

## [PSI5790 aula 6 parte 2. Início.]

[Nota para mim: Esta apostila deve ser revisada a partir de Inception\_v1. Deve decidir se mantém ou retira EfficientNet e Vision Transformer.]

# Modelos avançados de redes convolucionais

## 1. ImageNet e modelos pré-treinados para categorizar imagens.

Rede neural convolutional foi proposta em 1989 por LeCun [LeCun1989a, LeCun1989b]. Porém, o poder de rede neural convolucional só se tornou popular em 2012 com AlexNet classificando imagens de ImageNet.

ImageNet é um conjunto de imagens enorme projetado para ser usado em pesquisa de algoritmos de reconhecimento visual de objetos. [<https://www.image-net.org/index.php>]

Mais de 14 milhões imagens foram anotadas à mão pelo projeto para indicar quais objetos estão representados. ImageNet contém mais de 20.000 categorias.

Entre 2010 e 2017, o projeto ImageNet promoveu uma competição anual de software, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC, [<https://www.image-net.org/challenges/LSVRC/>]), onde programas competiam para classificar e detectar objetos em imagens. Esta competição usava um subconjunto de ImageNet com 1000 categorias não-sobrepostas [[Wiki-ImageNet](#)].

Esta competição usou aproximadamente 1,2 milhão de imagens de treino, 50 mil imagens de validação, e 150 mil imagens de teste, todas em alta resolução. O objetivo era classificar cada imagem em uma das 1000 categorias.

A lista das 1000 classes está, por exemplo, em:

<https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/> .

O conjunto de dados (na verdade, apenas .csv com links das imagens) está disponível em:

<https://www.kaggle.com/competitions/imagenet-object-localization-challenge>

A figura 1 mostra a evolução das taxas de erro “top-5” e o número de camadas do modelo ao longo dos anos. A figura 2 mostra as taxas de erro “top-1” de diferentes redes. A figura X mostra que, a partir de 2015 com ResNet, CNN passou a errar menos que um ser humano (a taxa de erro top-5 de um ser humano é aproximadamente 5%).

Erro “top-5” é a porcentagem de imagens onde o rótulo correto não é um dos 5 rótulos mais prováveis fornecidos pelo modelo.

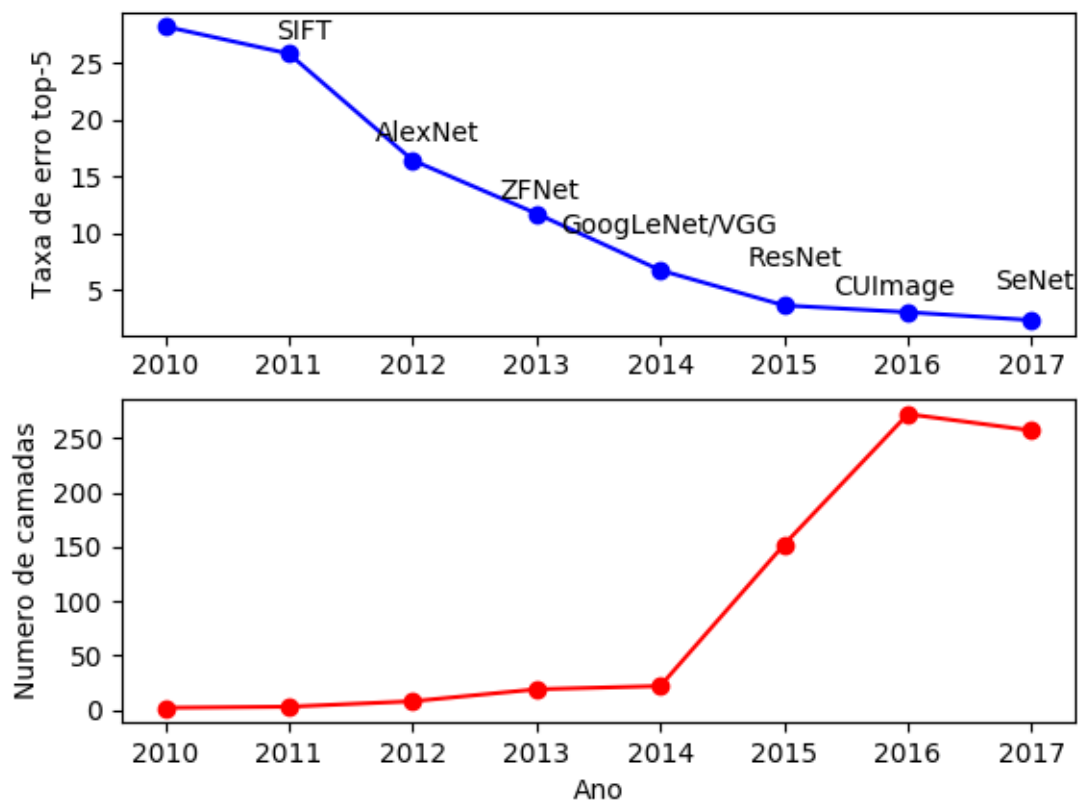


Figura 1: As taxas de erro “top-5” de ImageNet-ILSVRC foram caindo e número de camadas convolucionais da rede foi aumentando ao longo dos anos.

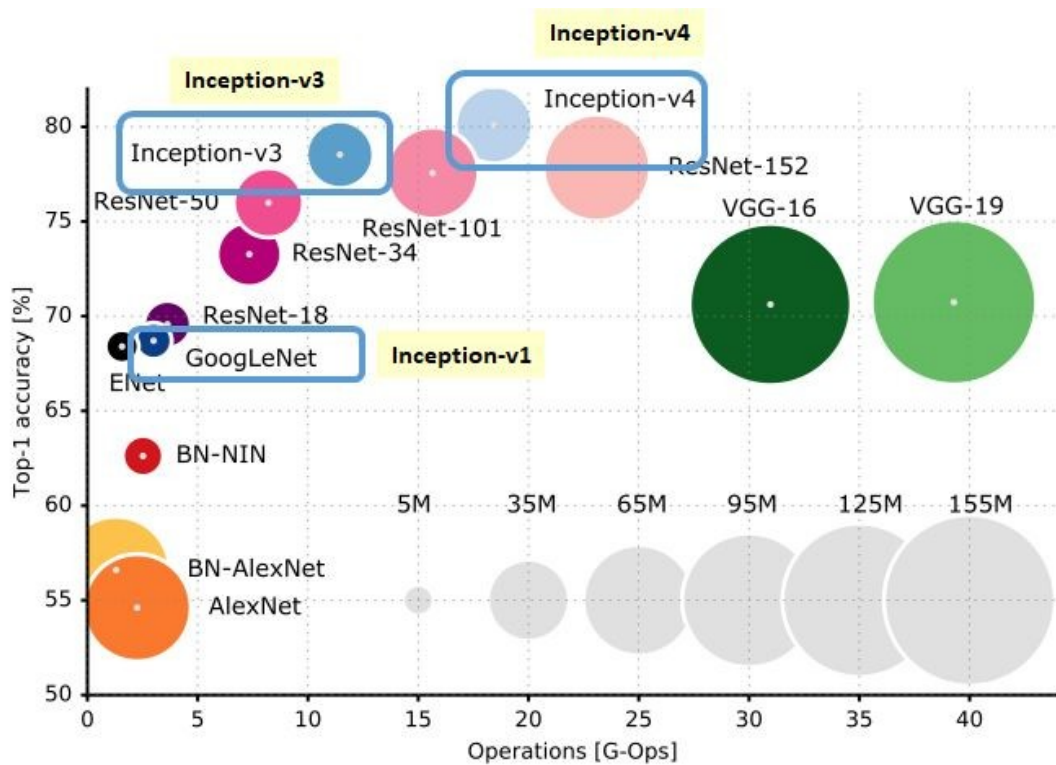


Figura 2: Taxas de acerto “top-1” da competição ImageNet.

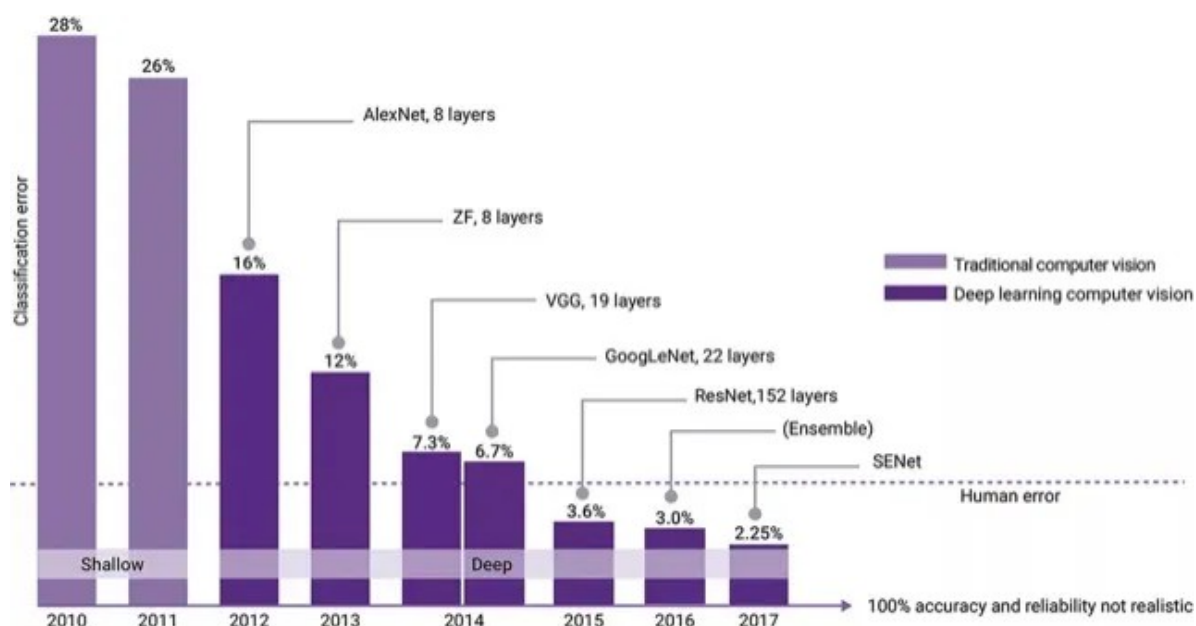


Figura X: Taxas de erro “top-5” da ImageNet.

Treinar uma rede convolucional usando ImageNet é uma tarefa que exige alto poder computacional e paciência. Porém, Keras fornece vários modelos pré-treinados para classificar ImageNet, prontos para serem usados. Copio na tabela abaixo alguns modelos que Keras oferece (veja <https://keras.io/api/applications/> para a lista completa).

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	98 MB	0.749	0.921	25,636,712	-
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-
EfficientNetB0	29 MB	-	-	5,330,571	-
EfficientNetB5	118 MB	83.6%	96.7%	30,562,527	312
EfficientNetB7	256 MB	84.3%	97.0%	66,658,687	-

Tabela 1: Alguns modelos pré-treinados que reconhece as classes de ImageNet.

Nota: Dependendo da versão de Keras no seu computador, alguns modelos do manual podem não estar disponíveis. Para listar as redes realmente disponíveis em seu Keras, execute em python:

```
>import tensorflow.keras.applications as app
>print(dir(app))
```

Resulta em Google Colab (2021):

```
['DenseNet121', 'DenseNet169', 'DenseNet201', 'EfficientNetB0', 'EfficientNetB1', 'EfficientNetB2', 'EfficientNetB3', 'EfficientNetB4', 'EfficientNetB5', 'EfficientNetB6', 'EfficientNetB7', 'InceptionResNetV2', 'InceptionV3', 'MobileNet', 'MobileNetV2', 'MobileNetV3Large', 'MobileNetV3Small', 'NASNetLarge', 'NASNetMobile', 'ResNet101', 'ResNet101V2', 'ResNet152', 'ResNet152V2', 'ResNet50', 'ResNet50V2', 'VGG16', 'VGG19', 'Xception', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__sys__', 'densenet', 'efficientnet', 'imagenet_utils', 'inception_resnet_v2', 'inception_v3', 'mobilenet', 'mobilenet_v2', 'mobilenet_v3', 'nasnet', 'resnet', 'resnet50', 'resnet_v2', 'vgg16', 'vgg19', 'xception']
```

Podemos usar essas redes pré-treinadas para classificar as imagens. O programa 1 abaixo faz essa tarefa. A forma de chamá-lo é:  
\$python3 classif3.py

```
url='http://www.lps.usp.br/hae/apostila/cifar_pre treinado.zip'
import os; nomeArq=os.path.split(url)[1]
if not os.path.exists(nomeArq):
    print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
    os.system("wget -U 'Firefox/50.0' "+url)
else:
    print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
print("Descompactando arquivos novos de",nomeArq)
os.system("unzip -u "+nomeArq)
```

Programa: Baixa as imagens exemplos no diretório local.

```
1 #classif3.py
2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3';
3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
4 import tensorflow as tf; import tensorflow.keras as keras
5 from tensorflow.keras.preprocessing import image; from matplotlib import pyplot as plt
6 import numpy as np; import sys; from sys import argv
7
8 # Use os comandos abaixo se for chamar do prompt
9 # if (len(argv)!=2):
10 #     print("classif1.py nomeimg.ext");
11 #     sys.exit("Erro: Numero de argumentos invalido.");
12
13 # from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input, decode_predictions
14 # model = ResNet50(weights='imagenet')
15 # target_size = (224, 224)
16
17 # from tensorflow.keras.applications.inception_v3 import InceptionV3, preprocess_input, decode_predictions
18 # model = InceptionV3(weights='imagenet')
19 # target_size = (299, 299)
20
21 # from tensorflow.keras.applications.inception_resnet_v2 import \
22 #     InceptionResNetV2, preprocess_input, decode_predictions
23 # model = InceptionResNetV2(weights='imagenet')
24 # target_size = (299, 299)
25
26 from tensorflow.keras.applications.efficientnet import EfficientNetB0, preprocess_input, decode_predictions
27 model = EfficientNetB0(weights='imagenet')
28 target_size = (224, 224)
29
30 #img_path = argv[1] #Escreva aqui o diretorio e nome da imagem
31 img_path = "orantango.jpg"
32 img = image.load_img(img_path, target_size=target_size)
33 plt.imshow(img); plt.axis("off"); plt.show()
34 x = image.img_to_array(img); x = np.expand_dims(x, axis=0)
35 x = preprocess_input(x)
36
37 preds = model.predict(x)
38 p=decode_predictions(preds, top=3)[0]
39 # decode the results into a list of tuples (class, description, probability)
40 # (one such list for each sample in the batch)
41 #print('Predicted:', p)
42 # Predicted: [(u'n02504013', u'Indian_elephant', 0.82658225), ... ]
43
44 for i in range(len(p)):
45     print("%8.2f%% %s"%(100*p[i][2],p[i][1]))
```

Programa 1: Classifica imagem de entrada em categorias de ImageNet.

<https://colab.research.google.com/drive/1eKZj9jNxH7DfyL-7CpWiEfrWy7Weels?usp=sharing>

Classifiquei algumas imagens que achei na internet usando rede InceptionResNetV2 e os resultados estão na figura 3.

O programa acertou todos os objetos das imagens exceto “eiffel.jpg” e “onibus.jpg”. A imagem “eiffel.jpg” foi classificada como “igreja” por que Torre Eiffel não faz parte das 1000 categorias da ImageNet.

O programa não tinha como adivinhar o que não foi ensinado.

A imagem “onibus.jpg” foi classificada como “passenger\_car” (que significa “vagão de trem para passageiros” ou “automóvel para passageiros”).

Talvez o modelo tenha errado por que nunca viu um ônibus pintado da forma que aparece na figura. De qualquer forma, o programa está perto de acertar, dando 2ª e 3ª classificações como ônibus elétrico e mini ônibus.

O impressionante é que o programa consegue distinguir os sub-tipos de coelho: “coelho”, “lebre” e “coelho Angorá”.

Não sou capaz de distinguir esses sub-tipos, mas o programa parece ser capaz distingui-los. Além disso, o programa não só diz que a imagem é de um navio, mas diz que é “navio cruzado”, “porta-aviões”, “barco-bombeiro”, “ferry-boat”, etc.

No programa 1, é possível usar as redes “ResNet50”, “InceptionV3”, “InceptionResNetV2” ou “EfficientNetB0” dependendo de (des)comentar os trechos corretos nas linhas 13-28. Repare que cada uma das 4 redes possui suas próprias funções *preprocess\_input* e *decode\_predictions*. Também repare que cada uma das 4 redes redimensiona a imagem de entrada para tamanhos diferentes (224×224 ou 299×299). O tamanho de entrada “default” de cada rede pode ser consultado no manual de Keras [<https://keras.io/api/applications/>], clicando no modelo e procurando “input\_shape”.

*Nota:* Algumas redes, como EfficientNet, podem não ter *input\_shape* default.

A linha 32 carrega a imagem e a redimensiona para 224×224, usando a função *load\_img* de Keras.

Na linha 34, a função *image\_to\_array* converte classe *image* de Keras *img* para matriz numpy x tipo float32.

A linha 35 acrescenta uma dimensão extra na matriz x, convertendo-a de [299, 299, 3] para [1, 299, 299, 3]. Isto é necessário, pois a função *model.predict* pode fazer predição de um conjunto de imagens e está deve receber um vetor de imagens coloridas.

Na linha 36, a função *preprocess\_input* converte os elementos de x de 0 a 255 para ponto flutuante, aparentemente para obter uma distribuição gaussiana de média 0 e desvio 1. Cada modelo possui função *preprocess\_input* diferente.

Na linha 38, a função *model.predict* classifica cada uma das imagens de x. Se x tiver uma única imagem, vai fazer uma única classificação. A saída é um vetor de 1000 números que representam “pseudo-probabilidade” de pertencer a cada uma das 1000 categorias, gerada pelo *softmax*.

Na linha 39, a função *decode\_predictions* converte as 3 categorias mais prováveis de números para palavras compreensíveis para um ser humano.












 <p>chimpanzee.jpg 89.79% chimpanzee 0.94% siamang 0.46% guenon</p>	 <p>coelho.jpg 60.00% wood_rabbit 30.98% hare 1.77% Angora</p>	 <p>eiffel.jpg 90.28% church 1.38% fountain 1.01% bell_cote</p>
 <p>navio.jpg 94.10% liner 0.17% dock 0.11% fireboat</p>	 <p>onibus.jpg 56.95% passenger_car 14.63% trolleybus 14.17% minibus</p>	 <p>orangotango.jpg 87.75% orangutan 0.87% siamang 0.42% patas</p>
 <p>porta_avioes.jpg 91.23% aircraft_carrier 0.22% hen-of-the-woods 0.12% warplane</p>	 <p>tiger.jpg 79.68% tiger 12.74% tiger_cat 0.12% lynx</p>	 <p>trem_bala.jpg 92.98% bullet_train 0.09% electric_locomotive 0.08% sliding_door</p>

Figura 3: Classificação de algumas imagens da internet usando programa 1 com InceptionRes-NetV2.

*Exercício:* Obtenha algumas imagens da internet e classifique-as usando o programa `classif3.py`.

*Exercício:* Modifique o programa `classif3.py` para usar algum modelo diferente de EfficientNetB0. Teste o seu programa em umas 3 imagens (que você escolher). Comente se a rede classificou corretamente as imagens.

*Nota:* Antes da ImageNet, o conjunto de imagens Pascal Visual Object Classes era muito usado.  
<http://host.robots.ox.ac.uk/pascal/VOC/>

## 2. Cifar-10

### 2.1 Introdução

Gostaríamos de testar redes convolucionais avançadas que consigam classificar imagens mais complexas do que dígitos manuscritos, pois já conseguimos acertar virtualmente 100% dos dígitos do MNIST utilizando rede convolucional “tipo LeNet”. Seria bom usar o banco de dados ImageNet que descrevemos acima. Porém, ImageNet é muito grande e o treino demora demais. Em seu lugar, usaremos um banco de dados menor chamado Cifar-10 para testar as redes avançadas.

O conjunto de dados CIFAR-10 consiste em 60.000 imagens coloridas  $32 \times 32$  divididas em 10 classes, com 6.000 imagens por classe. Existem 50.000 imagens de treinamento e 10.000 imagens de teste. O conjunto de dados é dividido em cinco lotes de treinamento e um lote de teste, cada lote com 10.000 imagens. O lote de teste contém exatamente 1.000 imagens de cada classe. Os 5 lotes de treinamento somados contêm exatamente 5.000 imagens de cada classe. Porém, o número de imagens de uma classe num determinado lote de treino pode ser diferente de 1000.

Esse conjunto de imagens encontra-se em:

<http://www.cs.toronto.edu/%7Ekriz/cifar.html>

A figura 4 mostra as 10 classes do banco de dados (uma classe por linha), juntamente com 10 imagens de cada classe.

No site citado acima, há arquivos adequados para serem usados em Python, Matlab e C/C++. Os arquivos para C/C++ são:

```
data_batch_1.bin
data_batch_2.bin
data_batch_3.bin
data_batch_4.bin
data_batch_5.bin
test_batch.bin
```

Os dados estão organizados em seqüências de blocos de  $32 \times 32 \times 3 + 1 = 3073$  bytes, cada bloco representando uma imagem:

```
32×32 bytes para pixels red.
32×32 bytes para pixels green.
32×32 bytes para pixels blue.
1 byte para o rótulo (número de 0 a 9).
```

Ceikeikon/C++ possui a função de leitura de Cifar10:

```
void leCifar10(const string& nomearq,
               vector< Mat_<COR> >& images, vector<BYTE>& rotulos);
```

Python/Keras possui a função de leitura:

```
from tensorflow.keras.datasets import cifar10
(ax, ay), (qx, qy) = cifar10.load_data()
```

Em PyTorch:

```
import torchvision; from torchvision import datasets
A = datasets.CIFAR10('diretorio', download=True, train=True, transform=None)
Q = datasets.CIFAR10('diretorio', download=True, train=False, transform=None)
ax=A.data; ay=np.array(A.targets)
qx=Q.data; qy=np.array(Q.targets)
```

*Exercício:* Escreva um programa que imprime os 3 primeiros cachorros (categoria 5) e os 3 primeiros gatos (categoria 3) dos dados de teste Cifar10 como imagens PNG: cao1.png, cao2.png, cao3.png, gato1.png, gato2.png, gato3.png.

*Exercício:* Escreva um programa que imprime as 10 primeiras imagens de teste de cada classe, como na figura 4.

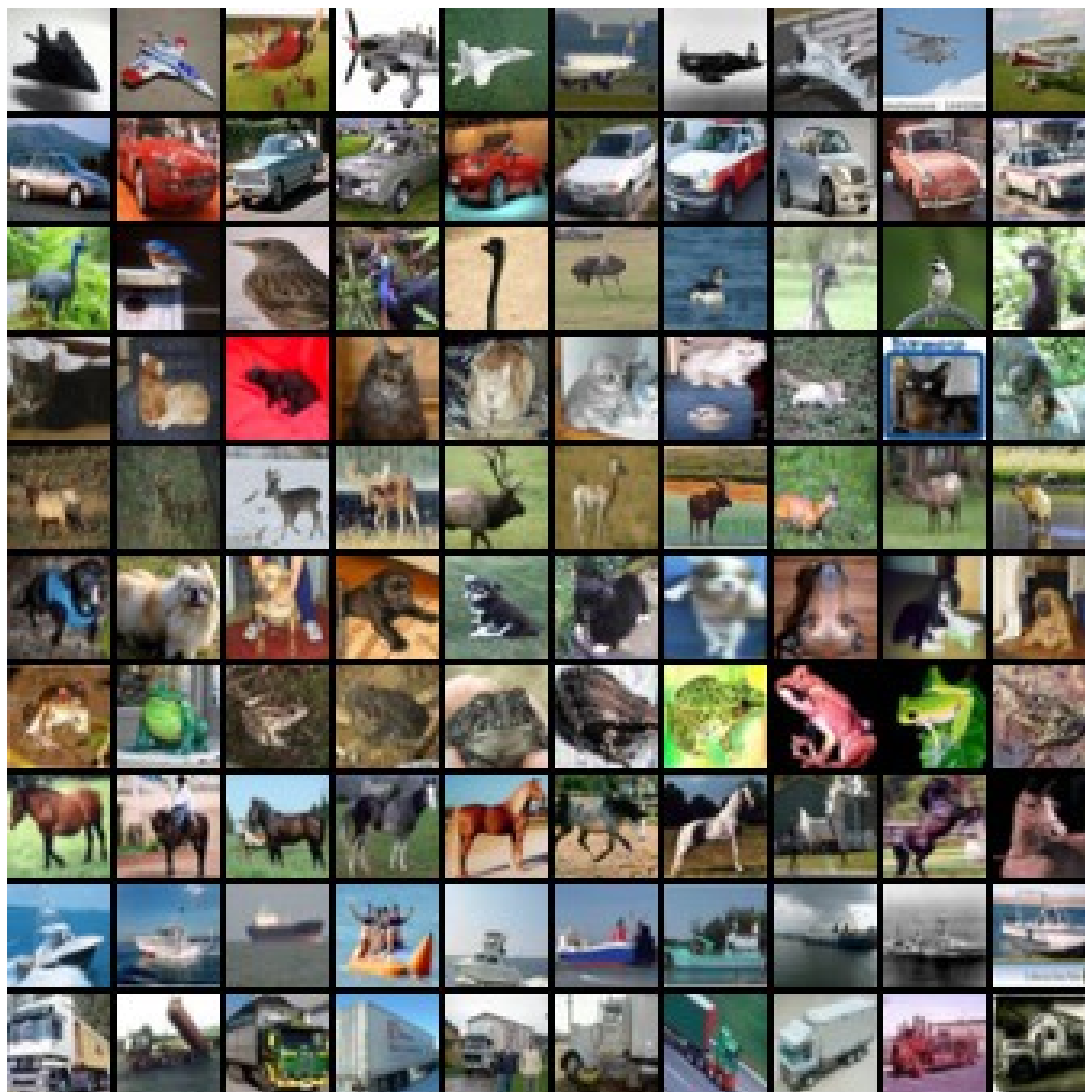


Figura 4: As 10 classes de Cifar-10 com algumas amostras. As classes são 0=airplane, 1=auto-mobile, 2=bird, 3=cat, 4=deer, 5=dog, 6=frog, 7=horse, 8=ship, 9=truck [0=avião, 1=automóvel, 2=pássaro, 3=gato, 4=veado, 5=cachorro, 6=sapo, 7=cavalo, 8=navio, 9=caminhão].



## 2.2 Taxas de acerto de Cifar-10

Taxa de acerto de um ser humano ao classificar imagens de teste de Cifar-10 é aproximadamente 94%, segundo:

<https://karpathy.github.io/2011/04/27/manually-classifying-cifar10/>

As taxas de erro de diferentes algoritmos na classificação Cifar-10 estão em:

[rodrigob] [https://rodrigob.github.io/are\\_we\\_there\\_yet/build/classification\\_datasets\\_results.html#43494641522d3130](https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html#43494641522d3130)

[benchmarks] <https://benchmarks.ai/cifar-10>

[paperswithcode] <https://paperswithcode.com/sota/image-classification-on-cifar-10>

A figura 5 abaixo mostra a evolução das taxas de acerto de Cifar-10 ao longo dos anos.

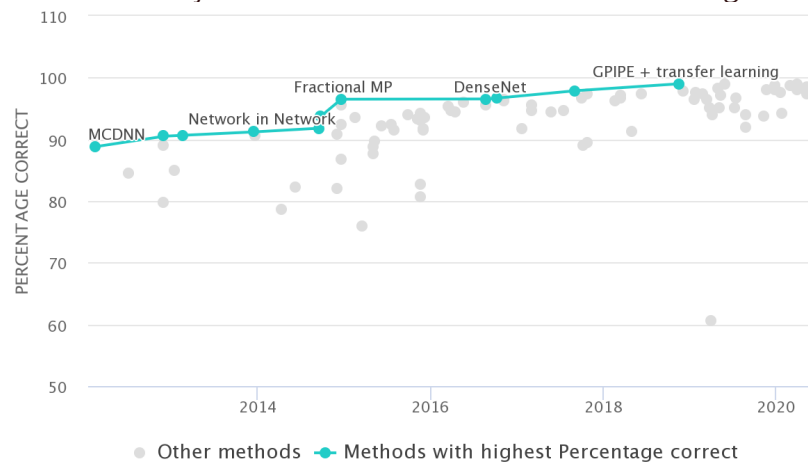


Figura 5: Taxas de acerto de Cifar-10 (retirado de [paperswithcode]).

Em 2013, a taxa de acertos era aproximadamente 90%, já usando rede convolucional. Em 6 anos, a taxa de acerto subiu para 99% (2019). A tabela 2 resume as taxas de acerto que obtive com diferentes algoritmos de aprendizagem. A maior taxa que consegui obter é 96%.

Nota: Taxa de acerto MNIST indica quanto esse mesmo algoritmo acerta quando classifica MNIST.

Nota: As redes LeNet, VGG, GoogleNet, ResNet e EfficientNet foram adaptados para poderem processar imagens pequenas 32×32.

	Método	Software usado	Taxa de acerto Cifar-10	Taxa de acerto MNIST
Grupo 1 Clássicos	Vizinho mais próximo	OpenCV/C++	15%	97,5%
	Árvore de decisão	OpenCV/C++	13%	77%
	Rede neural densa	Tiny-dnn/C++	53%	98,5%
Grupo 2 Redes convolucionais	“tipo LeNet” sem data augmentation	Keras/Python	73%	99,3%
	VGG sem data augmentation	Keras/Python	84%	?
	VGG com data augmentation	Keras/Python	92%	?
	GoogLeNet (Inception) simplificado	Keras/Python	92%	?
	ResNet simplificado	Keras/Python	86%	?
	ResNet	Keras/Python	92%	?
	ResNet com TTA	Keras/Python	93%	
Grupo 3 transfer learning	Transfer learning com EfficientNetB0 (transf-ead)	Keras/Python	96%	?

Tabela 2: Taxas de acerto de diversos algoritmos no problema Cifar-10 e MNIST.

Na tabela 2, os algoritmos estão divididos em dois grupos. No primeiro grupo estão os algoritmos clássicos (isto é, os algoritmos de aprendizado excluindo as redes convolucionais). Os métodos “vizinho mais próximo” e “árvore de decisão” possuem taxas de acerto 15% e 13% e portanto não são muito melhores que “chutes” (que teria acuracidade de 10%). Já a rede neural densa possui taxa de acerto de 53%.

Nota: As taxas de acerto desses algoritmos são muito mais altas ao classificar dígitos de MNIST. Com isso, podemos concluir que Cifar-10 é um problema mais difícil que MNIST.

No segundo grupo estão as redes convolucionais profundas. O modelo “tipo LeNet” que já usamos em MNIST apresenta acuracidade de 74% ao classificar Cifar-10. As redes VGG, GoogleNet e ResNet apresentam taxas de acerto próximas de 92%. Transfer learning com EfficientNetB0 obteve 96% de acerto.

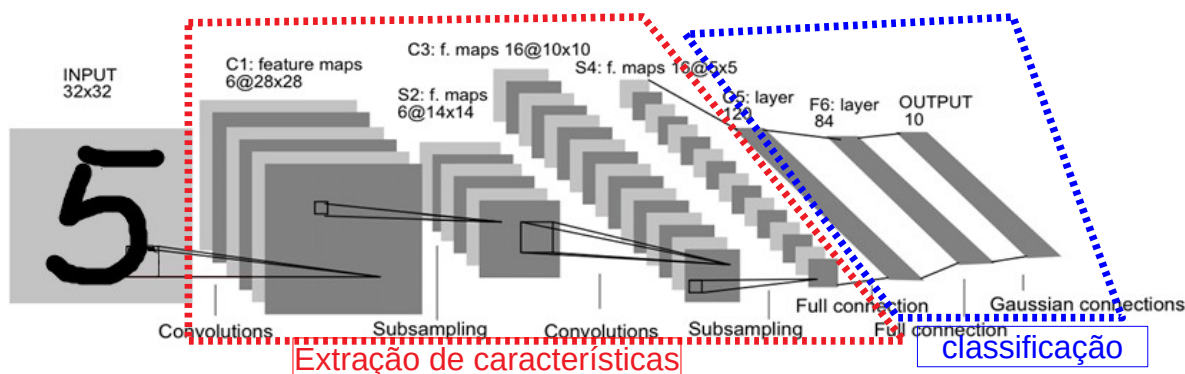


Figura 6: Modelo “tipo LeNet”.

*Exercício:* Escreva um programa que classifica Cifar10 usando rede neural densa em Keras.

### 3. Rede “tipo LeNet”

O programa 2 usa rede do “tipo LeNet” (figura 6) para classificar Cifar-10. Estou usando a expressão “rede tipo LeNet” para designar um modelo em que camadas convolucionais são seguidas pela maxpooling, e o modelo termina com camadas densas.

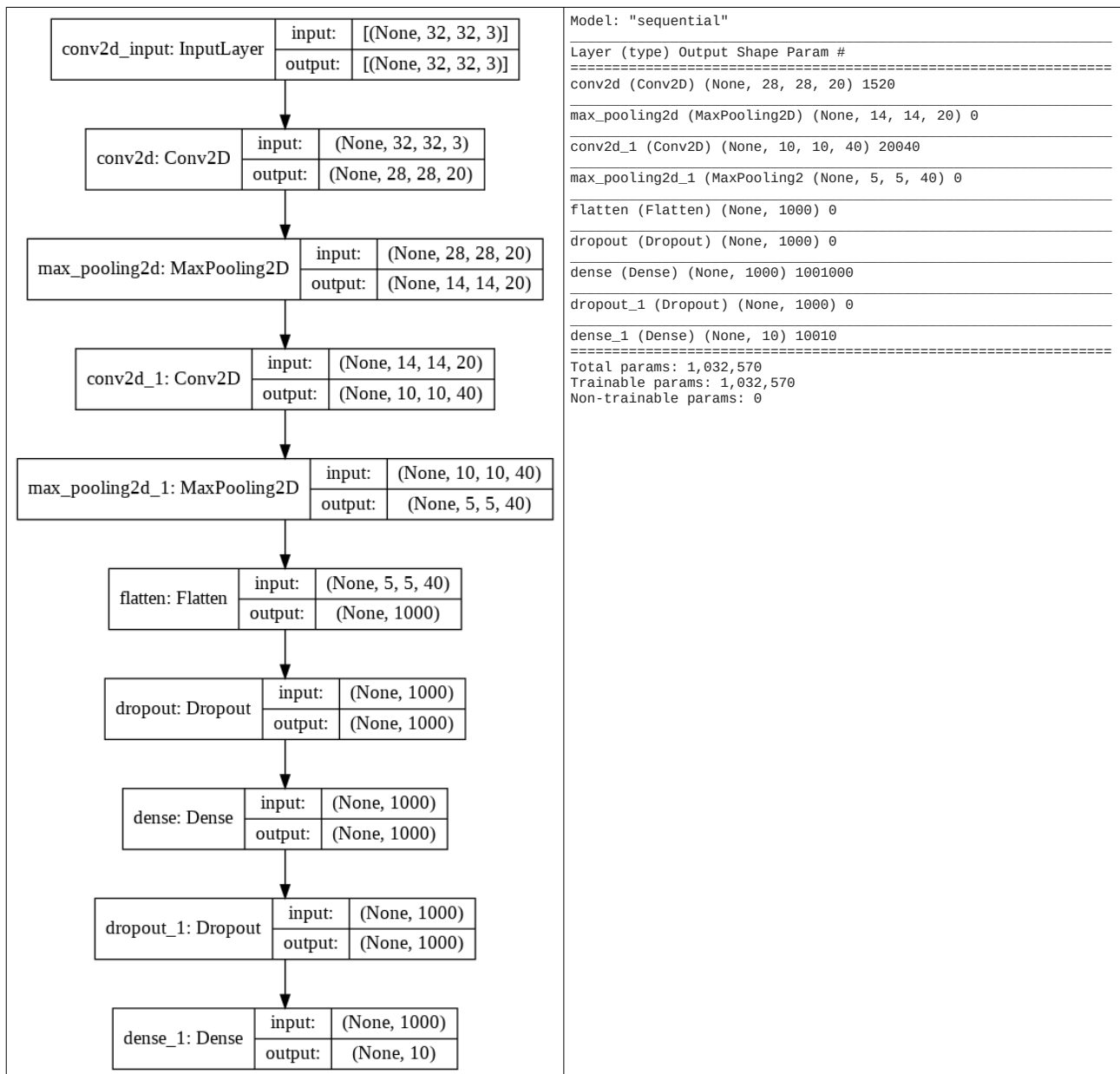
A acuracidade obtida foi de 74,4%, após 30 épocas. Mesmo executando mais épocas, a acuracidade de teste não vai além de 75%. Este programa também mostra como exibir o gráfico (história) do custo e da acuracidade ao longo do treino (em amarelo).

Nota: O programa usa os dados de teste para validação – não é inteiramente correto fazer isto.

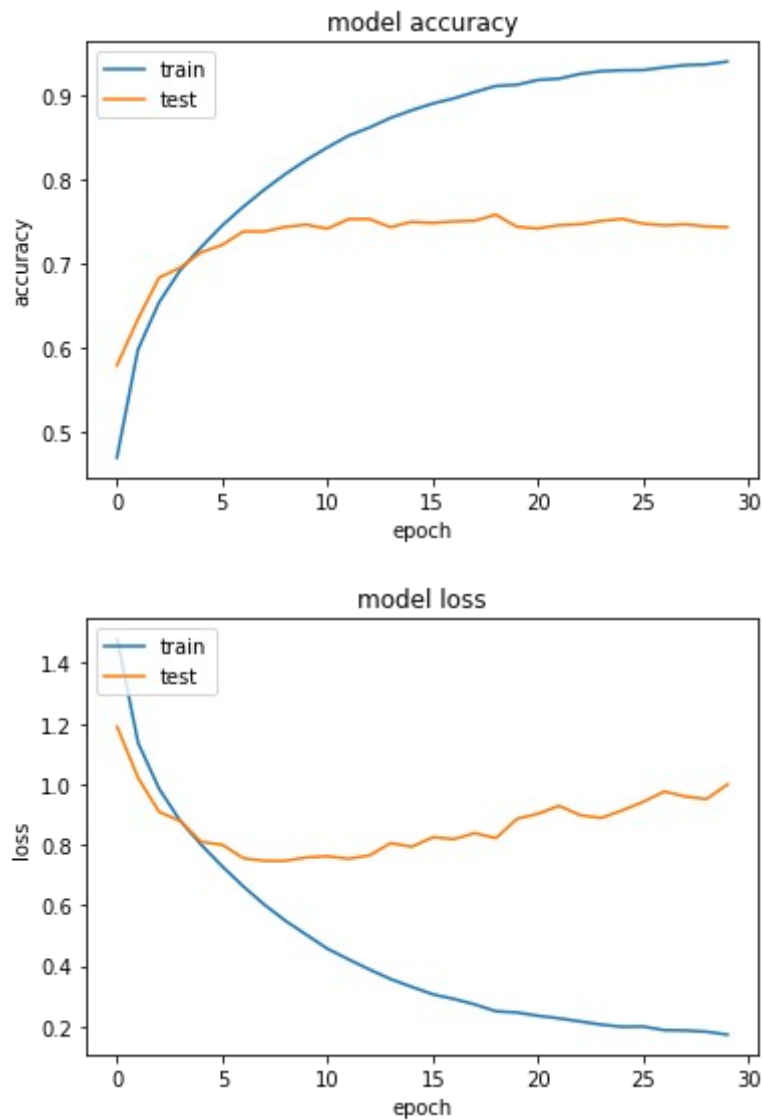
```
1 #cnn1.py
2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
4 import tensorflow.keras as keras
5 from tensorflow.keras.datasets import cifar10
6 from tensorflow.keras.models import Sequential
7 from tensorflow.keras.layers import Dropout, Conv2D, MaxPooling2D, Dense, Flatten
8 from tensorflow.keras import optimizers
9 import matplotlib.pyplot as plt; import numpy as np
10
11 def impHistoria(history):
12     print(history.history.keys())
13     plt.plot(history.history['accuracy'])
14     plt.plot(history.history['val_accuracy'])
15     plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
16     plt.legend(['train', 'test'], loc='upper left')
17     plt.show()
18     plt.plot(history.history['loss'])
19     plt.plot(history.history['val_loss'])
20     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
21     plt.legend(['train', 'test'], loc='upper left')
22     plt.show()
23
24 batch_size = 100; num_classes = 10; epochs = 30
25
26 n1, nc = 32, 32
27 (ax, ay), (qx, qy) = cifar10.load_data()
28
29 #ax = ax.reshape(ax.shape[0], n1, nc, 3)
30 #qx = qx.reshape(qx.shape[0], n1, nc, 3)
31 input_shape = (n1, nc, 3)
32
33 ax = ax.astype('float32'); ax /= 255; ax -= 0.5; # -0.5 a +0.5
34 qx = qx.astype('float32'); qx /= 255; qx -= 0.5; # -0.5 a +0.5
35 ay = keras.utils.to_categorical(ay, num_classes)
36 qy = keras.utils.to_categorical(qy, num_classes)
37
38 model = Sequential()
39 model.add(Conv2D(20, kernel_size=(5,5), activation='relu', input_shape=input_shape))
40 model.add(MaxPooling2D(pool_size=(2,2)))
41 model.add(Conv2D(40, kernel_size=(5,5), activation='relu'))
42 model.add(MaxPooling2D(pool_size=(2,2)))
43 model.add(Flatten())
44 model.add(Dropout(0.25))
45 model.add(Dense(1000, activation='relu'))
46 model.add(Dropout(0.25))
47 model.add(Dense(num_classes, activation='softmax'))
48
49 from tensorflow.keras.utils import plot_model
50 plot_model(model, to_file='cnn1.png', show_shapes=True)
51 model.summary()
52
53 opt=optimizers.Adam()
54 model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
55
56 history=model.fit(ax, ay, batch_size=batch_size, epochs=epochs,
57                 verbose=2, validation_data=(qx, qy))
58 impHistoria(history)
59
60 score = model.evaluate(qx, qy, verbose=2)
61 print('Test loss:', score[0])
62 print('Test accuracy:', score[1])
63 model.save('cnn1.h5')
```

Programa 2: Rede “tipo LeNet” para Cifar-10. Acuracidade 74,4%.

[https://colab.research.google.com/drive/1TrIo0Z2uLNcvxdQb\\_Orz-yb\\_IRsOYHoO?usp=sharing](https://colab.research.google.com/drive/1TrIo0Z2uLNcvxdQb_Orz-yb_IRsOYHoO?usp=sharing)



Epoch 1/30 - 5s - loss: 1.4811 - accuracy: 0.4704 - val\_loss: 1.1895 - val\_accuracy: 0.5798  
Epoch 5/30 - 2s - loss: 0.7995 - accuracy: 0.7202 - val\_loss: 0.8096 - val\_accuracy: 0.7141  
Epoch 10/30 - 2s - loss: 0.5025 - accuracy: 0.8236 - val\_loss: 0.7583 - val\_accuracy: 0.7470  
Epoch 15/30 - 1s - loss: 0.3306 - accuracy: 0.8827 - val\_loss: 0.7933 - val\_accuracy: 0.7503  
Epoch 20/30 - 2s - loss: 0.2460 - accuracy: 0.9129 - val\_loss: 0.8856 - val\_accuracy: 0.7446  
Epoch 25/30 - 2s - loss: 0.1987 - accuracy: 0.9302 - val\_loss: 0.9134 - val\_accuracy: 0.7538  
Epoch 30/30 - 1s - loss: 0.1728 - accuracy: 0.9405 - val\_loss: 0.9991 - val\_accuracy: 0.7440  
Test loss: 0.9991  
**Test accuracy: 0.7440**



**Exercício:** No programa 2 (cnn1.py), foram utilizados como dados de validação os dados de teste qx, qy2 (o que não é recomendado). Em vez disso, utilize 10% dos dados de treino como dados de validação. Grave o modelo obtido toda vez que a taxa de acerto de validação atingir um valor máximo. No final do treino, haverá dois modelos: um modelo na memória do computador e outro gravado no HD. Verifique quais são as taxas de acerto de teste desses dois modelos.

Pode usar “model.fit(... validation\_split=0.1)”. Isto fará com que Keras reserve 10% dos dados de treino para validação. Vamos usar as seguintes quantidades de imagens: 45000 para treino, 5000 para validação e 10000 para teste.

Solução privada: <https://colab.research.google.com/drive/1bpgkCh7v8hyxOdTSDWoUFHdM2xMS2OiH#scrollTo=a4aVP3czWi1v>  
 Nota: 74.09% modelo na memória. 74.12% melhor modelo gravado. Usando ativação swish, acuracidade cai para 72.5% e 72.9%.

**[PSI5790 aula 6 parte 2. Fim.]**

## [PSI5790 aula 7. Início.]

### Melhorando rede “tipo LeNet”

Vamos duplicar o número de convoluções (linhas 39 e 41 do programa 2) e rodar durante 200 épocas. Mesmo com isso, a acuracidade de teste chegou a apenas 75,5% (antes era 74,4%).

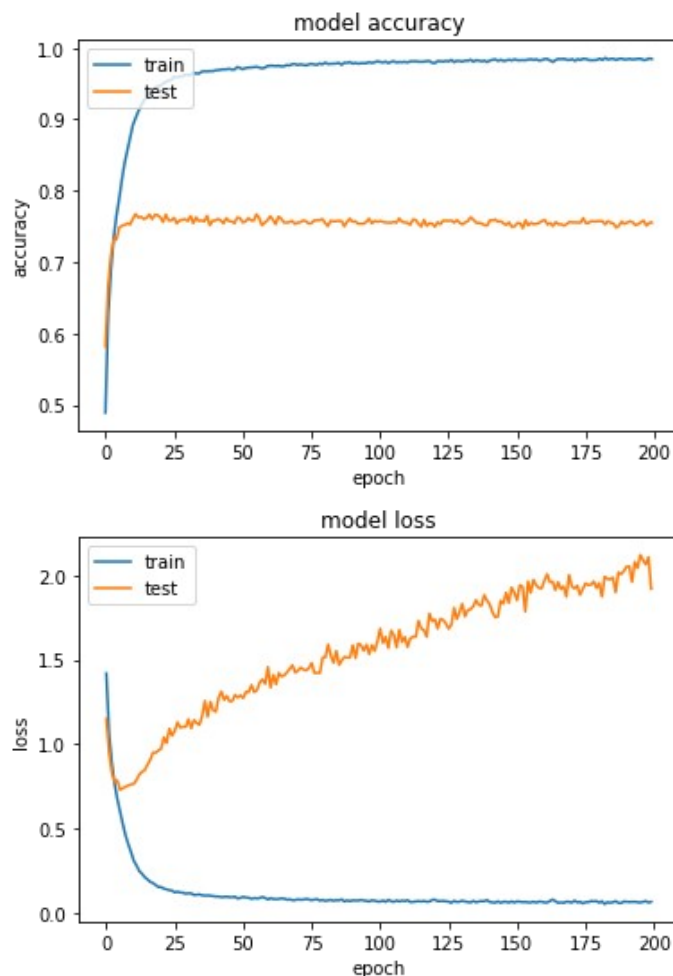
Veja na figura “model loss” abaixo que a função custo de treino diminui continuamente com o aumento das épocas, porém a função custo de teste aumenta após atingir um mínimo. Quanto mais o modelo aprende a classificar corretamente os dados de treino, pior classifica os dados de teste - isto caracteriza “overfitting”. O modelo torna-se especializado em classificar os dados de treino, mas erra ao tentar classificar os dados não vistos.

A tentação aqui é parar o treino quando a função custo de teste atinge o valor mínimo (ou quando a acuracidade de teste atinge o valor máximo). Mas é errado fazer isso. Você consegue explicar por quê? Explique como poderia tomar uma ação semelhante, sem fazer nada “ilícito”.

*Resposta:* Use “model.fit(..., validation\_split=0.2)”. Isto fará com que reserve 20% dos dados de treino para validação.

```
... #cnn2.py
39 model.add(Conv2D(40, kernel_size=(5,5), activation='relu', input_shape=input_shape))
...
41 model.add(Conv2D(80, kernel_size=(5,5), activation='relu'))
```

Saida Epoch 200/200 - 2s - loss: 0.0650 - accuracy: 0.9844 - val\_loss: 1.9211 - val\_accuracy: 0.7555  
Test loss: 1.9210  
Test accuracy: 0.7555



Programa 3: Modelo “tipo LeNet” para Cifar-10 com mais convoluções, mostrando “overfitting”.

Acuracidade 75,5%. <https://colab.research.google.com/drive/10D3mg-vcxGNUKIMQ-ErP4gkdUDFRmGhW?usp=sharing>



*Exercício:* Rode `cnn1.py` (programa 2).

*Exercício:* Modifique `cnn1.py` (programa 2) para obter programa `caogato1.py` que distingue cachorros dos gatos. Para isso:

- Coloque dentro dos tensores `ax`, `ay`, `qx` e `qy` somente as imagens de cachorro e gato.
- Modifique o programa `cnn1.py` para distinguir cachorros dos gatos. Mostre a taxa de acertos obtida.
- Mostre na tela as imagens de 3 primeiros cachorros e 3 primeiros gatos do conjunto de testes e classifique-as (mostre as probabilidades calculadas pelo softmax), para se certificar de que o seu programa está fazendo o que o enunciado pede.

**Exercício:** Vamos supor que as imagens de “sapos” (classe 6) do Cifar-10 representam exames de imagem de pacientes “doentes” e as demais categorias representam exames de pacientes “sadios”. Modifique o programa `cnn1.py` para que identifique se um paciente está “doente” (sapo) ou “sadio” (não-sapo). Para isso, você pode usar uma das três técnicas abaixo:

- Modifique `cnn1.py` para que a saída tenha 2 neurônios (em vez de 10 originais). A ativação 0 indicará a probabilidade do paciente ser sadio e a ativação 1 indicará a probabilidade de ser doente.
- Modifique `cnn1.py` para que a saída tenha um único neurônio. Neste caso, quanto mais a saída estiver próximo de 1, maior será a chance do paciente estar doente. Aqui, precisa trocar softmax pelo sigmoide e trocar a função de custo para `binary_crossentropy`.
- Mantenha `cnn1.py` com 10 neurônios de saída. Use a probabilidade  $p_s$  do sapo como a probabilidade do paciente estar doente. Evidentemente, a probabilidade do paciente estar sadio deve ser  $1-p_s$  (que deve ser igual à soma das outras 9 classes).

Faça o seu programa imprimir os seguintes dados:

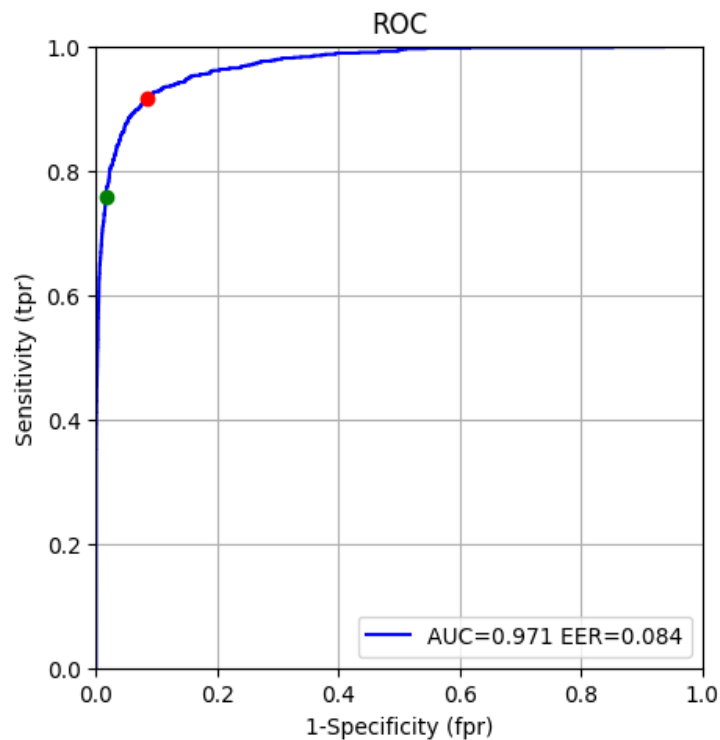
- Adote `limiar=0.5` e calcule sensibilidade, especificidade e acuracidade. Para mim, deu respectivamente 75.9%, 98.3% e 96.1%
- Plote curva ROC, coloque um ponto em EER e outro no ponto com `limiar=0.5`. Calcule AUC. Para mim, deu o gráfico mostrado abaixo, com `AUC=0.971`.
- Calcule o limiar e acuracidade em EER. Note que sensibilidade, especificidade e acuracidade são todas iguais em EER. Para mim, deu `limiar=0.073` e `acuracidade=91.6%`.

[https://en.wikipedia.org/wiki/Sensitivity\\_and\\_specificity](https://en.wikipedia.org/wiki/Sensitivity_and_specificity)

[https://en.wikipedia.org/wiki/Receiver\\_operating\\_characteristic](https://en.wikipedia.org/wiki/Receiver_operating_characteristic)

[https://www.w3schools.com/python/python\\_ml\\_auc\\_roc.asp](https://www.w3schools.com/python/python_ml_auc_roc.asp)

[https://scikit-learn.org/stable/auto\\_examples/model\\_selection/plot\\_roc.html](https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html)



Solução privada em: <https://colab.research.google.com/drive/1h3OxemQfqtmUFpmRnbCzHtevEHlIyyV9#scrollTo=a4aVP3czWi1v>

*Exercício:* Faça um programa que:

- Lê o modelo caogato1.h5 criado no exercício anterior.
- Lê uma imagem colorida 32×32.
- Responde se a imagem é de cachorro ou de gato.

*Exercício:* Altere cnn1.py (programa 2) para obter taxa de acerto de teste maior que 75%, mas continuando a usar rede convolucional “tipo LeNet”.

*Exercício:* Altere cnn1.py (programa 2) imprimir 2 imagens classificadas incorretamente para cada classe. Isto é, imprima 2 aviões que foram classificados incorretamente, 2 automóveis, 2 pássaros, etc.

## 4. Data augmentation

Antes de prosseguirmos, vamos estudar as classes de Keras que permitem fazer “data augmentation”. Data augmentation consiste em distorcer (em geral aleatoriamente) as imagens de treino, obtendo uma quantidade maior de imagens de treino. Já usamos “data augmentation” para melhorar taxa de acerto de MNIST, na aula “convkeras-ead”. Lá, deslocamos as imagens um pixel nos sentidos N, S, L e O.

Keras possui funções que distorcem as imagens de forma mais sofisticada. As funções de data augmentation de Keras permitem, por exemplo:

- Deslocar a imagem horizontal ou verticalmente;
- Espelhar a imagem horizontal ou verticalmente;
- Rotacionar a imagem;
- Mudar brilho/contraste da imagem;
- Redimensionar a imagem;
- Cropping.

Há um tempo atrás, data augmentation era feito em Keras com *ImageDataGenerator*:

<https://keras.io/api/preprocessing/image/>

<https://machinelearningmastery.com/how-to-configure-image-data-augmentation-when-training-deep-learning-neural-networks/>

Parece que este método está obsoleto. Ele continua funcionando, mas de uma forma diferente de antigamente.

Método moderno de fazer data augmentation em Keras é usar camadas de pré-processamento, introduzidas mais recentemente (2020):

[https://keras.io/api/layers/preprocessing\\_layers/](https://keras.io/api/layers/preprocessing_layers/)

## 5.2 Preprocessing Layers

Camadas de pré-processamento foram introduzidas mais recentemente em Keras (2020):

[https://keras.io/api/layers/preprocessing\\_layers/](https://keras.io/api/layers/preprocessing_layers/)

O programa abaixo mostra três camadas de pré-processamento em funcionamento. Antes de executar este programa, deve baixar as imagens exemplos rodando programa X - “célula que faz *download* das 3 imagens”. A rotação é especificada como fração de 360 graus. Assim, para especificar 15 graus, deve fornecer o parâmetro 15/360.

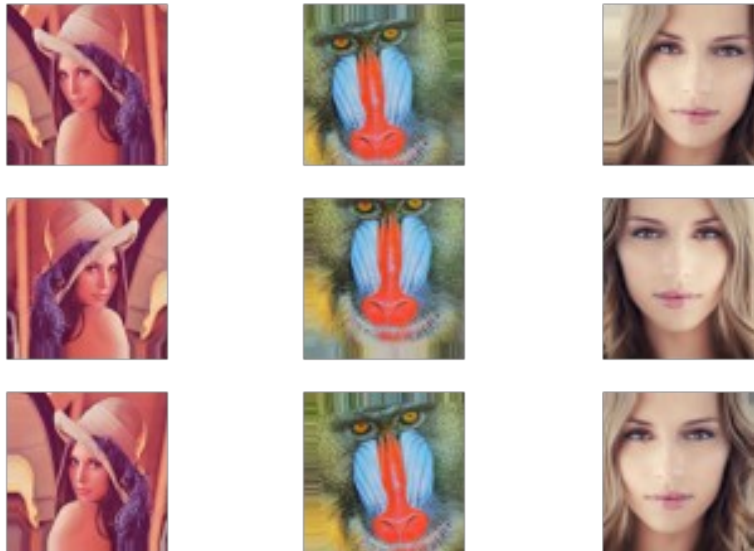
```
url='http://www.lps.usp.br/hae/apostila/cifar_pretreinado.zip'
import os; nomeArq=os.path.split(url)[1]
if not os.path.exists(nomeArq):
    print("Baixando o arquivo",nomeArq,"para diretorio default",os.getcwd())
    os.system("wget -U 'Firefox/50.0' "+url)
else:
    print("O arquivo",nomeArq,"ja existe no diretorio default",os.getcwd())
print("Descompactando arquivos novos de",nomeArq)
os.system("unzip -u "+nomeArq)
```

Programa X: Célula que faz *download* das 3 imagens.

```
1 #prelayer2.py
2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
4 import numpy as np
5 from tensorflow.keras.models import Sequential
6 from tensorflow.keras.layers import RandomFlip, RandomRotation, RandomZoom, RandomTransla-
7 tion
8 import matplotlib.pyplot as plt
9 import cv2; import sys
10
11 X = []
12 a=cv2.imread('lenna.jpg',1); a=cv2.cvtColor(a,cv2.COLOR_BGR2RGB); X.append(a)
13 a=cv2.imread('mandrill.jpg',1); a=cv2.cvtColor(a,cv2.COLOR_BGR2RGB); X.append(a)
14 a=cv2.imread('face.jpg',1); a=cv2.cvtColor(a,cv2.COLOR_BGR2RGB); X.append(a)
15 X=np.array(X).astype("float32")
16
17 layer = Sequential(
18     [
19         RandomRotation(15/360,fill_mode="nearest",interpolation="bilinear"),
20         RandomTranslation(0.1, 0.1,fill_mode="nearest",interpolation="bilinear"),
21         RandomFlip("horizontal")
22     ]
23 )
24
25 for l in range(3):
26     transformedX=layer(X).numpy()
27     for c in range(3):
28         plt.subplot(3, 3, 3*l+c+1)
29         image = transformedX[c].astype("uint8")
30         plt.imshow(image)
31         plt.axis("off")
32 plt.show()
```

Programa Y: prelayer2.py

<https://colab.research.google.com/drive/1y9mndSnxnnv-d4sjH7kTKzc6K54vF0vc?usp=sharing>



O programa abaixo mostra como distorcer as imagens de Cifar10 usando preprocessing layers.

```
#prelayer3.py
import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
import tensorflow.keras as keras
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import RandomFlip, RandomRotation, RandomZoom, RandomTranslation
import matplotlib.pyplot as plt; import numpy as np

(ax, ay), (qx, qy) = cifar10.load_data()
nl, nc = ax.shape[1], ax.shape[2] #32x32

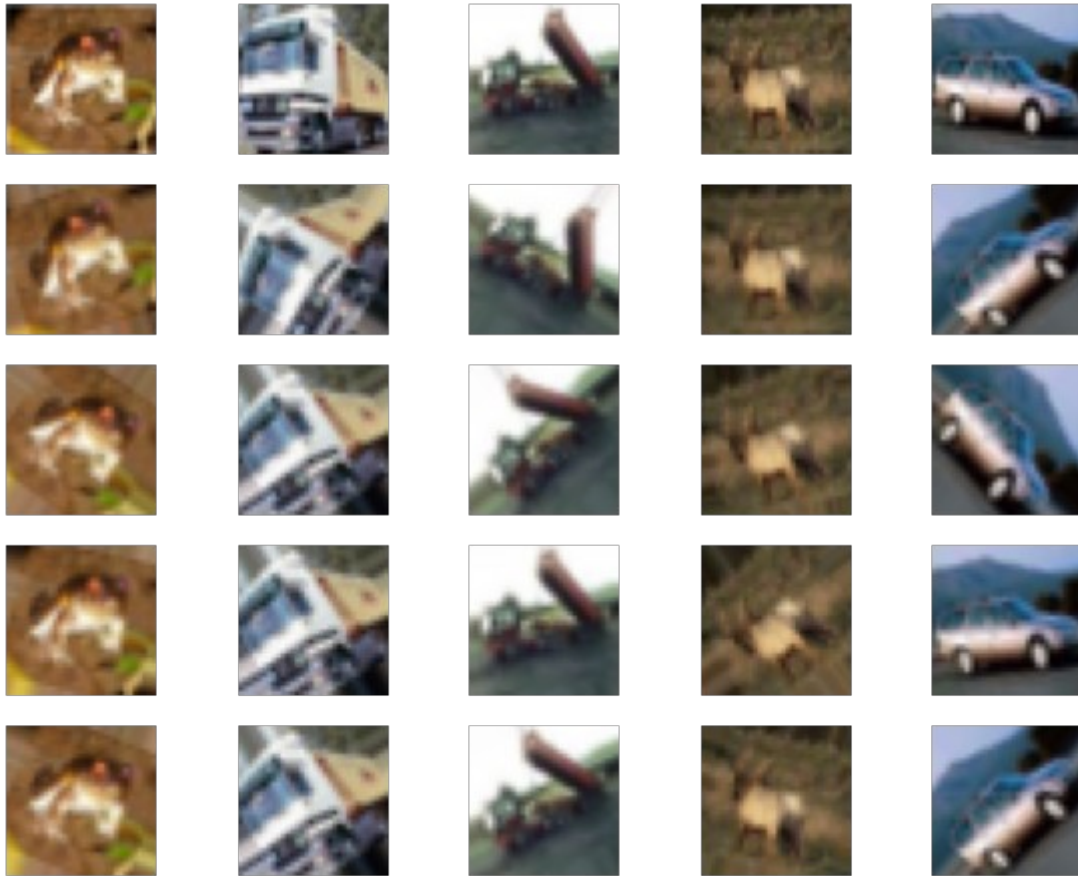
ax = ax.astype('float32'); ax /= 255; #0 a 1
qx = qx.astype('float32'); qx /= 255; #0 a 1

prelayer = Sequential(
    [ RandomRotation(60/360, fill_mode="nearest", interpolation="bilinear"),
      #RandomTranslation(0.1, 0.1, fill_mode="nearest", interpolation="bilinear"),
      #RandomFlip("horizontal"),
      #Outras transformacoes
    ]
)

fig = plt.figure()
fig.set_size_inches(10, 8)
nc=5; nl=5
X=ax[0:nc,:,:]
for c in range(nc):
    a = fig.add_subplot(nl, nc, c+1)
    image=X[c]; a.imshow(image); a.axis("off")
for l in range(1,nl):
    transformedX=prelayer(X).numpy()
    for c in range(nc):
        a = fig.add_subplot(nl, nc, nc*l+c+1)
        image = transformedX[c]; a.imshow(image); a.axis("off")
plt.show()
```

Programa Z: prelayer3.py

<https://colab.research.google.com/drive/1y9mndSnxnnv-d4sjH7kTKzc6K54vF0vc?usp=sharing#scrollTo=5BoQPcFqRpL>



**[PSI5790 Aula 7. Lição de casa #1/2. Vale 5.]** Modifique o programa Z para visualizar as seguintes transformações aleatórias, uma de cada vez:

- 1) RandomContrast
- 2) RandomBrightness

Veja: [https://keras.io/api/layers/preprocessing\\_layers/](https://keras.io/api/layers/preprocessing_layers/)

Descreva quais devem ser os parâmetros de brilho/contraste úteis para visão computacional.



*Exercício:* Modifique o programa 1 da apostila convkeras-ead (cnn1.py que classifica MNIST) para aumentar taxa de acerto fazendo *data augmentation*. Você conseguiu chegar a qual taxa de erro de teste? O programa original comete 0,66% de erro.

**[PSI5790 Aula 7. Lição de casa #2/2. Vale 5.]** Modifique o programa 2 desta apostila (cnn1.py que classifica Cifar10) para aumentar a taxa de acerto usando as rotinas de *data augmentation* de Keras. O programa original tem taxa de acerto de teste de 74,4%. Você precisa atingir uma taxa de acerto de teste de mais de 76%. Conseguiu chegar a qual taxa de acerto de teste?

Nota: Não mude os seguintes parâmetros, para que o treino não demore excessivamente:

```
batch_size = 100; epochs = 30
```

Sugestão: Usando *preprocessing layers*, precisa fornecer *input\_shape* no primeiro layer, algo como:

```
prelayer = Sequential(  
    [ AlgumaTransformacao(parametros, input_shape=(32, 32, 3)),  
      OutrasTransformacoes...  
    ]  
)
```

Nota: Uma alternativa é fornecer o formato de entrada usando o comando `model.build(input_shape)`, após ter definido a rede:

```
model = Sequential()  
model.add(prelayer)  
model.add(OutrosLayers)  
model.build(input_shape=(None, 32, 32, 3))
```

Também é possível usar a camada `Input(shape)`. [https://keras.io/api/layers/core\\_layers/input/](https://keras.io/api/layers/core_layers/input/)

Solução privada em `cnn1_prelayer1.ipynb`.

[https://colab.research.google.com/drive/1jl5wG-FXnCaZWxbAaO8jGB46XSmX\\_1Mv](https://colab.research.google.com/drive/1jl5wG-FXnCaZWxbAaO8jGB46XSmX_1Mv)

[https://colab.research.google.com/drive/1jun\\_EKzjTKy5OVIePy7eYcnolHAXVqVt](https://colab.research.google.com/drive/1jun_EKzjTKy5OVIePy7eYcnolHAXVqVt)

<https://colab.research.google.com/drive/1ulixrKhHtkS8K0Bhb5UQXKDg7WUea5O7>

*Observação:* Nos meus testes, obtive os seguintes resultados em classificar Cifar-10:

1) Usando *ImageDataGenerator* e acertando os parâmetros, cheguei à taxa de acerto de 78,1%.

2) Usando *Preprocessing Layer*, cheguei à taxa de 77,7% usando interpolação bilinear.

Não sei porque *ImageDataGenerator* acerta mais, apesar de supostamente fazer o mesmo processamento que *Preprocessing Layer*. Usando *Preprocessing Layer*, processamento fica 2-3 vezes mais rápido que usando *ImageDataGenerator*, provavelmente porque o primeiro roda em GPU enquanto que o segundo roda em CPU.

Nota: Talvez devido à pequena resolução das imagens de Cifar10, nem todas os aumentos de dados dão bom resultado.

## As principais CNN “avançadas”

Vamos estudar, a seguir, as ideias atrás das principais redes CNN “avançadas”. O blog seguinte faz uma boa descrição delas:

<https://theaisummer.com/cnn-architectures/>

### 5. AlexNet

O desenvolvimento de redes mais modernas está intimamente ligado à ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [Das2017, Wiki-ImageNet]. Em 2012, AlexNet [Krizhevsky2012, Krizhevsky2017] venceu ILSVRC com 16% de erro “top-5”, muito menor que 26% do segundo lugar. A taxa de erro “top-1” de AlexNet foi 37,5%. Foi uma revolução em visão computacional. Este trabalho mostrou definitivamente o poder das redes convolucionais.

AlexNet é basicamente “modelo tipo LeNet” com mais camadas. Enquanto que LeNet foi projetado para classificar pequenas imagens com  $28 \times 28$  ou  $32 \times 32$  pixels, AlexNet foi projetado para classificar imagens grandes, da ordem de  $224 \times 224$  pixels. Contém 8 camadas, sendo as 5 primeiras convolucionais ( $11 \times 11$ ,  $5 \times 5$  ou  $3 \times 3$ , algumas delas seguidas por max-pooling, veja figura 7) e as 3 últimas são camadas densas. Utiliza “overlapping max-pooling” com janela  $3 \times 3$  e stride de 2.

AlexNet introduziu várias inovações (que já usamos nos exercícios anteriores):

- 1) Usou a função de ativação *Relu* (em vez de *tanh* que era padrão naquele tempo). *Relu* diminui o tempo de treinamento e minimiza o problema de gradiente que “desaparece” ou “explode”. Note que a derivada da *Relu* é 0 ou 1 para qualquer  $x$ .
- 2) Usou “data augmentation” e “dropout” para diminuir “overfitting”.

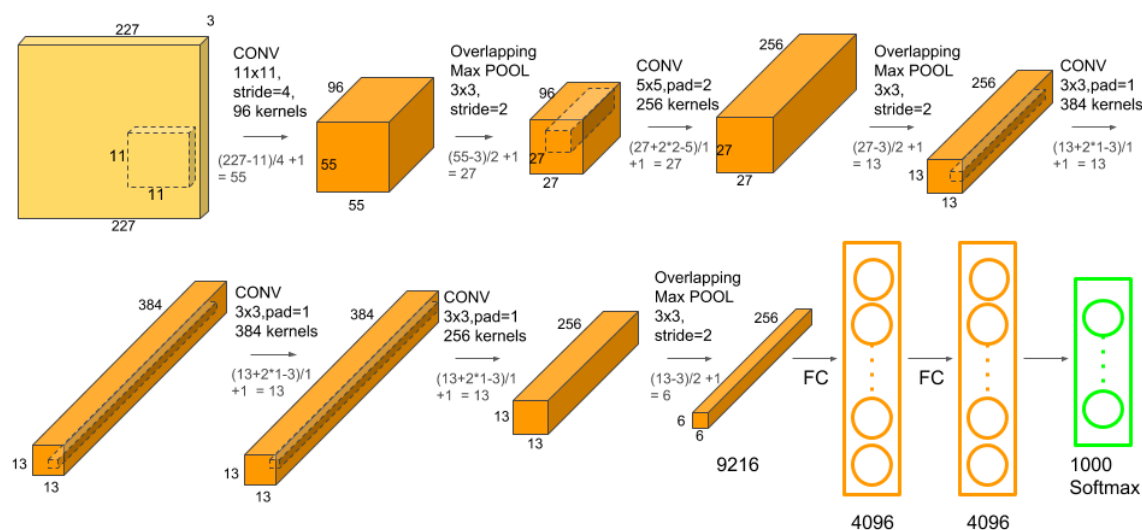


Figura 7: Modelo AlexNet.

**Exercício:** Implemente uma rede “inspirada em AlexNet” e classifique Cifar10. Qual foi a taxa de erro obtida?

## 6 Rede “inspirada em VGG”

### 6.1 Introdução

O VGG é uma arquitetura de rede neural convolucional que recebeu o nome do Visual Geometry Group de Oxford, que o desenvolveu. Ficou em segundo lugar em competição ILSVR (ImageNet) de 2014. Vimos na seção anterior que AlexNet utilizava camadas convolucionais com janelas grandes ( $11 \times 11$  e  $5 \times 5$ ). VGG substituiu essas convoluções grandes por sequências de camadas convolucionais  $3 \times 3$  [Simonyan2014]. A ideia é que várias camadas de convoluções  $3 \times 3$  conseguem aprender o mesmo que convoluções usando janelas maiores. Com isso, a rede tornou-se mais profunda.

A sequência de duas camadas convolucionais  $3 \times 3$  possui campo de visão  $5 \times 5$  com  $9+9=18$  pesos, o mesmo que uma camada convolucional  $5 \times 5$  que possui 25 pesos (estamos ignorando os vieses). A sequência de 4 camadas  $3 \times 3$  possui campo de visão  $9 \times 9$  com  $9+9+9+9=36$  pesos, enquanto que uma camada convolucional  $9 \times 9$  possui 81 pesos. Assim, uma sequência de convoluções  $3 \times 3$  consegue ter o mesmo campo de visão que uma convolução grande, com menos parâmetros. A figura 9 mostra a estrutura da rede VGG.

O programa 5, inspirado em VGG mas com poucas camadas convolucionais, resolve Cifar-10 usando somente convoluções  $3 \times 3$ . A acuracidade obtida (74,3%) é semelhante às dos modelos “tipo LeNet” (programas 2 e 3 com acuracidades 74,4% e 75,5% respectivamente). Basicamente, trocou as duas convoluções  $5 \times 5$  por quatro convoluções  $3 \times 3$  (destacado em amarelo). Camada convolucional com padding=“same” faz imagem de saída ter o mesmo tamanho da imagem de entrada. Os pixels fora do domínio da imagem de entrada são preenchidos com zeros.

*Exercício:* Resolva o problema MNIST usando uma rede convolucional inspirado em VGG usando somente convoluções  $3 \times 3$ . Conseguiu obter taxa de erro menor do que rede “tipo Lenet”?

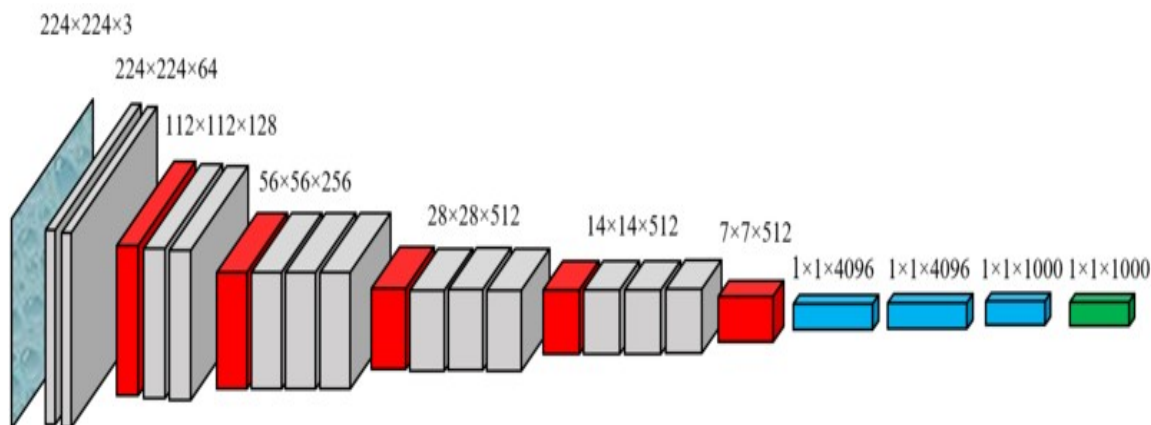


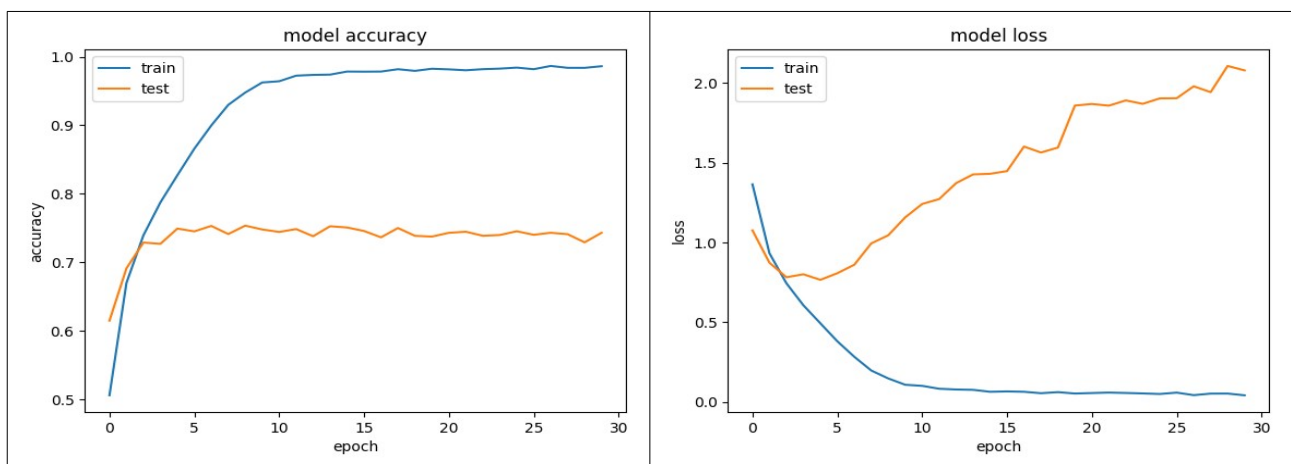
Figura 9: Rede VGG. Cinza = convolução + relu. Vermelho = max pooling. Azul = fully connected + relu. Verde = softmax [Wikipedia].

```

1 #vgg1_umaum.py
2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
4 import tensorflow.keras as keras
5 from tensorflow.keras.datasets import cifar10
6 from tensorflow.keras.models import Sequential
7 from tensorflow.keras.layers import Dropout, Conv2D, MaxPooling2D, Dense, Flatten
8 from tensorflow.keras import optimizers
9 import matplotlib.pyplot as plt; import numpy as np
10
11 def impHistoria(history):
12     print(history.history.keys())
13     plt.plot(history.history['accuracy']); plt.plot(history.history['val_accuracy'])
14     plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
15     plt.legend(['train', 'test'], loc='upper left'); plt.show()
16     plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])
17     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
18     plt.legend(['train', 'test'], loc='upper left'); plt.show()
19
20 batch_size = 100; num_classes = 10; epochs = 30
21 nl, nc = 32,32; input_shape = (nl, nc, 3)
22 (ax, ay), (qx, qy) = cifar10.load_data()
23 ax = ax.astype('float32'); ax /= 255; ax = 2*(ax-0.5) #-1 a +1
24 qx = qx.astype('float32'); qx /= 255; qx = 2*(qx-0.5) #-1 a +1
25 ay = keras.utils.to_categorical(ay, num_classes)
26 qy = keras.utils.to_categorical(qy, num_classes)
27
28 model = Sequential() #32x32x3
29 model.add(Conv2D(20, kernel_size=(3,3), activation='relu', padding='same', input_shape=input_shape))
30 model.add(MaxPooling2D(pool_size=(2,2))) #16x16x20
31 model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='same'))
32 model.add(MaxPooling2D(pool_size=(2,2))) #8x8x40
33 model.add(Conv2D(80, kernel_size=(3,3), activation='relu', padding='same'))
34 model.add(MaxPooling2D(pool_size=(2,2))) #4x4x80
35 model.add(Conv2D(160, kernel_size=(3,3), activation='relu', padding='same')) #160x4x4
36 model.add(Flatten())
37 model.add(Dense(1000, activation='relu'))
38 model.add(Dense(num_classes, activation='softmax'))
39
40 from tensorflow.keras.utils import plot_model
41 plot_model(model, to_file='vgg1.png', show_shapes=True); model.summary()
42 opt=optimizers.Adam()
43 model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
44 history=model.fit(ax, ay, batch_size=batch_size, epochs=epochs,
45                 verbose=2, validation_data=(qx, qy))
46 impHistoria(history)
47
48 score = model.evaluate(qx, qy, verbose=0)
49 print('Test loss:', score[0]); print('Test accuracy:', score[1])
50 model.save('vgg1_umaum.h5')

```

Programa 5: Rede para classificar Cifar-10 com somente 4 camadas convolucionais 3×3. Atinge acuracidade de 74,2%. [https://colab.research.google.com/drive/1zWpVPwDqI\\_-Fb54r2vI3ev-tEpL4Vb2h?usp=sharing](https://colab.research.google.com/drive/1zWpVPwDqI_-Fb54r2vI3ev-tEpL4Vb2h?usp=sharing)



Test loss: 2.0786325931549072  
Test accuracy: 0.7429999709129333

## 6.2 Rede “tipo VGG” com mais camadas

Para poder aumentar a taxa de acerto seria necessário tornar a rede mais profunda, colocando mais camadas convolucionais. Só que, colocando muitas camadas, a rede deixa de convergir. O programa 6, VGG com 10 camadas convolucionais 3×3, não converge. A acuracidade é sempre 10% (isto é, igual à “chute”) até o final do programa com 30 épocas (pode ser que, se tiver sorte, a rede convirja).

```
#vgg2_umaum.py
(...)
model = Sequential()

model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='same', input_shape=input_shape))
model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2))) #16x16x40

model.add(Conv2D(80, kernel_size=(3,3), activation='relu', padding='same'))
model.add(Conv2D(80, kernel_size=(3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2))) #8x8x80

model.add(Conv2D(160, kernel_size=(3,3), activation='relu', padding='same'))
model.add(Conv2D(160, kernel_size=(3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2))) #4x4x160

model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2))) #2x2x320

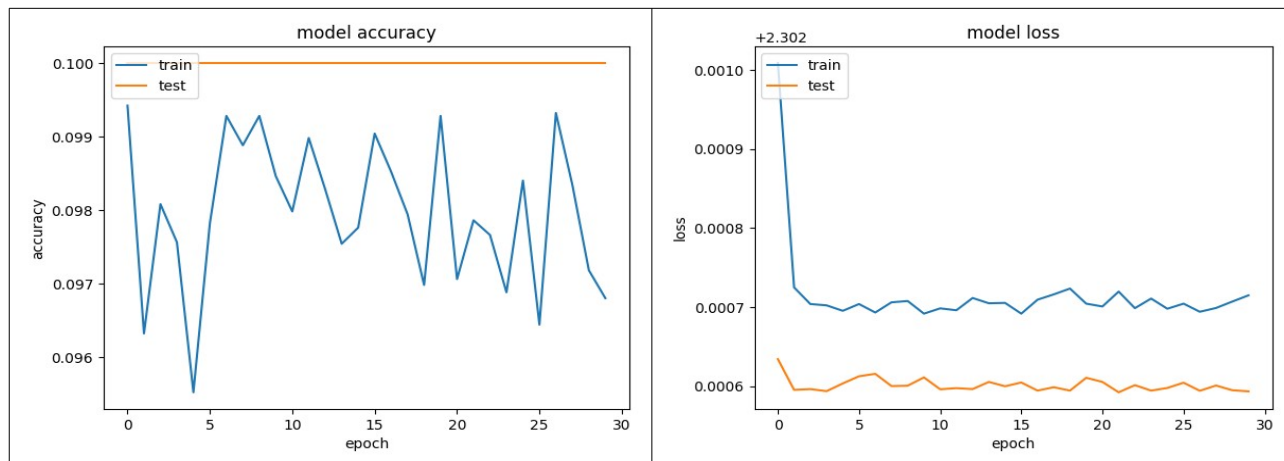
model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
model.add(MaxPooling2D(pool_size=(2,2))) #1x1x320

model.add(Flatten())
model.add(Dense(1000, activation='relu'))

model.add(Dense(num_classes, activation='softmax'))
(...)
```

Programa 6: Rede “inspirada em VGG” para classificar Cifar-10 com 10 camadas convolucionais. A rede não converge e a acuracidade é 10% após 30 épocas.

<https://colab.research.google.com/drive/irLhrcMtrYwMehQJfULgE9oMQM27aQrXH?usp=sharing>



Test loss: 2.3025922775268555

Test accuracy: 0.10000000149011612

*Exercício:* Você consegue fazer o programa 6 convergir (quase sempre), sem usar batch normalization?

### 6.3 Batch normalization

Para fazer convergir uma rede com muitas camadas, devemos usar algum “truque”. VGG original não usou “batch normalization”, pois esta ainda não havia sido inventada. Como vimos no programa 6, é difícil treinar uma rede neural muito profunda.

Gradiente que explode ou desaparece.

O programa 7 é igual ao programa 6, mas com camadas “batch normalization”. Ele atinge acuracidade de teste de 83,9%. Agora, a acuracidade melhorou substancialmente em relação à acuracidade da rede “tipo LeNet” (75,5%).

Já vimos que cada atributo de entrada deve seguir mais ou menos distribuição normal com média zero e desvio-padrão um. Isto pode ser conseguido subtraindo a média e dividindo pelo desvio-padrão os dados de entrada. Porém, nas camadas internas da rede os dados podem não seguir distribuição normal(0,1). Batch normalization normaliza os lotes de dados de camadas internas da rede. Com isso, batch normalization permite que cada camada da rede trabalhe “um pouco mais independentemente” de outras camadas.

Se algum atributo de entrada estiver no intervalo de 100 a 1000 (por exemplo), o treino da rede neural não vai convergir. Devemos mudar a escala desse atributo para algo como -1 a +1, -0.5 a +0.5, etc.

Durante o treino, batch normalization calcula a média  $\mu_B$  e desvio-padrão  $\sigma_B$  do lote B. No caso de imagens, são calculadas média e desvio-padrão para cada banda de cor (ou para cada mapa de atributos se for camada interna). Depois, normaliza-se o batch subtraindo  $\mu_B$  e dividindo por  $\sigma_B + \epsilon$  cada elemento de B, onde  $\epsilon$  no denominador é um número pequeno para evitar divisão por zero.

*Detalhes:*

Nem sempre esta distribuição é adequada para alimentar a camada seguinte. Consequentemente, a saída de batch normalization é “desnormalizada” multiplicando por um parâmetro “desvio padrão”  $\gamma$  (inicializada com 1) e somando um parâmetro “média”  $\beta$  (inicializada com 0).

$$B = ((B - \mu_B) / (\sigma_B + \epsilon)) * \gamma + \beta$$

onde B são os elementos do batch. Os parâmetros ótimos  $\gamma$  e  $\beta$  são atualizados pela retro-propagação.

Nota: Não dá para usar média e desvio de teste, pois pode querer classificar uma única imagem.

Durante a predição, quais média  $\mu_B$ , desvio  $\sigma_B$  devem ser usados: a média dos parâmetros do treino ou do lote do teste? Normalização de lote usa  $\mu_B$ ,  $\sigma_B$ ,  $\gamma$  e  $\beta$  calculados no treino. Porém,  $\mu_B$  e  $\sigma_B$  são atualizados durante a predição, para se ajustar à estatística dos dados de teste:

$$B = (B - \text{moving\_}\mu_B) / (\text{moving\_}\sigma_B + \epsilon) * \gamma + \beta$$

onde moving\_ $\mu_B$  e moving\_ $\sigma_B$  são  $\mu_B$  e  $\sigma_B$  calculados durante o treino e atualizados durante a predição como média móvel.

Veja [[https://keras.io/api/layers/normalization\\_layers/batch\\_normalization/](https://keras.io/api/layers/normalization_layers/batch_normalization/)] para mais detalhes.



<b>Input:</b>	Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$ ;
	Parameters to be learned: $\gamma, \beta$
<b>Output:</b>	$\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$
	$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i$ // mini-batch mean
	$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2$ // mini-batch variance
	$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$ // normalize
	$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i)$ // scale and shift

**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

[Do artigo original de batch-normalization \[Ioffe2015\].](#)

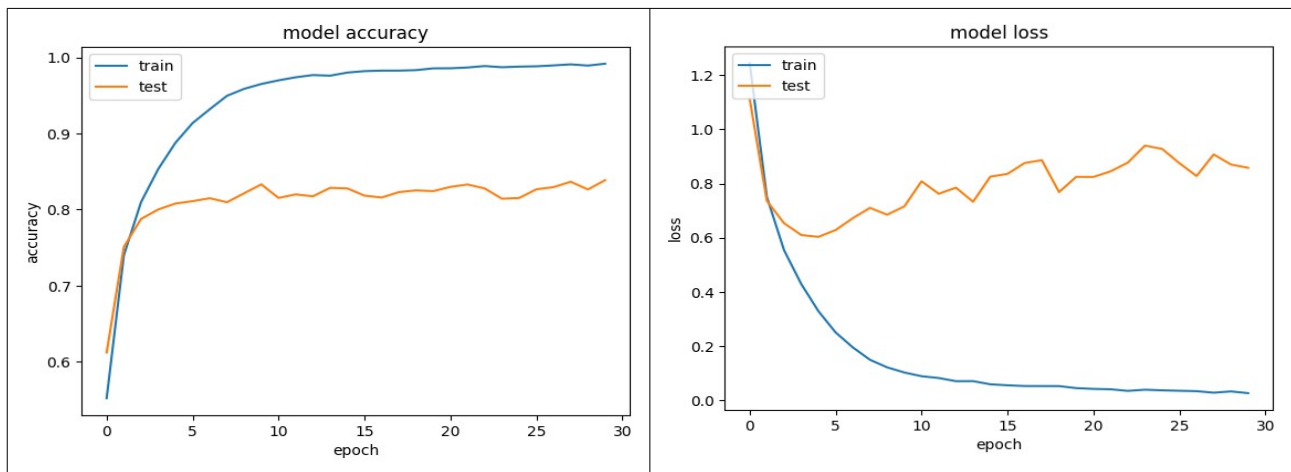
```

1 #vgg3_umaum.py
2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
4 import tensorflow.keras as keras
5 from tensorflow.keras.datasets import cifar10
6 from tensorflow.keras.models import Sequential
7 from tensorflow.keras.layers import Dropout, Conv2D, MaxPooling2D, Dense, \
8     Flatten, BatchNormalization
9 from tensorflow.keras import optimizers
10 import matplotlib.pyplot as plt; import numpy as np
11
12 def impHistoria(history):
13     print(history.history.keys())
14     plt.plot(history.history['accuracy']); plt.plot(history.history['val_accuracy'])
15     plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
16     plt.legend(['train', 'test'], loc='upper left'); plt.show()
17     plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])
18     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
19     plt.legend(['train', 'test'], loc='upper left'); plt.show()
20
21 batch_size = 100; num_classes = 10; epochs = 30
22 nl, nc = 32,32; input_shape = (nl, nc, 3)
23 (ax, ay), (qx, qy) = cifar10.load_data()
24 ax = ax.astype('float32'); ax /= 255; ax = 2*(ax-0.5) #-1 a +1
25 qx = qx.astype('float32'); qx /= 255; qx = 2*(qx-0.5) #-1 a +1
26 ay = keras.utils.to_categorical(ay, num_classes)
27 qy = keras.utils.to_categorical(qy, num_classes)
28
29 model = Sequential()
30 model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='same', input_shape=input_shape))
31 model.add(BatchNormalization())
32 model.add(Conv2D(40, kernel_size=(3,3), activation='relu', padding='same'))
33 model.add(BatchNormalization())
34 model.add(MaxPooling2D(pool_size=(2,2))) #16x16x40
35
36 model.add(Conv2D(80, kernel_size=(3,3), activation='relu', padding='same'))
37 model.add(BatchNormalization())
38 model.add(Conv2D(80, kernel_size=(3,3), activation='relu', padding='same'))
39 model.add(BatchNormalization())
40 model.add(MaxPooling2D(pool_size=(2,2))) #8x8x80
41
42 model.add(Conv2D(160, kernel_size=(3,3), activation='relu', padding='same'))
43 model.add(BatchNormalization())
44 model.add(Conv2D(160, kernel_size=(3,3), activation='relu', padding='same'))
45 model.add(BatchNormalization())
46 model.add(MaxPooling2D(pool_size=(2,2))) #4x4x160
47
48 model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
49 model.add(BatchNormalization())
50 model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
51 model.add(BatchNormalization())
52 model.add(MaxPooling2D(pool_size=(2,2))) #2x2x320
53
54 model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
55 model.add(BatchNormalization())
56 model.add(Conv2D(320, kernel_size=(3,3), activation='relu', padding='same'))
57 model.add(BatchNormalization())
58 model.add(MaxPooling2D(pool_size=(2,2))) #1x1x320
59
60 model.add(Flatten())
61 model.add(Dense(1000, activation='relu'))
62 model.add(Dense(num_classes, activation='softmax'))
63
64 from tensorflow.keras.utils import plot_model
65 plot_model(model, to_file='vgg2.png', show_shapes=True); model.summary()
66 opt=optimizers.Adam()
67 model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
68 history=model.fit(ax, ay, batch_size=batch_size, epochs=epochs,
69                 verbose=2, validation_data=(qx, qy))
70 impHistoria(history)
71
72 score = model.evaluate(qx, qy, verbose=0)
73 print('Test loss:', score[0]); print('Test accuracy:', score[1])
74 model.save('vgg3_umaum.h5')

```

Programa 7: Rede “inspirada em VGG” para classificar Cifar-10 com 10 camadas convolucionais e batch normalizations. Atinge acuracidade de teste de **83,9%**.

<https://colab.research.google.com/drive/1ovCGSJ4dOLylua0SbEb3dZi2qHNjsJdW?usp=sharing>



Test loss: 0.858138382434845  
Test accuracy: 0.838699996471405

Epoch 30/30 - 12s - loss: 0.0257 - accuracy: 0.9919 - val\_loss: 0.8581 - val\_accuracy: 0.8387

*Exercício:* Para descobrir como funciona BatchNormalization, coloquei artificialmente uma camada BatchNormalization bem no início da rede do programa 7:

```
model.add(BatchNormalization(input_shape=input_shape))
```

imprimi os seus parâmetros antes e depois do treino com o comando:

```
filters = model.get_layer(index=0).get_weights()
print(filters)
```

e obtive uma lista com 4 vetores numpy, cada vetor com 3 elementos. Cada vetor tem 3 elementos, pois as imagens de entrada têm 3 bandas RGB. A lista antes do treino:

```
[array([1., 1., 1.], dtype=float32), array([0., 0., 0.], dtype=float32), array([0., 0., 0.], dtype=float32), array([1., 1., 1.], dtype=float32)]
```

A lista após o treino:

```
[array([0.9631283, 1.0356276, 0.9850968], dtype=float32), array([0.03289677, 0.00249975, 0.03328805], dtype=float32), array([0.4916695, 0.48219675, 0.44590157], dtype=float32), array([0.06051334, 0.05886611, 0.06790181], dtype=float32)]
```

1) Modifique o programa 7 colocando uma camada de BatchNormalization no início da rede e imprima os seus parâmetros antes e depois do treino, como fiz acima. Basta treinar poucas épocas, por exemplo 5.

2) Descubra o que cada um dos 4 vetores representa, escrevendo  $\mu_B$ ,  $\sigma_B$ ,  $\gamma$  e  $\beta$  na ordem em que esses vetores aparecem na lista.

## 6.4 Data augmentation

A acuracidade de teste do programa 7 é 83,9% mas a acuracidade de treino é 99,2%, indicando que há “overfitting”. Este problema pode ser minimizado fazendo “data augmentation”.

O programa 8 implementa VGG com data augmentation. Além disso, acrescenta camadas “dropout”, também para diminuir overfitting e executa durante 200 épocas. Com isso, atinge 92,0% de acuracidade. Esta é uma taxa de acerto muito boa. Interessantemente, se fizer somente horizontal\_flip aqui, obtém uma taxa de acerto menor (90,25%).

*Nota:* Aqui, estou colocando os valores de entrada no intervalo de 0 a 1. Porém, se colocar no intervalo [-1, +1] ou [-0.5, +0.5], dá resultado praticamente igual.

O site abaixo disponibiliza um programa inspirado em VGG melhor ainda:

<https://raw.githubusercontent.com/geifmany/cifar-vgg/master/cifar10vgg.py>

Ele atinge 93,3% de taxa de acerto no Cifar-10, usando mais alguns “truques”, entre eles:

- Normaliza as imagens de entrada, tanto durante o treino como durante o teste.
- Utiliza regularização L2. Isto faz com que sejam geradas preferencialmente as redes com pesos pequenos.
- Diminui learning rate à medida em que o treino progride.

*Exercício:* Acrescente as 3 ideias acima no programa 8 e verifique a acuracidade de teste que consegue atingir. Ajuste os parâmetros para obter a maior taxa de acerto de teste.

*Nota:* vgg4\_umaum.py atinge 92,0% de acuracidade de teste. Precisa obter valor melhor.

Abaixo, VGG com data augmentation.

```
1 #vgg4_prelayer_umaum.py
2 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
3 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
4 import tensorflow as tf
5 import tensorflow.keras as keras
6 from tensorflow.keras.datasets import cifar10
7 from tensorflow.keras.models import Sequential
8 from tensorflow.keras.preprocessing.image import ImageDataGenerator
9 from tensorflow.keras.layers import Dropout, Conv2D, MaxPooling2D, Dense, Flatten, BatchNormalization
10 from tensorflow.keras.layers import RandomFlip, RandomRotation, RandomZoom, RandomTranslation
11 from tensorflow.keras import optimizers
12 import matplotlib.pyplot as plt; import numpy as np; import math
13
14 def impHistoria(history):
15     print(history.history.keys())
16     plt.plot(history.history['accuracy']); plt.plot(history.history['val_accuracy'])
17     plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
18     plt.legend(['train', 'test'], loc='upper left'); plt.show()
19     plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])
20     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
21     plt.legend(['train', 'test'], loc='upper left'); plt.show()
22
23 batch_size = 100; num_classes = 10; epochs = 200
24 nl, nc = 32,32; input_shape = (nl, nc, 3)
25 (ax, ay), (qx, qy) = cifar10.load_data()
26 ax = ax.astype('float32'); ax /= 255; ax = 2*(ax-0.5) #-1 a +1
27 qx = qx.astype('float32'); qx /= 255; qx = 2*(qx-0.5) #-1 a +1
28 ay = tf.keras.utils.to_categorical(ay, num_classes)
29 qy = tf.keras.utils.to_categorical(qy, num_classes)
30
31 def create_model():
32     model = Sequential(
33         [
34             RandomRotation(0.042, fill_mode="nearest", interpolation="bilinear"), #15 graus: 15*pi/180
35             RandomTranslation(0.1, 0.1, fill_mode="nearest", interpolation="bilinear"),
36             RandomFlip("horizontal"),
37
38             Conv2D(64, kernel_size=(3,3), activation='relu', padding='same', input_shape=input_shape),
39             BatchNormalization(), Dropout(0.3),
40             Conv2D(64, kernel_size=(3,3), activation='relu', padding='same'),
41             BatchNormalization(),
42             MaxPooling2D(pool_size=(2,2)), #20x16x16x3
43
44             Conv2D(128, kernel_size=(3,3), activation='relu', padding='same'),
45             BatchNormalization(), Dropout(0.3),
46             Conv2D(128, kernel_size=(3,3), activation='relu', padding='same'),
47             BatchNormalization(),
48             MaxPooling2D(pool_size=(2,2)), #40x8x8x3
49
50             Conv2D(256, kernel_size=(3,3), activation='relu', padding='same'),
51             BatchNormalization(), Dropout(0.3),
52             Conv2D(256, kernel_size=(3,3), activation='relu', padding='same'),
53             BatchNormalization(),
54             MaxPooling2D(pool_size=(2,2)), #80x4x4x3
55
56             Conv2D(512, kernel_size=(3,3), activation='relu', padding='same'), #160x4x4x3
57             BatchNormalization(), Dropout(0.3),
58             Conv2D(512, kernel_size=(3,3), activation='relu', padding='same'), #160x4x4x3
59             BatchNormalization(),
60             Dropout(0.3),
61             Conv2D(512, kernel_size=(3,3), activation='relu', padding='same'), #160x4x4x3
62             BatchNormalization(),
63             MaxPooling2D(pool_size=(2,2)), #160x2x2x3
64
65             Conv2D(512, kernel_size=(3,3), activation='relu', padding='same'), #160x2x2x3
66             BatchNormalization(), Dropout(0.3),
67             Conv2D(512, kernel_size=(3,3), activation='relu', padding='same'), #160x2x2x3
68             BatchNormalization(), Dropout(0.3),
69             Conv2D(512, kernel_size=(3,3), activation='relu', padding='same'), #160x2x2x3
70             BatchNormalization(),
71             MaxPooling2D(pool_size=(2,2)), #160x1x1x3
72
73             Flatten(),
74             Dense(512, activation='relu'),
75             BatchNormalization(), Dropout(0.3),
76
77             Dense(num_classes, activation='softmax')
78         ]
79     )
80
81     opt=optimizers.Adam()
82     model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
83
84     return model
85
86 model=create_model()
87 #from tensorflow.keras.utils import plot_model
```

```

88 #plot_model(model, to_file='vgg_prelayer.png', show_shapes=True); model.summary()
89
90 history=model.fit(ax, ay, batch_size=batch_size, epochs=epochs, verbose=2,
91                 validation_data=(qx, qy))
92 impHistoria(history)
93
94 score = model.evaluate(qx, qy, verbose=0)
95 print('Test loss:', score[0]); print('Test accuracy:', score[1])
96 #model.save('vgg4_prelayer_umaum.keras')

```

Programa 8: VGG com data augmentation usando preprocessing layers. Atinge acuracidade de teste de 91,4%.

[https://colab.research.google.com/drive/1IMff9yeZLkIH\\_LNMApxI\\_PEK74WY67Ri?usp=sharing](https://colab.research.google.com/drive/1IMff9yeZLkIH_LNMApxI_PEK74WY67Ri?usp=sharing)

Epoch 199/200 - 28s - loss: 0.0504 - accuracy: 0.9825 - val\_loss: 0.5356 - val\_accuracy: 0.9139  
Epoch 200/200 - 28s - loss: 0.0545 - accuracy: 0.9821 - val\_loss: 0.4202 - val\_accuracy: 0.9143

Test loss: 0.4201729893684387

Test accuracy: 0.9143000245094299

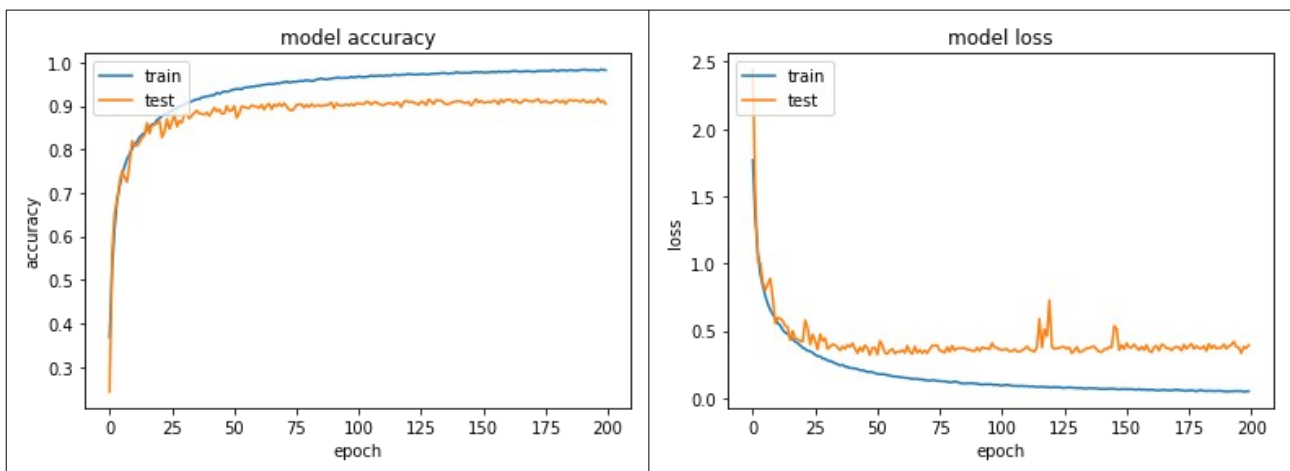


Figura:



*Exercício:* Adapte o programa 8 (vgg4\_prelayer.py) para classificar fashion\_mnist e obtenha acuracidade de teste de pelo menos 94%. Compare com a acuracidade usando LeNet.

*Nota 1:* Este programa demora para executar. Para não demorar excessivamente, vamos convencionar executar apenas 30 épocas. Mesmo assim, demora algo como 20 minutos. Leve isto em conta ao planejar fazer este exercício.

*Nota 2:* Obtive taxa de acerto de teste de 91% classificando fashion\_mnist com LeNet e 94% com VGG.

*Nota 3:* Você deve redimensionar as imagens  $28 \times 28 \times 1$  para  $32 \times 32 \times 1$ , pois 28 só é divisível por 2 duas vezes. Para isso, pode fazer zoom nas imagens ou inserir 2 linhas/colunas brancas/pretas nas quatro bordas das imagens, usando por exemplo o comando:

```
cv2.copyMakeBorder(src, top, bottom, left, right, borderType[, dst[, value ]]) → dst  
ax[i]=cv2.copyMakeBorder(AX[i],2,2,2,2,cv2.BORDER_CONSTANT,0)
```

*Nota 4:* Você deve pensar quais são as distorções permitidas. Provavelmente, não pode continuar fazendo espelhamento horizontal (todos os calçados estão voltados para o lado esquerdo). Possivelmente, deslocamento de 10% e rotação de 15 graus são exagerados.

A solução privada está em "Meu Drive/Colab Notebooks/algpi\_licao/cifar/vgg4\_fashion.ipynb" e atinge acuracidade de teste de 94%.  
[https://colab.research.google.com/drive/1assbYwqRwn1L1diclhKiPft7BiRXAg\\_Y](https://colab.research.google.com/drive/1assbYwqRwn1L1diclhKiPft7BiRXAg_Y)

*Exercício:* Modifique o programa 8 para obter programa aviao.py que distingue aviões dos navios. Para isso:

a) Coloque dentro dos tensores ax, ay, qx e qy somente as imagens de aviões e navios.

b) Modifique o programa 8 para distinguir aviões de navios. Grave os pesos treinados como aviao.h5.

Qual é a acuracidade obtida?

*Exercício:* Crie uma camada personalizada (custom layer) que execute normalização de lote sem  $\gamma$  e  $\beta$ . Use essa nova camada no programa 8 e verifique o desempenho.

**Resumo:** Nas aulas anteriores, começamos resolver a classificação Cifar-10. Usando LeNet (sem data augmentation), chegamos à taxa de acerto de 75%. Mesmo com data augmentation, a taxa de acerto não passou de 77%. Usando VGG (junto com batch normalization e data augmentation) chegamos à taxa de acerto de 91-92%. A ideia principal do VGG é usar sequência de convoluções 3×3, em vez de usar convoluções maiores. Hoje, estudaremos Inception v1 (GoogLeNet). Também estudaremos ResNet, com taxas de acerto de também 91%-92%. Usando ResNet com “test-time augmentation” (TTA), chegaremos a taxa de acerto de mais de 93,3%. Fazendo transfer learning de EfficientNetB0 (apostila transfer-eat), conseguiremos acuracidade de 96,4%.

## 6 Rede “inspirada em Inception v1” (GoogLeNet)

Inception v1 ou GoogLeNet [Szegedy2015] foi o campeão de ILSVRC 2014. Vamos ver as ideias principais deste modelo [Tsang2018, <https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvic-2014-image-classification-c2b3565a64e7>].

### 1) Convolução 1×1

À primeira vista, convolução 1×1 pode parecer inútil, pois a saída seria igual à entrada. Mas repare que convolução 1×1 não está sendo aplicada numa imagem 2D monocromática, mas num volume 3D (linhas × colunas × bandas). A convolução 1×1 tira média ponderada dos pixels nas diversas bandas (ou mapas de atributos, figuras F e G). Se a imagem de entrada for colorida com as bandas RGB, convolução 1×1 tira média ponderadas das 3 bandas pixel a pixel.

Convolução 1×1 pode ser utilizada como módulo que reduz dimensão, para reduzir a computação e o número de parâmetros da rede.

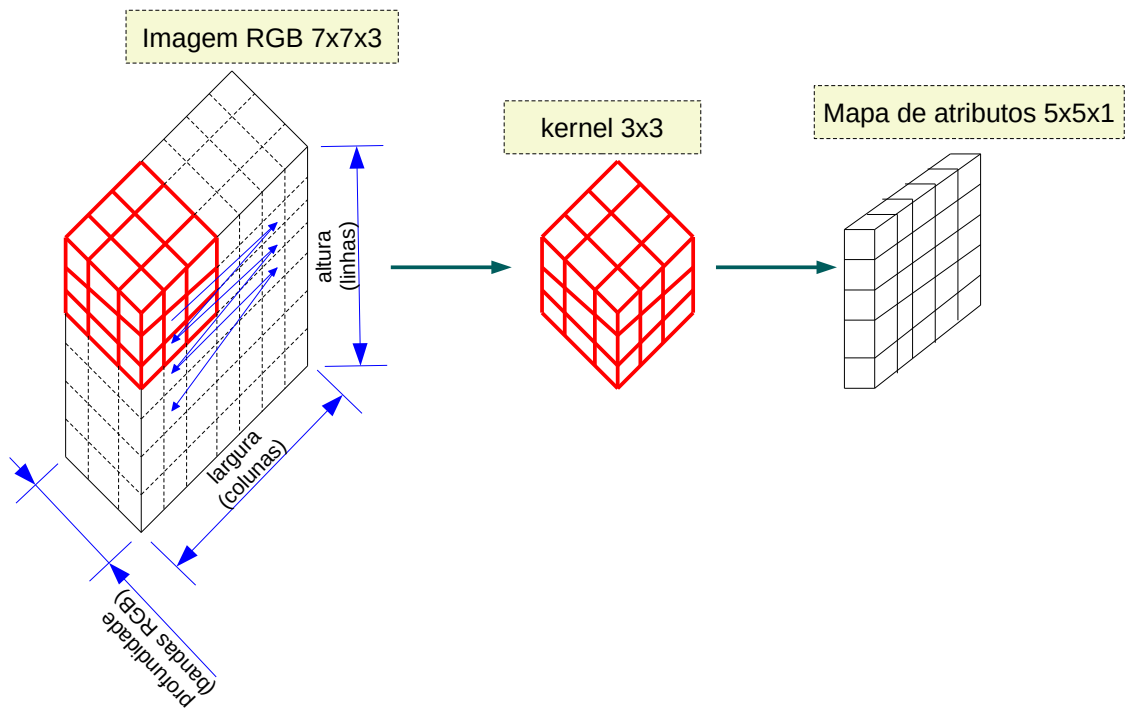


Figura F: Convolução 3x3 aplicada numa imagem RGB 7x7. A saída é um mapa de atributos 5x5x1.

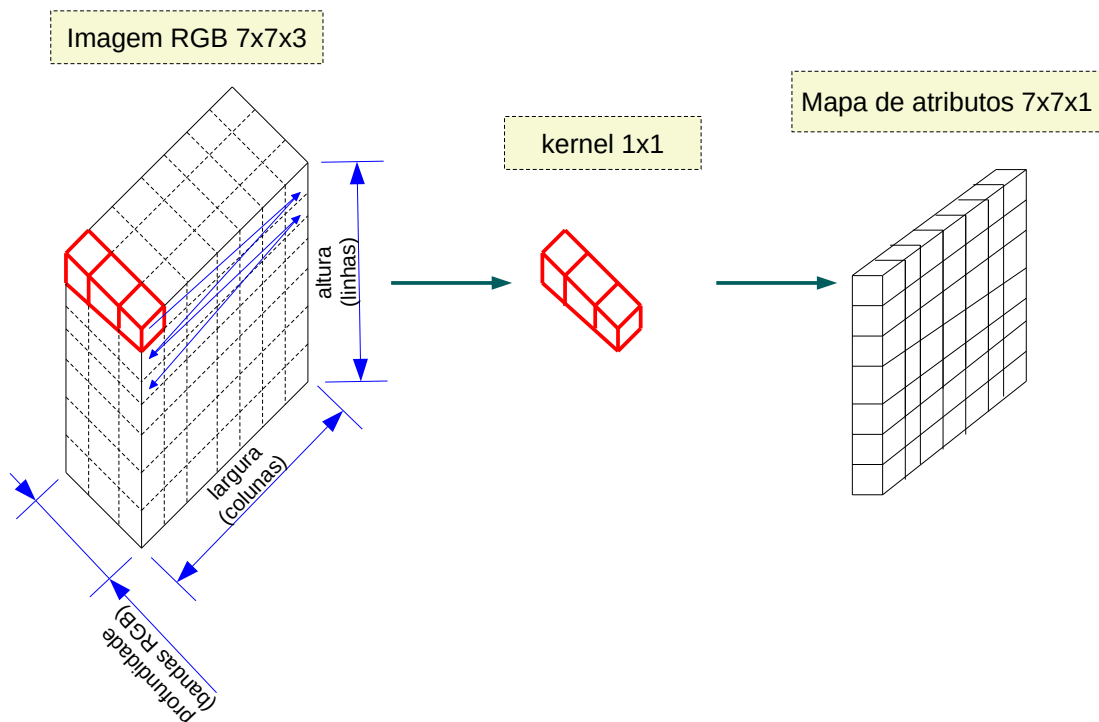


Figura G: Convolução 1x1 aplicada numa imagem RGB 7x7. A saída é um mapa de atributos 7x7x1.

A figura 10 mostra redução de 480 mapas de atributos  $14 \times 14$  para 48 mapas  $14 \times 14$  usando 48 convoluções  $5 \times 5$ . Fazendo redução direta, o número total de operações é  $(5 \times 5 \times 480) \times (14 \times 14 \times 48) = 112.9\text{M}$  (figura 10a).

Por outro lado, usando 16 convolução  $1 \times 1$  como passo intermediário (figura 10b):

Número de operações de 16 convoluções  $1 \times 1 = (1 \times 1 \times 480) \times (14 \times 14 \times 16) = 1.5\text{M}$

Número de operações de 48 convoluções  $5 \times 5 = (5 \times 5 \times 16) \times (14 \times 14 \times 48) = 3.8\text{M}$

Número total de operações =  $1.5\text{M} + 3.8\text{M} = 5.3\text{M}$

Claramente, 5M é muito menor que 113M.

O número de pesos da redução usando 48 convoluções  $5 \times 5$  é  $5 \times 5 \times 480 \times 48 = 576.000$  (figura 10a).

O número de pesos usando 16 convoluções  $1 \times 1$  e 48 convoluções  $5 \times 5$  é  $1 \times 1 \times 480 \times 16 + 5 \times 5 \times 16 \times 48 = 7.680 + 19.200 = 26.880$  (figura 10b). Claramente, 27k é muito menor que 576k.

Convolução  $1 \times 1$  pode ajudar a reduzir o número de pesos da rede, o tamanho do modelo, o número de operações e também overfitting.

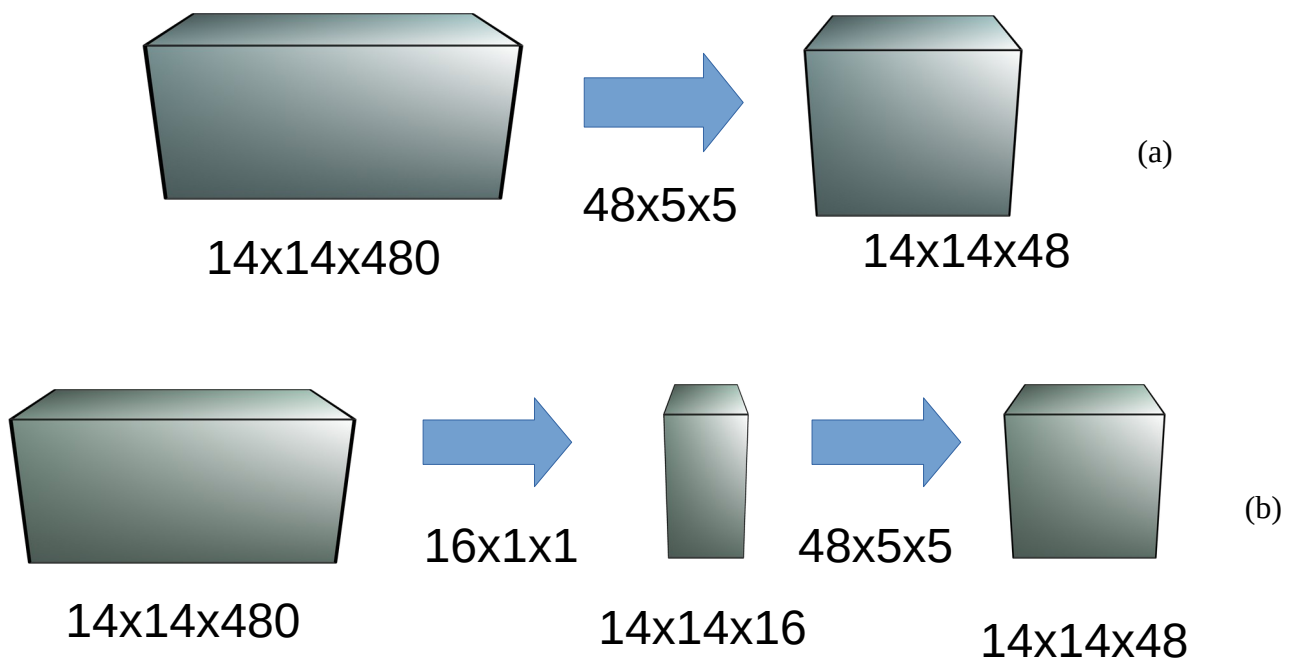


Figura 10: (a) Reduzir diretamente  $14 \times 14 \times 480$  atributos para  $14 \times 14 \times 48$  atributos usando 48 convoluções  $5 \times 5$  (113M operações e 576k pesos). (b) Usar 16 convoluções  $1 \times 1$  na redução (5M operações e 27k pesos).

## 2) Módulo Inception

A principal ideia de Inception consiste em aplicar convoluções de diferentes tamanhos em cada camada. A intuição atrás desta ideia é que há características que são mais facilmente detectáveis fazendo convolução de um determinado tamanho. Veja as figuras 11 e 12 abaixo.

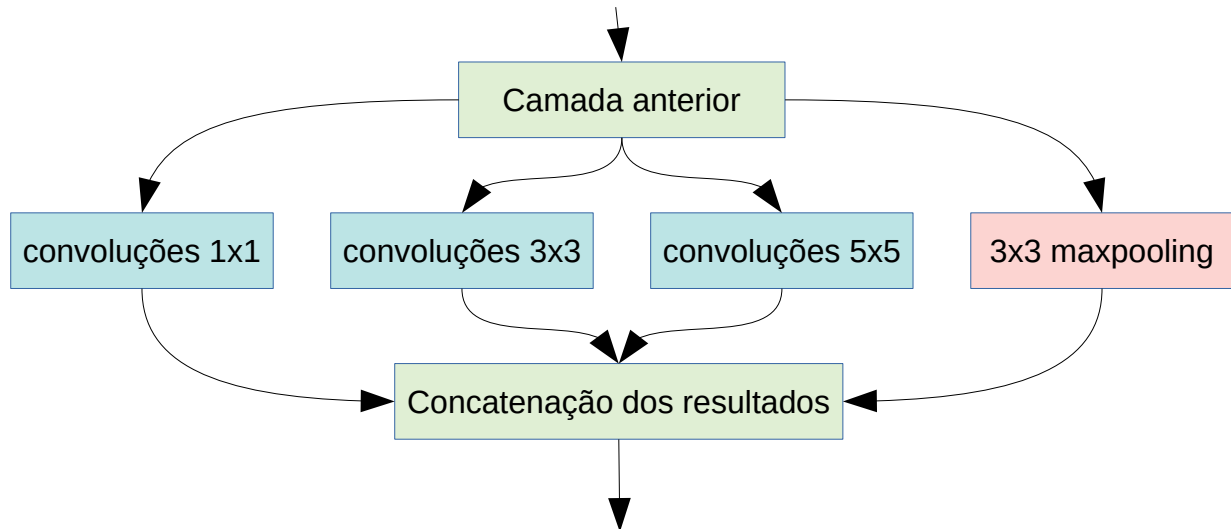


Figura 11: Módulo Inception “ingênuo”.

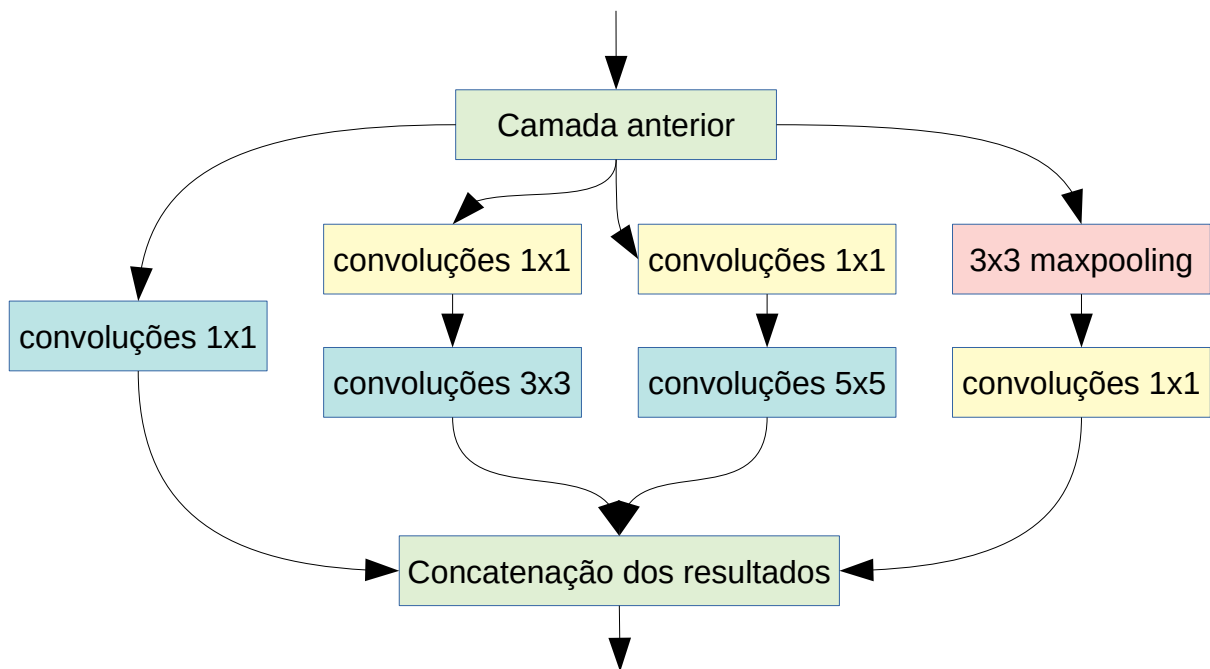


Figura 12: Módulo inception com redução de dimensionalidade (usando convoluções 1×1).

Um módulo Inception contém o seguinte número de convoluções:

- 1×1 com 64 saídas
- 3×3 com 128 saídas
- 5×5 com 32 saídas
- maxpooling 3×3 stride 1 com 32 saídas

Volume concatenado terá 64+128+32+32 mapas de atributos, ou seja, 256 mapas.

A novidade aqui está na descrição da rede em Keras. Não é possível descrever módulo Inception usando API (Application Programming Interface) sequencial, pois esse módulo não é sequencial: possui desvios e concatenações. Vamos usar a API funcional que permite descrever redes mais complexas. Além disso, vamos escrever uma função que define um módulo Inception, para poder replicá-lo facilmente várias vezes:

```
def moduloInception(nfiltros, x):
    tower_0 = Conv2D(nfiltros, (1,1), padding='same', activation='relu')(x) #conv2d_1
    tower_1 = Conv2D(2*nfiltros, (1,1), padding='same', activation='relu')(x) #conv2d_2
    tower_1 = Conv2D(2*nfiltros, (3,3), padding='same', activation='relu')(tower_1) #conv2d_3
    tower_2 = Conv2D(nfiltros//2, (1,1), padding='same', activation='relu')(x) #conv2d_4
    tower_2 = Conv2D(nfiltros//2, (5,5), padding='same', activation='relu')(tower_2) #conv2d_5
    tower_3 = MaxPooling2D((3,3), strides=(1,1), padding='same')(x) #max_pooling2d_1
    tower_3 = Conv2D(nfiltros//2, (1,1), padding='same', activation='relu')(tower_3) #conv2d_6
    x = keras.layers.concatenate([tower_0, tower_1, tower_2, tower_3], axis = 3)
    x = BatchNormalization()(x)
    return x
```

Desenhando as camadas:

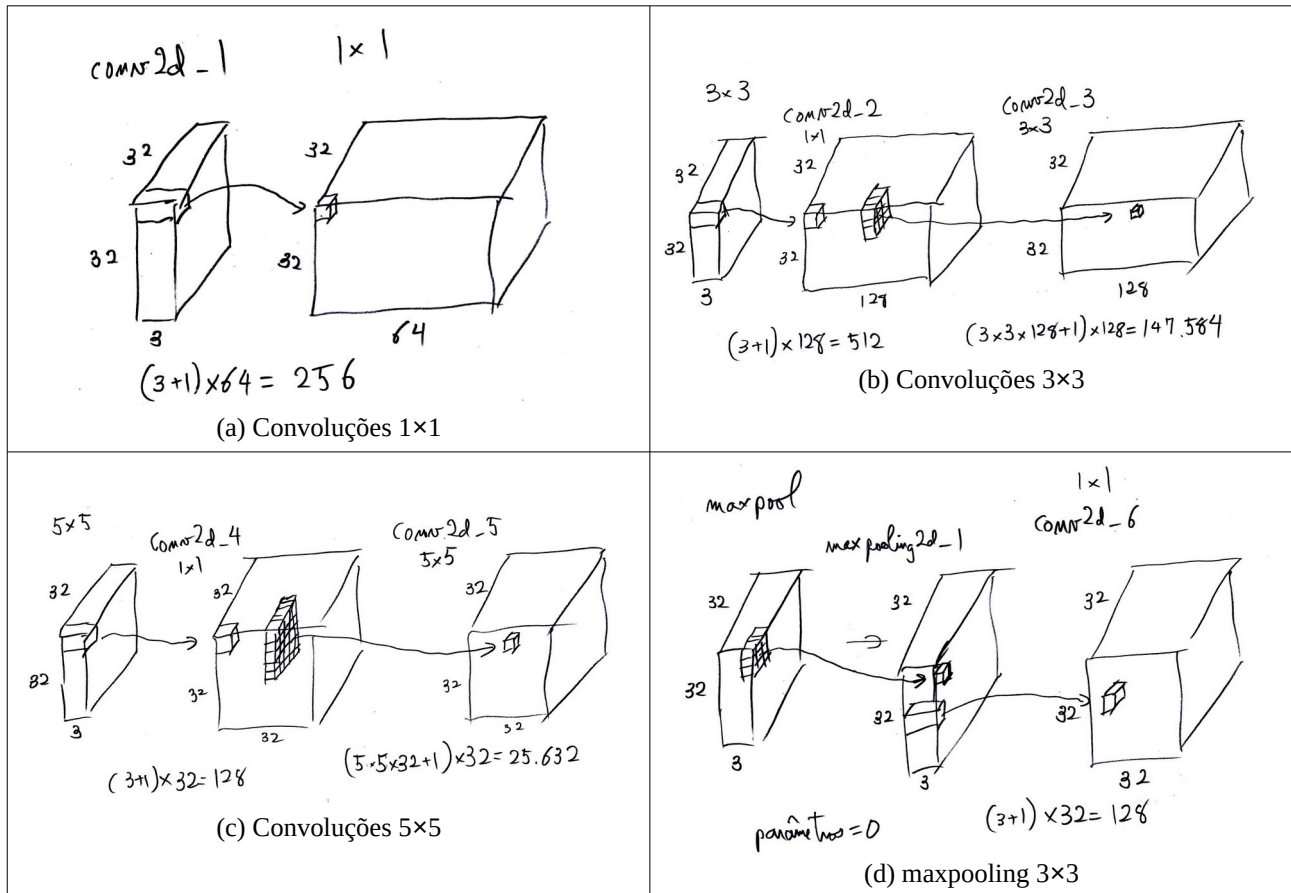


Figura 13: Diferentes convoluções de uma camada Inception.

3) As camadas de Inception são aplicadas em sequência, intercaladas por camadas maxpooling 2x2, para diminuir a resolução. Para classificar imagens de Cifar-10 (32x32x3), vamos usar 6 camadas Inception, intercaladas com 2 camadas maxpooling 2x2, diminuindo a resolução final para 8x8.

4) No final do processamento, teremos  $64+128+32+32=256$  mapas de atributos 8x8. Aqui, em vez de usar camadas densas para chegar à classificação final (como temos feito até agora), vamos calcular *global average pooling*, isto é, a média de cada mapa de atributo 8x8. A intuição de calcular *global average pooling* pode ser descrita como “para saber se um gato aparece na imagem, tire a média das probabilidades de gato aparecer em cada subregião da imagem”. Note que *global average pooling* joga fora a informação de local da imagem onde aparece o objeto de interesse (gato). *Global*

*average pooling* resulta em  $64+128+32+32=256$  atributos (números). Estes valores alimentam uma camada densa com 10 saídas.

```
output = AveragePooling2D(8)(x) ou output = GlobalAveragePooling2D()(x)
output = Flatten()(output)
outputs= Dense(10, activation='softmax')(output)
```

A quantidade de parâmetros da rede diminui substancialmente usando *global average pooling* (em vez de camadas densas), o que acelera o processamento. Além disso, *global average pooling* possibilita alimentar a rede com imagens maiores do que  $32 \times 32$ . Veja a propriedade das redes completamente convolucionais, na apostila *segment-ead*.

Rede “inspirada” em inception v1 para resolver Cifar-10 está no programa 9. Não é a Inception original.

Neste programa estamos usando dois novos “truques”:

- 1) Diminuimos *learning rate* toda vez que a acuracidade para de melhorar (trechos em amarelo).
- 2) Fazemos regularização L2 (trechos em verde – veja anexo E).

O resultado obtido, com ou sem esses dois truques, é semelhante ao de VGG: possui a acuracidade próxima de 92%. Também note que acuracidade de treino é 99.78%, indicando que a rede acerta praticamente todos os dados de treino e que há overfitting.



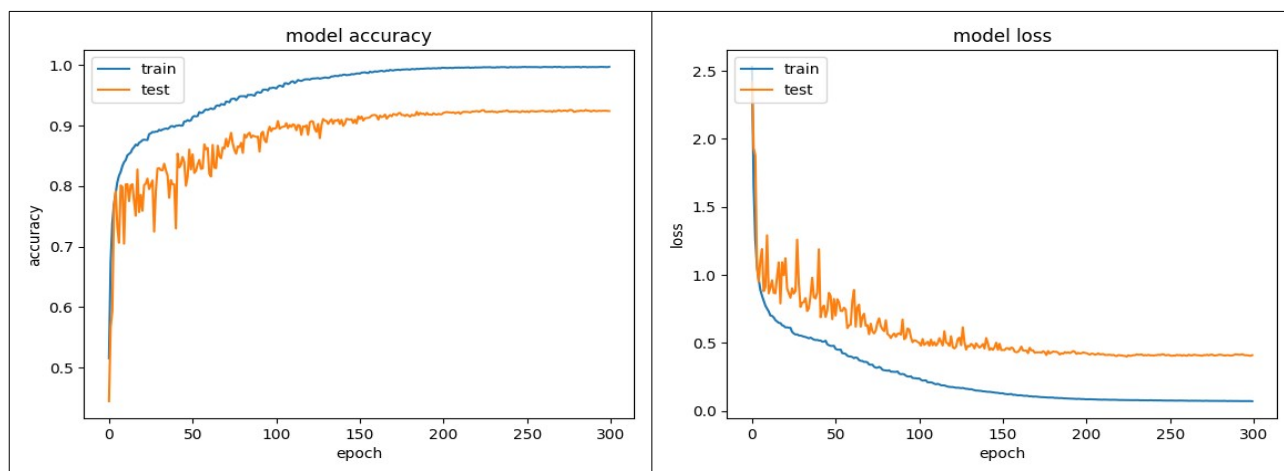
```

1 #inception_prelayer1.py
2 #Rede inspirada em inception para classificar CIFAR-10
3 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
4 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
5 import tensorflow.keras as keras
6 from tensorflow.keras.datasets import cifar10
7 from tensorflow.keras.models import Model, load_model
8 from tensorflow.keras.layers import Dense, Conv2D, BatchNormalization, Activation, Dropout
9 from tensorflow.keras.layers import AveragePooling2D, MaxPooling2D, Input, Flatten
10 from tensorflow.keras.regularizers import l2
11 from tensorflow.keras.optimizers import Adam
12 from tensorflow.keras.activations import relu
13 from tensorflow.keras.callbacks import ReduceLRonPlateau
14 from tensorflow.keras.layers import RandomFlip, RandomRotation, RandomZoom, RandomTranslation
15 from inspect import currentframe, getframeinfo
16 import numpy as np; import os
17 import matplotlib.pyplot as plt; import numpy as np
18
19 def impHistoria(history):
20     print(history.history.keys())
21     plt.plot(history.history['accuracy']); plt.plot(history.history['val_accuracy'])
22     plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
23     plt.legend(['train', 'test'], loc='upper left'); plt.show()
24     plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])
25     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
26     plt.legend(['train', 'test'], loc='upper left'); plt.show()
27
28 nomeprog="inception_prelayer1";
29 batch_size = 100; num_classes = 10; epochs = 300
30 nl, nc = 32,32; input_shape = (nl, nc, 3)
31 (ax, ay), (qx, qy) = cifar10.load_data()
32 ax = ax.astype('float32'); ax /= 255; ax -= 0.5 #-0.5 a +0.5
33 qx = qx.astype('float32'); qx /= 255; qx -= 0.5 #-0.5 a +0.5
34 ay = keras.utils.to_categorical(ay, num_classes)
35 qy = keras.utils.to_categorical(qy, num_classes)
36
37 def moduloInception(nfiltros, x):
38     kweight=5e-4; bweight=5e-4;
39     tower_0 = Conv2D(nfiltros, (1,1), padding='same', activation='relu',
40                     kernel_regularizer=l2(kweight),bias_regularizer=l2(bweight))(x) #conv2d_1
41     tower_1 = Conv2D(2*nfiltros, (1,1), padding='same', activation='relu',
42                     kernel_regularizer=l2(kweight),bias_regularizer=l2(bweight))(x) #conv2d_2
43     tower_1 = Conv2D(2*nfiltros, (3,3), padding='same', activation='relu',
44                     kernel_regularizer=l2(kweight),bias_regularizer=l2(bweight))(tower_1) #conv2d_3
45     tower_2 = Conv2D(nfiltros//2, (1,1), padding='same', activation='relu',
46                     kernel_regularizer=l2(kweight),bias_regularizer=l2(bweight))(x) #conv2d_4
47     tower_2 = Conv2D(nfiltros//2, (5,5), padding='same', activation='relu',
48                     kernel_regularizer=l2(kweight),bias_regularizer=l2(bweight))(tower_2) #conv2d_5
49     tower_3 = MaxPooling2D((3,3), strides=(1,1), padding='same')(x) #max_pooling2d_1
50     tower_3 = Conv2D(nfiltros//2, (1,1), padding='same', activation='relu',
51                     kernel_regularizer=l2(kweight),bias_regularizer=l2(bweight))(tower_3) #conv2d_6
52     x = keras.layers.concatenate([tower_0, tower_1, tower_2, tower_3], axis = 3)
53     # x=Dropout(0.3)(x)
54     x = BatchNormalization()(x)
55     return x
56
57 inputs = Input(shape=input_shape)
58 x = RandomRotation(0.042,fill_mode="nearest",interpolation="bilinear")(inputs) #15 graus: 15*pi/180
59 x = RandomTranslation(0.1, 0.1,fill_mode="nearest",interpolation="bilinear")(x)
60 x = RandomFlip("horizontal")(x)
61 x = moduloInception(64,x); x = moduloInception(64,x)
62 x = MaxPooling2D(2)(x); #16x16x(64+128+32+32)
63 x = moduloInception(64,x); x = moduloInception(64,x)
64 x = MaxPooling2D(2)(x); #8x8x(64+128+32+32)
65 x = moduloInception(64,x); x = moduloInception(64,x) #8x8x(64+128+32+32)
66 output = AveragePooling2D(8)(x) #1x1x(64+128+32+32)=256
67 output = Flatten()(output) #64+128+32+32=256
68 outputs= Dense(10, activation='softmax')(output)
69
70 #Pode escolher entre construir modelo novo ou continuar o treino de onde parou
71 model = Model(inputs=inputs, outputs=outputs)
72 #model = load_model(nomeprog+'.keras')
73
74 from tensorflow.keras.utils import plot_model
75 plot_model(model, to_file=nomeprog+'.png', show_shapes=True);
76 model.summary()
77
78 opt=Adam()
79 model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
80
81 reduce_lr = ReduceLRonPlateau(monitor='accuracy',
82                               factor=0.9, patience=2, min_lr=0.00001, verbose=True)
83 history=model.fit(ax, ay, batch_size=batch_size,
84                  epochs=epochs, verbose=2, validation_data=(qx, qy),
85                  callbacks=[reduce_lr])
86 impHistoria(history)
87
88 score = model.evaluate(qx, qy, verbose=0)
89 print('Test loss: %.4f'%(score[0]))
90 print('Test accuracy: %.2f %%%(100*score[1]))
91 print('Test error: %.2f %%%(100*(1-score[1]))
92 model.save(nomeprog+'.keras')
93

```

Programa 9: Rede inspirada em Inception usando preprocessing layer.  
[https://colab.research.google.com/drive/1nrCjgyHLM3QjU5uMBdNgBPddaKGM\\_Q-Y](https://colab.research.google.com/drive/1nrCjgyHLM3QjU5uMBdNgBPddaKGM_Q-Y)

Epoch 298/300 - 54s - accuracy: 0.9965 - loss: 0.0732 - val\_accuracy: 0.9244 - val\_loss: 0.4076 - learning\_rate: 1.0000e-05  
 Epoch 299/300 - 54s - accuracy: 0.9964 - loss: 0.0729 - val\_accuracy: 0.9240 - val\_loss: 0.4052 - learning\_rate: 1.0000e-05  
 Epoch 300/300 - 54s - accuracy: 0.9970 - loss: 0.0719 - val\_accuracy: 0.9238 - val\_loss: 0.4105 - learning\_rate: 1.0000e-05  
 Test loss: 0.4105  
**Test accuracy: 92.38 %**  
 Test error: 7.62 %



O modelo inception\_prelayer1.keras está em:

[https://drive.google.com/file/d/15blA1SMMJpWc\\_-FzM3Md5qjxmvntdYX7/view?usp=drive\\_link](https://drive.google.com/file/d/15blA1SMMJpWc_-FzM3Md5qjxmvntdYX7/view?usp=drive_link)

OU [http://www.lps.usp.br/hae/apostila/inception\\_prelayer1.keras](http://www.lps.usp.br/hae/apostila/inception_prelayer1.keras)

*Exercício:* Reescreva o programa 1 (regression.py) da aula “densakeras-ead” usando API funcional. Teste para verificar que está fazendo a mesma tarefa que fazia API sequencial.

*Exercício:* Reescreva o programa 2 (abc1.py) da aula “densakeras-ead” usando API funcional. Teste para verificar que está fazendo a mesma tarefa que fazia API sequencial.

*Exercício recomendado:* Reescreva o programa 7 (cnn1.py) da aula “convkeras-ead” utilizando API funcional. Teste para verificar que está fazendo a mesma tarefa que fazia API sequencial.

*Exercício:* Modifique inception1.py de forma que o novo programa, caogato3.py, consiga reconhecer cachorros e gatos do Cifar-10.

*Exercício:* Aumente a taxa de acerto do caogato3.py usando:

- Ensemble de 3 redes neurais (o resultado será a categoria com a maioria dos votos).
- Distorça cada imagem-teste de 3 formas diferentes (o resultado será a categoria com a maioria dos votos).
- Combine ensemble de 3 redes neurais com 3 distorções.

*Exercício:* Modifique inception1.py para que classifique fashion-mnist.

## 7 Rede “inspirada em ResNet”

ResNet ou Residual Networks [Zhang2016] ganhou ImageNet2015. VGG tinha 19 camadas convolucionais, enquanto que ResNet tem 152 camadas. ResNet melhorado tem 1001 camadas convolucionais. É muito difícil treinar redes tão profundas, pois o gradiente tende a desaparecer ou explodir em redes profundas. Se você multiplicar um número um pouco menor que 1 muitas vezes por ele mesmo, o resultado tende a zero (vanishing gradient). Se você multiplicar um número um pouco maior que 1 muitas vezes, o resultado tende a infinito (exploding gradient).

Para treinar redes tão profundas, ResNet utiliza a ideia de “conexão-identidade” (figura 14) juntamente com batch normalization e regularização L2 (anexo E). Figura 15 mostra VGG, uma rede muito profunda sem conexão identidade, e ResNet com conexão identidade.

A regularização L2 dá preferência às redes com pesos pequenos, fazendo a rede aprender  $F(x)$  “simples” (anexo E).

Digamos que a rede deva aprender uma certa função  $G(x)$ . Esta função pode ser escrita como  $G(x)=F(x)+x$ . Espera-se que  $G(x)$  seja próxima da função identidade  $G(x)\approx x$  e portanto  $F(x)$  consista de flutuações em torno da função constantemente nula. A ideia atrás da conexão-identidade é que é mais fácil aprender resíduo  $F(x)$  do que a função original  $G(x)$ .

Depois de aplicar várias camadas ResNet com conexões-identidade, o modelo calcula average-pooling, seguida por uma camada densa que faz a classificação final (10 categorias no caso de Cifar10).

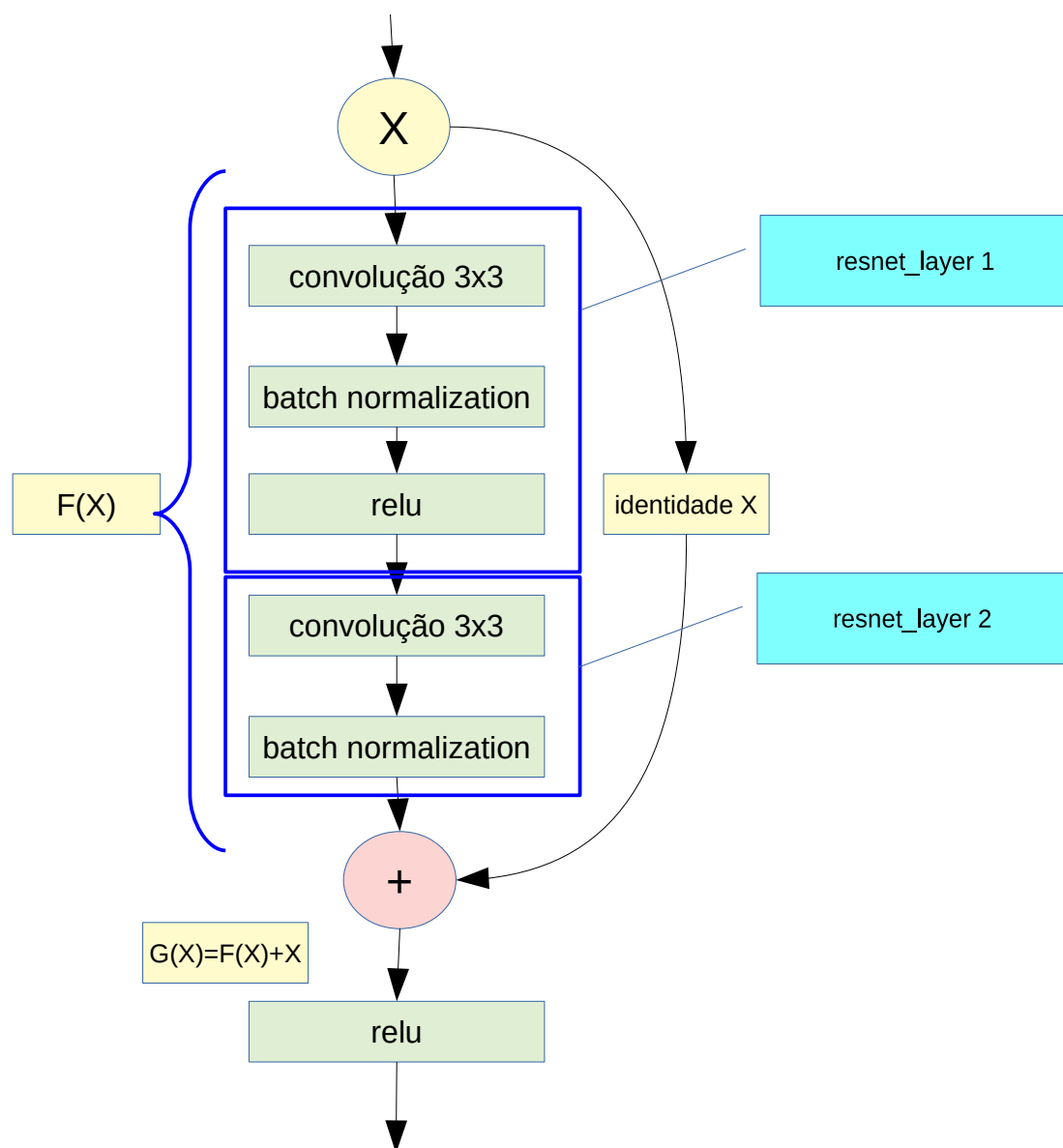


Figura 14: Camada de ResNet mostrando conexão-identidade.

Cada camada de ResNet (da figura 14) é implementada em Keras usando a camada ResNet (mostrada na figura A).

<pre>def resnet_layer(inputs, num_filters=16,                  kernel_size=3, strides=1,                  activation='relu',                  batch_normalization=True):     x = Conv2D(num_filters,                kernel_size=kernel_size,                strides=strides, padding='same',                kernel_initializer='he_normal',                kernel_regularizer=l2(1e-4))(inputs)     if batch_normalization:         x = BatchNormalization()(x)     if activation is not None:         x = Activation(activation)(x)     return x</pre>	<pre>y = resnet_layer(inputs=x,                  num_filters=num_filters) y = resnet_layer(inputs=y,                  num_filters=num_filters,                  activation=None) x = keras.layers.add([x, y]) x = Activation('relu')(x)</pre>
---	---

Figura A: Módulo ResNet

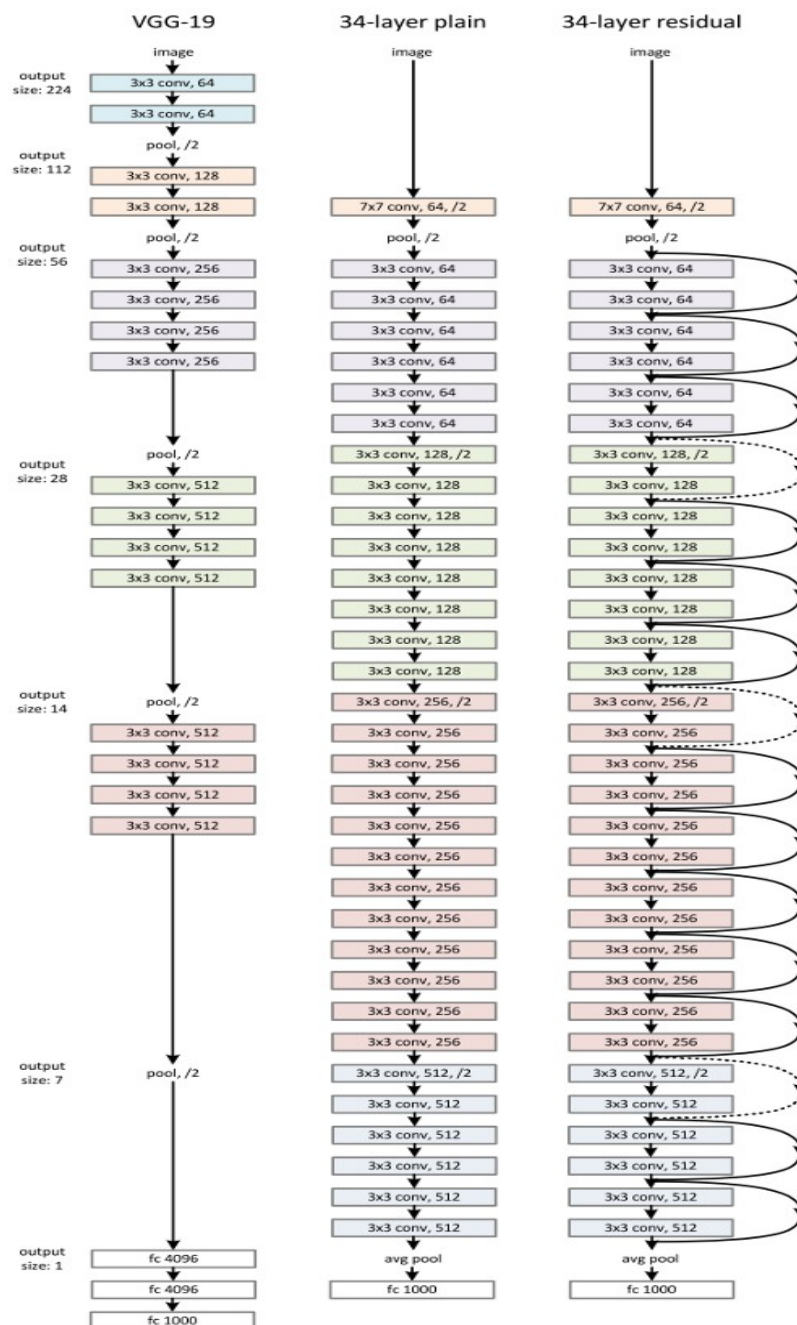


Figura 15: VGG, rede muito profunda e ResNet (figura do artigo original [He2015]).

Há um exemplo de Keras para resolução de Cifar10 usando ResNet:

[https://keras.io/zh/examples/cifar10\\_resnet/](https://keras.io/zh/examples/cifar10_resnet/)

Simplifiquei o programa para facilitar o entendimento, eliminando os loops e as condições que eram usados para criar a rede. A acuracidade de teste final é 91,24% (provavelmente melhoraria se deixasse treinando por mais tempo usando learning rate menor). Há uma melhora súbita de acuracidade na época 80, quando se diminui o learning rate.

```
1 #resnet_prelayer1.py
2 #Rede inspirada em resnet para classificar CIFAR-10
3 import os; os.environ['TF_CPP_MIN_LOG_LEVEL']='3'
4 os.environ['TF_FORCE_GPU_ALLOW_GROWTH'] = 'true'
5 import tensorflow.keras as keras
6 from tensorflow.keras.datasets import cifar10
7 from tensorflow.keras.models import Model, load_model
8 from tensorflow.keras.layers import Dense, Conv2D, BatchNormalization, Activation
9 from tensorflow.keras.layers import GlobalAveragePooling2D, Input, Flatten
10 from tensorflow.keras.regularizers import l2
11 from tensorflow.keras.optimizers import Adam
12 from tensorflow.keras.activations import relu
13 from tensorflow.keras.callbacks import ReduceLROnPlateau
14 from tensorflow.keras.layers import RandomFlip, RandomRotation, RandomZoom, RandomTranslation
15 from inspect import currentframe, getframeinfo
16 import numpy as np; import os; import sys
17 import matplotlib.pyplot as plt; import numpy as np
18 from tensorflow.keras.callbacks import LearningRateScheduler
19
20 def impHistoria(history):
21     print(history.history.keys())
22     plt.plot(history.history['accuracy']); plt.plot(history.history['val_accuracy'])
23     plt.title('model accuracy'); plt.ylabel('accuracy'); plt.xlabel('epoch')
24     plt.legend(['train', 'test'], loc='upper left'); plt.show()
25     plt.plot(history.history['loss']); plt.plot(history.history['val_loss'])
26     plt.title('model loss'); plt.ylabel('loss'); plt.xlabel('epoch')
27     plt.legend(['train', 'test'], loc='upper left'); plt.show()
28
29 def resnet_layer(inputs, num_filters=16, kernel_size=3,
30                 strides=1, activation='relu', batch_normalization=True):
31     x = Conv2D(num_filters, kernel_size=kernel_size, strides=strides,
32               padding='same', kernel_initializer='he_normal',
33               kernel_regularizer=l2(1e-4))(inputs)
34     if batch_normalization: x = BatchNormalization()(x)
35     if activation is not None: x = Activation(activation)(x)
36     return x
37
38 def lr_schedule(epoch):
39     lr = 1e-3
40     if epoch > 180: lr *= 0.5e-3
41     elif epoch > 160: lr *= 1e-3
42     elif epoch > 120: lr *= 1e-2
43     elif epoch > 80: lr *= 1e-1
44     print('Learning rate: ', lr)
45     return lr
46
47 nomeprog="resnet1_prelayer1"
48 batch_size = 32; num_classes = 10; epochs = 200
49 nl, nc = 32, 32
50 (ax, ay), (qx, qy) = cifar10.load_data()
51 input_shape = (nl, nc, 3)
52 ax = ax.astype('float32'); ax /= 255 #0 a 1
53 qx = qx.astype('float32'); qx /= 255 #0 a 1
54 ax -= 0.5; qx -= 0.5 #-0.5 a +0.5
55 ay = keras.utils.to_categorical(ay, num_classes);
56 qy = keras.utils.to_categorical(qy, num_classes)
57
58 inputs = Input(shape=input_shape)
59 x = RandomRotation(0.042, fill_mode="nearest", interpolation="bilinear")(inputs) #Programa original nao faz rotacao
60 x = RandomTranslation(0.1, 0.1, fill_mode="nearest", interpolation="bilinear")(x)
61 x = RandomFlip("horizontal")(x)
62
63 x = resnet_layer(inputs=x)
64
65 num_filters = 16
66 y = resnet_layer(inputs=x, num_filters=num_filters, strides=1)
67 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
68 x = keras.layers.add([x, y]); x = Activation('relu')(x)
69
70 y = resnet_layer(inputs=x, num_filters=num_filters, strides=1)
71 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
72 x = keras.layers.add([x, y]); x = Activation('relu')(x)
73
74 y = resnet_layer(inputs=x, num_filters=num_filters, strides=1)
75 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
76 x = keras.layers.add([x, y]); x = Activation('relu')(x)
77
78 num_filters *= 2
79 y = resnet_layer(inputs=x, num_filters=num_filters, strides=2)
80 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
81 x = resnet_layer(inputs=x, num_filters=num_filters, kernel_size=1,
82                 strides=2, activation=None, batch_normalization=False)
83 x = keras.layers.add([x, y]); x = Activation('relu')(x)
84
85 y = resnet_layer(inputs=x, num_filters=num_filters, strides=1)
86 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
87 x = keras.layers.add([x, y]); x = Activation('relu')(x)
88
89 y = resnet_layer(inputs=x, num_filters=num_filters, strides=1)
90 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
91 x = keras.layers.add([x, y]); x = Activation('relu')(x)
92
93 num_filters *= 2
```

```

94 y = resnet_layer(inputs=x, num_filters=num_filters, strides=2)
95
96 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
97 x = resnet_layer(inputs=x, num_filters=num_filters, kernel_size=1,
98                 strides=2, activation=None, batch_normalization=False)
99 x = keras.layers.add([x, y]); x = Activation('relu')(x)
100
101 y = resnet_layer(inputs=x, num_filters=num_filters, strides=1)
102 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
103 x = keras.layers.add([x, y]); x = Activation('relu')(x)
104
105 y = resnet_layer(inputs=x, num_filters=num_filters, strides=1)
106 y = resnet_layer(inputs=y, num_filters=num_filters, activation=None)
107 x = keras.layers.add([x, y]); x = Activation('relu')(x)
108
109 x = GlobalAveragePooling2D()(x)
110 y = Flatten()(x)
111 outputs = Dense(num_classes, activation='softmax', kernel_initializer='he_normal')(y)
112
113 #Pode escolher entre construir modelo novo ou continuar o treino de onde parou
114 model = Model(inputs=inputs, outputs=outputs)
115 #model = load_model(nomeprog+'.keras')
116
117 model.summary()
118 # from tensorflow.keras.utils import plot_model
119 # plot_model(model, to_file=nomeprog+'.png', show_shapes=True)
120
121 opt=Adam()
122 model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
123
124 lr_scheduler = LearningRateScheduler(lr_schedule)
125 lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1), cooldown=0, patience=5, min_lr=0.5e-6)
126 callbacks = [lr_reducer, lr_scheduler]
127
128 history=model.fit(ax, ay, batch_size=batch_size,
129                 epochs=epochs, verbose=2, validation_data=(qx, qy),
130                 callbacks=callbacks)
131 impHistoria(history)
132
133 score = model.evaluate(qx, qy, verbose=0)
134 print('Test loss: %.4f'%(score[0]))
135 print('Test accuracy: %.2f %%%%(100*score[1]))
136 print('Test error: %.2f %%%%(100*(1-score[1]))
137 model.save(nomeprog+'.keras')

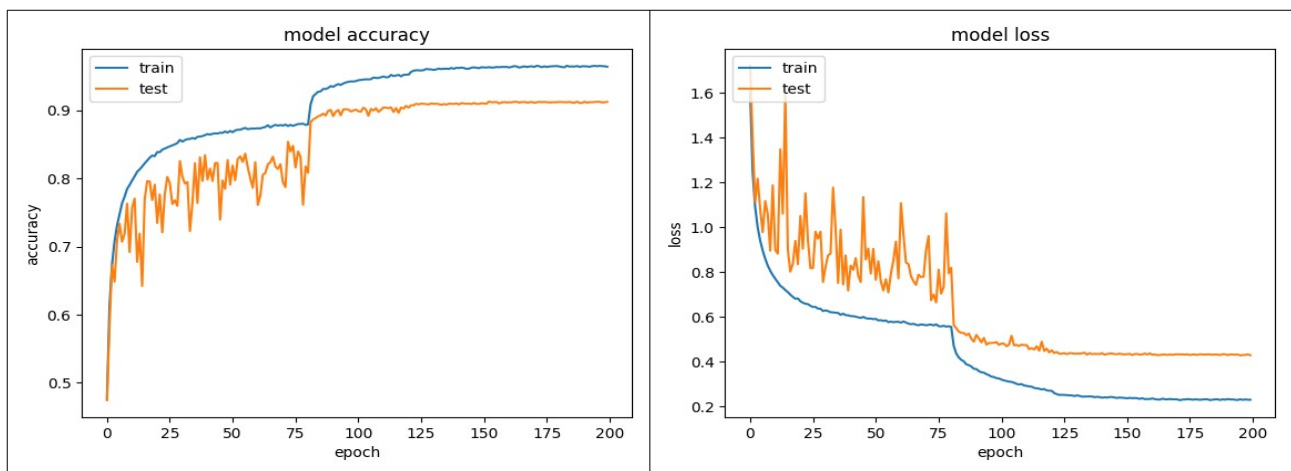
```

## Programa 10: ResNet simplificado com data augmentation. Atinge acuracidade de 91,24%.

Programa está em: <https://colab.research.google.com/drive/1uIX2uYfiOGZKUIG5fVjnPpFmMeUlhQmd#scrollTo=3CzP-xmHHqh5>

Rede treinada resnet\_prelayer1.keras está em: [https://drive.google.com/file/d/1CgswjoBi8Kw-uirRpNzGTxuOvyYiKifp/view?usp=drive\\_link](https://drive.google.com/file/d/1CgswjoBi8Kw-uirRpNzGTxuOvyYiKifp/view?usp=drive_link)

OU [http://www.lps.usp.br/hae/apostila/resnet\\_prelayer1.keras](http://www.lps.usp.br/hae/apostila/resnet_prelayer1.keras)



Epoch 198/200 - 41s - accuracy: 0.9649 - loss: 0.2299 - val\_accuracy: 0.9116 - val\_loss: 0.4287 - learning\_rate: 5.0000e-07  
Epoch 199/200 - 41s - accuracy: 0.9648 - loss: 0.2287 - val\_accuracy: 0.9119 - val\_loss: 0.4313 - learning\_rate: 5.0000e-07  
Epoch 200/200 - 41s - accuracy: 0.9639 - loss: 0.2287 - val\_accuracy: 0.9124 - val\_loss: 0.4277 - learning\_rate: 5.0000e-07  
Test loss: 0.4277  
Test accuracy: 91.24 %  
Test error: 8.76 %



## 8. Test-time augmentation (TTA) e ensemble

Digamos que você acertou os hiperparâmetros de um modelo para o seu problema da melhor forma possível, atingindo a maior acuracidade observada nos testes. É possível melhorar ainda mais a taxa de acerto? Existem três formas de aumentar a acuracidade da predição, sem ter de melhorar a acuracidade do modelo:

1. *Ensemble*: Treinar  $n$  redes neurais com estruturas iguais mas inicializadas com pesos diferentes (isto é, executar o treino  $n$  vezes obtendo  $n$  modelos diferentes). Depois, dada uma imagem a classificar QX, faz predição de QX com as  $n$  redes. Por fim, escolhe a classe de maior probabilidade média ou aquela com mais votos. Uma alternativa é treinar  $n$  redes neurais com estruturas diferentes (por exemplo, usar VGG, Inception e ResNet) – costuma dar resultado melhor do que usar redes com estruturas iguais.
2. *Test-time augmentation (TTA)*: Distorcer cada imagem teste QX de  $m$  formas diferentes (da mesma forma que distorcemos as imagens de treino durante *data augmentation*). Faz predição das  $m$  versões distorcidas de QX e escolhe a classe de maior probabilidade média ou aquela com mais votos. É o chamado “test-time augmentation”.
3. Combinar as estratégias (1 - ensemble) e (2 - TTA) e fazer  $n \times m$  predições e fazer a classificação final baseada nessas  $n \times m$  predições.

Como as estratégias (1) e (3) são demoradas (pois tem que treinar  $n$  redes diferentes), vamos testar aqui somente a estratégia (2) TTA (como exercício ou como lição de casa).

**Exercício:** Usando a rede `resnet_prelayer1.keras` (91,24% de acuracidade de teste), distorça cada uma das 10.000 imagens de teste de Cifar10 de 11 formas diferentes, faça predições das 11 imagens distorcidas e escolha a classe com a maior probabilidade média. A acuracidade aumentou?

*Nota:* As camadas de preprocessing layer não funcionam durante a predição. Tive que criar uma pequena rede somente com preprocessing layers e distorcer as imagens usando essa pequena rede.

*Nota:* A rede treinada está salva como `resnet_prelayer1.keras`

[https://drive.google.com/file/d/1CgswjoBi8Kw-uiNRpNzGTxuOvvYiKifp/view?usp=drive\\_link](https://drive.google.com/file/d/1CgswjoBi8Kw-uiNRpNzGTxuOvvYiKifp/view?usp=drive_link)

OU [http://www.lps.usp.br/hae/apostila/resnet\\_prelayer1.keras](http://www.lps.usp.br/hae/apostila/resnet_prelayer1.keras)

*Nota:* Obtive acuracidade de 93.25% fazendo TTA com 11 distorções.

*Cuidado:* O programa 10 (`resnet_prelayer1.py`) trabalha com os valores das imagens de entrada no intervalo de -0,5 a +0,5.

[Solução privada em <https://colab.research.google.com/drive/1l-s9aRnYZWB9zBtX3fFaUiKfNoyQryfq#scrollTo=3CzP-xmHHqh5> ]

*Exercício:* Modifique resnet1.py de forma que o novo programa caogato2.py consiga reconhecer cachorros e gatos. Para isso:

- a) Coloque dentro dos tensores ax, ay, qx e qy somente as imagens de cachorro (categoria original 5, nova categoria 0) e gato (categoria original 3, nova categoria 1).
- b) Modifique o programa para distinguir cachorros dos gatos.

*Exercício:* Modifique resnet1.py para atingir acuracidade maior que 92%. Anote claramente quais foram as alterações feitas e qual foi a acuracidade obtida.

*Exercício:* Adapte um dos modelos desta apostila (VGG, Inception ou Resnet) para classificar fashion-MNIST. Procure obter uma taxa de acerto alta – deixe claro a taxa obtida.

*Nota 1:* Obtive taxa de acerto de teste de 94% com VGG. Recorde de acerto é 96,91%, segundo Wikipedia.

[https://en.wikipedia.org/wiki/Fashion\\_MNIST](https://en.wikipedia.org/wiki/Fashion_MNIST)

*Nota 2:* Pode ser necessário redimensionar as imagens do fashion-MNIST de  $28 \times 28 \times 1$  para  $32 \times 32 \times 1$ , pois 28 só é divisível por 2 duas vezes. Para isso, pode fazer zoom nas imagens ou inserir 2 linhas/colunas de uma certa cor nas quatro bordas das imagens, por exemplo usando o comando:

```
cv2.copyMakeBorder(src, top, bottom, left, right, borderType[, dst[, value ] ]) → dst  
ax[i]=cv2.copyMakeBorder(AX[i],2,2,2,2,cv2.BORDER_CONSTANT,255)
```

## 9. EfficientNet

Neste momento (2022), as arquiteturas EfficientNet [https://arxiv.org/pdf/1905.11946.pdf, Tan2019], EfficientNet V2, e ConvNeXt [Liu2022] são consideradas o estado da arte em classificação de imagens usando CNN, que atingem alta acuracidade usando poucos parâmetros.

<https://datamonje.medium.com/image-classification-with-efficientnet-better-performance-with-computational-efficiency-f480fdb00ac6>

<https://paperswithcode.com/sota/image-classification-on-imagenet>

[https://keras.io/examples/vision/image\\_classification\\_efficientnet\\_fine\\_tuning/](https://keras.io/examples/vision/image_classification_efficientnet_fine_tuning/)

<https://keras.io/api/applications/efficientnet/>

EfficientNet obtém taxas de acerto “estado da arte” usando menos parâmetros do que outros modelos de redes. “It’s incredible that EfficientNet-B1 is 7.6× smaller and 5.7× faster than ResNet-152.” [https://theaisummer.com/cnn-architectures/ ]

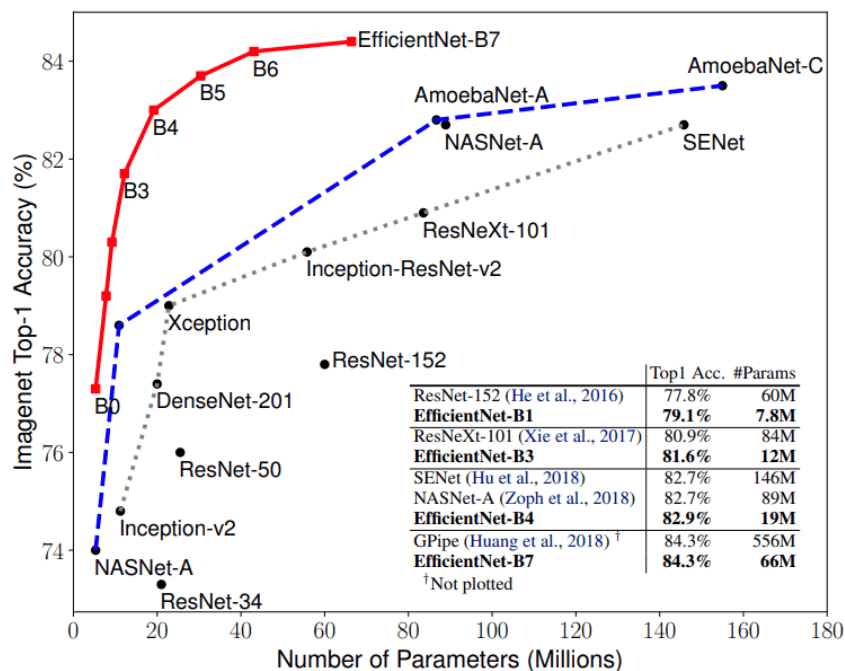


Figura F: Retirado de <https://arxiv.org/abs/1905.11946>

Tabela T: As 8 versões de EfficientNet original.

<https://keras.io/api/applications/> [https://keras.io/examples/vision/image\\_classification\\_efficientnet\\_fine\\_tuning/](https://keras.io/examples/vision/image_classification_efficientnet_fine_tuning/)

Base model	Input resolution	Output 7×7 maps	Acuracidade top-1
EfficientNetB0	224×224	1280	77.1%
EfficientNetB1	240	?	79.1%
EfficientNetB2	260	?	80.1%
EfficientNetB3	300	?	81.6%
EfficientNetB4	380	1792	82.9%
EfficientNetB5	456	?	83.6%
EfficientNetB6	528	?	84.0%
EfficientNetB7	600×600	2560	84.3%

As versões de *EfficientNet*: EfficientNet original possui 8 versões, numeradas de EfficientNet-B0 até B7 (tabela T). A versão B0 é o menor modelo, com quantidade pequena de parâmetros e a menor taxa de acerto. Recebe uma imagem colorida  $224 \times 224 \times 3$  e produz 1280 mapas de atributos  $7 \times 7$ . Estes mapas passam por average-pooling e uma camada densa, para classificar a imagem nas 1000 categorias da ImageNet.

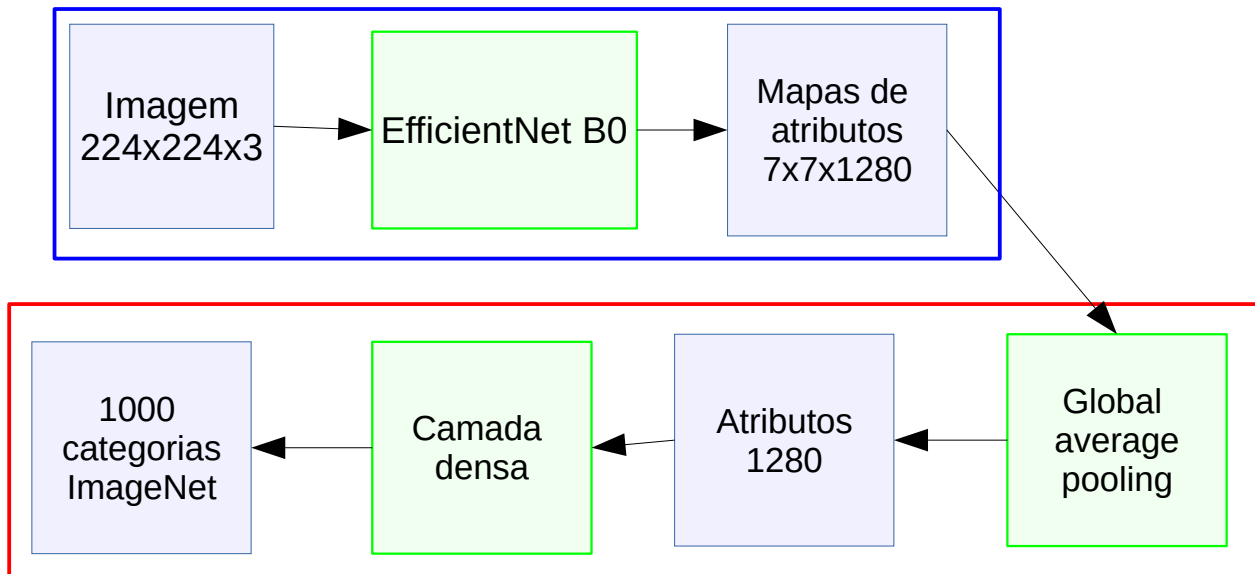


Figura: Modelo base de EfficientNetB0 em azul e as camadas de topo em vermelho.

É possível fazer transfer learning de EfficientNet-B0 retirando as camadas de average pooling e camada densa finais, para obter 1280 mapas de atributos de  $7 \times 7$  que podem ser usados em muitas aplicações.

EfficientNet-B0 foi projetado automaticamente por “neural architecture search” que procurou o modelo que melhor atendia as especificações fornecidas pelos autores usando um número mínimo de parâmetros. Depois, esta rede pequena foi redimensionada para chegar a outras 7 versões.

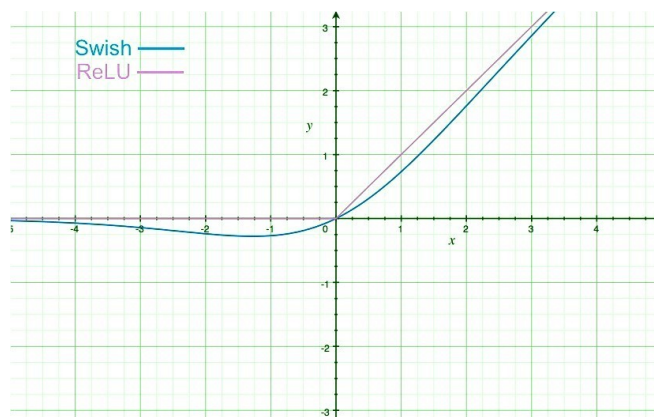
*Compound scaling*: A ideia principal do artigo [Tan2019] é como redimensionar EfficientNetB0 para chegar a outros 7 modelos maiores. Temos visto até agora que a acuracidade de CNN melhora quando se aumenta o tamanho da rede colocando mais camadas, usando mais mapas de atributos ou aumentando a resolução da imagem de entrada. Só que os melhores parâmetros de largura (width – número de mapas de atributos), profundidade (depth – quantidade de camadas) e resolução (resolution – tamanho da imagem de entrada e dos mapas de atributos intermediários) tinham que ser procurados manualmente para cada nova versão do modelo.

O que artigo propõe é um método para aumentar o tamanho da rede de forma eficiente e otimizada em largura, profundidade e resolução, procurando minimizar o aumento no número de parâmetros. Ao contrário da prática convencional que dimensiona arbitrariamente esses fatores, o método de redimensionamento EfficientNet dimensiona uniformemente a largura, a profundidade e a resolução da rede com um conjunto de coeficientes de dimensionamento fixos.

Nota: Se quisermos usar  $2^N$  vezes mais recursos computacionais, podemos aumentar a profundidade da rede em  $\alpha^N$ , a largura em  $\beta^N$ , e o tamanho da imagem em  $\gamma^N$ , onde  $\alpha$ ,  $\beta$  e  $\gamma$  são coeficientes constantes determinados por uma pesquisa de grade no modelo pequeno original. [<https://paperswithcode.com/method/efficientnet> ]. O artigo original escolheu:  $\alpha=1.2$ ,  $\beta=1.1$  e  $\gamma=1.15$ , sendo  $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$ .

**Swish activation (ou Silu):** EfficientNet utiliza Swish activation em vez de ReLU. Parece que trocar ReLU por Swish em alguns modelos aumenta em 0.6% a taxa de acerto em ImageNet.

$$\text{Swish}(x) = x \cdot \text{sigmoid}(x)$$



**GELU activation:**

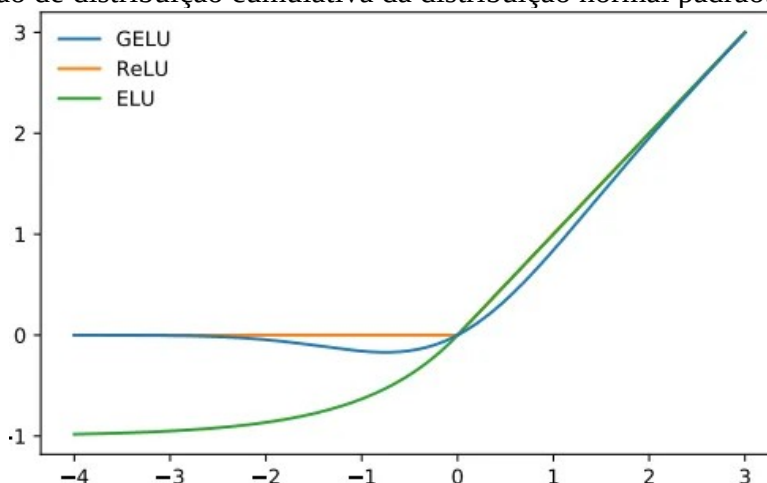
<https://www.baeldung.com/cs/gelu-activation-function>

<https://medium.com/@shauryagoel/gelu-gaussian-error-linear-unit-4ec59fb2e47c>

GELU (Gaussian Error Linear Unit) é outra função de ativação popular.

$$\text{GELU}(x) = x \times \text{CDF}(x)$$

onde CDF é a função de distribuição cumulativa da distribuição normal padrão.



GELU pode ser aproximada usando:

$$\text{GELU}(x) \approx x \times \text{sigmoide}(1.702 \times x)$$

Tanto Swish como GELU estão implementadas em Keras;

**Exercício interessante:** Troque ReLU por Swish ou GELU em numa das 3 redes com taxa de acerto 92% em Cifar-10 vistas nesta apostila: VGG (programa 8a ou 8b), Inception (programa 9) e ResNet (programa 10). Verifique se a taxa de acerto aumenta.

**Exercício interessante:** Trocando relu por swish na rede que classifica MNIST e verifique se a taxa de acerto aumenta.

*Convolução em profundidade (depth-wise convolution):* Convolução em profundidade calcula convoluções independentes em cada banda de cor (ou em cada canal do mapa de atributos). Os sites abaixo explicam bem a diferença entre convolução “normal” e “depth-wise”.

<https://medium.com/@zurister/depth-wise-convolution-and-depth-wise-separable-convolution-37346565d4ec>

<https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/>

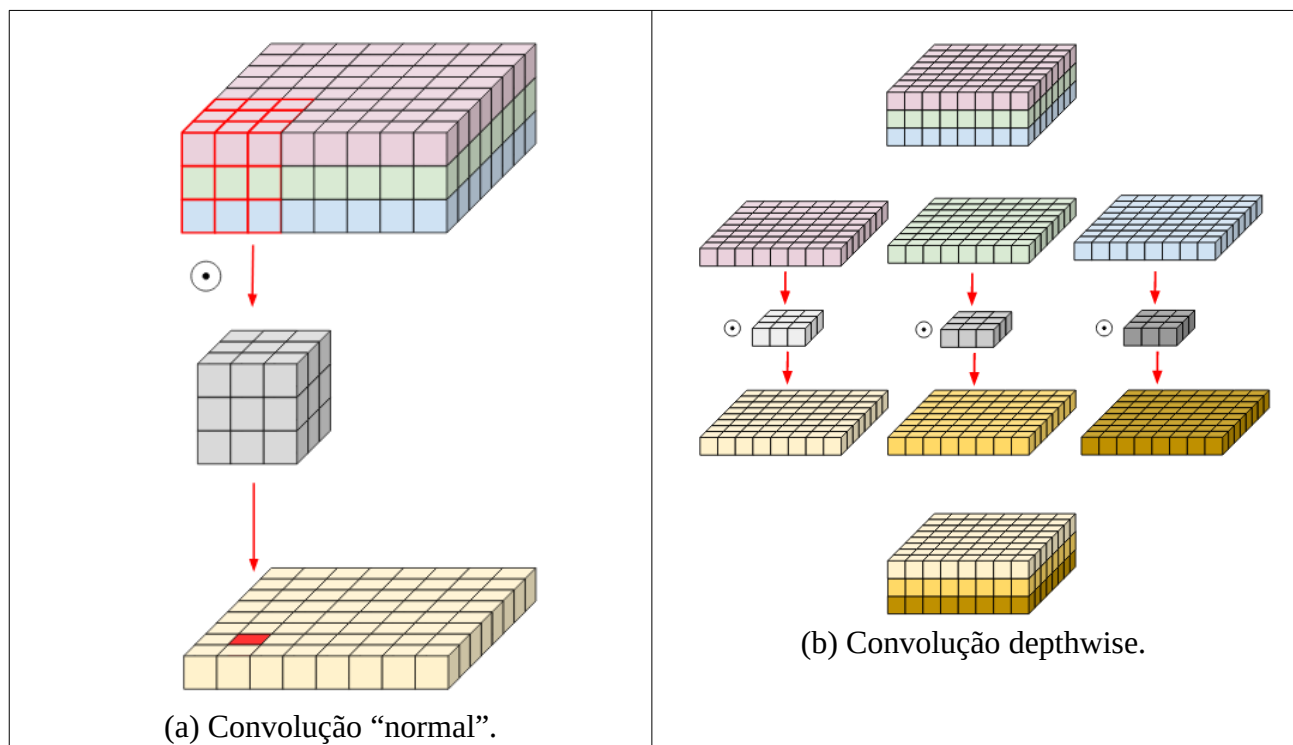


Figura F: Convoluções normal e depthwise.

[<https://eli.thegreenplace.net/2018/depthwise-separable-convolutions-for-machine-learning/> ]

*Bloco residual invertido:* Um bloco residual invertido (figura B) consiste em:

- (1) Expandir os mapas de ativação de entrada usando convoluções  $1 \times 1$  para aumentar a profundidade dos mapas de atributos.
- (2) Efetuar convoluções de profundidade (depthwise)  $3 \times 3$ .
- (3) Efetuar convoluções  $1 \times 1$  (point-wise) que reduzem o número de canais no mapa de atributos.
- (4) Usar uma conexão identidade liga entrada à saída (como no ResNet).

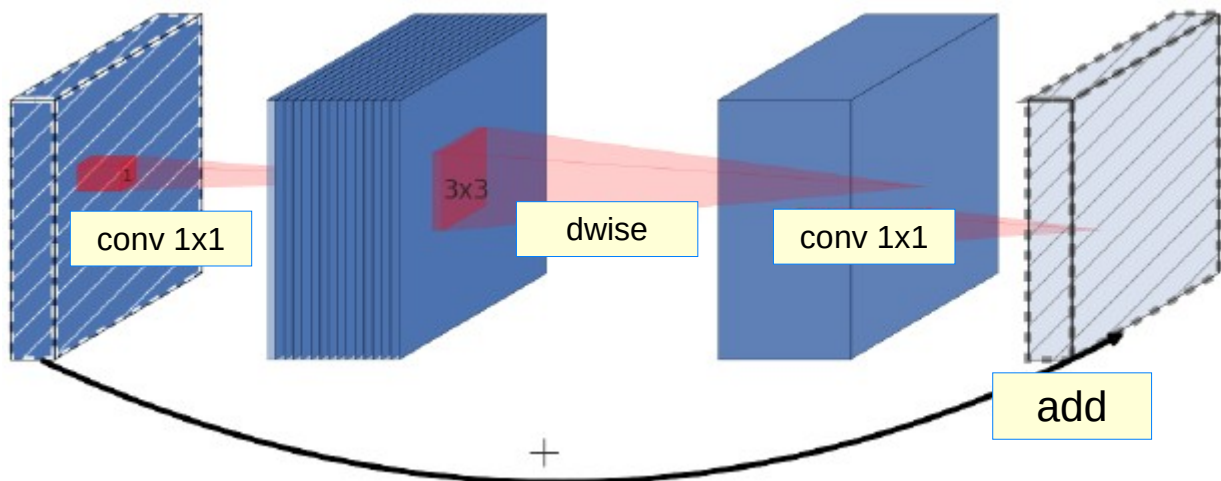


Figura B: Bloco residual invertido.

<https://datamonje.medium.com/image-classification-with-efficientnet-better-performance-with-computational-efficiency-f480fdb00ac6>

O código Keras deste bloco é:

```
from keras.layers import Conv2D, DepthwiseConv2D, Add
def inverted_residual_block(x, expand=64, squeeze=16):
    block = Conv2D(expand, (1,1), activation='relu')(x)
    block = DepthwiseConv2D((3,3), activation='relu')(block)
    block = Conv2D(squeeze, (1,1), activation='relu')(block)
    return Add()([block, x])
```

## Squeeze and excitation (SE Nets)

<https://towardsdatascience.com/squeeze-and-excitation-networks-9ef5e71eacd7>

Squeeze and excitation consegue aumentar substancialmente o desempenho de praticamente qualquer rede. A ideia é simples: calcular o peso de cada canal de um bloco convolucional para que a rede possa ajustar de forma adaptativa a ponderação de cada canal no mapa de atributos.

Para calcular os pesos dos canais, primeiro calcula-se global average pooling do tensor de entrada, resultando num vetor com  $ch$  elementos. Este vetor passa pela camada densa com  $ch/ratio$  neurônios (é a etapa de apertar – *squeeze*). Depois, passa por uma segunda camada densa com  $ch$  neurônios. As saídas desta camada passarão pela ativação sigmoide, para resultar em  $ch$  números entre 0 e 1 que serão os pesos que multiplicarão cada canal

```
def se_block(in_block, ch, ratio=16):  
    x = GlobalAveragePooling2D()(in_block)  
    x = Dense(ch//ratio, activation='relu')(x) # ch//ratio pode resultar zero e causar erro.  
    x = Dense(ch, activation='sigmoid')(x)  
    return multiply([in_block, x]) # in_block e x nao tem o mesmo shape e pode gerar erro.
```

Programa: Trecho retirado de <https://towardsdatascience.com/squeeze-and-excitation-networks-9ef5e71eacd7>. Provavelmente tem erros, mas ilustra os conceitos de squeeze and excitation.

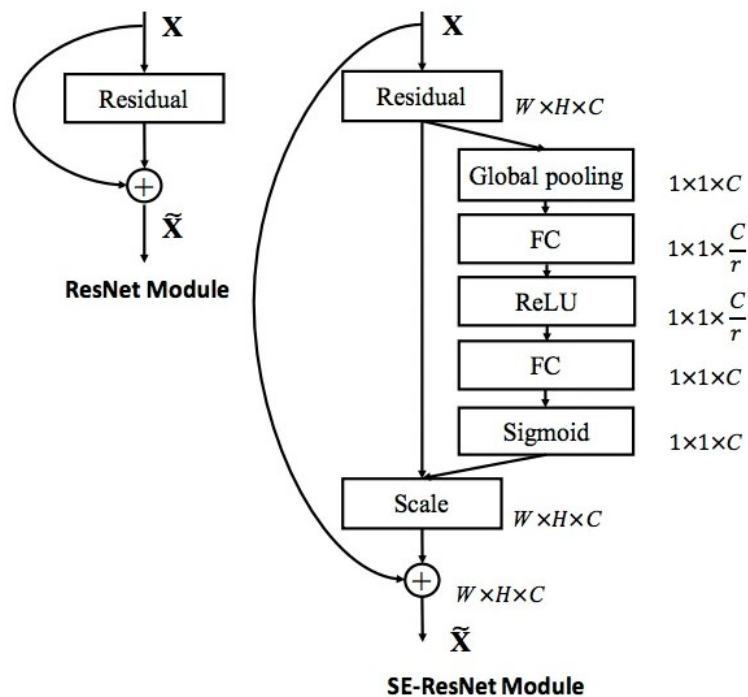


Figura: Módulo ResNet (esquerda) e o módulo alterado com squeeze and excitation (direita). Retirado de <https://towardsdatascience.com/squeeze-and-excitation-networks-9ef5e71eacd7>.



## Bloco MBConv de EfficientNet

EfficientNet usa blocos MBConv, que utilizam todas as ideias expostas acima.

Veja <https://datamonje.medium.com/image-classification-with-efficientnet-better-performance-with-computational-efficiency-f480fdb00ac6>

O uso de EfficientNet para classificar Cifar-10 está na apostila transf-ead.

## 10. Vision Transformer

Muitas arquiteturas recentes usam a técnica “vision transformer”.

<https://viso.ai/deep-learning/vision-transformer-vit/>

<https://machinelearningmastery.com/the-vision-transformer-model/>

<https://semiengineering.com/achieving-greater-accuracy-in-real-time-vision-processing-with-transformers/>

[https://keras.io/examples/vision/image\\_classification\\_with\\_vision\\_transformer/](https://keras.io/examples/vision/image_classification_with_vision_transformer/)

Apesar destas arquiteturas funcionarem bem com quantidade de dados muito grande, costumam ser pior que CNN quando tem poucos dados.

**Exercício:** Na apostila convkeras-ead, conseguimos classificar MNIST com taxa de erro de teste de **0,37%** (taxa de acerto de teste de **99,63%**). Use qualquer técnica ou combinação das técnicas vistas nesta apostila para chegar a taxa de erro menor que **0,3%** para classificar MNIST.

*Nota:* Não resolvi este exercício. Não sei se é possível resolvê-la.

Possíveis técnicas:

- a) Usar rede inspirada nas ideias de VGG, Inception, ResNet ou EfficientNet.
- b) Usar data augmentation mais sofisticada.
- c) Trocar convoluções maiores por sequências de convoluções 3x3.
- d) Usar módulo Inception (convolução 1x1, 3x3, 5x5, e maxpooling 3x3 com stride 1) com convoluções 1x1.
- e) Usar conexão-identidade como no ResNet.
- f) Usar bloco residual invertido como no EfficientNet.
- g) Usar “squeeze and excitation” como no EfficientNet.
- h) Usar batch normalization.
- i) Usar regularização L1 ou L2.
- j) Usar dropout.
- k) Usar função callback para diminuir learning rate quando o desempenho da rede para de melhorar.
- l) Usar TTA (test time augmentation).
- m) Usar ensemble.
- n) Usar ativação Swish.

**[PSI5790 aula 7. Fim.]**

## Referências

[Krizhevsky2012] KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems, v. 25, p. 1097-1105, 2012.

[Krizhevsky2017] Krizhevsky, Alex; Sutskever, Ilya; Hinton, Geoffrey E. (2017-05-24). "ImageNet classification with deep convolutional neural networks" (PDF). Communications of the ACM. 60 (6): 84–90.

[AlexNet] <https://towardsdatascience.com/alexnet-the-architecture-that-challenged-cnns-e406d5297951>

[Wiki-ImageNet] <https://en.wikipedia.org/wiki/ImageNet>

[Wiki-AlexNet] <https://en.wikipedia.org/wiki/AlexNet>

[Das2017] Siddharth Das, Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more... <https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5>

[Simonyan2014] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv:1409.1556* (2014).

[Ioffe2015] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." arXiv preprint arXiv:1502.03167 (2015). <https://arxiv.org/pdf/1502.03167v3.pdf>

[He2015] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385, 2015.

[Tsang2018] <https://medium.com/coinmonks/paper-review-of-googlenet-inception-v1-winner-of-ilsvlc-2014-image-classification-c2b3565a64e7>

[Szegedy2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 1-9).

[Zhang2016] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

[Tan2019] Tan, M., & Le, Q. (2019, May). Efficientnet: Rethinking model scaling for convolutional neural networks. In International conference on machine learning (pp. 6105-6114). PMLR.

[Liu2022] Liu, Z., Mao, H., Wu, C. Y., Feichtenhofer, C., Darrell, T., & Xie, S. (2022). A convnet for the 2020s. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 11976-11986).

Site com as taxas de acertos das principais redes: <https://github.com/keras-team/keras-applications>

Manual de Keras com as principais redes: <https://keras.io/api/applications/>

## Anexo A: Estrutura de VGG16

Model: "vgg16"

Layer (type)	Output Shape	Param #
=====		
input_2 (InputLayer)	[(None, 224, 224, 3)]	0
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
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: 138,357,544		
Trainable params: 138,357,544		
Non-trainable params: 0		

## Anexo B: Estrutura de InceptionV3

Model: "inception\_v3"

Layer (type)	Output Shape	Param #	Connected to
=====			
input_3 (InputLayer)	(None, 299, 299, 3)	0	input_3[0][0]
conv2d (Conv2D)	(None, 149, 149, 32)	864	conv2d[0][0]
batch_normalization (BatchNormaliza	(None, 149, 149, 32)	96	batch_normalization[0][0]
activation (Activation)	(None, 149, 149, 32)	0	activation[0][0]
conv2d_1 (Conv2D)	(None, 147, 147, 32)	9216	conv2d_1[0][0]
batch_normalization_1 (BatchNor	(None, 147, 147, 32)	96	batch_normalization_1[0][0]
activation_1 (Activation)	(None, 147, 147, 32)	0	activation_1[0][0]
conv2d_2 (Conv2D)	(None, 147, 147, 64)	18432	conv2d_2[0][0]
batch_normalization_2 (BatchNor	(None, 147, 147, 64)	192	batch_normalization_2[0][0]
activation_2 (Activation)	(None, 147, 147, 64)	0	activation_2[0][0]
max_pooling2d (MaxPooling2D)	(None, 73, 73, 64)	0	max_pooling2d[0][0]
conv2d_3 (Conv2D)	(None, 73, 73, 80)	5120	conv2d_3[0][0]
batch_normalization_3 (BatchNor	(None, 73, 73, 80)	240	batch_normalization_3[0][0]
activation_3 (Activation)	(None, 73, 73, 80)	0	activation_3[0][0]
conv2d_4 (Conv2D)	(None, 71, 71, 192)	138240	conv2d_4[0][0]
batch_normalization_4 (BatchNor	(None, 71, 71, 192)	576	batch_normalization_4[0][0]
activation_4 (Activation)	(None, 71, 71, 192)	0	activation_4[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 35, 35, 192)	0	max_pooling2d_1[0][0]
conv2d_8 (Conv2D)	(None, 35, 35, 64)	12288	conv2d_8[0][0]
batch_normalization_8 (BatchNor	(None, 35, 35, 64)	192	batch_normalization_8[0][0]
activation_8 (Activation)	(None, 35, 35, 64)	0	activation_8[0][0]
conv2d_6 (Conv2D)	(None, 35, 35, 48)	9216	conv2d_6[0][0]
conv2d_9 (Conv2D)	(None, 35, 35, 96)	55296	conv2d_9[0][0]
batch_normalization_6 (BatchNor	(None, 35, 35, 48)	144	batch_normalization_6[0][0]
batch_normalization_9 (BatchNor	(None, 35, 35, 96)	288	batch_normalization_9[0][0]
activation_6 (Activation)	(None, 35, 35, 48)	0	activation_6[0][0]
activation_9 (Activation)	(None, 35, 35, 96)	0	activation_9[0][0]
average_pooling2d (AveragePooli	(None, 35, 35, 192)	0	average_pooling2d[0][0]
conv2d_5 (Conv2D)	(None, 35, 35, 64)	12288	conv2d_5[0][0]
conv2d_7 (Conv2D)	(None, 35, 35, 64)	76800	conv2d_7[0][0]
conv2d_10 (Conv2D)	(None, 35, 35, 96)	82944	conv2d_10[0][0]
conv2d_11 (Conv2D)	(None, 35, 35, 32)	6144	conv2d_11[0][0]
batch_normalization_5 (BatchNor	(None, 35, 35, 64)	192	batch_normalization_5[0][0]
batch_normalization_7 (BatchNor	(None, 35, 35, 64)	192	batch_normalization_7[0][0]
batch_normalization_10 (BatchNo	(None, 35, 35, 96)	288	batch_normalization_10[0][0]
batch_normalization_11 (BatchNo	(None, 35, 35, 32)	96	batch_normalization_11[0][0]
activation_5 (Activation)	(None, 35, 35, 64)	0	activation_5[0][0]
activation_7 (Activation)	(None, 35, 35, 64)	0	activation_7[0][0]
activation_10 (Activation)	(None, 35, 35, 96)	0	activation_10[0][0]
activation_11 (Activation)	(None, 35, 35, 32)	0	activation_11[0][0]
mixed0 (Concatenate)	(None, 35, 35, 256)	0	mixed0[0][0]
conv2d_15 (Conv2D)	(None, 35, 35, 64)	16384	conv2d_15[0][0]
batch_normalization_15 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_15[0][0]
activation_15 (Activation)	(None, 35, 35, 64)	0	activation_15[0][0]
conv2d_13 (Conv2D)	(None, 35, 35, 48)	12288	conv2d_13[0][0]
conv2d_16 (Conv2D)	(None, 35, 35, 96)	55296	conv2d_16[0][0]
batch_normalization_13 (BatchNo	(None, 35, 35, 48)	144	batch_normalization_13[0][0]
batch_normalization_16 (BatchNo	(None, 35, 35, 96)	288	batch_normalization_16[0][0]
activation_13 (Activation)	(None, 35, 35, 48)	0	activation_13[0][0]
activation_16 (Activation)	(None, 35, 35, 96)	0	activation_16[0][0]
average_pooling2d_1 (AveragePoo	(None, 35, 35, 256)	0	average_pooling2d_1[0][0]
conv2d_12 (Conv2D)	(None, 35, 35, 64)	16384	conv2d_12[0][0]
conv2d_14 (Conv2D)	(None, 35, 35, 64)	76800	conv2d_14[0][0]
conv2d_17 (Conv2D)	(None, 35, 35, 96)	82944	conv2d_17[0][0]
conv2d_18 (Conv2D)	(None, 35, 35, 64)	16384	conv2d_18[0][0]
batch_normalization_12 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_12[0][0]
batch_normalization_14 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_14[0][0]
batch_normalization_17 (BatchNo	(None, 35, 35, 96)	288	batch_normalization_17[0][0]
batch_normalization_18 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_18[0][0]
activation_12 (Activation)	(None, 35, 35, 64)	0	activation_12[0][0]
activation_14 (Activation)	(None, 35, 35, 64)	0	activation_14[0][0]
activation_17 (Activation)	(None, 35, 35, 96)	0	activation_17[0][0]
activation_18 (Activation)	(None, 35, 35, 64)	0	activation_18[0][0]
mixed1 (Concatenate)	(None, 35, 35, 288)	0	mixed1[0][0]
conv2d_22 (Conv2D)	(None, 35, 35, 64)	18432	conv2d_22[0][0]
batch_normalization_22 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_22[0][0]
activation_22 (Activation)	(None, 35, 35, 64)	0	activation_22[0][0]
conv2d_20 (Conv2D)	(None, 35, 35, 48)	13824	conv2d_20[0][0]
conv2d_23 (Conv2D)	(None, 35, 35, 96)	55296	conv2d_23[0][0]
batch_normalization_20 (BatchNo	(None, 35, 35, 48)	144	batch_normalization_20[0][0]
batch_normalization_23 (BatchNo	(None, 35, 35, 96)	288	batch_normalization_23[0][0]
activation_20 (Activation)	(None, 35, 35, 48)	0	activation_20[0][0]
activation_23 (Activation)	(None, 35, 35, 96)	0	activation_23[0][0]
average_pooling2d_2 (AveragePoo	(None, 35, 35, 288)	0	average_pooling2d_2[0][0]
conv2d_19 (Conv2D)	(None, 35, 35, 64)	18432	conv2d_19[0][0]
conv2d_21 (Conv2D)	(None, 35, 35, 64)	76800	conv2d_21[0][0]
conv2d_24 (Conv2D)	(None, 35, 35, 96)	82944	conv2d_24[0][0]
conv2d_25 (Conv2D)	(None, 35, 35, 64)	18432	conv2d_25[0][0]
batch_normalization_19 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_19[0][0]
batch_normalization_21 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_21[0][0]
batch_normalization_24 (BatchNo	(None, 35, 35, 96)	288	batch_normalization_24[0][0]
batch_normalization_25 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_25[0][0]
activation_19 (Activation)	(None, 35, 35, 64)	0	activation_19[0][0]
activation_21 (Activation)	(None, 35, 35, 64)	0	activation_21[0][0]
activation_24 (Activation)	(None, 35, 35, 96)	0	activation_24[0][0]
activation_25 (Activation)	(None, 35, 35, 64)	0	activation_25[0][0]
mixed2 (Concatenate)	(None, 35, 35, 288)	0	mixed2[0][0]
conv2d_27 (Conv2D)	(None, 35, 35, 64)	18432	conv2d_27[0][0]
batch_normalization_27 (BatchNo	(None, 35, 35, 64)	192	batch_normalization_27[0][0]
activation_27 (Activation)	(None, 35, 35, 64)	0	activation_27[0][0]
conv2d_28 (Conv2D)	(None, 35, 35, 96)	55296	

batch_normalization_28 (BatchNo	(None, 35, 35, 96)	288	conv2d_28[0][0]
activation_28 (Activation)	(None, 35, 35, 96)	0	batch_normalization_28[0][0]
conv2d_26 (Conv2D)	(None, 17, 17, 384)	995328	mixed2[0][0]
conv2d_29 (Conv2D)	(None, 17, 17, 96)	82944	activation_28[0][0]
batch_normalization_26 (BatchNo	(None, 17, 17, 384)	1152	conv2d_26[0][0]
batch_normalization_29 (BatchNo	(None, 17, 17, 96)	288	conv2d_29[0][0]
activation_26 (Activation)	(None, 17, 17, 384)	0	batch_normalization_26[0][0]
activation_29 (Activation)	(None, 17, 17, 96)	0	batch_normalization_29[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 17, 17, 288)	0	mixed2[0][0]
mixed3 (Concatenate)	(None, 17, 17, 768)	0	activation_26[0][0]
			activation_29[0][0]
			max_pooling2d_2[0][0]
			mixed3[0][0]
conv2d_34 (Conv2D)	(None, 17, 17, 128)	98304	conv2d_34[0][0]
batch_normalization_34 (BatchNo	(None, 17, 17, 128)	384	batch_normalization_34[0][0]
activation_34 (Activation)	(None, 17, 17, 128)	0	activation_34[0][0]
conv2d_35 (Conv2D)	(None, 17, 17, 128)	114688	conv2d_35[0][0]
batch_normalization_35 (BatchNo	(None, 17, 17, 128)	384	batch_normalization_35[0][0]
activation_35 (Activation)	(None, 17, 17, 128)	0	mixed3[0][0]
conv2d_31 (Conv2D)	(None, 17, 17, 128)	98304	activation_35[0][0]
conv2d_36 (Conv2D)	(None, 17, 17, 128)	114688	conv2d_31[0][0]
batch_normalization_31 (BatchNo	(None, 17, 17, 128)	384	conv2d_36[0][0]
batch_normalization_36 (BatchNo	(None, 17, 17, 128)	384	batch_normalization_31[0][0]
activation_31 (Activation)	(None, 17, 17, 128)	0	batch_normalization_36[0][0]
activation_36 (Activation)	(None, 17, 17, 128)	0	activation_31[0][0]
conv2d_32 (Conv2D)	(None, 17, 17, 128)	114688	activation_36[0][0]
conv2d_37 (Conv2D)	(None, 17, 17, 128)	114688	conv2d_32[0][0]
batch_normalization_32 (BatchNo	(None, 17, 17, 128)	384	conv2d_37[0][0]
batch_normalization_37 (BatchNo	(None, 17, 17, 128)	384	batch_normalization_32[0][0]
activation_32 (Activation)	(None, 17, 17, 128)	0	batch_normalization_37[0][0]
activation_37 (Activation)	(None, 17, 17, 128)	0	mixed3[0][0]
average_pooling2d_3 (AveragePoo	(None, 17, 17, 768)	0	mixed3[0][0]
conv2d_30 (Conv2D)	(None, 17, 17, 192)	147456	activation_32[0][0]
conv2d_33 (Conv2D)	(None, 17, 17, 192)	172032	activation_37[0][0]
conv2d_38 (Conv2D)	(None, 17, 17, 192)	172032	average_pooling2d_3[0][0]
conv2d_39 (Conv2D)	(None, 17, 17, 192)	147456	conv2d_30[0][0]
batch_normalization_30 (BatchNo	(None, 17, 17, 192)	576	conv2d_33[0][0]
batch_normalization_33 (BatchNo	(None, 17, 17, 192)	576	conv2d_38[0][0]
batch_normalization_38 (BatchNo	(None, 17, 17, 192)	576	conv2d_39[0][0]
batch_normalization_39 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_30[0][0]
activation_30 (Activation)	(None, 17, 17, 192)	0	batch_normalization_33[0][0]
activation_33 (Activation)	(None, 17, 17, 192)	0	batch_normalization_38[0][0]
activation_38 (Activation)	(None, 17, 17, 192)	0	batch_normalization_39[0][0]
activation_39 (Activation)	(None, 17, 17, 192)	0	activation_30[0][0]
mixed4 (Concatenate)	(None, 17, 17, 768)	0	activation_33[0][0]
			activation_38[0][0]
			activation_39[0][0]
			mixed4[0][0]
			conv2d_44[0][0]
			batch_normalization_44[0][0]
			activation_44[0][0]
			conv2d_45[0][0]
			batch_normalization_45[0][0]
			mixed4[0][0]
			activation_45[0][0]
			conv2d_41[0][0]
			conv2d_46[0][0]
			batch_normalization_41[0][0]
			batch_normalization_46[0][0]
			activation_41[0][0]
			activation_46[0][0]
			conv2d_42[0][0]
			conv2d_47[0][0]
			batch_normalization_42[0][0]
			conv2d_47[0][0]
			activation_42[0][0]
			batch_normalization_47[0][0]
			mixed4[0][0]
			mixed4[0][0]
			activation_42[0][0]
			activation_47[0][0]
			average_pooling2d_4[0][0]
			conv2d_40[0][0]
			conv2d_43[0][0]
			conv2d_48[0][0]
			conv2d_49[0][0]
			batch_normalization_40[0][0]
			batch_normalization_43[0][0]
			batch_normalization_48[0][0]
			batch_normalization_49[0][0]
			activation_40[0][0]
			activation_43[0][0]
			activation_48[0][0]
			activation_49[0][0]
			mixed5[0][0]
			conv2d_54[0][0]
			batch_normalization_54[0][0]
			activation_54[0][0]
			conv2d_55[0][0]
			batch_normalization_55[0][0]
			mixed5[0][0]
			activation_55[0][0]
			conv2d_51[0][0]
			conv2d_56[0][0]
			batch_normalization_51[0][0]
			batch_normalization_56[0][0]
			activation_51[0][0]
			activation_56[0][0]
			conv2d_52[0][0]
			conv2d_57[0][0]
			batch_normalization_52[0][0]
			batch_normalization_57[0][0]
			mixed5[0][0]
			mixed5[0][0]
			activation_52[0][0]
			activation_57[0][0]
			average_pooling2d_5[0][0]
			conv2d_50[0][0]
			conv2d_53[0][0]
			conv2d_58[0][0]
			conv2d_59[0][0]
			batch_normalization_50[0][0]
			batch_normalization_53[0][0]

batch_normalization_58 (BatchNo	(None, 17, 17, 192)	576	conv2d_58[0][0]
batch_normalization_59 (BatchNo	(None, 17, 17, 192)	576	conv2d_59[0][0]
activation_50 (Activation)	(None, 17, 17, 192)	0	batch_normalization_50[0][0]
activation_53 (Activation)	(None, 17, 17, 192)	0	batch_normalization_53[0][0]
activation_58 (Activation)	(None, 17, 17, 192)	0	batch_normalization_58[0][0]
activation_59 (Activation)	(None, 17, 17, 192)	0	batch_normalization_59[0][0]
mixed6 (Concatenate)	(None, 17, 17, 768)	0	activation_50[0][0]
			activation_53[0][0]
			activation_58[0][0]
			activation_59[0][0]
			mixed6[0][0]
conv2d_64 (Conv2D)	(None, 17, 17, 192)	147456	conv2d_64[0][0]
batch_normalization_64 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_64[0][0]
activation_64 (Activation)	(None, 17, 17, 192)	0	activation_64[0][0]
conv2d_65 (Conv2D)	(None, 17, 17, 192)	258048	conv2d_65[0][0]
batch_normalization_65 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_65[0][0]
activation_65 (Activation)	(None, 17, 17, 192)	0	mixed6[0][0]
conv2d_61 (Conv2D)	(None, 17, 17, 192)	147456	activation_65[0][0]
conv2d_66 (Conv2D)	(None, 17, 17, 192)	258048	conv2d_61[0][0]
batch_normalization_61 (BatchNo	(None, 17, 17, 192)	576	conv2d_66[0][0]
batch_normalization_66 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_61[0][0]
activation_61 (Activation)	(None, 17, 17, 192)	0	batch_normalization_66[0][0]
activation_66 (Activation)	(None, 17, 17, 192)	0	activation_61[0][0]
conv2d_62 (Conv2D)	(None, 17, 17, 192)	258048	activation_66[0][0]
conv2d_67 (Conv2D)	(None, 17, 17, 192)	258048	conv2d_62[0][0]
batch_normalization_62 (BatchNo	(None, 17, 17, 192)	576	conv2d_67[0][0]
batch_normalization_67 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_62[0][0]
activation_62 (Activation)	(None, 17, 17, 192)	0	batch_normalization_67[0][0]
activation_67 (Activation)	(None, 17, 17, 192)	0	mixed6[0][0]
average_pooling2d_6 (AveragePoo	(None, 17, 17, 768)	0	mixed6[0][0]
conv2d_60 (Conv2D)	(None, 17, 17, 192)	147456	activation_62[0][0]
conv2d_63 (Conv2D)	(None, 17, 17, 192)	258048	activation_67[0][0]
conv2d_68 (Conv2D)	(None, 17, 17, 192)	258048	average_pooling2d_6[0][0]
conv2d_69 (Conv2D)	(None, 17, 17, 192)	147456	conv2d_60[0][0]
batch_normalization_60 (BatchNo	(None, 17, 17, 192)	576	conv2d_63[0][0]
batch_normalization_63 (BatchNo	(None, 17, 17, 192)	576	conv2d_68[0][0]
batch_normalization_68 (BatchNo	(None, 17, 17, 192)	576	conv2d_69[0][0]
batch_normalization_69 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_60[0][0]
activation_60 (Activation)	(None, 17, 17, 192)	0	batch_normalization_63[0][0]
activation_63 (Activation)	(None, 17, 17, 192)	0	batch_normalization_68[0][0]
activation_68 (Activation)	(None, 17, 17, 192)	0	batch_normalization_69[0][0]
activation_69 (Activation)	(None, 17, 17, 192)	0	activation_60[0][0]
mixed7 (Concatenate)	(None, 17, 17, 768)	0	activation_63[0][0]
			activation_68[0][0]
			activation_69[0][0]
			mixed7[0][0]
conv2d_72 (Conv2D)	(None, 17, 17, 192)	147456	conv2d_72[0][0]
batch_normalization_72 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_72[0][0]
activation_72 (Activation)	(None, 17, 17, 192)	0	activation_72[0][0]
conv2d_73 (Conv2D)	(None, 17, 17, 192)	258048	conv2d_73[0][0]
batch_normalization_73 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_73[0][0]
activation_73 (Activation)	(None, 17, 17, 192)	0	mixed7[0][0]
conv2d_70 (Conv2D)	(None, 17, 17, 192)	147456	activation_73[0][0]
conv2d_74 (Conv2D)	(None, 17, 17, 192)	258048	conv2d_70[0][0]
batch_normalization_70 (BatchNo	(None, 17, 17, 192)	576	conv2d_74[0][0]
batch_normalization_74 (BatchNo	(None, 17, 17, 192)	576	batch_normalization_70[0][0]
activation_70 (Activation)	(None, 17, 17, 192)	0	batch_normalization_74[0][0]
activation_74 (Activation)	(None, 17, 17, 192)	0	activation_70[0][0]
conv2d_71 (Conv2D)	(None, 8, 8, 320)	552960	activation_74[0][0]
conv2d_75 (Conv2D)	(None, 8, 8, 192)	331776	conv2d_71[0][0]
batch_normalization_71 (BatchNo	(None, 8, 8, 320)	960	conv2d_75[0][0]
batch_normalization_75 (BatchNo	(None, 8, 8, 192)	576	batch_normalization_71[0][0]
activation_71 (Activation)	(None, 8, 8, 320)	0	batch_normalization_75[0][0]
activation_75 (Activation)	(None, 8, 8, 192)	0	mixed7[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 768)	0	activation_71[0][0]
mixed8 (Concatenate)	(None, 8, 8, 1280)	0	activation_75[0][0]
			max_pooling2d_3[0][0]
			mixed8[0][0]
conv2d_80 (Conv2D)	(None, 8, 8, 448)	573440	conv2d_80[0][0]
batch_normalization_80 (BatchNo	(None, 8, 8, 448)	1344	batch_normalization_80[0][0]
activation_80 (Activation)	(None, 8, 8, 448)	0	mixed8[0][0]
conv2d_77 (Conv2D)	(None, 8, 8, 384)	491520	activation_80[0][0]
conv2d_81 (Conv2D)	(None, 8, 8, 384)	1548288	conv2d_77[0][0]
batch_normalization_77 (BatchNo	(None, 8, 8, 384)	1152	conv2d_81[0][0]
batch_normalization_81 (BatchNo	(None, 8, 8, 384)	1152	batch_normalization_77[0][0]
activation_77 (Activation)	(None, 8, 8, 384)	0	batch_normalization_81[0][0]
activation_81 (Activation)	(None, 8, 8, 384)	0	activation_77[0][0]
conv2d_78 (Conv2D)	(None, 8, 8, 384)	442368	activation_77[0][0]
conv2d_79 (Conv2D)	(None, 8, 8, 384)	442368	activation_81[0][0]
conv2d_82 (Conv2D)	(None, 8, 8, 384)	442368	activation_81[0][0]
conv2d_83 (Conv2D)	(None, 8, 8, 384)	442368	mixed8[0][0]
average_pooling2d_7 (AveragePoo	(None, 8, 8, 1280)	0	mixed8[0][0]
conv2d_76 (Conv2D)	(None, 8, 8, 320)	409600	conv2d_78[0][0]
batch_normalization_78 (BatchNo	(None, 8, 8, 384)	1152	conv2d_79[0][0]
batch_normalization_79 (BatchNo	(None, 8, 8, 384)	1152	conv2d_82[0][0]
batch_normalization_82 (BatchNo	(None, 8, 8, 384)	1152	conv2d_83[0][0]
batch_normalization_83 (BatchNo	(None, 8, 8, 384)	1152	average_pooling2d_7[0][0]
conv2d_84 (Conv2D)	(None, 8, 8, 192)	245760	conv2d_76[0][0]
batch_normalization_76 (BatchNo	(None, 8, 8, 320)	960	batch_normalization_78[0][0]
activation_78 (Activation)	(None, 8, 8, 384)	0	batch_normalization_79[0][0]
activation_79 (Activation)	(None, 8, 8, 384)	0	batch_normalization_82[0][0]
activation_82 (Activation)	(None, 8, 8, 384)	0	batch_normalization_83[0][0]
activation_83 (Activation)	(None, 8, 8, 384)	0	conv2d_84[0][0]
batch_normalization_84 (BatchNo	(None, 8, 8, 192)	576	batch_normalization_76[0][0]
activation_76 (Activation)	(None, 8, 8, 320)	0	activation_78[0][0]
mixed9_0 (Concatenate)	(None, 8, 8, 768)	0	activation_79[0][0]
			activation_82[0][0]
			activation_83[0][0]
concatenate (Concatenate)	(None, 8, 8, 768)	0	batch_normalization_84[0][0]
			activation_76[0][0]
activation_84 (Activation)	(None, 8, 8, 192)	0	mixed9_0[0][0]
mixed9 (Concatenate)	(None, 8, 8, 2048)	0	concatenate[0][0]
			activation_84[0][0]
			mixed9[0][0]
conv2d_89 (Conv2D)	(None, 8, 8, 448)	917504	conv2d_89[0][0]
batch_normalization_89 (BatchNo	(None, 8, 8, 448)	1344	batch_normalization_89[0][0]
activation_89 (Activation)	(None, 8, 8, 448)	0	mixed9[0][0]
conv2d_86 (Conv2D)	(None, 8, 8, 384)	786432	



conv2d_90 (Conv2D)	(None, 8, 8, 384)	1548288	activation_89[0][0]
batch_normalization_86 (BatchNo	(None, 8, 8, 384)	1152	conv2d_86[0][0]
batch_normalization_90 (BatchNo	(None, 8, 8, 384)	1152	conv2d_90[0][0]
activation_86 (Activation)	(None, 8, 8, 384)	0	batch_normalization_86[0][0]
activation_90 (Activation)	(None, 8, 8, 384)	0	batch_normalization_90[0][0]
conv2d_87 (Conv2D)	(None, 8, 8, 384)	442368	activation_86[0][0]
conv2d_88 (Conv2D)	(None, 8, 8, 384)	442368	activation_86[0][0]
conv2d_91 (Conv2D)	(None, 8, 8, 384)	442368	activation_90[0][0]
conv2d_92 (Conv2D)	(None, 8, 8, 384)	442368	activation_90[0][0]
average_pooling2d_8 (AveragePoo	(None, 8, 8, 2048)	0	mixed9[0][0]
conv2d_85 (Conv2D)	(None, 8, 8, 320)	655360	mixed9[0][0]
batch_normalization_87 (BatchNo	(None, 8, 8, 384)	1152	conv2d_87[0][0]
batch_normalization_88 (BatchNo	(None, 8, 8, 384)	1152	conv2d_88[0][0]
batch_normalization_91 (BatchNo	(None, 8, 8, 384)	1152	conv2d_91[0][0]
batch_normalization_92 (BatchNo	(None, 8, 8, 384)	1152	conv2d_92[0][0]
conv2d_93 (Conv2D)	(None, 8, 8, 192)	393216	average_pooling2d_8[0][0]
batch_normalization_85 (BatchNo	(None, 8, 8, 320)	960	conv2d_85[0][0]
activation_87 (Activation)	(None, 8, 8, 384)	0	batch_normalization_87[0][0]
activation_88 (Activation)	(None, 8, 8, 384)	0	batch_normalization_88[0][0]
activation_91 (Activation)	(None, 8, 8, 384)	0	batch_normalization_91[0][0]
activation_92 (Activation)	(None, 8, 8, 384)	0	batch_normalization_92[0][0]
batch_normalization_93 (BatchNo	(None, 8, 8, 192)	576	conv2d_93[0][0]
activation_85 (Activation)	(None, 8, 8, 320)	0	batch_normalization_85[0][0]
mixed9_1 (Concatenate)	(None, 8, 8, 768)	0	activation_87[0][0]
			activation_88[0][0]
concatenate_1 (Concatenate)	(None, 8, 8, 768)	0	activation_91[0][0]
			activation_92[0][0]
activation_93 (Activation)	(None, 8, 8, 192)	0	batch_normalization_93[0][0]
mixed10 (Concatenate)	(None, 8, 8, 2048)	0	activation_85[0][0]
			mixed9_1[0][0]
			concatenate_1[0][0]
			activation_93[0][0]
avg_pool (GlobalAveragePooling2	(None, 2048)	0	mixed10[0][0]
predictions (Dense)	(None, 1000)	2049000	avg_pool[0][0]

=====

Total params: 23,851,784  
Trainable params: 23,817,352  
Non-trainable params: 34,432

## Anexo C: Estrutura de ResNet50 (v1)

Model: "resnet50"				
Layer (type)	Output Shape	Param #	Connected to	
input_1 (InputLayer)	(None, 224, 224, 3)	0	input_1[0][0]	
conv1_pad (ZeroPadding2D)	(None, 230, 230, 3)	0	conv1_pad[0][0]	
conv1_conv (Conv2D)	(None, 112, 112, 64)	9472	conv1_conv[0][0]	
conv1_bn (BatchNormalization)	(None, 112, 112, 64)	256	conv1_bn[0][0]	
conv1_relu (Activation)	(None, 112, 112, 64)	0	conv1_relu[0][0]	
pool1_pad (ZeroPadding2D)	(None, 114, 114, 64)	0	pool1_pad[0][0]	
pool1_pool (MaxPooling2D)	(None, 56, 56, 64)	0	pool1_pool[0][0]	
conv2_block1_1_conv (Conv2D)	(None, 56, 56, 64)	4160	conv2_block1_1_conv[0][0]	
conv2_block1_1_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block1_1_bn[0][0]	
conv2_block1_1_relu (Activation)	(None, 56, 56, 64)	0	conv2_block1_1_relu[0][0]	
conv2_block1_2_conv (Conv2D)	(None, 56, 56, 64)	36928	conv2_block1_2_conv[0][0]	
conv2_block1_2_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block1_2_bn[0][0]	
conv2_block1_2_relu (Activation)	(None, 56, 56, 64)	0	conv2_block1_2_relu[0][0]	
conv2_block1_0_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block1_0_conv[0][0]	
conv2_block1_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block1_3_conv[0][0]	
conv2_block1_0_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block1_0_bn[0][0]	
conv2_block1_3_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block1_3_bn[0][0]	
conv2_block1_add (Add)	(None, 56, 56, 256)	0	conv2_block1_add[0][0]	
conv2_block1_out (Activation)	(None, 56, 56, 256)	0	conv2_block1_out[0][0]	
conv2_block2_1_conv (Conv2D)	(None, 56, 56, 64)	16448	conv2_block2_1_conv[0][0]	
conv2_block2_1_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block2_1_bn[0][0]	
conv2_block2_1_relu (Activation)	(None, 56, 56, 64)	0	conv2_block2_1_relu[0][0]	
conv2_block2_2_conv (Conv2D)	(None, 56, 56, 64)	36928	conv2_block2_2_conv[0][0]	
conv2_block2_2_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block2_2_bn[0][0]	
conv2_block2_2_relu (Activation)	(None, 56, 56, 64)	0	conv2_block2_2_relu[0][0]	
conv2_block2_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block2_3_conv[0][0]	
conv2_block2_3_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block2_3_bn[0][0]	
conv2_block2_add (Add)	(None, 56, 56, 256)	0	conv2_block2_add[0][0]	
conv2_block2_out (Activation)	(None, 56, 56, 256)	0	conv2_block2_out[0][0]	
conv2_block3_1_conv (Conv2D)	(None, 56, 56, 64)	16448	conv2_block3_1_conv[0][0]	
conv2_block3_1_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block3_1_bn[0][0]	
conv2_block3_1_relu (Activation)	(None, 56, 56, 64)	0	conv2_block3_1_relu[0][0]	
conv2_block3_2_conv (Conv2D)	(None, 56, 56, 64)	36928	conv2_block3_2_conv[0][0]	
conv2_block3_2_bn (BatchNormali	(None, 56, 56, 64)	256	conv2_block3_2_bn[0][0]	
conv2_block3_2_relu (Activation)	(None, 56, 56, 64)	0	conv2_block3_2_relu[0][0]	
conv2_block3_3_conv (Conv2D)	(None, 56, 56, 256)	16640	conv2_block3_3_conv[0][0]	
conv2_block3_3_bn (BatchNormali	(None, 56, 56, 256)	1024	conv2_block3_3_bn[0][0]	
conv2_block3_add (Add)	(None, 56, 56, 256)	0	conv2_block3_add[0][0]	
conv2_block3_out (Activation)	(None, 56, 56, 256)	0	conv2_block3_out[0][0]	
conv3_block1_1_conv (Conv2D)	(None, 28, 28, 128)	32896	conv3_block1_1_conv[0][0]	
conv3_block1_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block1_1_bn[0][0]	
conv3_block1_1_relu (Activation)	(None, 28, 28, 128)	0	conv3_block1_1_relu[0][0]	
conv3_block1_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block1_2_conv[0][0]	
conv3_block1_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block1_2_bn[0][0]	
conv3_block1_2_relu (Activation)	(None, 28, 28, 128)	0	conv3_block1_2_relu[0][0]	
conv3_block1_0_conv (Conv2D)	(None, 28, 28, 512)	131584	conv3_block1_0_conv[0][0]	
conv3_block1_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block1_3_conv[0][0]	
conv3_block1_0_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block1_0_bn[0][0]	
conv3_block1_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block1_3_bn[0][0]	
conv3_block1_add (Add)	(None, 28, 28, 512)	0	conv3_block1_add[0][0]	
conv3_block1_out (Activation)	(None, 28, 28, 512)	0	conv3_block1_out[0][0]	
conv3_block2_1_conv (Conv2D)	(None, 28, 28, 128)	65664	conv3_block2_1_conv[0][0]	
conv3_block2_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block2_1_bn[0][0]	
conv3_block2_1_relu (Activation)	(None, 28, 28, 128)	0	conv3_block2_1_relu[0][0]	
conv3_block2_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block2_2_conv[0][0]	
conv3_block2_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block2_2_bn[0][0]	
conv3_block2_2_relu (Activation)	(None, 28, 28, 128)	0	conv3_block2_2_relu[0][0]	
conv3_block2_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block2_3_conv[0][0]	
conv3_block2_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block2_3_bn[0][0]	
conv3_block2_add (Add)	(None, 28, 28, 512)	0	conv3_block2_add[0][0]	
conv3_block2_out (Activation)	(None, 28, 28, 512)	0	conv3_block2_out[0][0]	
conv3_block3_1_conv (Conv2D)	(None, 28, 28, 128)	65664	conv3_block3_1_conv[0][0]	
conv3_block3_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block3_1_bn[0][0]	
conv3_block3_1_relu (Activation)	(None, 28, 28, 128)	0	conv3_block3_1_relu[0][0]	
conv3_block3_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block3_2_conv[0][0]	
conv3_block3_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block3_2_bn[0][0]	
conv3_block3_2_relu (Activation)	(None, 28, 28, 128)	0	conv3_block3_2_relu[0][0]	
conv3_block3_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block3_3_conv[0][0]	
conv3_block3_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block3_3_bn[0][0]	
conv3_block3_add (Add)	(None, 28, 28, 512)	0	conv3_block3_add[0][0]	
conv3_block3_out (Activation)	(None, 28, 28, 512)	0	conv3_block3_out[0][0]	
conv3_block4_1_conv (Conv2D)	(None, 28, 28, 128)	65664	conv3_block4_1_conv[0][0]	
conv3_block4_1_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block4_1_bn[0][0]	
conv3_block4_1_relu (Activation)	(None, 28, 28, 128)	0	conv3_block4_1_relu[0][0]	
conv3_block4_2_conv (Conv2D)	(None, 28, 28, 128)	147584	conv3_block4_2_conv[0][0]	
conv3_block4_2_bn (BatchNormali	(None, 28, 28, 128)	512	conv3_block4_2_bn[0][0]	
conv3_block4_2_relu (Activation)	(None, 28, 28, 128)	0	conv3_block4_2_relu[0][0]	
conv3_block4_3_conv (Conv2D)	(None, 28, 28, 512)	66048	conv3_block4_3_conv[0][0]	
conv3_block4_3_bn (BatchNormali	(None, 28, 28, 512)	2048	conv3_block4_3_bn[0][0]	
conv3_block4_add (Add)	(None, 28, 28, 512)	0	conv3_block4_add[0][0]	
conv3_block4_out (Activation)	(None, 28, 28, 512)	0	conv3_block4_out[0][0]	
conv4_block1_1_conv (Conv2D)	(None, 14, 14, 256)	131328	conv4_block1_1_conv[0][0]	
conv4_block1_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block1_1_bn[0][0]	
conv4_block1_1_relu (Activation)	(None, 14, 14, 256)	0	conv4_block1_1_relu[0][0]	
conv4_block1_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block1_2_conv[0][0]	
conv4_block1_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block1_2_bn[0][0]	
conv4_block1_2_relu (Activation)	(None, 14, 14, 256)	0	conv4_block1_2_relu[0][0]	
conv4_block1_0_conv (Conv2D)	(None, 14, 14, 1024)	525312	conv4_block1_0_conv[0][0]	
conv4_block1_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block1_3_conv[0][0]	
conv4_block1_0_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block1_0_bn[0][0]	
conv4_block1_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block1_3_bn[0][0]	
conv4_block1_add (Add)	(None, 14, 14, 1024)	0	conv4_block1_add[0][0]	



conv4_block1_out (Activation)	(None, 14, 14, 1024)	0	conv4_block1_add[0][0]
conv4_block2_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block1_out[0][0]
conv4_block2_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block2_1_conv[0][0]
conv4_block2_1_relu (Activation)	(None, 14, 14, 256)	0	conv4_block2_1_bn[0][0]
conv4_block2_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block2_1_relu[0][0]
conv4_block2_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block2_2_conv[0][0]
conv4_block2_2_relu (Activation)	(None, 14, 14, 256)	0	conv4_block2_2_bn[0][0]
conv4_block2_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block2_2_relu[0][0]
conv4_block2_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block2_3_conv[0][0]
conv4_block2_add (Add)	(None, 14, 14, 1024)	0	conv4_block1_out[0][0]
			conv4_block2_3_bn[0][0]
conv4_block2_out (Activation)	(None, 14, 14, 1024)	0	conv4_block2_add[0][0]
conv4_block3_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block2_out[0][0]
conv4_block3_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block3_1_conv[0][0]
conv4_block3_1_relu (Activation)	(None, 14, 14, 256)	0	conv4_block3_1_bn[0][0]
conv4_block3_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block3_1_relu[0][0]
conv4_block3_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block3_2_conv[0][0]
conv4_block3_2_relu (Activation)	(None, 14, 14, 256)	0	conv4_block3_2_bn[0][0]
conv4_block3_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block3_2_relu[0][0]
conv4_block3_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block3_3_conv[0][0]
conv4_block3_add (Add)	(None, 14, 14, 1024)	0	conv4_block2_out[0][0]
			conv4_block3_3_bn[0][0]
conv4_block3_out (Activation)	(None, 14, 14, 1024)	0	conv4_block3_add[0][0]
conv4_block4_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block3_out[0][0]
conv4_block4_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block4_1_conv[0][0]
conv4_block4_1_relu (Activation)	(None, 14, 14, 256)	0	conv4_block4_1_bn[0][0]
conv4_block4_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block4_1_relu[0][0]
conv4_block4_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block4_2_conv[0][0]
conv4_block4_2_relu (Activation)	(None, 14, 14, 256)	0	conv4_block4_2_bn[0][0]
conv4_block4_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block4_2_relu[0][0]
conv4_block4_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block4_3_conv[0][0]
conv4_block4_add (Add)	(None, 14, 14, 1024)	0	conv4_block3_out[0][0]
			conv4_block4_3_bn[0][0]
conv4_block4_out (Activation)	(None, 14, 14, 1024)	0	conv4_block4_add[0][0]
conv4_block5_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block4_out[0][0]
conv4_block5_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block5_1_conv[0][0]
conv4_block5_1_relu (Activation)	(None, 14, 14, 256)	0	conv4_block5_1_bn[0][0]
conv4_block5_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block5_1_relu[0][0]
conv4_block5_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block5_2_conv[0][0]
conv4_block5_2_relu (Activation)	(None, 14, 14, 256)	0	conv4_block5_2_bn[0][0]
conv4_block5_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block5_2_relu[0][0]
conv4_block5_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block5_3_conv[0][0]
conv4_block5_add (Add)	(None, 14, 14, 1024)	0	conv4_block4_out[0][0]
			conv4_block5_3_bn[0][0]
conv4_block5_out (Activation)	(None, 14, 14, 1024)	0	conv4_block5_add[0][0]
conv4_block6_1_conv (Conv2D)	(None, 14, 14, 256)	262400	conv4_block5_out[0][0]
conv4_block6_1_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block6_1_conv[0][0]
conv4_block6_1_relu (Activation)	(None, 14, 14, 256)	0	conv4_block6_1_bn[0][0]
conv4_block6_2_conv (Conv2D)	(None, 14, 14, 256)	590080	conv4_block6_1_relu[0][0]
conv4_block6_2_bn (BatchNormali	(None, 14, 14, 256)	1024	conv4_block6_2_conv[0][0]
conv4_block6_2_relu (Activation)	(None, 14, 14, 256)	0	conv4_block6_2_bn[0][0]
conv4_block6_3_conv (Conv2D)	(None, 14, 14, 1024)	263168	conv4_block6_2_relu[0][0]
conv4_block6_3_bn (BatchNormali	(None, 14, 14, 1024)	4096	conv4_block6_3_conv[0][0]
conv4_block6_add (Add)	(None, 14, 14, 1024)	0	conv4_block5_out[0][0]
			conv4_block6_3_bn[0][0]
conv4_block6_out (Activation)	(None, 14, 14, 1024)	0	conv4_block6_add[0][0]
conv5_block1_1_conv (Conv2D)	(None, 7, 7, 512)	524800	conv4_block6_out[0][0]
conv5_block1_1_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block1_1_conv[0][0]
conv5_block1_1_relu (Activation)	(None, 7, 7, 512)	0	conv5_block1_1_bn[0][0]
conv5_block1_2_conv (Conv2D)	(None, 7, 7, 512)	2359808	conv5_block1_1_relu[0][0]
conv5_block1_2_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block1_2_conv[0][0]
conv5_block1_2_relu (Activation)	(None, 7, 7, 512)	0	conv5_block1_2_bn[0][0]
conv5_block1_0_conv (Conv2D)	(None, 7, 7, 2048)	2099200	conv4_block6_out[0][0]
conv5_block1_3_conv (Conv2D)	(None, 7, 7, 2048)	1050624	conv5_block1_2_relu[0][0]
conv5_block1_0_bn (BatchNormali	(None, 7, 7, 2048)	8192	conv5_block1_0_conv[0][0]
conv5_block1_3_bn (BatchNormali	(None, 7, 7, 2048)	8192	conv5_block1_3_conv[0][0]
conv5_block1_add (Add)	(None, 7, 7, 2048)	0	conv5_block1_0_bn[0][0]
			conv5_block1_3_bn[0][0]
conv5_block1_out (Activation)	(None, 7, 7, 2048)	0	conv5_block1_add[0][0]
conv5_block2_1_conv (Conv2D)	(None, 7, 7, 512)	1049088	conv5_block1_out[0][0]
conv5_block2_1_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block2_1_conv[0][0]
conv5_block2_1_relu (Activation)	(None, 7, 7, 512)	0	conv5_block2_1_bn[0][0]
conv5_block2_2_conv (Conv2D)	(None, 7, 7, 512)	2359808	conv5_block2_1_relu[0][0]
conv5_block2_2_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block2_2_conv[0][0]
conv5_block2_2_relu (Activation)	(None, 7, 7, 512)	0	conv5_block2_2_bn[0][0]
conv5_block2_3_conv (Conv2D)	(None, 7, 7, 2048)	1050624	conv5_block2_2_relu[0][0]
conv5_block2_3_bn (BatchNormali	(None, 7, 7, 2048)	8192	conv5_block2_3_conv[0][0]
conv5_block2_add (Add)	(None, 7, 7, 2048)	0	conv5_block1_out[0][0]
			conv5_block2_3_bn[0][0]
conv5_block2_out (Activation)	(None, 7, 7, 2048)	0	conv5_block2_add[0][0]
conv5_block3_1_conv (Conv2D)	(None, 7, 7, 512)	1049088	conv5_block2_out[0][0]
conv5_block3_1_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block3_1_conv[0][0]
conv5_block3_1_relu (Activation)	(None, 7, 7, 512)	0	conv5_block3_1_bn[0][0]
conv5_block3_2_conv (Conv2D)	(None, 7, 7, 512)	2359808	conv5_block3_1_relu[0][0]
conv5_block3_2_bn (BatchNormali	(None, 7, 7, 512)	2048	conv5_block3_2_conv[0][0]
conv5_block3_2_relu (Activation)	(None, 7, 7, 512)	0	conv5_block3_2_bn[0][0]
conv5_block3_3_conv (Conv2D)	(None, 7, 7, 2048)	1050624	conv5_block3_2_relu[0][0]
conv5_block3_3_bn (BatchNormali	(None, 7, 7, 2048)	8192	conv5_block3_3_conv[0][0]
conv5_block3_add (Add)	(None, 7, 7, 2048)	0	conv5_block2_out[0][0]
			conv5_block3_3_bn[0][0]
conv5_block3_out (Activation)	(None, 7, 7, 2048)	0	conv5_block3_add[0][0]
avg_pool (GlobalAveragePooling2	(None, 2048)	0	conv5_block3_out[0][0]
predictions (Dense)	(None, 1000)	2049000	avg_pool[0][0]

Total params: 25,636,712  
 Trainable params: 25,583,592  
 Non-trainable params: 53,120

## Anexo D: EfficientNetB0

Model: "efficientnetb0"

Layer (type)	Output Shape	Param #	Connected to
input_7 (InputLayer)	[(None, 224, 224, 3)]	0	input_7[0][0]
rescaling_3 (Rescaling)	(None, 224, 224, 3)	0	rescaling_3[0][0]
normalization_3 (Normalization)	(None, 224, 224, 3)	7	normalization_3[0][0]
stem_conv_pad (ZeroPadding2D)	(None, 225, 225, 3)	0	stem_conv_pad[0][0]
stem_conv (Conv2D)	(None, 112, 112, 32)	864	stem_conv[0][0]
stem_bn (BatchNormalization)	(None, 112, 112, 32)	128	stem_bn[0][0]
stem_activation (Activation)	(None, 112, 112, 32)	0	stem_activation[0][0]
block1a_dwconv (DepthwiseConv2D)	(None, 112, 112, 32)	288	block1a_dwconv[0][0]
block1a_bn (BatchNormalization)	(None, 112, 112, 32)	128	block1a_bn[0][0]
block1a_activation (Activation)	(None, 112, 112, 32)	0	block1a_activation[0][0]
block1a_se_squeeze (GlobalAveragePooling2D)	(None, 32)	0	block1a_se_squeeze[0][0]
block1a_se_reshape (Reshape)	(None, 1, 1, 32)	0	block1a_se_reshape[0][0]
block1a_se_reduce (Conv2D)	(None, 1, 1, 8)	264	block1a_se_reduce[0][0]
block1a_se_expand (Conv2D)	(None, 1, 1, 32)	288	block1a_se_expand[0][0]
block1a_se_excite (Multiply)	(None, 112, 112, 32)	0	block1a_se_excite[0][0]
block1a_project_conv (Conv2D)	(None, 112, 112, 16)	512	block1a_project_conv[0][0]
block1a_project_bn (BatchNormalization)	(None, 112, 112, 16)	64	block1a_project_bn[0][0]
block2a_expand_conv (Conv2D)	(None, 112, 112, 96)	1536	block2a_expand_conv[0][0]
block2a_expand_bn (BatchNormalization)	(None, 112, 112, 96)	384	block2a_expand_bn[0][0]
block2a_expand_activation (Activation)	(None, 112, 112, 96)	0	block2a_expand_activation[0][0]
block2a_dwconv_pad (ZeroPadding2D)	(None, 113, 113, 96)	0	block2a_dwconv_pad[0][0]
block2a_dwconv (DepthwiseConv2D)	(None, 56, 56, 96)	864	block2a_dwconv[0][0]
block2a_bn (BatchNormalization)	(None, 56, 56, 96)	384	block2a_bn[0][0]
block2a_activation (Activation)	(None, 56, 56, 96)	0	block2a_activation[0][0]
block2a_se_squeeze (GlobalAveragePooling2D)	(None, 96)	0	block2a_se_squeeze[0][0]
block2a_se_reshape (Reshape)	(None, 1, 1, 96)	0	block2a_se_reshape[0][0]
block2a_se_reduce (Conv2D)	(None, 1, 1, 4)	388	block2a_se_reduce[0][0]
block2a_se_expand (Conv2D)	(None, 1, 1, 96)	480	block2a_se_expand[0][0]
block2a_se_excite (Multiply)	(None, 56, 56, 96)	0	block2a_se_excite[0][0]
block2a_project_conv (Conv2D)	(None, 56, 56, 24)	2304	block2a_project_conv[0][0]
block2a_project_bn (BatchNormalization)	(None, 56, 56, 24)	96	block2a_project_bn[0][0]
block2b_expand_conv (Conv2D)	(None, 56, 56, 144)	3456	block2b_expand_conv[0][0]
block2b_expand_bn (BatchNormalization)	(None, 56, 56, 144)	576	block2b_expand_bn[0][0]
block2b_expand_activation (Activation)	(None, 56, 56, 144)	0	block2b_expand_activation[0][0]
block2b_dwconv (DepthwiseConv2D)	(None, 56, 56, 144)	1296	block2b_dwconv[0][0]
block2b_bn (BatchNormalization)	(None, 56, 56, 144)	576	block2b_bn[0][0]
block2b_activation (Activation)	(None, 56, 56, 144)	0	block2b_activation[0][0]
block2b_se_squeeze (GlobalAveragePooling2D)	(None, 144)	0	block2b_se_squeeze[0][0]
block2b_se_reshape (Reshape)	(None, 1, 1, 144)	0	block2b_se_reshape[0][0]
block2b_se_reduce (Conv2D)	(None, 1, 1, 6)	870	block2b_se_reduce[0][0]
block2b_se_expand (Conv2D)	(None, 1, 1, 144)	1008	block2b_se_expand[0][0]
block2b_se_excite (Multiply)	(None, 56, 56, 144)	0	block2b_se_excite[0][0]
block2b_project_conv (Conv2D)	(None, 56, 56, 24)	3456	block2b_project_conv[0][0]
block2b_project_bn (BatchNormalization)	(None, 56, 56, 24)	96	block2b_project_bn[0][0]
block2b_drop (Dropout)	(None, 56, 56, 24)	0	block2b_drop[0][0]
block2b_add (Add)	(None, 56, 56, 24)	0	block2b_add[0][0]
block3a_expand_conv (Conv2D)	(None, 56, 56, 144)	3456	block3a_expand_conv[0][0]
block3a_expand_bn (BatchNormalization)	(None, 56, 56, 144)	576	block3a_expand_bn[0][0]
block3a_expand_activation (Activation)	(None, 56, 56, 144)	0	block3a_expand_activation[0][0]
block3a_dwconv_pad (ZeroPadding2D)	(None, 59, 59, 144)	0	block3a_dwconv_pad[0][0]
block3a_dwconv (DepthwiseConv2D)	(None, 28, 28, 144)	3600	block3a_dwconv[0][0]
block3a_bn (BatchNormalization)	(None, 28, 28, 144)	576	block3a_bn[0][0]
block3a_activation (Activation)	(None, 28, 28, 144)	0	block3a_activation[0][0]
block3a_se_squeeze (GlobalAveragePooling2D)	(None, 144)	0	block3a_se_squeeze[0][0]
block3a_se_reshape (Reshape)	(None, 1, 1, 144)	0	block3a_se_reshape[0][0]
block3a_se_reduce (Conv2D)	(None, 1, 1, 6)	870	block3a_se_reduce[0][0]
block3a_se_expand (Conv2D)	(None, 1, 1, 144)	1008	block3a_se_expand[0][0]
block3a_se_excite (Multiply)	(None, 28, 28, 144)	0	block3a_se_excite[0][0]
block3a_project_conv (Conv2D)	(None, 28, 28, 40)	5760	block3a_project_conv[0][0]
block3a_project_bn (BatchNormalization)	(None, 28, 28, 40)	160	block3a_project_bn[0][0]
block3b_expand_conv (Conv2D)	(None, 28, 28, 240)	9600	block3b_expand_conv[0][0]
block3b_expand_bn (BatchNormalization)	(None, 28, 28, 240)	960	block3b_expand_bn[0][0]
block3b_expand_activation (Activation)	(None, 28, 28, 240)	0	block3b_expand_activation[0][0]
block3b_dwconv (DepthwiseConv2D)	(None, 28, 28, 240)	6000	block3b_dwconv[0][0]
block3b_bn (BatchNormalization)	(None, 28, 28, 240)	960	block3b_bn[0][0]
block3b_activation (Activation)	(None, 28, 28, 240)	0	block3b_activation[0][0]
block3b_se_squeeze (GlobalAveragePooling2D)	(None, 240)	0	block3b_se_squeeze[0][0]
block3b_se_reshape (Reshape)	(None, 1, 1, 240)	0	block3b_se_reshape[0][0]
block3b_se_reduce (Conv2D)	(None, 1, 1, 10)	2410	block3b_se_reduce[0][0]
block3b_se_expand (Conv2D)	(None, 1, 1, 240)	2640	block3b_se_expand[0][0]
block3b_se_excite (Multiply)	(None, 28, 28, 240)	0	block3b_se_excite[0][0]
block3b_project_conv (Conv2D)	(None, 28, 28, 40)	9600	block3b_project_conv[0][0]
block3b_project_bn (BatchNormalization)	(None, 28, 28, 40)	160	block3b_project_bn[0][0]
block3b_drop (Dropout)	(None, 28, 28, 40)	0	block3b_drop[0][0]
block3b_add (Add)	(None, 28, 28, 40)	0	block3b_add[0][0]
block4a_expand_conv (Conv2D)	(None, 28, 28, 240)	9600	block4a_expand_conv[0][0]
block4a_expand_bn (BatchNormalization)	(None, 28, 28, 240)	960	block4a_expand_bn[0][0]
block4a_expand_activation (Activation)	(None, 28, 28, 240)	0	block4a_expand_activation[0][0]
block4a_dwconv_pad (ZeroPadding2D)	(None, 29, 29, 240)	0	block4a_dwconv_pad[0][0]
block4a_dwconv (DepthwiseConv2D)	(None, 14, 14, 240)	2160	block4a_dwconv[0][0]
block4a_bn (BatchNormalization)	(None, 14, 14, 240)	960	block4a_bn[0][0]
block4a_activation (Activation)	(None, 14, 14, 240)	0	block4a_activation[0][0]
block4a_se_squeeze (GlobalAveragePooling2D)	(None, 240)	0	block4a_se_squeeze[0][0]
block4a_se_reshape (Reshape)	(None, 1, 1, 240)	0	block4a_se_reshape[0][0]
block4a_se_reduce (Conv2D)	(None, 1, 1, 10)	2410	block4a_se_reduce[0][0]
block4a_se_expand (Conv2D)	(None, 1, 1, 240)	2640	block4a_se_expand[0][0]
block4a_se_excite (Multiply)	(None, 14, 14, 240)	0	block4a_se_excite[0][0]
block4a_project_conv (Conv2D)	(None, 14, 14, 80)	19200	block4a_project_conv[0][0]
block4a_project_bn (BatchNormalization)	(None, 14, 14, 80)	320	block4a_project_bn[0][0]
block4b_expand_conv (Conv2D)	(None, 14, 14, 480)	38400	block4b_expand_conv[0][0]
block4b_expand_bn (BatchNormalization)	(None, 14, 14, 480)	1920	block4b_expand_bn[0][0]
block4b_expand_activation (Activation)	(None, 14, 14, 480)	0	block4b_expand_activation[0][0]
block4b_dwconv (DepthwiseConv2D)	(None, 14, 14, 480)	4320	block4b_dwconv[0][0]

block4b_bn (BatchNormalization)	(None, 14, 14, 480)	1920	block4b_dwconv[0][0]	
block4b_activation (Activation)	(None, 14, 14, 480)	0	block4b_bn[0][0]	
block4b_se_squeeze (GlobalAveragePooling2D)	(None, 480)	0	block4b_activation[0][0]	
block4b_se_reshape (Reshape)	(None, 1, 1, 480)	0	block4b_se_squeeze[0][0]	
block4b_se_reduce (Conv2D)	(None, 1, 1, 20)	9620	block4b_se_reshape[0][0]	
block4b_se_expand (Conv2D)	(None, 1, 1, 480)	10080	block4b_se_reduce[0][0]	
block4b_se_excite (Multiply)	(None, 14, 14, 480)	0	block4b_activation[0][0]	
block4b_project_conv (Conv2D)	(None, 14, 14, 80)	38400	block4b_se_expand[0][0]	
block4b_project_bn (BatchNormalization)	(None, 14, 14, 80)	320	block4b_se_excite[0][0]	
block4b_drop (Dropout)	(None, 14, 14, 80)	0	block4b_project_conv[0][0]	
block4b_add (Add)	(None, 14, 14, 80)	0	block4b_project_bn[0][0]	
block4c_expand_conv (Conv2D)	(None, 14, 14, 480)	38400	block4b_drop[0][0]	
block4c_expand_bn (BatchNormalization)	(None, 14, 14, 480)	1920	block4a_project_bn[0][0]	
block4c_expand_activation (Activation)	(None, 14, 14, 480)	0	block4b_add[0][0]	
block4c_dwconv (DepthwiseConv2D)	(None, 14, 14, 480)	4320	block4c_expand_conv[0][0]	
block4c_bn (BatchNormalization)	(None, 14, 14, 480)	1920	block4c_expand_bn[0][0]	
block4c_activation (Activation)	(None, 14, 14, 480)	0	block4c_expand_activation[0][0]	
block4c_se_squeeze (GlobalAveragePooling2D)	(None, 480)	0	block4c_dwconv[0][0]	
block4c_se_reshape (Reshape)	(None, 1, 1, 480)	0	block4c_bn[0][0]	
block4c_se_reduce (Conv2D)	(None, 1, 1, 20)	9620	block4c_activation[0][0]	
block4c_se_expand (Conv2D)	(None, 1, 1, 480)	10080	block4c_se_squeeze[0][0]	
block4c_se_excite (Multiply)	(None, 14, 14, 480)	0	block4c_se_reshape[0][0]	
block4c_project_conv (Conv2D)	(None, 14, 14, 80)	38400	block4c_se_reduce[0][0]	
block4c_project_bn (BatchNormalization)	(None, 14, 14, 80)	320	block4c_activation[0][0]	
block4c_drop (Dropout)	(None, 14, 14, 80)	0	block4c_se_expand[0][0]	
block4c_add (Add)	(None, 14, 14, 80)	0	block4c_se_excite[0][0]	
block5a_expand_conv (Conv2D)	(None, 14, 14, 480)	38400	block4c_project_conv[0][0]	
block5a_expand_bn (BatchNormalization)	(None, 14, 14, 480)	1920	block4c_project_bn[0][0]	
block5a_expand_activation (Activation)	(None, 14, 14, 480)	0	block4c_drop[0][0]	
block5a_dwconv (DepthwiseConv2D)	(None, 14, 14, 480)	12000	block4b_add[0][0]	
block5a_bn (BatchNormalization)	(None, 14, 14, 480)	1920	block4c_add[0][0]	
block5a_activation (Activation)	(None, 14, 14, 480)	0	block5a_expand_conv[0][0]	
block5a_se_squeeze (GlobalAveragePooling2D)	(None, 480)	0	block5a_expand_bn[0][0]	
block5a_se_reshape (Reshape)	(None, 1, 1, 480)	0	block5a_expand_activation[0][0]	
block5a_se_reduce (Conv2D)	(None, 1, 1, 20)	9620	block5a_dwconv[0][0]	
block5a_se_expand (Conv2D)	(None, 1, 1, 480)	10080	block5a_bn[0][0]	
block5a_se_excite (Multiply)	(None, 14, 14, 480)	0	block5a_activation[0][0]	
block5a_project_conv (Conv2D)	(None, 14, 14, 112)	53760	block5a_se_squeeze[0][0]	
block5a_project_bn (BatchNormalization)	(None, 14, 14, 112)	448	block5a_se_reshape[0][0]	
block5b_expand_conv (Conv2D)	(None, 14, 14, 672)	75264	block5a_se_reduce[0][0]	
block5b_expand_bn (BatchNormalization)	(None, 14, 14, 672)	2688	block5a_activation[0][0]	
block5b_expand_activation (Activation)	(None, 14, 14, 672)	0	block5a_se_expand[0][0]	
block5b_dwconv (DepthwiseConv2D)	(None, 14, 14, 672)	16800	block5a_se_excite[0][0]	
block5b_bn (BatchNormalization)	(None, 14, 14, 672)	2688	block5a_project_conv[0][0]	
block5b_activation (Activation)	(None, 14, 14, 672)	0	block5a_project_bn[0][0]	
block5b_se_squeeze (GlobalAveragePooling2D)	(None, 672)	0	block5b_expand_conv[0][0]	
block5b_se_reshape (Reshape)	(None, 1, 1, 672)	0	block5b_expand_bn[0][0]	
block5b_se_reduce (Conv2D)	(None, 1, 1, 28)	18844	block5b_expand_activation[0][0]	
block5b_se_expand (Conv2D)	(None, 1, 1, 672)	19488	block5b_dwconv[0][0]	
block5b_se_excite (Multiply)	(None, 14, 14, 672)	0	block5b_bn[0][0]	
block5b_project_conv (Conv2D)	(None, 14, 14, 112)	75264	block5b_activation[0][0]	
block5b_project_bn (BatchNormalization)	(None, 14, 14, 112)	448	block5b_se_squeeze[0][0]	
block5b_drop (Dropout)	(None, 14, 14, 112)	0	block5b_se_reshape[0][0]	
block5b_add (Add)	(None, 14, 14, 112)	0	block5b_se_reduce[0][0]	
block5c_expand_conv (Conv2D)	(None, 14, 14, 672)	75264	block5b_activation[0][0]	
block5c_expand_bn (BatchNormalization)	(None, 14, 14, 672)	2688	block5b_se_expand[0][0]	
block5c_expand_activation (Activation)	(None, 14, 14, 672)	0	block5b_se_excite[0][0]	
block5c_dwconv (DepthwiseConv2D)	(None, 14, 14, 672)	16800	block5b_project_conv[0][0]	
block5c_bn (BatchNormalization)	(None, 14, 14, 672)	2688	block5b_project_bn[0][0]	
block5c_activation (Activation)	(None, 14, 14, 672)	0	block5b_drop[0][0]	
block5c_se_squeeze (GlobalAveragePooling2D)	(None, 672)	0	block5a_project_bn[0][0]	
block5c_se_reshape (Reshape)	(None, 1, 1, 672)	0	block5b_add[0][0]	
block5c_se_reduce (Conv2D)	(None, 1, 1, 28)	18844	block5c_expand_conv[0][0]	
block5c_se_expand (Conv2D)	(None, 1, 1, 672)	19488	block5c_expand_bn[0][0]	
block5c_se_excite (Multiply)	(None, 14, 14, 672)	0	block5c_expand_activation[0][0]	
block5c_project_conv (Conv2D)	(None, 14, 14, 112)	75264	block5c_dwconv[0][0]	
block5c_project_bn (BatchNormalization)	(None, 14, 14, 112)	448	block5c_bn[0][0]	
block5c_drop (Dropout)	(None, 14, 14, 112)	0	block5c_activation[0][0]	
block5c_add (Add)	(None, 14, 14, 112)	0	block5c_se_squeeze[0][0]	
block6a_expand_conv (Conv2D)	(None, 14, 14, 672)	75264	block5c_se_reshape[0][0]	
block6a_expand_bn (BatchNormalization)	(None, 14, 14, 672)	2688	block5c_se_reduce[0][0]	
block6a_expand_activation (Activation)	(None, 14, 14, 672)	0	block5c_activation[0][0]	
block6a_dwconv_pad (ZeroPadding2D)	(None, 17, 17, 672)	0	block5c_se_expand[0][0]	
block6a_dwconv (DepthwiseConv2D)	(None, 7, 7, 672)	16800	block5c_se_excite[0][0]	
block6a_bn (BatchNormalization)	(None, 7, 7, 672)	2688	block5c_project_conv[0][0]	
block6a_activation (Activation)	(None, 7, 7, 672)	0	block5c_project_bn[0][0]	
block6a_se_squeeze (GlobalAveragePooling2D)	(None, 672)	0	block5c_drop[0][0]	
block6a_se_reshape (Reshape)	(None, 1, 1, 672)	0	block5b_add[0][0]	
block6a_se_reduce (Conv2D)	(None, 1, 1, 28)	18844	block5c_add[0][0]	
block6a_se_expand (Conv2D)	(None, 1, 1, 672)	19488	block6a_expand_conv[0][0]	
block6a_se_excite (Multiply)	(None, 7, 7, 672)	0	block6a_expand_bn[0][0]	
block6a_project_conv (Conv2D)	(None, 7, 7, 192)	129024	block6a_expand_activation[0][0]	
block6a_project_bn (BatchNormalization)	(None, 7, 7, 192)	768	block6a_dwconv_pad[0][0]	
block6b_expand_conv (Conv2D)	(None, 7, 7, 1152)	221184	block6a_dwconv[0][0]	
block6b_expand_bn (BatchNormalization)	(None, 7, 7, 1152)	4608	block6a_bn[0][0]	
block6b_expand_activation (Activation)	(None, 7, 7, 1152)	0	block6a_activation[0][0]	
block6b_dwconv (DepthwiseConv2D)	(None, 7, 7, 1152)	28800	block6a_se_squeeze[0][0]	
block6b_bn (BatchNormalization)	(None, 7, 7, 1152)	4608	block6a_se_reshape[0][0]	
block6b_activation (Activation)	(None, 7, 7, 1152)	0	block6a_se_reduce[0][0]	
block6b_se_squeeze (GlobalAveragePooling2D)	(None, 1152)	0	block6a_activation[0][0]	
block6b_se_reshape (Reshape)	(None, 1, 1, 1152)	0	block6a_se_expand[0][0]	
block6b_se_reduce (Conv2D)	(None, 1, 1, 48)	55344	block6a_se_excite[0][0]	
block6b_se_expand (Conv2D)	(None, 1, 1, 1152)	56448	block6a_project_conv[0][0]	
block6b_se_excite (Multiply)	(None, 7, 7, 1152)	0	block6a_project_bn[0][0]	
block6b_project_conv (Conv2D)	(None, 7, 7, 192)	221184	block6b_expand_conv[0][0]	
block6b_project_bn (BatchNormalization)	(None, 7, 7, 192)	768	block6b_expand_bn[0][0]	

block6b_drop (Dropout)	(None, 7, 7, 192)	0	block6b_project_bn[0][0]	
block6b_add (Add)	(None, 7, 7, 192)	0	block6b_drop[0][0]	
block6c_expand_conv (Conv2D)	(None, 7, 7, 1152)	221184	block6a_project_bn[0][0]	
block6c_expand_bn (BatchNormali	(None, 7, 7, 1152)	4608	block6b_add[0][0]	
block6c_expand_activation (Acti	(None, 7, 7, 1152)	0	block6c_expand_conv[0][0]	
block6c_dwconv (DepthwiseConv2D	(None, 7, 7, 1152)	28800	block6c_expand_bn[0][0]	
block6c_bn (BatchNormalization)	(None, 7, 7, 1152)	4608	block6c_expand_activation[0][0]	
block6c_activation (Activation)	(None, 7, 7, 1152)	0	block6c_dwconv[0][0]	
block6c_se_squeeze (GlobalAvera	(None, 1152)	0	block6c_bn[0][0]	
block6c_se_reshape (Reshape)	(None, 1, 1, 1152)	0	block6c_activation[0][0]	
block6c_se_reduce (Conv2D)	(None, 1, 1, 48)	55344	block6c_se_squeeze[0][0]	
block6c_se_expand (Conv2D)	(None, 1, 1, 1152)	56448	block6c_se_reshape[0][0]	
block6c_se_excite (Multiply)	(None, 7, 7, 1152)	0	block6c_se_reduce[0][0]	
block6c_project_conv (Conv2D)	(None, 7, 7, 192)	221184	block6c_activation[0][0]	
block6c_project_bn (BatchNormal	(None, 7, 7, 192)	768	block6c_se_expand[0][0]	
block6c_drop (Dropout)	(None, 7, 7, 192)	0	block6c_se_excite[0][0]	
block6c_add (Add)	(None, 7, 7, 192)	0	block6c_project_conv[0][0]	
block6d_expand_conv (Conv2D)	(None, 7, 7, 1152)	221184	block6c_project_bn[0][0]	
block6d_expand_bn (BatchNormali	(None, 7, 7, 1152)	4608	block6c_drop[0][0]	
block6d_expand_activation (Acti	(None, 7, 7, 1152)	0	block6b_add[0][0]	
block6d_dwconv (DepthwiseConv2D	(None, 7, 7, 1152)	28800	block6c_add[0][0]	
block6d_bn (BatchNormalization)	(None, 7, 7, 1152)	4608	block6d_expand_conv[0][0]	
block6d_activation (Activation)	(None, 7, 7, 1152)	0	block6d_expand_bn[0][0]	
block6d_se_squeeze (GlobalAvera	(None, 1152)	0	block6d_expand_activation[0][0]	
block6d_se_reshape (Reshape)	(None, 1, 1, 1152)	0	block6d_dwconv[0][0]	
block6d_se_reduce (Conv2D)	(None, 1, 1, 48)	55344	block6d_bn[0][0]	
block6d_se_expand (Conv2D)	(None, 1, 1, 1152)	56448	block6d_activation[0][0]	
block6d_se_excite (Multiply)	(None, 7, 7, 1152)	0	block6d_se_squeeze[0][0]	
block6d_project_conv (Conv2D)	(None, 7, 7, 192)	221184	block6d_se_reshape[0][0]	
block6d_project_bn (BatchNormal	(None, 7, 7, 192)	768	block6d_se_reduce[0][0]	
block6d_drop (Dropout)	(None, 7, 7, 192)	0	block6d_activation[0][0]	
block6d_add (Add)	(None, 7, 7, 192)	0	block6d_se_expand[0][0]	
block7a_expand_conv (Conv2D)	(None, 7, 7, 1152)	221184	block6d_se_excite[0][0]	
block7a_expand_bn (BatchNormali	(None, 7, 7, 1152)	4608	block6d_project_conv[0][0]	
block7a_expand_activation (Acti	(None, 7, 7, 1152)	0	block6d_project_bn[0][0]	
block7a_dwconv (DepthwiseConv2D	(None, 7, 7, 1152)	10368	block6d_drop[0][0]	
block7a_bn (BatchNormalization)	(None, 7, 7, 1152)	4608	block6c_add[0][0]	
block7a_activation (Activation)	(None, 7, 7, 1152)	0	block6d_add[0][0]	
block7a_se_squeeze (GlobalAvera	(None, 1152)	0	block7a_expand_conv[0][0]	
block7a_se_reshape (Reshape)	(None, 1, 1, 1152)	0	block7a_expand_bn[0][0]	
block7a_se_reduce (Conv2D)	(None, 1, 1, 48)	55344	block7a_expand_activation[0][0]	
block7a_se_expand (Conv2D)	(None, 1, 1, 1152)	56448	block7a_dwconv[0][0]	
block7a_se_excite (Multiply)	(None, 7, 7, 1152)	0	block7a_bn[0][0]	
block7a_project_conv (Conv2D)	(None, 7, 7, 320)	368640	block7a_activation[0][0]	
block7a_project_bn (BatchNormal	(None, 7, 7, 320)	1280	block7a_se_squeeze[0][0]	
top_conv (Conv2D)	(None, 7, 7, 1280)	409600	block7a_se_reshape[0][0]	
top_bn (BatchNormalization)	(None, 7, 7, 1280)	5120	block7a_se_reduce[0][0]	
top_activation (Activation)	(None, 7, 7, 1280)	0	block7a_activation[0][0]	
avg_pool (GlobalAveragePooling2	(None, 1280)	0	block7a_se_expand[0][0]	
top_dropout (Dropout)	(None, 1280)	0	block7a_se_excite[0][0]	
predictions (Dense)	(None, 1000)	1281000	block7a_project_conv[0][0]	
			block7a_project_bn[0][0]	
			top_conv[0][0]	
			top_bn[0][0]	// Ultima camada que aparece com ou sem topo
			top_activation[0][0]	// Desta camada adiante, so aparece com topo
			avg_pool[0][0]	
			top_dropout[0][0]	

=====

Total params: 5,330,571  
Trainable params: 5,288,548  
Non-trainable params: 42,023

## Anexo E: Regularização L1 e L2

A regularização L1 e L2 faz com que backpropagation dê preferência às redes com pesos pequenos em valor absoluto, fazendo a rede aprender modelos “simples”. Isto ajuda a diminuir “overfitting”. Seja uma rede neural com pesos  $w$  e vieses  $b$ . As normas L1 e L2 do vetor  $w$  são:

$$\|w\|_1 = |w_1| + |w_2| + \dots + |w_n| \quad \text{e} \quad \|w\|_2 = \sqrt{|w_1|^2 + |w_2|^2 + \dots + |w_n|^2}$$

A predição  $\hat{y}$  da entrada  $x$  pela rede neural é:

$$\hat{y} = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$$

A função perda ou custo sem regularização é:

$$\text{Loss} = \text{Error}(y, \hat{y})$$

A função de perda com regularização L1 é:

$$\text{Loss} = \text{Error}(y, \hat{y}) + \lambda \sum_{i=1}^n |w_i|$$

A função de perda com regularização L2 é:

$$\text{Loss} = \text{Error}(y, \hat{y}) + \lambda \sum_{i=1}^n w_i^2$$

Na verdade, aqui está minimizando norma L2 ao quadrado.

Onde  $\lambda > 0$  é o parâmetro de regularização que deve ser escolhido manualmente. Isto faz com que retropropagação “prefira” redes com pesos com valores absolutos pequenos, pois a função perda será menor nessas redes.

Veja, por exemplo, [<https://towardsdatascience.com/intuitions-on-l1-and-l2-regularisation-235f2db4c261>] para maiores detalhes.