Visualizar rede convolucional

1. Introdução

Os modelos de redes neurais funcionam como “caixas pretas”, onde não é possível saber como eles chegaram às respostas apresentadas. Isto levanta a seguinte questão: “Como você pode confiar nas decisões de um modelo se você não consegue saber como ele chegou lá?”. Exponho abaixo algumas ideias apresentadas em [https://www.pyimagesearch.com/2020/03/09/grad-cam-visualize-class-activation-maps-with-keras-tensorflow-and-deep-learning/].

Esse site conta uma “lenda urbana” que diz que um grupo de pesquisadores estava querendo detectar tanques camuflados em fotos, com financiamento do exército americano. Para isso, coletaram um banco de dados com 200 imagens, 100 com tanques camuflados e 100 sem tanques contendo somente árvores e florestas. Os pesquisadores pegaram 50+50 imagens para treino e 50+50 imagens para teste, treinaram uma rede convolucional e testaram, obtendo taxa de acerto de 100%. Isto os deixou muito felizes. Porém, quando fizeram testes em campo, o desempenho do sistema era muito ruim. Voltaram para laboratório, mudaram o modelo de rede, fizeram treinos novamente e obtiveram novamente outro sistema com 100% de acerto no laboratório, mas que novamente não funcionava em campo. Após várias tentativas, um dos pesquisadores observou que todas as fotos com tanques camuflados foram tiradas em dias ensolarados enquanto que todas as fotos sem tanques foram tiradas em dias nublados. O grupo tinha criado um “detector de nuvens”. O site diz que esta história é falsa. Porém, situações bastante semelhantes realmente acontecem no desenvolvimento prático de sistemas inteligentes. Para evitar histórias como acima, o desenvolvedor deve entender o que está acontecendo dentro da rede neural.

Na aula “convkeras-ead”, vimos como visualizar as convoluções e as ativações das primeiras camadas. Nesta aula, vamos ver como visualizar os atributos das camadas de topo.

Uma forma de entender o funcionamento de uma rede é verificar se a rede estava “prestando atenção” para a região correta da imagem para tomar a decisão. Veja a figura 1. A saída da categoria “gato” mostra a região da imagem que fez a rede dizer que a imagem é de gato, e a saída da categoria “cachorro” mostra a região que fez a rede tomar a decisão de que a imagem é de cachorro. Se os pesquisadores acima tivessem utilizado uma técnica dessas, certamente teriam notado que a rede não estava olhando para o tanque camuflado mas para as nuvens.

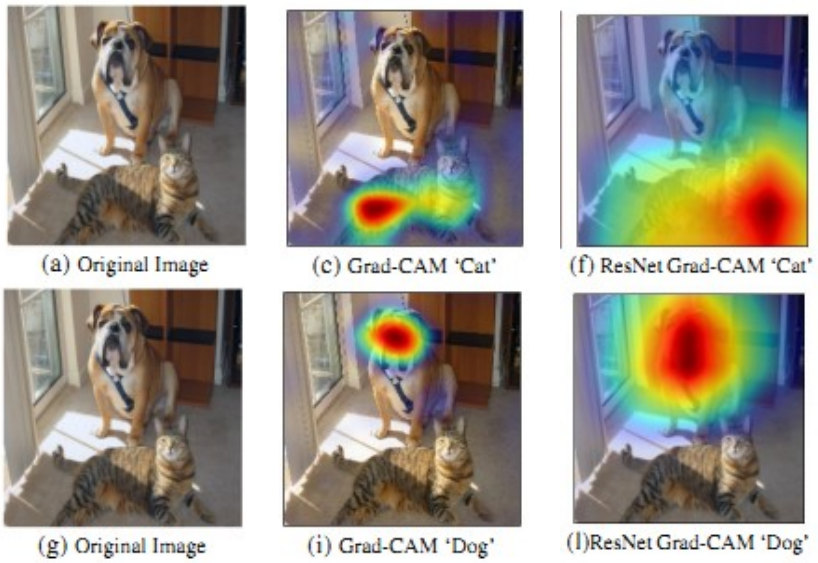

Figura 1: (c, i) Saída de Grad-CAM das categorias "gato" e "cachorro" usando VGG-16. (f-l) Usando ResNet-18. [Selvaraju2019]

2. Class Activation Map (CAM)

Class Activation Map (CAM) foi introduzido no artigo [Zhou2015]. A figura 2 mostra a região para onde a rede "estava olhando" para decidir que a ação nas imagens é de escovar dentes.

Figura 2: Exemplo de CAM [Zhou2015].

O site [https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-class-activation-maps-fe94eda4cef1] aplica CAM ao bem-conhecido banco de dados MNIST. Ele utiliza uma rede “semelhante à VGG” (onde todas as convoluções são 3×3) para classificar dígitos de MNIST e aplicar CAM. Porém, em vez de ter camadas densas no topo de rede (como na VGG), há uma camada AveragePooling. O código original está disponível em [https://github.com/Divyanshupy/CLASS-ACTIVATION-MAPS].

O programa 1 (cam1-train.py) treina a rede cam1.h5 que classifica MNIST. Ele atinge acuracidade de 99,3%.

```
#cam1-train.py - testada em Colab com TF2
#https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-class-activation-maps-fe94eda4cef1
#https://github.com/Divyanshupy
#cada epoch demora 8s e chega a acuracidade de 0.9927

import tensorflow.keras as keras
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential,Model,load_model
from keras.layers import Dense,Conv2D,Flatten,MaxPooling2D,GlobalAveragePooling2D
from keras.utils import plot_model
from keras import optimizers
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
nomeprog="cam1"

def modelo():
  model = Sequential()
  model.add(Conv2D(16,input_shape=(28,28,1),kernel_size=(3,3),activation='relu',padding='same'))
  model.add(MaxPooling2D(pool_size=(2,2)))

  model.add(Conv2D(32,kernel_size=(3,3),activation='relu',padding='same'))
  model.add(MaxPooling2D(pool_size=(2,2)))

  model.add(Conv2D(64,kernel_size=(3,3),activation='relu',padding='same'))
  model.add(MaxPooling2D())

  model.add(Conv2D(128,kernel_size=(3,3),activation='relu',padding='same'))
  model.add(GlobalAveragePooling2D())
  model.add(Dense(10,activation='softmax'))

  model.compile(loss='categorical_crossentropy',metrics=['accuracy'],optimizer=optimizers.Adam())
  return model

(AX,AY),(QX,QY)  = mnist.load_data()
AX = AX.reshape((AX.shape[0],AX.shape[1],AX.shape[2],1))
QX = QX.reshape((QX.shape[0],QX.shape[1],QX.shape[2],1))
AX = 1 - AX.astype('float')/255; QX  = 1 - QX.astype('float')/255
nclasses =10
AY = keras.utils.to_categorical(AY, nclasses)
QY = keras.utils.to_categorical(QY, nclasses)

#Escolha entre treinar do zero ou carregar modelo ja treinado
model=modelo()
#model=load_model(nomeprog+".h5")

model.summary()
plot_model(model, to_file=nomeprog+'.png', show_shapes=True)

model.fit(AX, AY, batch_size=100, epochs=30, validation_split=0.1, shuffle=True)
model.evaluate(QX, QY)
model.save(nomeprog+'.h5')
```

Programa 1: Gera rede que classifica MNIST, com maxpooling.
https://colab.research.google.com/drive/1tRKchdpRB11igFupczeAc55u7YoE8dlr?usp=sharing

O programa 2 (cam1-test.py) usa a rede cam1.h5 para fazer predição e mostrar CAM.

```
#cam1-test.py - testado em Colab com TF2
import tensorflow.keras as keras
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import load_model, Model
from tensorflow.keras.utils import plot_model
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
import scipy as sp
import cv2

nomeprog="cam1"

(AX,AY),(QX,QY)  = mnist.load_data()
AX = AX.reshape((AX.shape[0],AX.shape[1],AX.shape[2],1))
QX = QX.reshape((QX.shape[0],QX.shape[1],QX.shape[2],1))
AX = 1 - AX.astype('float')/255; QX  = 1 - QX.astype('float')/255
nclasses =10
AY = keras.utils.to_categorical(AY, nclasses)
QY = keras.utils.to_categorical(QY, nclasses)

model=load_model(nomeprog+'.h5')
gap_weights = model.layers[-1].get_weights()[0]
print("gap_weights.shape",gap_weights.shape)

cam_model = Model( inputs=model.input, outputs=(model.layers[-3].output, model.layers[-1].output) )
print("Nome do layer de features:",model.layers[-3].name)
print("Nome do layer de results:", model.layers[-1].name)

features,results = cam_model.predict(QX)
print("features.shape=",features.shape,"results.shape=",results.shape)
print()

for idx in range(10):
  pred = np.argmax(results[idx])
  print('Predicted Class = ' +str( pred )+ ', Probability = ' + str(results[idx][pred]))
  cam_features = features[idx,:,:,:]; print("cam_features.shape",cam_features.shape)
  cam_weights = gap_weights[:,pred];  print("cam_weights.shape",cam_weights.shape)
  cam_output = np.dot(cam_features,cam_weights)
  cam_output = cv2.resize(cam_output,(28,28),interpolation=cv2.INTER_NEAREST)
  cam_output = cv2.normalize(cam_output, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX)

  plt.figure(facecolor='white')
  plt.imshow(np.squeeze(QX[idx],-1), cmap='gray', alpha=0.4)
  plt.colorbar(shrink=0.5)
  plt.imshow(cam_output, cmap='inferno', alpha=0.6)
  plt.colorbar(shrink=0.5)
  plt.axis('off')
  plt.show()
```

Programa 2: Gera CAM (Class Activation Map) usando a rede gerada pelo programa 1.
https://colab.research.google.com/drive/1tRKchdpRB11igFupczeAc55u7YoE8dlr?usp=sharing

A estrutura da rede cam1.h5 está ilustrada na figura 3 e as últimas camadas estão na figura 4. Ao classificar uma imagem, a rede produz atributos *cam_features* no formato 3×3×128. Esses atributos passam por average pooling, resultando num vetor de 128 elementos que por sua vez passa por uma rede densa resultando em 10 saídas (figuras 3 e 4).

Os 128 atributos são multiplicados por 128 pesos w_c dentro da camada densa (*gap_weights* no programa) para resultar em cada uma das 10 saídas da rede. Vamos supor que a saída y^7 foi a maior, sendo a imagem classificada como c=7 (*pred* no programa). Neste caso, para gerar CAM, devemos pegar os 128 pesos w_7 (*cam_weights* correspondentes à saída "7") e calcular o produto escalar com os atributos 128×3×3 (*cam_features*) para resultar na imagem CAM 3×3 (*cam_output*).

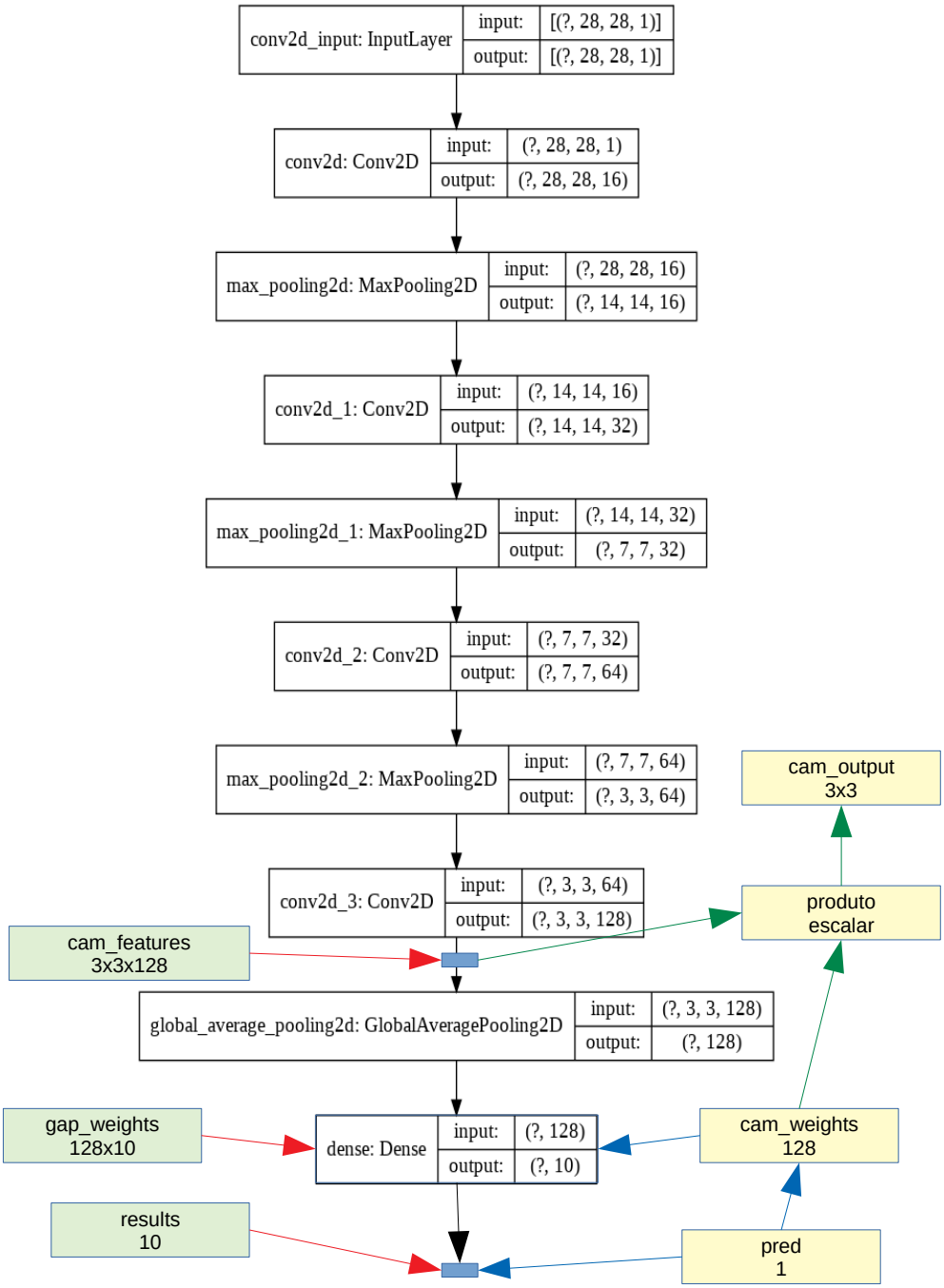

Figura 3: Estrutura da rede usada pelos programas 1 e 2.

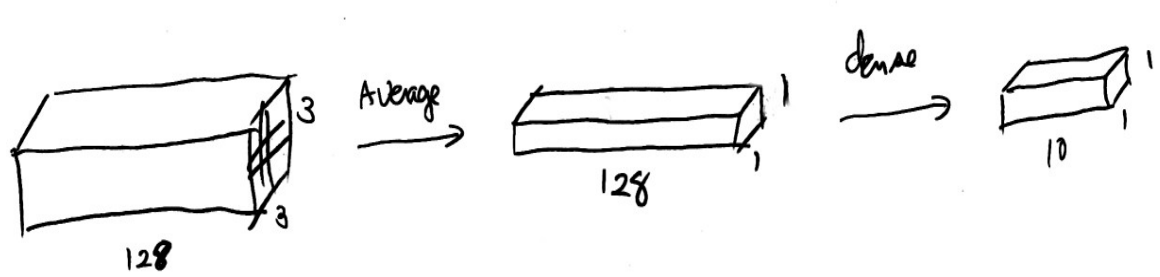

Figura 4: As últimas camadas da rede.

O problema deste processo é que CAM gerada é de baixíssima resolução, com 3×3 pixels, tornando o resultado inútil (figura 5). O programa original do site faz interpolação bicúbica que faz parecer que CAM possui resolução maior, mas não a impressão é falsa.

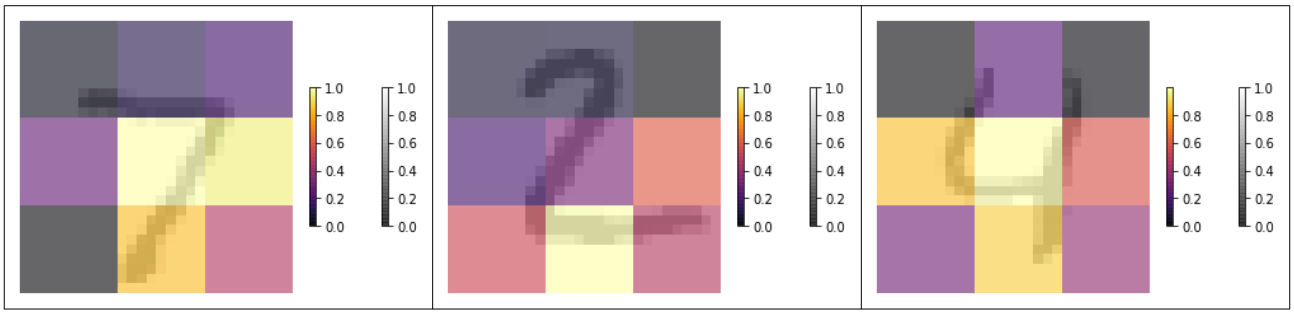

Figura 5: CAM geradas pelos programas 1 e 2 (cam1*.py) sobrepostas às imagens originais.

Como podemos aumentar a resolução de CAM? Uma solução é modificar a estrutura da rede, para que os atributos antes de average pooling possuam alta resolução. O programa 3 faz isso, onde as camadas *maxpooling* do programa 1 foram eliminadas. Assim, os atributos *cam_features* têm dimensão 28×28×128, o que faz gerar imagem CAM 28×28. O programa 4 faz predição usando a rede cam2.h5 e gera CAM com resolução 28×28. A taxa de acerto de teste obtida é 98,8%. Muito provavelmente, seria possível obter taxa de acerto maior acertando melhor os parâmetros da rede ou treinando por mais épocas.

A figura 7 mostra algumas imagens CAM sobrepostas às imagens de entrada. Em todas as imagens, a rede estava prestando atenção aos pixels pretos e vizinhança. Não estava prestando atenção aos pixels brancos do fundo. Para reconhecer o dígito “7”, a rede estava prestando atenção na intersecção entre linhas “horizontal” e “diagonal secundário” e também na extremidade inferior da linha diagonal. Para reconhecer dígito “2”, a rede prestou atenção na curva superior e na interseção entre “diagonal secundário” e “linha horizontal”. Para reconhecer o dígito “4”, a rede prestou atenção nos dois “braços abertos” e na extremidade inferior da “reta vertical”.

```
#cam2-train.py
#https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-class-activation-maps-fe94eda4cef1
#https://github.com/Divyanshupy
#Cada epoca demora 35s e chega a acuracidade de 0.9812

import tensorflow.keras as keras
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential,Model,load_model
from keras.layers import Dense,Conv2D,Flatten,MaxPooling2D,GlobalAveragePooling2D
from keras.utils import plot_model
from keras import optimizers
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
nomeprog="cam2"

def modelo():
  model = Sequential()
  model.add(Conv2D(16,input_shape=(28,28,1),kernel_size=(3,3),activation='relu',padding='same'))
  model.add(Conv2D(32,kernel_size=(3,3),activation='relu',padding='same'))
  model.add(Conv2D(64,kernel_size=(3,3),activation='relu',padding='same'))
  model.add(Conv2D(128,kernel_size=(3,3),activation='relu',padding='same'))
  model.add(GlobalAveragePooling2D())
  model.add(Dense(10,activation='softmax'))

  model.compile(loss='categorical_crossentropy',metrics=['accuracy'],optimizer='adam')
  return model

(AX,AY),(QX,QY)  = mnist.load_data()
AX = AX.reshape((AX.shape[0],AX.shape[1],AX.shape[2],1))
QX = QX.reshape((QX.shape[0],QX.shape[1],QX.shape[2],1))
AX = 1 - AX.astype('float')/255; QX  = 1 - QX.astype('float')/255
nclasses =10
AY = keras.utils.to_categorical(AY, nclasses)
QY = keras.utils.to_categorical(QY, nclasses)

#Escolha entre treinar do zero ou carregar modelo ja treinado
model=modelo()
#model=load_model(nomeprog+".h5")

model.summary()
plot_model(model, to_file=nomeprog+'.png', show_shapes=True)

model.fit(AX, AY, batch_size=100, epochs=30, validation_split=0.1, shuffle=True)
model.evaluate(QX, QY)
model.save(nomeprog+'.h5')
```

Programa 3: Treina rede cam2.h5. https://colab.research.google.com/drive/1amK-1rHAOHxL4LxiHFyBCI_g906hbTKs?usp=sharing

```python
#cam2-test.py aplica CAM na rede ja treinada
import tensorflow.keras as keras
from keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from keras.models import load_model
from keras.utils import plot_model
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
import scipy as sp
import cv2

nomeprog="cam2"

(AX,AY),(QX,QY)  = mnist.load_data()
AX = AX.reshape((AX.shape[0],AX.shape[1],AX.shape[2],1))
QX = QX.reshape((QX.shape[0],QX.shape[1],QX.shape[2],1))
AX = 1 - AX.astype('float')/255; QX  = 1 - QX.astype('float')/255
nclasses =10
AY = keras.utils.to_categorical(AY, nclasses)
QY = keras.utils.to_categorical(QY, nclasses)

model=load_model(nomeprog+'.h5')
gap_weights = model.layers[-1].get_weights()[0]
print("gap_weights.shape",gap_weights.shape)

cam_model = Model( inputs=model.input, outputs=(model.layers[-3].output, model.layers[-1].output) )
print("Nome do layer de features:",model.layers[-3].name)
print("Nome do layer de results:", model.layers[-1].name)

features,results = cam_model.predict(QX)
print("features.shape",features.shape)
print("results.shape",results.shape)
print()

for idx in range(10):
  features_for_one_img = features[idx,:,:,:]
  print("features_for_one_img.shape",features_for_one_img.shape)
  height_roomout = AX.shape[1]/features_for_one_img.shape[0]
  width_roomout  = AX.shape[2]/features_for_one_img.shape[1]
  #print("height_roomout", height_roomout, "width_roomout", width_roomout)

  cam_features = sp.ndimage.zoom(features_for_one_img, (height_roomout, width_roomout, 1), order=1)
  print("cam_features.shape",cam_features.shape)
  pred = np.argmax(results[idx])

  cam_weights = gap_weights[:,pred]
  print("cam_weights.shape",cam_weights.shape)
  cam_output  = np.dot(cam_features,cam_weights)
  print('Predicted Class = ' +str( pred )+ ', Probability = ' + str(results[idx][pred]))

  plt.figure(facecolor='white')
  plt.imshow(np.squeeze(QX[idx],-1), cmap='gray', alpha=0.4)
  plt.colorbar(shrink=0.5)
  cam_output = cv2.normalize(cam_output, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX)
  plt.imshow(cam_output, cmap='inferno', alpha=0.6)
  plt.colorbar(shrink=0.5)
  plt.axis('off')
  plt.show()
```

Programa 4: Faz predição com rede cam2.h5 e gera CAM.

https://colab.research.google.com/drive/1amK-1rHAOHxL4LxiHFyBCI_g906hbTKs?usp=sharing

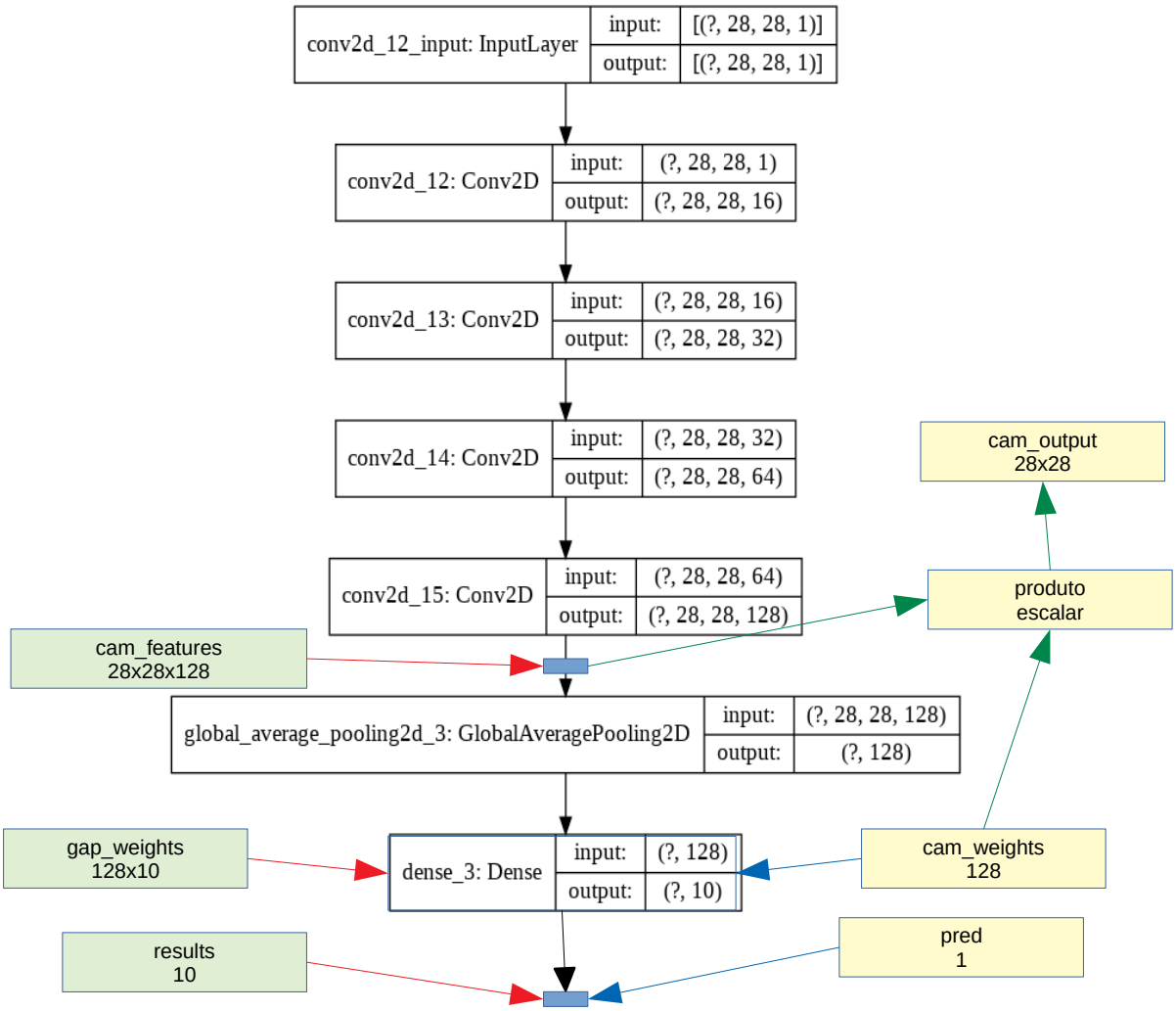

Figura 6: Estrutura da rede cam2.h5

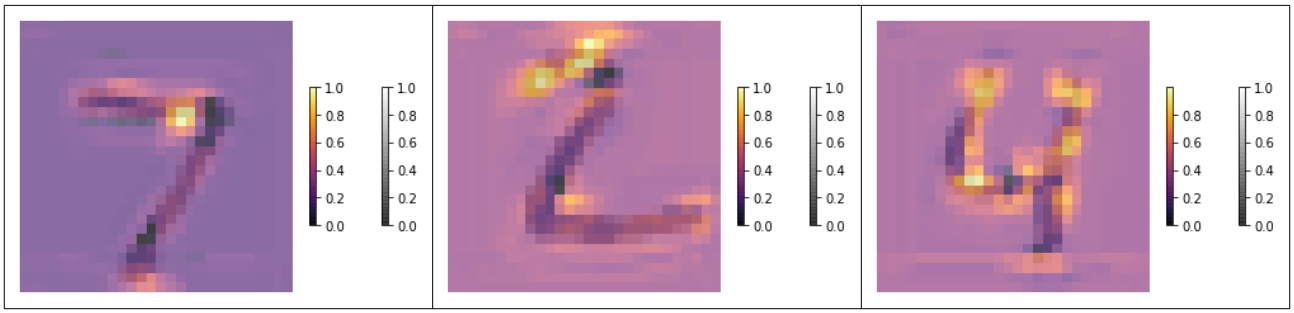

Figura 7: CAM geradas pelos programas 3 e 4 (cam2*.py) sobrepostas às imagens originais.

3. Grad-CAM (gradient-weighted class activation mapping)

Note que, para gerar CAM (class activation map), tivemos que criar um modelo de rede especial e treiná-lo, pois não é possível gerar CAM (por exemplo) da rede "tipo LeNet" nem "tipo VGG" onde há várias camadas densas antes da camada de saída final. Um dos defeitos de CAM é que ele requer que mapa de atributos da última camada convolucional preceda diretamente camada softmax, isto é, uma estrutura como:

mapa de atributos → global average pooling → camada densa → softmax.

Vamos chamar esta estrutura de "estrutura CAM". Note que as redes dos programas 1-4 possuem esta estrutura. Porém, muitas outras redes não possuem esta estrutura.

Seria possível visualizar CAM sem ter que alterar a rede para se conformar à "estrutura CAM" e sem ter que retreiná-la? O artigo [Selvaraju2017] introduz Grad-CAM que permite fazer isso. Esta técnica não requer qualquer modificação ao modelo existente e permite aplicá-la a praticamente qualquer rede neural convolucional, incluindo aquelas usadas para legendar imagens e para responder questões visuais [https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-gradcam-554a85dd4e48].

Os atributos de topo (mais pertos da saída) da rede convolucional capturam melhor as informações de alto nível. Por outro lado, camadas convolucionais naturalmente retém informações espaciais que são perdidas em camadas densas. Assim, podemos esperar que a última camada convolucional possua a melhor informação semântica sem perder a informação espacial.

Grad-CAM utiliza a informação da gradiente que flui pela última camada convolucional para entender a importância de cada neurônio pela decisão de interesse. Qualquer implementação de rede neural precisa calcular esses gradientes para efetuar retro-propagação, de forma que não há "trabalho extra" para gerar Grad-CAM.

Vamos supor que a última camada convolucional gere n mapas de atributos $u \times v$. A intuição de Grad-CAM é que os mapas de atributos que mais contribuem para classificar a imagem como classe c devem ter derivadas parciais (em relação à saída da classe c) altas. Calculando a média ponderada dos n mapas de atributos usando as derivadas parciais como pesos, é possível determinar as regiões para onde a rede "estava olhando" para classificar a imagem como classe c.

Para obter o mapa Grad-CAM $L^c_{Grad-CAM} \in \mathbb{R}^{u \times v}$ para alguma classe c, Grad-CAM calcula o gradiente da saída final y^c da classe c em relação a cada mapa de ativação A^k da última camada convolucional, isto é, $\partial y^c / \partial A^k$. Na figura 11, mapa de ativação A possui dimensão u×v×n = 14×14×512 e cada uma das k $\partial y^c / \partial A^k$ possui dimensão 14×14.

Nota: O artigo original sugere pegar a saída antes do softmax. Porém, em Keras, normalmente a última camada consiste de camadas densa e softmax sobrepostas, de forma que (eu saiba) não deve ser possível pegar a saída antes do softmax de uma rede já treinada. Porém, provavelmente não há diferença em pegar o gradiente antes ou depois do softmax. A própria implementação-exemplo que fica no site de Keras pega a saída depois de softmax [https://keras.io/examples/vision/grad_cam/].

Grad-CAM calcula as médias globais (global average pool) desses gradientes sobre as dimensões de largura e altura para obter o peso α^c_k da importância do mapa de atributos k para a classe c:

$$\alpha^c_k = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A^k_{ij}} \qquad (1)$$

Na figura 11, Z é 14×14 e, dada uma classe c, existem 512 α^c_k.

Depois, efetua a combinação ponderada dos mapas de atributos usando os pesos calculados acima, seguida pelo ReLU, para obter Grad-CAM ou “heatmap”:

$$L^c_{Grad-CAM} = ReLU\left(\sum_k \alpha^c_k A^k\right) \qquad (2)$$

Na figura 11, heatmap possui dimensão 14×14. Grad-CAM aplica ReLU à combinação linear dos mapas, pois só interessam aqueles atributos que possuem influência positiva na classe c, isto é, os atributos cuja intensidade deve aumentar para aumentar y^c.

Há uma implementação de Grad-CAM entre os exemplos de Keras [https://keras.io/examples/vision/grad_cam/]. Ela usa o modelo pré-treinado Xception. O “problema” é que no Xception mapa de atributos da última camada convolucional precede diretamente a predição, como na “estrutura CAM” (figuras 8 e 9). Portanto, poderia ter usado simplesmente CAM para visualizar a ativação de Xception. Alterei esse programa para usar rede VGG16, que não se encaixa na “estrutura CAM” (e portanto não dá para usar CAM para visualizar a ativação). O modelo VGG16 está nas figuras 10 e 11. Repare que entre predição e última camada convolucional há duas camadas densas.

```
Model: "xception"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 299, 299, 3) 0
__________________________________________________________________________________________________
block1_conv1 (Conv2D)           (None, 149, 149, 32) 864         input_1[0][0]
__________________________________________________________________________________________________

__________________________________________________________________________________________________
(...)
__________________________________________________________________________________________________
block14_sepconv1 (SeparableConv (None, 10, 10, 1536) 1582080     add_11[0][0]
__________________________________________________________________________________________________
block14_sepconv1_bn (BatchNorma (None, 10, 10, 1536) 6144        block14_sepconv1[0][0]
__________________________________________________________________________________________________
block14_sepconv1_act (Activatio (None, 10, 10, 1536) 0           block14_sepconv1_bn[0][0]
__________________________________________________________________________________________________
block14_sepconv2 (SeparableConv (None, 10, 10, 2048) 3159552     block14_sepconv1_act[0][0]
__________________________________________________________________________________________________
block14_sepconv2_bn (BatchNorma (None, 10, 10, 2048) 8192        block14_sepconv2[0][0]
__________________________________________________________________________________________________
block14_sepconv2_act (Activatio (None, 10, 10, 2048) 0           block14_sepconv2_bn[0][0]
__________________________________________________________________________________________________
avg_pool (GlobalAveragePooling2 (None, 2048)         0           block14_sepconv2_act[0][0]
__________________________________________________________________________________________________
predictions (Dense)             (None, 1000)         2049000     avg_pool[0][0]
==================================================================================================
```

Figura 8: Parte do modelo Xception.

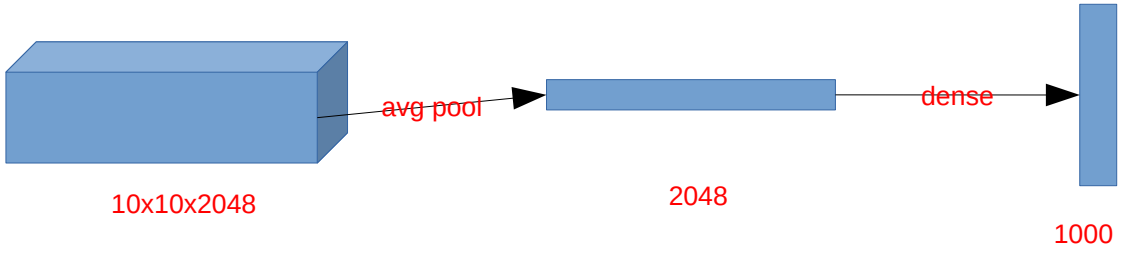

Figura 9: As últimas camadas de Xception.

Exercício: Gere CAM da imagem *hummingbird.jpg* (figura 12) usando modelo Xception.

```
Model: "vgg16"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_17 (InputLayer)        [(None, 224, 224, 3)]     0         [Entrada do last_conv_layer_model]
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808   [Saída do last_conv_layer_model]
                                                                 [Entrada do classifier_model]
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000   [Saída do classifier]
=================================================================
```

Figura 10: Modelo VGG16.

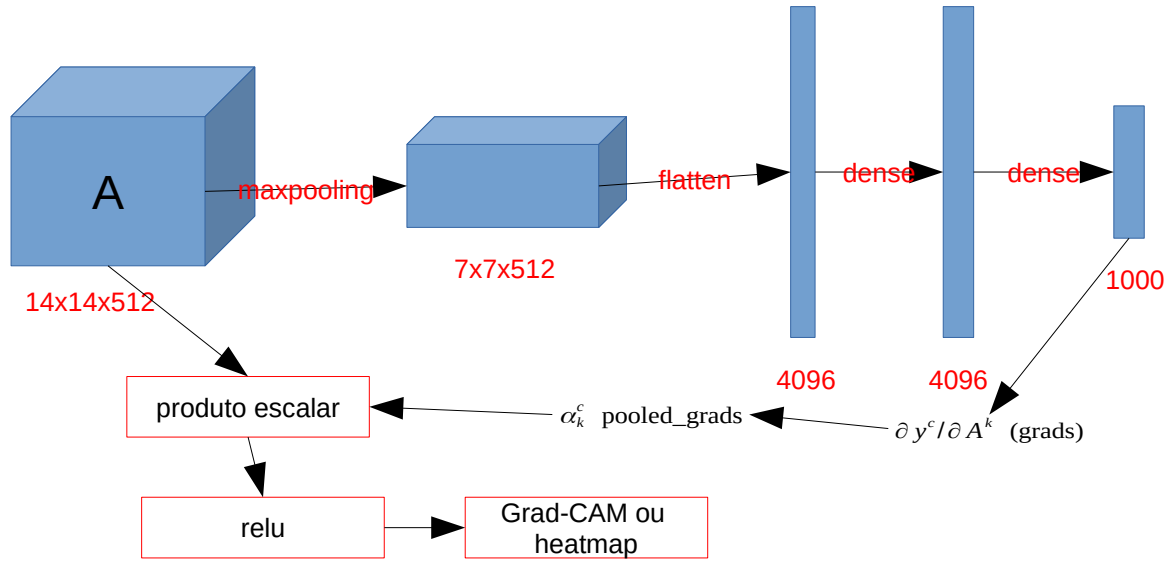

Figura 11: As últimas camadas de VGG16 e o cálculo de Grad-CAM.

```python
#gradcam_vgg.py - testada em Colab com TF2
#Adaptado de https://keras.io/examples/vision/grad_cam/
import numpy as np
import tensorflow as tf
from tensorflow import keras
# from IPython.display import Image
import matplotlib.pyplot as plt
import matplotlib.cm as cm

model_builder = keras.applications.vgg16.VGG16
img_size = (224, 224)
preprocess_input = keras.applications.vgg16.preprocess_input
decode_predictions = keras.applications.vgg16.decode_predictions

last_conv_layer_name = "block5_conv3"
classifier_layer_names = ["block5_pool", "flatten", "fc1", "fc2", "predictions"]

img_path="hummingbird.jpg"
img=keras.preprocessing.image.load_img(img_path)
plt.imshow(img); plt.axis("off"); plt.show()

def get_img_array(img_path, size):
  # `img` is a PIL image of size 299x299
  img = keras.preprocessing.image.load_img(img_path, target_size=size)
  # `array` is a float32 Numpy array of shape (299, 299, 3)
  array = keras.preprocessing.image.img_to_array(img)
  # We add a dimension to transform our array into a "batch"
  # of size (1, 299, 299, 3)
  array = np.expand_dims(array, axis=0)
  return array

def make_gradcam_heatmap(img_array, model,
                         last_conv_layer_name, classifier_layer_names):
  # First, we create a model that maps the input image to the activations
  # of the last conv layer
  last_conv_layer = model.get_layer(last_conv_layer_name)
  last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)

  #Imprime o modelo da rede ate a ultima camada convolucional
  print("\nlast_conv_layer_model:"); last_conv_layer_model.summary(); print()

  # Second, we create a model that maps the activations of the last conv
  # layer to the final class predictions
  classifier_input = keras.Input(shape=last_conv_layer.output.shape[1:])
  x = classifier_input
  for layer_name in classifier_layer_names:
    x = model.get_layer(layer_name)(x)
  classifier_model = keras.Model(classifier_input, x)

  #Imprime o modelo da rede da ultima camada convolucional ate predicao
  print("\nclassifier_model:"); classifier_model.summary(); print()

  # Then, we compute the gradient of the top predicted class for our input image
  # with respect to the activations of the last conv layer
  with tf.GradientTape() as tape:
    # Compute activations of the last conv layer and make the tape watch it
    last_conv_layer_output = last_conv_layer_model(img_array)
    tape.watch(last_conv_layer_output)
    # Compute class predictions
    preds = classifier_model(last_conv_layer_output)
    print("preds:",type(preds),preds.shape,preds.dtype)
    top_pred_index = tf.argmax(preds[0])
    print("top_pred_index:",top_pred_index)
    top_class_channel = preds[:, top_pred_index]
    print("top_class_channel:",top_class_channel,
          type(top_class_channel),top_class_channel.shape,top_class_channel.dtype)

  # This is the gradient of the top predicted class with regard to
  # the output feature map of the last conv layer
  grads = tape.gradient(top_class_channel, last_conv_layer_output)
  print("grads:",type(grads),grads.shape,grads.dtype)

  # This is a vector where each entry is the mean intensity of the gradient
  # over a specific feature map channel
  pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))
  print("pooled_grads:",type(pooled_grads),pooled_grads.shape,pooled_grads.dtype)

  # We multiply each channel in the feature map array
  # by "how important this channel is" with regard to the top predicted class
  print("last_conv_layer_output:",type(last_conv_layer_output),
        last_conv_layer_output.shape,last_conv_layer_output.dtype)
  last_conv_layer_output = last_conv_layer_output.numpy()[0]
  print("last_conv_layer_output:",type(last_conv_layer_output),
        last_conv_layer_output.shape,last_conv_layer_output.dtype)
  pooled_grads = pooled_grads.numpy()
  print("pooled_grads:",type(pooled_grads),pooled_grads.shape,pooled_grads.dtype)
  for i in range(pooled_grads.shape[-1]):
    last_conv_layer_output[:, :, i] *= pooled_grads[i]
  print("last_conv_layer_output:",type(last_conv_layer_output),
```

```
90          last_conv_layer_output.shape,last_conv_layer_output.dtype)
91
92    # The channel-wise mean of the resulting feature map
93    # is our heatmap of class activation
94    heatmap = np.mean(last_conv_layer_output, axis=-1)
95
96    # For visualization purpose, we will also normalize the heatmap between 0 & 1
97    # Elimina parte negativa (relu)
98    heatmap = np.maximum(heatmap, 0) / np.max(heatmap)
99    return heatmap
100
101 # Prepare image
102 img_array = preprocess_input(get_img_array(img_path, size=img_size))
103
104 # Make model
105 model = model_builder(weights="imagenet"); #model.summary()
106
107 # Print what the top predicted class is
108 preds = model.predict(img_array)
109 print("Predicted:", decode_predictions(preds, top=1)[0])
110
111 # Generate class activation heatmap
112 heatmap = make_gradcam_heatmap(
113   img_array, model, last_conv_layer_name, classifier_layer_names
114 )
115
116 # Display heatmap
117 plt.matshow(heatmap); plt.show()
118
119 # We load the original image
120 img = keras.preprocessing.image.load_img(img_path)
121 img = keras.preprocessing.image.img_to_array(img)
122
123 # We rescale heatmap to a range 0-255
124 heatmap = np.uint8(255 * heatmap)
125
126 # We use jet colormap to colorize heatmap
127 jet = cm.get_cmap("jet")
128
129 # We use RGB values of the colormap
130 jet_colors = jet(np.arange(256))[:, :3]
131 jet_heatmap = jet_colors[heatmap]
132
133 # We create an image with RGB colorized heatmap
134 jet_heatmap = keras.preprocessing.image.array_to_img(jet_heatmap)
135 jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
136 jet_heatmap = keras.preprocessing.image.img_to_array(jet_heatmap)
137
138 # Superimpose the heatmap on original image
139 superimposed_img = jet_heatmap * 0.4 + img
140 superimposed_img = keras.preprocessing.image.array_to_img(superimposed_img)
141
142 # Save the superimposed image
143 save_path = "cam_"+img_path; superimposed_img.save(save_path)
144
145 # Display Grad CAM
146 plt.imshow(superimposed_img); plt.axis("off"); plt.show()
```

Programa 5: Grad-CAM usando VGG16.
https://colab.research.google.com/drive/1HzzMwY-wa72UdNdYHyzumXBYsP9oTUwX?usp=sharing

O programa 5 mostra a implementação de Grad-CAM. Executando esse programa com imagem "hummingbird.jpg" (figura 12), obtém a saída:

```
Predicted: [('n01833805', 'hummingbird', 0.9999659)]
```

que classifica corretamente a imagem como beija-flor, com probabilidade 99,99%.

A parte importante do programa é a função *make_gradcam_heatmap*, que recebe como parâmetros a imagem (*img_array*), o modelo da rede (*model*), o nome da última camada convolucional (*last_conv_layer_name*) e a lista das camadas entre a última camada convolucional até a predição (*classifier_layer_names*).

Essa função "enxerga" VGG16 como duas sub-redes (linhas 33-50 do programa, figuras 11 e 13):

1. *last_conv_layer_model*: Parte da VGG16 que vai da entrada até a última camada convolucional.
2. *classifier_model*: Parte da VGG16 que vai da última camada convolucional até a predição.

Depois, o programa calcula os gradientes $\partial y^c / \partial A^k$ (variável *grads*, linhas 54-70). Não é possível acessar estes gradientes em Keras. Devemos utilizar as funções de baixo nível de TensorFlow, em especial *GradientTape*. Quem tiver mais interesse, veja (por exemplo):

https://www.tensorflow.org/guide/autodiff?hl=en
https://www.tensorflow.org/api_docs/python/tf/GradientTape
https://stackoverflow.com/questions/53953099/what-is-the-purpose-of-the-tensorflow-gradient-tape
https://www.pyimagesearch.com/2020/03/23/using-tensorflow-and-gradienttape-to-train-a-keras-model/

Imprimindo as informações sobre variável *grads* ($\partial y^c / \partial A^k$, linha 70), obtemos:

```
grads: <class 'tensorflow.python.framework.ops.EagerTensor'> (1, 14, 14, 512) <dtype: 'float32'>
```

Como esperávamos, é um tensor de TensorFlow com forma 14x14x512, que pode ser convertido facilmente num tensor de NumPy.

Agora, vamos calcular α_k^c através de um cálculo de média como mostrado na equação (1). A linha 74 faz isso, armazenando α_k^c na variável pooled_grads. Imprimindo informações sobre variável *pooled_grads* (α_k^c), obtemos:

```
pooled_grads: <class 'tensorflow.python.framework.ops.EagerTensor'> (512,) <dtype: 'float32'>
```

É um vetor de 512 posições, como esperávamos.

As linhas 77-97 aplicam a equação (2), calculando produto escalar entre α_k^c e o mapa de atributos, aplicando ReLU e obtendo Grad-CAM. Figura 14 mostra Grad-CAM na resolução original 14x14. Figura 15 mostra Grad-CAM redimensionada e sobreposta à imagem original.

Figura 12: hummingbird.jpg [imagem em: http://www.lps.usp.br/hae/apostila/visualiza-ead.zip]

```
last_conv_layer_model:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_13 (InputLayer)        [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
(...)
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
_________________________________________________________________

classifier_model:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
input_14 (InputLayer)        [(None, 14, 14, 512)]     0
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
fc1 (Dense)                  (None, 4096)              102764544
_________________________________________________________________
fc2 (Dense)                  (None, 4096)              16781312
_________________________________________________________________
predictions (Dense)          (None, 1000)              4097000
=================================================================
Total params: 123,642,856
Trainable params: 123,642,856
Non-trainable params: 0
_________________________________________________________________
```

Figura 13: Grad-CAM enxerga VGG16 como duas redes: last_conv_layer_model e classifier_model.

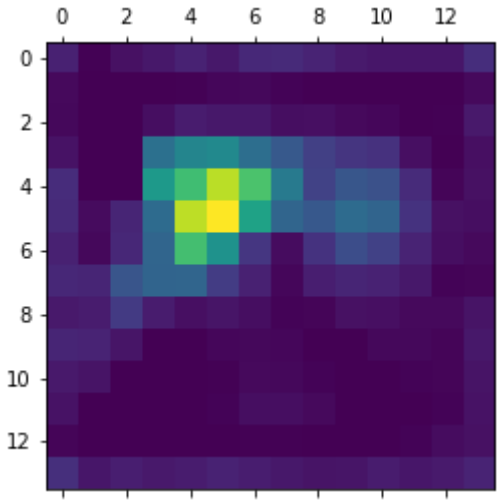

Figura 14: Grad-CAM com resolução 14x14.

Figura 15: Imagem original sobreposta à Grad-CAM.

Exercício: Gere Grad-CAM de 3 imagens (diferentes de *hummingbird.jpg*) usando programa 5.

Exercício: Gere os dois Grad-CAMs referentes às classes gato e cachorro da imagem *catdog2.png* [imagem em: http://www.lps.usp.br/hae/apostila/visualiza-ead.zip]

	
CAM da classe “boxer”	CAM da classe “tiger_cat”

Exercício: Altere o programa 5 para gerar Grad-CAM de resolução 28x28 (em vez de 14x14). Pegue o mapa de atributos na saída da camada `block4_conv3`. Continue usando imagem *hummingbird.jpg* e modelo VGG16.

Referências

[Selvaraju2017] Selvaraju, Ramprasaath R., et al. "Grad-cam: Visual explanations from deep networks via gradient-based localization." *Proceedings of the IEEE international conference on computer vision*. 2017. https://arxiv.org/abs/1610.02391