Reconhecimento facial utilizando CNN

1. Introdução

Este trabalho foi realizado por **Bruno Guedes Spinelli**, aluno do curso de Engenharia Biomédica da UFRN e os códigos se encontram nos links abaixo:

- Aquisição do banco de dados:

https://colab.research.google.com/drive/11dyPgsVC6lRfAOLdryb8Lil_mCVJ8Eqm

- Treinamento e teste da rede:

https://colab.research.google.com/drive/1vhDC63X0GseEyZxQf0aoQRaWSDaL_wvl

Reconhecimento facial é uma técnica de processamento digital de imagens e inteligência artificial para identificar a quais pessoas pertencem as faces humanas presente em uma imagem. Essa técnica é extremamente poderosa e vem sendo altamente explorada por diversas empresas e estados.

O reconhecimento facial baseia-se na ideia de biometria facial, isto é, cada rosto humano, normalmente, apresenta características únicas que o distingue dos demais. Muitas empresas vêm apostando em biometrias faciais para autenticar seus processos digitais e aprimorar serviços, entre elas Nubank, 99, Loggi, Banco Original, Neon e Gol. Atualmente este tipo de tecnologia, também, já é utilizada para a identificação de criminosos em diversos países, inclusive no Brasil.

O objetivo deste trabalho é desenvolver um algoritmo de reconhecimento facial utilizando o como base o modelo de rede neural convolucional utilizada em sala para reconhecer e identificar caracteres escritos à mão.

2. Metodologia

2.1 - Banco de imagens

Detectores Haar Cascade são algoritmos que possuem a finalidade de analisar diferentes atributos de uma imagem e então avaliar se estes atributos representam ou não os objetos de interesse. Esses sistemas utilizam janelas deslizantes sobre a imagem e para que um objeto seja considerado verdadeiro ele deve ser considerado positivo por todas as janelas.

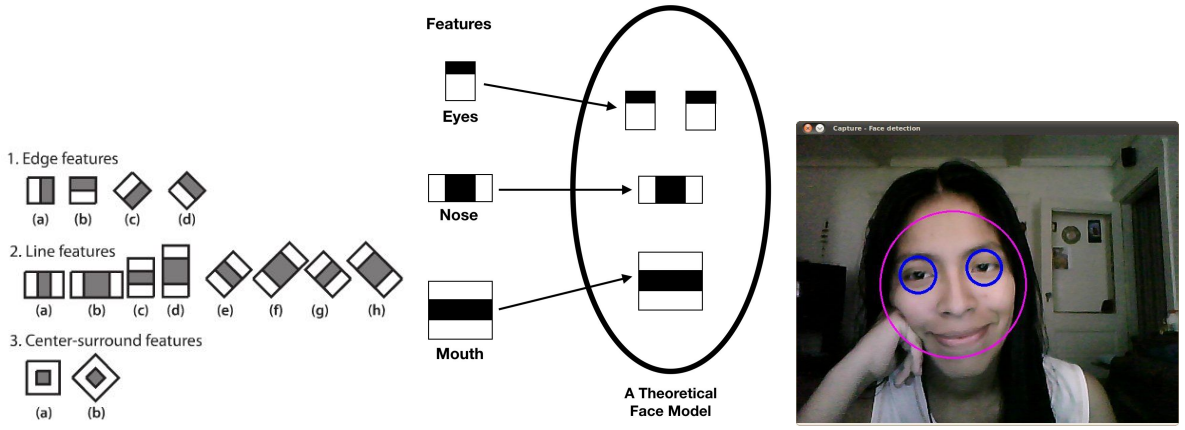

Figura 1 - Tipos de janelas utilizadas em algoritmos de Haar Cascade, representação de um rosto com janelas, resultado de uma deteção facial utilizando Haar Cascade.

Foram selecionados 8 voluntários para a aquisição de imagens para o treinamento da rede, sendo eles identificados pelas letras (A,B, J, M, R, P, R, S). Primeiramente foi necessário a geração de um banco de imagens padronizadas. Para isso a aquisição das imagen foi realizada por meio da gravação de vídeos de aproximadamente 20 segundos, em resolução de 1080p (1920×1080), na posição vertical e em diversos ângulos, de cada voluntário.

Em seguida os vídeos foram processados utilizando um algoritmo, utilizando Haar Cascade para a deteção de faces, que detectava o rosto contido em cada frame do vídeo, recortava-o e salvava-o, assim gerando diversas imagens contendo o rosto recortado do voluntário presente no vídeo.

Figura 2 - Resultado do processamento dos vídeos. Faces dos voluntários, respectivamente, A, B, J, M, R, P e S.

2.2 - Rede Convolucional

Uma Rede Neural Convolucional (ConvNet / Convolutional Neural Network / CNN) é um algoritmo de Aprendizado Profundo que pode captar uma imagem de entrada, atribuir importância (pesos e vieses que podem ser aprendidos) a vários aspectos / objetos da imagem e ser capaz de diferenciar um do outro. O pré-processamento exigido em uma ConvNet é muito menor em comparação com outros algoritmos de classificação. Enquanto nos métodos primitivos os filtros são feitos à mão, com treinamento suficiente, as ConvNets têm a capacidade de aprender esses filtros / características. Uma CNN pode ser dividida em duas partes: extração de características (Conv, Relu, Pooling, Dropout) e uma rede neural tradicional.

2.2.1 - Convolução

Matematicamente, uma convolução é uma operação linear que a partir de duas funções, gera uma terceira (normalmente chamada de feature map). No contexto de imagens, podemos entender esse processo como um filtro/kernel que transforma uma imagem de entrada.

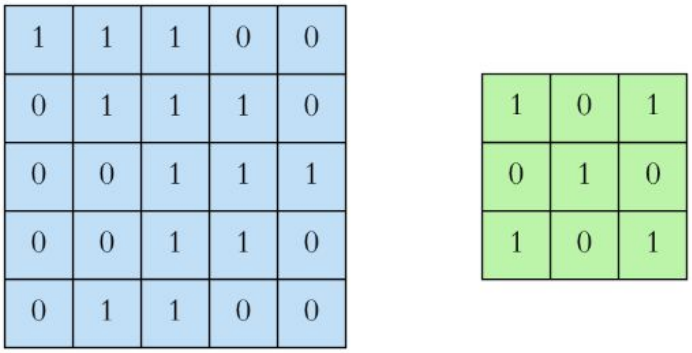

Figura 3 - Convolução.

2.2.2 - Relu

Uma rede neural sem função de ativação torna-se um modelo linear. Se o seu problema é linear, existem outros modelos mais simples que te atenderão tão bem quanto uma rede neural. Infelizmente a maioria dos problemas complexos não são lineares. Portanto, para adicionar a não linearidade a rede, utilizamos as funções de ativação. Nos dias de hoje, e principalmente no contexto de imagens, a mais utilizada é a função ReLU.

Matematicamente a função ReLU é definida como y = max(0, x). O gráfico a seguir é a ilustração desta função.

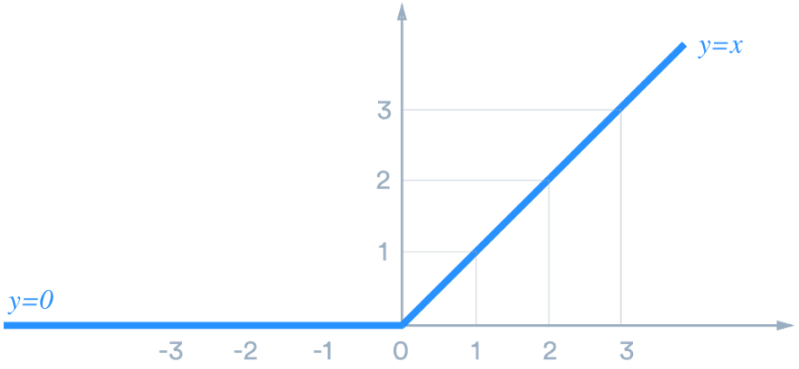

Figura 4 - Função Relu.

2.2.3 - Pooling

Pooling é um processo de downsamping. É um processo simples de redução da dimensionalidade/features maps. Em uma forma leviana de pensar, podemos entender essa transformação como uma redução do tamanho da imagem.

A principal motivação dessa operação no modelo, é de diminuir sua variância a pequenas alterações e também de reduzir a quantidade de parâmetros treinados pela rede.

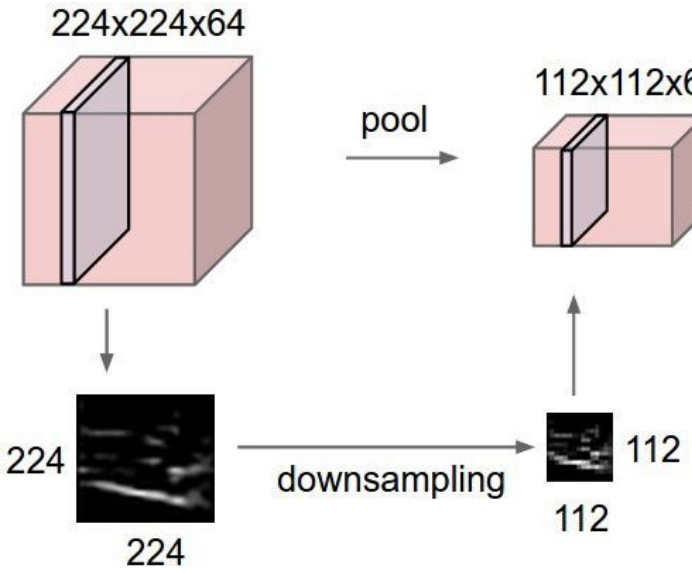

Figura 5 - Pooling.

Existem 3 operações diferentes de Pooling (MaxPooling, SumPooling, AvaragePooling). Todas elas seguem o mesmo princípio e só se diferem na forma como calculam o valor final. A mais utilizada nos dias de hoje é a MaxPooling.

A operação de MaxPooling retira o maior elemento de determinada região da matrix (considerando o tamanho do pool aplicado). Posteriormente, é feito um deslizamento considerando um parâmetro de stride (similar a operação de convolução) para aplicação de uma nova operação.

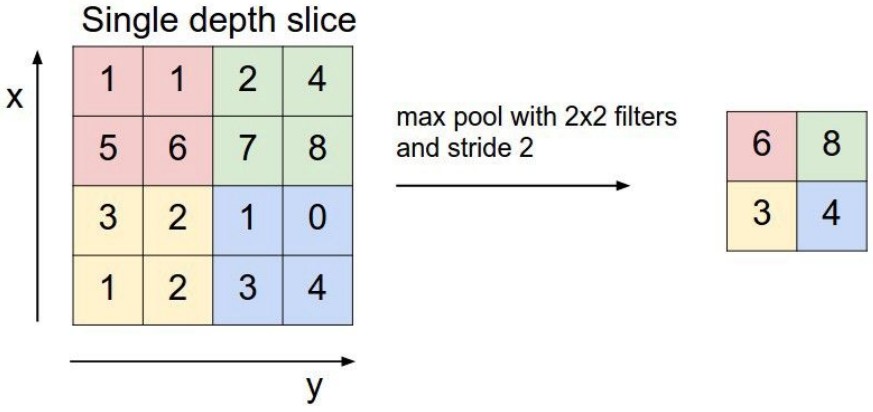

Figura 6 - MaxPooling.

2.2.4 - Dropout

A camada de Dropout é utilizada para evitar que determinadas partes da rede neural tenham muita responsabilidade e consequentemente, possam ficar muito sensíveis a pequenas alterações. Essa camada recebe um hyper-parâmetro que define uma probabilidade de “desligar” determinada área da rede neural durante o processo de treinamento.

2.3.5 - Flatten

Essa camada opera uma transformação na matriz da imagem, alterando seu formato para um array. Por exemplo, uma imagem em grayscale de 28x28 será transformada para um array de 784 posições.

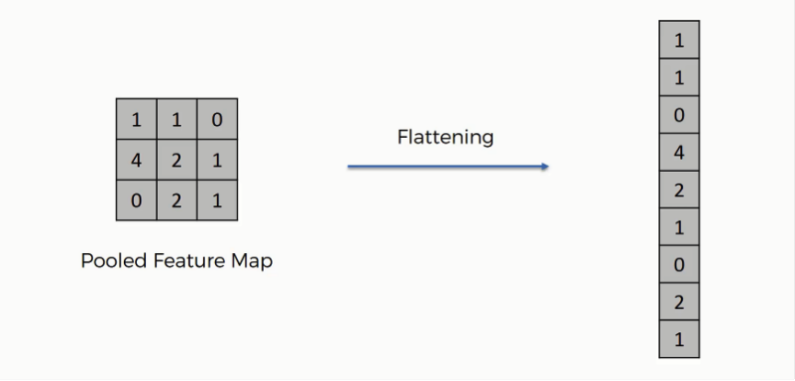

Figura 7 - Flateen.

2.3.6 - Rede Tradicional (Dense Layers)

Rede neural é um modelo computacional baseado no sistema nervoso central humano. Elas são capazes de reconhecer padrões em uma massa de dados de forma a classificá-los em alguma categoria ou fazer a regressão de algum valor.

3. Códigos

3.1 - Aquisição do banco de imagens

Antes de realizar o treinamento da rede foi necessária a geração do banco de imagens. Para isso foi desenvolvido um código em Python com a capacidade de receber um vídeo e salvar rosto contido em cada frame em um arquivo de imagem do tipo JPG.

Para o funcionamento desse algoritmo foi necessária a importação das seguintes bibliotecas e do arquivo contendo os parâmetros de Haar Cascade para identificação de rostos:

```
import numpy as np
import cv2
import requests
from PIL import Image

!curl -o haarcascade_frontalface_default.xml https://raw.githubusercontent.com/opencv/opencv/master/data/haarcascades/haarcascade_frontalface_default.xml
```

O algoritmo recebe um vídeo em formato MP4 utilizando a função **cv2.VideoCapture()**, armazena o primeiro frame do vídeo em **image** e se foi possível ler o frame em **success**. Em seguida utilizando o algoritmo de **Haar Cascade** para detecção de faces é feita a detecção da face no frame armazenado, logo após o rosto é cortado da imagem e armazenado na variável **roi_color**, logo após o rosto é redimensionado para o tamanho de 100 x 100 e salvo em um arquivo de imagem do tipo JPG por meio da função **cv2.imwrite()**. Esse processo é repetido até que não seja possível ler um frame do vídeo:

```
vidcap = cv2.VideoCapture('R.mp4')
success,image = vidcap.read()
count = 0
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
faceCascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")
faces = faceCascade.detectMultiScale(
    gray,
    scaleFactor=1.3,
    minNeighbors=3,
    minSize=(400, 400)
)
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
    roi_color = image[y:y + h, x:x + w]
    resized = cv2.resize(roi_color, (100,100))
    cv2.imwrite('faces2/R/' + str(w) + str(h) + '_faces.jpg', resized)
while success:
  success,image = vidcap.read()
  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
  faceCascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")
  faces = faceCascade.detectMultiScale(
    gray,
    scaleFactor=1.3,
    minNeighbors=3,
    minSize=(400, 400)
    )
  for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 2)
    roi_color = image[y:y + h, x:x + w
    resized = cv2.resize(roi_color, (100,100))
    cv2.imwrite('faces2/R/' + str(w) + str(h) + '_facesa%d.jpg' % count, resized)
  print('Read a new frame: ', success)
  count += 1
```

3.2 - Treinamento da Rede

Para o treinamento da rede de reconhecimento facial foi necessária a importação das seguintes bibliotecas e funções:

```
import os
import numpy as np
import cv2
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
import matplotlib.pyplot as plt
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from keras.utils.np_utils import to_categorical
import random
from keras.layers import Flatten, Dropout
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
np.random.seed(0)
```

A primeira coisa a ser feita pelo algoritmo é a descompactação do arquivo compactado onde se encontram as pastas que contendo as imagens das faces dos voluntários:

```
# Descompactando um arquivo Zip
!unzip faces2.zip
```

Em seguida o algoritmo faz a leitura e o armazenamento das imagens em variáveis:

```
images = []
labels = []
def traverse_dir(path):
    for file_or_dir in os.listdir(path):
        abs_path = os.path.abspath(os.path.join(path, file_or_dir))
        print(abs_path)
        if os.path.isdir(abs_path):  # dir
            traverse_dir(abs_path)
        else:                        # file
            if file_or_dir.endswith('.jpg'):
                image = read_image(abs_path)
                images.append(image)
                labels.append(path[len(path)-1])

    return images, labels
```

```
def read_image(file_path):
    image = cv2.imread(file_path)
    return image


def extract_data(path):
    images, labels = traverse_dir(path)
    images = np.array(images)

    return images, labels
```

```
images, labels = extract_data('./faces/')
labels = np.reshape(labels, [-1])
```

Os rótulos devem ser transformados em números de 0 a 7 (8 voluntários):

```
le = preprocessing.LabelEncoder()
le.fit(labels)
list(le.classes_)
```

```
['A', 'B', 'J', 'M', 'O', 'P', 'R', 'S']
```

```
labels_enc = le.transform(labels)
```

```
print(labels_enc[0])
print(le.inverse_transform([0]))
print(labels[0])
```

```
7
['A']
S
```

E os dados devem ser separados em dados de treino e teste:

```
X_train, X_test, y_train, y_test = train_test_split(images, labels_enc, test_size=0.3, random_state=1)
```

```
y_train = to_categorical(y_train,8)
y_test = to_categorical(y_test,8)
```

```
print(X_train.shape)
print(X_test.shape)
```

```
(3629, 100, 100, 3)
(1556, 100, 100, 3)
```

```
print(y_train.shape)
print(y_test.shape)
```

```
(3629, 8)
(1556, 8)
```

```
X_train = X_train/255
X_test = X_test/255
```

Logo após, é construído o modelo de **CNN**, baseado no modelo já apresentado em sala, apresentando 2 **Convoluções** com **MaxPooling** e a respectiva rede neural. Foram realizadas algumas alterações na rede original, apresentada em sala, como a adição de **Dropout**, a mudança da matriz de entrada para aceitar imagens em **RGB**, e a mudança do otimizador para um otimizador do tipo **SGD**:

```
num_classes = 8;
def leNet_model():
  model = Sequential()
  model.add(Conv2D(30, (5,5), input_shape=(100,100,3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Dropout(0.25))
  model.add(Conv2D(15,(3,3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(Dropout(0.25))
  model.add(Flatten())
  model.add(Dense(500, activation='relu'))
  model.add(Dense(num_classes, activation='softmax'))
  sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
  model.compile(optimizer=sgd, loss='categorical_crossentropy', metrics=['accuracy'])
  return model
```

Por fim, é realizado o treino da rede:

```
history = model.fit(X_train, y_train, validation_split=0.1, epochs=5, batch_size = 150, verbose = 1, shuffle = 1)

Train on 3266 samples, validate on 363 samples
Epoch 1/5
3266/3266 [==============================] - 25s 8ms/step - loss: 1.7777 - acc: 0.4247 - val_loss: 1.3187 - val_acc: 0.5234
Epoch 2/5
3266/3266 [==============================] - 24s 7ms/step - loss: 1.0438 - acc: 0.6938 - val_loss: 0.3828 - val_acc: 0.8705
Epoch 3/5
3266/3266 [==============================] - 24s 7ms/step - loss: 0.2806 - acc: 0.9094 - val_loss: 0.1243 - val_acc: 0.9807
Epoch 4/5
3266/3266 [==============================] - 24s 7ms/step - loss: 0.0617 - acc: 0.9847 - val_loss: 0.0265 - val_acc: 0.9945
Epoch 5/5
3266/3266 [==============================] - 24s 7ms/step - loss: 0.0419 - acc: 0.9868 - val_loss: 0.0341 - val_acc: 0.9862
```

4. Experimentos

4.1 - Treinamento e resultado do treinamento

O treinamento da rede foi realizado com centenas de imagens de 8 voluntários:

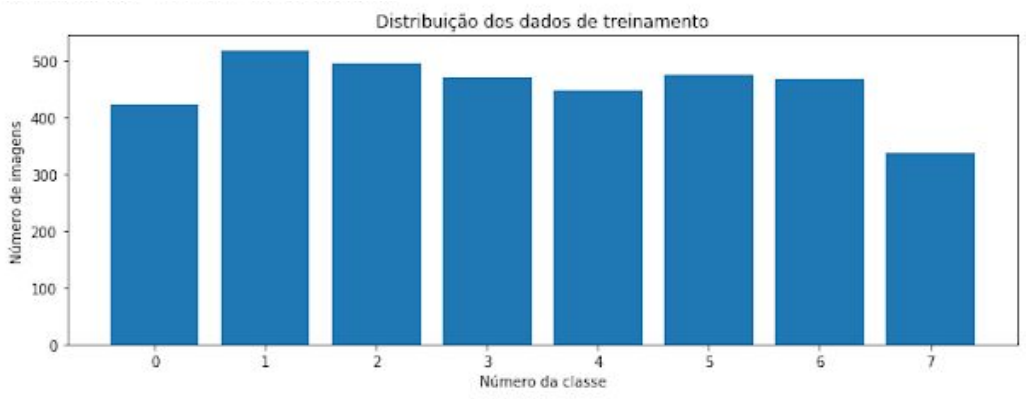

Figura 8 - Número de imagens por voluntário.

Figura 9 - Imagens das faces dos voluntários e seus rótulos (respectivamente: A, B, J, M, O, P, R e S).

Abaixo encontra-se o gráfico de acurácia por época de treinamento:

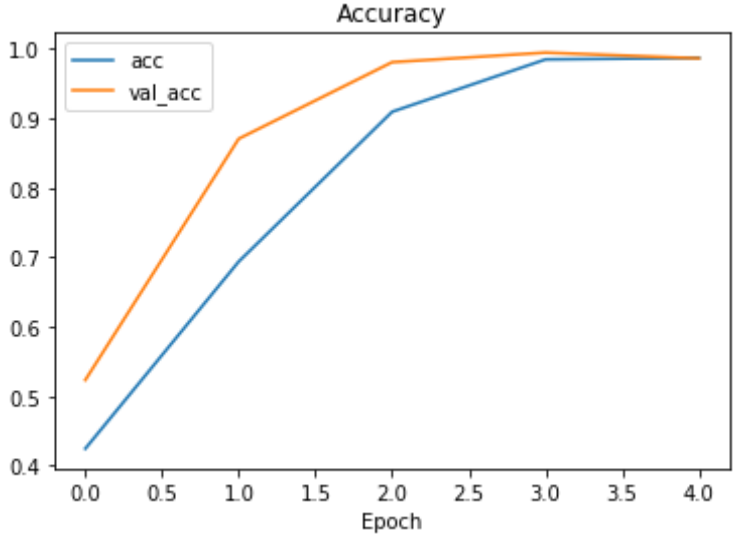

Figura 10 - Acurácia por época

4.2 - Testes da rede

4.2.1 - Teste com dados de teste

Inicialmente a rede foi testada utilizando os dados previamente selecionados como dados de teste, quando obteve um resultado impressionante de mais de 99% de acerto.

```
score = model.evaluate(X_test, y_test, verbose=0)
print(type(score))
print('Test score:',score[0])
print('Test accuracy:',score[1])
```

```
<class 'list'>
Test score: 0.025608987195517838
Test accuracy: 0.9916452442159382
```

Figura 11 - Teste com dados de teste.

Em seguida, foram realizados testes com imagens completamente novas e/ou que não estavam presentes no banco de imagens original.

4.2.2 - Testes com novas imagens:

- Teste 1: Imagem do voluntário J

```
Predicted digit: [2]
['J']
```

Figura 12 - Imagem da face de J e predição da rede.

- Teste 2: Imagem do voluntário R

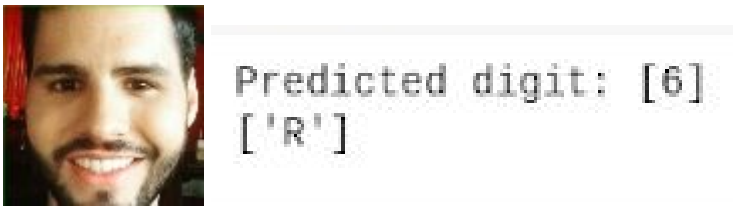

Figura 13 - Imagem da face de R e predição da rede.

- Teste 3: Imagem do voluntário B

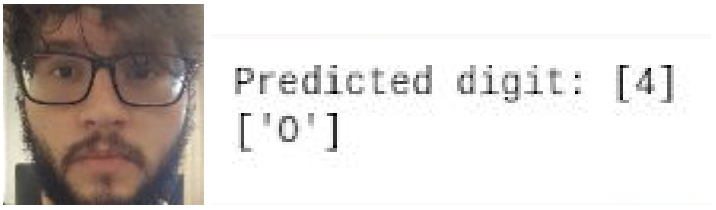

Figura 14 - Imagem da face de B e predição da rede.

- Teste 4: Imagem do voluntário M

Figura 15 - Imagem da face de M e predição da rede.

- Teste 5: Imagem do voluntário A

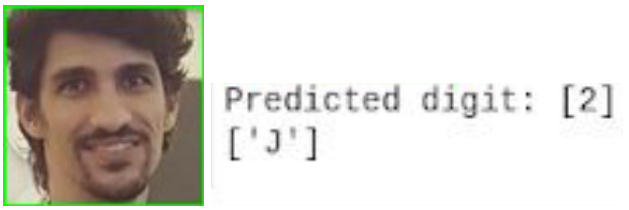

Figura 16 - Imagem da face de A e predição da rede.

- Teste 6: Imagem do voluntário O

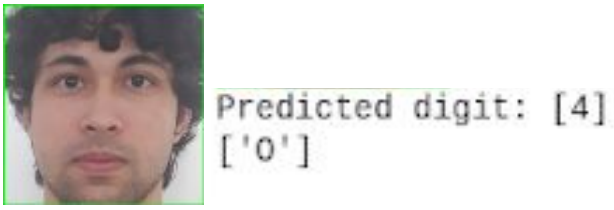

Figura 17 - Imagem da face de O e predição da rede.

4.3 - Conclusão

Nos testes com os dados de teste a rede obteve uma performance impressionante com mais de 99% de acurácia, entretanto nos testes com novas imagens ela não performou tão bem quanto no teste anterior. Entretanto já podemos verificar que a rede de fato consegue realizar algumas previsões de forma correta e possivelmente caso seja exposta a um banco de imagens mais amplo, com mais imagens, mais ângulos, maiores diferenças de iluminação ela consiga realizar os dois testes com a mesma performance.