Pontifícia Universidade Católica
do Rio de Janeiro

Homework 2
ELE2765 - Deep Learning

Thiago Matheus Bruno da Silva - 1413286

2020
Pontifícia Universidade Católica do Rio de Janeiro

1 Objetivo

O objetivo do trabalho foi efetuar transfer learning em uma rede CNN classificadora de imagens utilizando como backbone a Resnet50v1. Assim, foi pedido que fossem efetuadas 3 diferentes abordagens de fine tunning, descogelando camadas anteriores e adicionando novas à rede. Abaixo, estão os resultados de cada abordagem com os hiperparâmetros utilizados.

2 Resultados

2.1 Questão 5

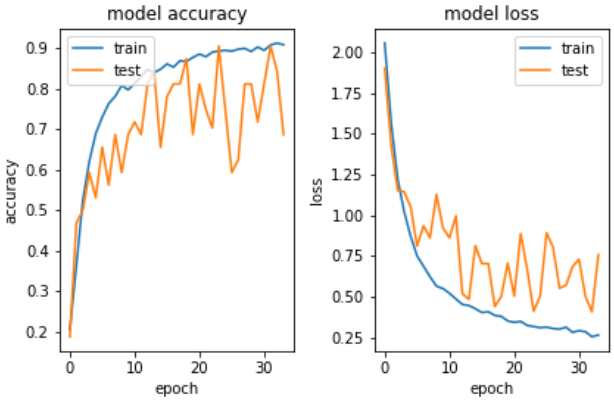

Figura 1: Modelo 1

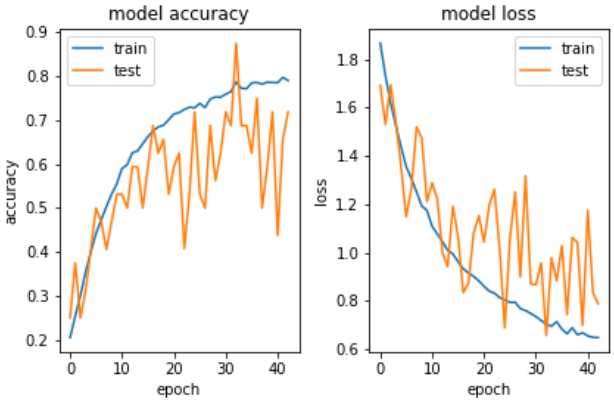

Figura 2: Modelo 2

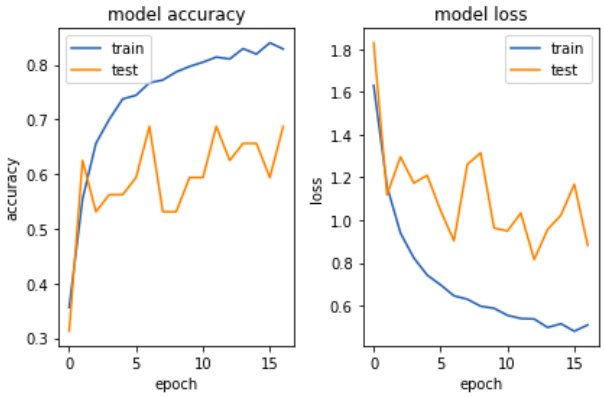

Figura 3: Modelo 3

2.2 Questão 6

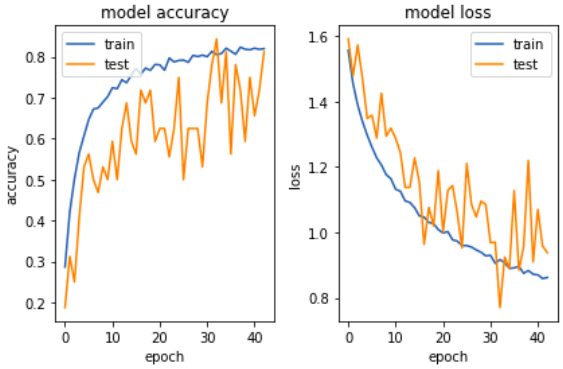

Figura 4: Modelo 4

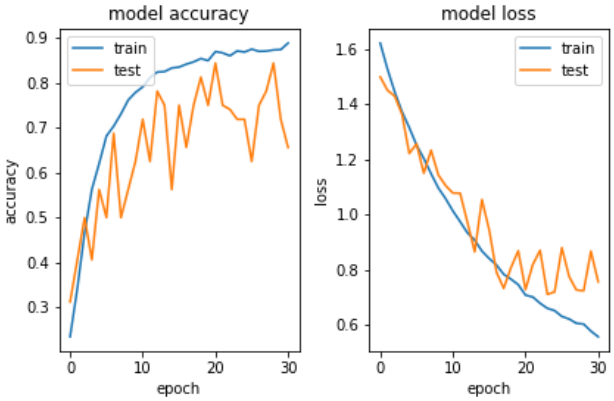

Figura 5: Modelo 5

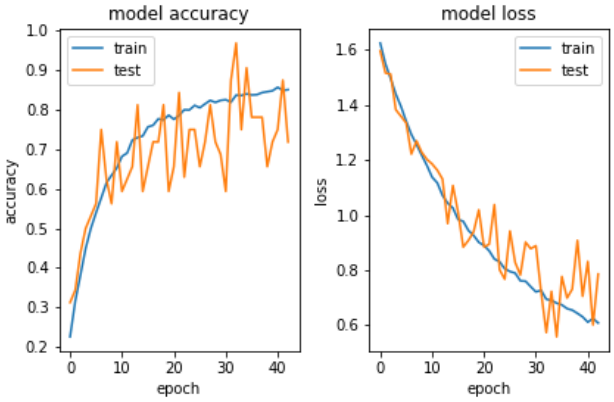

Figura 6: Modelo 6

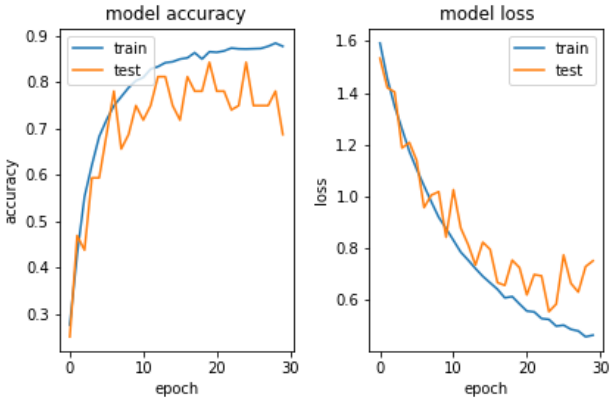

Figura 7: Modelo 7

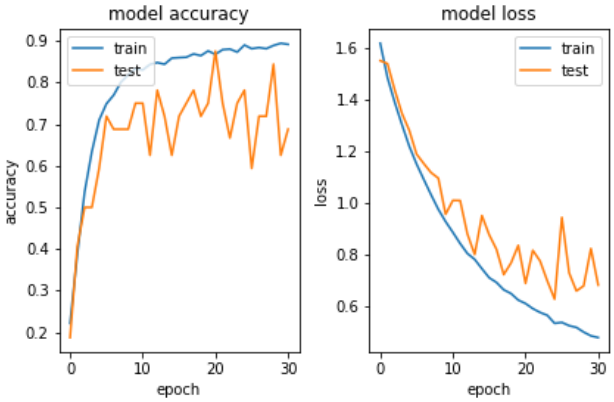

Figura 8: Modelo 8

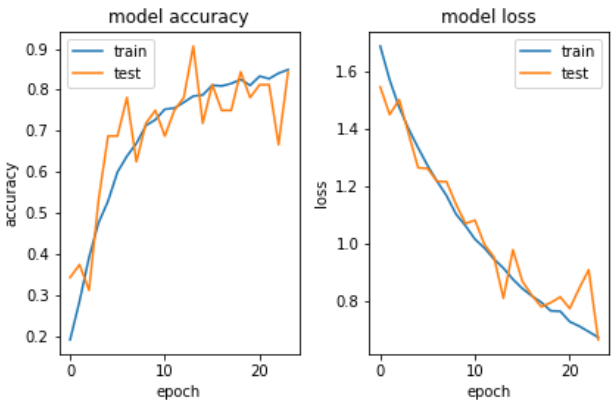

Figura 9: Modelo 9

2.3 Questão 7

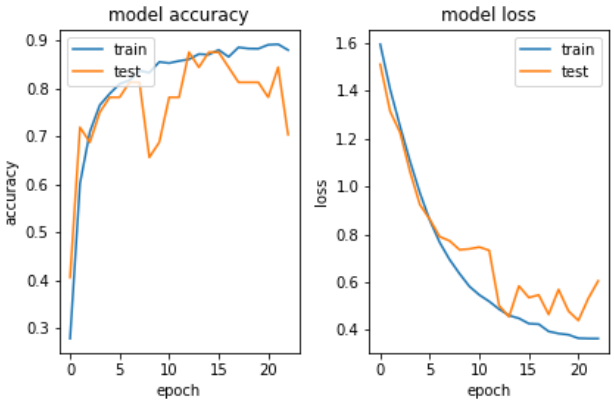

Figura 10: Modelo 10

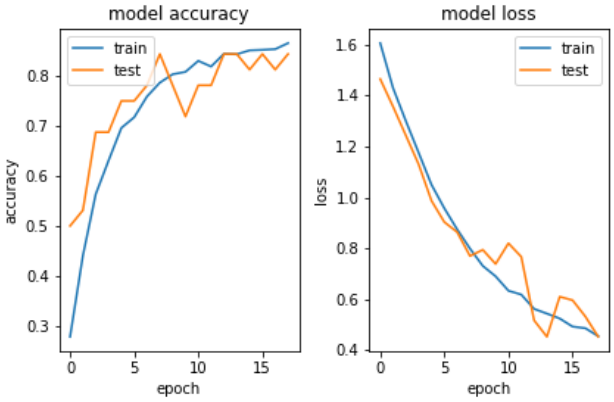

Figura 11: Modelo 11

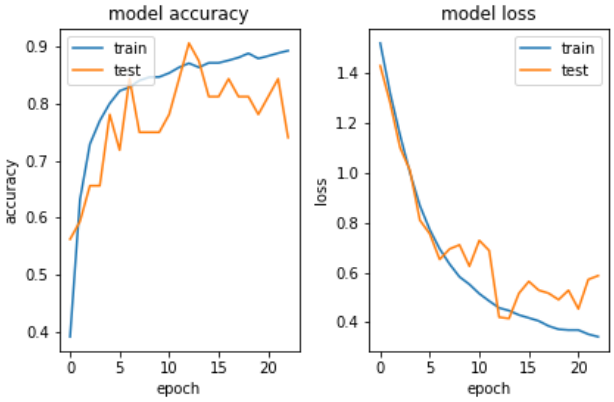

Figura 12: Modelo 12

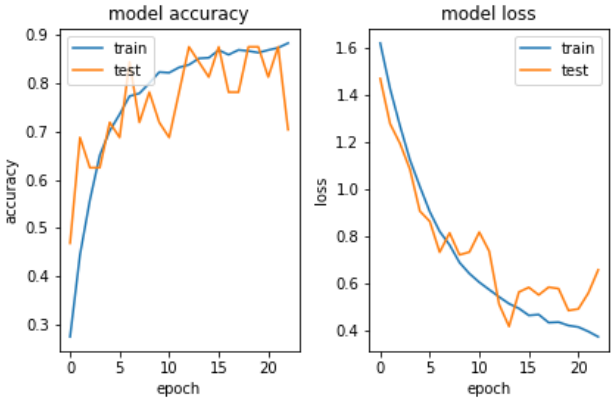

Figura 13: Modelo 13

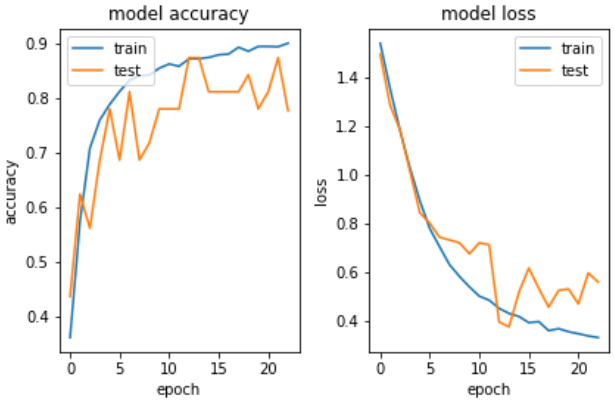

Figura 14: Modelo 14

2.4 Hiperparâmetros

Modelos/Hiperparâmetros	Learning rate	Decay	Dropout	1st dense layer	2nd dense layer	3rd dense layer	Test Loss	Test Accuracy
Modelo 1	5,00E-06	0,001	0.25				0,52999	0,85143
Modelo 2	1E-06	0,001	0				1,26781	0,70857
Modelo 3	1,00E-05	0,01	0				0,86317	0,68000
Modelo 4	5,00E-06	0,001	0.25	512	512	512	1,34390	0,76571
Modelo 5	1,00E-06	0,001	0	512	1024	1024	0,98356	0,77143
Modelo 6	1,00E-06	0,001	0,25	512	1024	1024	0,93507	0,80571
Modelo 7	1,00E-06	0,001	0,25	1024	2048	2048	0,85349	0,86857
Modelo 8	1,00E-06	0,001	0	1024	1024	1024	0,83125	0,86286
Modelo 9	1,00E-06	0,001	0,25	1024	1024	1024	1,21581	0,78857
Modelo 10	1,00E-06	0,001	0	512	1024	1024	0,52568	0,84571
Modelo 11	1,00E-06	0,001	0,25	512	1024	1024	0,87438	0,78857
Modelo 12	1,00E-06	0,001	0	1024	1024	1024	0,52680	0,84000
Modelo 13	1,00E-06	0,001	0,25	1024	1024	1024	0,56724	0,82857
Modelo 14	1,00E-06	0,001	0,25	1024	2048	2048	0,42535	0,88571

Figura 15: Modelos com seus respectivos hiperparâmetros

3 Transfer Learning

3.1 Dataset

Antes de estudar as abordagens tomadas e os modelos obtidos, gostaria de especificar o dataset utilizado. Apenas o dataset das flores foi utilizado e nenhum data augmentation foi usada em nenhum teste, pois nenhum resultado estava melhorando significativamente. Além disso, foi necessŕio a criação de um terceiro dataset de testes, devido a pouca quantidade de imagens para treino. Fiz vários treinamentos dividindo o dataset em 60%, 20%, 20%, treino, validação e teste, respectivamente. Os resultados ficaram muito diferentes do esperado para as diferentes abordagens de fine tunning. Portanto, decidi dividr o dataset das flores em 80% para treino e 20% para validação e criei um terceiro dataset de teste contendo 35 imagens para cada classe de imagens pegas do google.

3.2 Comentários

Em praticamente todos os modelos utilizei um learning rate baixo e com um decaimento mais suave, uma vez que estamos utilizando pesos pré-treinados. Já que o pesos já estão inicializado e treinamos apenas uma parte da rede, é importante atualizar lentamente as últimas camadas para que aprendam lentamente $high\ level features$.

Na primeira tentativa de fine tunning, questão 5, foi pedido apenas para descongelar o último bloco de convolução da Resnet50. Realizei três experimentos nessa etapa, porém em nenhum deles obtive um resultado muito acurado e terminei com todos os modelos "overfitados". Como podemos ver, somente os modelos 2 e 3 chegam a 80% de acurácia no treino e 70% e 60% no testes, respectivamente. Além disso, suas funções de custo no treino também divergem bastante da validação em 0.8 e 1.2, respectivamente. Essa distância entre as curvas de validação e treino caracterizam um overfitting. Mesmo assim, tentei consertar o problema utilizando uma pequena taxa de 0.25 dropout, resultando no modelo 1. Fica assim evidente que uma certa quantidade de dropout pode ajudar a melhorar o modelo e evitar um pouco de overfitting. Mesmo obtendo uma melhora nas curvas (treino chegando aos 90%), podemos ver que o modelo ainda apresenta uma divergência entre as curvas de treino e validação para acurácia e função de custo. Outro fato importante é o resultado nos testes que se avaliarmos somente os números teríamos um modelo com 85% de acurácia e um erro de 0.5 que é relativamente pequeno se comparado aos outros modelos. Entretanto,

podemos ver que apesar do resultado parecer "bom"o modelo está claramente "overfitado"e não deve ser confiado.

Na segunda abordagem, foi pedido para descongelar um outro bloco convolucional logo anterior ao último e, também, adicionar mais uma camada $fully - connected$ antes da camada $softmax$. Assim, esperamos que com o ajuste de mais uma camada de pesos pré-treinados e o treinamento a partir do zero de camadas de totalmente conectadas melhorem a performance do modelo. Sendo assim comecei os treinamentos utilizando uma quantidade relativamente baixa de neurônios e fui aumentando com o tempo, além de intercalar com tentativas de dropout , aplicada antes de última camada, assim como foi feito na questão anterior. O que podemos ver é que temos uma melhora do modelo 4 ao modelo 6, tanto por parte do resultado nos testes quanto nos gráficos. Sendo assim, dupliquei o número de neurônios em cada camada e gerei o modelo 7. Esperando assim uma melhora na classificação dos $features\ vectors$ gerados pela resnet. Assim como o esperado, os resultados foram bem melhores e o gráficos mais suaves. Entretanto, as curvas de validação parecem "overfitadas"tanto para acurácia quanto para a função de custo, pois divergem um pouco do gráfico de treino. Logo, busquei diminuir um pouco o número de parâmetros e deixei as três camadas com 1024 neurônios resultando nos modelos 8 e 9. O modelo 8 ainda parece bem "overfitado", porém com o uso do dropout parece que o modelo ficou bem ajustado às curvas de treino, porém ainda apresenta um erro relativamente grande nos testes de 0.83 e uma acurácia de 86%. O modelo 9 por sua vez, apresenta-se mais bem comportado. As curvas de treino da acurácia e de perda se apresentam ajustadas as de validação. Apesar de ter uma $loss$ relativamente alta nos testes, vemos uma acurácia de 78%. Isso evidencia de novo, que não podemos confiar apenas no resultados de testes sem olhar as curvas de treino e validação. Além disso, o fine tunning, apresentou uma melhora significativa, com os devidos hiperparâmetros ajustados, em comparação a questão anterior.

Na última abordagem, foi pedido para retirar as 3 últimas camadas convolucionais e substituir por camadas com pesos zerados. Ao implementar essas novas camadas, segui os mesmos números de filtros da original, 512, 512 e 2048 utilizando os filtros 1x1, 3x3 e 1x1, respectivamente. Aplicando essa implementação, vamos ter um número maior ainda de parâmetros e junto com as camadas classificadoras esperamos conseguir um modelo que consiga compreender melhor $high\ level\ feature$, já que estes não são compartilhados por classes diferentes. Assim, como o número de parâmetros mudou pela adição de camadas convolucionais, fiz vários testes para ajustar as camadas de classificação ao número correto de parâmetros, tentando achar uma configuração melhor que a obtida na questão anterior. Podemos ver pelos gráficos que as configurações com muitos e poucos parâmetros, modelo 14 e modelo 10, apresentam um pouco "overfitado"s, mesmo o resultado de ambos sendo bem altos nos testes. Já os outros modelos, apresentam uma boa coerência com as curvas de treino e só começam a "overfitar"no final, quando o modelo é interrompido pelo early stopping. Logo é nítido, que os modelos não "overfitados"da questão 7: 11, 12 e 13 foram melhores do que o modelo 9, o único da questão 6 que não "overfitou". O que mostra uma melhora de desempenho conforme inserimos mais camadas ao final da rede.

4 Conclusão

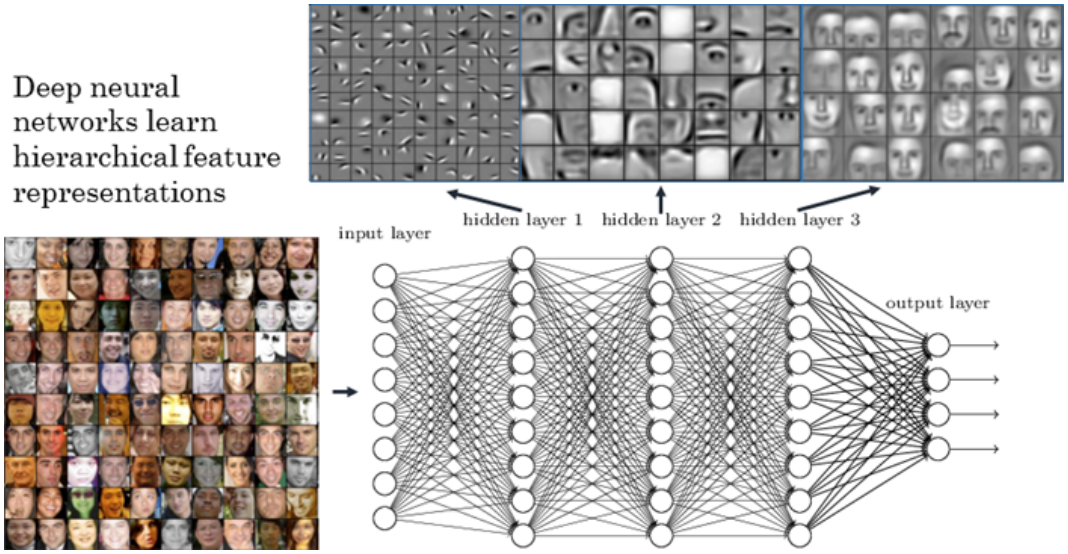

Figura 16: Niveis de textura

Podemos ver que no geral os modelos com camadas novas convolucionais e novas camadas classificatórias apresentam um resultado melhor que os anteriores. Isso ocorre, porque as redes convolucionais começam aprendendo nas primeiras camadas detalhes de textura pequenos, *low features*, e conforme a rede vai se aprofundando ela vai aprendendo características maiores da imagem, *high level features*. Sendo assim, ao mudarmos de domínio, no nosso caso de dataset, essas texturas de nível maior mudam. Isso dificulta a classificação correta da imagem. Como vimos nos testes anteriores, conforme fomos descongelando camadas convolucionais no final da rede os resultados foram melhorando. Não só descongelando, porém, também retreinando do zero essas últimas camadas o que possibilitava um aprendizado mais localizado para extração de características melhores da imagem, as quais "cuspiam"ao final da rede um *feature vector* que melhor representava a imagem de entrada. Ele então, seria classificado pela MLP, logo antes da camada de saída softmax, que seria responsável por classificar esse vetor de texturas ao final das camadas convolucionais. Também vimos, que a inserção dessas multi camadas perceptron também ajudou melhores resultados com essas novas classes que estávamos querendo classificar no nosso dataset e que não havia no dataset original da ImageNet. Além, disso, sabemos que o domínio do dataset de flores não é muito diferente do dataset da ImageNet que possui várias categorias plantas em seu dataset, contribuindo ainda mais para um melhor resultado.

5 Código

train_transfer_learning

April 29, 2020

[ ]:
```
from keras.layers import Input, Dense, Conv2D, MaxPool2D, Flatten, Dropout,␣
↪BatchNormalization, Activation, Add
from keras.optimizers import Adam
from keras.models import Model
from keras.callbacks import ModelCheckpoint, EarlyStopping
import keras
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from keras.preprocessing.image import ImageDataGenerator
from keras.applications.resnet50 import preprocess_input
from keras.applications.resnet import ResNet50 as resnet50

import os
from tqdm import tqdm
import json
import glob
```

[ ]:
```
import tensorflow as tf
gpus= tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(gpus[0], True)
```

[ ]:
```
def resnet50_conv(shape):
    '''
        Load Resnet convolutional layers' and set the input image shape of␣
↪network.

        ================================================================

        input:  A tupe containing 3 dimensions with input shape (H,W,C) RGB.

        returns: Model with convolutional layers only.
    '''
    input_tensor = Input(shape=shape)
    resnet_model_top = resnet50(include_top=True, weights='imagenet',␣
↪input_tensor=input_tensor)
    resnet_model_top.summary()
```

```
    return resnet_model_top
```

[ ]:
```
# Defines input shape and gets resnet50v1 backbone
input_shape = (224,224, 3)
(w, h, _) = input_shape
net = resnet50_conv(input_shape)
```

[ ]:
```
# Helps to see the name and position of each layer.
lay_nb = 0
for layer in net.layers:
    #print(layer)
    print(layer.name)
    #print(layer.output)
    lay_nb += 1
    lay_name = layer.name
#     if layer.name == 'conv4_block6_out':
#         break
    if layer.name == 'avg_pool':
        break
print(lay_nb)
print(lay_name)
```

[ ]:
```
def create_custom_model(net, optm, classes, questao6):
    '''
        Fine tunning implementation according to exercise 5 and 6.

        ================================================================================

        net:  Resnet50 pre-trained model

        optm: Optmizer used to train the network.

        classes: Number of classes in dataset.

        questao6: If wants to implement the Exercise 5 implementation put the␣
 ↪flag to False. Othewise,
        It will implement exercise 6 with custom fully-connected layers and␣
 ↪another convolutional block.


        returns: Model compiled.
    '''

    # Gets the average pooling layer.
    x = net.layers[-2].output
```

```
    # flatten + fully-connected + softmax activation
    if(questao6 == True):
        x = Dense(1024, activation='relu')(x)
        x = Dense(1024, activation='relu')(x)
        x = Dense(1024, activation='relu')(x)
    x = Dropout(0.25)(x)
    x = Dense(classes, activation='softmax')(x)

    custom_model = Model(net.input,x)

    # Froze layers
    for layer in custom_model.layers[:176]:
        if( "conv5_block3" in layer.name or "avg_pool" in layer.name):
            layer.trainable = True
        else:
            layer.trainable = False
        if(questao6 == True):
            if( "conv5_block2" in layer.name):
                layer.trainable = True

    # Compile it
    custom_model.compile(loss='categorical_crossentropy',
                         optimizer=optm,
                         metrics=['accuracy'])

    return custom_model
```

[ ]:
```
def create_custom_model_q7(net, optm, classes):

    '''
        Fine tunning implementation according to exercise 7. convolutional␣
↪layer 5 is implemented from
        scratch. Other layers before convolutional layer 5 are frozen.
        Custom dense layers are implemented also.

        ==========================================================================

        net:  Resnet50 pre-trained model

        optm: Optmizer used to train the network.

        classes: Number of classes in dataset.

        returns: Model compiled.
    '''

    layer_dict = dict([(layer.name, layer) for layer in net.layers])
```

```
    conv4_block6_out = layer_dict['conv4_block6_out'].output

    # Custom Convolutional layers
    # custom_conv_block1
    momemtum=0.8
    activation_f = 'relu'
    # conv_block_1_1
    x = Conv2D(512, (1,1), strides=1, activation='linear',␣
↪padding='same')(conv4_block6_out)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Activation(activation_f)(x)
    # conv_block_1_2
    x = Conv2D(512, (3,3), strides=1, activation='linear', padding='same')(x)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Activation(activation_f)(x)
    # conv_block_1_0
    y = Conv2D(2048, (1,1), strides=1, activation='linear',␣
↪padding='same')(conv4_block6_out)
    y = BatchNormalization(momentum=momemtum)(y)
    # conv_block_1_3
    x = Conv2D(2048, (1,1), strides=1, activation='linear', padding='same')(x)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Add()([y, x])
    custom_conv_block1_out = Activation(activation_f)(x)

    # custom_conv_block2
    x = Conv2D(512, (1,1), strides=1, activation='linear',␣
↪padding='same')(custom_conv_block1_out)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Activation(activation_f)(x)
    x = Conv2D(512, (3,3), strides=1, activation='linear', padding='same')(x)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Activation(activation_f)(x)
    x = Conv2D(2048, (1,1), strides=1, activation='linear', padding='same')(x)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Add()([custom_conv_block1_out, x])
    custom_conv_block2_out = Activation(activation_f)(x)

    # custom_conv_block3
    x = Conv2D(512, (1,1), strides=1, activation='linear',␣
↪padding='same')(custom_conv_block2_out)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Activation(activation_f)(x)
    x = Conv2D(512, (3,3), strides=1, activation='linear', padding='same')(x)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Activation(activation_f)(x)
```

```
    x = Conv2D(2048, (1,1), strides=1, activation='linear', padding='same')(x)
    x = BatchNormalization(momentum=momemtum)(x)
    x = Add()([custom_conv_block2_out, x])
    custom_conv_block3_out = Activation(activation_f)(x)

    #Global Aerage Pooling
    x = layer_dict['avg_pool'](custom_conv_block3_out)

    # custom fully-connected + softmax activation
    x = Dense(1024, activation='relu')(x)
    x = Dense(1024, activation='relu')(x)
    x = Dense(1024, activation='relu')(x)
    x = Dropout(0.25)(x)
    x = Dense(classes, activation='softmax')(x)

    custom_model = Model(net.input,x)

    # Froze layers
    for layer in custom_model.layers[:143]:
            layer.trainable = False


    # Compile it
    custom_model.compile(loss='categorical_crossentropy',
                         optimizer=optm,
                         metrics=['accuracy'])

    return custom_model
```

[ ]:
```
lr = 0.5e-5
optm = Adam(learning_rate = lr, decay=0.001)
classes = 5
#custom_resnet =␣
 ↪create_custom_model(net,optm,lay_name,lay_nb,classes,questao6=False)
custom_resnet = create_custom_model_q7(net,optm,lay_name,lay_nb,classes)
```

[ ]:
```
custom_resnet.summary()
```

[ ]:
```
train_datagen = ImageDataGenerator(validation_split=0.2,␣
 ↪preprocessing_function=preprocess_input) # set validation split
```

[ ]:
```
#Defines the batch-size and keras generators
batch_size = 32

train_data_dir = 'Dataset/flower_photos/all'
```

```
train_generator = train_datagen.flow_from_directory(
    train_data_dir,
    target_size=(w,h),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=True, seed=42,
    subset='training') # set as training data

val_generator = train_datagen.flow_from_directory(
    train_data_dir, # same directory as training data
    target_size=(w,h),
    batch_size=batch_size,
    class_mode='categorical',
    shuffle=True, seed=42,
    subset='validation') # set as validation data
```

[ ]:
```
# Gets the total number of images in dataset
filenames = train_generator.filenames
samples = len(filenames)
print(samples)
```

[ ]:
```
# Defines Early Stopping and sabe the best model during training before␣
 ↪overfitting.
file_name = 'best_model.h5'
checkpointer = ModelCheckpoint(file_name, monitor='val_accuracy',␣
 ↪save_best_only=True)
early_stop = EarlyStopping(monitor = 'val_accuracy', min_delta = 0.001,
mode = 'max', patience = 10)
callbacks=[checkpointer,early_stop]
```

[ ]:
```
# Defines number of epochs and train the model
epochs = 100
steps_in_epoch = samples // batch_size

history = custom_resnet.fit_generator(train_generator,␣
 ↪steps_per_epoch=steps_in_epoch, epochs=epochs,
                                validation_data=val_generator, ␣
 ↪validation_steps=1,
                                verbose=1,callbacks=callbacks)
```

[ ]:
```
def graph_training_history(history):
    acc_train = history['accuracy']
    acc_test = history['val_accuracy']
    loss_train = history['loss']
    loss_test = history['val_loss']
    plt.rcParams['axes.facecolor']='white'
    fig = plt.figure(1)
```

```
    # summarize history for accuracy
    plt.subplot(121)
    plt.plot(acc_train)
    plt.plot(acc_test)
    plt.title('model accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper left')
    plt.tight_layout()

    # summarize history for loss
    plt.subplot(122)
    plt.plot(loss_train)
    plt.plot(loss_test)
    plt.title('model loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train', 'test'], loc='upper right')
    plt.tight_layout()

    plt.show()
    fig.savefig('weights/history.png', dpi=fig.dpi)
```

[ ]:
```
graph_training_history(history.history)
```

[ ]:
```
# Saving history
with open('weights/history_model.json', 'w') as f:
    json.dump(str(custom_resnet.history.history), f)
```

[ ]:
```

```