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

Lucas Samuel Zanchet

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*):

```
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:

```
roi = np.expand_dims(roi, axis = 0)
roi.shape
### (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.

```
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:

```
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

```
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()
```

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

```
#Reduz a learning rate quando o modelo não está progredindo
lr_reducer = ReduceLROnPlateau(monitor='val_loss', factor = 0.9, patience=3, verbose = 1)

#Para o treinamento quando não há mais queda do loss de validação
early_stopper = EarlyStopping(monitor='val_loss', min_delta=0, patience = 8, verbose = 1, mode = 'auto')

#Salva os melhores modelos
checkpointer = ModelCheckpoint(arquivo_modelo, monitor='val_acc', verbose = 1, save_best_only=True)
```

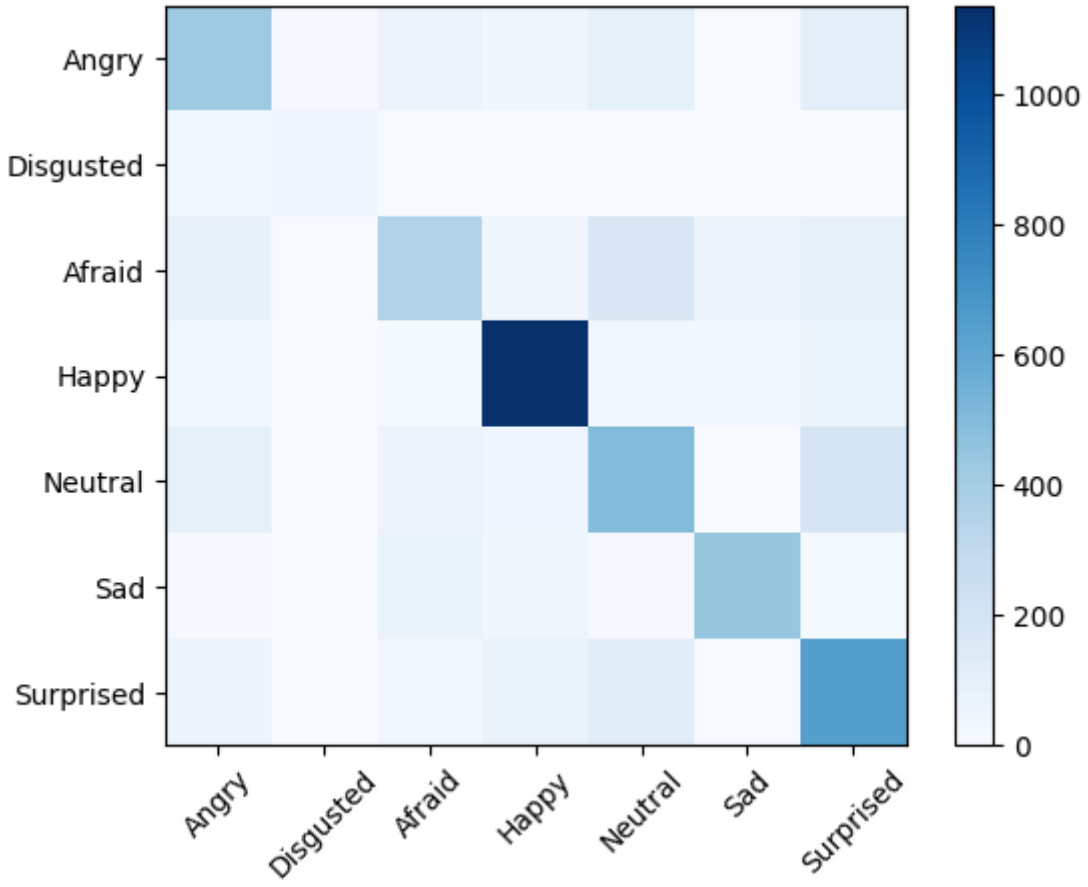
Após treinar o modelo obtive as seguintes estatísticas:

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

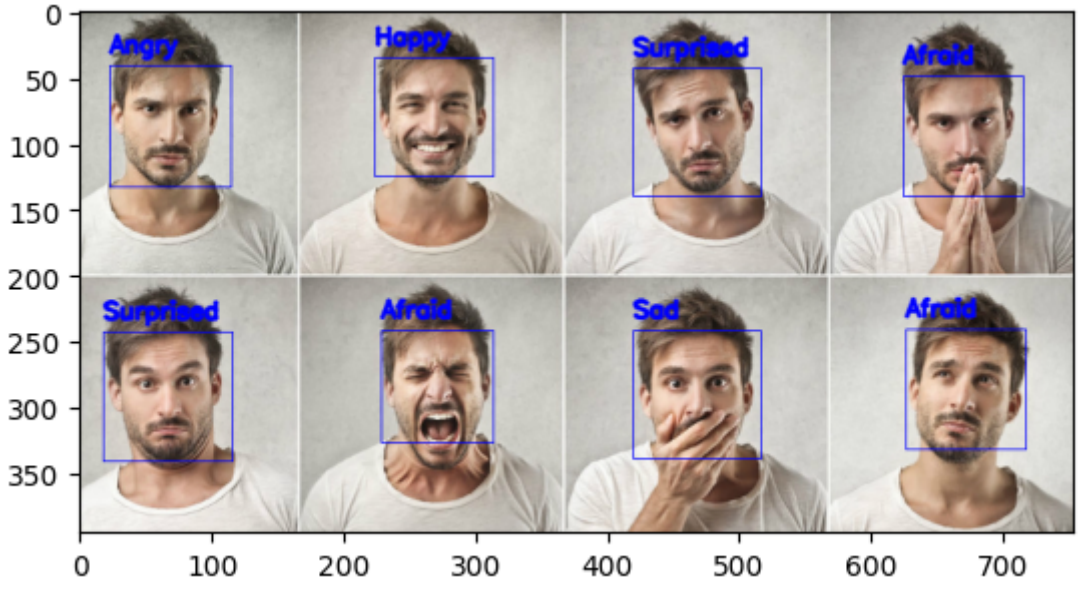
Com a seguinte matriz de confusão:

	Angry	Disgusted	Afraid	Happy	Neutral	Sad	Surprised
Angry	425	12	64	42	94	6	112
Disgusted	22	46	5	2	4	1	1
Afraid	86	6	365	43	165	56	89

	Angry	Disgusted	Afraid	Happy	Neutral	Sad	Surprised
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:

```
loaded_model = tensorflow.keras.models.load_model('modelo_01_expressoes.keras')
```

Outras arquiteturas

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

```
---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
```

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.

```
from tensorflow.keras.applications import VGG16

vgg = VGG16(input_shape=(width, height, 3), weights='imagenet', include_top=False)

vgg.trainable=False
global_average_layer = GlobalAveragePooling2D()
prediction_layer = Dense(num_classes,activation='softmax')

model = Sequential([
        vgg,
        global_average_layer,
        prediction_layer
])
```

Depois disso treinamos o novo modelo normalmente.