SWS3009A Deep Learning

Lab 6 – Classification over MQTT

1. INTRODUCTION

In this lab we will be looking at three topics:

i. MQTT: We look at how to set up MQTT connections.
ii. Integration between MQTT and Keras: We look at how to create a classification server using MQTT and the classifier you built in Lab 4 using Transfer Learning.

2. SUBMISSION

You are to work in TEAMS OF 2. You only need to upload ONE copy of the answerbook to Canvas.

Deadline:

Wednesday 12 July 2023 2359 hours (11.59 pm)

3. MESSAGE QUEUING TELEMETRY TRANSPORT

Now that we have built a classifier for the flowers dataset in Lab 4, let's build a more complete classification 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
```

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. MQTT Topics and Messages

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. Writing MQTT Clients

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. Entering our Paho MQTT Client Code

```python
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:

```python
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.

```
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:

```
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.

```
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. Running Our Client Code

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

<u>Question 1.</u> **(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.

<u>Question 2</u>. **(3 MARKS)**

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

c. <u>Publishing Messages</u>

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.

4. INTEGRATING WITH KERAS

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. Designing Your MQTT Messages

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 (for groupnumber use any value from 00 to 99):

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. Declaring the on_message Function

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. Setting Up The Connection

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. Loading the Image

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. Sending the Image

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. Creating Main

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. **Making the Connection**

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. Running Our Classification Server

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. Tensorflow does not work properly with MQTT. To get Tensorflow to work with MQTT, you need to set_session. The code fragment below shows how to do this.

```
import tensorflow as tf
from tensorflow.python.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

<u>Question 3</u>

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

<u>Question 4</u>

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

c. <u>Running your Classifier</u>

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 5

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 6

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