VGG16 Transfer Learning Approach Deep Convolutional Neural networks may take days to train and require lots of computational resources. So to overcome this we will use Transfer Learning for implementing VGG16 with Keras.

Transfer learning is a technique whereby a deep neural network model that was trained earlier on a similar problem is leveraged to create a new model at hand. One or more layers from the already trained model are used in the new model. We will go through more details in a subsequent section below.

Define training and testing path of dataset It is necessary to put the images of both classes of dog and cat in separate subfolders under train and test folder

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

Reading sample Images Let us just read a couple of random images for the data set to see what types of images we have. The use of cv2.imshow is disabled in Colab because it causes Jupyter sessions to crash, so as a substitution, we are using cv2_imshow().

```
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 tensorflow.keras.optimizers import SGD
from keras.callbacks import EarlyStopping,ModelCheckpoint
from keras.layers import MaxPool2D
from google.colab.patches import cv2_imshow
```

Preparation of datasets We generally encountered problems where we try to load a dataset but there is not enough memory in your machine.

Keras provides the ImageDataGenerator class that defines the configuration for image data preparation and augmentation. The generator will progressively load the images in your dataset, allowing you to work with both small and very large datasets containing thousands or millions of images that may not fit into system memory.

```
train_path="/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/Train"
test_path="/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/Validation"
class_names=os.listdir(train_path)
class_names_test=os.listdir(test_path)
```

```
print(class_names)
print(class_names_test)
```

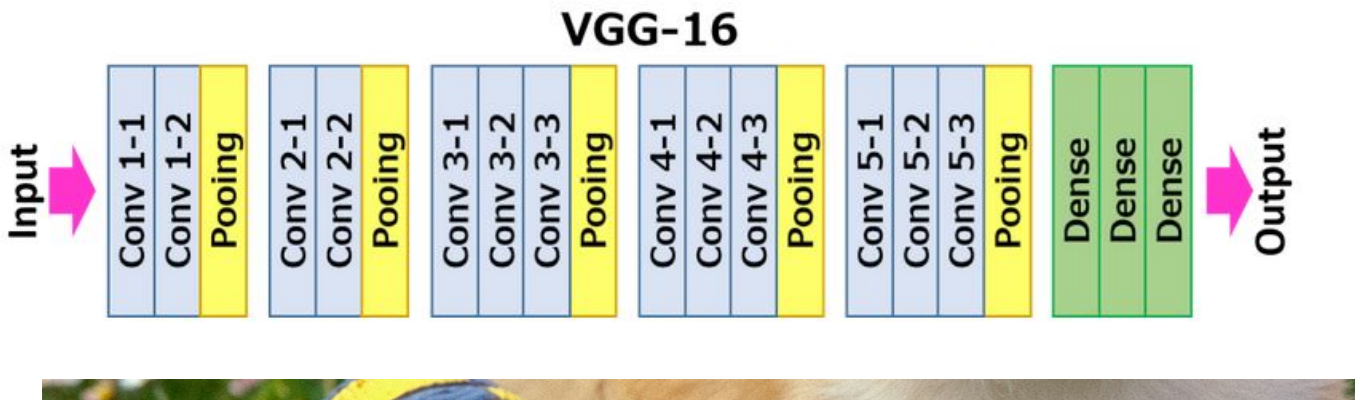
```
['dog', 'cat']
['cat', 'dog']
```

```
#Sample datasets images
image_dog=cv2.imread("/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/Train/dog/Dog-Picture6.jpg")
cv2_imshow(image_dog)
image_cat=cv2.imread("/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/Train/cat/cat001.jpg")
cv2_imshow(image_cat)
```

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

```
train_generator = train_datagen.flow_from_directory("/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/Trai
test_generator = test_datagen.flow_from_directory("/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/Valida
```

```
Found 3 images belonging to 2 classes.
Found 4 images belonging to 2 classes.
```




```
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(1, activation='sigmoid', name='output'))
    return model
```

```
model=VGG16()
```

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 224, 224, 64)      1792

 conv2d_1 (Conv2D)           (None, 224, 224, 64)      36928

 max_pooling2d (MaxPooling2D  (None, 112, 112, 64)     0
 )
```

```
conv2d_2 (Conv2D)           (None, 112, 112, 128)     73856

conv2d_3 (Conv2D)           (None, 112, 112, 128)     147584

max_pooling2d_1 (MaxPooling  (None, 56, 56, 128)      0
2D)

conv2d_4 (Conv2D)           (None, 56, 56, 256)       295168

conv2d_5 (Conv2D)           (None, 56, 56, 256)       590080

conv2d_6 (Conv2D)           (None, 56, 56, 256)       590080

max_pooling2d_2 (MaxPooling  (None, 28, 28, 256)      0
2D)

conv2d_7 (Conv2D)           (None, 28, 28, 512)       1180160

conv2d_8 (Conv2D)           (None, 28, 28, 512)       2359808

conv2d_9 (Conv2D)           (None, 28, 28, 512)       2359808

max_pooling2d_3 (MaxPooling  (None, 14, 14, 512)      0
2D)

conv2d_10 (Conv2D)          (None, 14, 14, 512)       2359808

conv2d_11 (Conv2D)          (None, 14, 14, 512)       2359808

conv2d_12 (Conv2D)          (None, 14, 14, 512)       2359808

vgg16 (MaxPooling2D)        (None, 7, 7, 512)         0

flatten (Flatten)           (None, 25088)             0

fc1 (Dense)                 (None, 256)               6422784

fc2 (Dense)                 (None, 128)               32896

output (Dense)              (None, 1)                 129

=================================================================
Total params: 21,170,497
Trainable params: 21,170,497
Non-trainable params: 0
_________________________________________________________________
```

As we discussed in the above section that we will use Transfer Learning for implementing VGG16 with Keras. So we will reuse the model weights from pre-trained models that were developed for standard computer vision benchmark datasets like ImageNet. We have downloaded pre-trained weights that do not have top layers weights. As you see above we have replaced the last three layers by our own layer and pre-trained weights do not contain the weights of newly three dense layers. So that is why we have to download pre-trained layer without top. (Link to download these weights are given at the bottom of the article) https://drive.google.com/uc?export=download&id=1pjBpzyKE7QmvKZRy5zUFxDXYZQ5F7ow0

Since we only have to initialize the weight to the last convolutional that is why we have called model and pass input as model input and output as the last convolutional block

```
Vgg16 = Model(inputs=model.input, outputs=model.get_layer('vgg16').output)
```

Now load the weights using the function load_weights of Keras

```
Vgg16.load_weights("/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/vgg16_weights_tf_dim_ordering_tf_kerr
```

As we know that initial layers learn very general features and as we go higher up the network, the layers tend to learn patterns more specific to the task it is being trained on. So using these properties of the layer we want to keep the initial layers intact (freeze that layer) and retrain the later layers for our task. This is also termed as finetuning of a network.

The advantage of finetuning is that we do not have to train the entire layer from scratch and hence the amount of data required for training is not much either. Also, parameters that need to be updated are less and hence the amount of time required for training will also be less.

In Keras, each layer has a parameter called “trainable”. For freezing the weights of a particular layer, we should set this parameter to False, indicating that this layer should not be trained. After that, we go over each layer and select which layers we want to train. In our case, we are freezing all the convolutional block of VGG16

```
for layer in Vgg16.layers:
    layer.trainable = False
```

```
for layer in model.layers:
    print(layer, layer.trainable)
```

```
  <keras.layers.convolutional.Conv2D object at 0x7f742add0290> False
  <keras.layers.convolutional.Conv2D object at 0x7f7427579c90> False
  <keras.layers.pooling.MaxPooling2D object at 0x7f742c1fd510> False
  <keras.layers.convolutional.Conv2D object at 0x7f742928db50> False
  <keras.layers.convolutional.Conv2D object at 0x7f7429263f90> False
  <keras.layers.pooling.MaxPooling2D object at 0x7f7429274850> False
  <keras.layers.convolutional.Conv2D object at 0x7f7429272f50> False
  <keras.layers.convolutional.Conv2D object at 0x7f742927add0> False
  <keras.layers.convolutional.Conv2D object at 0x7f7429205710> False
  <keras.layers.pooling.MaxPooling2D object at 0x7f742920b9d0> False
  <keras.layers.convolutional.Conv2D object at 0x7f7429206250> False
  <keras.layers.convolutional.Conv2D object at 0x7f742920e310> False
  <keras.layers.convolutional.Conv2D object at 0x7f742920eed0> False
  <keras.layers.pooling.MaxPooling2D object at 0x7f7429218fd0> False
  <keras.layers.convolutional.Conv2D object at 0x7f742920e2d0> False
  <keras.layers.convolutional.Conv2D object at 0x7f74292137d0> False
  <keras.layers.convolutional.Conv2D object at 0x7f7429230c50> False
  <keras.layers.pooling.MaxPooling2D object at 0x7f742922c290> False
  <keras.layers.core.flatten.Flatten object at 0x7f74292232d0> True
  <keras.layers.core.dense.Dense object at 0x7f7429248ed0> True
  <keras.layers.core.dense.Dense object at 0x7f742921c290> True
  <keras.layers.core.dense.Dense object at 0x7f7429213cd0> True
```

We then compile the VGG16 using the compile function. This function expects three parameters: the optimizer, the loss function, and the metrics of performance. We are going to use stochastic gradient descent as an optimizer. Also since we are doing binary classification, we use binary_crossentropy as the choice for the loss function.

```
opt = SGD(lr=1e-4, momentum=0.9)
model.compile(loss="binary_crossentropy", optimizer=opt,metrics=["accuracy"])
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/gradient_descent.py:102: UserWarning
  super(SGD, self).__init__(name, **kwargs)
```

Early Stoping

A basic problem that arises in training a neural network is to decide how many epochs a model should be trained. Too many epochs may lead to overfitting of the model and too few epochs may lead to underfitting of the model. So to overcome this problem the concept of Early Stoping is used.

In this technique, we can specify an arbitrarily large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset. Keras supports the early stopping of training via a callback called EarlyStopping.

Below are various arguments in EarlyStopping.

monitor – This allows us to specify the performance measure to monitor in order to end training.

mode – It is used to specify whether the objective of the chosen metric is to increase maximize or to minimize.

verbose – To discover the training epoch on which training was stopped, the “verbose” argument can be set to 1. Once stopped, the callback will print the epoch number.

patience – The first sign of no further improvement may not be the best time to stop training. This is because the model may coast into a plateau of no improvement or even get slightly worse before getting much better. We can account for this by adding a delay to the trigger in terms of the number of epochs on which we would like to see no improvement. This can be done by setting the “patience” argument.

```
es=EarlyStopping(monitor='val_accuracy', mode='max', verbose=1, patience=10)
```

Model Check Point

The EarlyStopping callback will stop training once triggered, but the model at the end of training may not be the model with the best performance on the validation dataset.

An additional callback is required that will save the best model observed during training for later use. This is known as the ModelCheckpoint callback.

The ModelCheckpoint callback is flexible in the way it can be used, but in this case, we will use it only to save the best model observed during training as defined by a chosen performance measure on the validation dataset.

```
mc = ModelCheckpoint('/content/gdrive/My Drive/best_model.h5', monitor='val_accuracy', mode='max', save_best_only=Tru
```

```
H = model.fit_generator(train_generator,validation_data=test_generator,epochs=20,verbose=1,callbacks=[mc,es])
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.fit_genera
  """Entry point for launching an IPython kernel.
Epoch 1/20
1/1 [==============================] - 2s 2s/step - loss: 0.0000e+00 - accuracy: 1.0000 - val_
Epoch 2/20
1/1 [==============================] - 1s 808ms/step - loss: 0.0000e+00 - accuracy: 1.0000 - v
Epoch 3/20
```

```
1/1 [==============================] - 1s 586ms/step - loss: 6.3491e-20 - accuracy: 1.0000 - v
Epoch 4/20
1/1 [==============================] - 0s 462ms/step - loss: 0.0000e+00 - accuracy: 1.0000 - v
Epoch 5/20
1/1 [==============================] - 0s 467ms/step - loss: 31.6266 - accuracy: 0.6667 - val_
Epoch 6/20
1/1 [==============================] - 0s 447ms/step - loss: 3.5264e-10 - accuracy: 1.0000 - v
Epoch 7/20
1/1 [==============================] - 0s 463ms/step - loss: 2.0825e-32 - accuracy: 1.0000 - v
Epoch 8/20
1/1 [==============================] - 0s 453ms/step - loss: 1.9261e-13 - accuracy: 1.0000 - v
Epoch 9/20
1/1 [==============================] - 0s 452ms/step - loss: 1.2871e-17 - accuracy: 1.0000 - v
Epoch 10/20
1/1 [==============================] - 0s 457ms/step - loss: 7.7024e-25 - accuracy: 1.0000 - v
Epoch 11/20
1/1 [==============================] - 0s 457ms/step - loss: 0.1363 - accuracy: 1.0000 - val_l
Epoch 12/20
1/1 [==============================] - 0s 466ms/step - loss: 2.0555e-04 - accuracy: 1.0000 - v
Epoch 12: early stopping
```

As we can see here that if we set epochs=100 the model training got a an early stopping as per what we had expected. The weights are saved in a file “best_model.h5“. In the future, we are not required to train the model again we can just load the weight into the model with below command.

```
model.load_weights("/content/gdrive/My Drive/best_model.h5")
```

Evaluating the model on test datasets Let us now evaluate the performance of our model on the unseen testing data set. We can see an accuracy of 75%.

```
model.evaluate_generator(test_generator)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: UserWarning: `Model.evaluate_g
  """Entry point for launching an IPython kernel.
[22.041967391967773, 0.75]
```

Serialize the Keras Model

It is the best practice of converting the model into JSON format to save it for the inference program in the future.

```
model_json = model.to_json()
with open("/content/gdrive/My Drive/model.json","w") as json_file:
  json_file.write(model_json)
```

Dogs Vs Cat Classification Inference Program

To recap what we have done till now –

We implemented the VGG16 model with Keras.

We saved the best training weights of the model in a file for future use.

We saved the model in JSON format for reusability.

Now it is time to write an inference program that will do the following –

Load the model that we saved in JSON format earlier.

Load the weight that we saved after training the model earlier.

Compile the model.

Load the image that we want to classify.

Perform classification.

```
from keras.models import model_from_json


def predict_(image_path):
    #Load the Model from Json File
    json_file = open('/content/gdrive/My Drive/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/gdrive/My Drive/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(np.expand_dims(image, axis=0))[0]
    if preds==0:
        print("Predicted Label:Cat")
    else:
        print("Predicted Label: Dog")
```

Perform Classification

```
predict_("/content/gdrive/MyDrive/Transfer Learning/Transfer Learning/train/dog/dog001.jpg")
```

```
predict_("/content/gdrive/MyDrive/AIML/CNN NEXT HALF/5XTRA/train/cat/cat001.jpg")
```

```
/usr/local/lib/python3.7/dist-packages/keras/optimizer_v2/gradient_descent.py:102: UserWarning
  super(SGD, self).__init__(name, **kwargs)
---------------------------------------------------------------------------
error                                     Traceback (most recent call last)
<ipython-input-48-836c68f1b139> in <module>()
----> 1 predict_("/content/gdrive/MyDrive/AIML/CNN NEXT HALF/5XTRA/train/cat/cat001.jpg")

<ipython-input-46-4af0dd3e81da> in predict_(image_path)
     12     #load the image you want to classify
     13     image = cv2.imread(image_path)
---> 14     image = cv2.resize(image, (224,224))
     15     cv2_imshow(image)
     16     #predict the image

error: OpenCV(4.1.2) /io/opencv/modules/imgproc/src/resize.cpp:3720: error: (-215:Assertion fa
function 'resize'
```

SEARCH STACK OVERFLOW