CENTRO DE PESQUISA E DESENVOLVIMENTO TECNOLÓGICO EM INFORMÁTICA E ELETROELETRÔNICA DE ILHÉUS

Leonardo dos Santos Vaz
Ivanildo Gomes da Silva

# IMPLEMENTAÇÃO E ANÁLISE DE CLASSIFICAÇÃO COM REDES CONVOLUCIONAIS E O DATASET CUFS

**Relatório Técnico:** Atividade avaliativa referente a unidade 10 da Trilha 3 do curso de Ciência de Dados da residência TIC 36.

Ilhéus

01/12/2024

## 1. RESUMO

O presente relatório técnico descreve a implementação e avaliação de redes convolucionais (CNNs) para a tarefa de classificação de imagens. Utilizando o dataset CUHK Face Sketch Database (CUFS), buscamos classificar imagens faciais de acordo com o sexo biológico (masculino ou feminino). O projeto abrange desde a preparação e anotação dos dados até a criação de um modelo CNN autoral, seguido por análises detalhadas de desempenho com métricas como F1-Score e AUC-ROC. Por fim, refletimos sobre os desafios enfrentados, resultados alcançados e possíveis melhorias no algoritmo CNN aqui desenvolvido.

## 2. INTRODUÇÃO

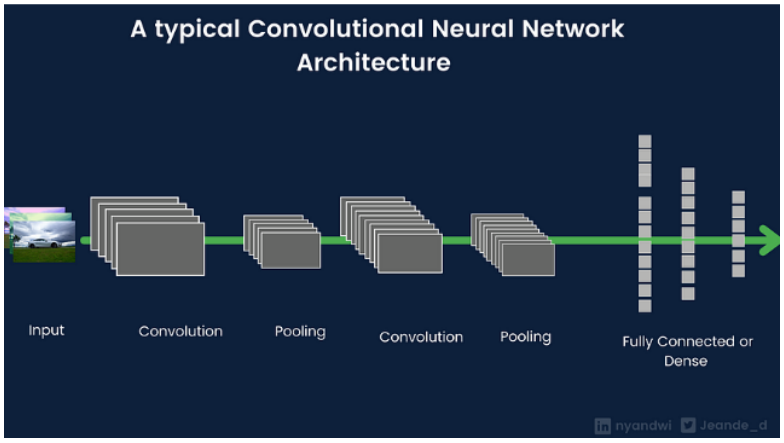
A classificação de imagens é um dos desafios mais relevantes na área de ciência de dados, com aplicações em reconhecimento facial, diagnóstico médico e muito mais. Redes neurais convolucionais (CNNs) têm se destacado por sua capacidade de extrair características relevantes diretamente das imagens, sem a necessidade de engenharia manual.

Uma configuração comum de uma rede neural convolucional (CNN) consiste em uma sequência de camadas convolucionais intercaladas com camadas de pooling, culminando em uma ou mais camadas totalmente interligadas (inteiramente ligadas). Esta organização possibilita que a rede capture atributos de baixo nível, como bordas e texturas, nas camadas iniciais, e, progressivamente, elabore representações mais abstratas e complexas nas camadas mais profundas.

**Funcionamento:**

1. **Camadas convolucionais:** Aplicam filtros (kernels) sobre a imagem de entrada, extraindo características locais. À medida que avançamos pelas camadas, o número de filtros aumenta, capturando características mais complexas.
2. **Camadas de pooling:** Reduzem a dimensionalidade da representação, diminuindo o número de parâmetros e controlando o overfitting.
3. **Camada totalmente conectada:** Transforma a saída da parte convolucional em um vetor unidimensional e a conecta a uma camada de classificação (softmax) ou regressão.

**Figura 1**




**Fonte:** Nyandwi, 2021.

Este projeto tem como objetivo explorar a aplicação de CNNs para classificar imagens faciais no dataset CUFS, contribuindo para o aprendizado prático dos conceitos fundamentais de redes convolucionais.

As camadas convolucionais serão responsáveis por extrair características relevantes das imagens, como bordas, texturas e padrões faciais. Usaremos filtros de tamanhos padrão, como (3x3) ou (5x5), que são amplamente utilizados por serem eficazes na captura de detalhes locais. O número de filtros aumentará progressivamente nas camadas mais profundas da rede, começando com 32 filtros na primeira camada e podendo atingir 128 ou mais nas camadas posteriores, permitindo que a rede aprenda características mais abstratas e de alto nível.

As camadas de pooling, como MaxPooling, serão adicionadas após cada bloco convolucional para reduzir a dimensionalidade espacial das imagens. Essa redução ajudará a minimizar o custo computacional, além de proporcionar invariância a pequenas translações nas imagens.

No trabalho, utilizamos técnicas como **data augmentation** (rotação, translação, zoom, shear e flip horizontal) para aumentar a diversidade do conjunto de treino e reduzir o overfitting. A **regularização L2** foi aplicada nas camadas densas com fator 0.001 para penalizar pesos excessivos. A arquitetura CNN foi projetada com convoluções progressivas (64, 128, 256 filtros) e MaxPooling para extração hierárquica de características. Além disso, aplicamos **Dropout** (0.4) nas camadas densas para maior generalização, e funções de ativação **ReLU** e **sigmoidal**, ajustando o modelo para a

classificação binária. Essas estratégias visaram melhorar a performance e mitigar o overfitting.

Devido a problemas relacionado  as características do dataset muitas microtarefas pode ter sido omitidas, umas vez que foram necessários muitos ajustes para obtenção de um modelo de classificação minimamente funcional.

## 3. METODOLOGIA

Neste tópico, descreveremos de maneira detalhada os procedimentos utilizados para a realização deste trabalho. Em linhas gerais, Serão feita a classificação das imagens obtidas no *dataset* fornecido para realização desta tarefa e obtivo por meio da plataforma *Kaggle*. Este dataset Inclui 188 faces do banco de dados de estudantes da Universidade Chinesa de Hong Kong (CUHK), 123 faces do banco de dados AR e 295 faces do banco de dados XM2VTS. Há 606 faces no total. As etapas do percurso metodológico deste estudo está organizado da seguinte forma:

- Preparação dos dados
- Implementação da arquitetura CNN
- Treinamento e avaliação

Deste modo, esperamos obter êxito na realização da tarefa solicitada para a unidade 10 da trilha 3 – Ciência de dados. A seguir, vamos apresentar os detalhes de cada etapa metodológica.

### 3.1. Prepatação dos Dados

O dataset utilizado foi o CUHK Face Sketch Database (CUFS), composto por 188 imagens na pasta *photos*. O ponto de partida foi realizado por meio do código fornecido na atividade. Este programa utiliza a API do *Kaggle*, e extrai os arquivos, e lista os arquivos presentes na pasta *photos*. A lista gerada pode ser usada posteriormente para processamento, como atribuição de rótulos (masculino ou feminino) ou carregamento das imagens para treinamento de um modelo de classificação.

**Figura 2**

```python
import os

!pip install kaggle
!kaggle datasets download -d arbazkhan971/cuhk-face-sketch-database-cufs --force
!unzip -oq "cuhk-face-sketch-database-cufs.zip"

def list_files_in_folder(folder_path):
  """Lists all files in a given folder."""
  try:
    file_list = os.listdir(folder_path)
    return file_list
  except FileNotFoundError:
    print(f"Error: Folder not found at {folder_path}")
    return []

photos_folder = "photos"
files_in_photos = list_files_in_folder(photos_folder)
```

**Figura 1:** Código (ponto de partida) fornecido na atividade. Este código utiliza a API do *Kaggle* para extrair os dados do dataset fornecido para esta atividade.

Após baixar e extrair o dataset, a próxima tarefa é preparar os dados para o modelo de classificação. Essa etapa envolve vários processos importantes, organizados da seguinte maneira:

- **Anotação dos Dados:**
  - Listagem dos nomes dos arquivos na pasta photos.
  - Visualização as imagens para entender as características principais de cada classe (masculino e feminino).
  - Criação dos rótulos (Labels) devemos atribuir 0 para masculino e 1 para feminino.

**Figura 2**

```python
import os
import pandas as pd
import shutil
from google.colab import drive

# Montar o Google Drive
drive.mount('/content/drive')

# Caminhos principais
project_folder = "/content/drive/My Drive/projeto_unidade10"
dataset_folder = os.path.join(project_folder, "dataset")
photos_folder = os.path.join(dataset_folder, "photos")
annotations_path = os.path.join(project_folder, "annotations.csv")
train_folder = os.path.join(dataset_folder, "train")
val_folder = os.path.join(dataset_folder, "val")
test_folder = os.path.join(dataset_folder, "test")

# Criar a estrutura de pastas
os.makedirs(os.path.join(train_folder, "0"), exist_ok=True)
os.makedirs(os.path.join(train_folder, "1"), exist_ok=True)
os.makedirs(val_folder, exist_ok=True)
os.makedirs(test_folder, exist_ok=True)
```

**Figura 2:** Nesse bloco, os diretório do Google Drive foram organizados para receber os dados obtidos por meio da API da kaggle.

**Figura 3**

```
# Função para exibir barra de progresso e rotular o conjunto de
treinamento
def annotate_train_images(folder, output_csv):
    """
    Rotula imagens no conjunto de treinamento e move os arquivos
para subpastas '0' e '1'.
    """
    files = [f for f in os.listdir(folder) if f.endswith(('.jpg',
'.png'))]
    annotations = []

    # Criar barra de progresso
    progress = widgets.IntProgress(value=0, max=len(files),
description="Progresso:")
    progress_label = widgets.Label(value=f"0/{len(files)} imagens
processadas.")
    display(progress, progress_label)

    # Função para exibir próxima imagem e processar rótulo
    def show_next(index):
        if index >= len(files):
            # Salvar anotações ao final
            with open(output_csv, 'w') as f:
                f.write("filename,label\n")
                f.write("\n".join(annotations))
            print(f"Anotações salvas em {output_csv}.")
            print("Rotulamento concluído!")
            return
```

**Figura 3:** As imagens são exibidas durante o processo de anotação para que o usuário possa observá-las antes de rotulá-las por meios das bibliotecas matplotlib e widget.

**Figura 4**

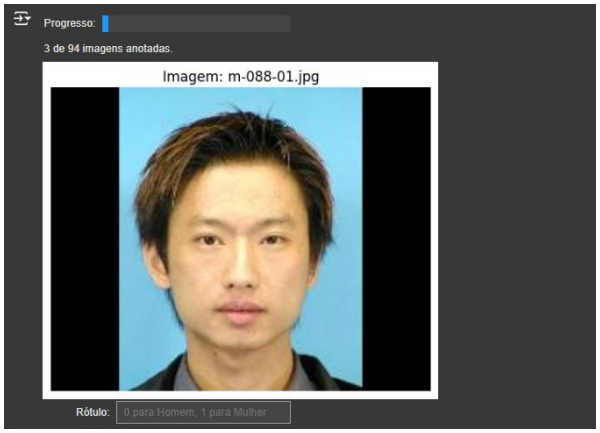



**Figura 4:** interface básica para a inserção dos rótulos com barra de progresso dinâmica, a interface facilita a rotulagem.

Esses foram os blocos relacionados a listagem dos dados, a seguir vamos apresentar os trechos relacionados ao pré-processamento das imagens. Essa tarefa é muito importante, uma vez que, nem todas as imagens a serem analisadas em uma situação real são disponibilizadas em padrões específicos, o pré-processamento pode auxiliar na uniformização das imagens do *dataset*.

- **Pré-processamento:**
    - Redimensionamento das imagens para 250x200 pixels.
    - Normalização dos valores RGB para o intervalo [0, 1].

**Figura 5**

```python
# Função de pré-processamento
def preprocess_images_with_aspect_ratio(input_folder, output_folder, target_size=(250, 200)):
    """
    Redimensiona as imagens para o tamanho especificado sem distorcer a proporção,
    adicionando padding para manter o tamanho final uniforme.
    """
    for file_name in os.listdir(input_folder):
        input_path = os.path.join(input_folder, file_name)
        if not file_name.endswith(('.jpg', '.png')):
            continue
        try:
            # Abrir a imagem
            img = Image.open(input_path)

            # Redimensionar a imagem mantendo a proporção
            img.thumbnail(target_size)  # Ajusta a imagem para caber dentro do tamanho máximo

            # Criar uma nova imagem com o tamanho exato e um fundo preto
            new_img = Image.new("RGB", target_size, (0, 0, 0))  # Fundo preto
            offset = ((target_size[0] - img.size[0]) // 2, (target_size[1] - img.size[1]) // 2)
            new_img.paste(img, offset)  # Centraliza a imagem redimensionada no fundo

            # Salvar a imagem processada
            output_path = os.path.join(output_folder, file_name)
            new_img.save(output_path)

        except Exception as e:
            print(f"Erro ao processar {file_name}: {e}")
```

**Figura 5:** O redimensionamento é realizado na função *preprocess_images_with_aspect_ratio*. Essa função redimensiona a imagem mantendo a proporção original e adiciona *padding* para ajustar ao tamanho final, garantindo que todas as imagens tenham exatamente 250x200 pixels.

**Figura 6**

```python
# Função para normalizar uma imagem
def normalize_image(image_path):
    try:
        # Abrir a imagem
        img = Image.open(image_path).convert("RGB")
        img_array = np.array(img, dtype=np.float32) / 255.0  # Normalizar os valores RGB para [0, 1]

        # Converter de volta para a imagem
        normalized_img = Image.fromarray((img_array * 255).astype(np.uint8))
        normalized_img.save(image_path)  # Salvar a imagem normalizada no mesmo caminho

    except Exception as e:
        print(f"Erro ao processar a imagem {image_path}: {e}")
```

**Figura 6:** A normalização é realizada nesta mesma função, transformando os valores de pixel para o intervalo [0, 1]. Isso é feito após o redimensionamento e rotulação (bibliotecas ***Pil*** e ***Numpy***).

- **Divisão do Dataset:** Seguindo uma proporção de 50%-30%-20% para treino, validação e teste, utilizando a seed 23 para replicabilidade.

**Figura 7**

```
import os
import random
import shutil
from PIL import Image
from google.colab import drive

# Montar o Google Drive
drive.mount('/content/drive')

# Caminhos no Google Drive
project_folder = "/content/drive/My Drive/projeto_unidade10/"
photos_folder = os.path.join(project_folder, "dataset/photos")
dataset_dirs = {
    "train": os.path.join(project_folder, "dataset/train"),
    "val": os.path.join(project_folder, "dataset/val"),
    "test": os.path.join(project_folder, "dataset/test")
}

# Configurar a seed para replicabilidade
random.seed(23)

# Criar as pastas necessárias
for key, path in dataset_dirs.items():
    if os.path.exists(path):
        shutil.rmtree(path)  # Apagar conteúdo antigo
    os.makedirs(path, exist_ok=True)
```

```
    print(f"Divisão concluída:")
    print(f"- Treino: {len(train_files)} imagens.")
    print(f"- Validação: {len(val_files)} imagens.")
    print(f"- Teste: {len(test_files)} imagens.")

    return train_files, val_files, test_files

# Dividir o dataset e preparar as pastas
if os.path.exists(photos_folder):
    train_files, val_files, test_files = split_dataset(
        photos_folder, dataset_dirs, ratios=(0.5, 0.3, 0.2)
    )

    # Processar as imagens em cada conjunto
    for key, folder in dataset_dirs.items():
        print(f"Processando imagens para o conjunto: {key}")
        preprocess_images_with_aspect_ratio(folder, folder)

    print("Imagens organizadas e processadas com sucesso.")
else:
    print(f"Erro: A pasta {photos_folder} não existe. Verifique o caminho.")
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Divisão concluída:
- Treino: 94 imagens.
- Validação: 56 imagens.
- Teste: 38 imagens.
Processando imagens para o conjunto: train
Processando imagens para o conjunto: val
Processando imagens para o conjunto: test
Imagens organizadas e processadas com sucesso.
```

**Figura 7:** Nestes blocos, foi realizada a divisão do *dataset* na proporção 50% para treinamento, 30% para validação e 20% para teste, a função random.seed(23) foi utilizada para garantir replicabilidade.

A configuração da random.seed(23) desempenha um papel importante no código. Ela garante a reprodutibilidade do processo de divisão do dataset.

Por fim, optamos por organizar a estrutura dos arquivos em uma pasta do google drive. Após a realização deste passo-a-passo, partimos para o desenvolvimento da arquitetura CNN.

O cuidado com a fase de anotação é fundamental, pois uma organização inadequada do dataset pode impactar na qualidade da aprendizagem do modelo CNN, deste modo, além de automatizar e tornar interativo o processo, verificamos visualmente as pastas para garantir que código foi implementado corretamente, alguns erros de implementação foram observados na automatização e só foram corrigidos devido a visualização dos dados.

## 3.2. Implementação da arquitetura CNN

Neste tópico, vamos apresentar a arquitetura CNN desenvolvida para este estudo. Nossa estratégia será baseada no desenvolvimento de uma arquitetura projetada especificamente para o problema de classificação binária (homem ou mulher). Na impossibilidade da utilização de *Transfer Learning* com modelos pré-treinados, desenvolvemos uma arquitetura própria a partir de estudo teórico das referências consultadas e projetos anteriores.

Toda a parte de machine learn foi feita em uma única célula, as etapas anteriores relacionadas a anotação dos dados (organização dos dados, listagem, rotulação, processamento de imagens) dividimos em células diferentes para evitar retrabalho, um vez que muitos ajustes foram necessários a fim de evitar problemas futuros durante o treinamento e validação. Na implementação CNN foi possível organizar tudo em uma única célula no colab.

### 3.2.1. Carregamento e preparação dos dados

Inicialmente, o código se concentra em **carregar e preparar os dados** para o treinamento do modelo CNN. Deste modo, os caminhos para as pastas de treino, validação e teste são definidos, organizados dentro do diretório principal no Google Drive (projeto_unidade10). Em seguida, a função *prepare_generators* é usada para configurar os geradores de dados. Esses geradores utilizam o *ImageDataGenerator* para processar as imagens de forma eficiente. Nas figuras detalhamos no código os blocos em que essas etapas foram realizadas

**Figura 8**

```
project_folder = "/content/drive/My Drive/projeto_unidade10/"
dataset_dirs = {
    "train": os.path.join(project_folder, "dataset/train"),
    "val": os.path.join(project_folder, "dataset/val"),
    "test": os.path.join(project_folder, "dataset/test")
}
```

**Figura 8:** Define os caminhos das pastas de treino, validação e teste no diretório principal do projeto.

**Figura 9**

```
def prepare_generators(dataset_dirs, target_size=(250, 200), batch_size=32):
    train_datagen = ImageDataGenerator(
        rescale=1.0/255.0,
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
    )
```

**Figura 9:** Trecho inicial do bloco que configura geradores de dados para treino, validação e teste, aplicando aumentação no conjunto de treino e normalização nos demais.

**Figura 10**

```
def prepare_generators(dataset_dirs, target_size=(250, 200), batch_size=32):
    train_datagen = ImageDataGenerator(
        rescale=1.0/255.0,
        rotation_range=20,
        width_shift_range=0.2,
        height_shift_range=0.2,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True
    )
```

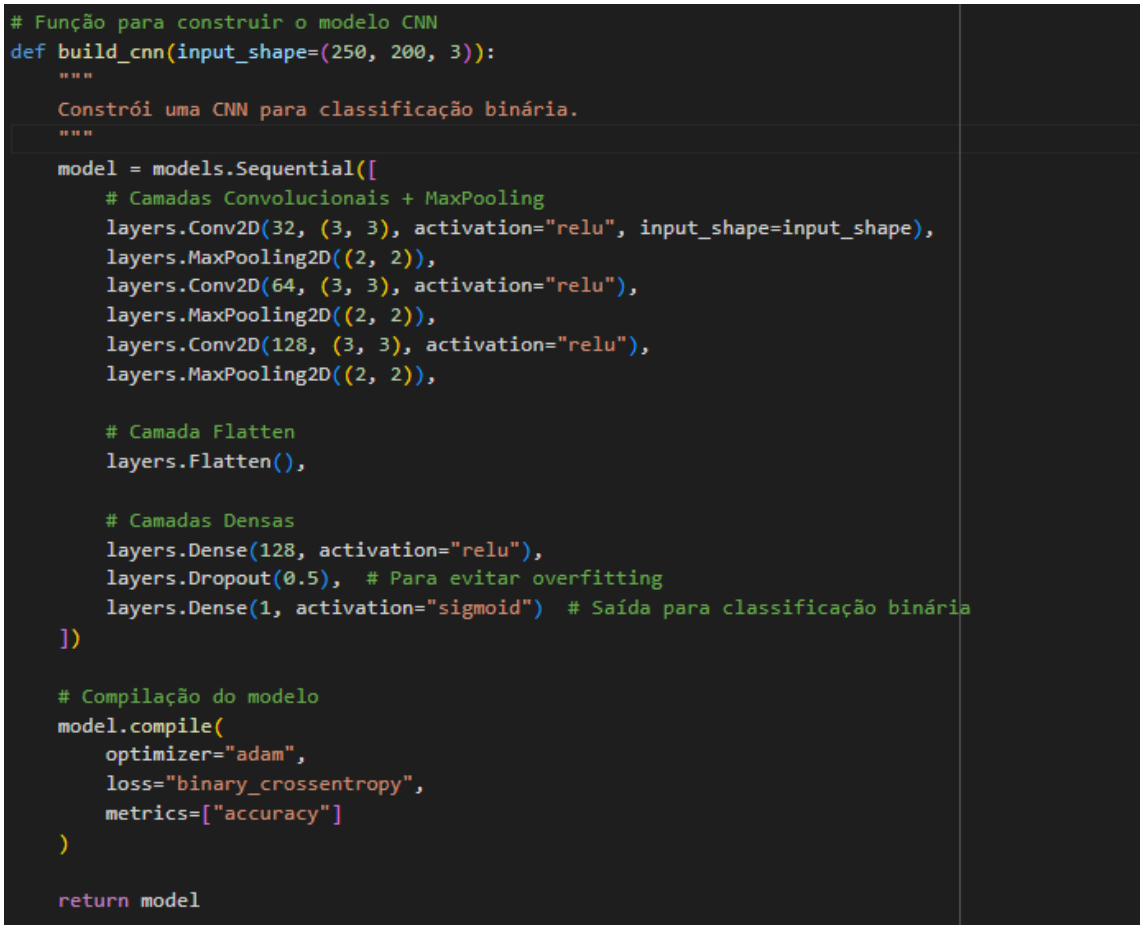
**Figura 10:** Trecho inicial do bloco que configura geradores de dados para treino, validação e teste, aplicando *data augmentation* no conjunto de treino e normalização nos demais.

### 3.2.2. Concepção do modelo CNN

Na etapa de **construção do modelo**, definimos a arquitetura de uma Rede Neural Convolucional (CNN) projetada para a tarefa de classificação

binária. Essa primeira rede foi composta por 3 camadas convolucionais seguidas por 3camadas de pooling, que tem por função, extrair e reduzir características relevantes das imagens. Em seguida, foram implementadas as camadas convolucionais são conectadas a uma camada flatten imprimindo formado unidimensional aos dados, seguida por camadas densas que tem a capacidade de captar padrões complexos. A última camada utiliza a função de ativação sigmoid para produzir uma saída entre 0 e 1, representando a probabilidade de cada classe. A rede é compilada com o otimizador Adam e a função de perda binária cross-entropy para uma aprendizagem eficiente.

**Figura 11**



```python
# Função para construir o modelo CNN
def build_cnn(input_shape=(250, 200, 3)):
    """
    Constrói uma CNN para classificação binária.
    """
    model = models.Sequential([
        # Camadas Convolucionais + MaxPooling
        layers.Conv2D(32, (3, 3), activation="relu", input_shape=input_shape),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(64, (3, 3), activation="relu"),
        layers.MaxPooling2D((2, 2)),
        layers.Conv2D(128, (3, 3), activation="relu"),
        layers.MaxPooling2D((2, 2)),

        # Camada Flatten
        layers.Flatten(),

        # Camadas Densas
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.5),  # Para evitar overfitting
        layers.Dense(1, activation="sigmoid")  # Saída para classificação binária
    ])

    # Compilação do modelo
    model.compile(
        optimizer="adam",
        loss="binary_crossentropy",
        metrics=["accuracy"]
    )

    return model
```

**Figura 11:** Neste bloco, definimos a arquitetura da CNN com camadas convolucionais, pooling, flatten, densas e uma saída para classificação binária. No final vemos o trecho que compila o modelo especificando o otimizador (ADAM), a função de perda e as métricas de avaliação.

Após definida a arquitetura, foi possível realizar o treinamento do modelo conforme será discutido no próximo tópico.

### 3.2.3. Treinamento e avaliação do modelo

Nesta etapa, o modelo é ajustado para captar os padrões nos dados de treinamento por meio de um processo iterativo que atualiza os pesos da rede neural. O treinamento busca promover a minimização da função de perda *(loss)* baseado nos rótulos reais e predições do modelo, utilizando também o otimizador Adam. Avaliamos o desempenho do modelo em tempo real com dados de validação para monitorar possíveis sinais de *overfitting.*

**Figura 12**

```
# Criar o modelo
cnn_model = build_cnn()

# Treinar o modelo
history = cnn_model.fit(
    train_generator,
    steps_per_epoch=len(train_generator),
    epochs=10,
    validation_data=val_generator,
    validation_steps=len(val_generator)
)
```

**Figura 12:** Inicia o treinamento da CNN com os dados de treino (train_generator), executando o número especificado de épocas, e avalia o desempenho após cada época usando o conjunto de validação (val_generator).

Em nosso código, utilizamos a principio 10 épocas para a realização do treinamento. A época é uma unidade de medida usada para representar o número de vezes que o algoritmo de treinamento passa por todo o conjunto de dados de treinamento. Durante uma época, todas as amostras do conjunto de dados são utilizadas para atualizar os pesos do modelo. Após o treinamento, o algoritmo segue para a fase de avaliação.

Em relação  as métricas utilizadas destacamos: Função de perda, F1-Score, curva ROC e área sobre a curva ROC (AUC-ROC). A função de perda usada no código foi a Binary Crossentropy, ela mede a diferença entre as probabilidades preditas pelo modelo e os rótulos reais em problemas de classificação binária. Quanto menor o valor da perda, melhor o modelo está ajustado aos dados.

**Figura 13**

$$\text{Loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i)]$$

```
model.compile(
    optimizer="adam",
    loss="binary_crossentropy",
    metrics=["accuracy"]
)
```

**Figura 13:** Equação e a região do código onde a função de perda é implementada. *loss="binary_crossentropy"* indica que a perda utilizada é a entropia cruzada para classificação binária.

Já a F1-Score é uma métrica que combina precisão e revocação em um único valor. É utilizada para avaliar o desempenho em datasets desbalanceados, pois considera tanto falsos positivos quanto falsos negativos:

**Figura 14**

```
f1 = f1_score(y_true, y_pred)
```

**Figura 14:** Região do código onde a métrica foi calculada. y_true: Rótulos reais das classes. y_pred: Previsões binárias (0 ou 1) geradas pelo modelo.

Já a curva ROC mostra a relação entre a taxa de verdadeiros positivos (TPR) e a taxa de falsos positivos (FPR) para diferentes limiares e deste modo, ajuda a visualizar o desempenho do modelo em diferentes cenários de decisão.

**Figura 15**

```
fpr, tpr, _ = roc_curve(y_true, y_pred_probs)
plt.plot(fpr, tpr, label=f"AUC = {auc:.4f}")
```

**Figura 15:** Trecho do código utilizado para gerar a curva ROC em que: fpr: é a Taxa de falsos positivos. tpr: Taxa de verdadeiros positivos.auc: Área sob a curva, incluída como legenda.

A AUC-ROC (Área Sob a Curva ROC) Mede a capacidade do modelo de distinguir entre classes (0 e 1). É importante porquê avalia como o modelo separa as classes, independentemente do limiar de decisão.

**Figura 16**

```
auc = roc_auc_score(y_true, y_pred_probs)
```

**Figura 16:** y_pred_probs representa as probabilidades preditas pelo modelo para a classe positiva.

É importante salientar que, durante a realização dos testes alguns resultados foram insatisfatórios que justificaram a modificação de hiperparâmetros, complexificação do modelo (adição de camadas e variação número de épocas) além de ajustes no dataset para testes e monitoramento dinâmico dos teste de validação. Ou seja, o modelo descrito na metodologia foi base para modificações posteriores, de modo que alguns detalhes no código foram alterados (complexificação do modelo). Essas modificações estão detalhadas nos tópicos posteriores, e podem ser consultadas no documento salvo no GIT HUB.

## 4. RESULTADOS

Neste tópico apresentaremos os resultados finais das implementações dos algoritmos desenvolvidos ao longo da execução deste projeto, os detalhes de sua execução serão apresentados nos tópicos finais. Foram feitas duas implementações a fim de observar o comportamento do modelo. Os resultados da primeira implementação indicaram uma precisão de 68,42% no conjunto de teste, com uma perda de 0,6387. A curva ROC apresentou um AUC de 0,4391, e o F1-Score foi calculado como 0,000.

**Figura 17**

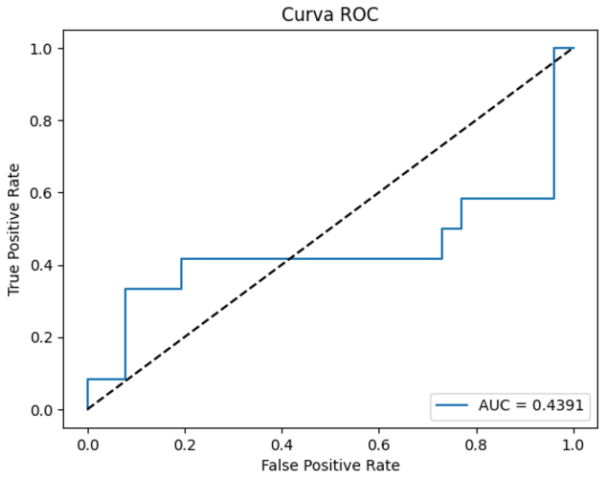



**Figura 17:** curva plotada durante a realização da avaliação do modelo CNN em sua primeira implementação.

**Tabela 1**

| Hiperparâmetro | Valor Utilizado |
|---|---|
| Tamanho do kernel | (3, 3) |
| Stride | Valor padrão (1, 1) |

| | |
|---|---|
| Padding | "Valid" (sem preenchimento adicional) |
| Número de filtros | 32, 64, 128 (em cada camada convolucional, respectivamente) |
| Tipo de pooling | MaxPooling |

**Tabela 1:** Hiperparâmetros utilizados na primeira implementação. Foram utilizadas 10 épocas.

Os resultados da segunda implementação indicaram uma precisão de 72,7% no conjunto de teste, com uma perda de 0,6425. A curva ROC apresentou um AUC de 0,4135, e o F1-Score foi calculado como 0,0000.

**Tabela 2**

| Hiperparâmetro | Valor Utilizado |
|---|---|
| Tamanho do kernel | (5, 5) |
| Stride | Valor padrão (1, 1) |
| Padding | "Same" (com preenchimento para manter o tamanho da entrada) |
| Número de filtros | 64, 128, 256 (em cada camada convolucional, respectivamente) |
| Tipo de pooling | MaxPooling |

**Tabela 2:** Hiperparâmetros utilizados na segunda implementação. Foram utilizadas 20 épocas.

**Figura 18**

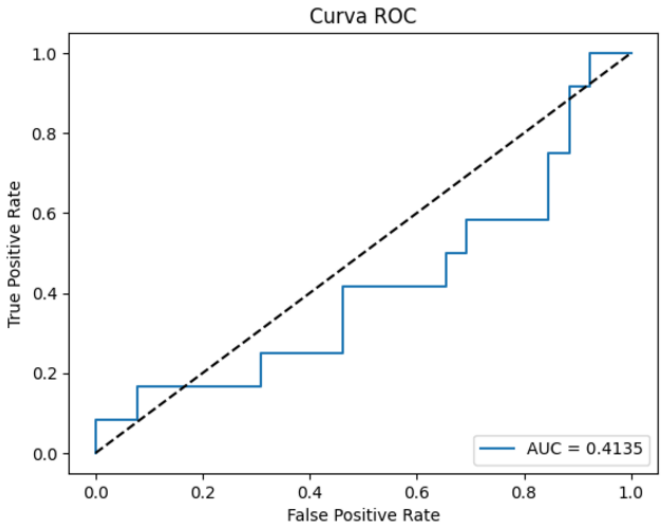



**Figura 18:** curva plotada durante a realização da avaliação do modelo CNN em sua segunda implementação.

## Tabela 3

| Hiperparâmetro | Valor Utilizado |
|---|---|
| Tamanho do kernel | (3, 3) |
| Stride | Valor padrão (1, 1) |
| Padding | "Same" (preenchimento para manter o tamanho) |
| Número de filtros | 64, 128, 256 (em cada camada convolucional, respectivamente) |
| Tipo de pooling | MaxPooling |

**Tabela 3:** Hiperparâmetros utilizados na terceira implementação. Foram utilizadas 20 épocas.

## Figura 19

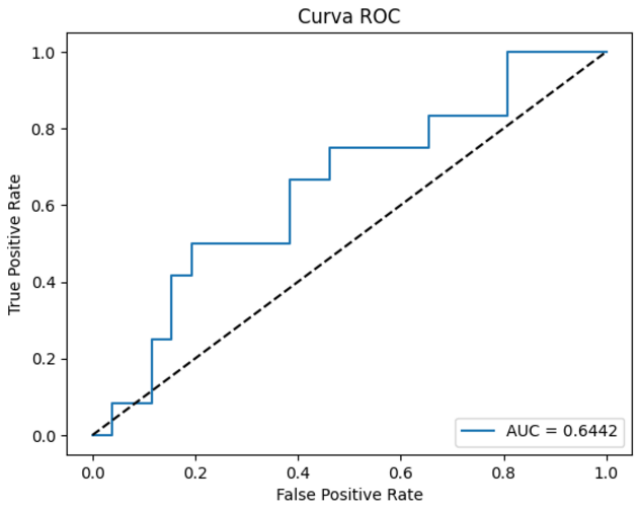



**Figura 19:** curva plotada durante a realização da avaliação do modelo CNN em sua terceira implementação.

## Tabela 3

| Hiperparâmetro | Valor Utilizado |
|---|---|
| Tamanho do kernel | (5, 5) |
| Stride | Valor padrão (1, 1) |
| Padding | "Valid" (sem preenchimento adicional) |
| Número de filtros | 32, 64, 128 (em cada camada convolucional, respectivamente) |
| Tipo de pooling | MaxPooling |
| Dropout | 0.5 |
| Regularização L2 | Fator de 0.01 |
| Função de perda | Binary Focal Loss |
| Otimizador | Adam com taxa de aprendizado 0.0005 |

**Tabela 3:** Hiperparâmetros utilizados na terceira implementação. Foram utilizadas 20 épocas.

## Figura 20

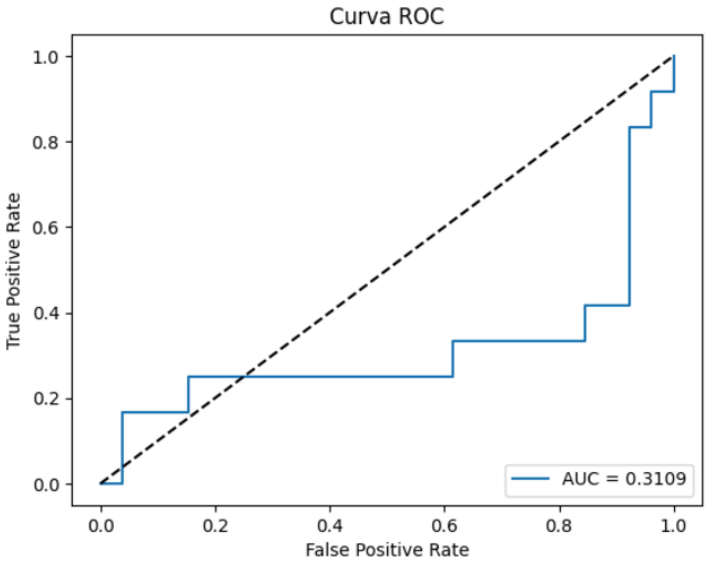



**Figura 20:** curva plotada durante a realização da avaliação do modelo CNN em sua quarta implementação.

## Tabela 5

| Hiperparâmetro | Valor Utilizado |
|---|---|
| Tamanho do kernel | (3, 3) |
| Stride | Valor padrão (1, 1) |
| Padding | "Valid" (sem preenchimento adicional) |
| Número de filtros | 32, 64, 128, 256 (em cada camada convolucional, respectivamente) |
| Tipo de pooling | MaxPooling |
| Dropout | 0.4 |
| Regularização L2 | Fator de 0.001 |
| Função de perda | Binary Crossentropy (com ajuste de classe para desbalanceamento) |
| Otimizador | Adam com taxa de aprendizado adaptativa (0.001 inicial) |

**Tabela 5:** Hiperparâmetros utilizados na ultima implementação. Foram utilizadas 12 épocas.

## Figura 21

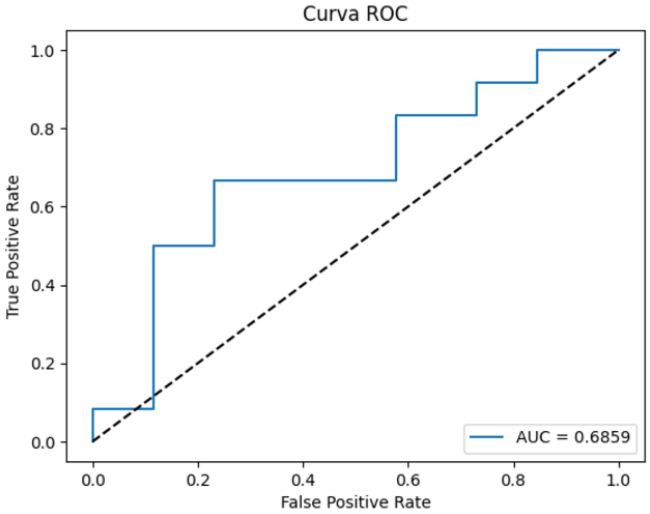



**Figura 21:** Curva plotada da última implementação da arquitetura CNN modificada para melhoras as métricas na ultima implementação.

Os dados foram organizados na tabela 6 onde podemos comparar as métricas de validação das 5 etapas do desenvolvimento da arquitetura CNN.

**Tabela 6**

| Implementação | Acurácia (%) | Loss | AUC-ROC | F1-Score | Observação |
|---|---|---|---|---|---|
| Primeira | 68,42 | 0,6387 | 0,4391 | 0 | Modelo inicial, sem ajustes |
| Segunda | 72,7 | 0,6425 | 0,4135 | 0 | Melhorou acurácia, F1 ainda ruim |
| Terceira | 72,7 | 0,6184 | 0,6442 | 0 | AUC-ROC avançou, mas F1 Score é 0 |
| Quarta | 68,42 | 10,356 | 0,3109 | 0 | Modelo instável, alto loss |
| Modelo Final | 71,05 | 0,6182 | 0,6859 | 0,6154 | Melhor balanço entre métricas |

**Tabela 6:** Principais métricas obtidas no processo de validação das 5 implementações da arquitetura CNN.

No próximo tópico, discutiremos os resultados da 5 tentativas de obter uma arquitetura CNN adequada a resolução do problema de classificação binária.

## 5. DISCUSSÃO

Neste tópico vamos discutir os resultados das implementações de 4 códigos conforme a arquitetura planejada. Selecionamos o melhor resultado dentre os 4 primeiros modelos CNN para tentar corrigir possíveis problemas de

balanceamento das classes que observamos durante as 4 implementações. Em relação ao balanceamento das classes, testamos duas metodologias: adição de cópias rotacionadas na classe (0) do conjunto train:

**Figura 22**

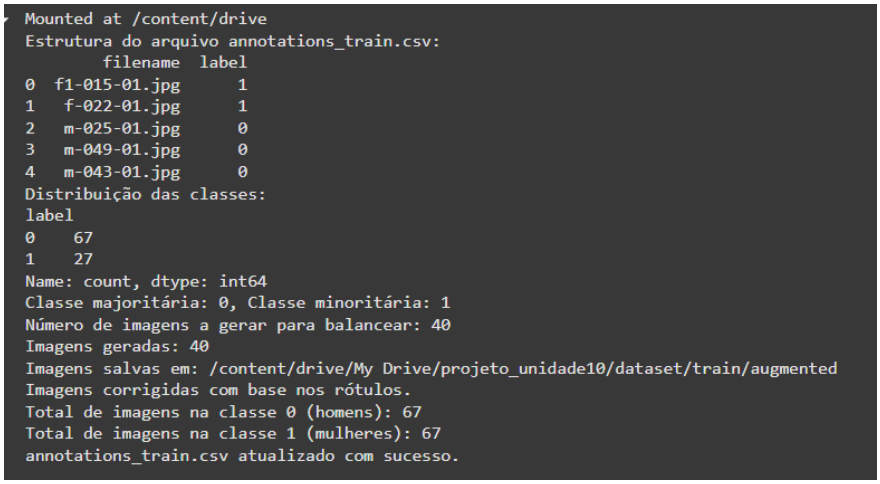



**Figura 22:** interface para o acompanhamento do balanceamento do dataset.

No primeiro caso, com cópias rotacionadas, foi possível aumentar o F1 score, mas ainda próximo de um modelo aleatório. Além disso, houve uma queda substâncial nas outras métricas, ou seja, os resultados indicam que o modelo não está aprendendo de forma efetiva:

- **AUC-ROC**: 0.5, equivalente a um modelo aleatório, indicando que o modelo não está distinguindo entre as classes.
- **F1-Score**: 0.48, também próximo ao desempenho de um modelo aleatório.
- **Accuracy de Treinamento**: Variando entre 27% e 31%, o que é muito baixo.
- **Accuracy de Validação**: Estável em 35.71%, mostrando que o modelo não está generalizando nem aprendendo no conjunto de treinamento.

**Loss**: Valores negativos no treinamento e extremamente altos na validação, sugerindo problemas na função de perda, dados ou no modelo.

**Figura 24**

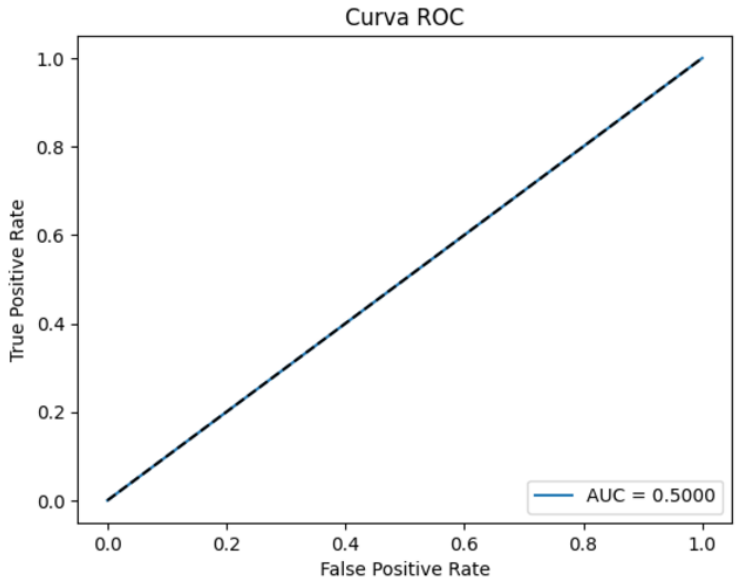



**Figura 24:** Gráfico com os resultados (curva ROC) dos treinos do algoritmo 3 com dataset balanceado com cópias rotacionadas.

Após isso tentamos balancear com a adição de cópias exatas na classe (0) do conjunto train, no entanto 0 modelo ainda apresenta desempenho insatisfatório, com loss negativo no treinamento e extremamente alto na validação, indicando problemas na função de perda ou nos dados. A accuracy baixa (30% no treinamento e 35.71% na validação) e o AUC-ROC de 0.5 sugerem que o modelo não está distinguindo os elementos e suas respectivas classes.

**Figura 25**

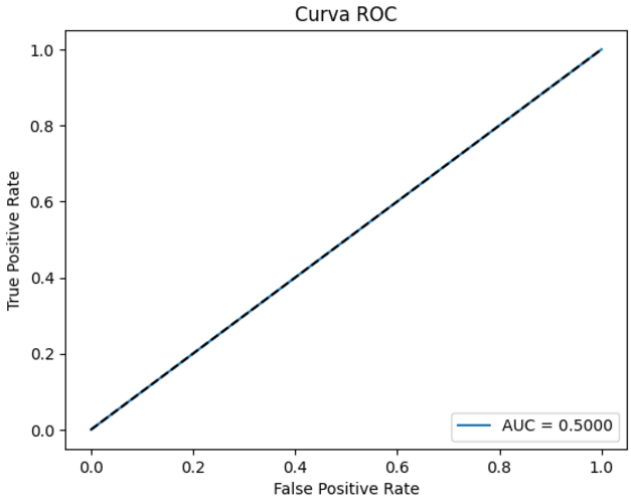



**Figura 25:** Gráfico com os resultados (curva ROC) dos treinos do algoritmo 3 com dataset balanceado com cópias exatas.

Removemos as alterações no *dataset* tentamos algumas modificações no modelo, a fim de equilibrar as métricas, e obter um aprendizado moderado, considerando as limitações relacionadas ao tamanho do dataset e principalmente, o desbalanceamento das classes. Então buscamos investigar manualmente como essas imagens estão sendo classificadas. Foi escrito um código para buscar as imagens classificas como homem e mulher na 3º implementação (uma vez que esta teve o melhor resultado até então) a partir disso foram feitas as alterações que podem ser conferidas na **tabela 5** nos resultados.

**Figura 26**

```
58 # Função para construir o modelo CNN
59 def build_cnn(input_shape=(250, 200, 3)):
60     model = tf.keras.models.Sequential([
61         tf.keras.layers.Conv2D(64, (3, 3), activation="relu", input_shape=input_shape),
62         tf.keras.layers.MaxPooling2D((2, 2)),
63         tf.keras.layers.Conv2D(128, (3, 3), activation="relu"),
64         tf.keras.layers.MaxPooling2D((2, 2)),
65         tf.keras.layers.Conv2D(256, (3, 3), activation="relu"),
66         tf.keras.layers.MaxPooling2D((2, 2)),
67         tf.keras.layers.Flatten(),
68         tf.keras.layers.Dense(128, activation="relu"),
69         tf.keras.layers.Dropout(0.5),
70         tf.keras.layers.Dense(1, activation="sigmoid")
71     ])
72     model.compile(
73         optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
74         loss="binary_crossentropy",
75         metrics=["accuracy"]
76     )
77     return model
```

**Figura 26:** interface para o acompanhamento do balanceamento do dataset.

Deste modo, na última implementação, melhorias foram feitas para aumentar o desempenho do modelo (A 3° implementação), especialmente o F1-Score. Foi utilizada a função de perda Binary Crossentropy ajustada para lidar com o desbalanceamento do dataset, enquanto o limiar de classificação foi ajustado para 0.27, otimizando o equilíbrio entre precisão e recall. A profundidade do modelo foi ampliada com mais filtros em cada camada convolucional, e uma regularização L2 foi adicionada para reduzir overfitting. O dropout foi ajustado para 0.4, e o otimizador Adam com taxa de aprendizado adaptativa (0.001) ajudou a melhorar a eficiência do treinamento. Essas alterações elevaram o F1-Score para 0.615, com melhorias também na AUC-ROC e acurácia.

Foram utilizados filtros de tamanho (3,3) aplicados em três camadas convolucionais sequenciais, com 64, 128 e 256 filtros respectivamente. O modelo também incluiu camadas de pooling (2,2) para redução dimensional, uma camada densa com 128 neurônios e uma função de ativação relu, além de dropout de 0,5 para regularização. A saída foi configurada com uma função de ativação sigmoid para a classificação binária.

O modelo alcançou uma acurácia de 71,05%, com AUC-ROC de 0,6859, indicando uma moderada capacidade de separação entre as classes. O F1-Score foi de 0,6154, demonstrando uma melhora significativa no equilíbrio entre precisão e recall. O ajuste no limiar de classificação contribuiu para essa melhoria nas métricas, mostrando que o modelo respondeu bem às otimizações realizadas nos hiperparâmetros e no treinamento.

Realizamos a análise visual das imagens classificadas pelo nosso modelo, foi desenvolvido uma aplicação em python para exibir os dados de treinamento, e organizar as imagens rotuladas em pasta para análise visual como pode ser obsevado nas figurAS

**Figura 27**

```
Classificação detalhada:
          filename  real_label  predicted_label
0    m-028-01.jpg           0                0
1    m-035-01.jpg           0                0
2    m-037-01.jpg           0                1
3    m-039-01.jpg           0                0
4    m-044-01.jpg           0                0
5    m-046-01.jpg           0                0
6    m-047-01.jpg           0                0
7    m-061-01.jpg           0                0
Relatório de Classificação:
              precision    recall  f1-score   support

       Homem       0.80      0.77      0.78        26
      Mulher       0.54      0.58      0.56        12

    accuracy                           0.71        38
   macro avg       0.67      0.68      0.67        38
weighted avg       0.72      0.71      0.71        38
```

**Figura 27:** Interface gerada em Python para observação visual da classificação do ultimo modelo nas pastas do drive.

**Figura 28**

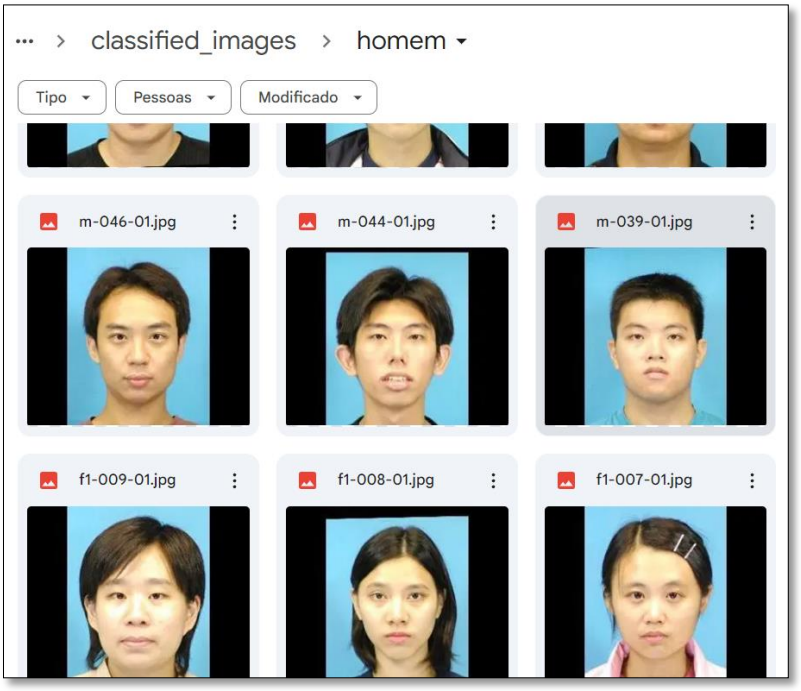



**Figura 28:** Imagens classificadas como Homem.

**Figura 29**

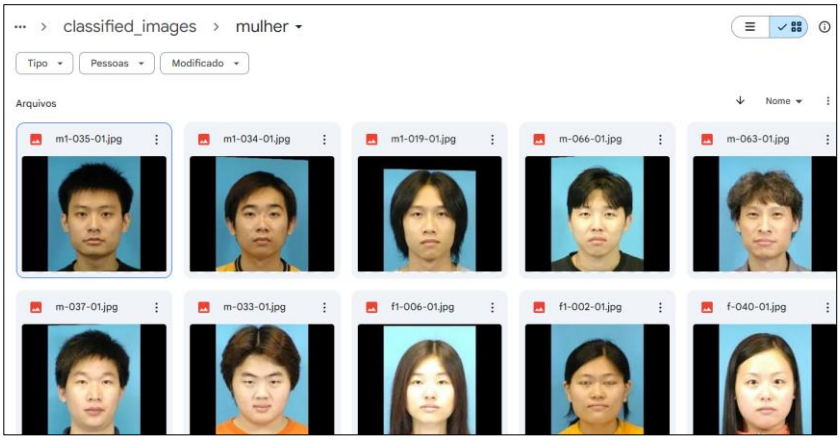



**Figura 29:** Imagens classificadas como mulher.

Observando os dados da tabela 27 percebemos que: O modelo aprendeu melhor a classificar homens, e isso pode ter a ver com a maior quantidade de fotos masculinas no dataset. Podemos observar visualmente que, os homens classificados como mulher apresentam lábios maiores ou cabelos longos, formato do rosto mais redondo ou cabelo. As mulheres classificadas como homens também apresentavam características andróginas e rostos pouco expressivos.

Estes resultados podem ter relação com:

- Tamanho do dataset
- Qualidade das imagens (caracteriscas como oleosidade da pele não pode ser avaliadas com imagens de baixa definição
- Desbalanceamento das classes no treinamento
- Arquitetura inadequada

Realizamos mais algumas tentativas com o mesmo código e as métricas apresentavam valores similares, aparentemente a arquitetura desenvolvida atingiu o seu limite, uma vez que, as modificações geravam *underfitting e overfitting*. sendo necessário pensar outras soluções para este desafio conforme serão elencadas no próximo tópico.

## 6. CONCLUSÃO E TRABALHOS FUTUROS

Uma possível solução para este modelo, considerando as características do dataset (e a impossibilidade de expansão), e a regra de não utilização de modelos pré-treinados seria recorrer a uma estrutura de **inputs dinâmicos com rotulação manual** pode ser uma abordagem eficaz para

melhorar o modelo, especialmente em tarefas onde o modelo apresenta dificuldades em entender padrões específicos. A ideia seria criar um **fluxo interativo de treinamento e ajuste** onde os humanos participam diretamente no refinamento do modelo. Aqui está uma visão geral de como isso poderia ser implementado:

- Esse processo pode ser **demorado** se houver muitas imagens para revisar.
- Necessário definir **critérios claros de revisão**, como limiar de confiança para identificar erros.
- É importante documentar as correções para verificar os impactos das alterações no modelo.

Essa abordagem combina **automação com supervisão humana**, maximizando o aprendizado do modelo a partir de erros, o que é particularmente útil em contextos onde os dados são limitados ou desafiadores. Essa abordagem se faz necessária, uma vez que é preciso descobrir a combinação mais adequada dos hiperparâmetros e camadas. Outra estratégia que pode ajudar no treinamento, é intervenção no treinamento, uma vez que há poucas imagens para treinar, a intervenção humana no momento do treinamento pode auxiliar no ajuste de pesos.

## 7. REFERÊNCIAS


ALMEIDA, Ednilson S.; LUDERMIR, Teresa B. *Redes Neurais Artificiais e Algoritmos Genéticos Aplicados à Previsão de Séries Temporais.* São Paulo: Editora da UFPE, 2011.

FERREIRA, André C. S. *Aprendizado de Máquina: Fundamentos e Aplicações.* Brasília: Editora UnB, 2019.

GARCÍA, Salvador; LUENGO, Julián; HERRERA, Francisco. *Data Preprocessing in Data Mining.* Cham: Springer, 2015. Disponível em: https://link.springer.com/book/10.1007/978-3-319-10247-4. Acesso em: 17 nov. 2024.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. Deep learning. Cambridge: MIT Press, 2016.

HAYKIN, Simon. *Redes Neurais: Princípios e Prática.* 2. ed. Porto Alegre: Bookman, 2001.

HAYKIN, Simon. *Redes neurais e aprendizado profundo: uma abordagem prática para o aprendizado de máquina*. Porto Alegre: Bookman, 2020.

PEDREIRA, César E. T.; NETO, J. R. D.; ARAÚJO, B. D. *Machine Learning com Python: Fundamentos e Aplicações.* Rio de Janeiro: Alta Books, 2018.

ROCHA, Alexandre C. *Introdução à Inteligência Artificial e Machine Learning.* São Paulo: Novatec, 2020.