Sai Ranganathan and Anuraag Polisetty
Net ids: sr50, anuraag5
CS 498: IOT Lab 4

Lab 4: Self-Driving Car - Cloud

Github repository link: https://github.com/sairanga123/CS498_lab4
Link to Machine Learning Box Videos and Files:
https://uofi.app.box.com/s/9g1aavdcd8z6evsnfilnrm5j8ph7uaru
Link to AWS Demo Videos: https://photos.app.goo.gl/1nubLJJH89NUJwrR6

IOT Lab 4 Part 1: Machine Learning

Step 2: Jupyter Notebook

Q1: Jupyter Notebook is beneficial to use over Terminal for python because it can have both markdown and python code to describe the code and explain how it works. In addition, we can also interact with the data and rerun chunks of code to run a section and recreate a graph or plot.

Step 4: Machine Learning

Tensorflow Playground Questions

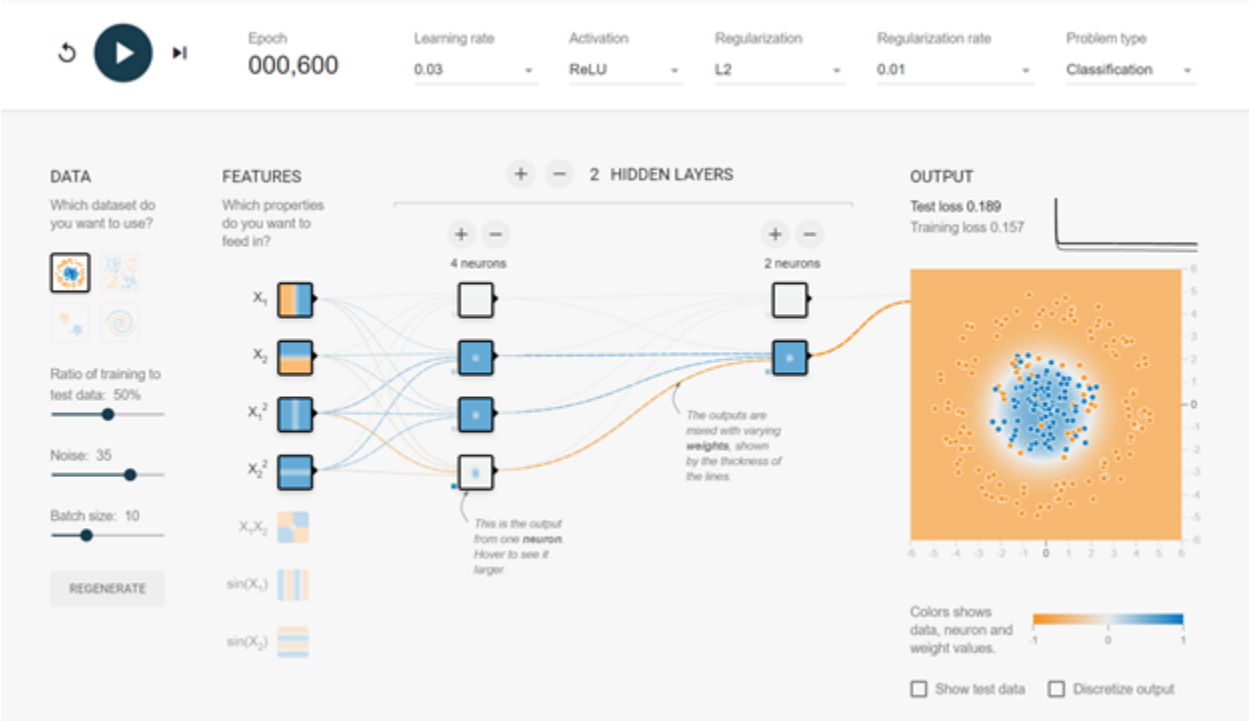
Q1. The activation functions do not all give a good training result. The Sigmoid and Linear activation functions had a training loss of 0.5 while the ReLU and Tanh activation functions only had a training loss of 0.001. This implies that the last two functions are overfitting the training data, while the first two are underfitting it.

Q2. At a learning rate of 10, the training loss is very high at around 0.977 after 600 epochs. I also observed that the entire graph is classified as either orange or blue while flipping between each color every so often. This means that the model might never converge to an accurate classification. With a very small learning rate of 0.00001, the training loss does not change and stays at around 0.5 even after 1000 epochs, which makes the model will converge extremely slowly.

Q3. The weights in each node change over time and also change faster with a higher learning rate. Weights are negative when they are orange and positive when they are blue, signaling that they give more weight to blue points and less weight to orange points.

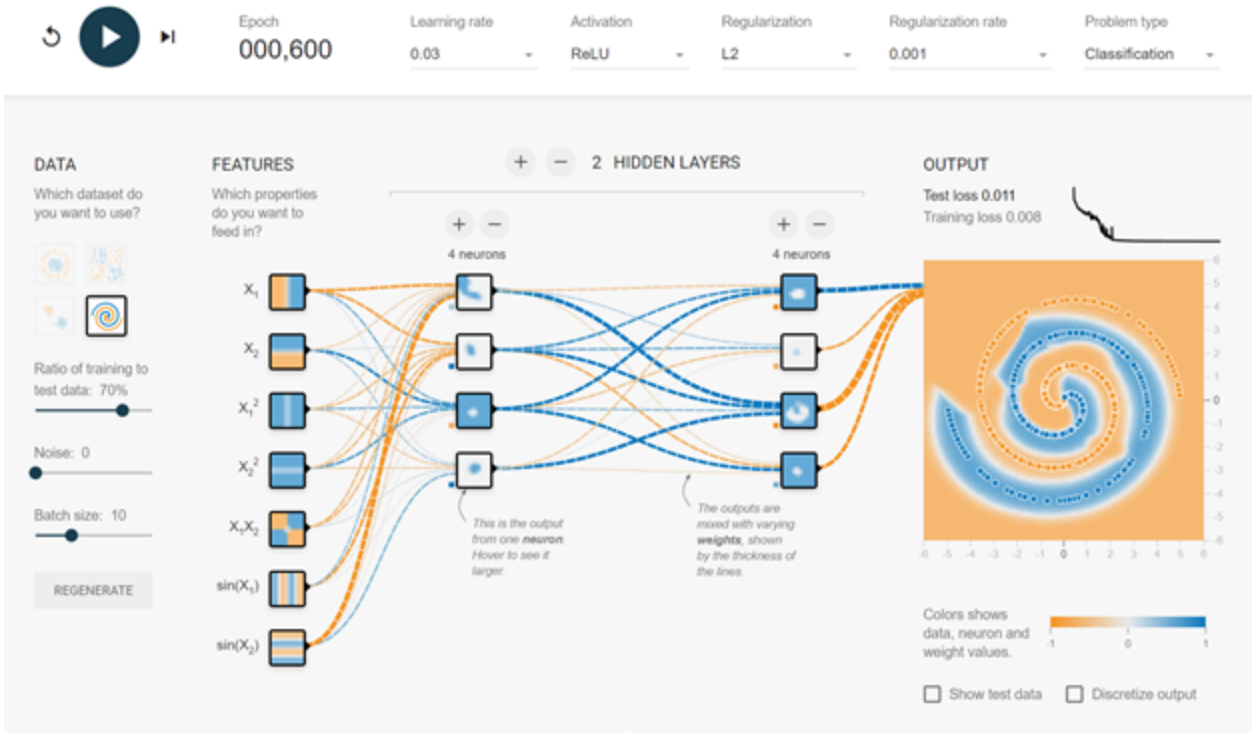
Q4. With the regularization set to 0.003, we ended up with a training loss of 0.146 and a test loss of 0.204. This shows a lower test loss as compared to a regularization rate of 0, which had a training loss of 0.142 and a test loss of 0.211. With a higher

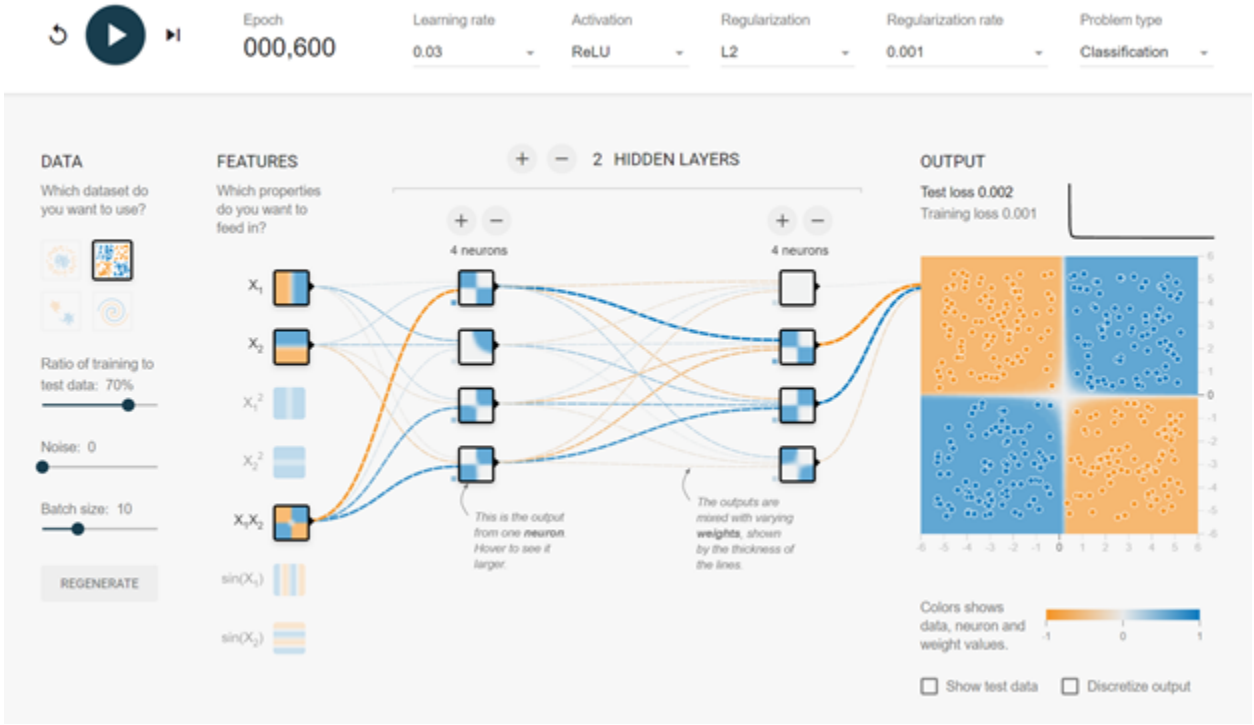
regularization rate, it was observed that the testing result was less overfit to the data and as a result performed better on new testing data.



Q5. Other features are necessary in order to have a more diverse range of points to describe the model with. With only x and y features, the model will be unable to account for curves and spirals in this data. However, by adding in the sin functions and squared functions, the neural network model will be able to better account for classifying the positions of data arranged in a spiral.

Q6. Making a model using the maximum of 6 hidden layers with 8 neurons each severely decreases the speed at which the model can train the new data. In addition, this also hinders its classification accuracy, resulting in a training loss of 0.5 and a test loss of 0.503. However, once I reduced regularization rate to 0.001, it resulted in a training loss of 0.006 and a test loss of 0.03, but still greatly reduced the speed of classification. Using the minimum of 1 hidden layer with 2 neurons we ended up with a training loss of 0.378 and a test loss of 0.364.

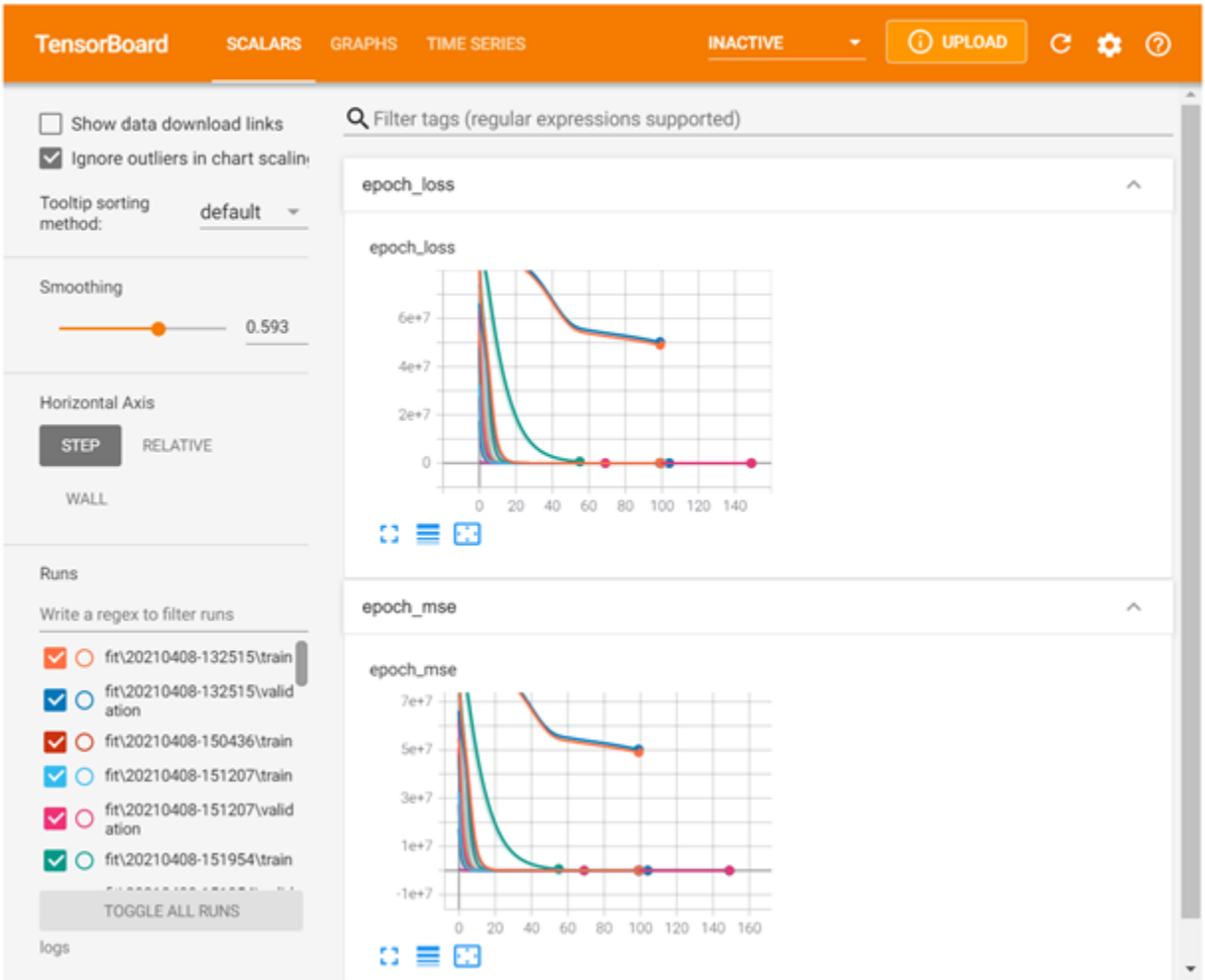




Model Training
Small Dataset:

```
Epoch 145/150
118/118 [==============================] - 1s 5ms/step - loss: 568.6168 - mse: 524.6368 - val_loss: 236.1041 - val_mse: 192.188
8
Epoch 146/150
118/118 [==============================] - 1s 6ms/step - loss: 154.5624 - mse: 110.6560 - val_loss: 102.9400 - val_mse: 59.0549
Epoch 147/150
118/118 [==============================] - 1s 6ms/step - loss: 119.6423 - mse: 75.7839 - val_loss: 1425.3003 - val_mse: 1381.43
81
Epoch 148/150
118/118 [==============================] - 1s 7ms/step - loss: 3304.3742 - mse: 3260.3646 - val_loss: 139.7119 - val_mse: 95.39
67
Epoch 149/150
118/118 [==============================] - 1s 6ms/step - loss: 127.4073 - mse: 83.0946 - val_loss: 95.8112 - val_mse: 51.4875
Epoch 150/150
118/118 [==============================] - 1s 5ms/step - loss: 131.6478 - mse: 87.3310 - val_loss: 166.7221 - val_mse: 122.4162
]: <tensorflow.python.keras.callbacks.History at 0x1a40ae05820>
```

```
%tensorboard --logdir logs
Reusing TensorBoard on port 6006 (pid 4916), started 0:01:21 ago. (Use '!kill 4916' to kill it.)
```



Large Dataset:

```
Epoch 20/30
118/118 - 31s - loss: 386.4644 - mse: 203.5798 - val_loss: 727.6017 - val_mse: 544.9117
Epoch 21/30
118/118 - 31s - loss: 382.3637 - mse: 199.8619 - val_loss: 684.5765 - val_mse: 502.2710
Epoch 22/30
118/118 - 31s - loss: 379.9140 - mse: 197.7966 - val_loss: 681.5717 - val_mse: 499.6420
Epoch 23/30
118/118 - 31s - loss: 374.9483 - mse: 193.2027 - val_loss: 724.6836 - val_mse: 543.1226
Epoch 24/30
118/118 - 31s - loss: 372.0387 - mse: 190.6625 - val_loss: 752.3967 - val_mse: 571.2098
Epoch 25/30
118/118 - 31s - loss: 368.4995 - mse: 187.4949 - val_loss: 611.1083 - val_mse: 430.2860
Epoch 26/30
118/118 - 31s - loss: 366.9173 - mse: 186.2801 - val_loss: 575.3104 - val_mse: 394.8569
Epoch 27/30
118/118 - 31s - loss: 367.1964 - mse: 186.9261 - val_loss: 760.8565 - val_mse: 580.7650
Epoch 28/30
118/118 - 31s - loss: 362.0042 - mse: 182.0803 - val_loss: 674.0829 - val_mse: 494.3303
Epoch 29/30
118/118 - 30s - loss: 358.5566 - mse: 178.9769 - val_loss: 693.3025 - val_mse: 513.9009
Epoch 30/30
118/118 - 31s - loss: 358.4848 - mse: 179.2530 - val_loss: 633.8889 - val_mse: 454.8224

<tensorflow.python.keras.callbacks.History at 0x2df75287940>
```

IOT Lab 4 Part 2: Cloud

Section 1:

In this section, we started by building the cloud. The first step is to get some basic background on the cloud. We learned concepts such as the MQTT protocol, pub/sub model, topics, brokers, rules and what lambda functions are. In the past we had used TCP based client-server communication model in order to send the messages from the raspberry pi to our computer acting as a client server. In this lab, however, we used the MQTT pub/sub model and we have to ask ourselves what are the pros and cons of such a communication method. Well the first pro is that this protocol ensures message delivery through its quality of service method. Quality of Service is different levels in which you can choose as to how secure and thoroughly you want your messages to be sent from the client to server or in this case the publisher to the subscriber. This quality of service attribute will be able to make sure that the messages get delivered and might send the message more than once if it gets stuck in the message queue ever. Another pro of MQTT is that it is lightweight as it is only 2 bytes and few lines of code in comparison to HTTP protocols. MQTT when paired with devices is also extremely efficient and battery friendly as it was developed by IBM to be used out in the desert with no contact to electricity grids. All that being said, there are are also cones when using this MQTT based pub/sub model. MQTT is not very friendly to the developer as it

is asynchronous and will not tell you exactly when the message has been sent. Due to this, there are issues with latency as things can get stuck in the message queue.

MQTT can generally handle a large number of clients. There have been cases reported where a single broker was able to send messages to more than millions of clients. The tradeoff comes in the quality of service. If the quality of service is set to at least once, there can be times that some latency can cause the message to be retried by the sender multiple times and acknowledgement happens asynchronously and so this might just deliver the same message way to many times to one client. When it is many clients and there is high traffic, this can cause problems in the entire network.

In the second part of section 1, we were able to set up an AWS account and set up AWS IOT. In order to set up a device to act as the IOT core for AWS Greengrass, we had created a ec2 instance that would act as the physical device that the core would live in and interact with multiple devices. For the

Figure 1. Things created on AWS IoT

third part, we created things which would then serve as a registry for the physical device in the IOT core. Then we created and configured rules and created many devices using the script that was provided. After, we were able to modify the emulator code and get it to run.

The last part of section 1 was to set up AWS Greengrass. The first thing that we had gotten to do was spin up an ec2 instance rather than use the raspberry pi as our physical device running as the Greengrass Core. After, we had created and deployed subscriptions on the Greengrass Core. We then created Lambda functions which would then be configured with our Greengrass Core. Look below for screenshot of the Lambda

function that was created:

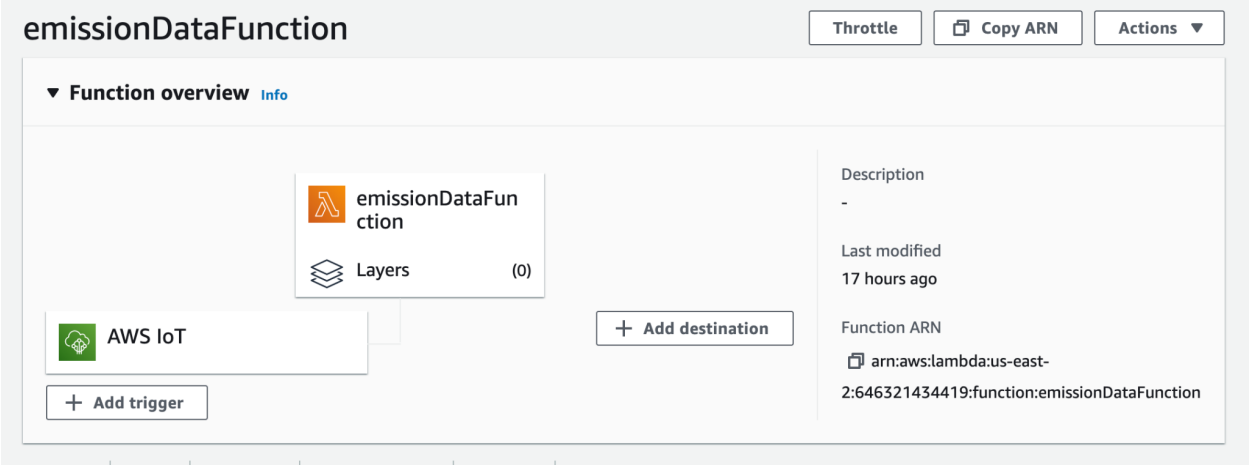


Figure 2. Lambda Function created for the emission data

After creating the Lambda function, we were able to create multiple devices, a publisher and a subscriber which would communicate through a topic. We created three cert files on our laptop as a device in the Greengrass group. As a part of step 5 and step 6, we were able to send messages from the publisher to the describer and show that we were able to receive the messages on the other side. The data we sent was some sample data from the emission data. You can see the live demo

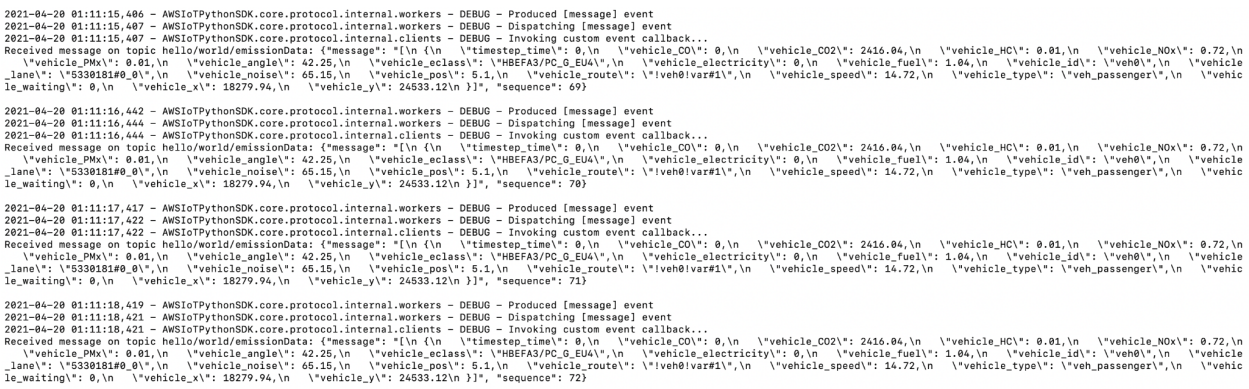


Figure 3. The receiver device getting back the JSON from the publisher

Section 2:

In this section we used a lambda function in order to process the incoming messages. By using our emulator, we are able to access the vehicles csv files along with their private keys in order to publish the data to the lambda function through the AWS IOT Core. Here below is a screenshot of us being able to send data to the lambda function

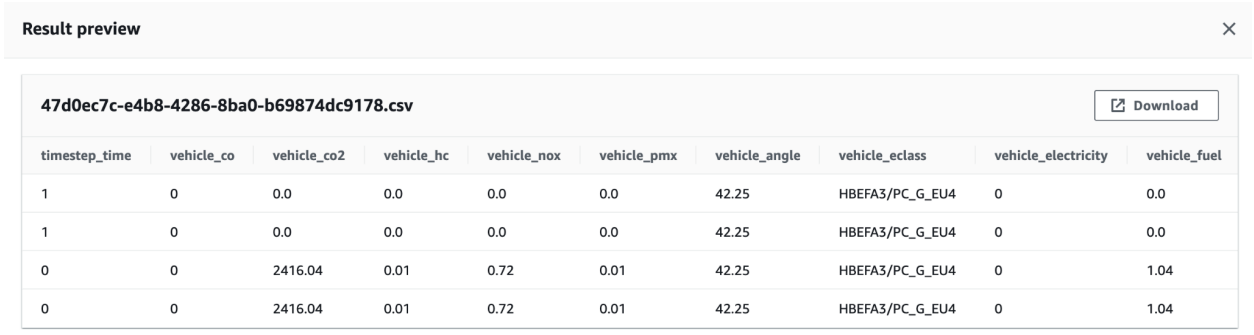
using a forwarding rule. The rule sends the data to the lambda function to process the data via the topic that we have established. In this case, we use the topic that we named called ‘hello/world/emissionData.’ Here is a screenshot of the results from the emulator that is then sent to the service.py in our lambda function.

```
done
[sairanga123@Sais-MacBook-Pro lab4 % python3 lab_4_emulator_client.py
wait
Users at state 1:  []
Users at state 2:  []
Users at state 3:  [0]
Users at state 4:  [4]
send now?
s
done
```

Figure 4. Emulator sending data from vehicle to the lambda function

Section 3:

In this section, we started using AWS IOT Analytics in order to analyze and make sense of our data from the vehicle emission files. These csv files contain information about the emissions of 5 different vehicles and by using Analytics we were able to create visualizations about the different kinds of data points available to us. One could then use these visualizations to draw any kind of conclusions about the vehicles and see what they can do to improve emissions overall.



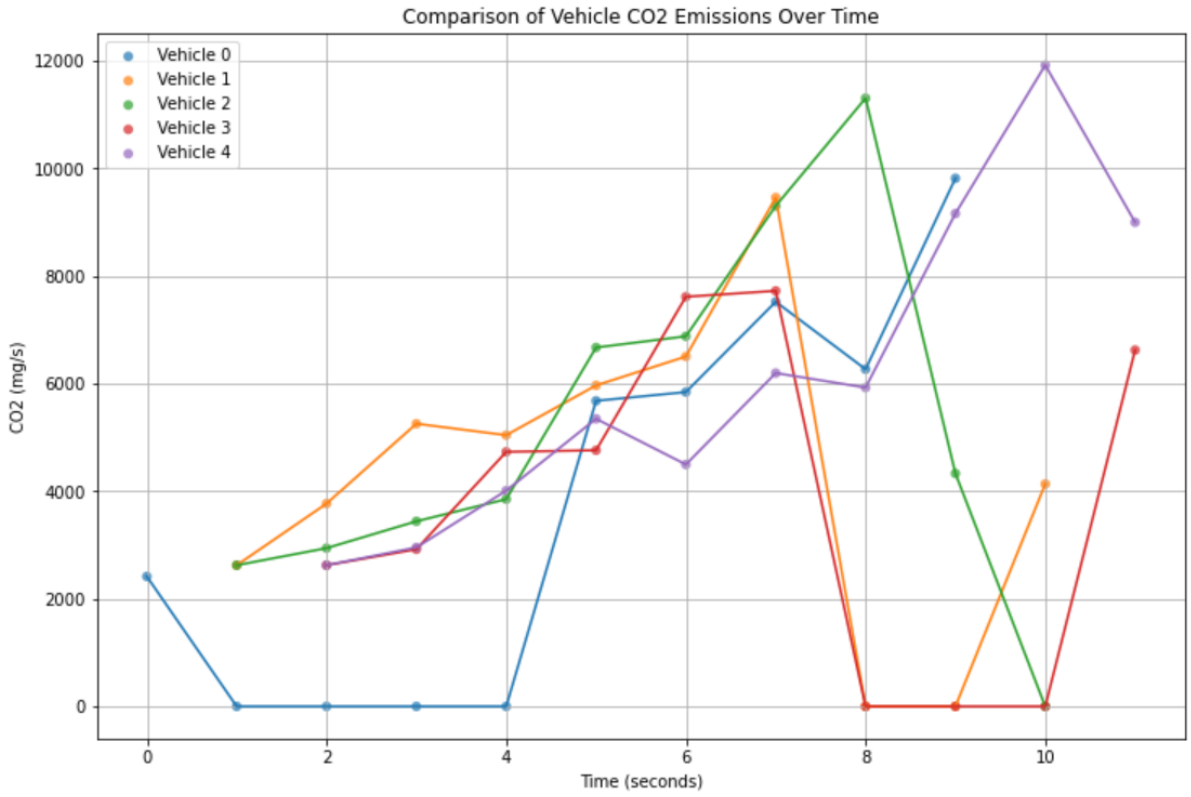
timestep_time	vehicle_co	vehicle_co2	vehicle_hc	vehicle_nox	vehicle_pmx	vehicle_angle	vehicle_eclass	vehicle_electricity	vehicle_fuel
1	0	0.0	0.0	0.0	0.0	42.25	HBEFA3/PC_G_EU4	0	0.0
1	0	0.0	0.0	0.0	0.0	42.25	HBEFA3/PC_G_EU4	0	0.0
0	0	2416.04	0.01	0.72	0.01	42.25	HBEFA3/PC_G_EU4	0	1.04
0	0	2416.04	0.01	0.72	0.01	42.25	HBEFA3/PC_G_EU4	0	1.04

Figure 5. Dataset digested from MQTT

Figure 6: Visualization of Vehicle Data in Jupyter Notebook

```
fig, ax = plt.subplots()
colors = ['blue', 'orange', 'green', 'red', 'purple']
p = 0
fig.set_size_inches(12,8)
for data in datasets:
    time = data['timestep_time']
    co2 = data['vehicle_CO2']
    ax.scatter(time, co2, c="tab:{}".format(colors[p]), label="Vehicle {}".format(p), alpha=0.7, edgecolors='none')
    plt.plot(time, co2)
    p = p + 1
ax.legend()
ax.set_ylabel('CO2 (mg/s)')
ax.set_xlabel('Time (seconds)')
ax.set_title("Comparison of Vehicle CO2 Emissions Over Time")
ax.grid(True)

plt.show()
```



Problems Encountered in the Lab:

One of the big problems that we had encountered in the lab was with the emulator and sending the data to the lambda function. We had configured the service.py correctly as to our knowledge but there were errors still happening with the Lambda function. We could see that the invocations were occurring when we had sent the data to the lambda function meaning that the AWS Iot was doing the job correctly, but we were getting

errors according to the CloudWatch Logs that it was unable to import the module ‘service.’

Measures used to Debug Issue:

We had seen that the lambda function was throwing us the first error when we had set the handler name to lambda_function.lambda_handler. We then realized that we needed to change the name to service.lambda_handler as that is where the service.py file was. However, we were unable to understand what the import issue was and as there are many intermediate steps in the process of sending the messages to the Lambda function, there were many places where an error could have occurred.

Individual Contribution of Group Members:

We had both worked together on this lab and had tried to call with a screen share to do pair coding. Anuraag had taken the upper hand on sharing and doing the coding for machine learning via jupyter notebook on the first part of the lab, while Sai had taken the upper hand on the second part of the lab regarding setting up the AWS IOT Greengrass and connecting it to the client devices to send messages.