

22 - Prática - Reconhecimento de Emoções com TensorFlow 2.0 e Python (III)

Lucas Samuel Zanchet

Utilizando um modelo já treinado

O reconhecimento de expressões faciais é uma das aplicações da visão computacional. Consiste em treinar uma rede neural convolucional para identificar as emoções através das distâncias entre certos pontos característicos do rosto.

No curso primeiro utilizamos um modelo já treinado para fazer testes. Fui além da aula e utilizei a webcam para fazer reconhecimento do meu próprio rosto, porém só funciona em máquina local, não no google colab.

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

Detectando ROI (*Region Of Interest*):

```
1 faces = face_detection.detectMultiScale(original,  
    scaleFactor = 1.1, minNeighbors = 3, minSize = (20,20))
```

Também redimensionamos as imagens e normalizamos os valores dos pixels

Por último adicionamos uma dimensão para ficar compatível com a entrada do modelo:

```
1 roi = np.expand_dims(roi, axis = 0)  
2 roi.shape  
3 ### (1, 48, 48, 1)
```

Fazemos isso para cada rosto na imagem e no fim podemos gerar o resultado abaixo:



Treinando um modelo do zero

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.

```
1 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:

```
1 face = [int(pixel) for pixel in pixel_sequence.split(' ')]
```

Normalizamos as imagens e deixamos-as no formato correto que o modelo irá aceitar.

Construção do modelo

```
1 num_features = 64
2 num_labels = 7
3 batch_size = 64
4 epochs = 100
5 width, height = 48, 48
6
7 model = Sequential()
8 model.add(Conv2D(num_features, kernel_size=(3,3),
9 activation='relu',
10 input_shape=(width, height, 1), data_format =
   'channels_last',
   kernel_regularizer = l2(0.01)))
```

```
11 model.add(Conv2D(num_features, kernel_size=(3,3),
12 activation='relu', padding='same'))
13 model.add(BatchNormalization())
14 model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
15 model.add(Dropout(0.5))
16 model.add(Conv2D(2*num_features, kernel_size=(3,3),
17 activation='relu', padding='same'))
18 model.add(BatchNormalization())
19 model.add(Conv2D(2*num_features, kernel_size=(3,3),
20 activation='relu', padding='same'))
21 model.add(BatchNormalization())
22 model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
23 model.add(Dropout(0.5))
24 model.add(Conv2D(2*2*num_features, kernel_size=(3,3),
25 activation='relu', padding='same'))
26 model.add(BatchNormalization())
27 model.add(Conv2D(2*2*num_features, kernel_size=(3,3),
28 activation='relu', padding='same'))
29 model.add(BatchNormalization())
30 model.add(MaxPooling2D(pool_size=(2,2), strides=(2,2)))
31 model.add(Dropout(0.5))
32 model.add(Flatten())
33 model.add(Dense(2*2*2*num_features, activation='relu'))
34 model.add(Dropout(0.4))
35 model.add(Dense(2*2*num_features, activation='relu'))
36 model.add(Dropout(0.4))
37 model.add(Dense(2*num_features, activation='relu'))
38 model.add(Dropout(0.5))
39 model.add(Dense(num_labels, activation = 'softmax'))
40 model.summary()
```

Também iremos criar os callbacks que serão utilizados no treinamento do modelo:

```

1  #Reduz a learning rate quando o modelo não está progredindo
2  lr_reducer = ReduceLR0nPlateau(monitor='val_loss', factor =
    0.9, patience=3, verbose = 1)
3
4  #Para o treinamento quando não há mais queda do loss de
    validação
5  early_stopper = EarlyStopping(monitor='val_loss',
    min_delta=0, patience = 8, verbose = 1, mode = 'auto')
6
7  #Salva os melhores modelos
8  checkpointer = ModelCheckpoint(arquivo_modelo,
    monitor='val_acc', verbose = 1, save_best_only=True)

```

Após treinar o modelo obtive as seguintes estatísticas:

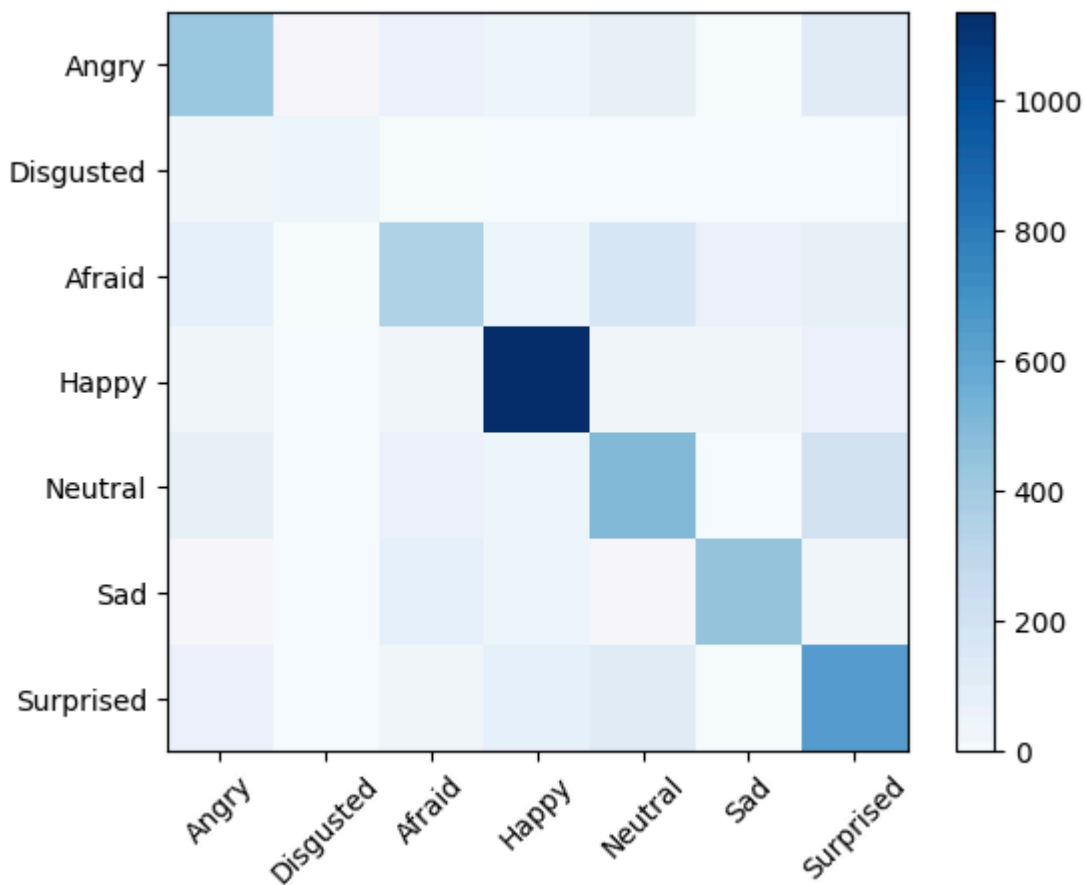
```

accuracy: 0.8556 - loss: 0.4001 - val_accuracy: 0.6566 -
val_loss: 1.2613

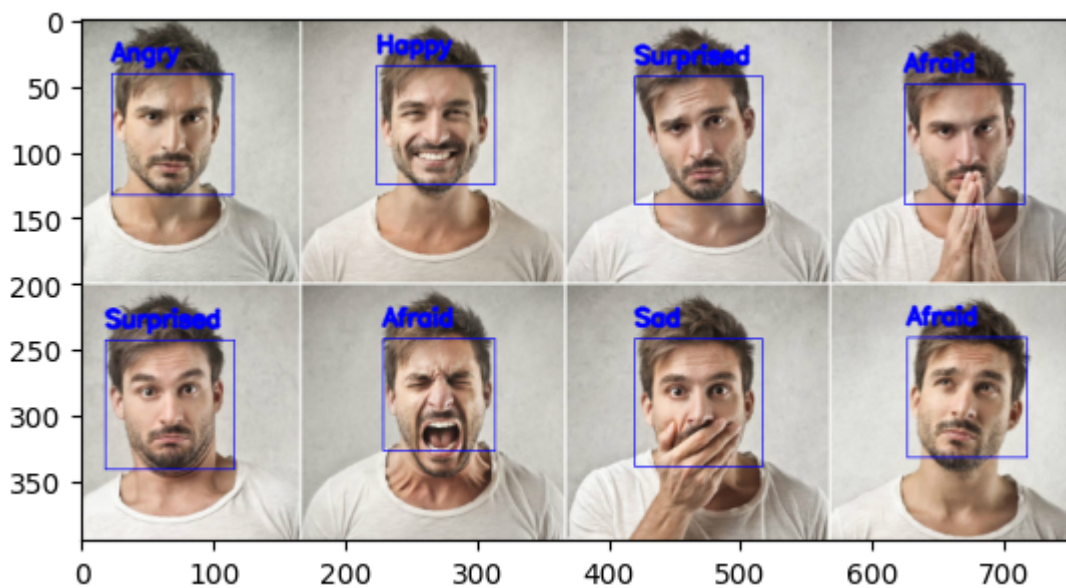
```

Com a seguinte matriz de confusão:

	Angry	Disgusted	Afraid	Happy	Neutral	Sad	Surpris
Angry	425	12	64	42	94	6	112
Disgusted	22	46	5	2	4	1	1
Afraid	86	6	365	43	165	56	89
Happy	33	0	15	1136	31	22	70
Neutral	96	0	65	45	501	2	191
Sad	12	1	71	44	10	447	16
Surprised	57	1	34	75	123	8	641



E os seguintes resultados:



No Keras 3 não utilizamos mais os arquivos `.json` e `.h5` para salvar um modelo, mas utilizamos os arquivos `.keras`, portanto caso quisermos carregar novamente o modelo salvo pelo Checkpointer:

```
1 loaded_model =
  tensorflow.keras.models.load_model('modelo_01_expressoes.kera
```

Outras arquiteturas

Foram apresentadas outras sugestões de arquiteturas para a tarefa. Abaixo estão as estatísticas de cada um dos modelos:

```
1  ---modelo_01_expressoes.h5---
2  Perda/Loss: 1.0704169079150685
3  Acurácia: 0.63917524
4
5  ---modelo_02_expressoes.h5---
6  Perda/Loss: 1.011896936947936
7  Acurácia: 0.6411257
8
9  ---modelo_03_expressoes.h5---
10 Perda/Loss: 1.073169717726584
11 Acurácia: 0.6308164
12
13 ---modelo_04_expressoes.h5---
14 Perda/Loss: 1.1690520508337832
15 Acurácia: 0.61604905
16
17 ---modelo_05_expressoes.h5---
18 Perda/Loss: 1.8206109304420128
19 Acurácia: 0.24547228
```

Porém o modelo mais interessante na minha opinião é o *Inception*. A arquitetura Inception, desenvolvida pelo Google em 2014, foi projetada para ser eficiente em termos de computação e ao mesmo tempo aumentar a precisão em tarefas de classificação de imagens. Ela ficou popular após o lançamento do modelo **GoogLeNet** (ou Inception v1), que venceu a competição ImageNet de 2014.

A ideia central do Inception é capturar informações em múltiplas escalas em cada camada convolucional da rede. Para isso, ele combina convoluções de diferentes tamanhos e pooling em paralelo, de forma a criar uma representação rica da imagem.

Portanto o engenheiro deixa ao o processo de *backpropagation* a responsabilidade de escolher qual tamanho de *kernel* utilizar.

Transferência de aprendizagem

Utilizando o modelo pré treinado VGG-16 podemos construir um outro modelo que utiliza o que esse modelo já está treinado para fazer.

```
1  from tensorflow.keras.applications import VGG16
2
3  vgg = VGG16(input_shape=(width, height, 3),
4              weights='imagenet', include_top=False)
5
6  vgg.trainable=False
7  global_average_layer = GlobalAveragePooling2D()
8  prediction_layer = Dense(num_classes,activation='softmax')
9
10 model = Sequential([
11     vgg,
12     global_average_layer,
13     prediction_layer
14 ])
```

Depois disso treinamos o novo modelo normalmente.

CONCLUSÃO

O reconhecimento de emoções tem aplicações como as que foram citadas no curso como uma melhor interação entre máquina e homem, para que a máquina compreenda a situação emocional do ser humano. Porém também há aplicações já utilizadas na indústria como a detecção de mentiras e a detecção de sono em motoristas. O mesmo princípio utilizado para detecção de emoções é utilizado também para reconhecimento facial, onde a rede neural convolucional aprende certas métricas do rosto que a permitem diferenciar traços faciais de pessoas.

REFERÊNCIAS

HaskDork. (2024, Setembro 17). Face Recognition with TensorFlow: A Step-by-Step Guide. HashDork. <https://hashdork.com/face-recognition-with-tensorflow/>