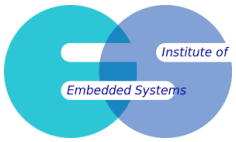
PROJECT WORK

Deep Learning Applications on the Sensor Parameters of a Drone Running in SITL

Author:
Emin Ç. NAKILCIOĞLU
Matr. No. 21756438

Supervisor:
Fin Hendrik BAHNSEN, M. Sc.
Examiner:
Prof. Dr.-Ing. Görschwin FEY



Computer Engineering
Institute of Embedded Systems

September 25, 2019

Contents

Chapter 1

Introduction

Usage of the unmanned air vehicles (UAV) and drones has started to widely spread in various aspects of the life. The purpose of their usage varies from recreational to academical goals. The developments that have been made in this industry in the recent past have played a major role in this ever-lasting increase of their usage. It would be fair to say that there is a significant number of advancements emerged from academical researches and they continue to grow [4]. Furthermore, the implementation of autopilots is another focus among researchers and companies. The problem arises from the need of anticipation of possible future malfunctioning in the internal system of UAVs that run on autopilot since internal malfunctions are considerably difficult to detect by an outsider and furthermore, an autopilot system cannot foresee possible future malfunctions that might be fatal to the vehicle. In order to solve this problem, an improvement for the unmanned air vehicles with autopilots with respect to their current incompetence in terms of working out the fatal anomalies that may occur during a flight is needed. The solution consists of two main phases.
The first step of the solution is to develop a system which can successfully accomplish the desired outcome on a simulation level and therefore, form a basis for the following phase. The next step is to turn the system into an on-board system that can function in real time and learn to adopt to the anomalies or malfunctions which can occur during the flight in real time, as well. This project focuses on the first phase of the solution. For this purpose, a passive monitoring system with an artificial neural network architecture is proposed. The network is trained on the dataset that consists of sensor data extracted from a simulation scenario and is expected to predict the next move of the drone with high accuracy given the dataset from a different simulation scenario.
The project requires a detailed plan to follow up the development process with firm steps (see Table 1.1). First of all, the multicopter is simulated and the data drawn from the simulation is pre-processed. As for the next step, an artificial neural network (ANN) architecture is designed and then sent to train on the datasets. In the training session, accuracy of the architecture trained on one particular dataset is tested on a validation dataset that is unfamiliar to the trained model. By this way, the reliability of the ANN architecture is put to the test. After the experiment is completed,the results are examined and the experiment is concluded.

TABLE 1.1: Project Plan

Stages of the Project
1. Simulation and Data Processing
2. Design of the ANN architecture
3. Experiment
4. Experimental Results and Conclusion

Chapter 2

Theoretical Background

In this chapter, the theories and concepts that were utilized and inspired by in order to design the neural network architecture are explained in a comprehensive manner.

2.1 Neural Networks(NNs)

Neural networks are composed of nodes or units (see Figure 2.1) connected by directed links. A link from unit i to unit j serves to propagate the activation a_i from i to j. Each link also has a numeric weight $w_{i,j}$ associated with it, which determines the strength and sign of the connection. Just as in linear regression models, each unit has a dummy input $a_0 = 1$ with an associated weight $w_{0,j}$. Each unit j first computes a weighted sum of its inputs

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i. \tag{2.1}$$

Then it applies an activation function g to this sum to derive the output

$$a_j = g\left(in_j\right) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right). \tag{2.2}$$

A feedforward neural network, aka multi-layer perceptron (MLP), is a series of logistic regression models stacked on top of each other, with the final layer also being an activation function that is decided depending on the type of neural network problem, such as `sigmoid` function for binary classification problems, `softmax` function for multi-class classification problems, tanh function and rectified linear unit function (`ReLu`) etc.

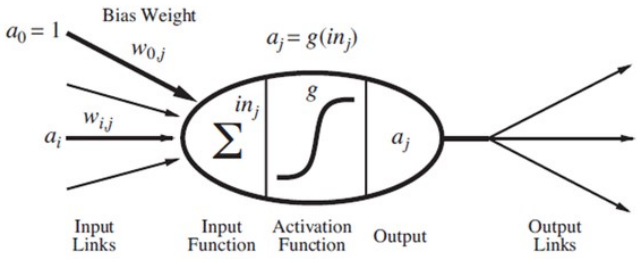


FIGURE 2.1: A simple mathematical model for a neuron

The activation function g is typically either a hard threshold (see Figure 2.2a), in which case the unit is called a perceptron, or a logistic function (see Figure 2.2b), in which case the term sigmoid perceptron is sometimes used. It is important that g is nonlinear, otherwise the whole model collapses into a large linear regression model of the form $y = w^T(Vx)$. One can show that an MLP is a universal approximator, meaning that it can model any suitably smooth function, given enough hidden units, to any desired level of accuracy [10]. Furthermore, the purpose of the hidden units is to learn nonlinear combinations of the original inputs; this is called

feature extraction or feature construction. These hidden features are then passed as input to the final generalized linear model (GLM) which is a term that refers to conventional linear regression models for a continuous response variable given continuous and/or categorical predictors [17].

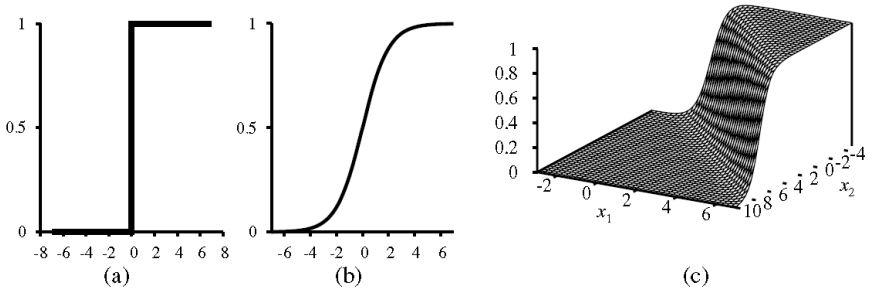


FIGURE 2.2: (a) The hard threshold function $Threshold(z)$ with 0/1 output. Note that the function is non-differentiable at $z = 0$. (b) The logistic function, $Logistic(z) = \frac{1}{1+e^{-z}}$, also known as the sigmoid function. (c) Plot of a logistic regression hypothesis $h_w(x) = Logistic(wx)$ [17].

2.1.1 Recurrent Neural Networks(RNNs)

A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A recurrent network, on the other hand, feeds its outputs back into its own inputs (see Figure 2.3). This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behaviour. Moreover, the response of the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory.

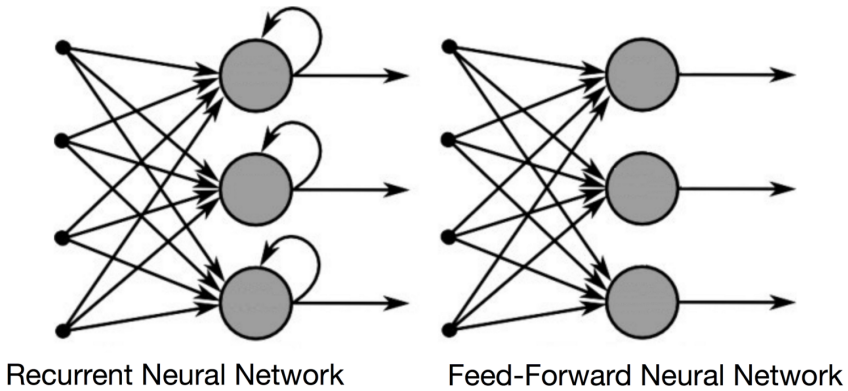


FIGURE 2.3: Feed forward neural network vs recurrent neural network

The recurrent neural network of Figure 2.4 and Equation 2.3 is universal in the sense that any function computable by a Turing machine can be computed by such a recurrent network of a finite size. The output can be read from the RNN after a number of time steps that is asymptotically linear in the number of time steps used by the Turing machine and asymptotically linear in the length of the input ([20], [18], [19], [11]).

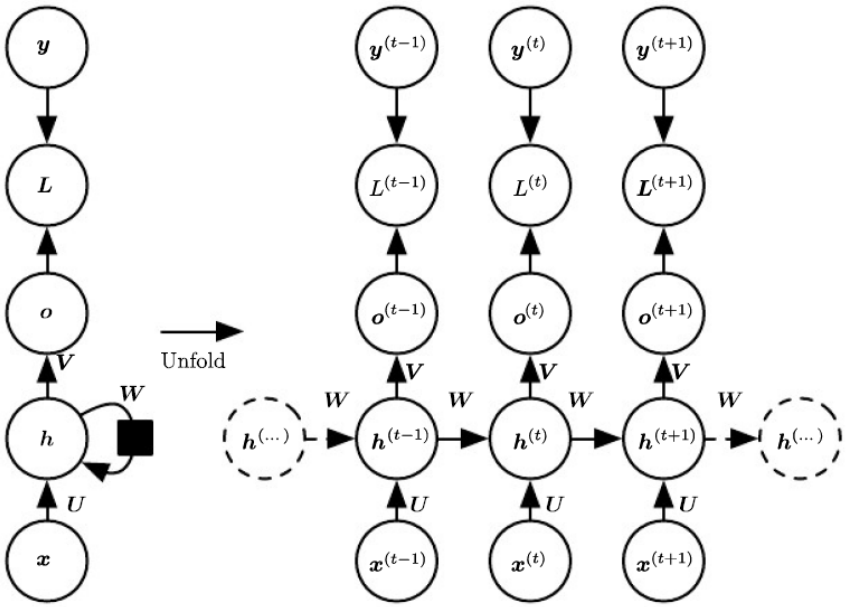


FIGURE 2.4: The computational graph to compute the training loss of a recurrent network that maps an input sequence of **x** values to a corresponding sequence of output **o** values. A loss L measures how far each **o** is from the corresponding training target **y**. When using softmax outputs, we assume **o** is the unnormalized logarithmic probabilities. The loss L internally computes $y = \texttt{softmax}(\mathbf{o})$ and compares this to the target **y**. The RNN has input to hidden connections parametrized by a weight matrix **U**, hidden-to-hidden recurrent connections parametrized by a weight matrix **W**, and hidden-to-output connections parametrized by a weight matrix **V**. Equation 2.3 defines forward propagation in this model. (Left) The RNN and its loss drawn with recurrent connections. (Right) The same seen as a time-unfolded computational graph, where each node is now associated with one particular time instance [7].

A natural way to represent discrete variables is to regard the output **o** as giving the unnormalized logarithmic probabilities of each possible value of the discrete variable. We can then apply the softmax operation as a post-processing step to obtain a vector **y** of normalized probabilities over the output. Forward propagation begins with a specification of the initial state $\mathbf{h}^{(0)}$. Then, for each time step from $t = 1$ to $t = \tau$, we apply the following update equations

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)} \tag{2.3}$$

$$h^{(t)} = \tanh(a^{(t)}) \tag{2.4}$$

$$o^{(t)} = c + Vh^{(t)} \tag{2.5}$$

$$\hat{y}^{(t)} = \texttt{softmax}(o^{(t)}), \tag{2.6}$$

where the parameters are the bias vectors **b** and **c** along with the weight matrices **U**,**V** and **W**, respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections. This is an example of a recurrent network that maps an input sequence to an output sequence of the same length. The total loss for a given sequence of **x** values paired with a sequence of **y** values would then be just the sum of the losses over all the time steps. For example, if $L^{(t)}$ is the

negative log-likelihood of $y^{(t)}$ given $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}$ then

$$L(\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(\tau)}\}, \{\mathbf{y}^{(1)}, ..., \mathbf{y}^{(\tau)}\}) = \sum_t L^{(t)} \tag{2.7}$$

$$= -\sum_t \log p_{\texttt{model}}(y|\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}\}), \tag{2.8}$$

where $p_{\texttt{model}}(y|\{\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}\})$ is given by reading the entry for $y^{(t)}$ from the model's output vector $\hat{y}^{(t)}$.

Long Short Term Memory(LSTM) Networks

The idea of introducing self-loops to produce paths where the gradient can flow for long durations is a core contribution of the initial long short-term memory (LSTM) model [9]. A crucial addition has been to make the weight on this self-loop conditioned on the context, rather than fixed [6]. By making the weight of this self-loop gated (controlled by another hidden unit), the time scale of integration can be changed dynamically. In this case, even for an LSTM with fixed parameters, the time scale of integration can change based on the input sequence, because the time constants are output by the model itself.

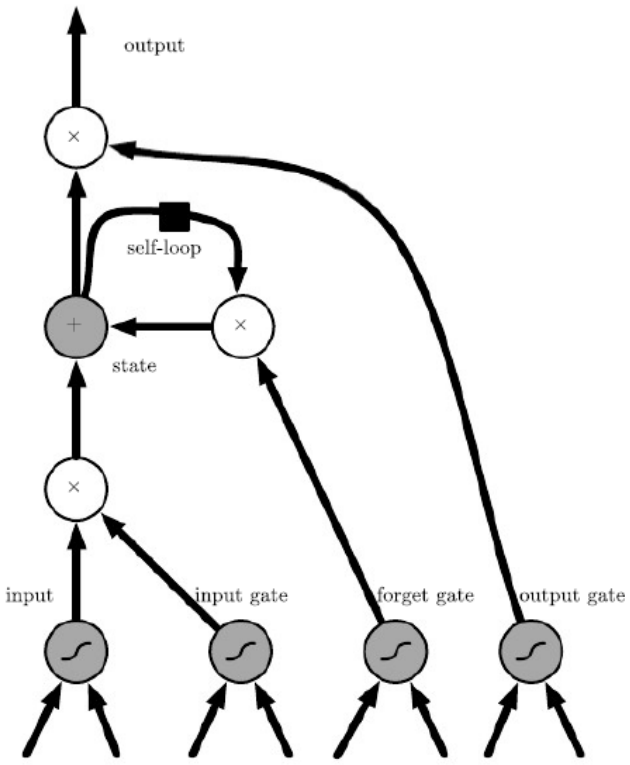


FIGURE 2.5: Block diagram of the LSTM recurrent network "cell". Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks. An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate. The output of the cell can be shut off by the output gate. All the gating units have a `sigmoid` nonlinearity, while the input unit can have any squashing nonlinearity. The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step [7].

LSTM block diagram is illustrated in Figure 2.5. Instead of a unit that simply applies an element-wise non-linearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition

to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but also has more parameters and a system of gating units that controls the flow of information. The most important component is the state unit $s_i^{(t)}$, which has a linear self-loop similar to the leaky units in some certain RNNs ([16] and [8]). Here, however, the self-loop weight (or the associated time constant) is controlled by a forget gate unit $f_i^{(t)}$ (for time step t and cell i), which sets this weight to a value between 0 and 1 via a `sigmoid` unit

$$f_i^{(t)} = \sigma(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}), \tag{2.9}$$

where $\mathbf{x}^{(t)}$ being current input vector and $\mathbf{h}^{(t)}$ being the current hidden layer vector, containing the outputs of all the LSTM cells, and $\mathbf{b}^f, \mathbf{U}^f, \mathbf{W}^f$ are respectively biases, input weights, and recurrent weights for the forget gates. The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}), \tag{2.10}$$

where **b, U** and **W** respectively denote the biases, input weights, and recurrent weights into the LSTM cell. The external input gate unit $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters

$$g_i^{(t)} = \sigma(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}). \tag{2.11}$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the output gate $q_i^{(t)}$, which also uses a `sigmoid` unit for gating

$$h_i^{(t)} = \tanh(s_i^{(t)}) q_i^{(t)} \tag{2.12}$$

$$q_i^{(t)} = \sigma(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}), \tag{2.13}$$

which has parameters $\mathbf{b}^o, \mathbf{U}^o$ and $\mathbf{W}^o$ for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in Figure 2.5.

Chapter 3

Experiment

3.1 Softwares and Modules

3.1.1 PX4

PX4 is an open source flight control software for drones and other unmanned vehicles. PX4 provides a flexible set of tools for drone developers to share technologies to create tailored solutions for drone applications. It also provides a standard to deliver drone hardware support and software stack, allowing an ecosystem to build and maintain hardware and software in a scalable way.
PX4 is part of Dronecode, a non-profit organization administered by Linux Foundation to foster the use of open source software on flying vehicles. Dronecode also hosts QGroundControl, MAVLink & the MAVSDK [15].

3.1.2 Software in the Loop

PX4 offers a feature where a SITL (Software in the Loop) simulation is run to simulate a flight on simulation. Figure 3.1 illustrates a simple example of a PX4 in SITL mode.
Software in the Loop is a simulation of a system which is modelled and virtually run under software without any hardware. To develop a control software, usually the system will have to pass four simulation verification phases before the software is implemented into a real system. The first is sometimes called MITL (Model-In-The-Loop) which only includes a mathematical model of the controller. The next is to verify the controller in a SITL (Software-In-The-Loop). Software in the Loop simulations verify the actual software integration of the controller to see if there is any unforeseen problems within the system. The third process is the PITL (Processor-In-The-Loop) which verifies there are no problems within the processor calculations. The last procedure is to have a HITL (Hardware-In-The-Loop) which includes the major hardware components in the control system to verify the controller is working properly.
PX4 supports both Software In the Loop (SITL) simulation, where the flight stack runs on computer, and Hardware In the Loop (HITL) simulation using a simulation firmware on a real flight controller board.

SITL Communication

All simulators communicate with PX4 using the Simulator MAVLink application programming interface (API). This API defines a set of MAVLink messages that supply sensor data from the simulated world to PX4 and return motor and actuator values from the flight code that will be applied to the simulated vehicle. Figure 3.2 shows the message flow between the flight stack and the simulator. The messages sent during the communication are described in Table 3.1.

Gazebo and QGroundControl

In order to create a SITL of the model drone that runs on PX4, an appropriate 3D simulator had to be chosen. For this purpose, Gazebo Simulator was chosen because of its high level of

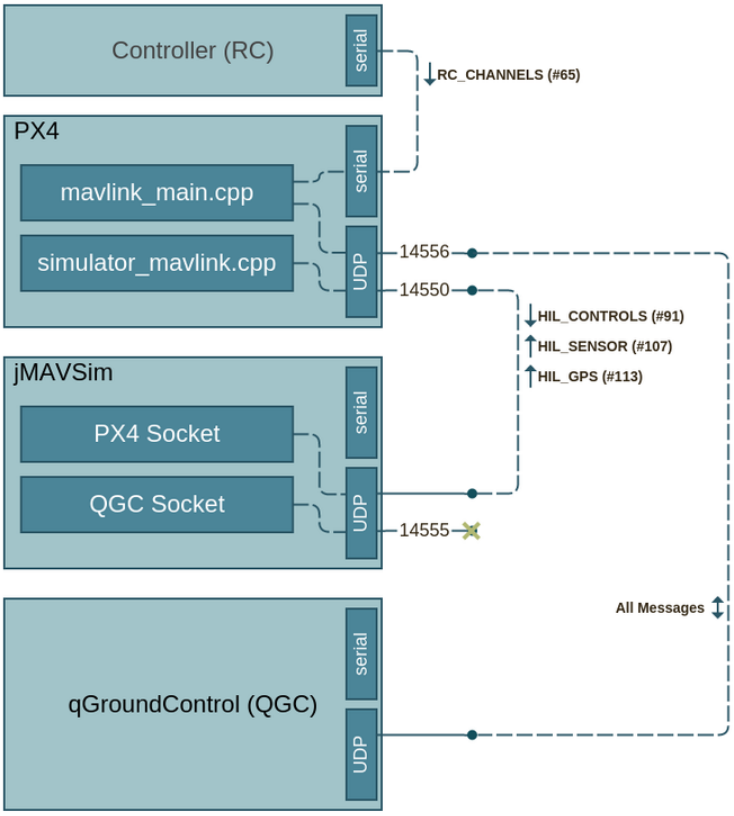


FIGURE 3.1: PX4 run on SITL mode [13].

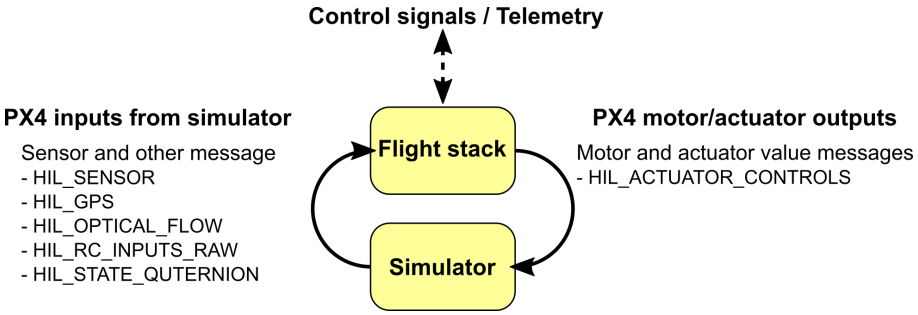


FIGURE 3.2: Simulator Communication [13].

compatibility with the further aspects of the project and also its strong practicality. Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs (see Figure 3.3).
QGroundControl provides full flight control and vehicle setup for PX4. It establishes the real-world connection between simulator and the air vehicle and reads the sensor data of the vehicle during the flight [14].

3.1.3 Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK or Theano [5]. In this project, Tensorflow, which is basically an end-to-end open source platform for machine learning [1], were chosen as the backend for the Keras.
One of the features of Keras is its ability to run on multiple graphics processing units (GPUs).

TABLE 3.1: Messages sent during the communication between PX4 and Simulator [13].

Message	Direction	Description
MAV_MODE:MAV_MODE_FLAG_HIL_ENABLED	NA	Mode flag when using simulation. All motors/actuators are blocked, but internal software is fully operational.
HIL_ACTUATOR_CONTROLS	PX4 to Sim	PX4 control outputs (to motors, actuators).
HIL_SENSOR	Sim to PX4	Simulated IMU readings in SI units in NED body frame.
HIL_GPS	Sim to PX4	The simulated GPS RAW sensor value.
HIL_OPTICAL_FLOW	Sim to PX4	Simulated optical flow from a flow sensor (e.g. PX4FLOW or optical mouse sensor)
HIL_STATE_QUATERNION	Sim to PX4	Contains the actual "simulated" vehicle position, attitude, speed etc. This can be logged and compared to PX4's estimates for analysis and debugging (for example, checking how well an estimator works for noisy (simulated) sensor inputs).
HIL_RC_INPUTS_RAW	Sim to PX4	The RAW values of the RC channels received.

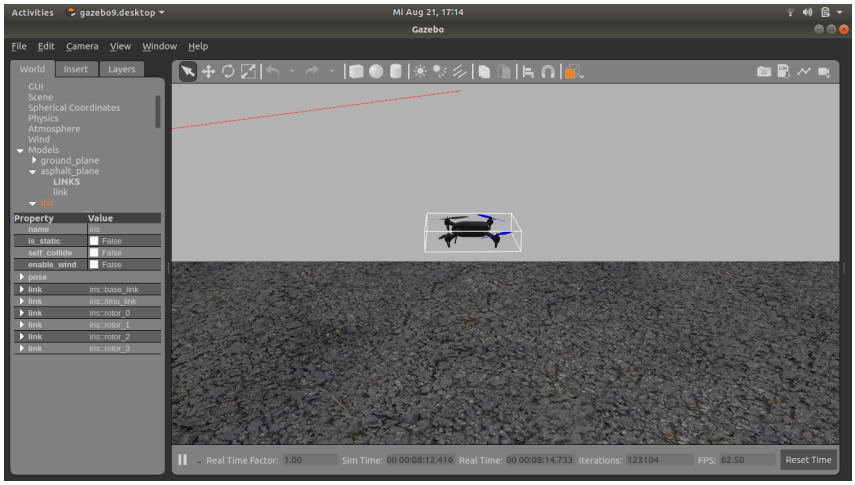


FIGURE 3.3: Gazebo's interface

There are two ways to run a single model on multiple GPUs [5]; data parallelism and device parallelism.

Data Parallelism Data parallelism consists in replicating the target model once on each device, and using each replica to process a different fraction of the input data.

Device Parallelism Device parallelism consists in running different parts of a same model on different devices.

The neural network architecture propesed in this project has taken advantage of the data parallelism.

3.2 Experimental Set-up

3.2.1 SITL

As mentioned in the Section 3.1.2, there are four simulation verification phases that needs to be passed before the software is implemented into a real system. In this project, only the first two phases was completed. Next two phases was left to be pursued for another further purpose. MITL has already been implemented in the simulation environment by the PX4. Therefore only SITL was performed additionally during the project.
The communication between PX4 and the simulator follows a well-defined pattern as described in the Section 3.1.2. The monitor only has an access to the information exchanged between the internal states of the PX4 and the simulator. This information is stored as a stack of communication vectors. The communication vector is a combination of time information, which contains 'timestamp' information of the sensor data, and the sensor data information. During the simulation, the information is stored as a ULog file. ULog is essentially a file format used for logging system data. Gazebo and QGroundControl use this file format to log device inputs (sensors, etc.), internal states (cpu load, attitude, etc.) and certain log messages, which together forms the aforementioned sensor data information.

3.2.2 Data Processing

The ULog file gathered from the simulation consists of three sections; header, definitions and data. The communication vector are stored in the data section. In order to store the communication vectors in a file which is more convenient to operate with, first the Ulog files were parsed and then stored as SQL-based databases.
In the next step of the data processing, the data was inspected in order to comprehend the structure and the content of the data. The inspection showed that each dataset consist of approximately 620 to 700 sensor parameters, which are referred as columns of the datasets in the rest of the text. In addition, the scale and distribution of the data drawn from the dataset are highly different for each parameter and therefore the dataset needs to be scaled before training to avoid any instability in the learning process [12]. The datasets also contain significantly high number of NaN (not-a-number) values which are caused by two circumstances. The first one is that the timestamps and time intervals used in the measurement of the sensor parameters vary by parameters. Therefore, when all of the sensor measurements are gathered together in a big SQL-based database, certain columns are lacking the information corresponding to certain timestamps. These gaps automatically are filled with NaN values during parsing. The other circumstance is that some sensors are used in particular situations. In other words, while there are certain number of sensors on the drone that are always activated regardless of the path and mission which the drone follows, there are also some specific sensors that are only activated during particular flight missions or by a real-time interaction between the drone and its surroundings. If data drawn from the simulation doesn't belongs to these particular missions, the data values corresponding to the particular sensors are again filled with NaN values during parsing. Table 3.2 shows the columns with the highest and the lowest number of NaN values that were extracted from a dataset. In addition to the table, an illustration of the distribution of missing data among the columns of the same dataset is presented in Figure 3.4.
Though the percentage of the missing values varies from dataset to dataset, every dataset used in the experiments contains missing data values. Since the absence of data reduces statistical power, which refers to the probability that the test will reject the null hypothesis when it is false, the missing data in the datasets requires to be handled before being sent to train. My approach is based on imputation using zero. Before the imputation, the columns, where more than 98.5 % of the data were NaN values, were dropped and the NaN values in the remaining columns were replaced with zero. To scale the dataset, normalization was also performed upon the dataset.
Datasets are fed into the neural network in batches to achieve a relatively speedy convergence.

TABLE 3.2: Total number and the percentage of NaN values in the columns with the highest and the lowest number of NaN values extracted from a dataset

Column Name	Total	Percentage
thrust[0]_x	58886	1.000000
thrust[1]_x	58886	1.000000
heading	58886	1.000000
heading_offset	58886	1.000000
thrust[2]_x	58886	1.000000
yawspeed_x	58886	1.000000
output[12]	58885	0.999983
output[15]	58885	0.999983
output[14]	58885	0.999983
output[13]	58885	0.999983
mag_device_id	58885	0.999983
output[10]	58885	0.999983
...	...	...
accelerometer_timestamp_relative	10094	0.171416
accelerometer_integral_dt	10094	0.171416
accelerometer_m_s2[2]	10094	0.171416
accelerometer_m_s2[1]	10094	0.171416
accelerometer_m_s2[0]	10094	0.171416
gyro_rad[1]	10094	0.171416
gyro_rad[2]	10094	0.171416
gyro_integral_dt	10094	0.171416
airspeed_timestamp_rel	5131	0.087134
distance_sensor_timestamp_rel	5131	0.087134
visual_odometry_timestamp_rel	5131	0.087134
vehicle_magnetometer_timestamp_rel	5131	0.087134
vehicle_air_data_timestamp_rel	5131	0.087134
optical_flow_timestamp_rel	5131	0.087134
gps_timestamp_rel	5131	0.087134
timestamp	0	0.000000

The batches are comprised of vector sequences. The number of the vector sequences in a batch is governed by the batch size whereas the number of vectors in a sequence is governed by the design parameter N. For example, if X_k is the k-th vector of the dataset, Seq_k and $Batch_m$ respectively represent the k-th vector sequence and the m-th batch sent into the network, then

$$Seq_k = \{X_k, ..., X_{k+N-1}\} \tag{3.1}$$

$$Batch_m = \{Seq_m, ..., Seq_{m+n-1}\}, \tag{3.2}$$

where N and n respectively denote the input sequence length and the batch size.

3.3 Neural Network Design

Main objective of the model to estimate the system correctness in terms of a probability to obtain a communication vector $\vec{u}^t$ given the $(N-1)$ previous discrete steps that have been already detected. The LSTM based model predicts the possible communication vector $\hat{\vec{u}}^t$ given the previous $(N-1)$ communication steps and then approximates the overall accuracy $P(\vec{u}^t|\vec{u}^{t-N}, ..., \vec{u}^{t-1})$ from accuracy values for the time information and the sensor data information. The ANN architecture proposed in this project is highly inspired by the monitor based

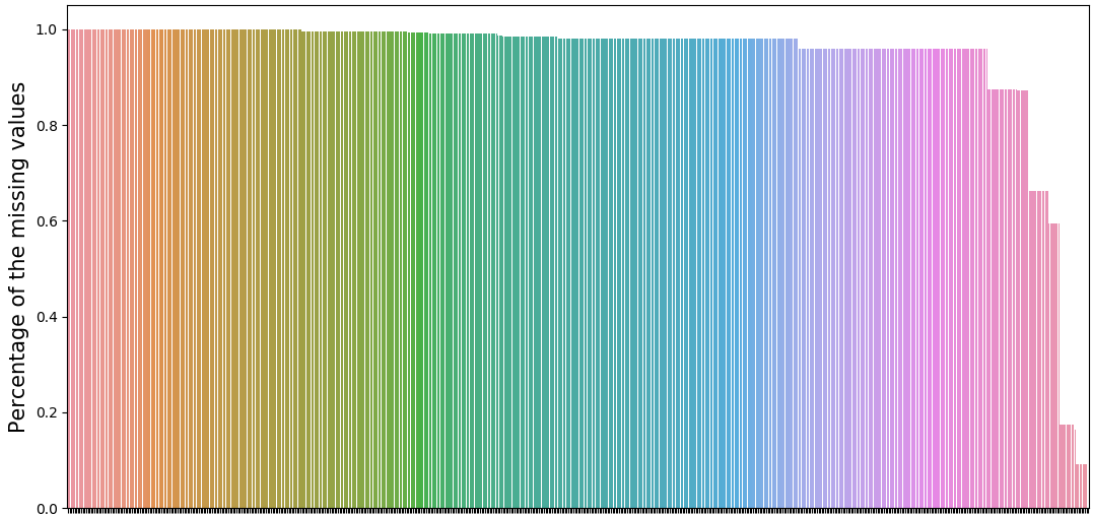


FIGURE 3.4: Illustration of missing data percentage of each column of the dataset where x-axis represents the columns of the datasets sorted by the percentage of missing data values in the columns while y-axis represents the percentage for the missing data

on LSTM neural networks which was proposed in [2].

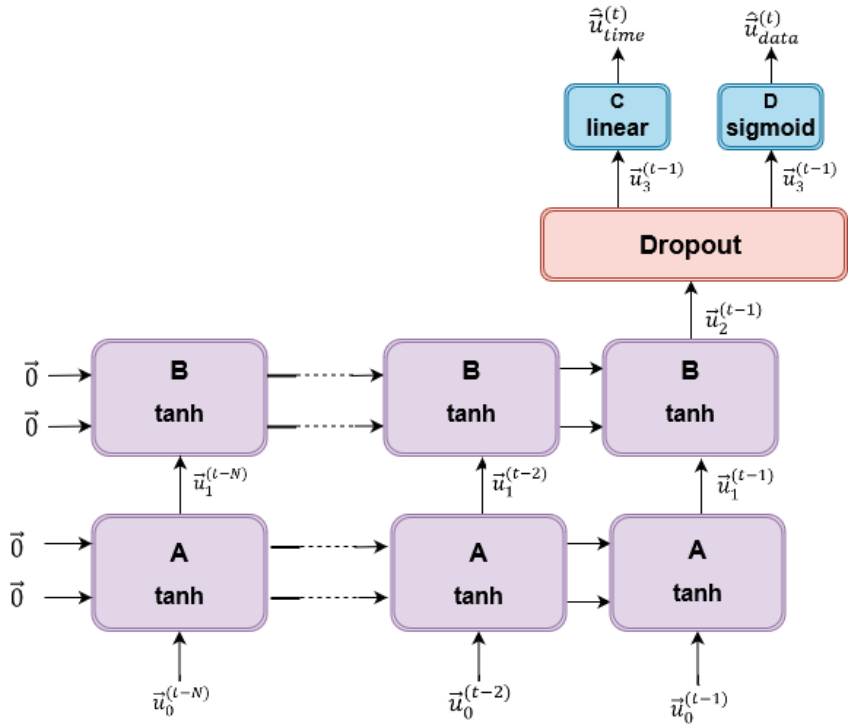


FIGURE 3.5: Flow chart of the LSTM based model which consists of its unfolded LSTM Layers (A and B) with tanh actşvatşon function , a droput layer and two feed-forward layers (C and D with activation functions of `linear` and `sigmoid`, respectively). Sequential data from the databses are fed into the model and model predicts $\hat{\vec{u}}^{t}_{time}$ and $\hat{\vec{u}}^{t}_{data}$ using layers C and D. Zero vectors $\vec{0}$ are used to initialize the hidden and cell states of the LSTM layers.

In Figure 3.5, an illustration of the neural network architecture is presented. Input sequence $\vec{u}_0$ is fed into the two stacked LSTM layers (A and B). The number of hidden states in the LSTM layers L and the input sequence length N are design parameters. A single LSTM cell of the layer is presented in Figure 3.6. The output of the second LSTM layer $\vec{u}_2^{t-1}$ is fed into a dropout

layer, which is added to the model to avoid over-fitting during training. Then, the dropout layer sends the input unchanged to two independent dense layers. The dimensions of these layers (C and D) are designated by the dimensions of the communication features.

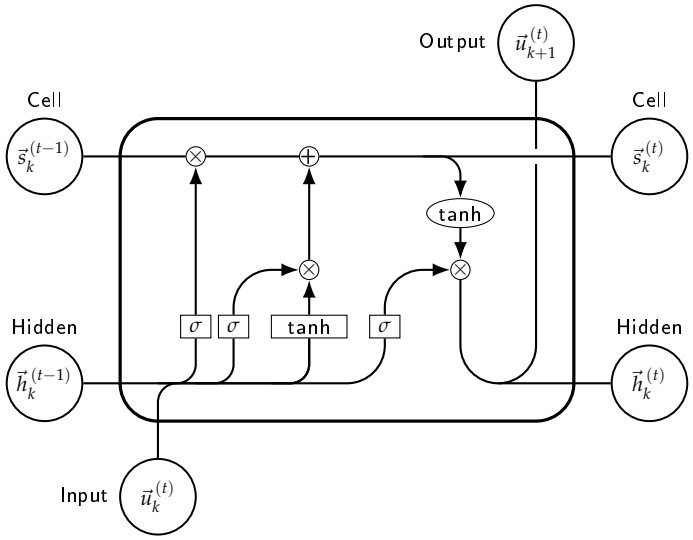
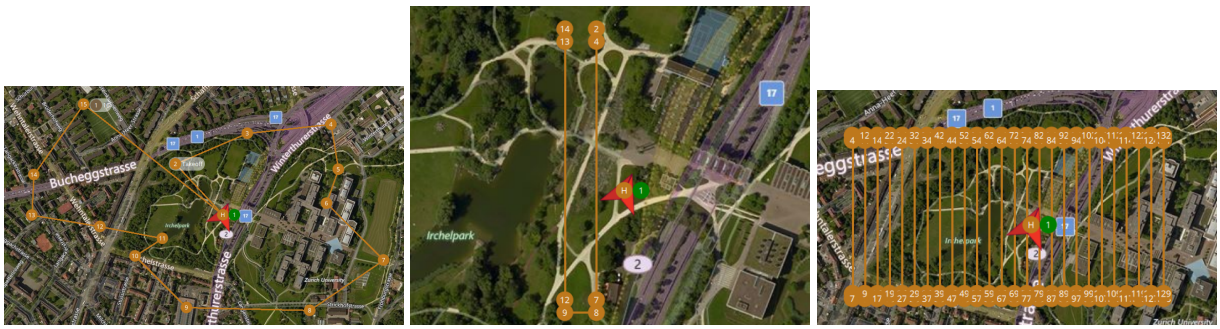


FIGURE 3.6: Single LSTM cell in the LSTM layer of the proposed model where k and t, respectively denote current layer and time step. Though $\vec{u}_{k+1}^{(t)}$ and $\vec{h}_k^{(t)}$ are identical vectors, they are written with a different notation to remain consistent with the LSTM model presented above. Internal mathematical operations with gates and activation functions (tanh and `sigmoid`) are explained comprehensively in Chapter 2.

Accuracy of the model's multiple outputs with respect to the current actual communication vector $\vec{u}^t$ is calculated by a different approach in both values. Time vector $\hat{\vec{u}}_{time}^t$ is continuous and therefore forms a regression problem. Layer C is trained with *Mean Squared Error* (MSE) loss. Accuracy $acc_{time} = P(\vec{u}_{time}^t | \vec{u}^{t-N}, ..., \vec{u}^{t-1})$ is assessed using the validation set. On the other hand, data vector $\vec{u}_{data}^t$ forms a multi-label multi-class classification problem and therefore is trained using a binary cross-entropy loss. Accuracy of the data $acc_{data} = P(\vec{u}_{data}^t | \vec{u}^{t-N}, ..., \vec{u}^{t-1})$ is assessed from $\hat{\vec{u}}_{data}^t$ using similarity function. The overall model loss, which is used in the back-propagation, is calculated by the mean of two individual losses of the output while the mean of both accuracy values is utilized in calculating the overall model accuracy. Furthermore, an *adadelta* optimizer [21] is utilized to train the multiple output model.

3.4 Experiment

Each flight mission has its unique set-up and the number of sensor in use differs in mission. Therefore, several flight patterns are followed by the drone in order to evaluate different scenarios for the experiment. Figure 3.7 illustrates these pattern. In the experiment, the dataset from one of the path is trained whereas remaining two datasets are used for validation data. Thus, three individual models fed by three different dataset are trained and validation sessions for 6 different scenario are performed during the experiment.
As mentioned in the Section 3.3, input sequence length N, the number of hidden neurons in both of the LSTM layers L are design parameters for the model. In addition to these design parameters, the fraction of muted neurons in the dropout layer D is added as a design parameter. An early stopping mechanism is also implemented in the network to avoid over-fitting [3]. The models are trained with the combinations of N as 15, 25, 50, 150 and also L as 20, 40, 50, 100, 200, 300 whereas values of 0.1, 0.25 and 0.5 for D is applied to each model. After the analysis of each dropout rate applied in the models, training provides more accurate results using the



(A) Path with Waypoints (B) Corridor Scan (C) Survey

FIGURE 3.7: 3 unique missions defined for the experiment where 'H' and the numbers respectively denote home and the direction of the flight

dropout rate of 0.1. Observed results of training the models with the dropout with $D = 0.1$ is presented in the next section.

TABLE 3.3: Observed overall validation accuracy results after testing the models with several validation scenarios.

		L=300 N=150	L=200 N=150	L=100 N=50	L=50 N=25	L=40 N=15	L=20 N=15
Trained Dataset	Waypoints	89 %	90 %	90.4 %	89 %	90 %	90 %
Validation Dataset	Survey						
Trained Dataset	Waypoints	89 %	89 %	90.2 %	90.2 %	90 %	91 %
Validation Dataset	Corridor Scan						
Trained Dataset	Survey	88.8 %	87.6 %	89 %	90.2 %	90.6 %	90.8 %
Validation Dataset	Waypoints						
Trained Dataset	Survey	88.9 %	89 %	90 %	90.2 %	90.4 %	90.3 %
Validation Dataset	Corridor Scan						
Trained Dataset	Corridor Scan	90.4 %	89.6 %	90.5 %	90.8 %	91.3 %	91.2 %
Validation Dataset	Waypoints						
Trained Dataset	Corridor Scan	90.4 %	90.8 %	90.5 %	90.8 %	90.1 %	90.5 %
Validation Dataset	Survey						

3.4.1 Experimental Results

In each of the experiments, Keras with Tensorflow backend was used. Training of the models are generally concluded after approximately 4 epochs of training and duration of an epoch in the training ranges between 2 and 10 minutes depending upon the design parameters since the models with a large input sequences takes more time to be completed.
Table 3.3 presents the overall validation accuracy results of the trained models with the dropout rate of 0.1. Each of three scenarios were tested with different validation datasets and overall validation accuracy of 90 % was obtained. Table 3.3 also shows that the models achieve higher accuracy levels where the number of hidden neurons and the input sequence length are rather small.

Chapter 4

Conclusion

The attained high validation accuracy levels indicates that the proposed ANN architecture offers a great deal of reliability for passively monitoring an auto-piloted air vehicle running in SITL. Results also shows that better performance and high overall accuracy levels are obtained when short sequences are fed into the LSTM model. The proposed architecture acts as a credible initial base model to be reused in a rather comprehensive future scenario.
In a future project, fault injection will firstly be performed on the architecture to examine its response. After that, the architecture will be implemented as an on-board mechanism in the air vehicle to address the certain on-board problems which auto-piloted air vehicles will most likely face while flying in real time. It will be actively monitoring the real-time system that is assumed to be suffering from malfunctioning which occurs in a manner, that is unpredictable and unforeseeable by an external observer, during a real-time flight.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from https://tensorflow.org. 2015. URL: http://tensorflow.org/.

[2] Fin Hendrik Bahnsen and Görschwin Fey. *Artificial Neuronal Networks for Local Monitoring of Embedded Applications and Devices*. 2018.

[3] Jason Brownlee. "Overfitting and underfitting with machine learning algorithms". In: *Machine Learning Mastery* (2016).

[4] Renee Cho. *How drones are advancing scientific research*. Ed. by phys.org. [Online; posted 19-June-2017]. 2017. URL: https://phys.org/news/2017-06-drones-advancing-scientific.html.

[5] François Chollet et al. *Keras*. https://keras.io. 2015.

[6] F. A. Gers, J. Schmidhuber, and F. Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12.10 (2000), pp. 2451–2471.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[8] S. El Hihi and Y. Bengio. "Hierarchical recurrent neural networks for long-term dependencies." In: *Advances in Neural Information Processing Systems* 8 (1995), pp. 493–499.

[9] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9 (1997), pp. 1735–1780.

[10] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2 (1991), pp. 251–257.

[11] Heikki Hyötyniemi. "Turing Machines Are Recurrent Neural Networks". In: 1996.

[12] Yann LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. London, UK, UK: Springer-Verlag, 1998, pp. 9–50. ISBN: 3-540-65311-2. URL: http://dl.acm.org/citation.cfm?id=645754.668382.

[13] the Dronecode Project, Inc., a Linux Foundation Collaborative Project. *PX4 Development Guide*. 2018. URL: https://dev.px4.io/master/en/.

[14] the Dronecode Project, Inc., a Linux Foundation Collaborative Project. *QGroundControl User Guide*. 2018. URL: https://docs.qgroundcontrol.com/en/.

[15] the Dronecode Project, Inc., a Linux Foundation Collaborative Project. *What Is PX4?* 2018. URL: https://px4.io/.

[16] M. C. Mozer. "The Induction of Multiscale Temporal Structure." In: *Advances in Neural Information Processing Systems, 4*. Ed. by J. E. Moody, S. J. Hanson, and R. P. Lippmann. San Mateo, CA: Morgan Kaufmann, 1992, pp. 275–282.

[17] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020, 9780262018029.

[18] Hava T. Siegelmann. "Computation beyond the turing limit." In: *Science* 268 5210 (1995), pp. 545–8.

[19] Hava T. Siegelmann and Eduardo D. Sontag. "On the Computational Power of Neural Nets". In: *J. Comput. Syst. Sci.* 50 (1995), pp. 132–150.

[20] Hava T. Siegelmann and Eduardo D. Sontag. "Turing computability with neural nets". In: 1991.

[21] Matthew D. Zeiler. "ADADELTA: An Adaptive Learning Rate Method". In: *CoRR* abs/1212.5701 (2012). URL: http://dblp.uni-trier.de/db/journals/corr/corr1212.html#abs-1212-5701.