CS3237 Introduction to IoT

Lab 4 – Transfer Learning and MQTT

1. INTRODUCTION

In this lab we will be looking at three topics:

i. Transfer Learning: We take a neural network that has already been pre-trained on data that is related, but not exactly the same as our own, and retrain it for our own data.
ii. MQTT: We look at how to set up MQTT connections.
iii. Integration between MQTT and Keras: We look at how to create a classification server using MQTT and the classifier you build in i).

2. SUBMISSION

There are a total of 7 questions for you to answer in Sections 4 and 5 of this lab. Please answer these questions in the enclosed answer book, and upload to your respective submission folders in Submissions\Lab 4 Submissions. This lab is worth 20 marks.

You are to work in TEAMS OF 2. You only need to upload ONE copy of the answerbook, named AxxxxxxY.docx, where AxxxxxxY is the student ID of the first person listed in the answerbook.

For Group B1 (Wednesday 2 pm to 4 pm) please submit by 11.59 pm on Friday, 2 October 2020. For Group B2 (Friday 12 pm to 2 pm) please submit by 11.59 pm on Monday 5 October 2020.

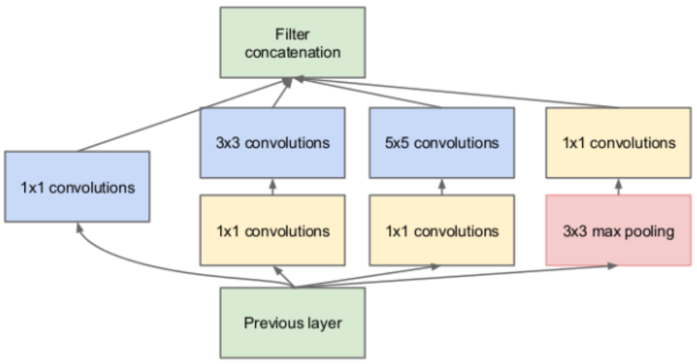
This is a very long lab, so please start early so that you can complete it on time. Every part of this lab is important in helping you with your project (and of course to answer the questions in the last two parts), so please do not skip anything.

3. TRANSFER LEARNING

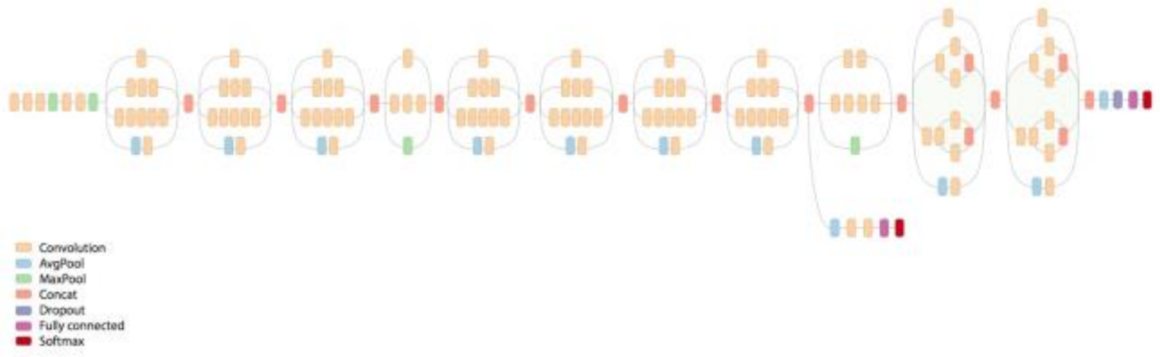
In our previous lab when we created dense and CNN networks for learning the MNIST and CIFAR-10 datasets, Keras (and TensorFlow) programs can take very, very long to run. If you were to train a CNN on a few million images on a GPU, it can take up to 2 to 3 weeks to complete training.

For this exercise we will use an extremely deep CNN called InceptionV3. InceptionV3 has hundreds of layers, and tens of millions of parameters to train. Here we will use a version of InceptionV3 that has been pre-trained on over 3 million images in over 4,000 categories. If we wanted to train our InceptionV3 model from scratch it would take several weeks even on a machine with several good GPUs.

InceptionV3 consists of “modules” that try out various filter and pooling configurations, and chooses the best ones.



Many of these modules are chained together to produce very deep neural networks:



The greatest beauty about InceptionV3 is that someone has already taken the time to train the hundreds of layers and tens of millions of parameters. All we have to do is to adapt the top layers. This is called "Transfer Learning".

Create a file called "inception.py" and follow the instructions below:

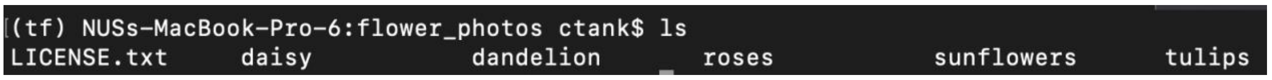
a. <u>Getting Training Images</u>

You need to obtain the training images for this exercise. To do so, at the LINUX or MacOS prompt type:

```
curl http://download.tensorflow.org/example_images/flower_photos.tgz | tar -xz
```

This will create a new directory called "flower_photos" with thousands of pictures of flowers sorted into 5 directories. Downloading this dataset may take some time as it is very large.

Change to the flower_photos directory and you will see that it has five directories:



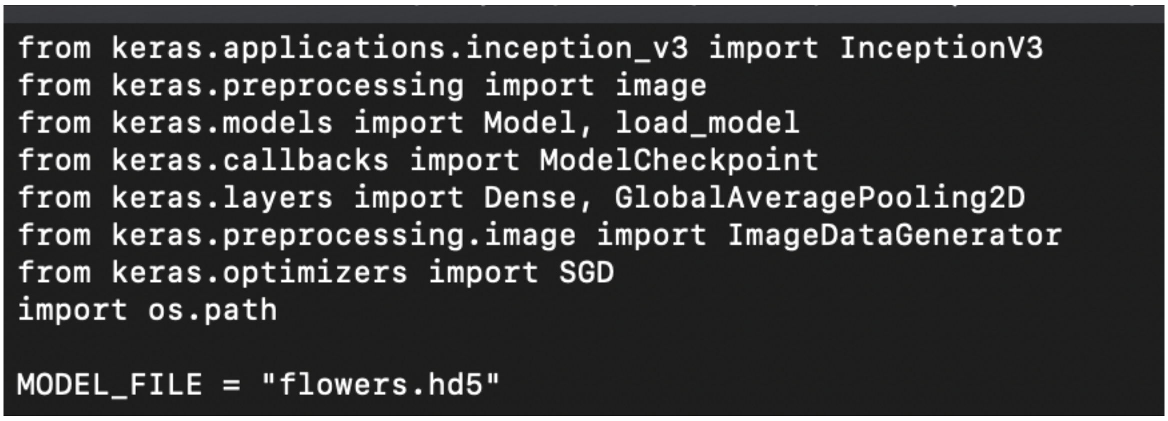
```
(tf) NUSs-MacBook-Pro-6:flower_photos ctank$ ls
LICENSE.txt      daisy           dandelion       roses           sunflowers      tulips
```

If you look in each directory you will see hundreds of images of flowers of each category. The Keras Data Generator, which we will see shortly, requires training data to be sorted like this into directories reflecting the various categories.

b. <u>Imports</u>

We need to import the InceptionV3 model from keras.applications. We also need to import in Pooling, Dense and Dropout layers that we will need to adapt the InceptionV3 model.

Key in the following into your inception.py file:

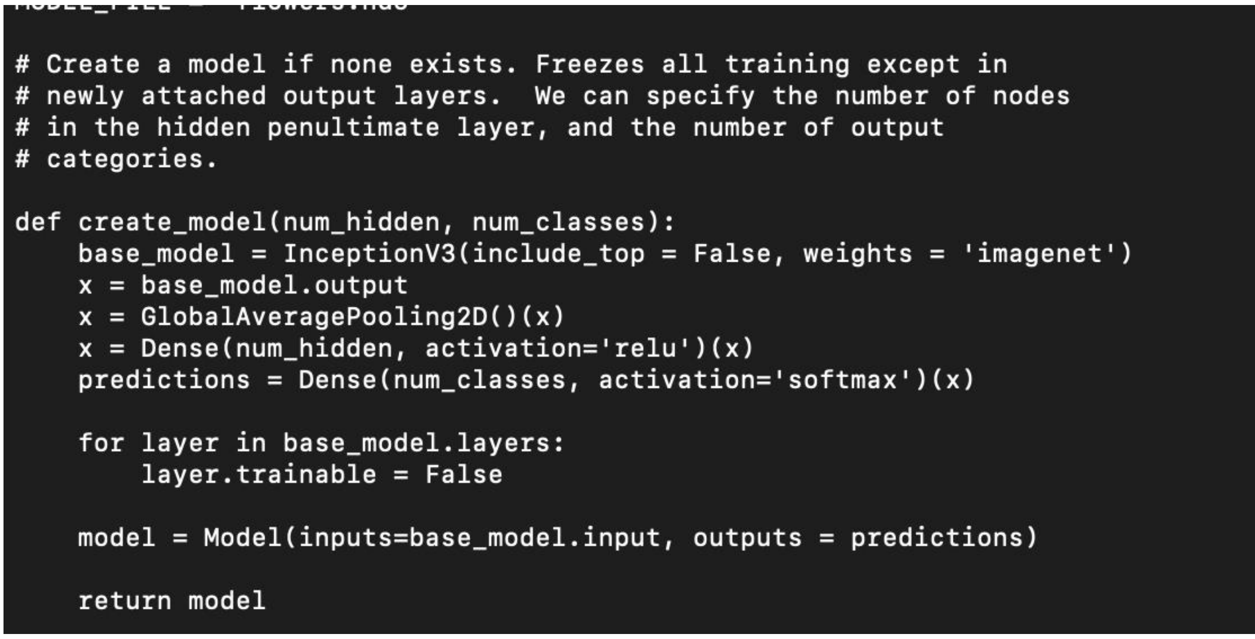


```
from keras.applications.inception_v3 import InceptionV3
from keras.preprocessing import image
from keras.models import Model, load_model
from keras.callbacks import ModelCheckpoint
from keras.layers import Dense, GlobalAveragePooling2D
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import SGD
import os.path

MODEL_FILE = "flowers.hd5"
```

c. <u>Defining your Model</u>

In this part we will do something special; we will ask TensorFlow (through Keras) to download the InceptionV3 model that has been trained on ImageNet. We will then add in new layers.

KEY IN THE FOLLOWING INTO inception.py

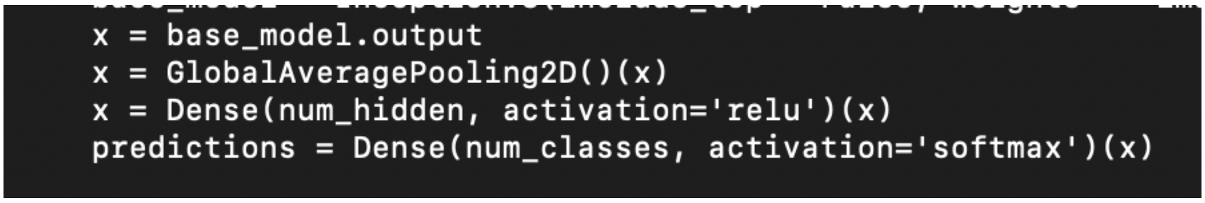


```
# Create a model if none exists. Freezes all training except in
# newly attached output layers.  We can specify the number of nodes
# in the hidden penultimate layer, and the number of output
# categories.

def create_model(num_hidden, num_classes):
    base_model = InceptionV3(include_top = False, weights = 'imagenet')
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(num_hidden, activation='relu')(x)
    predictions = Dense(num_classes, activation='softmax')(x)

    for layer in base_model.layers:
        layer.trainable = False

    model = Model(inputs=base_model.input, outputs = predictions)

    return model
```

<u>Explanation of what you keyed in</u>

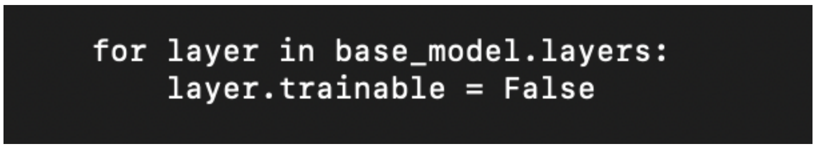
In this code the basic InceptionV3 model is put into "base_model". In the InceptionV3 function call we specify "include_top" to be False, which allows us to specify our own Dense layer. This is important because we will retrain the Dense layer to recognize new objects. We also specify "weights='imagenet'" because we want our InceptionV3 model to be pre-trained on the millions of ImageNet images.

Let's see how this code works. The following few lines stack new layers onto the InceptionV3 model: the first is a pooling layer that takes an average of the entire layer below (to reduce dimensionality). This is followed by a dense layer, using ReLu activation, followed by an output layer "predictions" using softmax.



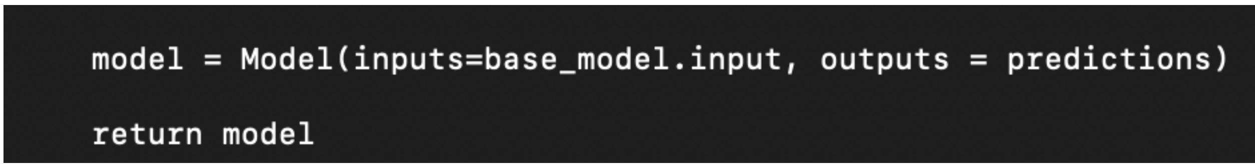
```
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(num_hidden, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)
```

We only want to train the top layers, so we freeze all the layers in the base model:



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

Then we create a new model based on what we have loaded and modified and return that:

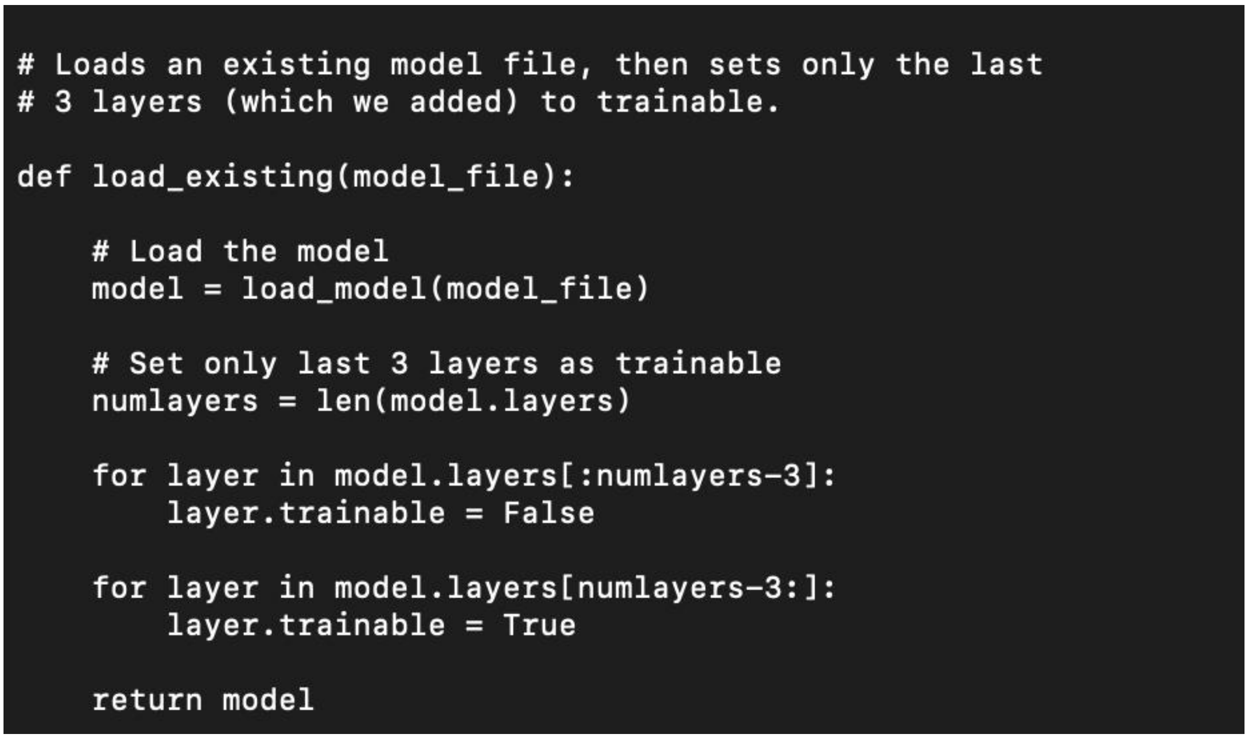


```
model = Model(inputs=base_model.input, outputs = predictions)

return model
```

d. <u>Loading an Existing Model</u>

We will create a new function called load_existing that will load any existing model that we may have previously saved. This uses the standard load_model function that we've used in our MNSTI CNN model, but adds code that freezes the layers in the original model.

KEY IN THE FOLLOWING INTO inception.py



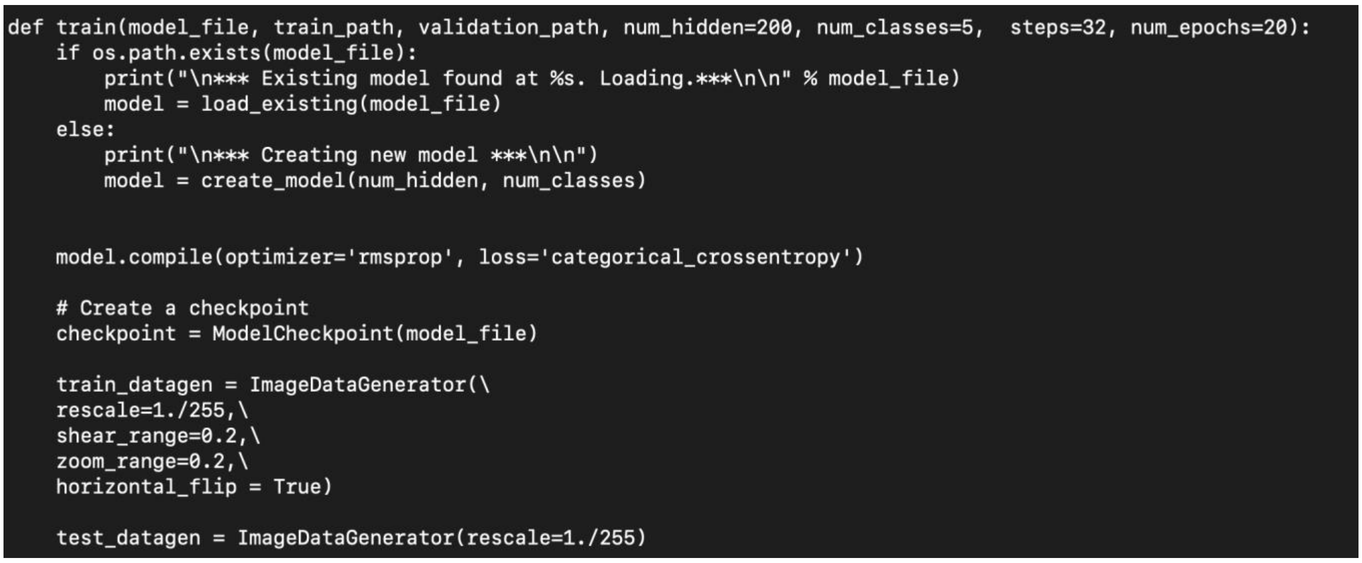
```
# Loads an existing model file, then sets only the last
# 3 layers (which we added) to trainable.

def load_existing(model_file):

    # Load the model
    model = load_model(model_file)

    # Set only last 3 layers as trainable
    numlayers = len(model.layers)

    for layer in model.layers[:numlayers-3]:
        layer.trainable = False

    for layer in model.layers[numlayers-3:]:
        layer.trainable = True

    return model
```

Recall that we added 3 layers of our own (pooling, dense and output), so we set layers up to numlayers-3 to be non-trainable, and set layers from numlayers-3 onwards to be trainable.

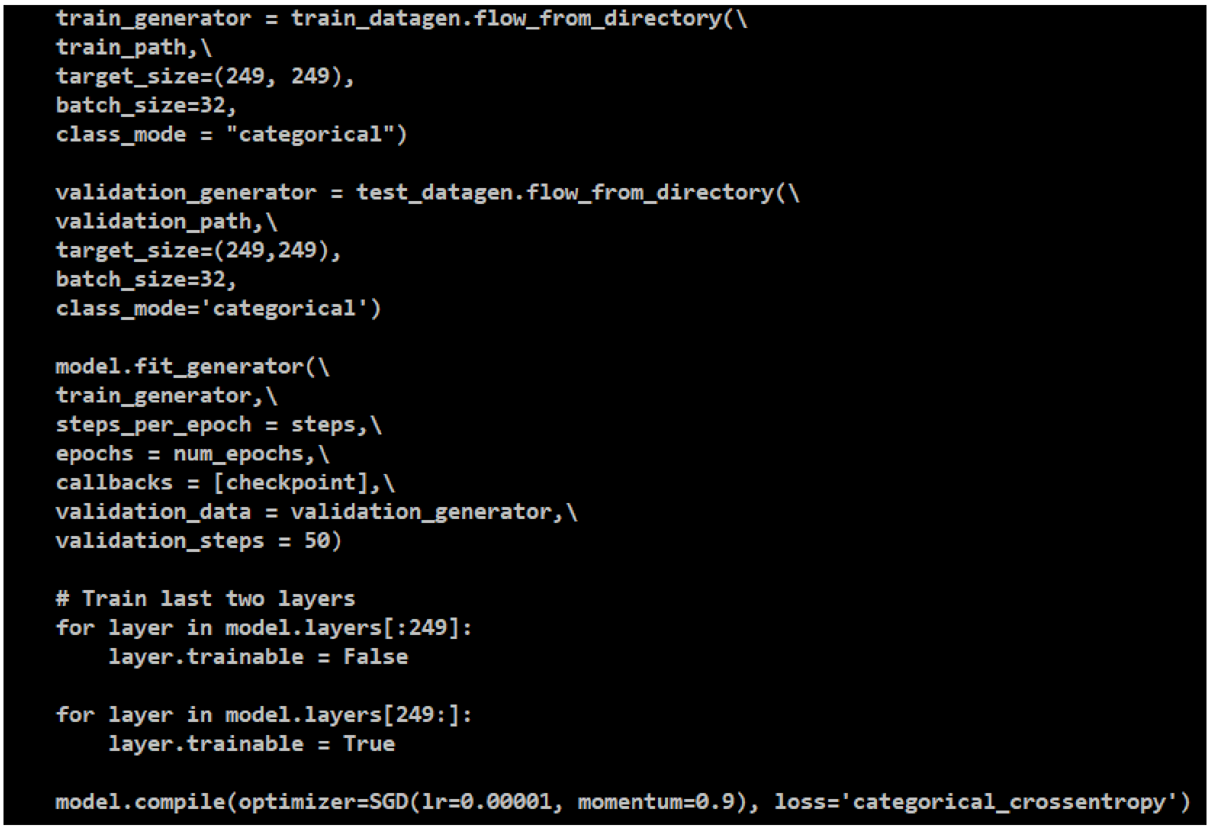
e. <u>Training the Model</u>

This part gets complicated. We first show you the entire train function then explain each part.

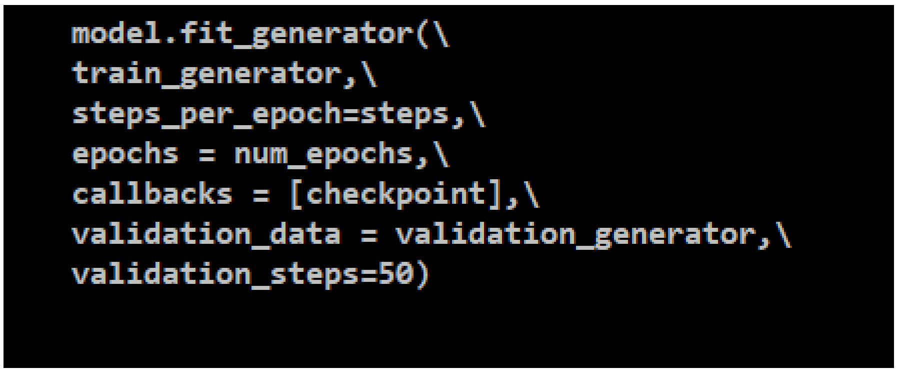
KEY IN THE FOLLOWING INTO inception.py



```
def train(model_file, train_path, validation_path, num_hidden=200, num_classes=5,  steps=32, num_epochs=20):
    if os.path.exists(model_file):
        print("\n*** Existing model found at %s. Loading.***\n\n" % model_file)
        model = load_existing(model_file)
    else:
        print("\n*** Creating new model ***\n\n")
        model = create_model(num_hidden, num_classes)


    model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

    # Create a checkpoint
    checkpoint = ModelCheckpoint(model_file)

    train_datagen = ImageDataGenerator(\
    rescale=1./255,\
    shear_range=0.2,\
    zoom_range=0.2,\
    horizontal_flip = True)

    test_datagen = ImageDataGenerator(rescale=1./255)
```



```
train_generator = train_datagen.flow_from_directory(\
train_path,\
target_size=(249, 249),
batch_size=32,
class_mode = "categorical")

validation_generator = test_datagen.flow_from_directory(\
validation_path,\
target_size=(249,249),
batch_size=32,
class_mode='categorical')

model.fit_generator(\
train_generator,\
steps_per_epoch = steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps = 50)

# Train last two layers
for layer in model.layers[:249]:
    layer.trainable = False

for layer in model.layers[249:]:
    layer.trainable = True

model.compile(optimizer=SGD(lr=0.00001, momentum=0.9), loss='categorical_crossentropy')
```

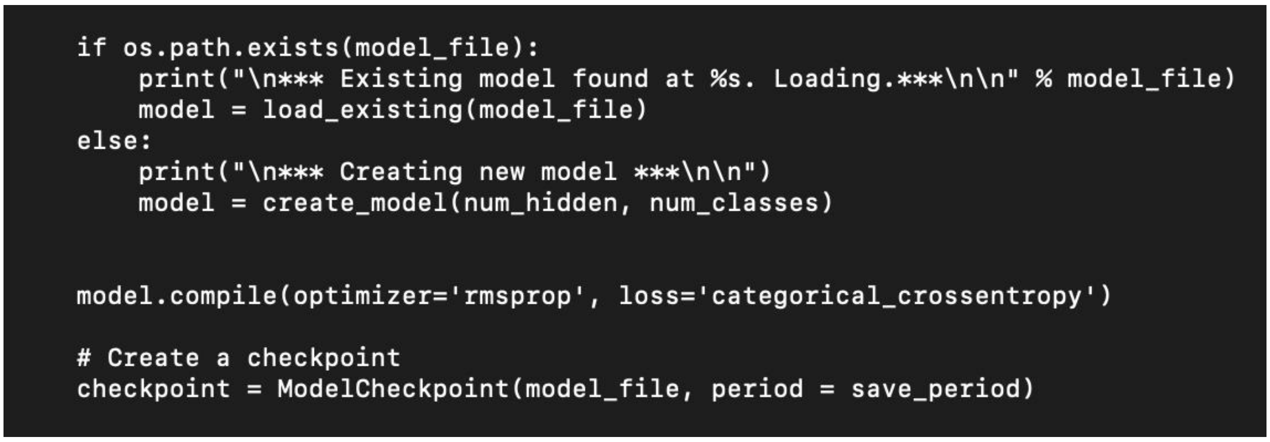


```
model.fit_generator(\
train_generator,\
steps_per_epoch=steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps=50)
```

<u>Explanation of what you've just keyed in</u>

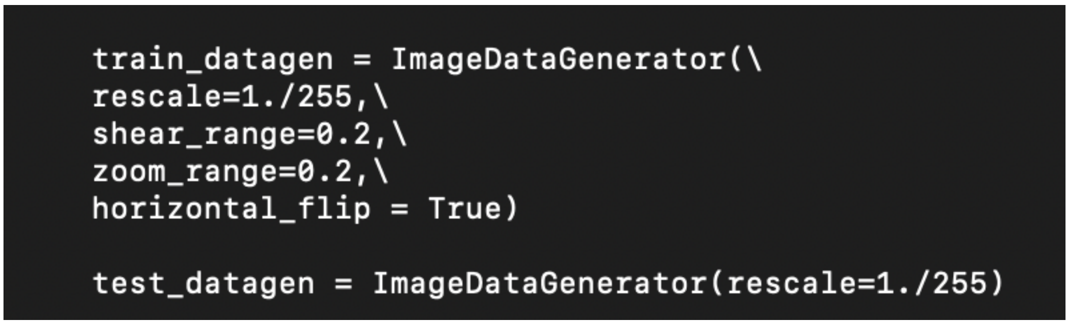
Let's now look at each part:

We begin first by checking if the model exists, and load it using our load_existing function if it does. Otherwise we create a new model calling create_model, and compile it with a categorical cross entropy loss, and using the RMSProp optimizer. We also create a checkpoint to save the weights after each epoch.



```
if os.path.exists(model_file):
    print("\n*** Existing model found at %s. Loading.***\n\n" % model_file)
    model = load_existing(model_file)
else:
    print("\n*** Creating new model ***\n\n")
    model = create_model(num_hidden, num_classes)


model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# Create a checkpoint
checkpoint = ModelCheckpoint(model_file, period = save_period)
```

We then create an ImageDataGenerator for training data and one more for testing data:



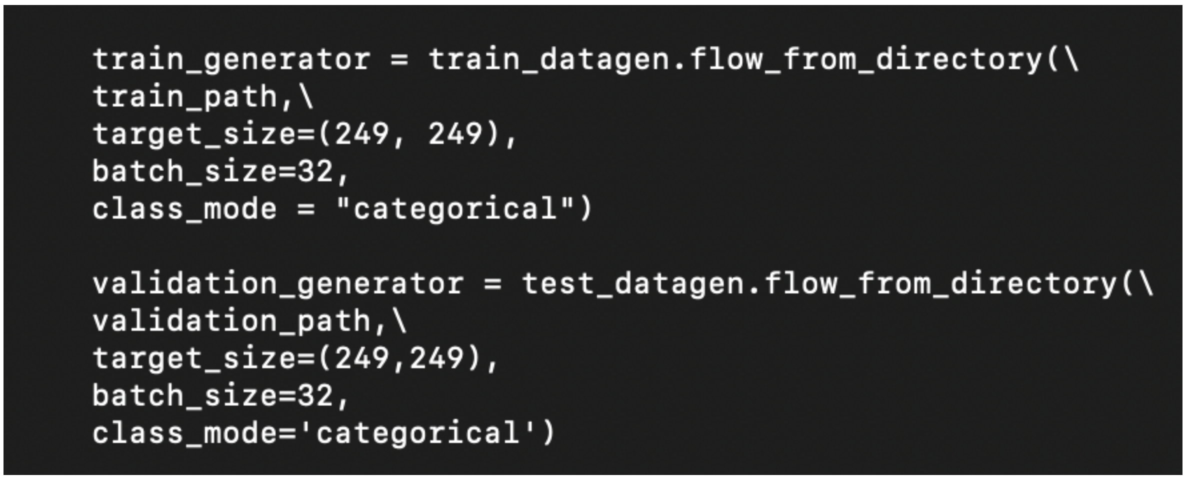
```
train_datagen = ImageDataGenerator(\
rescale=1./255,\
shear_range=0.2,\
zoom_range=0.2,\
horizontal_flip = True)

test_datagen = ImageDataGenerator(rescale=1./255)
```

Our “train_datagen” object will produce additional images from the images we provide it by shearing the image, zooming in or out, or by flipping the image horizontally. This is to artificially create new data to reduce overfitting. It also divides all values by 255 to scale them to [0,1].

Our “test_datagen” object is simpler; it just simply scales the data to [0,1].

We now use the “flow_from_directory” method in ImageDataGenerator to tell Keras to take all the images from the specified directory. IMPORTANT: the images must be sorted into directories corresponding to each category. The directories should also be named after each category.

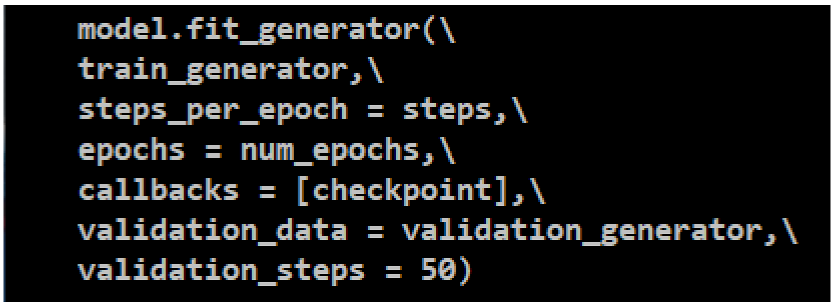
Now we actually create the generators:



```
train_generator = train_datagen.flow_from_directory(\
train_path,\
target_size=(249, 249),
batch_size=32,
class_mode = "categorical")

validation_generator = test_datagen.flow_from_directory(\
validation_path,\
target_size=(249,249),
batch_size=32,
class_mode='categorical')
```

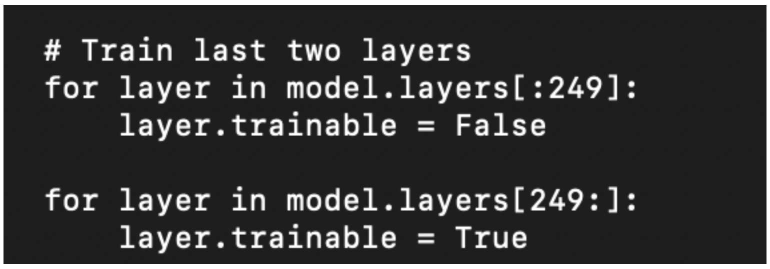
Here we specify the path in “train_path”, and we tell ImageDataGenerator to scale every image to 249 x 249 pixels (the size used to train InceptionV3). We tell it to present images in batches of 32, and to use “categorical” class mode. I.e. training labels will be presented as one-hot vectors instead of integers.

We do the same for “validation_generator”. We then call model.fit_generator **(on newer Keras versions, use model.fit intsead)** to train the network:



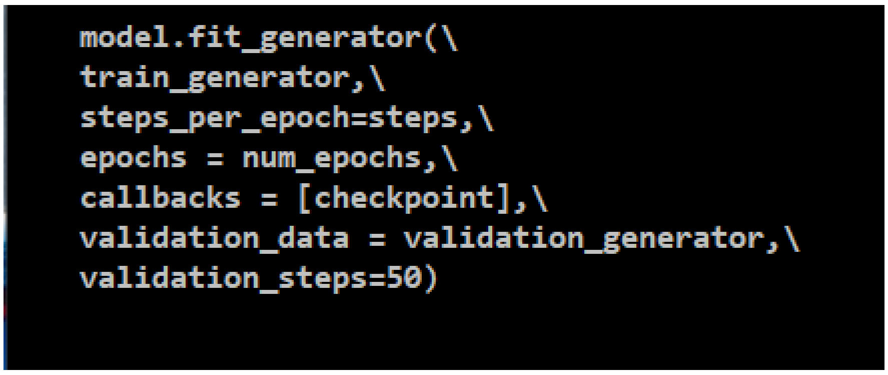
```
model.fit_generator(\
train_generator,\
steps_per_epoch = steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps = 50)
```

After training of the top layers is complete, we train some of the layers in the original InceptionV3 CNN to fine tune them. So we set all layers up to layer 249 to be non-trainable, and all layers from 249 onwards to be trainable:



```
# Train last two layers
for layer in model.layers[:249]:
    layer.trainable = False

for layer in model.layers[249:]:
    layer.trainable = True
```

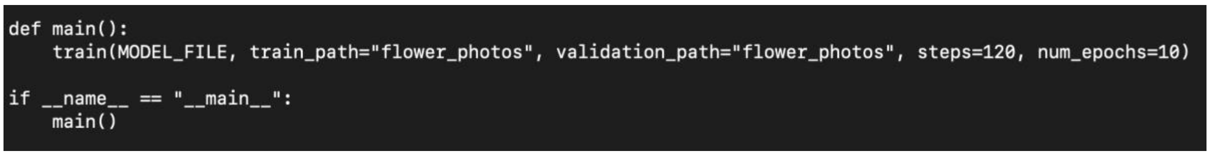
We then compile with an SGD optimizer instead of rmprop that we used earlier, and we train again with model.fit_generator **(again newer versions of Keras may ask you to use model.fit instead)**.



```
model.fit_generator(\
train_generator,\
steps_per_epoch=steps,\
epochs = num_epochs,\
callbacks = [checkpoint],\
validation_data = validation_generator,\
validation_steps=50)
```

f. Putting It Together

KEY IN THE FOLLOWING INTO inception.py



```
def main():
    train(MODEL_FILE, train_path="flower_photos", validation_path="flower_photos", steps=120, num_epochs=10)

if __name__ == "__main__":
    main()
```

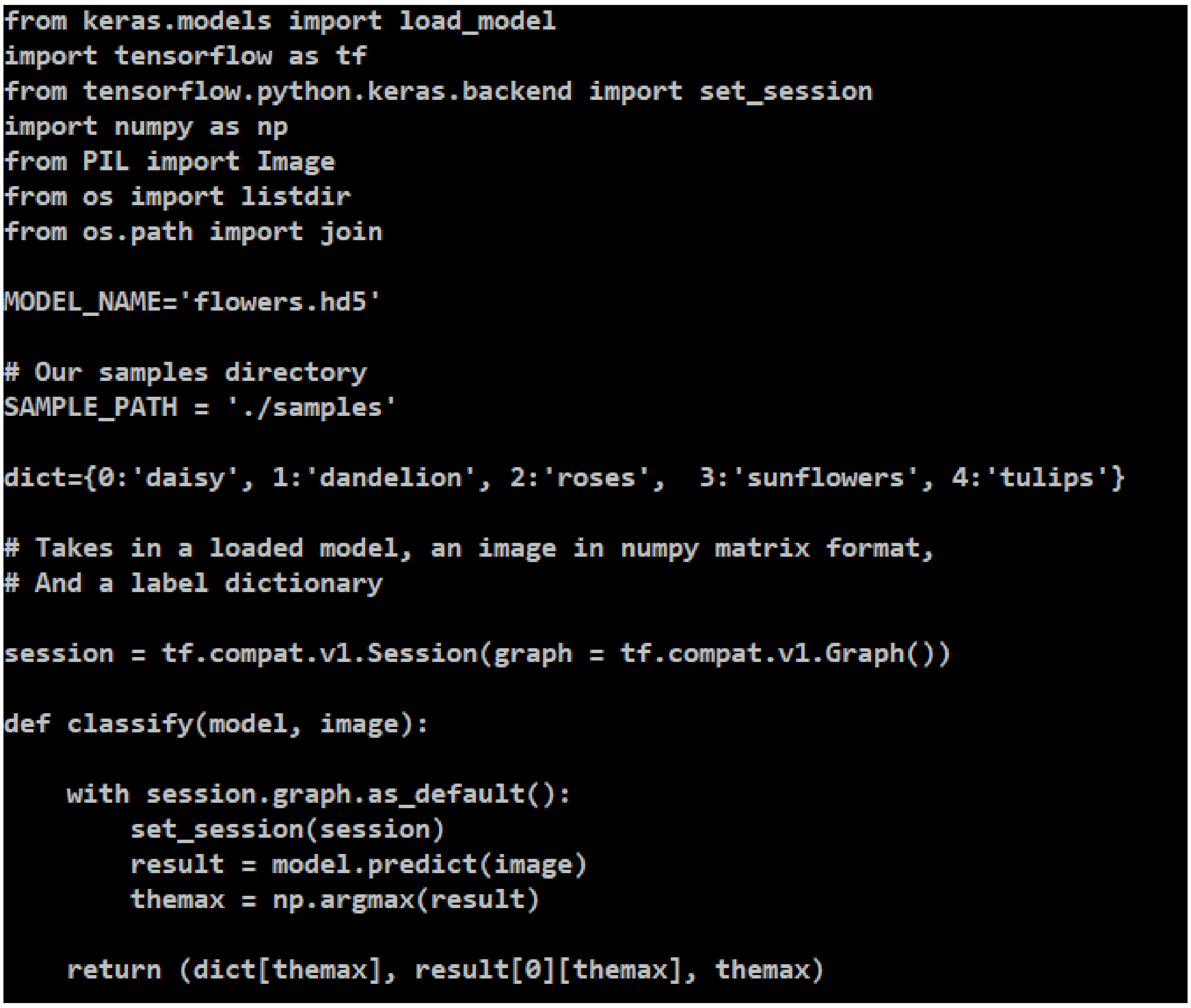
g. Starting the Training

Save your file, and type “python inception.py” to start the training.   If this is the first time you are running this program, it will download the ImageNet weights file for the InceptionV3 deep learning network. This can take a while as the network is very large.

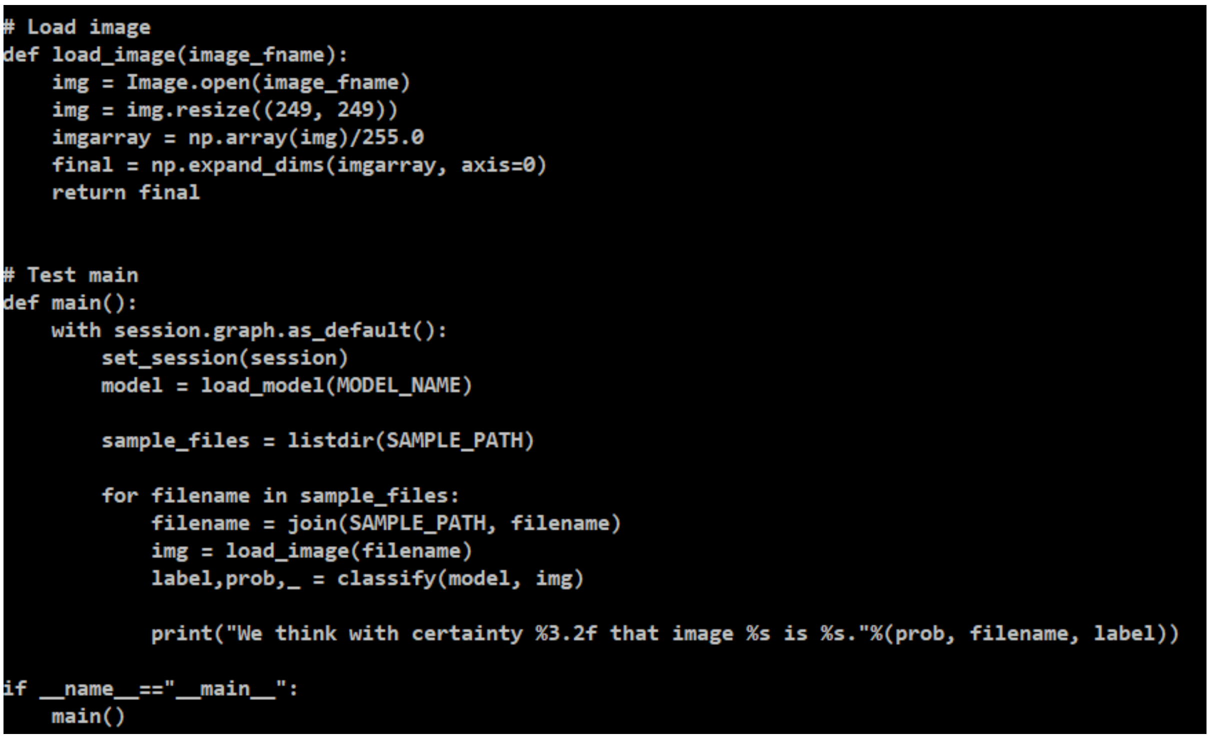
h. Predicting

We will now see how to use the model we have trained to predict based on images we give it. (Note: Pillow must be installed. If your system complains that the PIL module does not exist, install Pillow with “pip3 install pillow” without the quotes).

Create a file called “predict.py” and key in the following:



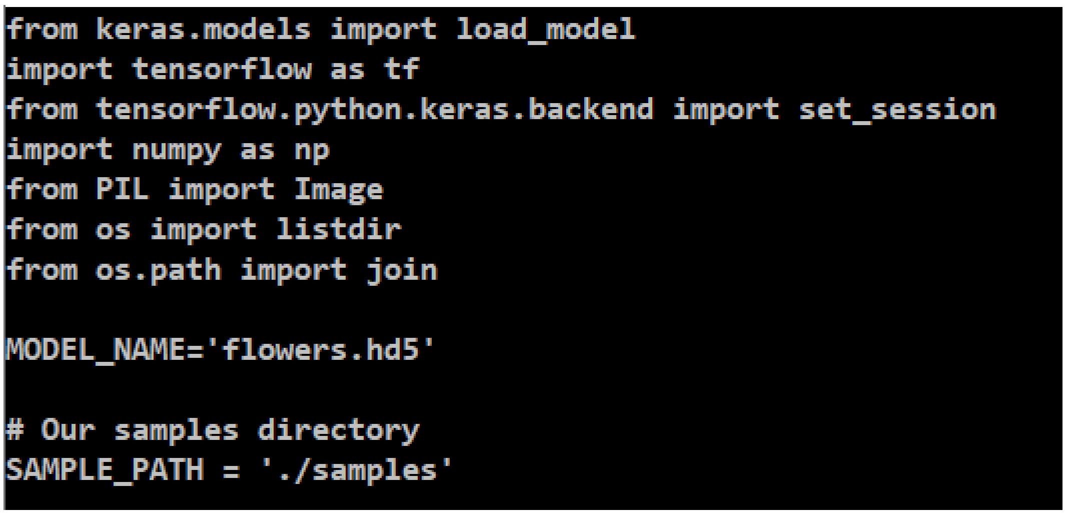
```
from keras.models import load_model
import tensorflow as tf
from tensorflow.python.keras.backend import set_session
import numpy as np
from PIL import Image
from os import listdir
from os.path import join

MODEL_NAME='flowers.hd5'

# Our samples directory
SAMPLE_PATH = './samples'

dict={0:'daisy', 1:'dandelion', 2:'roses',  3:'sunflowers', 4:'tulips'}

# Takes in a loaded model, an image in numpy matrix format,
# And a label dictionary

session = tf.compat.v1.Session(graph = tf.compat.v1.Graph())

def classify(model, image):

    with session.graph.as_default():
        set_session(session)
        result = model.predict(image)
        themax = np.argmax(result)

    return (dict[themax], result[0][themax], themax)
```



```
# Load image
def load_image(image_fname):
    img = Image.open(image_fname)
    img = img.resize((249, 249))
    imgarray = np.array(img)/255.0
    final = np.expand_dims(imgarray, axis=0)
    return final


# Test main
def main():
    with session.graph.as_default():
        set_session(session)
        model = load_model(MODEL_NAME)

        sample_files = listdir(SAMPLE_PATH)

        for filename in sample_files:
            filename = join(SAMPLE_PATH, filename)
            img = load_image(filename)
            label,prob,_ = classify(model, img)

            print("We think with certainty %3.2f that image %s is %s."%(prob, filename, label))

if __name__=="__main__":
    main()
```

<u>**Explanation of What You Keyed In**</u>

We go through each part to see what it is doing. We start first by importing what we need.



```
from keras.models import load_model
import tensorflow as tf
from tensorflow.python.keras.backend import set_session
import numpy as np
from PIL import Image
from os import listdir
from os.path import join

MODEL_NAME='flowers.hd5'

# Our samples directory
SAMPLE_PATH = './samples'
```

Most of this is very familiar to you except maybe PIL. PIL is Pillow, the Python Image Library and it lets us load and manipulate images. In addition to solve some threading problems in the Keras library, we also import "set_session", which allows Tensorflow to preserve its states correctly in a multithreaded environment (see later).

Following this we create a dictionary that lets us map predicted indexes back to class names. Sadly Keras has no way of doing this neatly so we use our own dictionary.

```
dict={0:'daisy', 1:'dandelion', 2:'roses',  3:'sunflowers', 4:'tulips'}
```

Next we create a new TensorFlow session, to preserve its states correctly in a multithreaded environment, which we will encounter when we integrate with MQTT:

```
session = tf.compat.v1.Session(graph = tf.compat.v1.Graph())
```

Next we write the classification function. It is relatively straightforward and just uses the model.predict method. The only thing special is the use of **session.graph.as_default**. This is used here to fix a multithreading bug in Keras that causes it to lose graphs:

```
def classify(model, image):

    with session.graph.as_default():
        set_session(session)
        result = model.predict(image)
        themax = np.argmax(result)

    return (dict[themax], result[0][themax], themax)
```

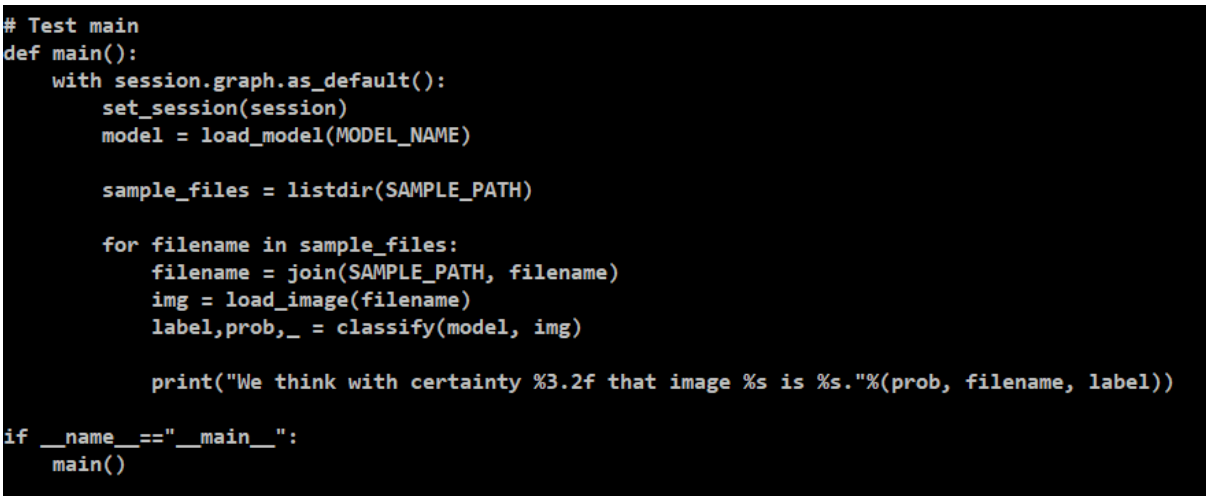
Here model.predict returns a vector of probabilities in “result”. We then use np.argmax to find the index with the highest probability. Finally we use this to index dict to get the label. We also return the probability itself and the inde

We now define a function called load_image:

```
# Load image
def load_image(image_fname):
    img = Image.open(image_fname)
    img = img.resize((249, 249))
    imgarray = np.array(img)/255.0
    final = np.expand_dims(imgarray, axis=0)
    return final
```

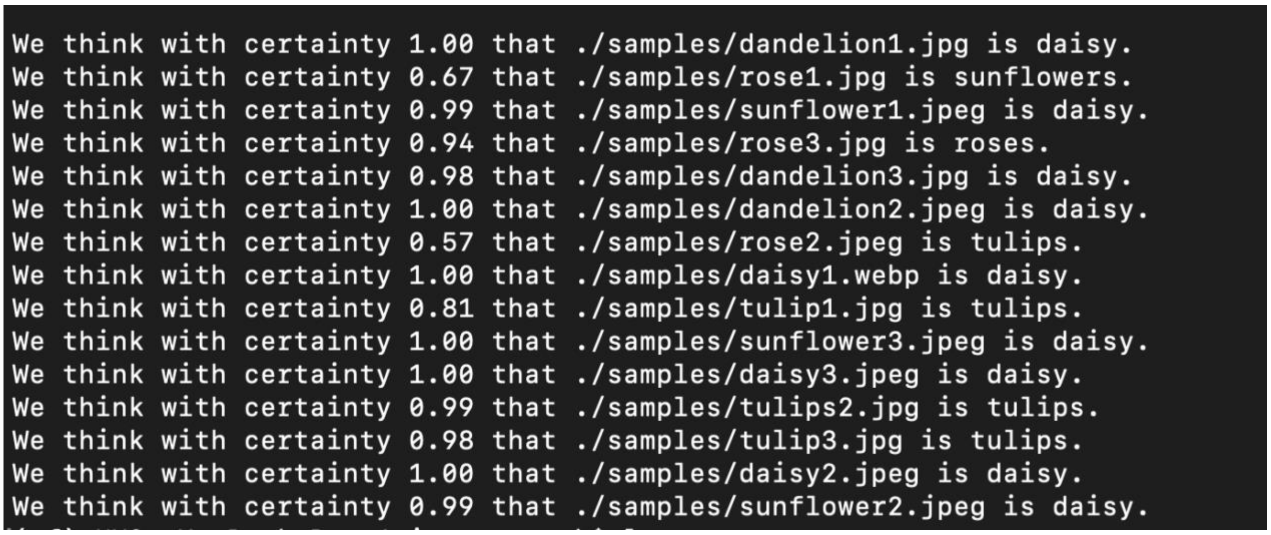
This uses the Open method from Image in PIL to load the JPG image, then calls the resize method to turn the image into a 249x249 image. We then turn the picture into a Numpy array, and add in one extra dimension to the image for the batch size (which model.classify needs).

Finally we have main() which loads the files in the “samples” directory and classifies the files it finds in there. We again use **session.graph.as_default** to preserve graphs:



```
# Test main
def main():
    with session.graph.as_default():
        set_session(session)
        model = load_model(MODEL_NAME)

        sample_files = listdir(SAMPLE_PATH)

        for filename in sample_files:
            filename = join(SAMPLE_PATH, filename)
            img = load_image(filename)
            label,prob,_ = classify(model, img)

            print("We think with certainty %3.2f that image %s is %s."%(prob, filename, label))

if __name__=="__main__":
    main()
```

Assuming that you have tulip2.jpg (or download any picture of a tulip and name it "tulip2.jpg") in your directory, when you run this program it will output:



```
We think with certainty 1.00 that ./samples/dandelion1.jpg is daisy.
We think with certainty 0.67 that ./samples/rose1.jpg is sunflowers.
We think with certainty 0.99 that ./samples/sunflower1.jpeg is daisy.
We think with certainty 0.94 that ./samples/rose3.jpg is roses.
We think with certainty 0.98 that ./samples/dandelion3.jpg is daisy.
We think with certainty 1.00 that ./samples/dandelion2.jpeg is daisy.
We think with certainty 0.57 that ./samples/rose2.jpeg is tulips.
We think with certainty 1.00 that ./samples/daisy1.webp is daisy.
We think with certainty 0.81 that ./samples/tulip1.jpg is tulips.
We think with certainty 1.00 that ./samples/sunflower3.jpeg is daisy.
We think with certainty 1.00 that ./samples/daisy3.jpeg is daisy.
We think with certainty 0.99 that ./samples/tulips2.jpg is tulips.
We think with certainty 0.98 that ./samples/tulip3.jpg is tulips.
We think with certainty 1.00 that ./samples/daisy2.jpeg is daisy.
We think with certainty 0.99 that ./samples/sunflower2.jpeg is daisy.
```

(Note: The sample output was taken after just ONE epoch of training. It isn't perfect but already quite good!)

TIP: As you can observe from running this program, it takes a long time to load the models. To speed up execution load up the model ONCE before performing the classification over many images.

4. MESSAGE QUEUING TELEMETRY TRANSPORT

Now that we have built a classifier for the flowers dataset, let's build a more complete IoT system that includes message passing over MQTT. For the first part we will just build a simple MQTT publisher and subscriber, and then we will see how to integrate MQTT and Keras.

1. Setting Up the MQTT Broker and Client

We will use Mosquitto for our MQTT Broker running on your laptop. Open a terminal, and Enter the following commands:

Install Mosquitto and its clients.

(Ubuntu, including Windows 10 WSL with Ubuntu)

```
sudo apt-get install mosquitto mosquitto-clients
pip install paho-mqtt
```

(MacOS)

```
brew install mosquitto
pip install paho-mqtt
```

MacOS automatically installs mosquitto-clients so there's no need to do so manually Paho MQTT is the Python library for communicating with the MQTT Broker.

We also need to install the Paho MQTT client for Python:

```
pip install paho-mqtt
```

2. <u>MQTT Topics and Messages</u>

MQTT messages are organized into topics. A topic can be of the form "X/Y", where "X" is some category, and "Y" is a particular topic in the category.

Mosquitto comes with two very useful programs:

mosquitto_sub: Subscribes and listens for messages for a particular topic.

mosquitto_pub: Publishes messages to a particular topic.

To see how these work:

(a) (Ubuntu) In the first terminal, type the following to start Mosquitto:

```
mosquitto
```

(MacOS) in the first terminal, type the following to start Mosquitto:

```
brew services start mosquitto
```

Note: If Mosquitto has already been started you will get an error message.

(b) Open up two more Ubuntu or MacOS terminals.
(c) In one terminal, start up a subscriber by typing:

```
mosquitto_sub –h localhost –t test/abc
```

This subscriber is only interested in the topic test/abc. We will call this the "subscriber terminal".

(d) In the second terminal publish a message to topic test/abc. We will call this terminal the “publisher terminal”.

```
mosquitto_pub -h localhost -t test/abc -m “Hello world”
```

You will see “Hello world” appear in the subscriber terminal.

(e) Again in the publisher terminal we publish a new message to topic test/def:

```
 mosquitto_pub -h localhost -t test/def -m “Hello!”
```

This time you will notice that the message DOES NOT appear in the subscriber terminal.

(f) Go back to the subscriber window in part (b) above. Press CTRL-C to exit the mosquitto_sub program. Restart it with:

```
mosquitto_sub -t test/# -h localhost
```

(g) Now in the publisher terminal in part (e), type:

```
mosquitto_pub -t test/abc -h localhost -m “TEST!”
```

You will see that it appears in the subscriber window. Likewise:

```
mosquitto_pub -t test/xyz -h localhost -m “TEST TEST!”
```

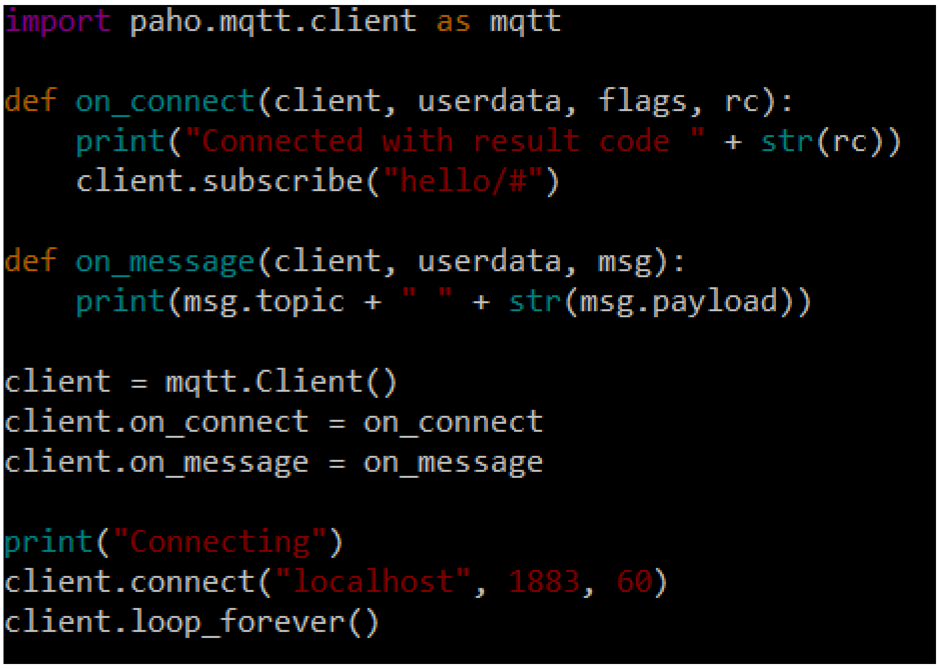
You will see that this again appears in the subscriber terminal even though test/xyz is a new topic.

This is because the subscriber has subscribed to test/#. The “#” means that it is interested in every topic under test/, which is why it receives all the messages.

3. <u>Writing MQTT Clients</u>

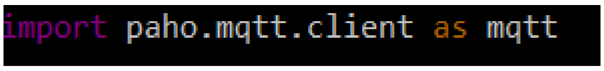
We will now start writing our first MQTT client. Using some sort of editor (vim is best), create a file called “mqtt.py” and key in the following code:

a. <u>Entering our Paho MQTT Client Code</u>



```
import paho.mqtt.client as mqtt

def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))
    client.subscribe("hello/#")

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

print("Connecting")
client.connect("localhost", 1883, 60)
client.loop_forever()
```

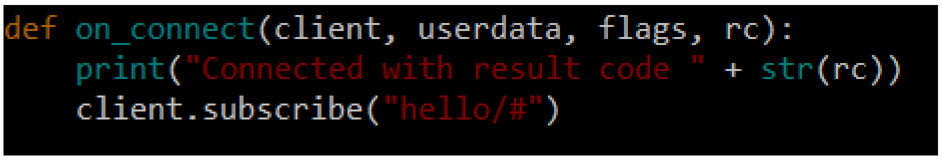
This is an explanation for the code:



```
import paho.mqtt.client as mqtt
```

This line brings in the Paho MQTT client library. Despite the name of the library (paho.mqtt.client), there is actually no Paho MQTT broker library.

We now create two callbacks. The first callback is called when Paho MQTT connects to the MQTT broker:



```
def on_connect(client, userdata, flags, rc):
    print("Connected with result code " + str(rc))
    client.subscribe("hello/#")
```

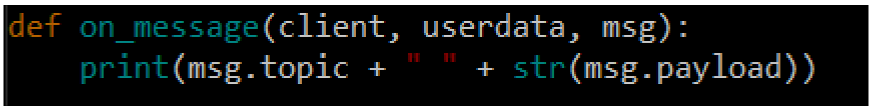
The parameters for on_connect are as follows:

Parameter	Description
client	Instance of the Paho MQTT client library.
userdata	Private user data that can be set in the MQTT Client() constructor. Not used here.
flags	A dictionary containing flags returned by the broker. Not used here.
rc	Result code: 0 – OK 1 – Connection refused, incorrect protocol version. 2 – Connection refused, invalid client identifier. 3 – Connection refused, service not available.

	4 – Connection refused, bad user name and password. 5 – Connection refused, not authorized.

In our on_connect code we simply subscribe to the “hello/#” topic, by calling client.subscribe.

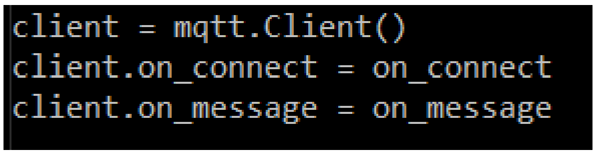
Our second callback is shown below. It is called whenever a new message comes in.



```python
def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))
```

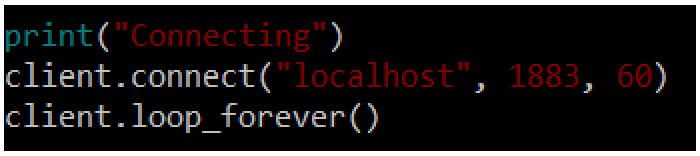
The most important parameter is msg, which contains the message topic and contents (payload).

Now we are ready to set up the Paho MQTT library. We call the Client() constructor, then point it to the on_connect and on_message callbacks:



```python
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
```

We then connect to the broker, which, since we are running this on the laptop, will be at localhost.



```python
print("Connecting")
client.connect("localhost", 1883, 60)
client.loop_forever()
```

The broker runs by default at port 1883 (the second parameter). The third parameter (60) tells the client the interval between “keepalive” messages, which keep the connection to the broker alive.

Finally we call “loop_forever” so that our program doesn’t exit, and we can wait for messages.

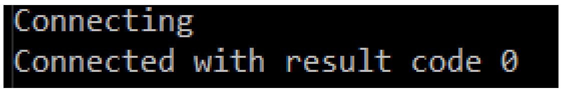
b. <u>Running Our Client Code</u>

We will now run our code to see what it does.

i. Ensure that mosquitto is running. Start mosquitto if it is not (see Step 5).
ii. Have at least two other terminals open.
iii. Run our program in one terminal:

```
python mqtt.py
```

If mosquitto is running and all goes well, you should see:

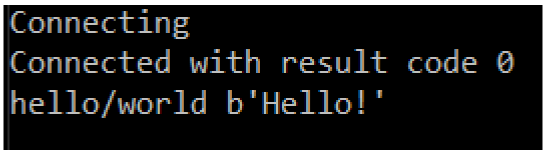


```
Connecting
Connected with result code 0
```

iv. Now in the other window, send a message over:

mosquitto_pub –h localhost –t hello/world –m "Hello!!"

v. If all goes well you should see:



```
Connecting
Connected with result code 0
hello/world b'Hello!'
```

QUESTIONS

Question 1. **(2 MARKS)**

Notice that there is a stray "b" attached to our Hello! message. Using Google or otherwise, explain what this "b" means and why it is present in our output.

Question 2. **(3 MARKS)**

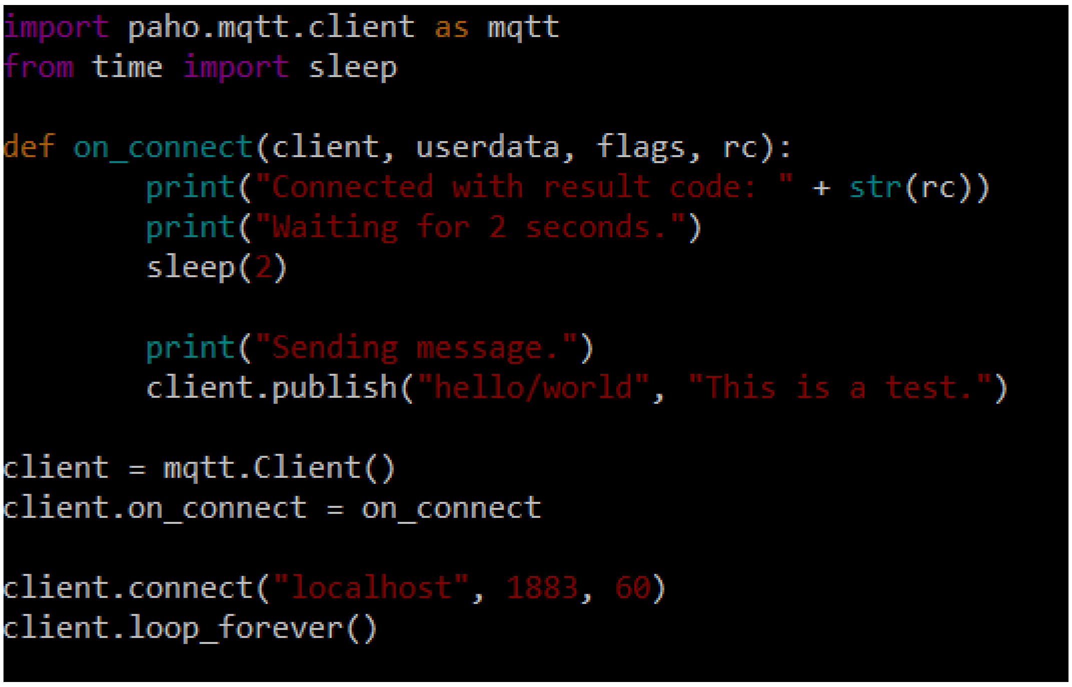
Using Google or otherwise, modify your program so that this "b" no longer appears. Paste your code into your answer book.

c. Publishing Messages

Paho-MQTT allows us to publish messages using the "publish" call:

```
client.publish(topic, payload)
```

To see how to use this, create a file called "pub.py" and enter the following code:



```
import paho.mqtt.client as mqtt
from time import sleep

def on_connect(client, userdata, flags, rc):
        print("Connected with result code: " + str(rc))
        print("Waiting for 2 seconds.")
        sleep(2)

        print("Sending message.")
        client.publish("hello/world", "This is a test.")

client = mqtt.Client()
client.on_connect = on_connect

client.connect("localhost", 1883, 60)
client.loop_forever()
```

This code should be fairly self-explanatory.

Now:

i. Ensure that Mosquitto is running.

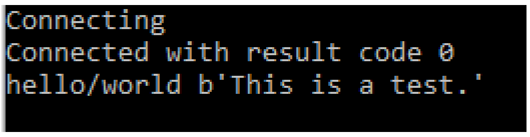
ii. Have two terminals open. In one, start the mqtt.py program that we wrote earlier:

```
python mqtt.py
```

iii. In the other, start this program:

```
python pub.py
```

iv. If all goes well, after two seconds pub.py will send a message over to mqtt.py and on mqtt.py you will see:



```
Connecting
Connected with result code 0
hello/world b'This is a test.'
```

Note: Here the 'b' still appears. It should not appear in your version.

5. <u>INTEGRATING WITH KERAS</u>

We will now create a classification server over MQTT. To begin we will look at how to transfer a binary file from your client to your server. Here the MQTT broker's address is assumed to be 192.168.0.1. You should replace this with your broker's actual IP address.

In our example we will be classifying images.

Step 1. <u>Designing Your MQTT Messages</u>

We start first by designing our MQTT messages.

All MQTT messages require a topic, and we will use this format:

Group_<group number>/IMAGE/<action>

Our set of actions are:

classify: Request to classify an image.
predict: Return predictions for image.

Thus if you are from group 99, your message topics will be:

Goup_99/IMAGE/classify
Group_99/IMAGE/predict

We now need to decide our payloads. For "classify" this is straightforward: we create a dictionary containing:

filename:	Name of the picture file
data:	Actual picture data.

Why do we include the filename? This is because the MQTT send and receive operation is asynchronous. This means that we can send out many images for classification without actually waiting for the results to come back. This means that when the results return, we may not have a way of reconciling results to the actual pictures. Thus we send over the filename (or some sort of ID), which the server will send back together with its classification to allow us to do this reconciliation. This is particularly critical if either your client or server (or both) is multithreaded.

For "predict" we return a dictionary containing:

filename:	Name of the picture file
prediction:	Prediction
score:	Confidence in the prediction
index:	Index for the prediction. E.g. 0:Roses, 1:Sunflowers, etc. This has to be agreed upon on both sides.

So if you send a file called "flower1.jpg" over to the classification server, it might return:

filename:	flower1.jpg
prediction:	Tulips

score:	0.98
index:	4

In summary our MQTT messages will look like this:

Topic	Payload
GROUP_<groupnumber>/IMAGE/classify	{filename: <filename>, data: <data>}
GROUP_<groupnumber>/IMAGE/predict	{filename: <filename>, prediction:<prediction>, score:<score>, index:<index>}

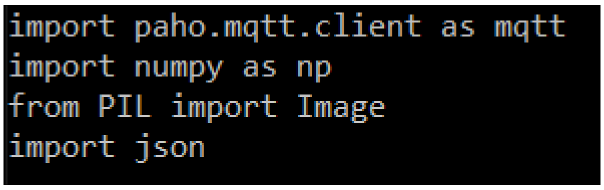
Now that we have created our message structure (i.e. "protocol"), we can now begin coding.

Step 2. Building the Sender

We begin first by building the program to send a picture of a flower for classification. Create a file called "send.py" and enter the following:

a. Imports

As always we begin with the exciting work of importing what we need. For this demo it is quite straightforward. We import the MQTT client, numpy and json. We will use numpy.tolist() to turn a numpy array into a list of numbers, and json to turn our Python dictionary into a JSON string and back, for transmission. Reasons for these two steps will be given later. We also import the Image class from Pillow (PIL) which allows us to load images into a NumPy array. In any case:



```
import paho.mqtt.client as mqtt
import numpy as np
from PIL import Image
import json
```

b. Declaring the on_connect Function

This part is straightforward. We check if rc is 0, and if so we say that we have connected successfully. We then subscribe to GROUP_xx/IMAGE/predict (we assume we are Group 99, please replace with your own group number), which is the topic the server is going to publish results with.

```
def on_connect(client, userdata, flags, rc):
        if rc == 0:
                print("Connected.")
                client.subscribe("Group_99/IMAGE/predict")
        else:
                print("Failed to connect. Error code: %d." % rc)
```

c. <u>Declaring the on_message Function</u>

As always we declare our on_message function. As per Step 1 we expect our server to return its classification in the form of a dictionary. However MQTT can only transmit byte streams, numbers and strings, and thus cannot transmit Python dictionaries. For this reason we convert it into a JSON string using JSON.dumps, and convert it back to a Python dictionary using JSON.loads, which is what's happening here.

We then print out the results:

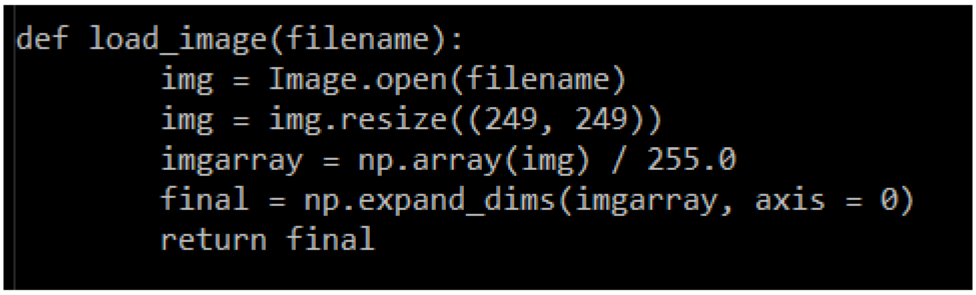
```
def on_message(client, userdata, msg):
        print("Received message from server.")
        resp_dict = json.loads(msg.payload)
        print("Filename: %s, Prediction: %s, Score: %3.4f"
% (resp_dict["filename"], resp_dict["prediction"], resp_dict["score"]))
```

d. <u>Setting Up The Connection</u>

For neatness we declare a setup function to set up the connection. This part should be easily understood, based on the MQTT lab:

```
def setup(hostname):

        client = mqtt.Client()
        client.on_connect = on_connect
        client.on_message = on_message
        client.connect(hostname)
        client.loop_start()
        return client
```

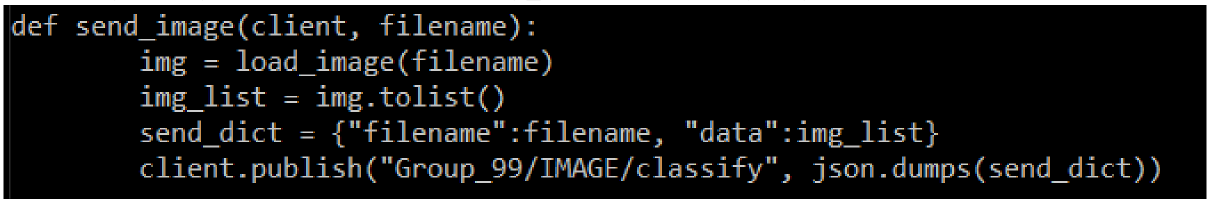
e. <u>Loading the Image</u>
Now we use the Image.open function from Pillow to load the image as a NumPy array. We resize the image to 249 x 249 pixels to suit the training on Inception (which we will use later on), and scale it from 0 to 1. We add one more column using expand_dims since Inception expects the first dimension to be the batches of images.



```
def load_image(filename):
        img = Image.open(filename)
        img = img.resize((249, 249))
        imgarray = np.array(img) / 255.0
        final = np.expand_dims(imgarray, axis = 0)
        return final
```

f. <u>Sending the Image</u>

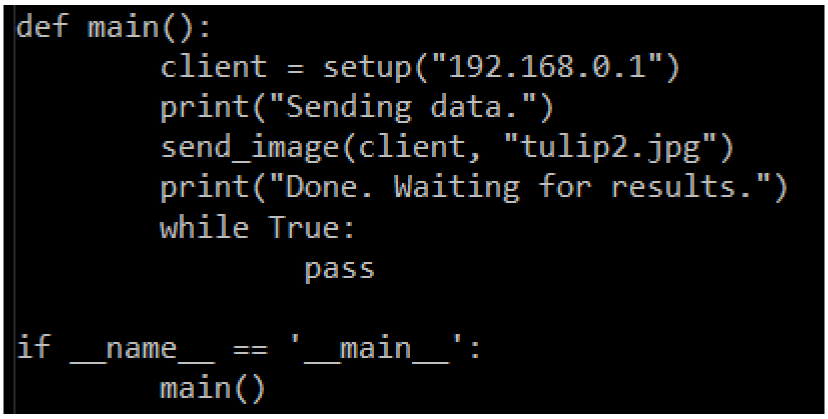
Now that we can load our images, we create a send_image function that takes in an MQTT client and the name of the image file, calls load_image to load the image. As mentioned earlier we will create a Python dictionary, and to send this dictionary we must turn it into a JSON string using json.dumps, which cannot handle NumPy arrays. Hence we call the tolist() function to turn it into a Python list, which json.dumps can convert. We create the dictionary, then publish it under the "Group_xx/IMAGE/classify" topic:



```
def send_image(client, filename):
        img = load_image(filename)
        img_list = img.tolist()
        send_dict = {"filename":filename, "data":img_list}
        client.publish("Group_99/IMAGE/classify", json.dumps(send_dict))
```

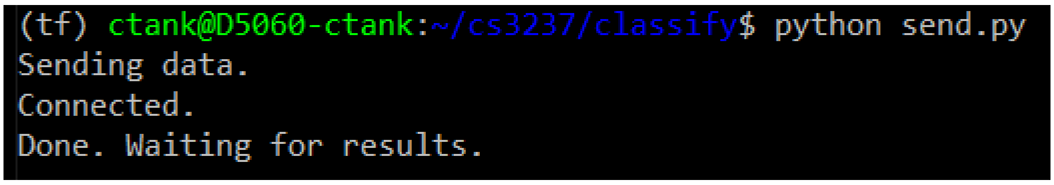
g. <u>Creating Main</u>

Finally we create our main routine which set up the connecting, send the "tulip2.jpg" image for classification, then loop indefinitely:



```
def main():
        client = setup("192.168.0.1")
        print("Sending data.")
        send_image(client, "tulip2.jpg")
        print("Done. Waiting for results.")
        while True:
                pass

if __name__ == '__main__':
        main()
```

Ensure that tulip2.jpg is in the same directory as send.py. If everything goes well, when you run the program with "python send.py" you should see:



```
(tf) ctank@D5060-ctank:~/cs3237/classify$ python send.py
Sending data.
Connected.
Done. Waiting for results.
```

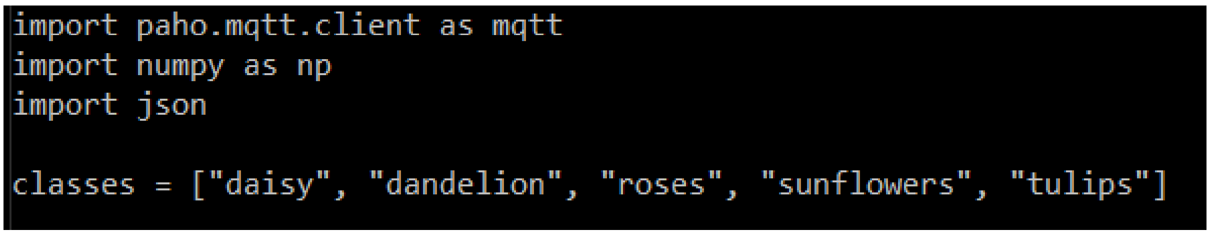
It will wait indefinitely since we don't have a server yet. We will address this now. Meanwhile hit CTRL-C to exit the send.py program.

Step 3. Creating the Server

We will now create the server side. Create a file called “receive.py” and enter the following:

a. Imports

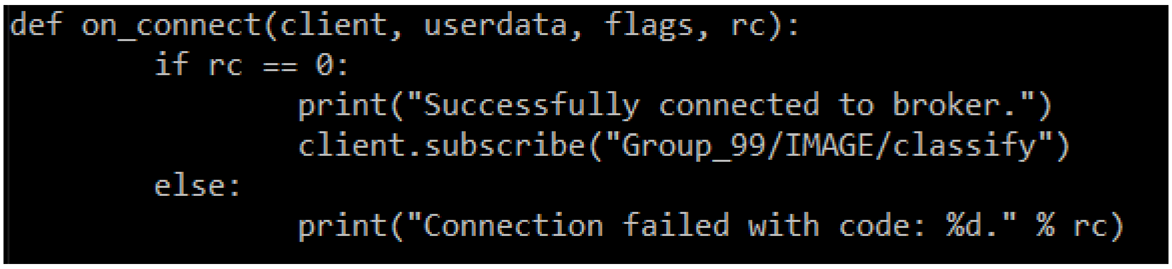
We begin as always with the imports. The server side is more straightforward since it doesn’t have to load images, so all we need are the MQTT library, NumPy and json. We also declare our classes of flowers as a global array, for use later on:



```
import paho.mqtt.client as mqtt
import numpy as np
import json

classes = ["daisy", "dandelion", "roses", "sunflowers", "tulips"]
```

b. The on_connect Function

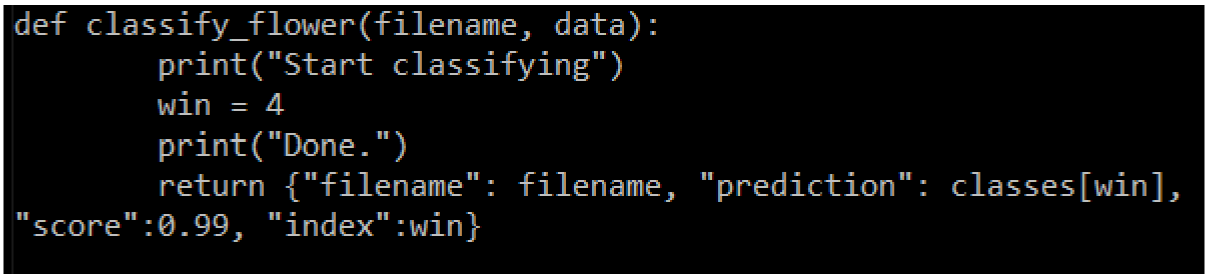
Similar to what we had done for send.py, except that we subscribe to Group_xx/IMAGE/classify instead:



```
def on_connect(client, userdata, flags, rc):
        if rc == 0:
                print("Successfully connected to broker.")
                client.subscribe("Group_99/IMAGE/classify")
        else:
                print("Connection failed with code: %d." % rc)
```

c. Classification

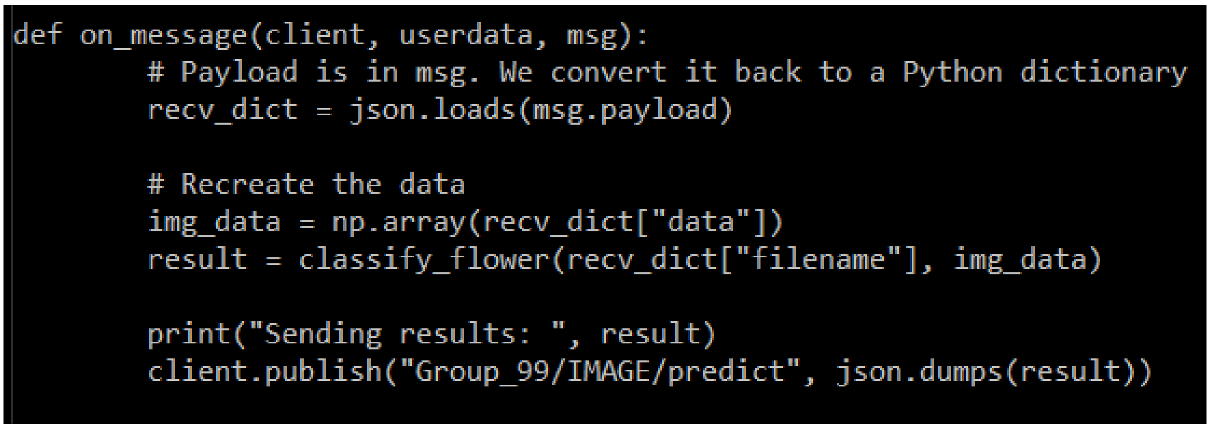
We declare a function that takes a filename and image data in the form of a NumPy array, then classifies it, and returns a dictionary in the format we designed in Step 1. Here this function is a dummy that always returns the same result. You will change this to return the classification from the Inception network you built in the previous lab.



```
def classify_flower(filename, data):
        print("Start classifying")
        win = 4
        print("Done.")
        return {"filename": filename, "prediction": classes[win],
"score":0.99, "index":win}
```

d. The on_message Function

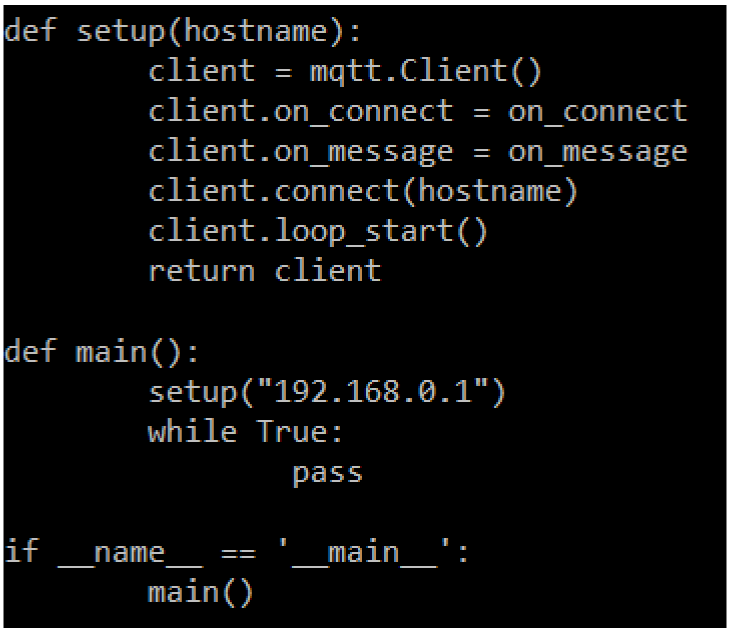
Our on_message function will first turn the JSON string from msg.payload back into a proper Python dictionary, then turn the list of numbers in the “data” field back into a NumPy array using np.array. It calls classify_flower with the filename and image data, then publishes the result using the Group_xx/IMAGE/predict topic. As always the dictionary returned by classify_flower needs to be turned into a JSON string using json.dumps:



```
def on_message(client, userdata, msg):
        # Payload is in msg. We convert it back to a Python dictionary
        recv_dict = json.loads(msg.payload)

        # Recreate the data
        img_data = np.array(recv_dict["data"])
        result = classify_flower(recv_dict["filename"], img_data)

        print("Sending results: ", result)
        client.publish("Group_99/IMAGE/predict", json.dumps(result))
```

e. **<u>Making the Connection</u>**

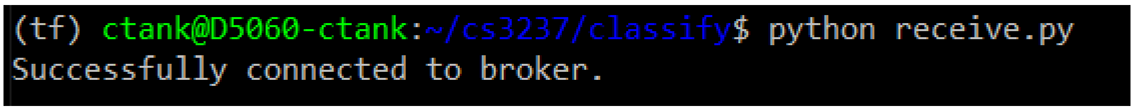
As before we create a setup function to connect to the broker, together with the main function:



```
def setup(hostname):
        client = mqtt.Client()
        client.on_connect = on_connect
        client.on_message = on_message
        client.connect(hostname)
        client.loop_start()
        return client

def main():
        setup("192.168.0.1")
        while True:
                pass

if __name__ == '__main__':
        main()
```

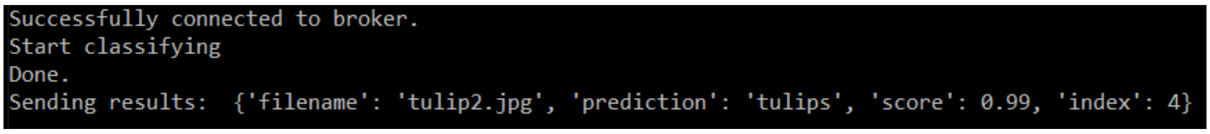
Step 4. <u>Running Our Classification Server</u>

Ensure that the send.py program is not running (it has to be run first). Now create two terminals. In one terminal run receive.py by typing “python receive.py”. You will see:



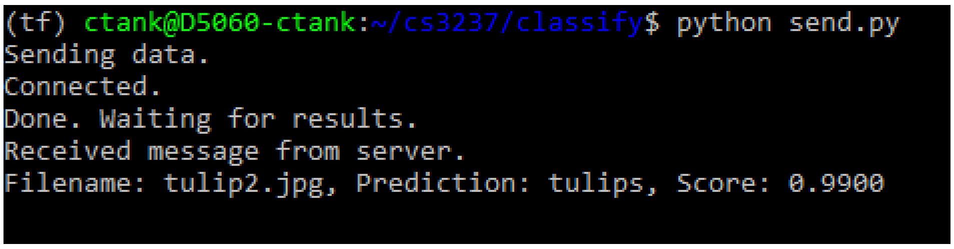
```
(tf) ctank@D5060-ctank:~/cs3237/classify$ python receive.py
Successfully connected to broker.
```

In the second terminal ensure that tulip2.jpg is in the same directory as send.py, then type “python send.py”. If all goes well you should see on the receive.py side:



```
Successfully connected to broker.
Start classifying
Done.
Sending results:  {'filename': 'tulip2.jpg', 'prediction': 'tulips', 'score': 0.99, 'index': 4}
```

And on the send.py side:



```
(tf) ctank@D5060-ctank:~/cs3237/classify$ python send.py
Sending data.
Connected.
Done. Waiting for results.
Received message from server.
Filename: tulip2.jpg, Prediction: tulips, Score: 0.9900
```

Step 5. Doing Actual Classification

You now need to modify your receive.py program to actually do classification. To start:

a. Preparation

Ensure that:

i) send.py and tulip2.jpg are in the same directory.
ii) Have your receive.py and flowers.hd5 are in the same directory.

b. Building your Classifier

The classify function in receive.py at the moment doesn't do anything useful at all. We will now fix this.

Tips:

1. It takes a very long time to load a model. Therefore you should load your model only once and use the same model repeatedly to do your classification.

2. As per section 3h, you need to set_session to ensure that Keras and Tensorflow work correctly within MQTT. Please refer to section 3h for details, but for your convenience a summary is shown here:

```
import tensorflow as tf
from tensorflow.pythin.keras.backend import set_session

session = tf.compat.v1.Session(graph = tf.compat.v1.Graph())

….

# When calling load_model:

with session.graph.as_default():
      set_session(session)
      # Call load_model

# When predicting
with session.graph.as_default():
      set_session(session)
      # Make prediction
```

Open receive.py and answer the following questions.

QUESTIONS

Question 3 **(2 MARKS)**

Explain, with code snippets, how you have loaded your model to avoid incurring long waiting times for each classification.

Question 4 **(2 MARKS)**

In Section 3 we froze some weights when we loaded our model. Why did we do that? Is this necessary here?

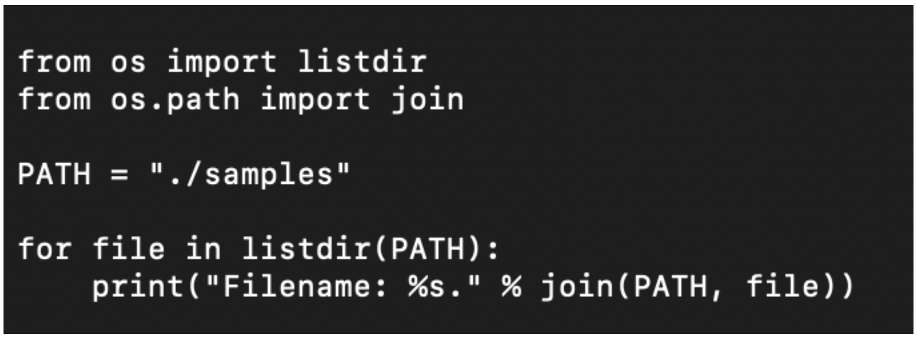
Question 5 **(4 MARKS)**

Cut and paste your new predict function, explaining how it works. No marks will be awarded for code without explanation.

c. Running your Classifier

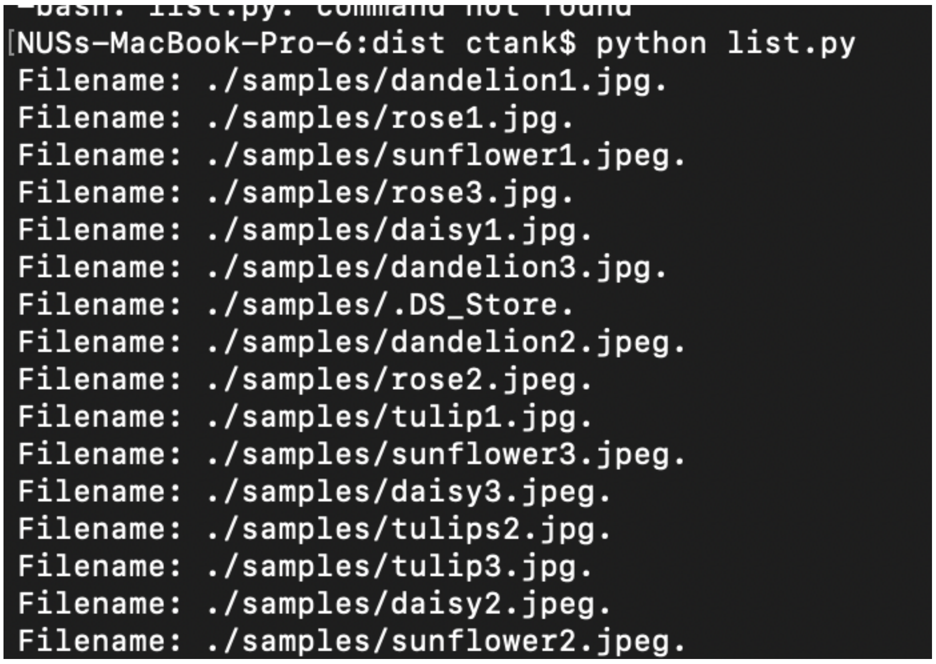
Using the "samples" directory from part 1 (copy the directory over if you have to). Ensure that your send.py program is in the sample directory as the "samples" directory.

The code fragment below shows how to get the name of every file in the "samples" directory.



```
from os import listdir
from os.path import join

PATH = "./samples"

for file in listdir(PATH):
    print("Filename: %s." % join(PATH, file))
```

If you run this in the same directory as "samples", you will see:



```
-bash: list.py: command not found
[NUSs-MacBook-Pro-6:dist ctank$ python list.py
 Filename: ./samples/dandelion1.jpg.
 Filename: ./samples/rose1.jpg.
 Filename: ./samples/sunflower1.jpeg.
 Filename: ./samples/rose3.jpg.
 Filename: ./samples/daisy1.jpg.
 Filename: ./samples/dandelion3.jpg.
 Filename: ./samples/.DS_Store.
 Filename: ./samples/dandelion2.jpeg.
 Filename: ./samples/rose2.jpeg.
 Filename: ./samples/tulip1.jpg.
 Filename: ./samples/sunflower3.jpeg.
 Filename: ./samples/daisy3.jpeg.
 Filename: ./samples/tulips2.jpg.
 Filename: ./samples/tulip3.jpg.
 Filename: ./samples/daisy2.jpeg.
 Filename: ./samples/sunflower2.jpeg.
```

QUESTIONS

Question 6 **(4 MARKS)**

Modify your send.py program to send all the files in the "samples" directory for classification and print out the results.

Cut, paste and explain the relevant sections of code in your answer book. No marks will be awarded for code without explanation.

Question 7 **(3 MARKS)**

Run your send.py program and cut and paste the output to your report. What is the accuracy of your classifier?