ARTIGO

A pandemia de COVID-19 afetou rapidamente nossa vida cotidiana, exigindo inúmeras adaptações no estilo de vida das interações pessoais. Usar uma máscara protetora de rosto tornou-se, portanto, uma dessas adaptações necessárias no dia a dia, sendo ainda, na maioria dos países, obrigatória a sua utilização em locais públicos, para fins de controle da disseminação do vírus. Portanto, a detecção de máscara facial tornou-se uma tarefa crucial para ajudar a sociedade global.

Dessa forma, decidimos elaborar um detector de máscaras faciais em fotos, programado em linguagem Python por meio do Google Colab. Este artigo apresenta uma abordagem simplificada para atingir esse objetivo usando alguns pacotes básicos de aprendizado de máquina, como TensorFlow, Keras, OpenCV e Scikit-Learn - pacotes da linguagem Python. O método proposto detecta corretamente o rosto da imagem e, em seguida, identifica se possui ou não uma máscara.

Elaboramos um passo a passo para você compreender melhor como todo esse trabalho foi realizado. Vamos lá?

1) Dados de treinamento

Para fazer o nosso programa “aprender” a detectar a presença de máscaras ou não em fotos, é necessário utilizar o que chamamos de “conjunto” ou “dados” de treinamento, os quais, nesse caso, seria um grande volume de fotos, em que há instâncias de rostos com máscara, e sem máscara.

Nesse caso, utilizamos o dataset Face Mask Detection, que consiste em 11.792 imagens, separadas em três diretórios:

- Diretório de Treino: 10.000 imagens;
- Diretório de Teste: 992 imagens;
- Diretório de Validação: 800 imagens.

2) Importar os pacotes necessários para o funcionamento do programa:

```
import os # Manipulação de pastas
import cv2 as cv # Pacote de visão computacional
import random # Criação de aleatoriedades
import numpy as np # Álgebra linear e matemática vetorizada
import matplotlib.pyplot as plt # Criação de gráficos
import warnings
warnings.filterwarnings('ignore')
```

3) Importar e inicializar o Dataset de treinamento:

```
from google.colab import files
files.upload() # Dando upload no arquivo kaggle
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/
!chmod 600 ~/.kaggle/kaggle.json
!ls ~/.kaggle
!mkdir "mask_det" # Nomeando o arquivo da forma que desejar
```

```
!kaggle datasets download ashishjangra27/face-mask-12k-images-dataset
-p "/content/mask_det" --unzip # Dando unzip no arquivo baixado

for dirname, _, filenames in os.walk('/content/mask_det'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

dirName = '/content/mask_det/Face Mask Dataset/'
```

No nosso caso, em que utilizamos um Dataset disponível no Kaggle, nós o inicializamos automaticamente no Google Colab, por meio do código acima. Para saber mais sobre essa etapa, vale a pena dar uma lida neste artigo, que ensina o passo a passo de como fazer downloads de datasets do Kaggle automaticamente no Colab.

Depois de Inicializar o Dataset, vamos passar para a etapa de inicialização da nossa Rede Neural, que será a responsável por fazer as predições esperadas para o funcionamento do nosso detector de uso de máscaras!

4) Importar os pacotes necessários para o funcionamento da rede:

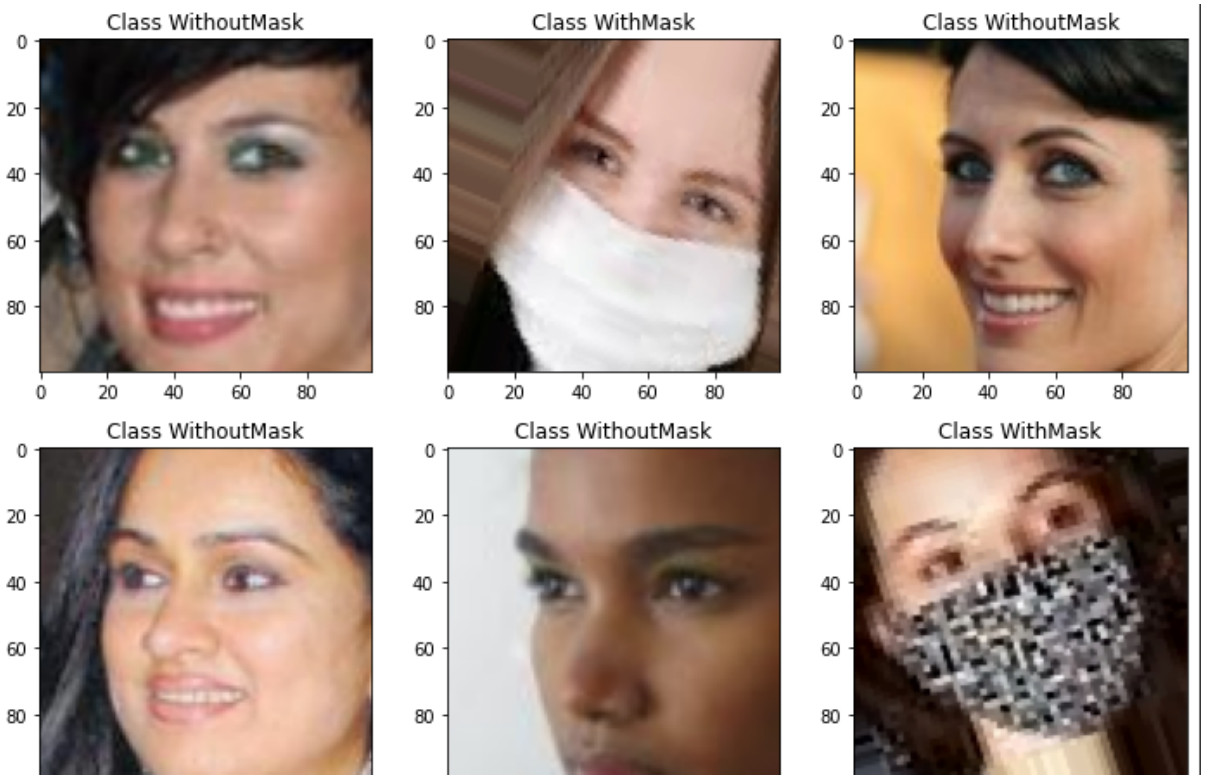
```
from tensorflow.keras.applications.vgg19 import VGG19, preprocess_input
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.preprocessing import image_dataset_from_directory
from sklearn.model_selection import train_test_split
from keras import Sequential
from keras.layers import Flatten, Dense
import tensorflow as tf
from tensorflow.keras.callbacks import EarlyStopping
```

5) Fazer o pré-processamento dos dados de teste, validação e treinamento, por meio do ImageDataGenerator:

```
train_datagen = ImageDataGenerator(
                    rescale=1./255,
                    horizontal_flip=True,
                    rotation_range=20,
                    width_shift_range=0.2,
                    height_shift_range=0.2)
datagen = ImageDataGenerator(rescale=1./255)

train_gen = train_datagen.flow_from_directory(directory=dirName +
'Train',
                                              batch_size=32,
                                              target_size=(100, 100))
test_gen = train_datagen.flow_from_directory(directory=dirName +
'Test',
                                              batch_size=32,
                                              target_size=(100, 100))
val_gen = train_datagen.flow_from_directory(directory=dirName +
'Validation',
                                              batch_size=32,
                                              target_size=(100, 100))
```

Para melhor visualização dos dados de treinamento, plotamos algumas amostras, juntamente com as suas respectivas classes

```
plt.rcParams['figure.figsize'] = (10,10)


for img, label in train_df.take(1):
  for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(img[i].numpy().astype('uint8'))
    clas = train_df.class_names
    plt.title(f'Class {clas[int(label[i])]}')

plt.tight_layout()
```

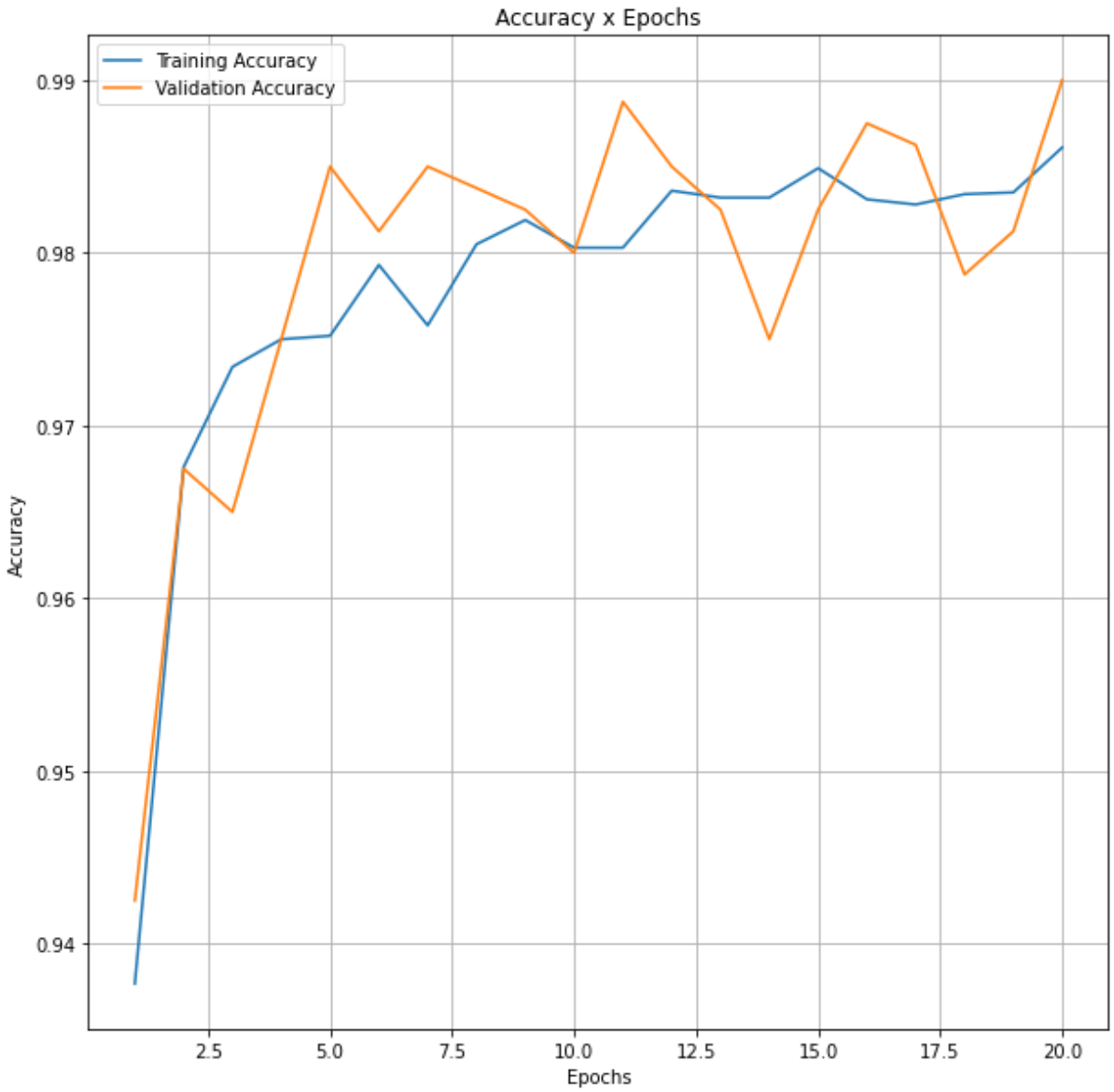


6) Hora de construir a nossa Rede Neural!

Para esse exercício, utilizamos uma rede neural pré-treinada por meio de transfer learning, a VGG19. Definimos a rede aqui:

```
vgg19 =
VGG19(weights='imagenet',include_top=False,input_shape=(100,100,3))
for layer in vgg19.layers:
    layer.trainable = False
model = Sequential()
model.add(vgg19)
model.add(Flatten())
model.add(Dense(2,activation='sigmoid'))
model.compile(optimizer="adam",loss="binary_crossentropy",metrics
="accuracy")
```

O resultado da implementação da rede, em termos de Acurácia, foi o seguinte:



Por meio desse gráfico, podemos ver que, apesar de algumas distorções das retas de treinamento e validação, os resultados de ambas tendem a ficar parecidos. Além disso, os dados de teste foram expostos à rede, o que resultou em uma acurácia de 99,19%, o que descarta a possibilidade de termos obtido uma boa acurácia com overfitting ou underfitting. Logo, por exemplo, para 10000 imagens, 9919 foram classificadas corretamente pela rede, o que é um resultado muito satisfatório para esse exercício.

Vale lembrar que é possível aumentar ou diminuir a acurácia de treinamento, validação e teste por meio da manipulação dos parâmetros da rede. Logo, adicionar/remover camadas, alterar o número de neurônios em cada camada, testar diferentes formas de pré processamento dos dados e alterar o número de épocas/learning rate/batch_size pode causar mudanças significativas - positivas ou negativas - nas suas predições!

Deixamos aqui, ao final desse artigo, o link do nosso GitHub, com o dataset e o notebook. Obrigada pela atenção na leitura, e bons estudos! (: