

Lab 4 - BCC406

REDES NEURAIS E APRENDIZAGEM EM PROFUNDIDADE

Convolução e CNN

Prof. Eduardo e Prof. Pedro

Objetivos:

- Aplicação de filtros em imagens por meio de convolução
- Modelagem de uma rede de convolução para o problema de classificação de gatos/não gatos.
- Notebook baseado em tensorflow e Keras.

Data da entrega : 11/02

- Complete o código (marcado com `ToDo`) e quando requisitado, escreva textos diretamente nos notebooks. Onde tiver *None*, substitua pelo seu código.
- Execute todo notebook e salve tudo em um PDF **nomeado** como "NomeSobrenome-LabX.pdf"
- Envie o PDF via google [FORM](#)

✓ Aplicando filtros e entendendo padding, stride e pooling (30pt)

A primeira etapa é importar os pacotes e montar o drive

```
1 import tensorflow as tf
2 from tensorflow import keras
3 from keras.models import Sequential
4 from keras.layers import Conv2D, MaxPool2D, AvgPool2D
5 from tensorflow.keras import datasets, layers, models
6 import os
7 import skimage
8 from skimage import io
9 from skimage.io import imread
10 from skimage.transform import resize
11 import numpy as np
```

```
12
13 %matplotlib inline
14 import matplotlib.pyplot as plt
```

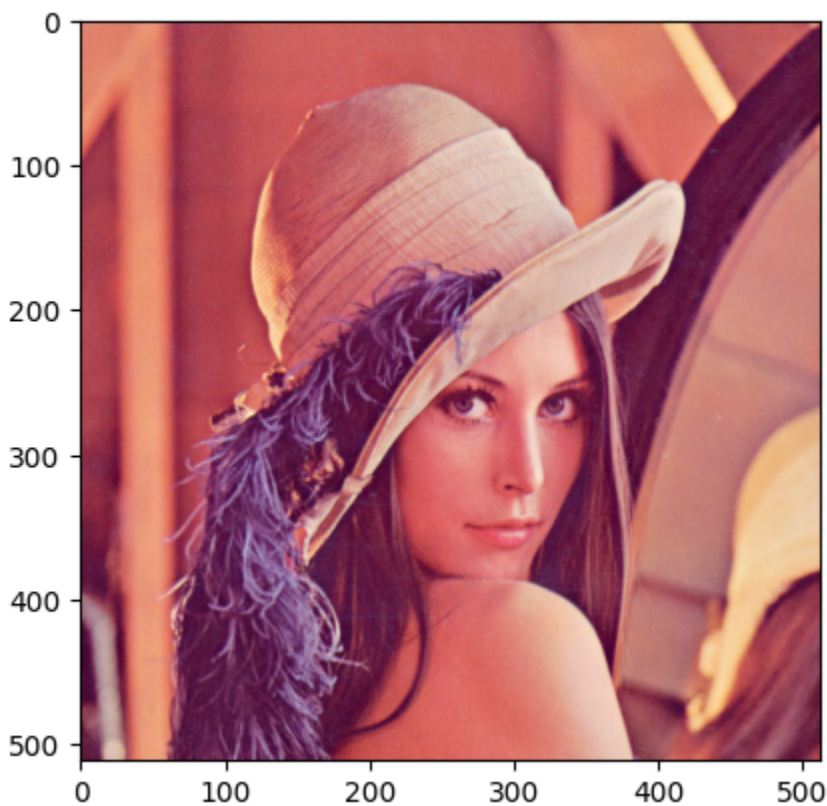
```
1 from google.colab import drive
2 drive.mount('/content/drive')
```

Mounted at /content/drive

Além dos passos anteriores, também iremos carregar uma imagem no disco para usá-la como exemplo para as próximas funções. (Imagem disponível na pasta de *Datasets* da disciplina)

```
1 sample_image = imread("/content/drive/MyDrive/Praticas redes neurais/Lenna.png")
2 sample_image= sample_image.astype(float)
3
4 size = sample_image.shape
5 print("sample image shape: ", sample_image.shape)
6
7 plt.imshow(sample_image.astype('uint8'));
```

sample image shape: (512, 512, 3)



Veja o shape da imagem:

```
1 sample_image.shape
```

```
(512, 512, 3)
```

✓ Criando e aplicando um filtro com convolução (10pt)

Utilize o `TF/Keras` para aplicar o filtro. Observe que nesta etapa não há necessidade de treinamento algum. O código abaixo cria 3 filtros de tamanho 5×5 , e adiciona *padding* de forma a manter a imagem de saída (filtrada) do mesmo tamanho da imagem de entrada (`padding = "same"`).

O objetivo do código abaixo é criar um objeto sequencial com apenas uma camada de convolução do tipo `tf.keras.layers.Conv2D`.

```
1 conv = Sequential([
2     Conv2D(filters=3, kernel_size=(5, 5), padding="same",
3         input_shape=(None, None, 3))
4 ])
5 conv.output_shape

/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:1
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
(None, None, None, 3)
```

```
1 conv.summary()
```

Model: "sequential"

| Layer (type) | Output Shape | Param |
|-----------------------------------|---|-------|
| conv2d (Conv2D) | (None , None , None , 3) | 22 |

```
Total params: 228 (912.00 B)
Trainable params: 228 (912.00 B)
Non-trainable params: 0 (0.00 B)
```

Quando usamos `TF/kertas`, as convoluções esperam vetores no formato: `(batch_size, dim1, dim2, dim3)`. Ou seja, a primeira posição é o tamanho do lote.

Uma imagem isolada é considerada um lote de tamanho 1, portanto, deve-se expandir mais uma dimensão do tensor.

```
1 img_in = np.expand_dims(sample_image, 0)
2 img_in.shape

(1, 512, 512, 3)
```

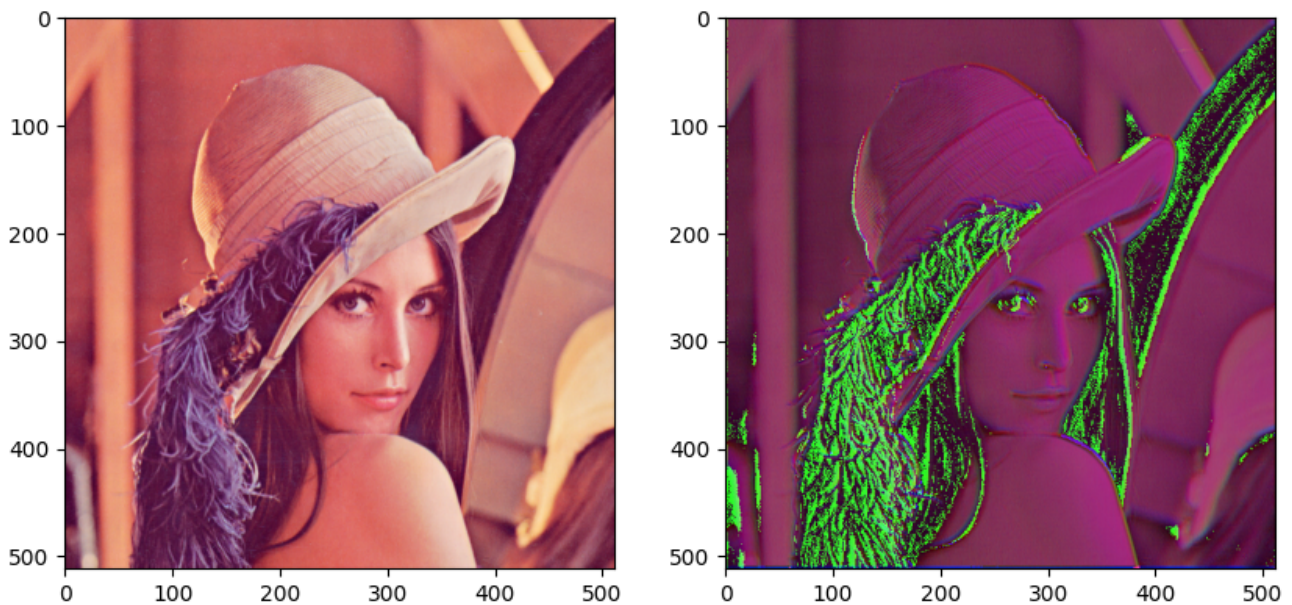
Agora, pode-se aplicar a convolução. Aplique a convolução na imagem de exemplo (expandida) e verifique o tamanho da imagem resultante (`img_out`). Use a função `predict` do objeto `conv` para aplicar a convolução.

```
1 img_out = conv(img_in)
2 img_out.shape

TensorShape([1, 512, 512, 3])
```

Plote as imagens lado a lado e observe o resultado. O parâmetro "same" no padding aplica um padding automático no sentido de garantir que a saída tenha o mesmo tamanho da entrada. Lembre-se que o padding adiciona zeros nas bordas da imagem, antes da aplicação da convolução.

```
1 fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(10, 5))
2 ax0.imshow(sample_image.astype('uint8'))
3 ax1.imshow(img_out[0].numpy().astype('uint8'));
```



Agora, crie **um** único filtro de tamanho 5×5 , e adicione *padding* oposto ao anterior (`valid` ao invés de `same`).

```

1 conv2 = Sequential([
2     Conv2D(filters=1, kernel_size=(5, 5), padding="valid",
3           input_shape=(None, None, 3))
4 ])
5 conv2.output_shape

(None, None, None, 1)

```

Um filtro $5 \times 5 \times 3$ tem a profundidade do filtro de acordo com a entrada, ou seja, tem-se $5 \times 5 \times 3 = 75$ valores que serão convoluídos pela imagem. Detalhe importante: **não se esqueça do bias! **

```
1 conv2.summary()
```

Model: "sequential_1"

| Layer (type) | Output Shape | Param |
|-------------------|-----------------------|-------|
| conv2d_1 (Conv2D) | (None, None, None, 1) | 7 |

Total params: 76 (304.00 B)
 Trainable params: 76 (304.00 B)
 Non-trainable params: 0 (0.00 B)

```

1 img_out = conv2(img_in)
2 img_out[0].shape

TensorShape([508, 508, 1])

```

ToDo: O que você observou no shape após a troca no conteúdo do *padding*?

Ao aplicar o padding como valid, o algoritmo realizou a convolução somente com os pixels originais

Como tivemos que expandir a primeira dimensao para aplicar a convolução, podemos remover a dimensão unitária para plotar a imagem, usando a função `squeeze()`

```

1 i = img_out[0].numpy().squeeze()
2 i.shape

(508, 508)

```

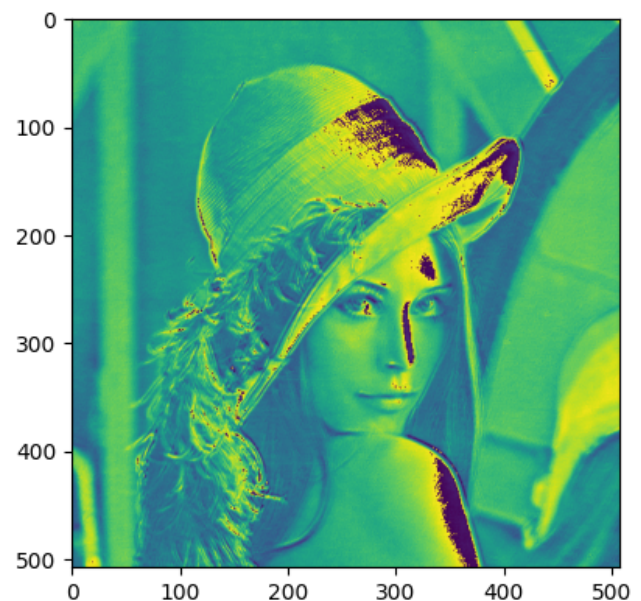
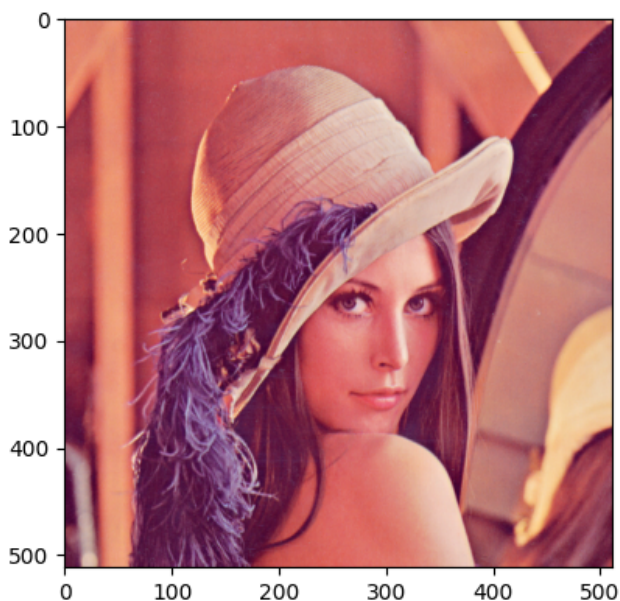
Agora com isso feito, é possível plotar as duas imagens lado a lado.

```
1 fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(10, 5))
```

```

1 fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(10, 5))
2 ax0.imshow(sample_image.astype('uint8'))
3 i = img_out[0].numpy().squeeze()
4 ax1.imshow(i.astype('uint8'));

```



ToDo: O que você observou nas imagens resultantes após a troca no conteúdo do *padding*?

Percebe-se uma mudança no padrão das cores da imagem assim como a aplicação dos filtros em partes

✓ Inicializando os filtros manualmente (10pt)

A função abaixo inicializa um array de dimensões 5,5,3,3 com todas as posições zero, exceto as posições 5,5,0,0 , 5,5,1,1 e 5,5,2,2 que recebem o valor 1/25.

```

1 def my_filter(shape=(5, 5, 3, 3), dtype=None):
2     array = np.zeros(shape=shape, dtype=np.float32)
3     array[:, :, 0, 0] = 1. / 25
4     array[:, :, 1, 1] = 1. / 25
5     array[:, :, 2, 2] = 1. / 25
6     return array

```

A transposição pode ser usada para facilitar a visualização da matriz resultante.

```
1 np.transpose(my_filter()), (2, 3, 0, 1))
```

```
array([[[[0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04]],  
  
        [[0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ]],  
  
        [[0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ]]]],  
  
       [[[0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ]],  
  
        [[0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04],  
          [0.04, 0.04, 0.04, 0.04, 0.04]],  
  
        [[0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ]]]],  
  
       [[[0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ]],  
  
        [[0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ],  
          [0. , 0. , 0. , 0. , 0. ]]]])
```



```
[0.04, 0.04, 0.04, 0.04, 0.04],
[0.04, 0.04, 0.04, 0.04, 0.04],
[0.04, 0.04, 0.04, 0.04, 0.04],
[0.04, 0.04, 0.04, 0.04, 0.04]]], dtype=float32)
```

A função definida acima é usada para carregar valores nos filtros, e ela pode ser usada para pré-inicializar os filtros do objeto `conv3` o qual possui uma convolução 2D.

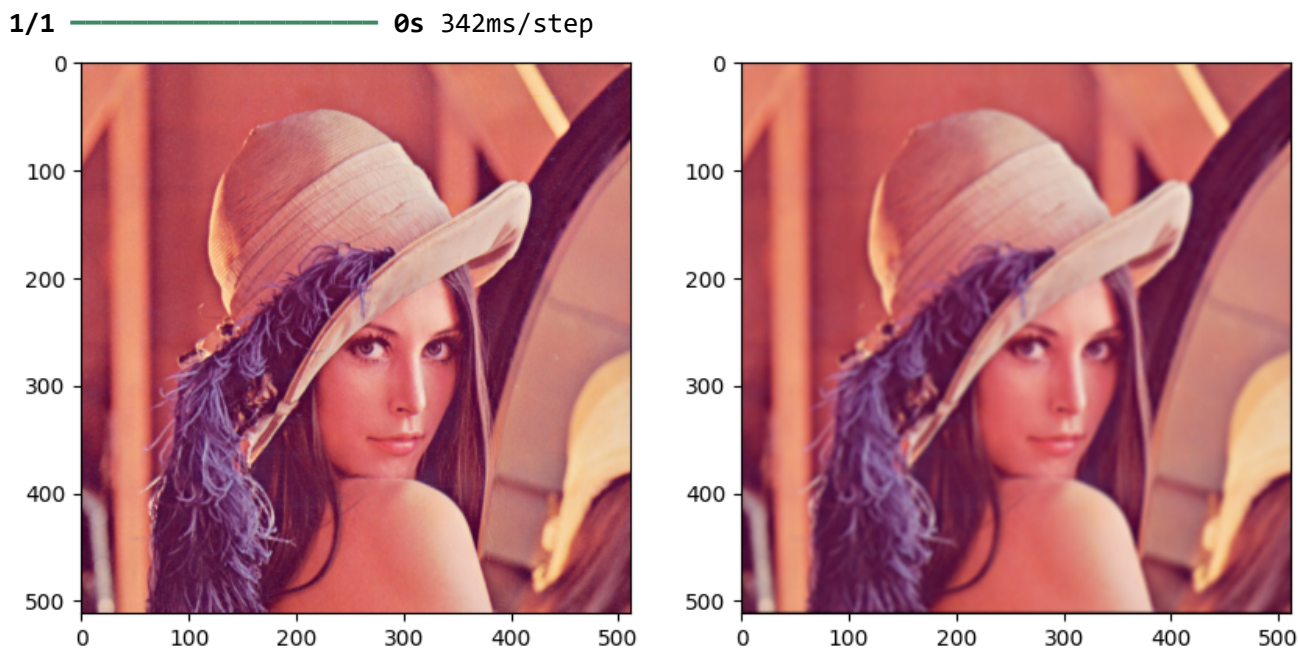
```
1 conv3 = Sequential([
2     Conv2D(filters=3, kernel_size=(5, 5), padding="same",
3         input_shape=(None, None, 3), kernel_initializer=my_filter)
4 ])
5 conv3.output_shape

(None, None, None, 3)
```

✓ Plote e observe o que aconte com a imagem (5pt)

Agora vamos testar o filtro criado na imagem de exemplo.

```
1 fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(10, 5))
2 ax0.imshow(img_in[0].astype('uint8'))
3 ax1.imshow(conv3.predict(img_in)[0].astype('uint8'));
```



ToDo: O que você observou após a aplicação do filtro criado manualmente na imagem original?

A imagem perdeu nitidez tendo uma adição de um borrão

✓ Criando um filtro de borda (5pt)

Crie uma nova função para gerar um filtro de borda nos 3 canais da imagem de entrada. O filtro deve ter o formato 3×3 e ter o formato $\begin{bmatrix} 0 & 0.2 & 0 \\ 0 & -0.2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$

```
1 def my_new_filter(shape=(1, 3, 3, 3), dtype=None):
2     my_filter = np.zeros(shape=shape, dtype=np.float32)
3     my_filter[0][0][1] = 0.2
4     my_filter[0][1][1] = -0.2
5     return my_filter
```

```
1 print(my_new_filter())
```

```
[[[ [ 0.  0.  0. ]
     [ 0.2 0.2 0.2]
     [ 0.  0.  0. ]]

  [[ 0.  0.  0. ]
   [-0.2 -0.2 -0.2]
   [ 0.  0.  0. ]]

  [[ 0.  0.  0. ]
   [ 0.  0.  0. ]
   [ 0.  0.  0. ]]]]
```

Inicialize o objeto `conv4` com seu novo filtro e aplique na imagem de entrada

```
1 conv4 = Sequential([
2     Conv2D(filters=3, kernel_size=(5, 5), padding="same",
3           input_shape=(None, None, 3), kernel_initializer=my_new_filter)
4 ])
5 conv4.output_shape

(None, None, None, 3)
```

Agora vamos plotar a imagem resultante.

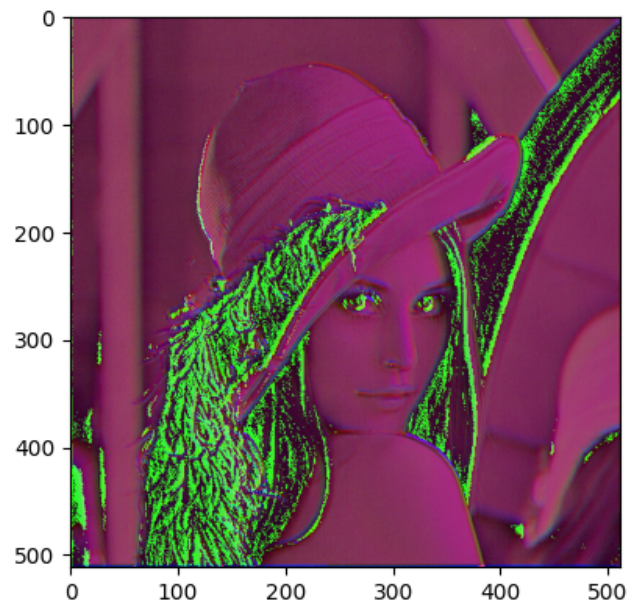
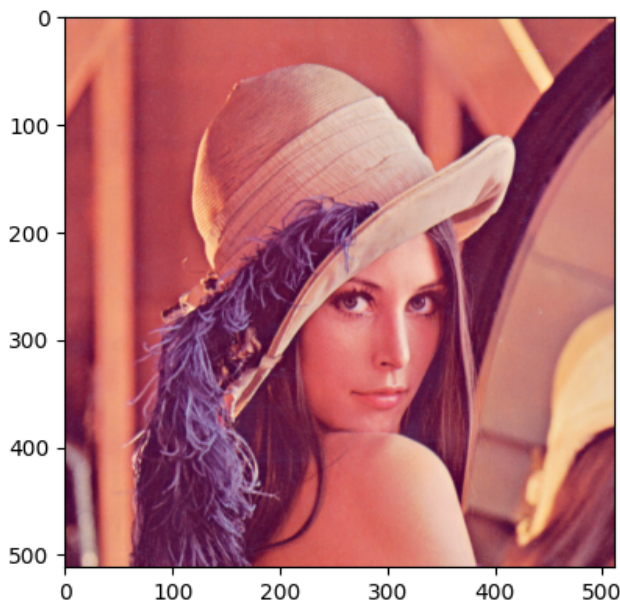
```
1 # Plota as duas imagens lado a lado (filtrada e não filtrada)
```

```

1 # Plote as duas imagens lado a lado (filtrada e não filtrada)
2 fig, (ax0, ax1) = plt.subplots(ncols=2, figsize=(10, 5))
3 ax0.imshow(img_in[0].astype('uint8'))
4 ax1.imshow(conv.predict(img_in)[0].astype('uint8'));

```

1/1 ————— 0s 69ms/step



ToDo: O que você observou após a aplicação do filtro criado manualmente na imagem original?

O filtro criado manualmente parece focar mais nos traços do cabelo e no portal, sendo que ambos ta

✓ Classificando imagens de gatos e cães (70pt)

Antes de qualquer coisa, primeiro é necessário carregar os dados.

✓ Carregando os dados de Gato e Não Gato (10pt)

Aqui você precisa carregar os dados e normalizá-los também. Nesta prática em específico, não é necessária a vetorização dos dados.

```

1 import h5py

```

```

2
3 def load_data():
4     train_dataset = h5py.File('/content/drive/MyDrive/Praticas redes neurais/Lab2/trai
5     train_X = np.array(train_dataset["train_set_x"][:]) # your train set features
6     train_Y = np.array(train_dataset["train_set_y"][:]) # your train set labels
7
8     test_dataset = h5py.File('/content/drive/MyDrive/Praticas redes neurais/Lab2/test_
9     test_x = np.array(test_dataset["test_set_x"][:]) # your test set features
10    test_Y = np.array(test_dataset["test_set_y"][:]) # your test set labels
11
12    classes = np.array(test_dataset["list_classes"][:]) # the list of classes
13
14
15    return train_X, train_Y, test_x, test_Y
16
17 train_X, train_Y, test_X, test_Y = load_data()

```

Observe o formato dos dados:

```

1 print(f'Shape dos dados de treino: {train_X.shape}') # ToDo: Mostrar o shape dos dados
2 print(f'Shape das labels de treino: {train_Y.shape}') # ToDo: Mostrar o shape dos dado
3
4 print(f'Shape dos dados de teste : {test_X.shape}') # ToDo: Mostrar o shape dos dados
5 print(f'Shape das labels de teste : {test_Y.shape}') # ToDo: Mostrar o shape dos dados

Shape dos dados de treino: (209, 64, 64, 3)
Shape das labels de treino: (209,)
Shape dos dados de teste : (50, 64, 64, 3)
Shape das labels de teste : (50,)

```

Dependendo da forma como você carregou os dados de rótulos, pode ser que ele tenha mais de uma dimensão. Se este for o seu caso, você pode usar a função `squeeze()` para o vetor de rótulos ficar somente com uma dimensão.

✓ Implementando a rede (20pt)

Implemente uma rede de convolução simples, contendo 3 camadas de convolução seguidas de duas camadas densas (totalmente conectadas) no final e por fim uma camada com ativação `sigmoid` para a classificação com um neurônio. Escolha filtros de tamanhos variados: (3,3) ou (5,5). Para cada camada, crie de 8 a 64 filtros.

Na camada densa, use de 64 a 256 neurônios.

```

1 # Implementa uma rede de convolução simples, chamada model

```

```

2
3 input_size = (train_X.shape[1], train_X.shape[2], 3)
4 n_classes = 1
5
6 model = models.Sequential()
7
8 model.add(layers.InputLayer(shape=(input_size)))
9
10 # ToDo : adicionar as outras camadas
11 model.add(Conv2D(filters=8, kernel_size=(5, 5), activation="relu", padding="valid"))
12 model.add(Conv2D(filters=32, kernel_size=(3, 3), activation="relu", padding="valid"))
13 model.add(Conv2D(filters=64, kernel_size=(3, 3), activation="relu", padding="valid"))
14
15
16 model.add(layers.Flatten()) # não esqueça da camada flatten ..
17
18 model.add(layers.Dense(256, activation='relu'))
19 model.add(layers.Dense(64, activation='relu'))
20 model.add(layers.Dense(1, activation = 'sigmoid'))
21
22

```

Agora usaremos o comando `model.summary()` para conferir a arquitetura que você construiu.

```
1 model.summary()
```

Model: "sequential_1"

| Layer (type) | Output Shape | Param |
|---------------------|--------------------|-----------|
| conv2d_3 (Conv2D) | (None, 60, 60, 8) | 60 |
| conv2d_4 (Conv2D) | (None, 58, 58, 32) | 2,33 |
| conv2d_5 (Conv2D) | (None, 56, 56, 64) | 18,49 |
| flatten_1 (Flatten) | (None, 200704) | |
| dense_3 (Dense) | (None, 256) | 51,380,48 |
| dense_4 (Dense) | (None, 64) | 16,44 |
| dense_5 (Dense) | (None, 1) | 6 |

Total params: 51,418,433 (196.15 MB)
 Trainable params: 51,418,433 (196.15 MB)
 Non-trainable params: 0 (0.00 B)

✓ Preparando o modelo para treinamento (5pt)

Compile o modelo usando o método de otimização adam e função de custo
(loss) binary_categorical_crossentropy.

```
1 model.compile(optimizer='adam', loss='binary_crossentropy', metrics = ["accuracy"])
```

✓ Treinando o modelo (5pt)

Treine o modelo por 30 épocas com batch_size = 100.

```
1 history = model.fit(x=train_X, y = train_Y, batch_size=100, epochs=30)
```

```
Epoch 1/30
3/3 ————— 11s 2s/step - accuracy: 0.5018 - loss: 717.5364
Epoch 2/30
3/3 ————— 2s 37ms/step - accuracy: 0.6590 - loss: 196.7712
Epoch 3/30
3/3 ————— 0s 35ms/step - accuracy: 0.3983 - loss: 11.9207
Epoch 4/30
3/3 ————— 0s 24ms/step - accuracy: 0.6403 - loss: 4.6185
Epoch 5/30
3/3 ————— 0s 24ms/step - accuracy: 0.5097 - loss: 1.6533
Epoch 6/30
3/3 ————— 0s 24ms/step - accuracy: 0.7975 - loss: 0.3473
Epoch 7/30
3/3 ————— 0s 23ms/step - accuracy: 0.7020 - loss: 0.4513
Epoch 8/30
3/3 ————— 0s 23ms/step - accuracy: 0.8140 - loss: 0.2937
Epoch 9/30
3/3 ————— 0s 23ms/step - accuracy: 0.9877 - loss: 0.2045
Epoch 10/30
3/3 ————— 0s 24ms/step - accuracy: 0.9939 - loss: 0.1981
Epoch 11/30
3/3 ————— 0s 23ms/step - accuracy: 0.9964 - loss: 0.1382
Epoch 12/30
3/3 ————— 0s 22ms/step - accuracy: 1.0000 - loss: 0.0935
Epoch 13/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 0.0651
Epoch 14/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 0.0302
Epoch 15/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 0.0158
Epoch 16/30
3/3 ————— 0s 22ms/step - accuracy: 1.0000 - loss: 0.0074
Epoch 17/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 0.0020
Epoch 18/30
3/3 ————— 0s 24ms/step - accuracy: 1.0000 - loss: 8.2766e-04
Epoch 19/30
```

```

3/3 ————— 0s 25ms/step - accuracy: 1.0000 - loss: 3.6828e-04
Epoch 20/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 2.8612e-04
Epoch 21/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 1.3873e-04
Epoch 22/30
3/3 ————— 0s 24ms/step - accuracy: 1.0000 - loss: 7.2860e-05
Epoch 23/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 4.8007e-05
Epoch 24/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 3.0536e-05
Epoch 25/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 1.8385e-05
Epoch 26/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 1.4299e-05
Epoch 27/30
3/3 ————— 0s 23ms/step - accuracy: 1.0000 - loss: 8.6631e-06
Epoch 28/30
3/3 ————— 0s 24ms/step - accuracy: 1.0000 - loss: 8.0026e-06
Epoch 29/30
3/3 ————— 0s 24ms/step - accuracy: 1.0000 - loss: 6.1079e-06

```

O retorno da função `fit()` é um objeto para armazenar o histórico do treino.

```

1 history.history.keys()

dict_keys(['accuracy', 'loss'])

```

Plote a acurácia e o custo (loss) do treino e da validação.

```

1 plt.plot(history.history['accuracy'], label='accuracy')
2 #plt.plot(history.history['val_acc'], label = 'val_accuracy')
3 plt.xlabel('Epoch')
4 plt.ylabel('Accuracy')
5 plt.ylim([0.5, 1.1])
6 plt.legend(loc='lower right')
7
8 plt.plot(history.history['loss'], label='loss')
9 #plt.plot(history.history['val_loss'], label = 'val_loss')
10 plt.xlabel('Epoch')
11 plt.ylabel('Accuracy')
12 plt.ylim([0.5, 1.1])
13 plt.legend(loc='lower right')
14
15 # ToDo: Coloque as suas variáveis de teste (x, y)
16 test_loss, test_acc = model.evaluate(test_X, test_Y, verbose=2)

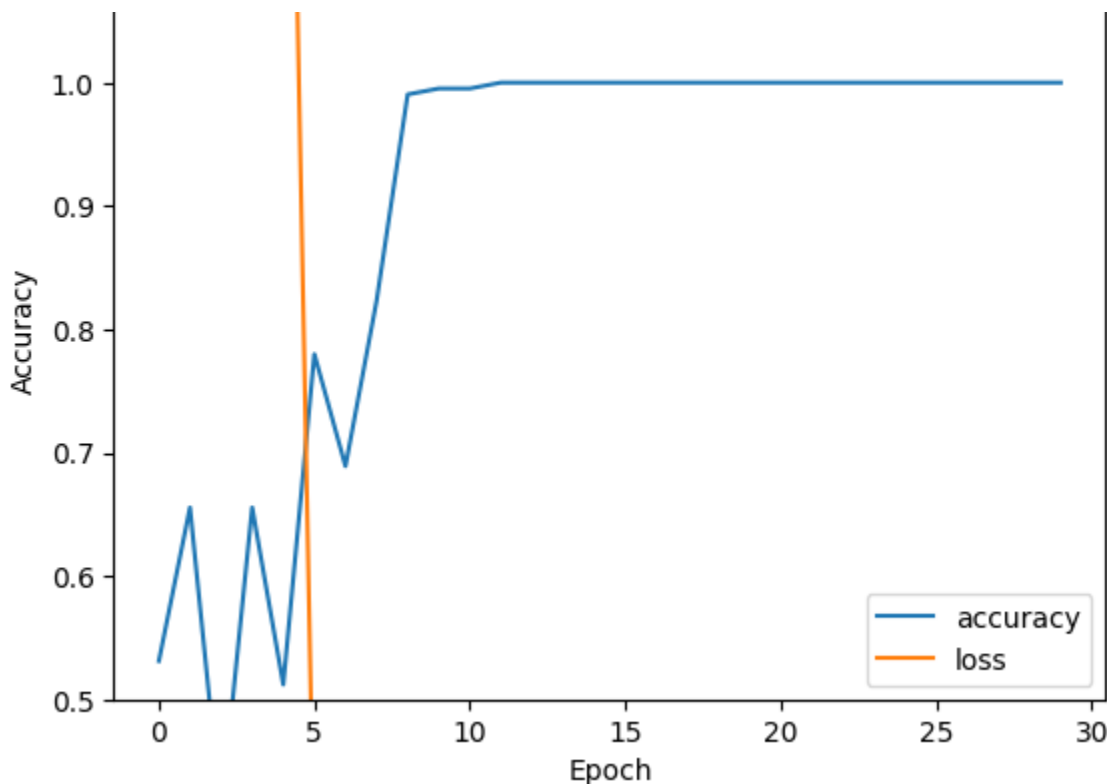
```

```

2/2 - 2s - 905ms/step - accuracy: 0.6400 - loss: 2.7502

```





Verificando a acurácia obtida:

```
1 print(test_acc)
```

```
0.6399999856948853
```

✓ Criando o seu próprio modelo (30pt)

O objetivo é agora você testar o mesmo cenário, mas criando os seus próprios modelos. A sua tarefa é criar/testar dois modelos. Para isso, você está livre para testar o que quiser, desde a quantidade de camadas de convolução e densas, até as funções de *loss* e ativação. Inclusive, se quiser, pode utilizar camadas de *pooling*.

✓ Modelo 1 (10pt)

```
1 from tensorflow.keras.layers import MaxPool2D, BatchNormalization, Dense, AveragePool2D
2
3 modelo1 = models.Sequential([
4     layers.InputLayer(shape=(input_size)),
5     Conv2D(filters=16, kernel_size=(3, 3), activation="relu", padding="same"),
6     Dropout(0.2),
7     MaxPool2D()])
```

```

7     max_pool2d(),
8     Conv2D(filters=16, kernel_size=(5, 5), activation="relu", padding="valid"),
9     MaxPool2D(),
10    Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding="valid"),
11    Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding="valid"),
12    MaxPool2D(),
13    Conv2D(filters=8, kernel_size=(3, 3), padding="valid"),
14
15    layers.Flatten(),
16
17    Dense(256, activation='relu'),
18    Dropout(0.2),
19    BatchNormalization(),
20    Dense(64, activation='relu'),
21    Dense(1, activation = 'sigmoid')
22
23 ])
24 modelo1.summary()
25
26 modelo1.compile(optimizer='adamw', loss='binary_crossentropy', metrics = ["accuracy"])
27 modelo1.fit(x=train_X, y = train_Y, batch_size=100, epochs=30)
28 modelo1.evaluate(test_X, test_Y)

```

Model: "sequential_27"

| Layer (type) | Output Shape | Param |
|---|--------------------|-------|
| conv2d_109 (Conv2D) | (None, 64, 64, 16) | 44 |
| dropout_10 (Dropout) | (None, 64, 64, 16) | |
| max_pooling2d_24 (MaxPooling2D) | (None, 32, 32, 16) | |
| conv2d_110 (Conv2D) | (None, 28, 28, 16) | 6,41 |
| max_pooling2d_25 (MaxPooling2D) | (None, 14, 14, 16) | |
| conv2d_111 (Conv2D) | (None, 12, 12, 32) | 4,64 |
| conv2d_112 (Conv2D) | (None, 10, 10, 64) | 18,49 |
| max_pooling2d_26 (MaxPooling2D) | (None, 5, 5, 64) | |
| conv2d_113 (Conv2D) | (None, 3, 3, 8) | 4,61 |
| flatten_27 (Flatten) | (None, 72) | |
| dense_104 (Dense) | (None, 256) | 18,68 |
| dropout_11 (Dropout) | (None, 256) | |
| batch_normalization_13 (BatchNormalization) | (None, 256) | 1,02 |
| dense_105 (Dense) | (None, 64) | 16,44 |
| dense_106 (Dense) | (None, 1) | 1 |

| | | |
|-------------------|-----------|---|
| dense_100 (dense) | (none, 1) | c |
|-------------------|-----------|---|

Total params: 70,841 (276.72 KB)
Trainable params: 70,329 (274.72 KB)
Non-trainable params: 512 (2.00 KB)

Epoch 1/30

3/3 ————— 9s 2s/step - accuracy: 0.5441 - loss: 0.7175

Epoch 2/30

3/3 ————— 0s 9ms/step - accuracy: 0.6356 - loss: 0.6505

Epoch 3/30

3/3 ————— 0s 9ms/step - accuracy: 0.7056 - loss: 0.5557

Epoch 4/30

3/3 ————— 0s 9ms/step - accuracy: 0.6368 - loss: 0.6205

Epoch 5/30

3/3 ————— 0s 10ms/step - accuracy: 0.6623 - loss: 0.5775

Epoch 6/30

3/3 ————— 0s 9ms/step - accuracy: 0.7175 - loss: 0.5189

Epoch 7/30

3/3 ————— 0s 9ms/step - accuracy: 0.7297 - loss: 0.4999

Epoch 8/30

3/3 ————— 0s 9ms/step - accuracy: 0.7260 - loss: 0.4817

Epoch 9/30

3/3 ————— 0s 9ms/step - accuracy: 0.7897 - loss: 0.4373

Epoch 10/30

3/3 ————— 0s 9ms/step - accuracy: 0.8094 - loss: 0.4226

Epoch 11/30

3/3 ————— 0s 9ms/step - accuracy: 0.8228 - loss: 0.4042

Epoch 12/30

3/3 ————— 0s 9ms/step - accuracy: 0.7950 - loss: 0.4070

Epoch 13/30

3/3 ————— 0s 9ms/step - accuracy: 0.7788 - loss: 0.4181

Epoch 14/30

3/3 ————— 0s 9ms/step - accuracy: 0.7825 - loss: 0.4126

Epoch 15/30

3/3 ————— 0s 11ms/step - accuracy: 0.8326 - loss: 0.3730

Epoch 16/30

3/3 ————— 0s 8ms/step - accuracy: 0.8175 - loss: 0.4072

Epoch 17/30

3/3 ————— 0s 9ms/step - accuracy: 0.8276 - loss: 0.3803

Epoch 18/30

3/3 ————— 0s 9ms/step - accuracy: 0.8375 - loss: 0.3612

Epoch 19/30

3/3 ————— 0s 9ms/step - accuracy: 0.8534 - loss: 0.3483

Epoch 20/30

3/3 ————— 0s 9ms/step - accuracy: 0.8680 - loss: 0.3543

Epoch 21/30

3/3 ————— 0s 9ms/step - accuracy: 0.8643 - loss: 0.3284

Epoch 22/30

3/3 ————— 0s 9ms/step - accuracy: 0.8632 - loss: 0.3566

Epoch 23/30

3/3 ————— 0s 10ms/step - accuracy: 0.8495 - loss: 0.3425

Epoch 24/30

3/3 ————— 0s 10ms/step - accuracy: 0.8411 - loss: 0.3464

Epoch 25/30

3/3 ————— 0s 10ms/step - accuracy: 0.8814 - loss: 0.3167

```

Epoch 26/30
3/3 ————— 0s 10ms/step - accuracy: 0.8730 - loss: 0.3125
Epoch 27/30
3/3 ————— 0s 9ms/step - accuracy: 0.8556 - loss: 0.3339
Epoch 28/30
3/3 ————— 0s 8ms/step - accuracy: 0.8581 - loss: 0.3200
Epoch 29/30
3/3 ————— 0s 12ms/step - accuracy: 0.8838 - loss: 0.3040
Epoch 30/30
3/3 ————— 0s 9ms/step - accuracy: 0.8729 - loss: 0.2933
2/2 ————— 1s 200ms/step - accuracy: 0.7300 - loss: 0.7985
[0.8686403632164001, 0.7200000286102295]

```

▼ Modelo 2 (10pt)

```

1 modelo2 = models.Sequential([
2     layers.InputLayer(shape=(input_size)),
3     Conv2D(filters=8, kernel_size=(3, 3), activation='relu', padding="same"),
4     AveragePooling2D((2,2)),
5     Conv2D(filters=32, kernel_size=(3, 3), activation='relu', padding="same"),
6     AveragePooling2D((2,2)),
7     Conv2D(filters=64, kernel_size=(3, 3), activation='relu', padding="same"),
8     MaxPool2D(),
9     Conv2D(filters=128, kernel_size=(3, 3), activation='relu', padding="valid"),
10
11
12     layers.Flatten(),
13
14     Dense(128, activation="relu"),
15     keras.layers.Dropout(0.2),
16     Dense(64, activation='relu'),
17     Dense(32, activation='relu'),
18     Dense(1, activation = 'sigmoid')
19
20 ])
21 modelo2.summary()
22
23 modelo2.compile(optimizer='adamw', loss='binary_crossentropy', metrics = ["accuracy"])
24 modelo2.fit(x=train_X, y = train_Y, batch_size=100, epochs=30)
25 modelo2.evaluate(test_X, test_Y)

```

Model: "sequential_34"

| Layer (type) | Output Shape | Param |
|----------------------|-------------------|-------|
| conv2d_136 (Conv2D) | (None, 64, 64, 8) | 22 |
| average_pooling2d_10 | (None, 32, 32, 8) | |

| | | |
|---|--------------------|--------|
| (AveragePooling2D) | | |
| conv2d_137 (Conv2D) | (None, 32, 32, 32) | 2,33 |
| average_pooling2d_11 (AveragePooling2D) | (None, 16, 16, 32) | |
| conv2d_138 (Conv2D) | (None, 16, 16, 64) | 18,49 |
| max_pooling2d_33 (MaxPooling2D) | (None, 8, 8, 64) | |
| conv2d_139 (Conv2D) | (None, 6, 6, 128) | 73,85 |
| flatten_34 (Flatten) | (None, 4608) | |
| dense_131 (Dense) | (None, 128) | 589,95 |
| dropout_18 (Dropout) | (None, 128) | |
| dense_132 (Dense) | (None, 64) | 8,25 |
| dense_133 (Dense) | (None, 32) | 2,08 |
| dense_134 (Dense) | (None, 1) | 3 |

Total params: 695,233 (2.65 MB)

Trainable params: 695,233 (2.65 MB)

Non-trainable params: 0 (0.00 B)

Epoch 1/30

3/3 ————— 8s 1s/step - accuracy: 0.4553 - loss: 5.8999

Epoch 2/30

3/3 ————— 0s 7ms/step - accuracy: 0.6081 - loss: 3.6671

Epoch 3/30

3/3 ————— 0s 7ms/step - accuracy: 0.4482 - loss: 1.0288

Epoch 4/30

3/3 ————— 0s 7ms/step - accuracy: 0.6150 - loss: 0.7978

Epoch 5/30

3/3 ————— 0s 6ms/step - accuracy: 0.5914 - loss: 0.7054

Epoch 6/30

3/3 ————— 0s 7ms/step - accuracy: 0.4536 - loss: 0.7848

Epoch 7/30

3/3 ————— 0s 7ms/step - accuracy: 0.6600 - loss: 0.6271

Epoch 8/30

3/3 ————— 0s 7ms/step - accuracy: 0.6568 - loss: 0.6438

Epoch 9/30

3/3 ————— 0s 7ms/step - accuracy: 0.6649 - loss: 0.6074

Epoch 10/30

3/3 ————— 0s 9ms/step - accuracy: 0.6872 - loss: 0.5556

Epoch 11/30

3/3 ————— 0s 10ms/step - accuracy: 0.6919 - loss: 0.5523

Epoch 12/30

3/3 ————— 0s 8ms/step - accuracy: 0.6700 - loss: 0.5539

Epoch 13/30


















3/3 ————— 0s 7ms/step - accuracy: 0.6955 - loss: 0.4962

Epoch 14/30

3/3 ————— 0s 8ms/step - accuracy: 0.7679 - loss: 0.4658

Epoch 15/30

```

3/3  0s 8ms/step - accuracy: 0.7068 - loss: 0.5041
Epoch 16/30
3/3  0s 7ms/step - accuracy: 0.7616 - loss: 0.4621
Epoch 17/30
3/3  0s 7ms/step - accuracy: 0.7600 - loss: 0.4644
Epoch 18/30
3/3  0s 7ms/step - accuracy: 0.8206 - loss: 0.4214
Epoch 19/30
3/3  0s 8ms/step - accuracy: 0.7642 - loss: 0.4275
Epoch 20/30
3/3  0s 8ms/step - accuracy: 0.8044 - loss: 0.4142
Epoch 21/30
3/3  0s 7ms/step - accuracy: 0.8045 - loss: 0.3725
Epoch 22/30
3/3  0s 7ms/step - accuracy: 0.8483 - loss: 0.3391
Epoch 23/30
3/3  0s 7ms/step - accuracy: 0.8508 - loss: 0.3126
Epoch 24/30
3/3  0s 7ms/step - accuracy: 0.8239 - loss: 0.3132
Epoch 25/30
3/3  0s 7ms/step - accuracy: 0.7959 - loss: 0.4159
Epoch 26/30
3/3  0s 8ms/step - accuracy: 0.8619 - loss: 0.3037
Epoch 27/30
3/3  0s 7ms/step - accuracy: 0.9070 - loss: 0.2564
Epoch 28/30
3/3  0s 9ms/step - accuracy: 0.8694 - loss: 0.2900
Epoch 29/30
3/3  0s 8ms/step - accuracy: 0.8825 - loss: 0.2467
Epoch 30/30
3/3  0s 10ms/step - accuracy: 0.8959 - loss: 0.2446
2/2  1s 412ms/step - accuracy: 0.7567 - loss: 0.6796
[0.6782644391059875, 0.7599999904632568]

```

▼ Avaliando o modelo que você criou (10pt)

O que você consegue analisar olhando os modelos que você criou e o modelo proposto? Essa análise pode envolver custo computacional, memória, etc.

Ambos os modelos apresentam resultados semelhantes de acurácia, sendo que o segundo obteve resulta

