Everything you need to know about

Great Learning · Follow
17 min read · Sep 23, 2021

Author: Rohini G

Introduction

This blog will give you an insight into VGG16 architecture and explain the same using a use-case for object detection.

ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) is an annual event to showcase and challenge computer vision models. In the 2014 ImageNet challenge, Karen Simonyan & Andrew Zisserman from Visual Geometry Group, Department of Engineering Science, University of Oxford showcased their model in the paper titled "VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION," which won the 1st and 2nd place in object detection and classification. The original paper can be downloaded from the below link:

1409.1556.pdf (arxiv.org)

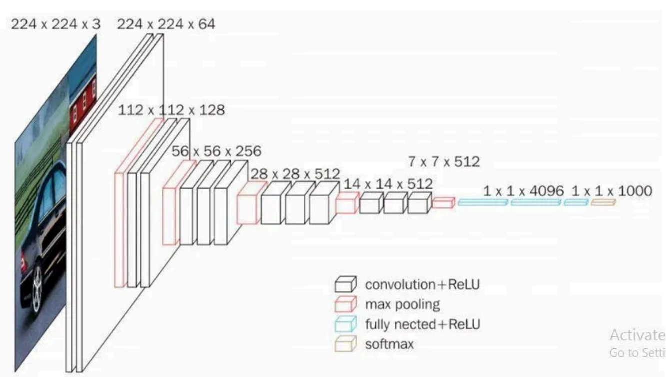
What is VGG16

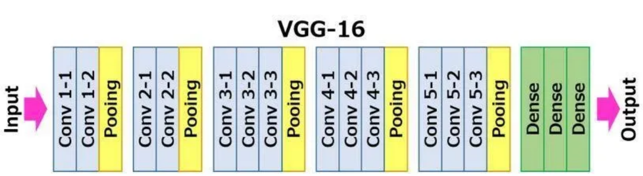
A convolutional neural network is also known as a ConvNet, which is a kind of artificial neural network. A convolutional neural network has an input layer, an output layer, and various hidden layers. VGG16 is a type of CNN (Convolutional Neural Network) that is considered to be one of the best computer vision models to date. The creators of this model evaluated the networks and increased the depth using an architecture with very small (3 × 3) convolution filters, which showed a significant improvement on the prior-art configurations. They pushed the depth to 16–19 weight layers making it approx — 138 trainable parameters.

What is VGG16 used for

VGG16 is object detection and classification algorithm which is able to classify 1000 images of 1000 different categories with 92.7% accuracy. It is one of the popular algorithms for image classification and is easy to use with transfer learning.

VGG16 Architecture





- The 16 in VGG16 refers to 16 layers that have weights. In VGG16 there are thirteen convolutional layers, five Max Pooling layers, and three Dense layers which sum up to 21 layers but it has only sixteen weight layers i.e., learnable parameters layer.
- VGG16 takes input tensor size as 224, 244 with 3 RGB channel
- Most unique thing about VGG16 is that instead of having a large number of hyper-parameters they focused on having convolution layers of 3x3 filter with stride 1 and always used the same padding and maxpool layer of 2x2 filter of stride 2.

- The convolution and max pool layers are consistently arranged throughout the whole architecture
- Conv-1 Layer has 64 number of filters, Conv-2 has 128 filters, Conv-3 has 256 filters, Conv 4 and Conv 5 has 512 filters.
- Three Fully-Connected (FC) layers follow a stack of convolutional layers: the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer.

Train vgg16 from scratch

Let us use the Stanford cars dataset as a use case for object detection and classification. You can download the dataset from the link below.

https://ai.stanford.edu/~jkrause/cars/car_dataset.html

Once you have downloaded the images then we can proceed with the steps written below.

Import the necessary libraries

%tensorflow_version 2.x

import tensorflow as tf

import cv2

import numpy as np

import os

from keras. preprocessing. Image import ImageDataGenerator

from keras.layers import Dense,Flatten,Conv2D,Activation,Dropout

from keras import backend as K

import keras

from keras.models import Sequential, Model

from keras.models import load_model

from keras.optimizers import SGD

from keras.callbacks import EarlyStopping,ModelCheckpoint

from keras.layers import MaxPool2D

from google.colab.patches import cv2_imshow

Detailed EDA (exploratory data analysis) has to be performed under the dataset. Check the image size/dimension and see if there are any images with too big or too small sizes that are outliers. Check other image-related EDA attributes like blurriness, whiteness, aspect ratio, etc.

Import the dataset and normalize the data to make it suitable for the VGG16 model to understand. The Stanford car dataset has cars of various sizes, pixel values, and dimensions. We change the image input tensor to 224, which the

VGG16 model uses. The objective of ImageDataGenerator is to import data with labels easily into the model. It is a very useful class as it has many functions to rescale, rotate, zoom, flip, etc. The most useful thing about this class is that it doesn’t affect the data stored on the disk. This class alters the data on the go while passing it to the model. The ImageDataGenerator will automatically label all the data inside the folder. In this way, data is easily ready to be passed to the neural network.

train_datagen = ImageDataGenerator(zoom_range=0.15,width_shift_range=0.2,height_shift_range=0.2,shear_range=0.15)

test_datagen = ImageDataGenerator()

train_generator = train_datagen.flow_from_directory(“/content/drive/MyDrive/Rohini_Capstone/Car Images/Train Images”,target_size=(224, 224),batch_size=32,shuffle=True,class_mode=’categorical’)

test_generator = test_datagen.flow_from_directory(“/content/drive/MyDrive/Rohini_Capstone/Car Images/Test Images/”,target_size=(224,224),batch_size=32,shuffle=False,class_mode=’categorical’)

Define the VGG16 model as sequential model

→ 2 x convolution layer of 64 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 2 x convolution layer of 128 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 3 x convolution layer of 256 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 3 x convolution layer of 512 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

→ 3 x convolution layer of 512 channel of 3x3 kernal and same padding

→ 1 x maxpool layer of 2x2 pool size and stride 2x2

I also add relu (Rectified Linear Unit) activation to each layer so that all the negative values are not passed to the next layer.

def VGG16():

model = Sequential()

model.add(Conv2D(input_shape=(224,224,3),filters=64,kernel_size=(3,3),padding=”same”, activation=”relu”))

model.add(Conv2D(filters=64,kernel_size=(3,3),padding=”same”, activation=”relu”))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=128, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(Conv2D(filters=128, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(Conv2D(filters=256, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2)))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2))) model.add(Conv2D(filters=512, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(Conv2D(filters=512, kernel_size=(3,3), padding=”same”, activation=”relu”))

model.add(MaxPool2D(pool_size=(2,2),strides=(2,2),name=’vgg16'))

model.add(Flatten(name=’flatten’))

model.add(Dense(256, activation=’relu’, name=’fc1'))

model.add(Dense(128, activation=’relu’, name=’fc2'))

model.add(Dense(196, activation=’softmax’, name=’output’))

return model

After creating all the convolution, pass the data to the dense layer so for that we flatten the vector which comes out of the convolutions and add:

→ 1 x Dense layer of 256 units

→ 1 x Dense layer of 128 units

→ 1 x Dense Softmax layer of 2 units

Change the output layer dimension to that of the number of classes in the below line. The Stanford dataset has 196 classes and hence the same is mentioned in the output layer. Also, the activation function Softmax is used as this is an object classification algorithm. The softmax layer will output the value between 0 and 1 based on the confidence of the model that which class the images belongs to.

model.add(Dense(196, activation=’softmax’, name=’output’))

build the model and print the summary to know the structure of the model.

model=VGG16()

model.summary()

Vgg16 = Model(inputs=model.input, outputs=model.get_layer(‘vgg16’).output)

Load the pre-trained weights of the VGG16 model (gg16_weights_tf_dim_ordering_tf_kernels_notop.h5)so that we don’t have to retrain the entire model.

Vgg16.load_weights(“/content/drive/MyDrive/Rohini_Capstone/vgg16_weights_tf_dim_ordering_tf_kernels_notop.h5”)

Set the early stopping , so that we can stop the training if the accuracy of the model reaches the max with that of the previous iterations.

es=EarlyStopping(monitor=’val_accuracy’, mode=’max’, verbose=1, patience=20)

Used the SGD (stochastic gradient descent) as the optimizer (we can also use ADAM) as the optimizer. Loss “Categorical cross-entropy was used as it is a 196 class model. We can use binary cross-entropy if there are only two classes. We specify the learning rate of the optimizer; here, in this case, it is set at 1e-6. If our training is bouncing a lot on epochs, then we need to decrease the learning rate so that we can reach global minima.

Compile the model, the optimizer and loss metrics.

opt = SGD(learning_rate=1e-6, momentum=0.9)

model.compile(loss=”categorical_crossentropy”, optimizer=opt,metrics=[“accuracy”])

The below code will not train the already trained VGG16 model so that we can use the pre-trained weights for classification. This is called transfer learning which is used to save a lot of effort and resources for re-training.

for layer in Vgg16.layers:

layer.trainable = False

for layer in model.layers:

print(layer, layer.trainable)

Output :

<keras.layers.convolutional.Conv2D object at 0x7f7f0cd09490> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb86edcd0> False

<keras.layers.pooling.MaxPooling2D object at 0x7f7eb9557d90> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb952b710> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb9503950> False

<keras.layers.pooling.MaxPooling2D object at 0x7f7f16ce6ed0> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb94f99d0> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb9514150> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb9510b10> False

<keras.layers.pooling.MaxPooling2D object at 0x7f7eb950e810> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb951df10> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb94aed90> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb95226d0> False

<keras.layers.pooling.MaxPooling2D object at 0x7f7eb951d710> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb951ac10> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb7047ad0> False

<keras.layers.convolutional.Conv2D object at 0x7f7eb95631d0> False

<keras.layers.pooling.MaxPooling2D object at 0x7f7eb94b3390> False

<keras.layers.core.Flatten object at 0x7f7eb94b6850> True

<keras.layers.core.Dense object at 0x7f7eb950e710> True

<keras.layers.core.Dense object at 0x7f7eb94f4bd0> True

<keras.layers.core.Dense object at 0x7f7eb94b35d0> True

Model checkpoint is created and we can save only the best model so that it can be used again for testing and validation of the model.

mc = ModelCheckpoint('/content/drive/MyDrive/Rohini_Capstone/vgg16_best_model_1.h5', monitor='val_accuracy', mode='max', save_best_only=True)

fit the model with the training and test data generator. We executed the model for 150 epochs.

H = model.fit_generator(train_generator,validation_data=test_generator,epochs=150,verbose=1,callbacks=[mc,es])

Epoch 1/150

255/255 [==============================] — 8155s 32s/step — loss: 0.0916 — accuracy: 0.9725 — val_loss: 2.6485 — val_accuracy: 0.5815

Epoch 2/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0736 — accuracy: 0.9801 — val_loss: 2.6269 — val_accuracy: 0.5834

Epoch 3/150

255/255 [==============================] — 203s 797ms/step — loss: 0.0849 — accuracy: 0.9763 — val_loss: 2.6085 — val_accuracy: 0.5845

Epoch 4/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0778 — accuracy: 0.9786 — val_loss: 2.5922 — val_accuracy: 0.5859

Epoch 5/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0635 — accuracy: 0.9840 — val_loss: 2.5806 — val_accuracy: 0.5867

Epoch 6/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0722 — accuracy: 0.9806 — val_loss: 2.5694 — val_accuracy: 0.5889

Epoch 7/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0668 — accuracy: 0.9802 — val_loss: 2.5613 — val_accuracy: 0.5897

Epoch 8/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0598 — accuracy: 0.9824 — val_loss: 2.5532 — val_accuracy: 0.5900

Epoch 9/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0605 — accuracy: 0.9833 — val_loss: 2.5450 — val_accuracy: 0.5905

Epoch 10/150

255/255 [==============================] — 204s 803ms/step — loss: 0.0724 — accuracy: 0.9824 — val_loss: 2.5381 — val_accuracy: 0.5911

Epoch 11/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0614 — accuracy: 0.9817 — val_loss: 2.5319 — val_accuracy: 0.5925

Epoch 12/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0642 — accuracy: 0.9842 — val_loss: 2.5265 — val_accuracy: 0.5933

Epoch 13/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0589 — accuracy: 0.9853 — val_loss: 2.5220 — val_accuracy: 0.5945

Epoch 14/150

255/255 [==============================] — 205s 803ms/step — loss: 0.0604 — accuracy: 0.9830 — val_loss: 2.5171 — val_accuracy: 0.5948

Epoch 15/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0543 — accuracy: 0.9866 — val_loss: 2.5129 — val_accuracy: 0.5958

Epoch 16/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0638 — accuracy: 0.9830 — val_loss: 2.5094 — val_accuracy: 0.5964

Epoch 17/150

255/255 [==============================] — 204s 799ms/step — loss: 0.0608 — accuracy: 0.9824 — val_loss: 2.5054 — val_accuracy: 0.5969

Epoch 18/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0626 — accuracy: 0.9844 — val_loss: 2.5017 — val_accuracy: 0.5976

Epoch 19/150

255/255 [==============================] — 203s 797ms/step — loss: 0.0558 — accuracy: 0.9847 — val_loss: 2.4989 — val_accuracy: 0.5972

Epoch 20/150

255/255 [==============================] — 203s 795ms/step — loss: 0.0627 — accuracy: 0.9812 — val_loss: 2.4961 — val_accuracy: 0.5979

Epoch 21/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0582 — accuracy: 0.9848 — val_loss: 2.4937 — val_accuracy: 0.5977

Epoch 22/150

255/255 [==============================] — 204s 799ms/step — loss: 0.0474 — accuracy: 0.9902 — val_loss: 2.4911 — val_accuracy: 0.5966

Epoch 23/150

255/255 [==============================] — 203s 795ms/step — loss: 0.0525 — accuracy: 0.9871 — val_loss: 2.4889 — val_accuracy: 0.5973

Epoch 24/150

255/255 [==============================] — 202s 794ms/step — loss: 0.0596 — accuracy: 0.9828 — val_loss: 2.4870 — val_accuracy: 0.5983

Epoch 25/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0520 — accuracy: 0.9869 — val_loss: 2.4852 — val_accuracy: 0.5982

Epoch 26/150

255/255 [==============================] — 203s 795ms/step — loss: 0.0590 — accuracy: 0.9849 — val_loss: 2.4837 — val_accuracy: 0.5988

Epoch 27/150

255/255 [==============================] — 203s 799ms/step — loss: 0.0519 — accuracy: 0.9862 — val_loss: 2.4820 — val_accuracy: 0.5988

Epoch 28/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0552 — accuracy: 0.9857 — val_loss: 2.4805 — val_accuracy: 0.5994

Epoch 29/150

255/255 [==============================] — 203s 799ms/step — loss: 0.0586 — accuracy: 0.9827 — val_loss: 2.4790 — val_accuracy: 0.5998

Epoch 30/150

255/255 [==============================] — 204s 798ms/step — loss: 0.0551 — accuracy: 0.9854 — val_loss: 2.4775 — val_accuracy: 0.6000

Epoch 31/150

255/255 [==============================] — 203s 799ms/step — loss: 0.0597 — accuracy: 0.9829 — val_loss: 2.4754 — val_accuracy: 0.6009

Epoch 32/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0566 — accuracy: 0.9855 — val_loss: 2.4746 — val_accuracy: 0.6013

Epoch 33/150

255/255 [==============================] — 204s 803ms/step — loss: 0.0635 — accuracy: 0.9827 — val_loss: 2.4741 — val_accuracy: 0.6014

Epoch 34/150

255/255 [==============================] — 205s 807ms/step — loss: 0.0568 — accuracy: 0.9848 — val_loss: 2.4726 — val_accuracy: 0.6010

Epoch 35/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0556 — accuracy: 0.9857 — val_loss: 2.4719 — val_accuracy: 0.6010

Epoch 36/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0518 — accuracy: 0.9889 — val_loss: 2.4708 — val_accuracy: 0.6019

Epoch 37/150

255/255 [==============================] — 205s 803ms/step — loss: 0.0440 — accuracy: 0.9905 — val_loss: 2.4704 — val_accuracy: 0.6017

Epoch 38/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0553 — accuracy: 0.9873 — val_loss: 2.4703 — val_accuracy: 0.6025

Epoch 39/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0561 — accuracy: 0.9871 — val_loss: 2.4694 — val_accuracy: 0.6027

Epoch 40/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0560 — accuracy: 0.9853 — val_loss: 2.4685 — val_accuracy: 0.6033

Epoch 41/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0520 — accuracy: 0.9902 — val_loss: 2.4676 — val_accuracy: 0.6039

Epoch 42/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0584 — accuracy: 0.9874 — val_loss: 2.4670 — val_accuracy: 0.6029

Epoch 43/150

255/255 [==============================] — 205s 806ms/step — loss: 0.0517 — accuracy: 0.9850 — val_loss: 2.4659 — val_accuracy: 0.6038

Epoch 44/150

255/255 [==============================] — 203s 797ms/step — loss: 0.0576 — accuracy: 0.9854 — val_loss: 2.4644 — val_accuracy: 0.6033

Epoch 45/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0617 — accuracy: 0.9829 — val_loss: 2.4634 — val_accuracy: 0.6035

Epoch 46/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0560 — accuracy: 0.9849 — val_loss: 2.4625 — val_accuracy: 0.6039

Epoch 47/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0489 — accuracy: 0.9878 — val_loss: 2.4620 — val_accuracy: 0.6042

Epoch 48/150

255/255 [==============================] — 206s 809ms/step — loss: 0.0494 — accuracy: 0.9878 — val_loss: 2.4613 — val_accuracy: 0.6040

Epoch 49/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0451 — accuracy: 0.9883 — val_loss: 2.4609 — val_accuracy: 0.6040

Epoch 50/150

255/255 [==============================] — 206s 810ms/step — loss: 0.0482 — accuracy: 0.9858 — val_loss: 2.4600 — val_accuracy: 0.6042

Epoch 51/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0514 — accuracy: 0.9871 — val_loss: 2.4597 — val_accuracy: 0.6043

Epoch 52/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0520 — accuracy: 0.9857 — val_loss: 2.4592 — val_accuracy: 0.6043

Epoch 53/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0566 — accuracy: 0.9857 — val_loss: 2.4586 — val_accuracy: 0.6040

Epoch 54/150

255/255 [==============================] — 203s 797ms/step — loss: 0.0485 — accuracy: 0.9882 — val_loss: 2.4585 — val_accuracy: 0.6043

Epoch 55/150

255/255 [==============================] — 203s 799ms/step — loss: 0.0545 — accuracy: 0.9855 — val_loss: 2.4581 — val_accuracy: 0.6045

Epoch 56/150

255/255 [==============================] — 206s 810ms/step — loss: 0.0557 — accuracy: 0.9871 — val_loss: 2.4576 — val_accuracy: 0.6045

Epoch 57/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0466 — accuracy: 0.9875 — val_loss: 2.4571 — val_accuracy: 0.6042

Epoch 58/150

255/255 [==============================] — 204s 803ms/step — loss: 0.0537 — accuracy: 0.9847 — val_loss: 2.4565 — val_accuracy: 0.6045

Epoch 59/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0538 — accuracy: 0.9873 — val_loss: 2.4562 — val_accuracy: 0.6045

Epoch 60/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0422 — accuracy: 0.9912 — val_loss: 2.4560 — val_accuracy: 0.6034

Epoch 61/150

255/255 [==============================] — 203s 799ms/step — loss: 0.0521 — accuracy: 0.9878 — val_loss: 2.4556 — val_accuracy: 0.6034

Epoch 62/150

255/255 [==============================] — 203s 797ms/step — loss: 0.0566 — accuracy: 0.9843 — val_loss: 2.4551 — val_accuracy: 0.6035

Epoch 63/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0496 — accuracy: 0.9881 — val_loss: 2.4541 — val_accuracy: 0.6038

Epoch 64/150

255/255 [==============================] — 203s 797ms/step — loss: 0.0489 — accuracy: 0.9855 — val_loss: 2.4543 — val_accuracy: 0.6042

Epoch 65/150

255/255 [==============================] — 204s 799ms/step — loss: 0.0495 — accuracy: 0.9883 — val_loss: 2.4541 — val_accuracy: 0.6037

Epoch 66/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0453 — accuracy: 0.9899 — val_loss: 2.4540 — val_accuracy: 0.6040

Epoch 67/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0470 — accuracy: 0.9879 — val_loss: 2.4532 — val_accuracy: 0.6043

Epoch 68/150

255/255 [==============================] — 203s 796ms/step — loss: 0.0461 — accuracy: 0.9897 — val_loss: 2.4528 — val_accuracy: 0.6040

Epoch 69/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0504 — accuracy: 0.9855 — val_loss: 2.4524 — val_accuracy: 0.6037

Epoch 70/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0497 — accuracy: 0.9880 — val_loss: 2.4519 — val_accuracy: 0.6039

Epoch 71/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0503 — accuracy: 0.9886 — val_loss: 2.4520 — val_accuracy: 0.6040

Epoch 72/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0526 — accuracy: 0.9836 — val_loss: 2.4516 — val_accuracy: 0.6037

Epoch 73/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0598 — accuracy: 0.9843 — val_loss: 2.4512 — val_accuracy: 0.6039

Epoch 74/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0450 — accuracy: 0.9896 — val_loss: 2.4508 — val_accuracy: 0.6047

Epoch 75/150

255/255 [==============================] — 202s 793ms/step — loss: 0.0475 — accuracy: 0.9858 — val_loss: 2.4505 — val_accuracy: 0.6047

Epoch 76/150

255/255 [==============================] — 202s 793ms/step — loss: 0.0556 — accuracy: 0.9846 — val_loss: 2.4498 — val_accuracy: 0.6047

Epoch 77/150

255/255 [==============================] — 203s 797ms/step — loss: 0.0551 — accuracy: 0.9850 — val_loss: 2.4494 — val_accuracy: 0.6047

Epoch 78/150

255/255 [==============================] — 203s 799ms/step — loss: 0.0652 — accuracy: 0.9849 — val_loss: 2.4487 — val_accuracy: 0.6048

Epoch 79/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0444 — accuracy: 0.9889 — val_loss: 2.4490 — val_accuracy: 0.6053

Epoch 80/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0498 — accuracy: 0.9880 — val_loss: 2.4488 — val_accuracy: 0.6053

Epoch 81/150

255/255 [==============================] — 204s 799ms/step — loss: 0.0462 — accuracy: 0.9883 — val_loss: 2.4489 — val_accuracy: 0.6051

Epoch 82/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0502 — accuracy: 0.9877 — val_loss: 2.4485 — val_accuracy: 0.6056

Epoch 83/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0442 — accuracy: 0.9897 — val_loss: 2.4480 — val_accuracy: 0.6059

Epoch 84/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0432 — accuracy: 0.9889 — val_loss: 2.4478 — val_accuracy: 0.6060

Epoch 85/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0528 — accuracy: 0.9854 — val_loss: 2.4472 — val_accuracy: 0.6068

Epoch 86/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0517 — accuracy: 0.9871 — val_loss: 2.4471 — val_accuracy: 0.6064

Epoch 87/150

255/255 [==============================] — 206s 809ms/step — loss: 0.0431 — accuracy: 0.9890 — val_loss: 2.4468 — val_accuracy: 0.6064

Epoch 88/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0466 — accuracy: 0.9894 — val_loss: 2.4464 — val_accuracy: 0.6070

Epoch 89/150

255/255 [==============================] — 204s 799ms/step — loss: 0.0442 — accuracy: 0.9899 — val_loss: 2.4459 — val_accuracy: 0.6064

Epoch 90/150

255/255 [==============================] — 203s 798ms/step — loss: 0.0445 — accuracy: 0.9887 — val_loss: 2.4462 — val_accuracy: 0.6070

Epoch 91/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0498 — accuracy: 0.9864 — val_loss: 2.4464 — val_accuracy: 0.6066

Epoch 92/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0471 — accuracy: 0.9874 — val_loss: 2.4465 — val_accuracy: 0.6070

Epoch 93/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0541 — accuracy: 0.9841 — val_loss: 2.4458 — val_accuracy: 0.6065

Epoch 94/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0527 — accuracy: 0.9880 — val_loss: 2.4458 — val_accuracy: 0.6061

Epoch 95/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0467 — accuracy: 0.9882 — val_loss: 2.4455 — val_accuracy: 0.6063

Epoch 96/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0529 — accuracy: 0.9862 — val_loss: 2.4447 — val_accuracy: 0.6066

Epoch 97/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0438 — accuracy: 0.9889 — val_loss: 2.4443 — val_accuracy: 0.6069

Epoch 98/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0513 — accuracy: 0.9875 — val_loss: 2.4444 — val_accuracy: 0.6068

Epoch 99/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0458 — accuracy: 0.9889 — val_loss: 2.4442 — val_accuracy: 0.6065

Epoch 100/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0538 — accuracy: 0.9865 — val_loss: 2.4443 — val_accuracy: 0.6059

Epoch 101/150

255/255 [==============================] — 206s 808ms/step — loss: 0.0470 — accuracy: 0.9879 — val_loss: 2.4440 — val_accuracy: 0.6073

Epoch 102/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0477 — accuracy: 0.9870 — val_loss: 2.4436 — val_accuracy: 0.6068

Epoch 103/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0458 — accuracy: 0.9884 — val_loss: 2.4432 — val_accuracy: 0.6070

Epoch 104/150

255/255 [==============================] — 204s 799ms/step — loss: 0.0573 — accuracy: 0.9856 — val_loss: 2.4429 — val_accuracy: 0.6070

Epoch 105/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0552 — accuracy: 0.9870 — val_loss: 2.4427 — val_accuracy: 0.6076

Epoch 106/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0456 — accuracy: 0.9884 — val_loss: 2.4429 — val_accuracy: 0.6075

Epoch 107/150

255/255 [==============================] — 205s 803ms/step — loss: 0.0505 — accuracy: 0.9861 — val_loss: 2.4425 — val_accuracy: 0.6083

Epoch 108/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0523 — accuracy: 0.9865 — val_loss: 2.4423 — val_accuracy: 0.6081

Epoch 109/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0516 — accuracy: 0.9880 — val_loss: 2.4424 — val_accuracy: 0.6086

Epoch 110/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0453 — accuracy: 0.9873 — val_loss: 2.4425 — val_accuracy: 0.6085

Epoch 111/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0459 — accuracy: 0.9894 — val_loss: 2.4424 — val_accuracy: 0.6090

Epoch 112/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0503 — accuracy: 0.9857 — val_loss: 2.4423 — val_accuracy: 0.6084

Epoch 113/150

255/255 [==============================] — 205s 807ms/step — loss: 0.0516 — accuracy: 0.9876 — val_loss: 2.4420 — val_accuracy: 0.6090

Epoch 114/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0503 — accuracy: 0.9857 — val_loss: 2.4419 — val_accuracy: 0.6086

Epoch 115/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0453 — accuracy: 0.9869 — val_loss: 2.4417 — val_accuracy: 0.6084

Epoch 116/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0466 — accuracy: 0.9876 — val_loss: 2.4418 — val_accuracy: 0.6081

Epoch 117/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0431 — accuracy: 0.9901 — val_loss: 2.4422 — val_accuracy: 0.6079

Epoch 118/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0515 — accuracy: 0.9881 — val_loss: 2.4422 — val_accuracy: 0.6080

Epoch 119/150

255/255 [==============================] — 204s 802ms/step — loss: 0.0440 — accuracy: 0.9898 — val_loss: 2.4418 — val_accuracy: 0.6084

Epoch 120/150

255/255 [==============================] — 207s 811ms/step — loss: 0.0478 — accuracy: 0.9879 — val_loss: 2.4417 — val_accuracy: 0.6078

Epoch 121/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0465 — accuracy: 0.9876 — val_loss: 2.4415 — val_accuracy: 0.6081

Epoch 122/150

255/255 [==============================] — 205s 803ms/step — loss: 0.0448 — accuracy: 0.9892 — val_loss: 2.4413 — val_accuracy: 0.6081

Epoch 123/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0496 — accuracy: 0.9885 — val_loss: 2.4411 — val_accuracy: 0.6081

Epoch 124/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0480 — accuracy: 0.9892 — val_loss: 2.4406 — val_accuracy: 0.6083

Epoch 125/150

255/255 [==============================] — 204s 801ms/step — loss: 0.0469 — accuracy: 0.9878 — val_loss: 2.4406 — val_accuracy: 0.6081

Epoch 126/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0460 — accuracy: 0.9887 — val_loss: 2.4406 — val_accuracy: 0.6080

Epoch 127/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0457 — accuracy: 0.9876 — val_loss: 2.4408 — val_accuracy: 0.6080

Epoch 128/150

255/255 [==============================] — 204s 800ms/step — loss: 0.0416 — accuracy: 0.9889 — val_loss: 2.4408 — val_accuracy: 0.6080

Epoch 129/150

255/255 [==============================] — 205s 804ms/step — loss: 0.0429 — accuracy: 0.9884 — val_loss: 2.4405 — val_accuracy: 0.6086

Epoch 130/150

255/255 [==============================] — 205s 805ms/step — loss: 0.0510 — accuracy: 0.9878 — val_loss: 2.4408 — val_accuracy: 0.6085

Epoch 131/150

255/255 [==============================] — 204s 799ms/step — loss: 0.0488 — accuracy: 0.9878 — val_loss: 2.4412 — val_accuracy: 0.6085

Epoch 00131: early stopping

The accuracy of 60% was achieved and model stopped due to early learning at 131 epochs.

Once the model is trained, we can also visualize the training/ validation accuracy

import matplotlib.pyplot as plt
plt.plot(model.history[“acc”])

plt.plot(model.history[‘val_acc’])
plt.plot(model.history[‘loss’])
plt.plot(hist.history[‘val_loss’])
plt.title(“model accuracy”)
plt.ylabel(“Accuracy”)
plt.xlabel(“Epoch”)
plt.legend([“Accuracy”,”Validation Accuracy”,”loss”,”Validation Loss”])
plt.show()

To do predictions on the trained model I need to load the best saved model and pre-process the image and pass the image to the model for output.

Use the saved model and load them to evaluate the model.

model.load_weights(“/content/drive/MyDrive/Rohini_Capstone/vgg16_best_model_1.h5”)

model.evaluate_generator(test_generator)

model_json = model.to_json()

with open(“/content/drive/MyDrive/Rohini_Capstone/vgg16_cars_model.json”,”w”) as json_file:

json_file.write(model_json)

from keras.models import model_from_json

run the prediction

def predict_(image_path):

#Load the Model from Json File

json_file =
open('/content/drive/MyDrive/Rohini_Capstone/vgg16_cars_model.json', 'r')

model_json_c = json_file.read()

json_file.close()

model_c = model_from_json(model_json_c)

#Load the weights

model_c.load_weights("/content/drive/MyDrive/Rohini_Capstone/vgg16_best_model.h5")

#Compile the model

opt = SGD(lr=1e-4, momentum=0.9)

model_c.compile(loss="categorical_crossentropy", optimizer=opt,metrics=["accuracy"])

#load the image you want to classify

image = cv2.imread(image_path)

image = cv2.resize(image, (224,224))

cv2_imshow(image)

#predict the image

preds = model_c.predict_classes(np.expand_dims(image, axis=0))[0]

print("Predicted Label",preds)

predict_("/content/drive/MyDrive/Rohini_Capstone/Car Images/Test Images/Rolls-Royce Phantom Sedan 2012/06155.jpg")

Predicted Label 176