### Relatório “20 - Prática: Reconhecimento de Emoções com TensorFlow 2.0 e Python (III)”

O reconhecimento de emoções é usado na inteligência artificial para que o algoritmo aja mais naturalmente, por exemplo em chat-bots de atendimento e assistentes virtuais, se você estiver triste, a IA irá mudar o tratamento como um humano. O ramo que estuda e desenvolve isso é o da Computação Afetiva.

### Fazendo detecções com modelos pré treinados:

No Google Colab, algumas funções de algumas bibliotecas não funcionam por padrão, nesses casos usamos:

from google.colab.patches import (biblioteca) (função)

Ex: from.google.colab.patches import cv2 imshow

Será usado uma versão específica do TensorFlow: %tensorflow\_version 2.x

import tensorflow

from tensorflow.keras.models import load\_model

from tensorflow.keras.preprocessing.image import img\_to\_array

tensorflow.\_\_version\_\_

Nesse projeto vamos usar arquivos ZIP direto do google drive:

from google.colab import drive

drive.mount('/content/gdrive')

Para extrairmos o zip:

path = "/content/gdrive/My Drive/Material.zip"

zip\_object = zipfile.ZipFile(file = path, mode = "r")

zip\_object.extractall('./')

zip\_object.close

No OpenCV temos a classe CascadeClassifier que faz as detecções de objetos, foi passado um XML com os parâmetros para detectar faces.

ROI (Região de interesse):

Primeiro detectamos as faces:

original = imagem.copy()

faces = face\_detection.detectMultiScale(original, scaleFactor = 1.1,

minNeighbors = 3, minSize = (20,20))

Ao chamar a variável faces, ela retornará a posição X, Y onde terá uma face e o tamanho da área onde contém o rosto:

array([[162, 40, 128, 128]], dtype=int32)

Será uma matriz contendo vetores referentes a cada rosto detectado, nesse caso temos apenas um vetor pois detectou apenas um rosto.

Tendo apenas uma dimensão de cores, a nossa área de interesse terá apenas duas dimensões, facilitando o trabalho:

cinza = cv2.cvtColor(original, cv2.COLOR\_BGR2GRAY)

Após convertermos, finalmente vamos definir nossa área de interesse:

roi = cinza[40:40 + 128, 162:162 + 128]

Se chamarmos imshow com essa área de interesse, mostrará apenas o rosto que será detectado.

128x128 ainda é um tamanho desnecessariamente grande para nosso objetivo, então vamos redimensionar a imagem para 48x48:

roi = cv2.resize(roi, (48, 48))

roi.dtype

dtype('uint8')

O tipo de dados padrão é “8-bit unsigned int”, valores de 0 a 255, vamos converter para Float para que possamos ter valores de 0 a 1 que indicam percentuais:

roi = roi.astype('float')

roi = roi / 255

E por fim vamos converter a imagem em um vetor para que o TensorFlow possa lê-la:

roi = img\_to\_array(roi)

roi = np.expand\_dims(roi, axis = 0)

Essa ultima linha adiciona mais uma dimensão de linha onde

roi.shape

(1, 48, 48, 1): quantidade de imagens, y, x, canais de cores

E finalmente vamos fazer o algoritmo predizer as emoções da imagem, retornando uma probabilidade para cada emoção:

preds = classificador\_emocoes.predict(roi)[0]

Chamando preds:

array([1.6802111e-05, 5.7679777e-09, 1.1206568e-05, 9.9066460e-01,

1.8033772e-05, 1.8518280e-05, 9.2708217e-03], dtype=float32)

Pegaremos a posição do maior valor desse vetor:

preds.argmax()

3

E vendo qual o índice 3 do vetor de expressões, mostrou que o dono do rosto estava feliz:

label = expressoes[preds.argmax()]

label

'Feliz'

### Criando uma rede neural convolucional para detectar as emoções, arquitetura 1:

Para isso vamos usar o dataset fer2013, já vamos usar ele já extraído em formato CSV. Dele, vamos pegar apenas os valores dos pixels de cada imagem.

pixels = data['pixels'].tolist()

Cada objeto dessa lista representa uma imagem, em formato string onde cada valor de pixel está separado do outro por espaço. Por conta disso, vamos converter em vetor de inteiros:

largura, altura = 48, 48

faces = []

amostras = 0

for pixel\_sequence in pixels:

face = [int(pixel) for pixel in pixel\_sequence.split(' ')]

face = np.asarray(face).reshape(largura, altura)

faces.append(face)

if (amostras < 10):

cv2\_imshow(face)

amostras += 1

O segundo for já converte os pixels em inteiros considerando que estão separados por espaço, logo abaixo converte a lista em vetor e reformula com as dimensões da largura e altura de imagem, em seguida eles são adicionados em uma lista “faces”, ou seja, a lista “faces” contém vetores que representam as imagens, cada um deles tem 48 outros vetores dentro que representam as linhas com 48 valores cada que representam as colunas. Para testarmos se a conversão deu certo, podemos usar o imshow para mostrar o vetor “face” como imagem, mostrando as imagens vemos que deu certo.

Em seguida converteremos a lista “faces” em vetor:

faces = np.asarray(faces)

Um vetor de vetores contendo outros vetores, parece uma grande matrioska.

faces.shape

(35887, 48, 48)

Se consultarmos o formato de faces com shape, dará que temos 35887 itens, cada item contendo 48 vetores, com 48 valores cada, ou seja, 35887 imagens de resolução 48x48.

Após isso, vamos adicionar mais uma dimensão para ficar no formato do TensorFlow, onde temos como shape a quantidade de imagens, altura, largura e quantos canais de cores.

faces = np.expand\_dims(faces, -1)

Sendo esse -1 indicando que queremos essa nova dimensão como ultimo item.

Fazendo o shape desse novo vetor confirmamos o formato apropriado para o TensorFlow:

faces.shape

(35887, 48, 48, 1)

E por fim vamos normalizar os valores dos pixels para um range de 0 a 1 para facilitar no processamento:

def normalizar(x):

x = x.astype('float32')

x = x / 255.0

return x

faces = normalizar(faces)

Agora para podermos comparar as probabilidades de cada emoção de acordo com a rede neural, vamos converter os valores das emoções para dummies:

emocoes = pd.get\_dummies(data['emotion']).values

Todos esses passos anteriores foi o pré processamento dos dados, agora vamos mexer com a rede neural.

Para criarmos a rede neural convolucional vamos importar umas funções:

from sklearn.model\_selection import train\_test\_split

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, Activation, Flatten

from tensorflow.keras.layers import Conv2D, MaxPooling2D, BatchNormalization

from tensorflow.keras.losses import categorical\_crossentropy

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.regularizers import l2

from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint

from tensorflow.keras.models import load\_model

from tensorflow.keras.models import model\_from\_json

Há várias funções que já usamos alguma vez aqui, mas há algumas inéditas bem interessantes tipo:

from tensorflow.keras.regularizers import l2

Usado para diminuirmos o overfitting e aumentar o desempenho da rede neural, ele adiciona uma penalidade caso o erro da rede esteja alto. Para mais detalhes consultar Regularizers na documentação do Keras.

from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping, ModelCheckpoint

Callbacks são funções que permitem utilizar certos recursos durante o treinamento do modelo.

EarlyStopping faz a parada automática a partir do momento que a rede não consegue se adaptar muito.

ModelCheckpoint salva o melhor modelo de acordo com o desempenho.

Nesse modelo não vamos usar apenas conjuntos de treino e teste, vamos também usar mais uma parte para validação.

Essa base de validação é usada toda vez que encerrar uma época de treinamento para validação em tempo real, ela é extraída da base de treinamento.

Recapitulando alguns termos:

Batch Size: quantidade de itens do conjunto de treinamento usados a cada atualização dos pesos.

num\_features: Número de filtros/mapas de características da rede.

De resto criamos as camadas convolucionais de forma similar ao que já vimos com as variáveis adicionais:

num\_features = 64

num\_labels = 7

batch\_size = 64

epochs = 100

width, height = 48, 48

model = Sequential()

model.add(Conv2D(num\_features, kernel\_size=(3,3), activation='relu',

input\_shape=(width, height, 1), data\_format = 'channels\_last',

kernel\_regularizer = l2(0.01)))

model.add(Conv2D(num\_features, kernel\_size=(3,3), activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2,2), strides=(2,2)))

model.add(Dropout(0.5))

model.add(Conv2D(2\*num\_features, kernel\_size=(3,3), activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(Conv2D(2\*num\_features, kernel\_size=(3,3), activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2,2), strides=(2,2)))

model.add(Dropout(0.5))

model.add(Conv2D(2\*2\*num\_features, kernel\_size=(3,3), activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(Conv2D(2\*2\*num\_features, kernel\_size=(3,3), activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2,2), strides=(2,2)))

model.add(Dropout(0.5))

model.add(Conv2D(2\*2\*2\*num\_features, kernel\_size=(3,3), activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(Conv2D(2\*2\*2\*num\_features, kernel\_size=(3,3), activation='relu', padding='same'))

model.add(BatchNormalization())

model.add(MaxPooling2D(pool\_size=(2,2), strides=(2,2)))

model.add(Dropout(0.5))

model.add(Flatten())

model.add(Dense(2\*2\*2\*num\_features, activation='relu'))

model.add(Dropout(0.4))

model.add(Dense(2\*2\*num\_features, activation='relu'))

model.add(Dropout(0.4))

model.add(Dense(2\*num\_features, activation='relu'))

model.add(Dropout(0.5))

model.add(Dense(num\_labels, activation = 'softmax'))

model.summary()

Compilação do modelo:

model.compile(loss = 'categorical\_crossentropy',

optimizer = Adam(lr = 0.001, beta\_1=0.9, beta\_2=0.999, epsilon=1e-7),

metrics = ['accuracy'])

beta\_1 e beta\_2: Taxa de decaimento exponencial.

episolon: Valor muito pequeno usado para evitar divisões por zero na implementação do otimizador.

Arquivos H5 salvam os pesos, JSON salvam a estrutura da rede neural.

lr\_reducer = ReduceLROnPlateau(monitor='val\_loss', factor = 0.9, patience=3, verbose = 1)

early\_stopper = EarlyStopping(monitor='val\_loss', min\_delta=0, patience = 8, verbose = 1, mode = 'auto')

checkpointer = ModelCheckpoint(arquivo\_modelo, monitor='val\_loss', verbose = 1, save\_best\_only=True)

monitor: Definimos a variável que a função verificará, nesse caso é o erro do conjunto de validação.

Plateau: Ponto da rede neural que não temos mais progresso, ou seja, quando o erro não está diminuindo e nem a acurácia aumentando.

factor: percentual do learning rate que se tornará o novo learing rate caso não haja melhorias.

patience: Tolerância máxima de épocas sem melhoria, no caso do lr\_reducer, se rodar 3 épocas e não houver melhoria, a função entrará em ação.

verbose: Se for 1, diz para mostrar se a função foi executada.

Fato confirmado: No módulo 18, o modelo com 10 dimensões latentes teve um resultado superior ao com 50, e em outros módulos anteriores acontecia de certas épocas antes de terminar, ter uma acurácia maior e menor erro. As vezes, muitas épocas faz o algoritmo acabar estragando os pesos e piorando, por conta disso é importante usar o early\_stopper não apenas para economizar recursos e tempo.

Salvando a arquitetura do modelo em um JSON:

model\_json = model.to\_json()

with open(arquivo\_modelo\_json, 'w') as json\_file:

json\_file.write(model\_json)

Treinando o modelo:

history = model.fit(np.array(X\_train), np.array(y\_train),

batch\_size = batch\_size,

epochs = epochs,

verbose = 1,

validation\_data = (np.array(X\_val), np.array(y\_val)),

shuffle=True,

callbacks=[lr\_reducer, early\_stopper, checkpointer])

Muito similar ao visto anteriormente, com a diferença que aqui definimos os callbacks.

A acurácia final do código é verificada usando a base de testes como visto anteriormente, usamos a função evaluate para saber o erro e acurácia finais:

scores = model.evaluate(np.array(X\_test), np.array(y\_test), batch\_size = batch\_size)

print('Acurácia: ' + str(scores[1]))

print('Erro: ' + str(scores[0]))

Acurácia: 0.61799943

Erro: 1.1469147179611017

Para verificarmos a matriz de confusão, antes é bom salvarmos os dados usados anteriormente no formato do NumPy:

np.save('mod\_xtest', X\_test)

np.save('mod\_ytest', y\_test)

### Usando o modelo criado e salvo anteriormente:

Agora vamos carregar os dados para usarmos na matriz de confusão, isso já pode ser feito em outro ambiente pois já são o modelo e pesos salvos, sem ligação com o código da CNN:

true\_y = []

pred\_y = []

x = np.load('mod\_xtest.npy')

y = np.load('mod\_ytest.npy')

Carregar o modelo:

json\_file = open(arquivo\_modelo\_json, 'r')

loaded\_model\_json = json\_file.read()

json\_file.close()

loaded\_model = model\_from\_json(loaded\_model\_json)

loaded\_model.load\_weights(arquivo\_modelo)

Os pesos “arquivo\_modelo” foram salvos automaticamente com a função de callback.

Agora vamos fazer as previsões em cima do modelo salvo:

y\_pred = loaded\_model.predict(x)

A previsão foi feita para todas as imagens de teste.

Serão criadas duas variáveis que receberão as previsões e os dados verdadeiros em formato de lista:

yp = y\_pred.tolist()

yt = y.tolist()

count = 0

for i in range(len(y)):

yy = max(yp[i])

yyt = max(yt[i])

pred\_y.append(yp[i].index(yy))

true\_y.append(yt[i].index(yyt))

if (yp[i].index(yy) == yt[i].index(yyt)):

count += 1

acc = (count / len(y)) \* 100

print('Acurácia no conjunto de teste: ' + str(acc))

Acurácia no conjunto de teste: 61.79994427417108

Esse código acima compara as maiores probabilidades previstas com os valores reais e calcula a acurácia manualmente.

np.save('truey\_mod01', true\_y)

np.save('predy\_mod01', pred\_y)

E com isso salvamos os valores reais e os previstos após carregar o modelo salvo, que usaremos na matriz de confusão.

cm = confusion\_matrix(y\_true, y\_pred)

expressoes = ['Raiva', 'Nojo', 'Medo', 'Feliz', 'Triste', 'Surpreso', 'Neutro']

titulo = 'Matriz de Confusão'

Criamos a matriz de confusão, é possível vê-la chamando pelo nome dela (cm), mas podemos vê-la de forma mais detalhada usando o Matplotlib:

import itertools

plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)

plt.title(titulo)

plt.colorbar()

tick\_marks = np.arange(len(expressoes))

plt.xticks(tick\_marks, expressoes, rotation = 45)

plt.yticks(tick\_marks, expressoes)

fmt = 'd'

thresh = cm.max() / 2.

for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):

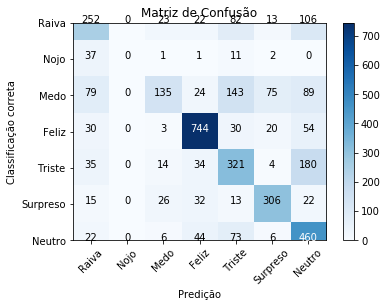
plt.text(j, i, format(cm[i, j], fmt), horizontalalignment='center', color='white' if cm[i,j] > thresh else 'black')

plt.ylabel('Classificação correta')

plt.xlabel('Predição')

plt.savefig('matriz\_confusao\_mod01.png')

Saída:



### Outras arquiteturas:

As demais arquiteturas usam estruturas bem similares, com diferença nos parâmetros como quantidade de filtros, batch\_size, kernel\_initializer, função de ativação, Dropout e quantidade de camadas. Podemos se basear em outras arquiteturas já feitas desde que seja permitido.

### Comparando modelos salvos:

Podemos comparar modelos salvos para saber qual o melhor, o seguinte código faz isso, usa o mesmo dataset e porções salvos na primeira arquitetura.

from tensorflow.keras.models import load\_model

import operator

# Nome dos modelos que serão comparados

arquivos\_modelos = ["modelo\_01\_expressoes.h5", "modelo\_02\_expressoes.h5", "modelo\_03\_expressoes.h5", "modelo\_04\_expressoes.h5", "modelo\_05\_expressoes.h5"]

modelos = {}

x\_test = np.load('Material/mod\_xtest.npy')

y\_test = np.load('Material/mod\_ytest.npy')

for modelo in arquivos\_modelos:

model = load\_model('Material/' + modelo)

# Calcula a acurácia do modelo obtida ao testar na base de teste

scores = model.evaluate(np.array(x\_test), np.array(y\_test), batch\_size=64)

print("---"+ str(modelo) +"---")

print("Perda/Loss: " + str(scores[0]))

print("Acurácia: " + str(scores[1]))

modelos[modelo] = str(scores[1])

print("\n")

Saída:

---modelo\_01\_expressoes.h5---

Perda/Loss: 1.0704169079150685

Acurácia: 0.63917524

---modelo\_02\_expressoes.h5---

Perda/Loss: 1.011896936947936

Acurácia: 0.6411257

---modelo\_03\_expressoes.h5---

Perda/Loss: 1.073169717726584

Acurácia: 0.6308164

---modelo\_04\_expressoes.h5---

Perda/Loss: 1.1690520508337832

Acurácia: 0.61604905

---modelo\_05\_expressoes.h5---

Perda/Loss: 1.8206109304420128

Acurácia: 0.24547228

Em casos que testamos vários modelos, podemos ordenar pela acurácia para ver quais os melhores, o do índice zero será o melhor:

# Ordena em ordem decrescente os modelos com base no valor da acurácia

order\_modelos = sorted(modelos.items(), key=operator.itemgetter(1), reverse=True)

print(order\_modelos)

('modelo\_02\_expressoes.h5', '0.6411257')

O seguinte código carrega o modelo, roda e mostra em uma foto real qual a emoção e as probabilidades:

imagem = cv2.imread("Material/testes/teste\_gabriel.png")

cascade\_faces = 'Material/haarcascade\_frontalface\_default.xml'

caminho\_modelo = 'Material/' + str(order\_modelos[0][0]) #ver saida acima

face\_detection = cv2.CascadeClassifier(cascade\_faces)

classificador\_emocoes = load\_model(caminho\_modelo, compile=False)

expressoes = ["Raiva", "Nojo", "Medo", "Feliz", "Triste", "Surpreso", "Neutro"]

from tensorflow.keras.models import load\_model

from tensorflow.keras.preprocessing.image import img\_to\_array

# Carrega o modelo

face\_detection = cv2.CascadeClassifier(cascade\_faces)

classificador\_emocoes = load\_model(caminho\_modelo, compile=False)

expressoes = ["Raiva", "Nojo", "Medo", "Feliz", "Triste", "Surpreso", "Neutro"]

original = imagem.copy()

faces = face\_detection.detectMultiScale(original,scaleFactor=1.1,minNeighbors=3,minSize=(20,20))

cinza = cv2.cvtColor(original, cv2.COLOR\_BGR2GRAY)

if len(faces) > 0:

for (fX, fY, fW, fH) in faces:

roi = cinza[fY:fY + fH, fX:fX + fW]

roi = cv2.resize(roi, (48, 48))

roi = roi.astype("float") / 255.0

roi = img\_to\_array(roi)

roi = np.expand\_dims(roi, axis=0)

preds = classificador\_emocoes.predict(roi)[0]

print(preds)

emotion\_probability = np.max(preds)

label = expressoes[preds.argmax()]

cv2.putText(original, label, (fX, fY - 10), cv2.FONT\_HERSHEY\_SIMPLEX, 0.65, (0, 0, 255), 2, cv2.LINE\_AA)

cv2.rectangle(original, (fX, fY), (fX + fW, fY + fH),(0, 0, 255), 2)

else:

print('Nenhuma face detectada')

probabilidades = np.ones((250, 300, 3), dtype="uint8") \* 255

# Mostra gráfico apenas se detectou uma face

if len(faces) == 1:

for (i, (emotion, prob)) in enumerate(zip(expressoes, preds)):

# Nome das emoções

text = "{}: {:.2f}%".format(emotion, prob \* 100)

w = int(prob \* 300)

cv2.rectangle(probabilidades, (7, (i \* 35) + 5),

(w, (i \* 35) + 35), (200, 250, 20), -1)

cv2.putText(probabilidades, text, (10, (i \* 35) + 23),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.45,

(0, 0, 0), 1, cv2.LINE\_AA)

cv2\_imshow(probabilidades)

cv2.imwrite("captura.jpg",original)

cv2.destroyAllWindows()

### Detecç**ões de emoções em vídeos:**

Vamos usar os mesmos modelos que salvamos, para usá-lo em videos terá uma implementação um pouco diferente:

import cv2

import numpy as np

import time

import pandas as pd

import matplotlib.pyplot as plt

from google.colab.patches import cv2\_imshow

import zipfile

import tensorflow

from google.colab import drive

drive.mount('/content/gdrive')

path = "/content/gdrive/My Drive/Videos.zip"

zip\_object = zipfile.ZipFile(file=path, mode="r")

zip\_object.extractall("./")

from tensorflow.keras.models import load\_model

diretorio = 'gdrive/My Drive/Cursos/Deteccao\_Expressoes\_Faciais/' # diretorio do drive onde estão os arquivos do curso

# vamos utilizar o Modelo da Arquitetura 2 pois foi o que se saiu melhor em nossos testes

model = load\_model(diretorio + "modelo\_02\_expressoes.h5")

arquivo\_video = diretorio + "testes/video\_teste06.MOV"

cap = cv2.VideoCapture(arquivo\_video)

conectado, video = cap.read()

print(video.shape) # mostra as dimensões do video

redimensionar = True

# deixe True para reduzir o tamanho do vídeo salvo caso este supere a largura máxima que vamos especificar abaixo.

# para manter o tamanho original deixe False

largura\_maxima = 600 # pixels. define o tamanho da largura (máxima) do vídeo a ser salvo. a altura será proporcional e é definida nos calculos abaixo

# se redimensionar = True então o video que será salvo terá seu tamanho em pixels reduzido SE for maior que a largura\_maxima

if (redimensionar and video.shape[1]>largura\_maxima):

# precisamos deixar a largura e altura proporcionais (mantendo a proporção do vídeo original) para que a imagem não fique com aparência esticada

proporcao = video.shape[1] / video.shape[0]

# para isso devemos calcular a proporção (largura/altura) e usaremos esse valor para calcular a altura (com base na largura que definimos acima)

video\_largura = largura\_maxima

video\_altura = int(video\_largura / proporcao)

else:

video\_largura = video.shape[1]

video\_altura = video.shape[0]

# se redimensionar = False então os valores da largura e altura permanecerão os mesmos do vídeo original

# nome do arquivo de vídeo que será salvo

nome\_arquivo = diretorio+'resultado\_video\_teste06.avi'

# definição do codec

fourcc = cv2.VideoWriter\_fourcc(\*'XVID')

# FourCC é um código de 4 bytes usado para especificar o codec de vídeo. A lista de códigos disponíveis pode ser encontrada no site fourcc.org

# Codecs mais usados: XVID, MP4V, MJPG, DIVX, X264...

# Por exemplo, para salvar em formato mp4 utiliza-se o codec mp4v (o nome do arquivo também precisa possuir a extensão .mp4)

# fourcc = cv2.VideoWriter\_fourcc(\*'mp4v')

# FPS - frames por segundo

fps = 24

# se quiser deixar o video um pouco mais lento pode diminuir o numero de frames por segundo para 20

saida\_video = cv2.VideoWriter(nome\_arquivo, fourcc, fps, (video\_largura, video\_altura))

from tensorflow.keras.preprocessing.image import img\_to\_array

# define se deixa no modo detalhado, que exibirá no canto da tela as barras com as probabilidades de cada emoção

# esse modo é pra ser usado de preferência com vídeos onde tem apenas um rosto em foco

# (não tem problema se aparecer outros rostos menores no fundo, pois abaixo colocamos uma condição para que seja considerado apenas a face com maior área.

# o importante é que não tenha mais de um rosto em foco disputando a atenção da cena)

unica\_face = True

# se deixar False então o programa vai detectar a emoção de todas as faces na imagem e não vai exibir os gráficos com as probabilidades no canto

haarcascade\_faces = diretorio + 'haarcascade\_frontalface\_alt.xml' # arquivo haarcascade

# define os tamanhos para as fontes

fonte\_pequena, fonte\_media = 0.4, 0.7

fonte = cv2.FONT\_HERSHEY\_SIMPLEX

expressoes = ["Raiva", "Nojo", "Medo", "Feliz", "Triste", "Surpreso", "Neutro"]

while (cv2.waitKey(1) < 0):

conectado, frame = cap.read()

if not conectado:

break # se ocorreu um problema ao carregar a imagem então interrompe o programa

t = time.time() # tempo atual, antes de iniciar (vamos utilizar para calcular quanto tempo levou para executar as operações)

# frame\_video = np.copy(frame) # faz uma copia do frame do video

if redimensionar: # se redimensionar = True então redimensiona o frame para os novos tamanhos

frame = cv2.resize(frame, (video\_largura, video\_altura))

face\_cascade = cv2.CascadeClassifier(haarcascade\_faces)

cinza = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY) # converte pra grayscale

faces = face\_cascade.detectMultiScale(cinza,scaleFactor=1.2, minNeighbors=5,minSize=(30,30))

if len(faces) > 0:

for (x, y, w, h) in faces:

# se detectar mais de uma face então considera aquela que possui uma maior area na imagem

if unica\_face and len(faces) > 1:

max\_area\_face = faces[0]

for face in faces:

if face[2] \* face[3] > max\_area\_face[2] \* max\_area\_face[3]:

max\_area\_face = face

face = max\_area\_face

(x,y,w,h) = max\_area\_face # retorna as coordenadas e tamanhos da maior face detectada nesse frame

frame = cv2.rectangle(frame,(x,y),(x+w,y+h+10),(255,50,50),2) # desenha retângulo ao redor da face

roi = cinza[y:y + h, x:x + w] # extrai apenas a região de interesse (ROI) que é onde contém o rosto

roi = cv2.resize(roi, (48, 48)) # antes de passar pra rede neural redimensiona para o tamanho das imagens de treinamento

roi = roi.astype("float") / 255.0 # normaliza

roi = img\_to\_array(roi) # converte para array para que a rede possa processar

roi = np.expand\_dims(roi, axis=0) # muda o shape da array

# faz a predição - calcula as probabilidades

result = model.predict(roi)[0]

print(result)

if result is not None:

if unica\_face:

for (index, (emotion, prob)) in enumerate(zip(expressoes, result)):

# nomes das emoções

text = "{}: {:.2f}%".format(emotion, prob \* 100)

barra = int(prob \* 150) # calcula do tamanho da barra, com base na probabilidade

espaco\_esquerda = 7 # é a coordenada x onde inicia a barra. define quantos pixels tem de espaçamento à esquerda das barras, pra não ficar muito no canto.

if barra <= espaco\_esquerda:

barra = espaco\_esquerda + 1

# se o tamanho da barra for menor que o espaço da esquerda então deixa a barra com 1 pixel de largura

# isso é feito pois estamos usando esse valor para passar por coordenada, se for menor então a barra irá crescer pra esquerda

cv2.rectangle(frame, (espaco\_esquerda, (index \* 18) + 7), (barra, (index \* 18) + 18), (200, 250, 20), -1)

cv2.putText(frame, text, (15, (index \* 18) + 15), cv2.FONT\_HERSHEY\_SIMPLEX, 0.25, (0, 0, 0), 1, cv2.LINE\_AA)

resultado = np.argmax(result) # encontra a emoção com maior probabilidade

cv2.putText(frame,expressoes[resultado],(x,y-10), fonte, fonte\_media,(255,255,255),1,cv2.LINE\_AA) # escreve a emoção acima do rosto

if unica\_face and len(faces) > 1:

break

# se ja executou a face maior então não precisa percorrer as outras porque vai fazer a mesma coisa (pois vai fazer o mesmo cálculo pra ver qual é a maior)

# então usamos break pra pular para o próximo frame

# tempo processado = tempo atual (time.time()) - tempo inicial (t)

cv2.putText(frame, " frame processado em {:.2f} segundos".format(time.time() - t), (20, video\_altura-20), fonte, fonte\_pequena, (250, 250, 250), 0, lineType=cv2.LINE\_AA)

cv2\_imshow(frame)

saida\_video.write(frame) # grava o frame atual

print("Terminou")

saida\_video.release()

cv2.destroyAllWindows()

Para maior precisão podemos usar o Dblib, mas ele é mais lento.

Os modelos acima não funcionaram tão bem com rostos inclinados por conta do modelo ter sido treinado mais com rostos centralizados.

### Seç**ão 4:**

Tudo que foi mostrado é o mesmo conteúdo do módulo 12, por conta disso estarei anexando o relatório do módulo 12 junto.