROC) e a Área Sob a Curva (AUC) são métricas importantes para avaliar o desempenho de modelos de classificação binária, especialmente em conjuntos de dados desbalanceados.

- Matriz de Confusão: Uma matriz de confusão fornece uma visão detalhada dos acertos e erros do modelo para cada classe.

Nem todas as métricas precisam ser calculadas diretamente no seu algoritmo, você pode importar o cálculo de algumas delas utilizando a biblioteca `sklearn.metrics` (Ex. *Matriz de Confusão* e *AUC*).