Capítulo 2

Detecção de Máscara em Python Usando OpenCV e Deep Learning

Vitória de Carvalho Brito, Patrick Ryan Sales dos Santos e Antonio Oseas de Carvalho Filho

Abstract

In view of the pandemic scenario we are experiencing, we propose a minicourse aimed at the application of Computer Vision techniques that can be useful to support the authorities in the control of restrictive measures. The minicourse aims to explain the concepts of techniques such as Convolutional Neural Network, Multi-task Cascaded Convolutional Networks, FaceNet and Multilayer Perceptron, practically applying such techniques in a Computer Vision system to detect whether people in a given environment are or are not using mask. This chapter describes the technologies used in the mini-course, as well as the steps involved in implementation.

Resumo

Diante do cenário de pandemia ao qual estamos vivenciando, propomos um minicurso voltado à aplicação de técnicas de Visão Computacional que podem ser úteis para apoiar as autoridades no controle das medidas restritivas. O minicurso tem como objetivo explicar os conceitos de técnicas como Rede Neural Convolucional, Multi-task Cascaded Convolutional Networks, FaceNet e Multilayer Perceptron, aplicando de forma prática tais técnicas em um sistema de Visão Computacional para detectar se pessoas em um determinado ambiente estão ou não utilizando máscara. Este capítulo descreve as tecnologias utilizadas no minicurso, bem como os passos envolvidos na implementação.

2.1. Introdução

Desde a descoberta do novo coronavírus (COVID-19) na China no final de 2019, a doença tornou-se uma preocupação mundial, principalmente devido à sua rápida disseminação. Em setembro de 2021, o número de casos confirmados notificados à Organização Mundial

da Saúde (OMS) já ultrapassava 223 milhões, enquanto o número de óbitos já ultrapassava 4 milhões [WHO 2021].

Em virtude do cenário de pandemia, diversas soluções computacionais têm sido propostas para apoiar as autoridades no controle das medidas restritivas, como por exemplo a detecção de máscara em tempo real. Os algoritmos de detecção de máscara podem ser utilizados em sistemas embarcados presentes em câmeras de supermercados, shoppings, aeroportos, hospitais, escolas e outros ambientes que acomodam um grande número de pessoas.

O propósito deste projeto é desenvolver um algoritmo para detecção de máscaras utilizando técnicas de *Deep Learning* para a caracterização e classificação das faces e a biblioteca OpenCV para a captura de vídeo em tempo real. Ao final, espera-se instigar o leitor quanto a utilização dos métodos de detecção facial e proporcionar um projeto que pode agregar o portfólio dos participantes.

2.1.1. Objetivos

Este capítulo visa construir uma aplicação de Visão Computacional e apresentar os conceitos envolvidos em cada etapa do sistema, bem como as tecnologias utilizadas. De maneira específica, pretende-se:

- Abordar, de maneira sucinta, os principais conceitos de Visão Computacional;
- Explanar os conceitos de Redes Neurais Convolucionais;
- Introduzir sobre a *Multi-task Cascaded Convolutional Networks*;
- Introduzir sobre a *FaceNet*;
- Introduzir sobre o classificador *MLP*;
- Aplicar, de maneira prática, os conceitos explanados.

2.1.2. Organização do Capítulo

Este capítulo está organizado da seguinte maneira: a Seção 2.2 apresenta os conceitos das técnicas utilizadas no desenvolvimento da aplicação; a Seção 2.3 detalha os passos envolvidos na execução do método, bem como os algoritmos criados; por fim, a Seção 2.4 mostra as considerações finais do trabalho, destacando as contribuições e sugestões de melhoria.

2.2. Referencial Teórico

Esta seção fornece um relato da literatura sobre os principais temas abordados na proposta deste trabalho, contribuindo para o entendimento das etapas desenvolvidas.

2.2.1. Visão Computacional

Visão computacional é a ciência responsável pela visão de uma máquina, pela forma como um computador enxerga o meio à sua volta, extraindo informações significativas

a partir de imagens capturadas por câmeras de vídeo, sensores, scanners, entre outros dispositivos. Estas informações permitem reconhecer, manipular e pensar sobre os objetos que compõem uma imagem [Ballard e Brown 1982].

O olho humano consegue perceber e interpretar objetos em uma imagem de forma muito rápida. Isso acontece no córtex visual do cérebro, uma das partes mais complexas no sistema de processamento do cérebro. Alguns cientistas concentraram seus estudos na tentativa de entender o funcionamento dessa parte do cérebro, para então trazer tais ideias para a Visão Computacional. É o que pesquisadores do MIT definem como "ensinar computadores a enxergarem como humano"[de Milano e Honorato 2014].

Dessa forma, a visão computacional fornece ao computador uma infinidade de informações precisas a partir de imagens e vídeos, de forma que o computador consiga executar tarefas inteligentes, simulando e aproximando-se da inteligência humana.

Em geral, um sistema de Visão Computacional é composto pelas seguintes etapas:

- **Aquisição:** nesta etapa as imagens são capturadas e representadas de forma computacional para serem interpretadas pela etapa seguinte.
- **Processamento de imagens:** o objetivo deste estágio é adequar e otimizar os dados visuais adquiridos. Para isso, podem ser aplicadas algumas técnicas como retirada de ruídos, rotação da imagem, aplicação de filtros, etc.;
- **Segmentação:** consiste em dividir a imagem em objeto(s) e fundo. Em outras palavras, essa etapa consiste em técnicas que de alguma maneira consigam formar padrões de agrupamento, gerando sub-regiões que possuem entre si alguma similaridade;
- **Extração de características:** tem como objetivo representar, através de valores, uma imagem ou partes dela. Estes valores são características fundamentais que representam propriedades contidas nas imagens;
- **Reconhecimento de padrões:** neste ponto as imagens são classificadas em função de suas características similares.

2.2.2. *Deep Learning*

O conceito de algoritmos de *Deep Learning* foi introduzido no final do século XX, permitindo que modelos computacionais compostos por várias camadas de processamento aprendam representações de dados com vários níveis de abstração. Em contraste com as abordagens tradicionais de *Machine Learning*, as tecnologias de *Deep Learning* estão progredindo recentemente em aplicações para reconhecimento de fala, processamento de linguagem natural, recuperação de informações, visão computacional e análise de imagens [Liu et al. 2017].

As Redes Neurais Convolucionais (*CNNs*, do inglês *Convolutional Neural Networks*) fazem parte de uma categoria de algoritmos de *Deep Learning* que tornou-se o novo padrão na área de Visão Computacional. As *CNNs* são baseadas no processamento de dados visuais, capaz de aplicar filtros nesses dados, mantendo a relação de vizinhança entre os

pixels da imagem ao longo do processamento da rede [LeCun et al. 1998]. Essa operação é conhecida como convolução, onde ocorre a somatória do produto ponto a ponto entre os valores de um filtro e cada posição da vizinhança do pixel de entrada.

As camadas convolucionais são formadas por um conjunto de filtros que são iniciados com um arranjo 3D, muitas das vezes chamado de volume. Cada filtro tem uma dimensão reduzida, porém estende-se por toda a profundidade do volume de entrada. No processo de treinamento da rede, esses filtros são ajustados automaticamente para que possam extrair características relevantes [Karpathy 2014].

O objetivo da camada de *pooling* é reduzir a dimensionalidade do volume de entrada, de modo geral esse progresso ocorre logo após uma camada convolucional, diminuindo consideravelmente o custo computacional da rede e o tempo de processamento. Por consequência, a camada de *pooling* tende a evitar o *overfitting*. No processo de *pooling*, cada valor da nova matriz é referente ao resultado de alguma métrica aplicada a uma região do mapa de convoluções. Diversas métricas podem ser aplicadas no *pooling*, como o máximo valor da região, o mínimo ou a média, dependendo do problema abordado.

A imagem de entrada fornece para a camada convolucional e de *pooling* a possibilidade de extrair as características relevantes acerca dessa determinada imagem. O objetivo da camada totalmente conectada é utilizar essas características para classificar a imagem em um rótulo pré-determinado. As camadas responsáveis pela classificação das características extraídas das camadas convolucionais são exatamente como uma rede artificial convencional [Haykin et al. 2009].

2.2.2.1. Multi-task Cascaded Convolutional Network

A *Multi-task Cascaded Convolutional Network* (*MTCNN*) é um método de detecção facial baseado em *CNN*, proposto por Zhang et al. [Zhang et al. 2016]. Como mostra a Figura 2.1, o modelo divide-se em três estágios capazes de reconhecer faces e locais de referência, como olhos, nariz e boca.

O primeiro estágio é uma *Fully Convolutional Network* (*FCN*). A diferença entre uma *CNN* e uma *FCN* é que a *FCN* não usa a camada densa como parte de sua arquitetura. A *FCN* usada é denominada *Proposal Network* (*P-Net*), ela é responsável por obter janelas candidatas e seus vetores de regressão de caixas delimitadoras. A regressão de caixa delimitadora é uma técnica popular para prever a localização de regiões quando o objetivo é detectar um objeto de alguma classe predefinida, neste caso faces. Depois de obter os vetores da caixa delimitadora, algum refinamento é feito para combinar regiões sobrepostas. A saída final desse estágio são todas as janelas candidatas após o refinamento para reduzir o volume de candidatas [Gradilla 2020].

Todas as regiões candidatas da *P-Net* são alimentadas na *Refine Network* (*R-Net*). Esta rede é uma *CNN*, não uma *FCN* como a anterior, pois há uma camada densa no último estágio da arquitetura da rede. A *R-Net* reduz ainda mais o número de regiões candidatas, realiza a calibração com a regressão das caixas delimitadoras e emprega uma supressão não máxima para mesclar as caixas candidatas sobrepostas. A saída da *R-Net* prevê se a entrada é uma face ou não, um vetor de 4 elementos (caixa delimitadora para a

face) e um vetor de 10 elementos para a localização do ponto de referência facial.

Por fim, a *MTCNN* possui a *Output Network* (*O-Net*) para operar o terceiro estágio. Este estágio é semelhante ao da *R-Net*, mas a *O-Net* visa descrever o rosto com mais detalhes e produzir as cinco posições dos marcos faciais para olhos, nariz e boca. De acordo com Edwin et. al. [Jose et al. 2019], a *MTCNN* supera de forma consistente os métodos convencionais sofisticados em relação à confiabilidade do desempenho em tempo real. Este desempenho em tempo real é de grande importância em um sistema de vigilância.

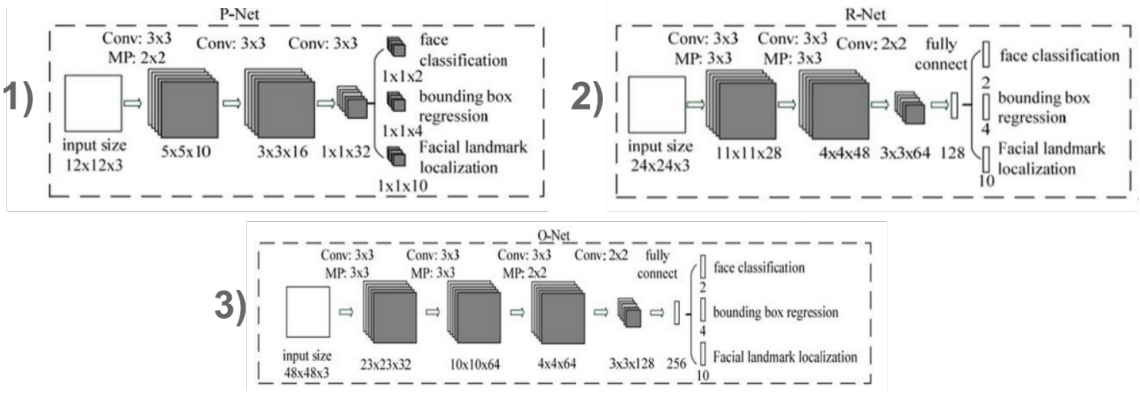

Figura 2.1: Arquitetura da *MTCNN*. Fonte: [Zhang et al. 2016]

2.2.2.2. *FaceNet*

A *FaceNet* é um modelo de reconhecimento facial proposto por Florian Schroff, et al. [Schroff et al. 2015], no Google. A *FaceNet* aprende diretamente um mapeamento a partir de imagens de rosto para um espaço euclidiano compacto onde as distâncias correspondem diretamente a uma medida de semelhança facial. Uma vez que este espaço foi produzido, tarefas como reconhecimento facial, verificação e agrupamento podem ser facilmente implementadas usando técnicas padrão com *embeddings FaceNet* como vetores de características.

O modelo é uma *CNN* profunda treinada por meio de uma função *triplet loss*. Esse treinamento é feito de forma que a distância quadrada L2 entre os *embeddings* corresponda à semelhança de faces: faces da mesma pessoa têm pequenas distâncias e faces de pessoas distintas têm grandes distâncias [Schroff et al. 2015]. A *FaceNet* representa cada face através de um vetor de 128 características.

Para calcular a *triplet loss* são necessárias 3 imagens, chamadas de âncora, positiva e negativa. A intuição por trás da função *triplet loss* é que a imagem âncora (imagem de uma pessoa A específica) esteja mais próxima das imagens positivas (todas as imagens da pessoa A) em comparação com as imagens negativas (todas as outras imagens).

Em outras palavras, o objetivo é que as distâncias entre o *embedding* da imagem âncora e o *embedding* das imagens positivas sejam menores em comparação com as distâncias entre o *embedding* da imagem âncora e o *embedding* das imagens negativas. As Figuras 2.2 e 2.3 ilustram o funcionamento da *FaceNet*.

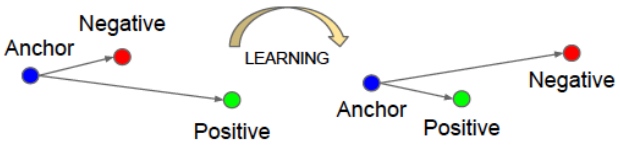

Figura 2.2: Triplet Loss. Fonte: [Kumar 2021].

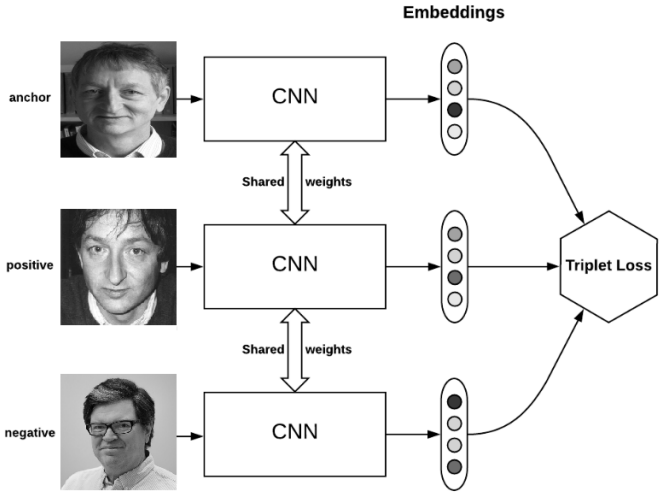

Figura 2.3: Ilustração do funcionamento da *FaceNet*. Fonte: [Kumar 2021].

2.2.2.3. *Multilayer Perceptron*

As Redes Neurais Artificiais tentam modelar o funcionamento do cérebro humano. O cérebro humano, por exemplo, consiste em bilhões de células individuais chamadas neurônios. Dado que o cérebro humano consiste em um grande número de neurônios, a quantidade e a natureza das conexões entre os neurônios são, nos níveis atuais de compreensão, quase impossíveis de avaliar [Ramchoun et al. 2016].

O *Multilayer Perceptron* (*MLP*) é o modelo mais utilizado em redes neurais usando o algoritmo de treinamento de retropropagação. A definição de arquitetura em redes *MLP* é um ponto muito relevante, pois a falta de conexões pode tornar a rede incapaz de resolver o problema de parâmetros ajustáveis insuficientes, enquanto um excesso de conexões pode causar um sobreajuste dos dados de treinamento [Ramchoun et al. 2017], especialmente quando usamos um grande número de camadas e neurônios.

O processo de aprendizado no *MLP* ocorre pela adaptação das conexões pesos a fim de obter uma diferença mínima entre a saída da rede e a saída desejada. Normalmente, utiliza-se o algoritmo de retropropagação baseada em técnicas de gradiente descendente [Ramchoun et al. 2016].

O *MLP* consiste em pelo menos três camadas de nós: uma camada de entrada, uma camada oculta e uma camada de saída. Exceto para os nós de entrada, cada nó é um neurônio que usa uma função de ativação não linear. Suas múltiplas camadas e ativação não linear distinguem o *MLP* de um perceptron linear. Ele pode distinguir dados que não

são linearmente separáveis [Ramchoun et al. 2016].

2.3. Metodologia

Esta seção descreve os passos envolvidos na construção da aplicação. Cada etapa será explicada e os algoritmos de implementação serão mostrados. Toda a implementação deste projeto está disponível no GitHub, bem como a base de imagens e as instruções para executar os códigos. A Figura 2.4 ilustra as etapas desenvolvidas.

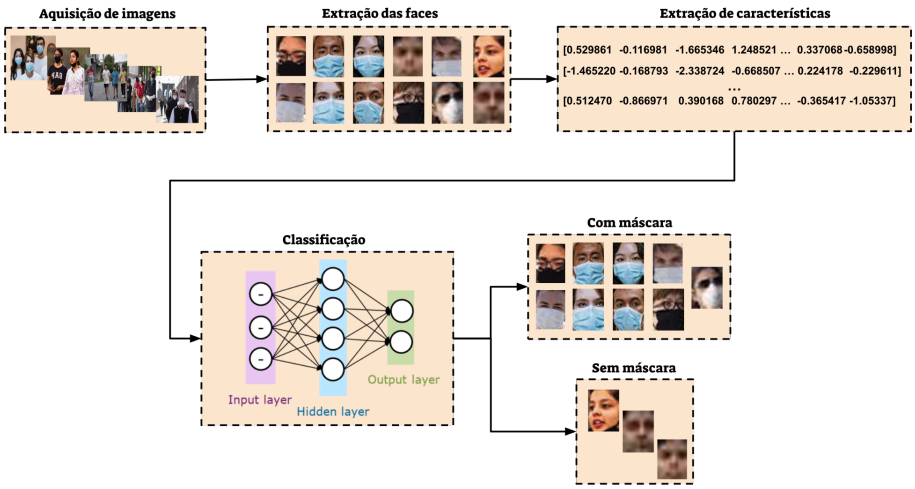

Figura 2.4: Fluxo da aplicação.

2.3.1. Aquisição de Imagens e Pré-processamento

A base de imagens utilizada no projeto foi adquirida, em sua maioria, através do material do professor Sandeco, em seu vídeo sobre detecção de máscaras. A base do professor Sandeco possui 898 imagens de pessoas com máscara e 958 de pessoas sem máscara. Para melhorar essa base, retiramos algumas imagens que não eram necessárias, como por exemplo imagens de pessoas em desenho/animação/não reais. Além disso, adicionamos algumas imagens de pessoas com óculos e máscaras, já que haviam poucas imagens desse tipo na base.

Ao final deste processamento, obtivemos um conjunto de dados que resultou em 1087 imagens de pessoas com máscara e 1000 de pessoas sem máscara, um conjunto relativamente balanceado. Tanto as imagens da base original, quanto da base resultante foram extraídas da internet através da pesquisa de imagens por palavras-chave relacionadas ao tema e métodos para download das imagens em lote. Após a aquisição, as imagens passaram por uma etapa de pré-processamento, onde as faces foram extraídas e redimensionadas para o tamanho 160x160, uma vez que este é o formato de entrada da *FaceNet*.

Após as importações necessárias para os códigos, podemos carregar a base de imagens como no exemplo abaixo.

```
from tensorflow import keras
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, cohen_kappa_score,
    confusion_matrix
```

```
from sklearn.neural_network import MLPClassifier
from skimage.io import imread, imshow
import numpy as np
from glob import glob
from tqdm.notebook import tqdm
import pandas as pd
import pickle

maskon = glob('./new_dataset/maskon/*.png')
maskoff = glob('./new_dataset/maskoff/*.png')

print(len(maskon), len(maskoff))
```

Código Fonte 2.1: Importações necessárias e carregamento das imagens.

2.3.2. Extração de Características

A extração de características das faces foi feita por meio do modelo *FaceNet*, que gera um vetor de 128 características para cada entrada, que chamamos de *embeddings*. Neste projeto, usamos o modelo pré-treinado da *FaceNet*, em sua versão implementada no keras. Nós disponibilizamos esse modelo no repositório do projeto no GitHub.

É necessário carregar o modelo treinado e em seguida extrair os *embeddings* de cada imagem através da *FaceNet*.

```
model = keras.models.load_model('./facenet_keras.h5')

data = maskon + maskoff

embeddings = []

for path in tqdm(data):

  try:
    # Lendo e normalizando a imagem
    img = imread(path).astype('float32')/255

    # Aplicando um reshape na imagem para deixar o formato de acordo
    com o input da FaceNet
    input = np.expand_dims(img, axis=0)

    # Extraindo o vetor de embeddings
    embeddings.append(model.predict(input)[0])

  except:
    print(f'Error in {path}')
    continue

labels = pd.DataFrame({
    'label': [1]*len(maskon) + [0]*len(maskoff)
})

df_embeddings = pd.concat([pd.DataFrame(embeddings), labels], axis=1)
```

Código Fonte 2.2: Extração de características com a *FaceNet*.

2.3.3. Classificação

Para classificar os *embeddings* extraídos na etapa anterior, utilizamos o classificador *MLP*. Utilizamos o *MLP* disponível na biblioteca sklearn, que possui vários algoritmos de regressão, classificação e agrupamento. Dividimos o conjunto de dados em treino e teste, com uma proporção de 80% e 20%, respectivamente.

Primeiro separamos o conjunto em treino e teste, em seguida instanciamos, treinamos e testamos o *MLP*.

```
X = np.array(df_embeddings.drop('label', axis=1))
y = np.array(df_embeddings['label'])

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
    =0.2, random_state=0)

print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)

mlp_model = MLPClassifier()

mlp_model.fit(X_train, y_train)

y_pred = mlp_model.predict(X_test)

print('Acurácia: ', accuracy_score(y_test, y_pred))
print('Kappa: ', cohen_kappa_score(y_test, y_pred))
print('Matriz de confusão:\n', confusion_matrix(y_test, y_pred))
```

Código Fonte 2.3: Classificação dos *embeddings* usando o *MLP*.

A acurácia mede a proporção de acerto do modelo, sem considerar o desbalanceamento de classes. Por este motivo, optamos por avaliar o método através do índice kappa também, que leva em consideração o desbalanceamento, aumentando a confiabilidade dos resultados.

Nossos testes mostraram acurácia de 0,9569 e kappa de 0,9135, mas esses valores podem variar de acordo com a aleatoriedade da divisão dos dados ou pela variação de parâmetros do classificador. Para finalizar a parte 1 deste projeto, treinamos o modelo novamente antes de salvá-lo, mas agora usando todos os dados para treino. Quanto mais exemplos de entrada nosso modelo recebe, melhor será seu aprendizado.

```
# Instanciando novamente
mlp_model = MLPClassifier()

# Treinando com todos os dados
mlp_model.fit(X, y)

# Mostrando a acurácia de treino
print(mlp_model.score(X, y))

# Salvando o modelo como um arquivo pickle
pickle.dump(mlp_model, open('./mlp_model.pkl', 'wb'))
```

Código Fonte 2.4: Treinando e salvando o modelo final de classificação. usando o *MLP*.

2.3.4. Detecção em Tempo Real

Com a execução de todas etapas anteriores, teremos um modelo de classificação treinado e pronto para ser utilizado em um cenário real. Para isso, utilizamos a biblioteca OpenCV, uma biblioteca de funções de programação voltada principalmente para a Visão Computacional em tempo real, exatamente o que precisamos.

No código abaixo, temos as importações e o carregamento dos modelos necessários. Apesar da *MTCNN* ter sido utilizada na aquisição de imagens, para extrair as faces de imagens que não estavam na base original, aqui ela é essencial. A *MTCNN* nos fornece um conjunto de coordenadas para cada face detectada, referentes à posição do olho esquerdo, olho direito, nariz, canto esquerdo da boca e canto direito da boca.

```
import pickle
from PIL import Image
from mtcnn import MTCNN
from tensorflow import keras
import cv2 as cv
import numpy as np

# Carregando modelo da FaceNet
facenet_model = keras.models.load_model('./facenet_keras.h5')

# Carregando modelo da MLP
mlp_model = pickle.load(open('./mlp_model.pkl', 'rb'))

# Carregando modelo da MTCNN
detector = MTCNN()
```

Código Fonte 2.5: Importações e carregamento dos modelos necessários.

Basicamente, a OpenCV nos fornecerá a imagem da webcam, a *MTCNN* detectará as faces existentes nessa imagem, a *FaceNet* extrairá as características das faces detectadas e o *MLP* classificará essas características quanto ao uso de máscara.

Abaixo, demos um nome para cada label numérico, pois será esse nome que nós utilizaremos para exibir em cima do retângulo da face, através da OpenCV. Também demos uma cor para cada label, lembrando que o modelo de cor padrão da OpenCV é BGR (blue, green e red), não RGB.

```
label_description = {
    0: 'NO MASK',
    1: 'MASK'
}

color = {
    0: (0, 0, 255),
    1: (0, 255, 0)
}
```

Código Fonte 2.6: Definindo descrição e cor dos labels.

Em seguida, criamos alguns métodos para facilitar a utilização dos modelos e a manipulação dos frames. O primeiro método chamamos de *get_faces*, ele é responsável

por extrair as faces de uma imagem através do modelo *MTCNN* e retorná-las como uma lista de imagens (faces) do tipo Image (Pillow). Como parâmetros dessa função, temos *image*, uma imagem de qualquer dimensão, e *size*, que refere-se à dimensão que as faces recebem antes de serem retornadas, o padrão é (160, 160).

```
def get_faces(image, size=(160, 160)):

    # Transformando imagem em um array numpy
    img = np.asarray(image)

    # Capturando as faces da imagem através da MTCNN
    results = detector.detect_faces(np.asarray(image))

    # Lista para armazenar as faces
    faces = []

    # Percorrendo a lista de faces detectadas
    for i in range(len(results)):

      try:

        # Caso a face tenha sido detectada com mais de 95% de certeza, essa condição é verdadeira
        if results[i]['confidence'] > 0.95:

          # Extraindo os pontos da face
          # w -> width (largura)
          # h -> height (altura)
          x1, y1, w, h = results[i]['box']
          x2, y2 = x1 + w, y1 + h

          # Extraindo a face da imagem fazendo slice nos pontos identificados pela MTCNN
          face = image[y1:y2, x1:x2]

          # Adicionando a face encontrada e suas informações na lista de faces
          # Cada face é redimensionada para o formato especificado na variável size
          faces.append({
              'x1': x1,
              'y1': y1,
              'x2': x2,
              'y2': y2,
              'face': np.array(Image.fromarray(face).resize(size)),
              'confidence': results[i]['confidence']
          })

      except:
        continue

    return faces
```

Código Fonte 2.7: Método responsável por extrair as faces de uma imagem através do modelo *MTCNN*.

Um ponto importante sobre o código acima é que retornamos uma face apenas se o modelo tiver mais de 95% de certeza de que aqueles pontos realmente são de uma face. Isso melhorou consideravelmente a precisão do método, pois antes o modelo identificava como faces algumas regiões sem relação alguma com o alvo.

O método acima retorna as faces encontradas em uma imagem. Agora precisamos de um método que retorne o vetor de *embeddings* da *FaceNet* para um rosto. Chamamos esse método de *get_embeddings* e atribuímos a ele os parâmetros *face*, que é uma imagem com dimensão 160x160 (referente a uma face), e *facenet_model*, uma instância do modelo *FaceNet* previamente treinado. O retorno é um array numpy com 128 posições, referentes às 128 características da *FaceNet* (*embeddings*).

```
def get_embeddings(face, facenet_model):

  # Convertendo a imagem para numpy e normalizando para o intervalo
    [0..1]
  img = np.array(face).astype('float32')/255

  # Expandindo a dimensão da imagem para adequá-la a entrada da FaceNet
  # O shape deve ficar (1, 160, 160)
  input = np.expand_dims(img, axis=0)

  # Fazendo a predição do modelo para extrair os embeddings da imagem
  embedding = facenet_model.predict(input)[0]

  return embedding
```

Código Fonte 2.8: Método responsável por extrair as características de uma face, usando a *FaceNet*.

Agora precisamos de um método para receber um vetor de características (*embeddings*) e retornar a classificação desses dados quanto ao uso da máscara, utilizando o classificador treinado. Chamamos esse método de *get_label*, passando os parâmetros *embedding*, um vetor de características de uma face, representado por um array numpy com 128 valores, e *mlp_model*, o modelo treinado do classificador *MLP*. O retorno é apenas uma variável label, que apresenta valor 0 (sem máscara) ou 1 (com máscara).

```
def get_label(embedding, mlp_model):

    # Expandindo dimensão do array para adequá-lo a entrada do
    classificador
    # A dimensão deve ficar (1, 128)
    embedding = np.expand_dims(embedding, axis=0)

    # Realizando a predição da probabilidade de acerto para cada classe
    proba = mlp_model.predict_proba(embedding)

    # Extraindo do vetor de probabilidades o label que apresentou o
    maior resultado
    label = np.argmax(proba)

    # Caso o modelo tenha menos de 95% de certeza sobre o maior
    resultado, o label predito receberá o valor inverso
```

```
    # Por exemplo, se o modelo não tiver 95% ou mais de certeza sobre
    classificar como label 1, o retorno será label 0
    if proba[0][label] < 0.95:
        label = abs(label-1)

    return label
```

Código Fonte 2.9: Método responsável por classificar o vetor de características de uma face quanto ao uso de máscara, através do *MLP* treinado.

Analisando o código acima, é possível perceber que também há um limiar de confiança, assim como na detecção de faces. O que fizemos foi uma condição que permite que a saída da classificação seja 1 ou 0 apenas se o modelo tiver mais de 95% de certeza sobre isso. Caso a probabilidade de acerto seja menor que esse limiar, a saída prevista será o contrário do que o modelo previu. Essa condição foi necessária para melhorar a precisão dos resultados, já que algumas vezes o modelo errava a classificação porque tinha uma confiança muito baixa sobre a classe prevista, mas retornava essa previsão por ser a maior entre as probabilidades.

O último método nós chamamos de *mark_points_in_frame*, que recebe um frame (imagem), executa o *get_faces* para extrair as faces dessa imagem, o *get_embeddings* para extrair as características das faces e o *get_label* para classificar as características em 0 (sem máscara) ou 1 (com máscara). O método também faz as marcações no frame através da OpenCV, desenhando um retângulo em torno das faces, colocando o label da face em cima do retângulo e inserindo um contador de pessoas sem máscara.

```
def mark_points_in_frame(frame):

    # Transformando a imagem em array numpy
    img = np.asarray(frame)

    # Extraindo faces da imagem
    faces = get_faces(img)

    # Iniciando contador responsável por marcar quantas pessoas sem má
    scara existem na imagem
    no_mask_count = 0

    # Percorrendo a lista de faces identificadas
    for face in faces:

        # Extraindo os pontos da face
        x1 = face['x1']
        x2 = face['x2']
        y1 = face['y1']
        y2 = face['y2']

        # Extraindo os embeddings da face
        emb = get_embeddings(face['face'], facenet_model)

        # Classificando a face em label 0 (maskon) ou label 1 (maskoff)
        label = get_label(emb, mlp_model)

```

```
        # Caso uma pessoa sem máscara seja identificada, o contador é incrementado
        if not label: no_mask_count += 1

        # Configurando o tipo e o tamanho da fonte para iniciar as marcações na imagem
        font = cv.FONT_HERSHEY_TRIPLEX
        font_scale = 0.5

        # Desenhando um retângulo em torno da face
        img = cv.rectangle(img, (x1, y1), (x2, y2), color[label], 2)

        # Parâmetros do método retangle: imagem, coordenada de início, coordenada de fim, cor e grossura das linhas

        # Escrevendo a classificação MASK ou NO MASK em cima do retângulo desenhado
        # Essa descrição é baseada no label
        cv.putText(img, label_description[label], (x1, y1-10), font, fontScale=font_scale,
                   color=color[label], thickness=1)

        # Escrevendo no topo do frame um informativo sobre quantas pessoas estão sem máscara na imagem
        cv.putText(img, f'People without mask: {no_mask_count}', (15, 15), font, fontScale=0.6,
                   color=(0, 0, 0), thickness=1)

        # Parâmetros do método putText: imagem, string a ser escrita, posição do texto no frame, estilo da fonte, tamanho da fonte, cor e grossura da fonte

    return img
```

Código Fonte 2.10: Método responsável por executar os métodos anteriores e desenhar na imagem a caixa delimitadora de cada face, bem como o label associado.

Finalmente, podemos executar o código que abrirá nossa webcam e realizará o processo de detecção de máscara em tempo real. É normal se os frames estiverem sendo renderizados lentamente, isso depende do poder computacional da sua máquina, pois o custo das predições dos modelos para cada frame influencia no tempo de renderização. Se sua máquina tiver uma GPU, a transmissão será bem mais fluida.

```
print('Iniciando captura...\n')

# Instanciando um objeto VideoCapture, para selecionar sua webcam
# Também é possível renderizar um vídeo pronto, basta você passar o caminho desse arquivo no lugar do parâmetro 0
vid = cv.VideoCapture(0)

print('Captura iniciada!')

# A captura dos frames através da sua webcam ou vídeo será feita até que você pressione a tecla 'q'
while (True):
```

```
    # Lendo a imagem e extraindo o frame
    # O método read() retorna dois resultados. O primeiro diz se a
    captura do frame foi feita com sucesso ou não, enquanto o segundo
    entrega o frame capturado.
    _, frame = vid.read()

    # Fazendo a detecção de máscara e as marcações nas faces
    identificadas no frame
    detec = mark_points_in_frame(frame)

    # Exibindo o frame resultante do método anterior
    cv.imshow('frame', detec)

    # Condição de parada do loop: pressione a tecla 'q'
    if cv.waitKey(1) & 0xFF == ord('q'):
        break

# Fechando o arquivo de vídeo ou dispositivo de captura
vid.release()

# Fechando as janelas abertas
cv.destroyAllWindows()

# Apagando o objeto da memória
del (vid)
```

Código Fonte 2.11: Captura da webcam com a OpenCV e detecção de máscara através dos métodos anteriormente definidos.

A Figura 2.5 mostra exemplos do algoritmo aplicado à imagens com e sem máscara, além de com e sem óculos.

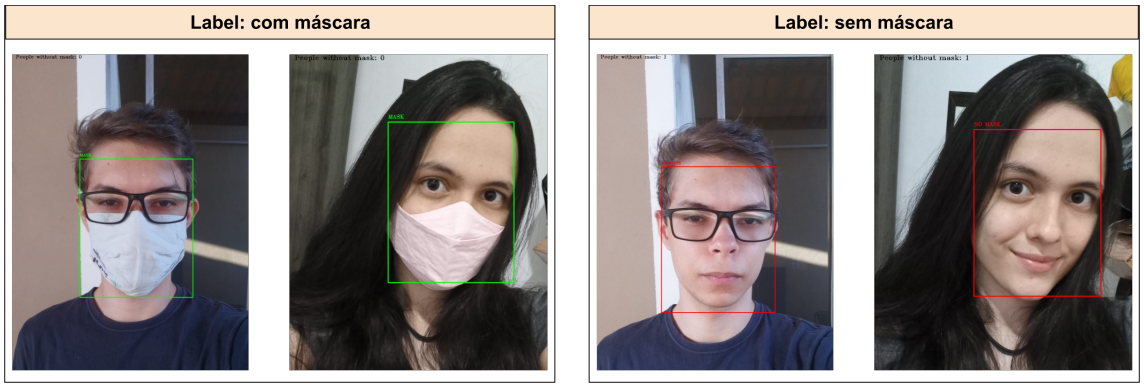

Figura 2.5: Exemplos do método de detecção nos cenários com e sem máscara e com e sem óculos.

2.4. Considerações Finais

Neste capítulo, construímos uma aplicação para detecção de máscaras usando métodos amplamente adotados na literatura para diversas finalidades em Visão Computacional.

No decorrer do capítulo, introduzimos os conceitos de Visão Computacional e *Deep Learning*, adentrando na explicação de cada uma das tecnologias utilizadas na detecção, a saber: *MTCNN*, *FaceNet* e *MLP*.

Embora existam várias melhorias possíveis a serem feitas no projeto para que o mesmo possa integrar um sistema real, os algoritmos de detecção de máscara podem ser utilizados em sistemas embarcados presentes em câmeras de supermercados, shoppings, aeroportos, hospitais, escolas e outros ambientes que acomodam um grande número de pessoas. Além disso, nós apresentamos uma aplicação robusta que pode agregar o portfólio dos alunos e instigar seu interesse quanto a infinidade de aplicações que o ecossistema de Inteligência Artificial nos permite desenvolver.

Destacamos algumas sugestões de melhoria na aplicação:

- Adicionar mais imagens à base de treino, imagens com pessoas em distância maior e imagens capturadas pela própria OpenCV, alternando entre o uso e o não uso de máscara. Com a base de treino incrementada, o classificador deverá ser treinado novamente;
- Utilizar um algoritmo de otimização de hiper parâmetros no classificador, como o Grid Search;
- Variar o limiar de confiança que atribuímos na classificação até encontrar um novo valor ideal;
- Avaliar abordagens alternativas de extração e classificação de características;
- Avaliar abordagens alternativas de detecção facial.

Referências

[Ballard e Brown 1982] Ballard, D. H. e Brown, C. M. (1982). Computer vision. englewood cliffs. *J: Prentice Hall*.

[de Milano e Honorato 2014] de Milano, D. e Honorato, L. B. (2014). Visao computacional.

[Gradilla 2020] Gradilla, R. (2020). Multi-task cascaded convolutional networks (mtcnn) for face detection and facial landmark alignment. Disponível em: https://medium.com/@iselagradilla94/multi-task-cascaded-convolutional-networks-mtcnn-for-face-detection-a Acessado em: 13 de setembro de 2021.

[Haykin et al. 2009] Haykin, S. S., Haykin, S. S., Haykin, S. S., e Haykin, S. S. (2009). *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA:.

[Jose et al. 2019] Jose, E., Manikandan, G., T P, M. H., e M H, S. (2019). Face recognition based surveillance system using facenet and mtcnn on jetson tx2.

[Karpathy 2014] Karpathy, A. (2014). Convolutional neural networks. http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/.

[Kumar 2021] Kumar, D. (2021). Introduction to facenet: A unified embedding for face recognition and clustering. Disponível em: https://medium.com/analytics-vidhya/introduction-to-facenet-a-unified-embedding-for-face-recognition-and- Acessado em: 16 de setembro de 2021.

[LeCun et al. 1998] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Liu et al. 2017] Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., e Alsaadi, F. E. (2017). A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26.

[Ramchoun et al. 2016] Ramchoun, H., Amine, M., Janati Idrissi, M. A., Ghanou, Y., e Ettaouil, M. (2016). Multilayer perceptron: Architecture optimization and training. *International Journal of Interactive Multimedia and Artificial Inteligence*, 4:26–30.

[Ramchoun et al. 2017] Ramchoun, H., Idrissi, M. A. J., Ghanou, Y., e Ettaouil, M. (2017). Multilayer perceptron: Architecture optimization and training with mixed activation functions. Em *Proceedings of the 2nd International Conference on Big Data, Cloud and Applications*, BDCA'17, New York, NY, USA. Association for Computing Machinery.

[Schroff et al. 2015] Schroff, F., Kalenichenko, D., e Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering. Em *Proceedings of the IEEE conference on computer vision and pattern recognition*, páginas 815–823.

[WHO 2021] WHO, W. H. O. (2021). Coronavirus disease (covid-19) outbreak situation. Disponível em: https://www.who.int/emergencies/diseases/novel-coronavirus-2019. Acessado em: 12 de setembro de 2021.

[Zhang et al. 2016] Zhang, K., Zhang, Z., Li, Z., e Qiao, Y. (2016). Joint face detection and alignment using multitask cascaded convolutional networks. *IEEE Signal Processing Letters*, 23(10):1499–1503.