CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Deep Learning

Deep Learning

Lecture: Convolutional Neural Networks –
Common Architectures
Checkpointing
Data Augmentation
Feature Extraction

Ted Scully

Data Augmentation and Multi-class Classification

1. We can easily adapt the example we saw in the previous slides for multi-class classification.
   a. Firstly we need to change the **loss function** we use for our model from binary (**binary_crossentropy**) to **model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])**

   b. Change the architecture of the CNN so that the final layer is a softmax as opposed to sigmoid unit. For example if dealing with 3 classes then the final component of your CNN would be **model.add(tf.keras.layers.Dense(3, activation='softmax')))**

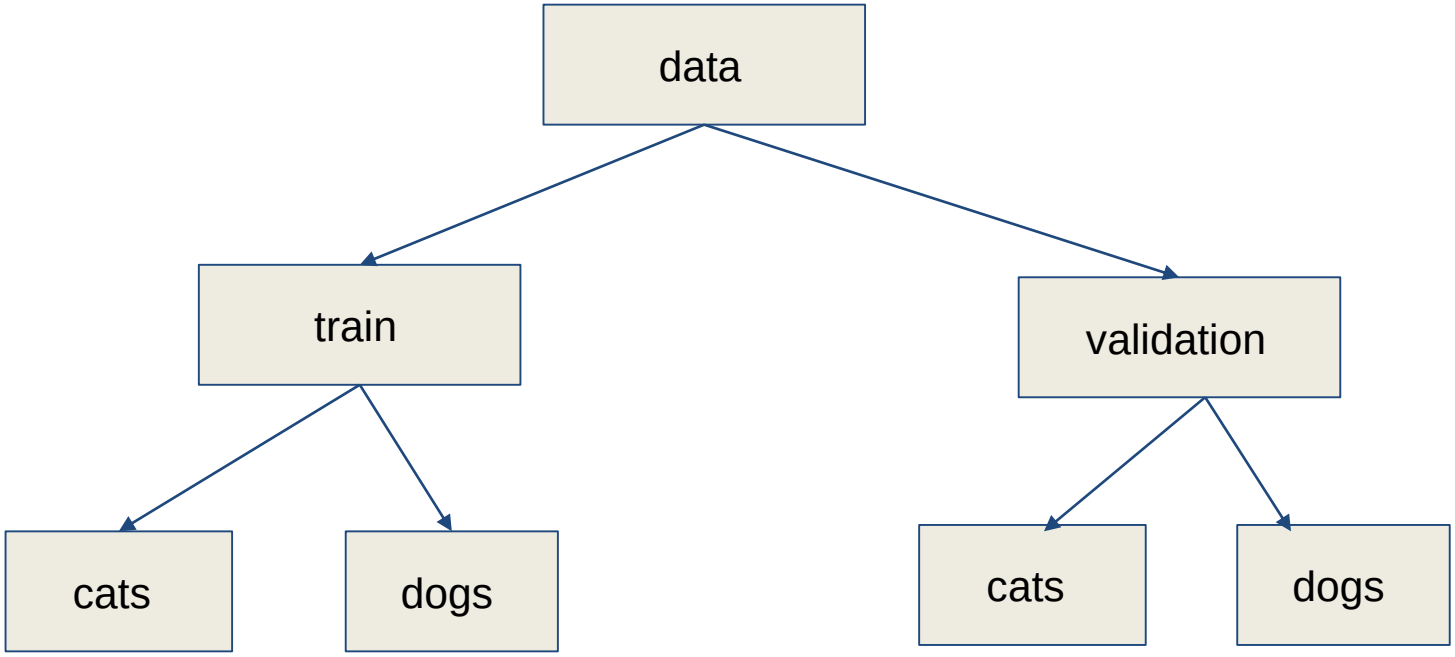
   c. Finally in the code where you create iterator using flow_from_dictionary then you must change the class mode from binary to sparse.

```
train_generator = trainDataGenerator.flow_from_directory(
    trainDataDir,
    target_size=(width, height),
    batch_size=batchSize,
    class_mode='sparse')
```

(Note the above assume the class labels are encoded as integers and not one hot encoded)

Reading Image Data Into NumPy Arrays

- While the flow_from_directory is a convenient method of automatically reading in data and performing augmentation, it can be slow down training because you are reading from disk.
- Instead we may want to **read the image data directly into a NumPy array** and perform basic pre-processing on the data before we pass it to your model.
- In the following example we will look at how to iteratively read image data using opencv along with the associated labels. We again assume the following directory structure.
- You should notice a significant improvement in speed when using this method.



The **label dictionary** will keep track of the total number of classes. For each class which is a string value, the code will insert a key value pair in the dictionary, where the **key is the class string name** and the **value is an int value** (which we will later use to cover all string class labels to integers). The following will be the contents of labelDictionary for our example {'dogs': 0, 'cats': 1}.

The procImage function will take in the location of the training or validation data. It will iterate through each sub folder, cats and dogs in this example. It will then use **openCV to resize the image and store as a NumPy array**. It will then add the NumPy array to a list of image.

Associated Colab Notebook

```
import tensorflow as tf
from conv.convModel import convModel
import matplotlib.pyplot as plt
import numpy as np
import cv2
import os
from glob import glob
import pandas as pd

# we will resize all images to this width and height
width, height= 150, 150
trainDataDir = '/home/ted.scully/Datasets/dogs_cats/train'
validationDataDir= '/home/ted.scully/Datasets/dogs_cats/validation'

NUM_EPOCHS = 60
batchSize = 64


# This dictionary will contain all labels with an associated int value
labelDictionary ={}

# The procImages function will read all image data from a folder
# convert it to a Numpy array and add to a list
# It will also add the label for the image, the folder name

trainImages, trainLabels = procImages(trainDataDir,
labelDictionary)

valImages, valLabels = procImages(validationDataDir,
labelDictionary)
```

Here we iterate through our train and validation folders.

Each time we iterate through a folder we access all images in the folder.

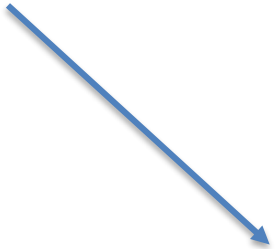
Notice for each image in the image directory we read the image using opencv. We then resize the image and add it to a list.

Associated Colab Notebook

```
def procImages(dataDir, labelDictionary):

    x = [] # will store images as arrays
    y = [] # store labels

    # list folders in directory
    directory = os.listdir(dataDir)

    # for each folder (remember each folder refers to a specific class)
    for label in directory:

        # add class label to label dictionary
        if label not in labelDictionary:
            labelDictionary[label] = len(labelDictionary)

        # create full path for image directory (append absolute path and
        # image directory path)
        sourceImages = os.path.join(dataDir, label)
        images = os.listdir(sourceImages)

        # for each image in directory,
        for image in images:

            # read the image from file, resize and add to a list
            full_size_image = cv2.imread(os.path.join(sourceImages, image))
            x.append(cv2.resize(full_size_image, (width,height),
                                interpolation=cv2.INTER_CUBIC))

            # add the class label to y
            y.append(label)

    return np.array(x), np.array(y)
```

Now returning to the main code we normalize all the data by dividing by 255.

You will notice at the end of the code we use to **labelDictionary** to **map** the training and validation **string labels to integer values**. In other words the trainLabels will no longer contain the words cat and dog and will now be replaces by 0 and 1 values.



```
# Normalize data
trainImages = trainImages.astype("float") / 255.0
valImages = valImages.astype("float") / 255.0

# Map string label values to integer values.
trainLabels = (pd.Series(trainLabels).map(labelDictionary)).values
valLabels = (pd.Series(valLabels).map(labelDictionary)).values
```

Associated Colab Notebook

Notice in the new code we only have a generator for the training data (we have already normalized both the training and test data).

Notice we use the flow method this time, which takes in the training images the corresponding class labels and the batch size.

Finally we call the fit_generator method (as we did previously). Notice this time we manually specify the validation data.

```
model = convModel.build(width, height, 3)

model.compile(loss='binary_crossentropy', optimizer='rmsprop',
              metrics=['accuracy'])

# In this example we just create an ImageDataGenerator for the
training data

trainDataGenerator =
tf.keras.preprocessing.image.ImageDataGenerator(
    shear_range=0.2,
    zoom_range=0.2,
    rotation_range=30,
    horizontal_flip=True)


train_generator = trainDataGenerator.flow(trainImages,
trainLabels, batch_size=32)



H = model.fit(train_generator,
                        validation_data=(valImages, valLabels),
                         steps_per_epoch=len(trainImages)/ batchSize,
                        epochs = NUM_EPOCHS)
```

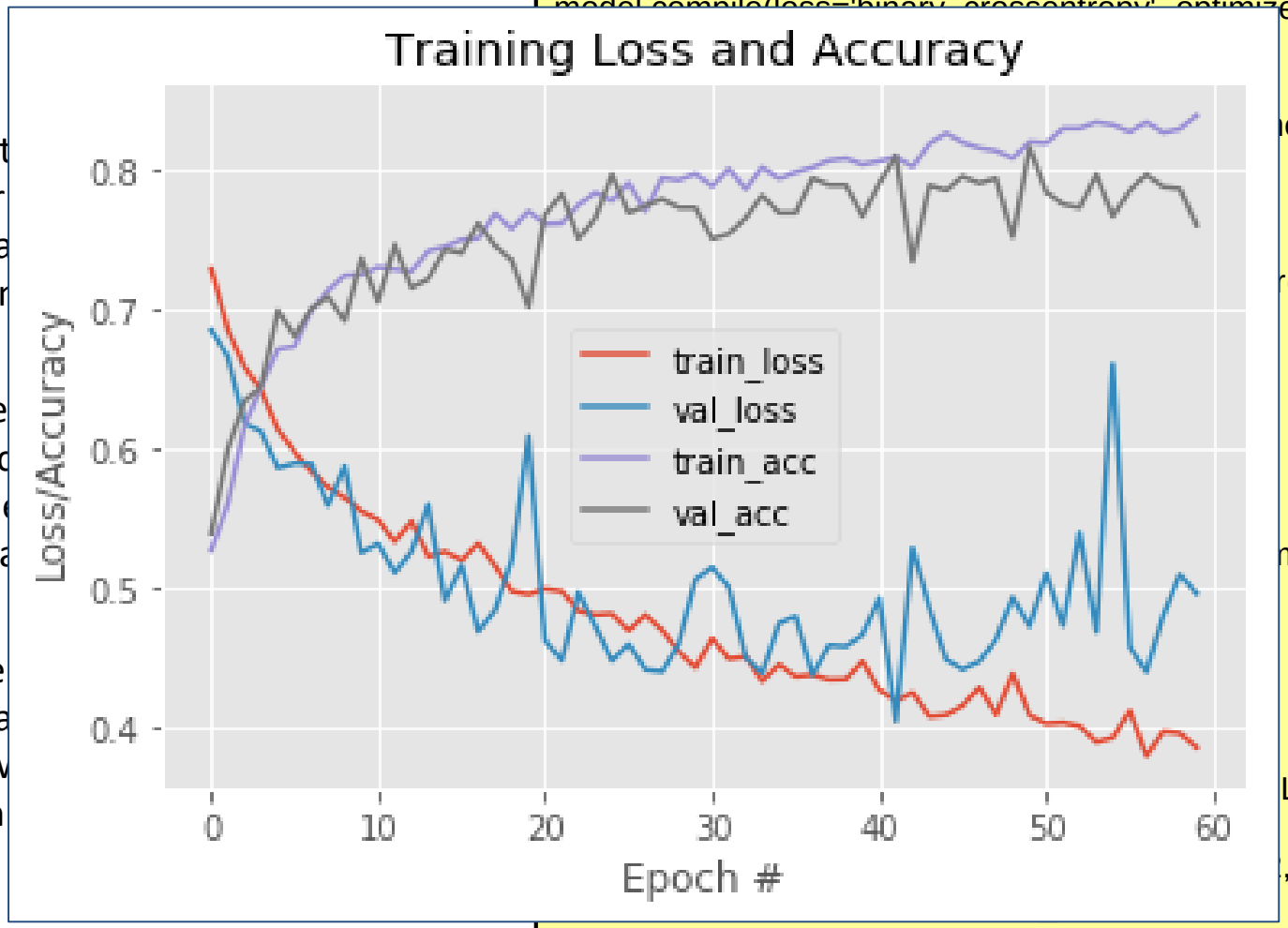
Associated Colab Notebook

```
model = convModel.build(width, height, 3)

model.compile(loss='binary_crossentropy', optimizer='rmsprop',
```

Notice in t
generator
have alrea
training ar

Notice we
time, whic
images th
and the ba

Finally we
method (a
this time v
validation



Associated [Colab Notebook]

Introduction to Transfer Learning

- It is rare to train a very deep convolutional neural network **from scratch**.

- The reason is that this is that the training process is very **computationally expensive** and time consuming. There are so many different architectures that it can be very time-consuming process.

- It is also possible that you may have a **relatively small dataset**, which could inhibit your ability to develop a convolutional neural network. Remember many of the common architectures have been build for ImageNet, which had a training set of 1.2 million images.

- What if we could use an existing **pre-trained classifier** and use it as a starting point for a new classification task? This is exactly what we do with Transfer Learning.

Types of Transfer Learning

- Deep neural networks trained on large-scale datasets such as **ImageNet** have achieved exceptional levels of performance and can be leveraged and applied to other image classification tasks.

- These networks learn a **set of rich, discriminating features to recognize 1,000 separate object classes**. It makes sense that these filters can be reused for classification tasks other than what the CNN was originally trained on.

- The two main types of Transfer Learning :
  - Networks as feature extractors
  - Fine-tuning a CNN

Networks as Feature Extractors

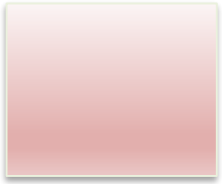
- Rather than allowing an image to forward propagate through the entire network. we can **stop the propagation at an specific layer**, and use the values outputted from the network at this stage as **feature vectors**.

- A common example would involve **removal of the final fully-connected layers**. For example, we could remove the last FC layer from a CNN trained on ImageNet (this layer outputs the 1000 class scores for different classes).

- We then treat the rest of the network **(all the convolutional layers) as a fixed feature extractor for the new dataset** and use the generated features as input into a new ML classifier.

- For example previously we looked at the VGG architecture. A common approach with feature extraction would be to remove the fully connected layer and push all images through the network and collect the resulting feature data. This then becomes your new dataset.

224*224*3



→ 2*Conv (f=64) → 224*224*64 → Pooling → 112*112*64

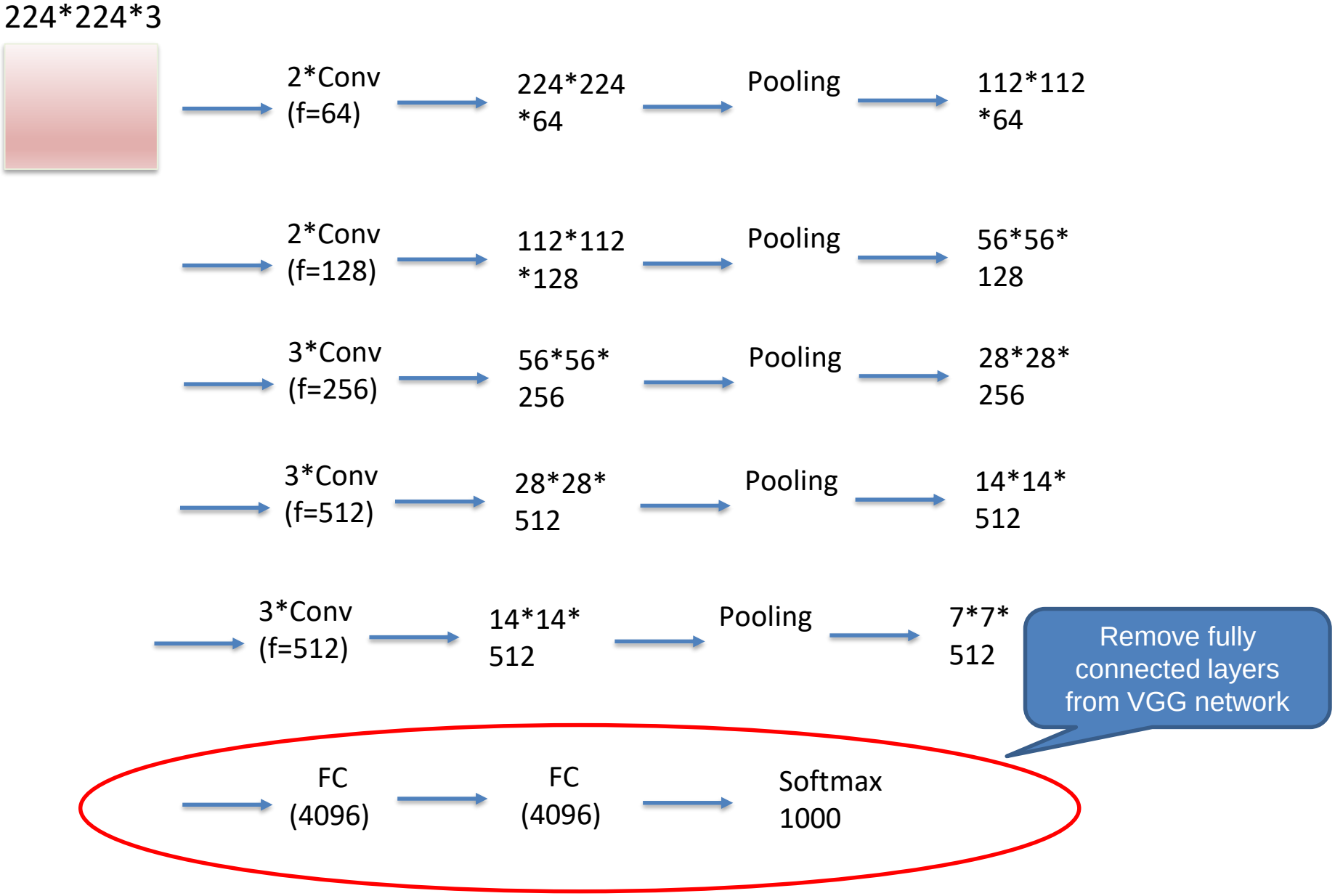
→ 2*Conv (f=128) → 112*112*128 → Pooling → 56*56*128

→ 3*Conv (f=256) → 56*56*256 → Pooling → 28*28*256

→ 3*Conv (f=512) → 28*28*512 → Pooling → 14*14*512

→ 3*Conv (f=512) → 14*14*512 → Pooling → 7*7*512

→ FC (4096) → FC (4096) → Softmax 1000


224*224*3
2*Conv (f=64)
224*224 *64
Pooling
112*112 *64
2*Conv (f=128)
112*112 *128
Pooling
56*56* 128
3*Conv (f=256)
56*56* 256
Pooling
28*28* 256
3*Conv (f=512)
28*28* 512
Pooling
14*14* 512
3*Conv (f=512)
14*14* 512
Pooling
7*7* 512
Remove fully connected layers from VGG network
FC (4096)
FC (4096)
Softmax 1000

- Notice the last layer in our network is now a max pooling layer which produces a feature map of the shape of 7*7*512. We can flatten this out this out in the normal way to create a single vector with **25,088** values **(7*7*512).**
- We can then treat this as a single feature vector for our image (think of this as a single line from your training set with 25,088 features).

Feature Extraction – Summary of Steps

- We can take the alternated pretrained VGG network (see previous slide).

- We push a new dataset of images through the network (images not from the original ImageNet competition).
- It will produced a matrix of images. Each row will have **25,088 values** (single feature vectors).

- Let's assume we have **5000 images training images.**

- Once we have the feature vector produced by the pre-trained network (see previous slide) we can **train an off-the-shelf machine learning models** such a Linear SVM, Logistic Regression classifier, or Random Forest on top of these features to obtain a classifier that recognizes new classes of images.

- We are using the CNN as an intermediary feature extractor. The downstream machine learning classifier will take care of learning the underlying patterns of the features extracted from the CNN.

Feature Extraction

- An efficient method of enabling feature selection is by **piping the entire dataset through the convolutional base and collecting the output to a NumPy array**. This array becomes our new dataset. This feature dataset could be saved to disk.

- In turn we can then use this new dataset as the training data for a new classifier. This approach is efficient because it only needs us to pass the training data through the convolutional base once.

- The VGG16 model, among others, comes pre-packaged with Keras. You can import it from the **keras.applications** module. Here's the list of some of the image-classification models (all pretrained on the ImageNet dataset) that are available as part of keras
  - Xception
  - Inception V3
  - ResNet50
  - VGG16
  - VGG19

This page shows the full range of models available:

```
import tensorflow as tf
vggModel = tf.keras.applications.VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))

print (vggModel.summary())
```

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

Notice there are no fully connected layers included in the network.

```
# resize all images to this width and height
width, height= 150, 150
trainDataDir = '/content/dogs_cats/data/train'
validationDataDir= '/content/dogs_cats/data/validation'
numTrainingSamples = 2000
numValidationSamples = 800
NUM_EPOCHS = 50
batchSize = 64

# This dictionary will contain all labels with an associated int value
labelDictionary ={}
# The procImages function will read all image data from a folder
# convert it to a Numpy array and add to a list
# It will also add the label for the image, the folder name
trainImages, trainLabels = procImages(trainDataDir, labelDictionary, width, height)
valImages, valLabels = procImages(validationDataDir, labelDictionary, width, height)

# Normalize data
trainImages = trainImages.astype("float") / 255.0
valImages = valImages.astype("float") / 255.0
# Map string label values to integer values.
trainLabels = (pd.Series(trainLabels).map(labelDictionary)).values
valLabels = (pd.Series(valLabels).map(labelDictionary)).values
```

Initial code is the same as we used previously with data augmentation example. We pre-process and load the images into NumPy arrays.

Associated Colab Notebook

In this code we push the training data through our VGG network and collect the results. The shape of the output will be **(2000, 4, 4, 512).** In the next line we reshape the image so that it becomes **(2000, 8192).** This will becomes the training dataset.

We then do the same for our validation data. The output shape is **(800, 4, 4, 512)**. We reshape it to be **(800, 8192)**

```
vggModel = tf.keras.applications.VGG16(weights='imagenet',
include_top=False, input_shape=(150, 150, 3))

print (vggModel.summary())




# based on Dogs and Cats dataset.
 featuresTrain = vggModel.predict(trainImages)
 featuresTrain = featuresTrain.reshape(featuresTrain.shape[0], -1)

 featuresVal = vggModel.predict(valImages)
 featuresVal = featuresTest.reshape(featuresVal.shape[0], -1)
```

Associated Colab Notebook

In the code that follows we use a random forest from Scikit Learn. Notice we use the data outputted from the CNN as both the training and subsequent validation data.

We treat it just like we could any training and validation dataset.

Our accuracy value now jumps up to <u>0.87</u>.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics

model = RandomForestClassifier(200)
model.fit(featuresTrain, trainLabels)

# evaluate the model
results = model.predict(featuresVal)
print (metrics.accuracy_score(results, valLabels))
```

Associated Colab Notebook

Create a New Model Using a Portion of an Original Model

- One issue you may have noticed in the previous code is that it only facilitates removal of the fully connected layer.

- You may want more specific control over the layers that you use for extraction.

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

For example we may want to cut the network at the highlighted point here (rather than just removing the fully connected layers).

Create a New Model Using a Portion of an Original Model

- One issue you may have noticed in the previous code is that it only facilitates removal of the fully connected layer.
- You may want more specific control over the layers that you use for extraction.
- In the code below we are using the Keras functional API that allows us to **specify particular entry and exit points from another network**. In this case het entry poin is the first layer in the VGG model and the output is the output of the ‘block4_conv2’ layer.

```
import tensorflow as tf

initialModel = tf.keras.applications.VGG19(weights='imagenet', include_top=False,
input_shape=(128, 128, 3))


newModel = tf.keras.Model(inputs=initialModel.input,
                          outputs=initialModel.get_layer('block4_conv2').output)
```

Layer (type)	Output Shape	Param #
input_6 (InputLayer)	[(None, 150, 150, 3)]	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808

Total params: 5,275,456