

Explicar a decisão tomada pela rede neural

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.

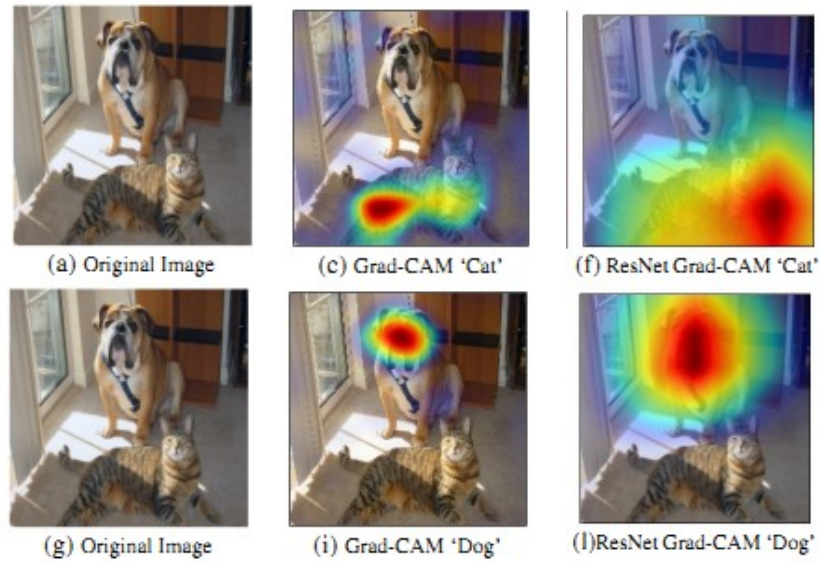


Figura 1: (c, i) Saída de Grad-CAM das categorias “gato” e “cachorro” usando VGG-16. (f-l) Usando ResNet-18. [\[Selvaraju2019\]](#)

2. Class Activation Mapping (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](#)].

A ideia intuitiva de CAM é a seguinte. Muitas redes convolucionais possuem a estrutura:

mapa de atributos A $\rightarrow$ global average pooling $\rightarrow$ camada densa $\rightarrow$ softmax.

Por exemplo, ResNet, Inception, EfficientNet, etc. Nessas redes, os atributos A da última camada convolucional (antes de average pooling) possuem as informações “mastigadas” de quais regiões de da imagem de entrada I foram importantes para que ela fosse classificada como classe c . Essas informações são perdidas quando se calcula average pooling. Para que essas informações fiquem explícitas, multiplica-se A pelos pesos responsáveis da camada densa pela classificação da imagem como c , obtendo uma imagem CAM com a mesma resolução espacial de mapa de atributos A , “pulando” average pooling.

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%.

```
1 #cam1-train.py - testada em Colab com TF2
2 #https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-class-activation-maps-
3 fe94eda4cef1
4 #https://github.com/Divyanshupy
5 #cada epoch demora 8s e chega a acuracidade de 0.9927
6
7 import tensorflow.keras as keras
8 from keras.datasets import mnist
9 import numpy as np
10 import matplotlib.pyplot as plt
11 from keras.models import Sequential, Model, load_model
12 from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, GlobalAveragePooling2D
13 from keras.utils import plot_model
14 from keras import optimizers
15 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
16 nomeprog="cam1"
17
18 def modelo():
19     model = Sequential()
20     model.add(Conv2D(16, input_shape=(28, 28, 1), kernel_size=(3, 3), activation='relu', padding='same'))
21     model.add(MaxPooling2D(pool_size=(2, 2)))
22
23     model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'))
24     model.add(MaxPooling2D(pool_size=(2, 2)))
25
26     model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'))
27     model.add(MaxPooling2D())
28
29     model.add(Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'))
30     model.add(GlobalAveragePooling2D())
31     model.add(Dense(10, activation='softmax'))
32
33     model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer=optimizers.Adam())
34     return model
35
36 (AX, AY), (QX, QY) = mnist.load_data()
37 AX = AX.reshape((AX.shape[0], AX.shape[1], AX.shape[2], 1))
38 QX = QX.reshape((QX.shape[0], QX.shape[1], QX.shape[2], 1))
39 AX = 1 - AX.astype('float')/255; QX = 1 - QX.astype('float')/255
40 nclasses = 10
41 AY = keras.utils.to_categorical(AY, nclasses)
42 QY = keras.utils.to_categorical(QY, nclasses)
43
44 #Escolha entre treinar do zero ou carregar modelo ja treinado
45 model=modelo()
46 #model=load_model(nomeprog+".h5")
47
48 model.summary()
49 plot_model(model, to_file=nomeprog+'.png', show_shapes=True)
50
51 model.fit(AX, AY, batch_size=100, epochs=30, validation_split=0.1, shuffle=True)
52 model.evaluate(QX, QY)
53 model.save(nomeprog+'.h5')
```

Programa 1: Treina rede que classifica MNIST, com GlobalAveragePooling.

<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.

```
1 #cam1-test.py - testado em Colab com TF2
2 import tensorflow.keras as keras
3 from tensorflow.keras.datasets import mnist
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from tensorflow.keras.models import load_model, Model
7 from tensorflow.keras.utils import plot_model
8 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
9 import scipy as sp
10 import cv2
11
12 nomeprog="cam1"
13
14 (AX,AY),(QX,QY) = mnist.load_data()
15 AX = AX.reshape((AX.shape[0],AX.shape[1],AX.shape[2],1))
16 QX = QX.reshape((QX.shape[0],QX.shape[1],QX.shape[2],1))
17 AX = 1 - AX.astype('float')/255; QX = 1 - QX.astype('float')/255
18 nclasses =10
19 AY = keras.utils.to_categorical(AY, nclasses)
20 QY = keras.utils.to_categorical(QY, nclasses)
21
22 model=load_model(nomeprog+'.h5')
23 gap_weights = model.layers[-1].get_weights()[0]
24 print("gap_weights.shape",gap_weights.shape)
25
26 cam_model = Model( inputs=model.input, outputs=(model.layers[-3].output, model.layers[-1].output) )
27 print("Nome do layer de features:",model.layers[-3].name)
28 print("Nome do layer de results:", model.layers[-1].name)
29
30 features,results = cam_model.predict(QX)
31 print("features.shape=",features.shape,"results.shape=",results.shape)
32 print()
33
34 for idx in range(10):
35     pred = np.argmax(results[idx])
36     print('Predicted Class = ' +str( pred )+ ', Probability = ' + str(results[idx][pred]))
37     cam_features = features[idx,:,:,:]; print("cam_features.shape",cam_features.shape)
38     cam_weights = gap_weights[:,pred]; print("cam_weights.shape",cam_weights.shape)
39     cam_output = np.dot(cam_features,cam_weights)
40     cam_output = cv2.resize(cam_output,(28,28),interpolation=cv2.INTER_NEAREST)
41     cam_output = cv2.normalize(cam_output, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX)
42
43     plt.figure(facecolor='white')
44     plt.imshow(np.squeeze(QX[idx,-1]), cmap='gray', alpha=0.4)
45     plt.colorbar(shrink=0.5)
46     plt.imshow(cam_output, cmap='inferno', alpha=0.6)
47     plt.colorbar(shrink=0.5)
48     plt.axis('off')
49     plt.show()
```

Programa 2: Gera CAM (Class Activation Map) usando a rede treinada no programa 1.

<https://colab.research.google.com/drive/1tRKchdpRB11igFupczeAc55u7YoE8dlr?usp=sharing>

A estrutura da rede *cam1.h5* está ilustrada na figura 3, as últimas camadas estão na figura 4. O programa 2 gera a imagem CAM.

Linha 23: O tensor *gap_weights* são os pesos da última camada densa e tem formato 128×10 .

Linha 30: Ao classificar um lote de imagens, a rede produz *features* no formato $? \times 3 \times 3 \times 128$ e *results* no formato $? \times 10$.

Linha 35: Desse lote, a classificação da imagem *QX[idx]* é *pred* (um número entre 0 e 9):
`pred = np.argmax(results[idx])`

Linha 37: A imagem *QX[idx]* possui atributos *cam_features* (formato $3 \times 3 \times 128$):
`cam_features = features[idx, :, :, :]`

Linha 38: Os pesos da última camada densa correspondentes à classe *pred* são *cam_weights*, um vetor de dimensão 128:
`cam_weights = gap_weights[:, pred]`

Linha 39: Os atributos *cam_features* ($3 \times 3 \times 128$) são multiplicados pelos pesos *gap_weights* (128×10) e a classe *pred* com a maior ativação é escolhida. Sabendo qual é a classe *pred* da imagem *QX[idx]*, é possível calcular a contribuição de cada região 3×3 para que a imagem fosse classificada como *pred*. Para isso, calcula-se a imagem *cam_output* (3×3) como produto de *cam_features* ($3 \times 3 \times 128$) com *cam_weights* (128,).

Linhas 40-41: A imagem *cam_output* é redimensionada para 28×28 e normalizada para o intervalo $[0, 1]$.

Linhas 43-49: A imagem do dígito de entrada *QX[idx]* é mostrada sobreposta à imagem *cam_output* colorida.

Exemplo: Vamos supor que a saída y_7 foi a maior, sendo a imagem classificada como classe $c=7$ (*pred* no programa). Neste caso, para gerar imagem CAM, devemos pegar os 128 pesos w_7 (*cam_weights* correspondentes à saída “7”) e calcular o produto escalar com os atributos $128 \times 3 \times 3$ (*cam_features*) para resultar na imagem CAM 3×3 (*cam_output*).

Resumindo, CAM mostra a contribuição de cada região 3×3 para que a imagem *QX[idx]* fosse classificada na classe *pred*.

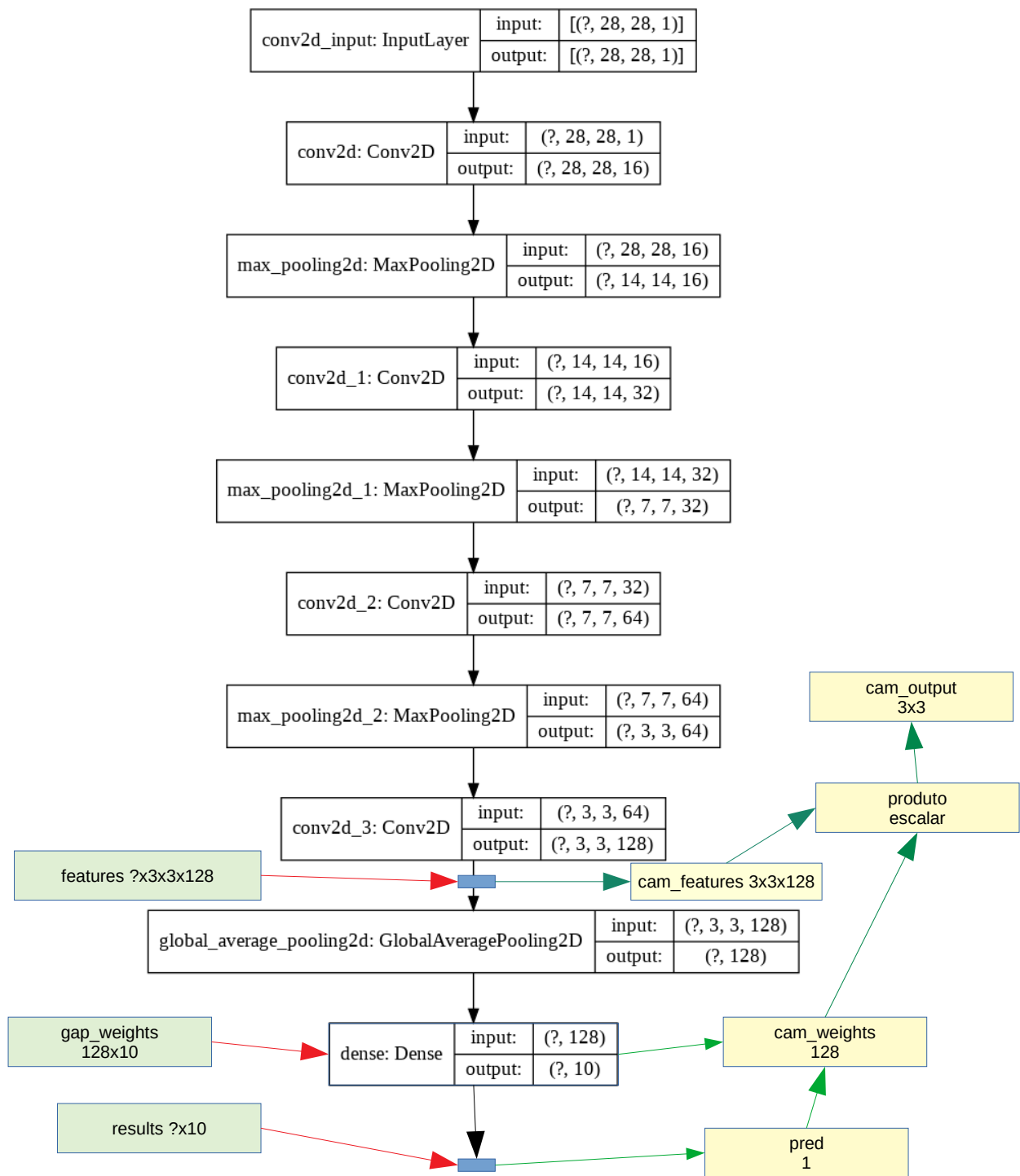


Figura 3: Estrutura da rede usada pelos programas 1 e 2.

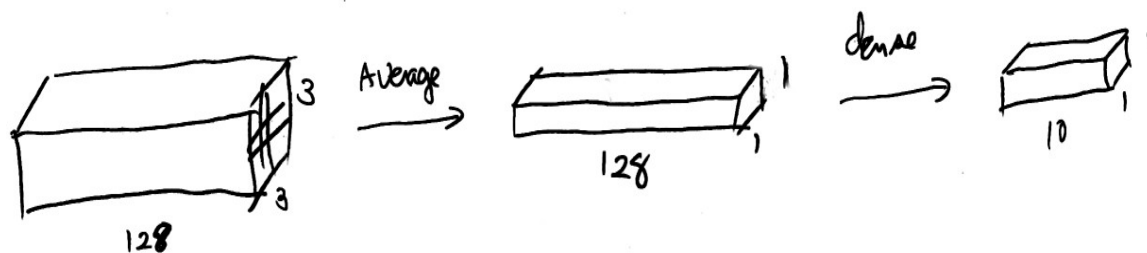


Figura 4: As últimas camadas da rede.

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 a impressão é falsa (figura F).

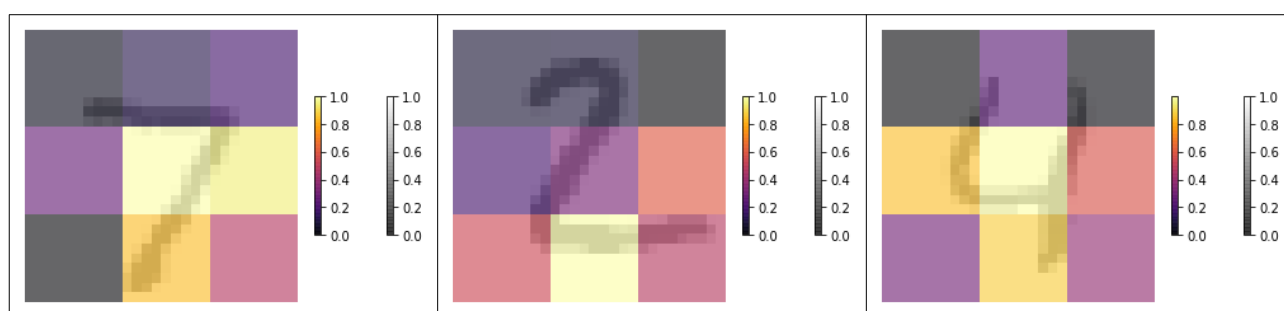


Figura 5: CAM geradas pelos programas 1 e 2 (cam1*.py) sobrepostas às imagens originais.

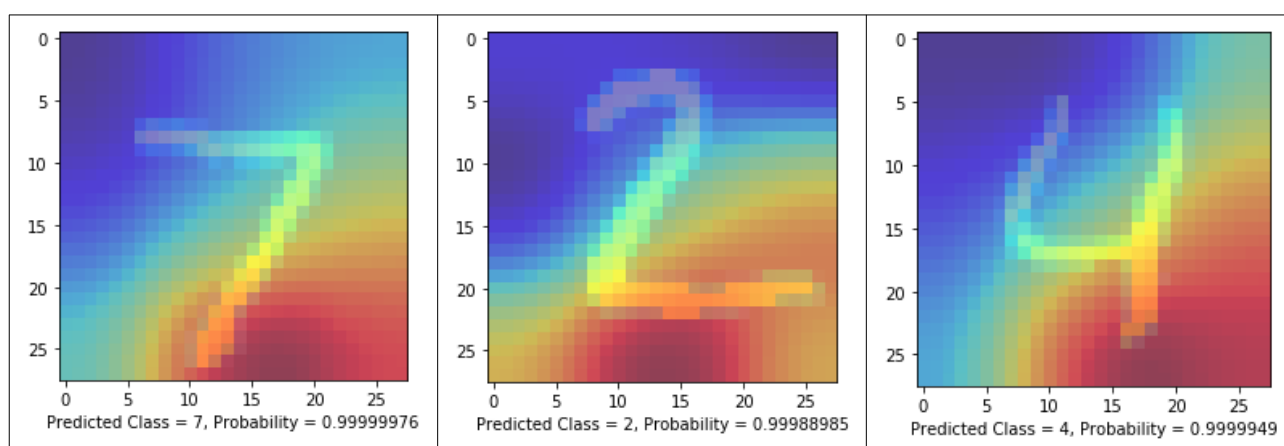


Figura F: As imagens do programa original com interpolação linear

[<https://github.com/Divyanshup/CLASS-ACTIVATION-MAPS/blob/master/Class%20Activation%20Maps%20with%20MaxPooling.ipynb>]

Como podemos aumentar a resolução de imagem 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 previsão 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 diago-

nal. 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”.

```

1 #cam2-train.py
2 #https://towardsdatascience.com/demystifying-convolutional-neural-networks-using-class-activation-maps-
3 fe94eda4cef1
4 #https://github.com/Divyanshupy
5 #Cada epoca demora 35s e chega a acuracidade de 0.9812
6
7 import tensorflow.keras as keras
8 from keras.datasets import mnist
9 import numpy as np
10 import matplotlib.pyplot as plt
11 from keras.models import Sequential, Model, load_model
12 from keras.layers import Dense, Conv2D, Flatten, MaxPooling2D, GlobalAveragePooling2D
13 from keras.utils import plot_model
14 from keras import optimizers
15 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
16 nomeprog="cam2"
17
18 def modelo():
19     model = Sequential()
20     model.add(Conv2D(16, input_shape=(28, 28, 1), kernel_size=(3, 3), activation='relu', padding='same'))
21     model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'))
22     model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same'))
23     model.add(Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same'))
24     model.add(GlobalAveragePooling2D())
25     model.add(Dense(10, activation='softmax'))
26
27     model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='adam')
28     return model
29
30 (AX, AY), (QX, QY) = mnist.load_data()
31 AX = AX.reshape((AX.shape[0], AX.shape[1], AX.shape[2], 1))
32 QX = QX.reshape((QX.shape[0], QX.shape[1], QX.shape[2], 1))
33 AX = 1 - AX.astype('float')/255; QX = 1 - QX.astype('float')/255
34 nclasses = 10
35 AY = keras.utils.to_categorical(AY, nclasses)
36 QY = keras.utils.to_categorical(QY, nclasses)
37
38 #Escolha entre treinar do zero ou carregar modelo ja treinado
39 model=modelo()
40 #model=load_model(nomeprog+".h5")
41
42 model.summary()
43 plot_model(model, to_file=nomeprog+'.png', show_shapes=True)
44
45 model.fit(AX, AY, batch_size=100, epochs=30, validation_split=0.1, shuffle=True)
46 model.evaluate(QX, QY)
47 model.save(nomeprog+'.h5')

```

Programa 3: Treina rede cam2.h5.

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

```

1 #cam2-test.py aplica CAM na rede ja treinada
2 import tensorflow.keras as keras
3 from keras.datasets import mnist
4 import numpy as np
5 import matplotlib.pyplot as plt
6 from keras.models import load_model
7 from keras.utils import plot_model
8 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
9 import scipy as sp
10 import cv2
11
12 nomeprog="cam2"
13
14 (AX,AY),(QX,QY) = mnist.load_data()
15 AX = AX.reshape((AX.shape[0],AX.shape[1],AX.shape[2],1))
16 QX = QX.reshape((QX.shape[0],QX.shape[1],QX.shape[2],1))
17 AX = 1 - AX.astype('float')/255; QX = 1 - QX.astype('float')/255
18 nclasses = 10
19 AY = keras.utils.to_categorical(AY, nclasses)
20 QY = keras.utils.to_categorical(QY, nclasses)
21
22 model=load_model(nomeprog+'.h5')
23 gap_weights = model.layers[-1].get_weights()[0]
24 print("gap_weights.shape",gap_weights.shape)
25
26 cam_model = Model( inputs=model.input, outputs=(model.layers[-3].output, model.layers[-1].output) )
27 print("Nome do layer de features:",model.layers[-3].name)
28 print("Nome do layer de results:", model.layers[-1].name)
29
30 features,results = cam_model.predict(QX)
31 print("features.shape",features.shape)
32 print("results.shape",results.shape)
33 print()
34
35 for idx in range(10):
36     features_for_one_img = features[idx,:,:,:]
37     print("features_for_one_img.shape",features_for_one_img.shape)
38     height_roomout = AX.shape[1]/features_for_one_img.shape[0]
39     width_roomout = AX.shape[2]/features_for_one_img.shape[1]
40     #print("height_roomout", height_roomout, "width_roomout", width_roomout)
41
42     cam_features = sp.ndimage.zoom(features_for_one_img, (height_roomout, width_roomout, 1), order=1)
43     print("cam_features.shape",cam_features.shape)
44     pred = np.argmax(results[idx])
45
46     cam_weights = gap_weights[:,pred]
47     print("cam_weights.shape",cam_weights.shape)
48     cam_output = np.dot(cam_features,cam_weights)
49     print('Predicted Class = ' +str( pred )+ ', Probability = ' + str(results[idx][pred]))
50
51     plt.figure(facecolor='white')
52     plt.imshow(np.squeeze(QX[idx,-1]), cmap='gray', alpha=0.4)
53     plt.colorbar(shrink=0.5)
54     cam_output = cv2.normalize(cam_output, None, alpha=0, beta=1, norm_type=cv2.NORM_MINMAX)
55     plt.imshow(cam_output, cmap='inferno', alpha=0.6)
56     plt.colorbar(shrink=0.5)
57     plt.axis('off')
58     plt.show()

```

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

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

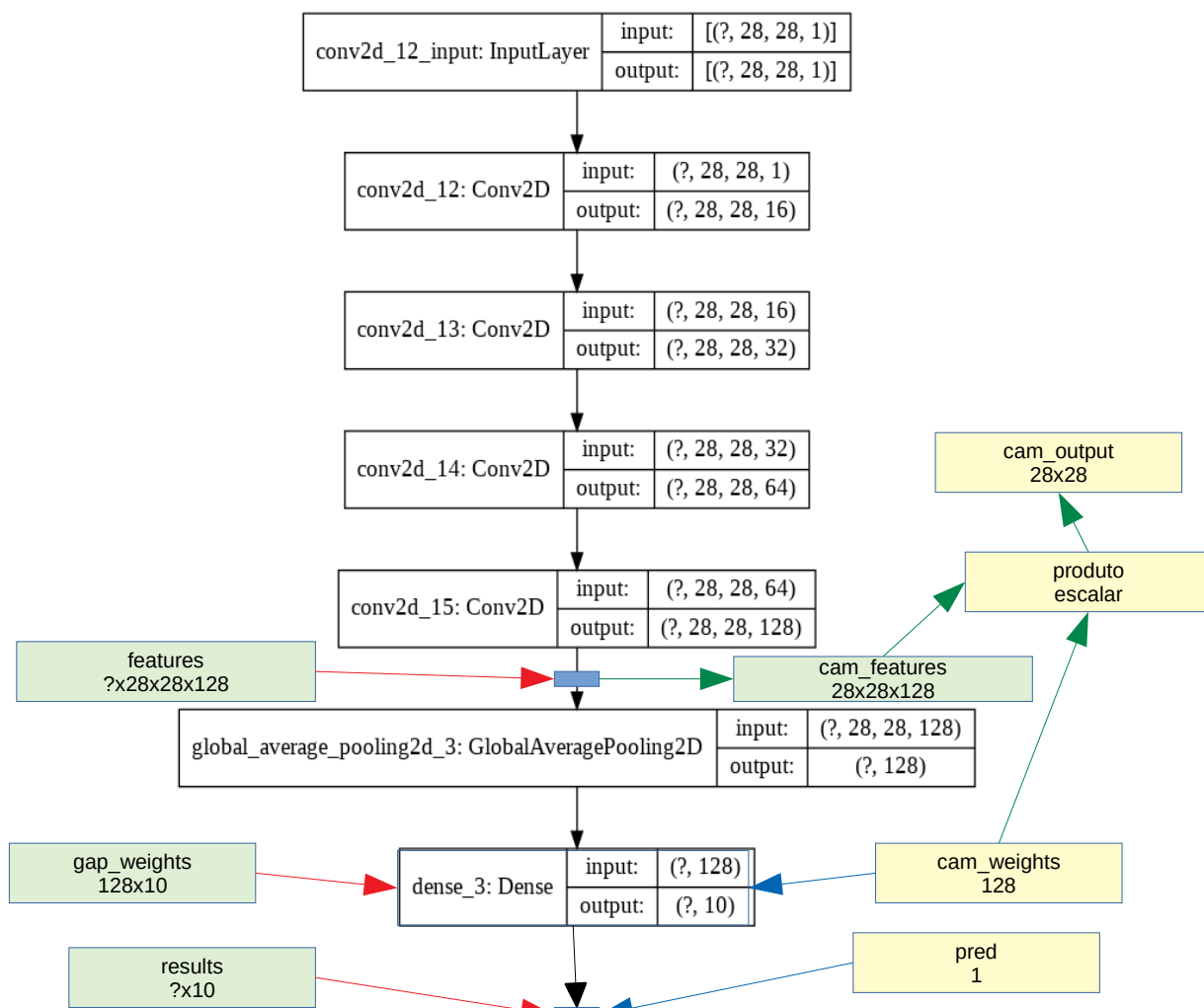


Figura 6: Estrutura da rede cam2.h5

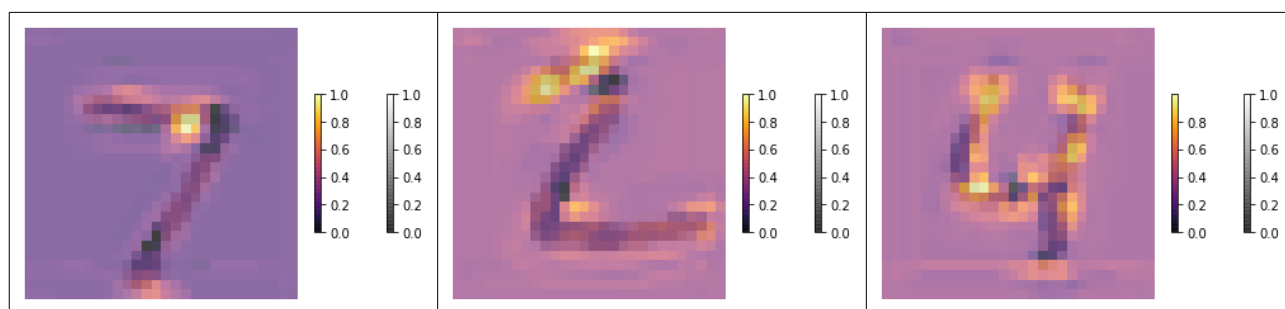


Figura 7: CAM geradas pelos programas 3 e 4 (`cam2*.py`) sobrepostas às imagens originais.

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 entre mapa de atributos da última camada convolucional e a camada softmax haja uma única camada densa, 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 retêm 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 convolucional 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 com resolução espacial $u \times v$. A intuição de Grad-CAM é que os 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 $L_{Grad-CAM}^c \in \mathbb{R}^{u \times v}$ 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 $u \times v \times n = 14 \times 14 \times 512$ 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_k^c da importância do mapa de atributos k para a classe c :

$$\alpha_k^c = \frac{1}{Z} \sum_i \sum_j \frac{\partial y^c}{\partial A_{ij}^k} \quad (1)$$

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

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

$$L_{Grad-CAM}^c = ReLU\left(\sum_k \alpha_k^c A^k\right) \quad (2)$$

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” (pois há 2 camadas densas no topo). O modelo VGG16 está nas figuras 10 e 11.

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 (BatchNormalizatio	(None, 10, 10, 1536)	6144	block14_sepconv1[0][0]
block14_sepconv1_act (Activation)	(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 (BatchNormalizatio	(None, 10, 10, 2048)	8192	block14_sepconv2[0][0]
block14_sepconv2_act (Activation)	(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.

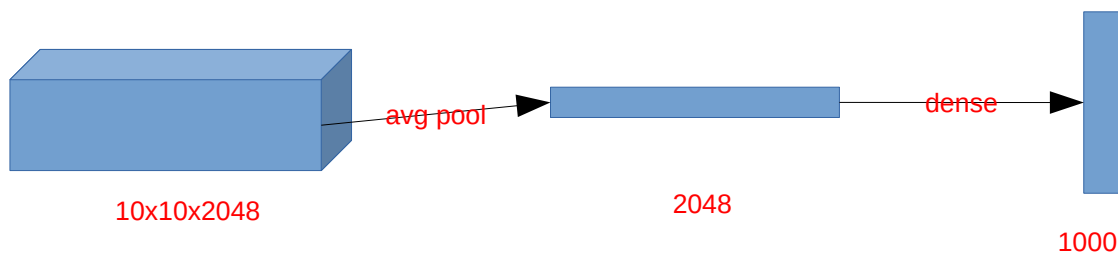


Figura 9: As últimas camadas de Xception.

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.

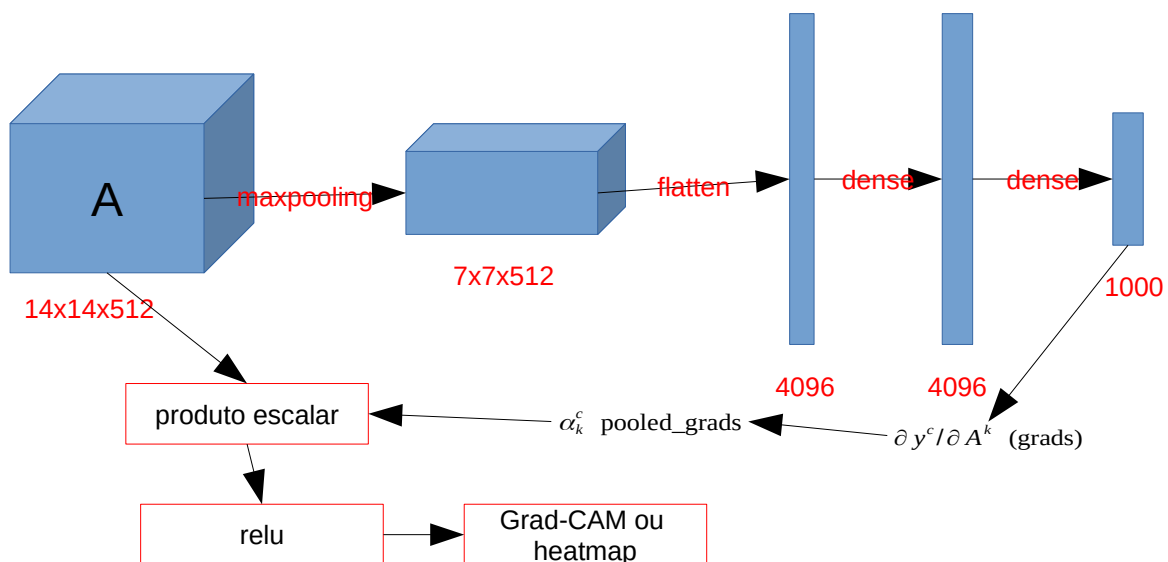


Figura 11: As últimas camadas de VGG16 e o cálculo de Grad-CAM.

```

1 #gradcam_vgg.py - testada em Colab com TF2
2 #Adaptado de https://keras.io/examples/vision/grad_cam/
3 import numpy as np
4 import tensorflow as tf
5 from tensorflow import keras
6 # from IPython.display import Image
7 import matplotlib.pyplot as plt
8 import matplotlib.cm as cm
9
10 model_builder = keras.applications.vgg16.VGG16
11 img_size = (224, 224)
12 preprocess_input = keras.applications.vgg16.preprocess_input
13 decode_predictions = keras.applications.vgg16.decode_predictions
14
15 last_conv_layer_name = "block5_conv3"
16 classifier_layer_names = ["block5_pool", "flatten", "fc1", "fc2", "predictions"]
17
18 img_path="hummingbird.jpg"
19 img=keras.preprocessing.image.load_img(img_path)
20 plt.imshow(img); plt.axis("off"); plt.show()
21
22 def get_img_array(img_path, size):
23     # `img` is a PIL image of size 299x299
24     img = keras.preprocessing.image.load_img(img_path, target_size=size)
25     # `array` is a float32 Numpy array of shape (299, 299, 3)
26     array = keras.preprocessing.image.img_to_array(img)
27     # We add a dimension to transform our array into a "batch"
28     # of size (1, 299, 299, 3)
29     array = np.expand_dims(array, axis=0)
30     return array
31
32 def make_gradcam_heatmap(img_array, model,
33                          last_conv_layer_name, classifier_layer_names):
34     # First, we create a model that maps the input image to the activations
35     # of the last conv layer
36     last_conv_layer = model.get_layer(last_conv_layer_name)
37     last_conv_layer_model = keras.Model(model.inputs, last_conv_layer.output)
38
39     #Imprime o modelo da rede ate a ultima camada convolucional
40     print("\nlast_conv_layer_model:"); last_conv_layer_model.summary(); print()
41
42     # Second, we create a model that maps the activations of the last conv
43     # layer to the final class predictions
44     classifier_input = keras.Input(shape=last_conv_layer.output.shape[1:])
45     x = classifier_input
46     for layer_name in classifier_layer_names:
47         x = model.get_layer(layer_name)(x)
48     classifier_model = keras.Model(classifier_input, x)
49
50     #Imprime o modelo da rede da ultima camada convolucional ate predicacao
51     print("\nclassifier_model:"); classifier_model.summary(); print()
52
53     # Then, we compute the gradient of the top predicted class for our input image
54     # with respect to the activations of the last conv layer
55     with tf.GradientTape() as tape:
56         # Compute activations of the last conv layer and make the tape watch it
57         last_conv_layer_output = last_conv_layer_model(img_array)
58         tape.watch(last_conv_layer_output)
59         # Compute class predictions
60         preds = classifier_model(last_conv_layer_output)
61         print("preds:", type(preds), preds.shape, preds.dtype)
62         top_pred_index = tf.argmax(preds[0])
63         print("top_pred_index:", top_pred_index)
64         top_class_channel = preds[:, top_pred_index]
65         print("top_class_channel:", top_class_channel,
66               type(top_class_channel), top_class_channel.shape, top_class_channel.dtype)
67
68     # This is the gradient of the top predicted class with regard to
69     # the output feature map of the last conv layer
70     grads = tape.gradient(top_class_channel, last_conv_layer_output)
71     print("grads:", type(grads), grads.shape, grads.dtype)
72
73     # This is a vector where each entry is the mean intensity of the gradient
74     # over a specific feature map channel
75     pooled_grads = tf.reduce_mean(grads, axis=(0, 1, 2))
76     print("pooled_grads:", type(pooled_grads), pooled_grads.shape, pooled_grads.dtype)
77
78     # We multiply each channel in the feature map array
79     # by "how important this channel is" with regard to the top predicted class
80     print("last_conv_layer_output:", type(last_conv_layer_output),
81           last_conv_layer_output.shape, last_conv_layer_output.dtype)
82     last_conv_layer_output = last_conv_layer_output.numpy()[0]
83     print("last_conv_layer_output:", type(last_conv_layer_output),
84           last_conv_layer_output.shape, last_conv_layer_output.dtype)
85     pooled_grads = pooled_grads.numpy()
86     print("pooled_grads:", type(pooled_grads), pooled_grads.shape, pooled_grads.dtype)
87     for i in range(pooled_grads.shape[-1]):
88         last_conv_layer_output[:, :, i] *= pooled_grads[i]
89     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 14×14×512, 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 75 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 78-98 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.

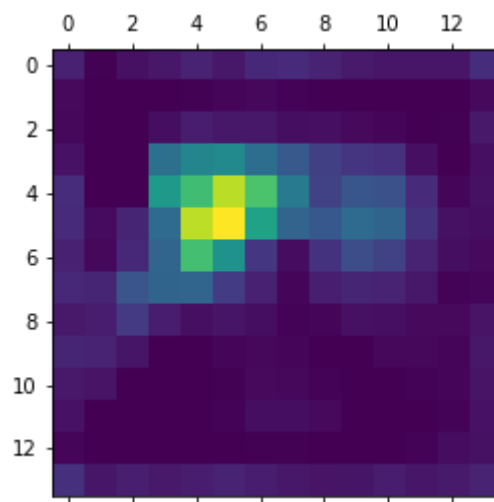


Figura 14: Grad-CAM com resolução 14x14.

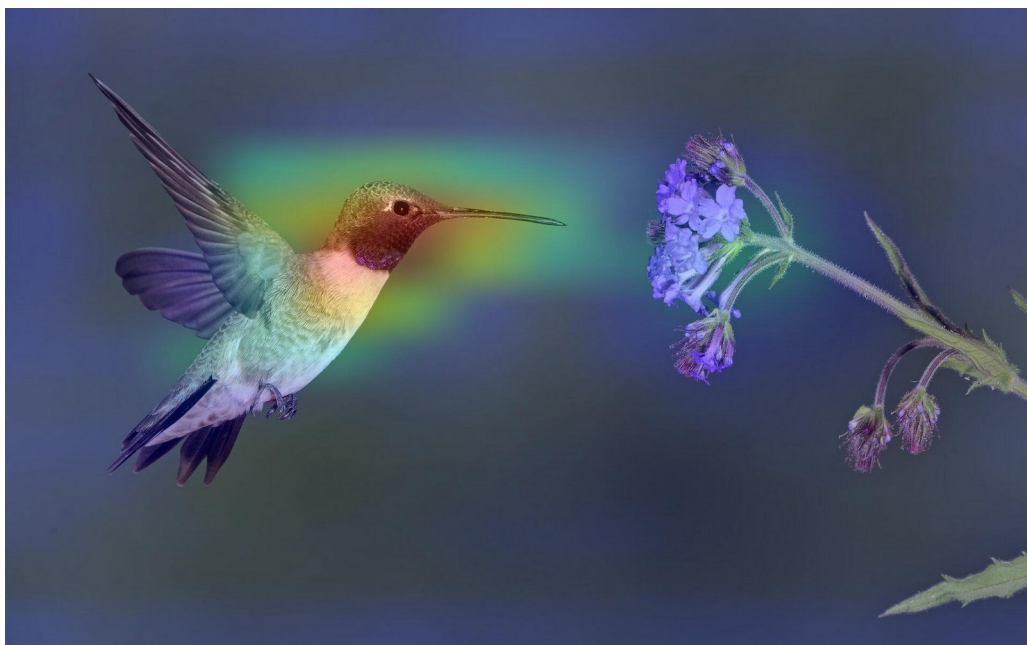


Figura 15: Imagem original sobreposta à Grad-CAM.

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>]



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 *humming-bird.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>

Apêndice A

Perguntando ao ChatGPT:

“What is the advantage of GRAD-CAM over CAM?”

Ela respondeu:

Grad-CAM (Gradient-weighted Class Activation Mapping) has several advantages over **CAM (Class Activation Mapping)**, making it more flexible and broadly applicable in neural network interpretability, especially for deep learning models.

1. Broader Applicability:

- **CAM** requires modifying the network architecture by removing fully connected layers and replacing them with global average pooling layers before the final classification layer. This restricts CAM's usage to specific types of architectures (typically CNNs with a single fully connected layer).
- **Grad-CAM**, on the other hand, can be applied to any convolutional neural network (CNN) architecture without requiring architectural changes. It uses the gradients of any target class flowing into the last convolutional layer to generate a coarse localization map.

2. Works with Pre-trained Networks:

- With **CAM**, you must modify and retrain the model with global average pooling layers. In contrast, **Grad-CAM** can be applied directly to pre-trained networks without needing to retrain or fine-tune the model, making it much easier to apply to existing models.

3. Class-Specific Explanations:

- **Grad-CAM** generates explanations that are specific to a particular class by computing gradients with respect to the class of interest. This allows you to visualize what part of the input image contributed most to the classification decision for that specific class.
- **CAM**, on the other hand, only highlights regions based on the network's final output, which might not always be directly tied to specific classes as flexibly as Grad-CAM.

4. Flexibility in Multi-task and Multi-label Settings:

- **Grad-CAM** can be used for more complex tasks like multi-task learning, where the network is trained to predict multiple outputs, or in multi-label classification scenarios, where more than one class is assigned to an image. The gradients help in isolating contributions of each task or label, making it more versatile.
- **CAM** is less flexible in these settings due to its reliance on architecture modifications.

5. More Granular Explanations:

- While both CAM and Grad-CAM produce coarse visualizations, **Grad-CAM** tends to provide more accurate and finer explanations as it leverages the gradient information, which can provide more detailed insights into which parts of the image influence the model's decision.

Summary of Grad-CAM Advantages:

- **No architectural modifications** required; works with any CNN.
- **Works with pre-trained networks** without retraining.
- Provides **class-specific explanations** by utilizing gradients.
- Flexible in **multi-label and multi-task learning** contexts.

- Offers **more granular and detailed explanations** due to gradient utilization.

Overall, **Grad-CAM** is a more powerful and general approach for model interpretability compared to **CAM**, making it suitable for a wider range of architectures and tasks.