Sensor Fault Detection and Prediction

By Salman Alfarisyi (salmanalfarisyi087@gmail.com)

Introduction

The following code can detect and predict sensor faults from the Q-Car and processes them on a server that is on the desktop. It creates a GUI which appears on the server-side and allows the user to see the performance of the algorithm and monitor the sensor values in real-time. This code runs on Python 3.7 and is meant to be run sort of like an application that is deployable. It contains a server-side and a client-side program. The server side is intended for the desktop while the client side is meant for the Q-car.

Prerequisites

The software requires the following libraries to be installed:

Server-Side:

- TensorFlow 1.15.0
- Pandas 1.1.5
- Matplotlib 3.4.2
- Quanser libraries (API, common, communications, devices, hardware, multimedia) 2020.12.9
- PySimpleGui 4.45.0
- Torch 1.7.0 (For GPU accelerated version its 1.7.0+cu101)
- NumPy 1.18.5
- Sckit-learn 0.24.2
- Sci-py 1.7.0

Client-Side:

- NumPy 1.18.5
- Quanser libraries (API, common, communications, devices, hardware, multimedia) 2020.12.9

Usage

Client-side

Install the prerequisites on the Q-car as mentioned before. Then place the files into the Q-car in any folder. Inside each file, there is a string variable called serverIP. Ensure that the IP address of the server is written in this variable. There are three files for the client. Here is the function of each:

- "sensor_probe_client.py" is the client-side that does not apply any degradation to the sensor signal.
- "sensor_probe_client_erratic.py" is a client-side file that applies a random erratic value to the gyroscope value.
- "sensor_probe_cliend_lin_err.py" is a client-side file that applies a linear degradation into the random erratic value of all sensors. This means that the scale of the random erratic value increases linearly over time.
- "sensor_probe_cliend_sin_err.py" is similar to "sensor_probe_cliend_lin_err.py" but instead of a linear degradation, it's sinusoidal degradation.

Server-side

The server prerequisites should be installed into the desktop that is on the same local network as the Q-car. In “server.py”, there is a variable called myServer which contains a string that contains the Q-car’s IP address. Write the Q-car’s IP address into this field. Once the correct IP address is written, then the server can be started.

To run the program, execute the “server.py” file. This will set up the server and get it up and running and a loading screen will pop up as it loads the server. Once the main GUI appears and it says “Ready”, then start up the client-side file to initiate the connection.

Once it connects, the server will say “Connected!” and the graphs should start showing the sensor data as well as the health score of the car’s sensor. This is the health index and a health index above 0 is considered a healthy car.

After about 4 minutes, the server should start creating predictions for the health index for the next two minutes. These predictions should keep updating as more health indexes are generated.

Tips

- For the best performance, use a server with a dedicated Nvidia GPU with CUDA 10.1 installed. Once CUDA is properly installed, the server should automatically use the available GPU. The server can be used without a GPU, but the performance may be slow.
- Start the server and client sides as quickly as possible to ensure that they connect. If they don’t connect, you may have to restart both sides.

How it works

Two neural networks are used on the server-side, a detector network, and a prediction network. Both networks are pre-trained and are saved in the libs folder. The detector network is used to generate the health index of the car from the incoming sensor data. This work is based on the work made by Safavi et.al (Accessible by https://www.mdpi.com/1424-8220/21/7/2547). and is a convolutional neural network that uses the sensor data from the client to create a health score that represents the health of the sensors in the car. Then a of health indexes of length 100 will be sent to the prediction network to predict the next 50 health indexes. To learn more about the algorithm, I highly recommend reading Safavi et.al. paper about it.

Some configuration is available with the software. The a2d2.py file which is inside the data_formatters folder which is inside the libs folder of the server-side program contains a function called “get_fixed_params”. This function contains the settings of how many health indexes it requires before it can start predicting future health indexes. “total_time_steps” contains the total amount of health indexes needed to start predicting data and the number of health indexes we want to predict. For example, if we want to use 100 health indexes for prediction and we want to predict the next 100 health indexes, we set “total_time_steps” to 200 because 100+100=200. “num_encoder_steps” specifies the total amount of health indexes we want to use as data for the prediction. So, using the example from earlier, we set “num_encoder_steps” to 100 because we want to use 100 health indexes to predict the next 100 health indexes.

Currently, there are a lot of hard-coded 100s and 150s as part of preparing the data required for prediction. These can be found in server.py and predictionNet.py. If you are changing the prediction health indexes, I highly recommend changing these values.

If you have any more questions, feel free to contact me.