The University of Manchester

Autonomous Flight Navigation With Deep Reinforcement Leaning

Obstacles Avoidance System for Fixed-Wing UAV

Phachara Laohrenu

ID: 10214424

BEng (Hons) Mechanical Engineering

School of Mechanical, Aerospace and Civil Engineering

Supervisor:

Prof. William Crowther

Date Submitted:

27th April 2020

ABSTRACT

Reinforcement Learning (RL) has been becoming more popular and successful in various robotic tasks, including UAVs. The contemporary autopilot system has a flaw of being inflexible for a sudden change in the environment, such as encountering obstacles. In this project, Deep Q-Learning (DQL) – one of the RL techniques – is used to train a simulated fixed-wing UAV to avoid obstacles. The performance of the UAV is compared between when using two different types of input; 2D LiDAR and a video camera with depth estimation algorithm. The environment for training the agent is simulated on a gaming engine, Unity3D, while the RL algorithm is written on Python. The results have shown that the RL controller with 2D LiDAR has a significantly better performance over the RL controller with depth estimation, and also outperforms the benchmark controller. It was found the time delay caused by slow computation time of depth estimation algorithm is the main factor that significantly deteriorates the ability of the UAV in avoiding obstacles. However, this time delay helps the agent to spend a lot fewer training steps when compared to the agent with LiDAR, who has no time delay. This is because of the time delay act as a frame skipping technique, which is known to help accelerate the learning rate of RL agents.

All of the codes, Unity file, and video of the result are available at my GitHub repository: https://github.com/phachara-laohrenu/DQN-ObstacleAvoidance-FixedWingUAV

TABLE OF CONTENTS

Final word count

Whole report : 20,087 words
Actual Content (excluding references, codes, etc.) : 12,492 words

TABLE OF FIGURES

TABLE OF TABLES

1. INTRODUCTION

1.1 Research Motivation

Traditionally UAV can be controlled autonomously by autopilot system to navigate itself from one point to another along a safely predefined path planned by a human or a path-planning algorithm. Unfortunately, both the human and the said algorithm heavily rely on the prior knowledge of the area, such as a 3D map, which is not always entirely accurate and could miss some small details of the map that could be the potential threats to the UAV. Therefore, there is a need for a reactive obstacle avoidance system that allows the UAV to fly through an unexplored terrain or to steer away from any unexpected incoming object in real-time.

There are various existing obstacle avoidance algorithms. However, this project is focusing on the use of Reinforcement Learning, which has proven to achieve successful results, as seen in [14], [15], [16], and [17]. Reinforcement Learning is a type of AI algorithm that has seen increasing popularity among navigation tasks due to its adaptability to the uncertainties. There are several pieces of research that implement the RL algorithm to obstacle avoidance for UAVs, such as [15], [16], and [17], which were done on the quadcopter model. However, there's hardly any research done on a fixed-wing model which has a completely different flight dynamic from a quadcopter. Therefore, this research is using a fixed-wing model as an aircraft model. The difference in flight dynamics of a fixed-wing model introduces some challenges for the experimental design and learning algorithm, which will be discussed in section 1.3.

1.2 Research Goal

In summary, this project makes a comparison between the performance of the agent in obstacle avoidance when using LiDAR and when using a depth map from the depth estimation algorithm as the input.

For the UAV to respond to an incoming object, it needs some input. There are two common types of devices that are currently being used to gather information for this task; 1. LiDAR sensor 2. monocular camera. A LiDAR sensor uses laser rays to measure the distance from the

laser source to the object that it hits and directly tell the agent about the position of the incoming object in the form of a sparse depth map. LiDAR sensors can also have a very wide field of view, with some of them can measure 360°. However, 3D LiDARs are very heavy, with some of them can weight up to almost 3kg[3]. Therefore, for an application with very limited payload, 2D LiDARs are more commonly used. Although this experiment limits the movement of the UAV to only a 2D horizontal plane, the UAV is still allowed to roll left and right. Thus, having only 2D input could compromise the ability of the UAV to avoid obstacles at a high roll angle (see figure 1). LiDAR sensors also have other flaws of being expensive, and short in range.

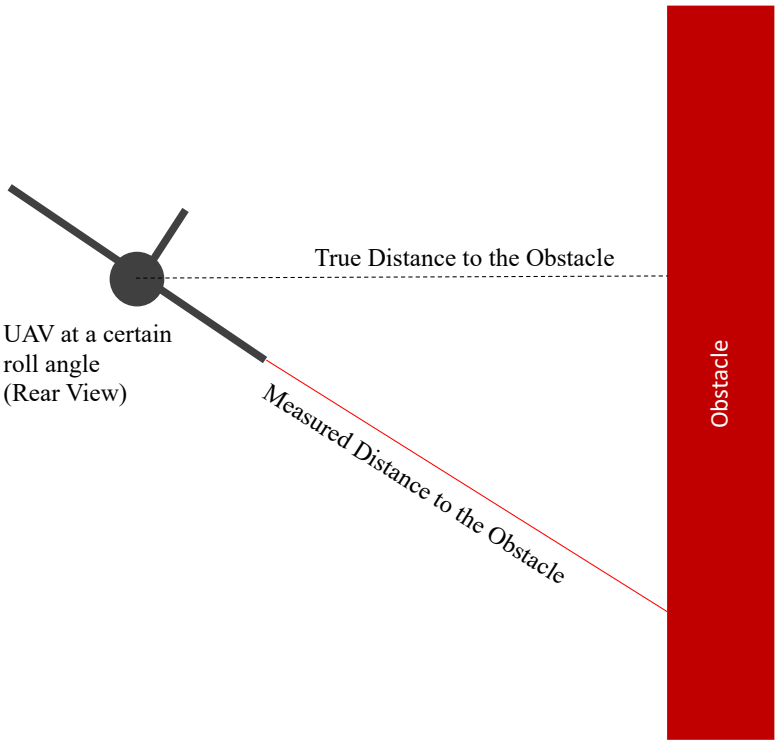

Figure 1: Rear view of a UAV flying pass an obstacle at a non-zero roll angle

The monocular camera, on the other hand, is very cheap and light. This makes it perfect for aerial vehicles as the payload is one of the most significant limitations. However, the image from a monocular camera cannot be efficiently utilised by the agent to tell the obstacle's position. There are various ways to use the RGB images from the camera to detect obstacles, but the most effective and is being studied most extensively is to use depth estimation algorithm to turn a single RGB image into a depth image[4]. Depth estimation uses deep neural network with

supervised or unsupervised learning to estimate the depth in each pixel[4]. However, the depth estimation algorithm still suffers from slow computation time[5], which is very important to the obstacle avoidance task of a fixed-wing UAV since fixed-wing UAV needs to fly at a specific minimum speed to maintain its lift. Having a slow computation time could cause the UAV to crash very often due to not being able to avoid the obstacle in time. The monocular camera also has a limited field of view, usually only around 60 degrees horizontally[6]. This limited field of view could affect the ability of the agent to find the optimal path to avoid the obstacles around it.

Therefore, this project compares the performance of the agent between when using LiDAR and when using a depth map from the depth estimation algorithm as the input. The project also tries to address the limitations above of LiDAR and depth estimation algorithm in the simulation for a realistic comparison.

The performance of the UAV with the RL algorithm when using these two inputs is also assessed against another existing obstacle avoidance, Braitenberg controller, as a benchmark for the performance. This is done by taking the best-performed neural network models from each of the two inputs (let's call them RL controllers) and test them against the Braitenberg controller. The detail about the Braitenberg controller will be discussed later in section 4.3.

1.3 Difference Between Quadcopter and Fixed-Wing Models

Since this project will be focused on a fixed-wing UAV, there are some challenges risen from dissimilarity of the manoeuvre between these two models that are needed to be addressed:

1. Unlike the quadcopter model, the fixed-wing model cannot stop and hover at a certain position. This means that it is forced to take action quickly once it detects the obstacles. Therefore, the algorithm cannot be too complex as it would slow down the decision-making process of the UAV.
2. A fixed-wing model has a minimum turning radius. Being unable to suddenly turn or reverse limits the ability of the UAV to avoid obstacles in certain scenarios. It is now crucial to consider if the environment setup is possible or not too harsh for the UAV to complete the task. For example, ensuring that the density of the generated obstacles on the map and detectable range is sufficient for the given UAV's turning radius.

3. A fixed-wing UAV has a non-zero roll angle during turning. While practically, a quadcopter also rolls as it turns, the roll angle in a typical quadcopter (excluding the racing type), is relatively small so that some experiment, such as [17], simulates their quadcopter in such a way that it always stays level to the ground. This means the experiments with quadcopter models can be easily simplified into a 2D problem where 2D LiDAR is fully sufficient. Since the roll angle of a fixed-wing UAV is not neglectable, 2D LiDAR may not be sufficient in some situation while depth image that contains 3D information may be more suitable.

2. RELATED WORK

There is a lot of pieces research done on using reinforcement learning for obstacle avoidance task of UAVs, such as [15], [16], [17], [18], and [19]. All of these works, except [18] simplified the movement of the agent into horizontal plane, while [18] formulate the problem in the 3D world. Unfortunately, all of these researches use quadcopter as their UAV model, and there's hardly any other research that uses fixed-wing UAV. As mentioned previously that there's some significant difference between the quadcopter and fixed-wing aircraft, this section will also review the work done on cars since the movement of fixed-wing UAV is very similar to that of a car when the problem is simplified into a horizontal plane. Since there are two different types of inputs being investigated, this section will review the related works based on their input type.

2.1 LiDAR Input

LiDAR has been widely used for obstacle avoidance system due to its accuracy in detecting obstacles. [14] implements LiDAR for their simulated cars. Their result shows very high performance with a very fast learning rate. However, the performance of the agent during both training and testing phase is still inconsistent. This could be caused by their learning algorithm and experimental setup rather than the input. [18] compares the performance of their quadcopter when using depth image and when using raw RGB image as input. Since their depth image is simulated directly from their software, then it is comparable to using LiDAR. However, their RGB images are directly fed into the RL algorithm. This is why their result shows a clear

superiority in performance when using depth image. Nevertheless, their experiment using DQN shows a good and stable learning curve only in one of the three maps. This might be due to their other two map require the UAV to also move in a vertical direction to avoid the obstacles, and thus the problem becomes much more complicated.

2.2 Depth Image Input

[15], [19], and [20] use depth estimation algorithm to process the raw RGB images into depth images by using deep neural network. This method, is by far, the most studied one nowadays. The results from all of these three works shown that depth-estimated images can be used to guide the agent away from obstacles very effectively. However, the unmanned vehicles in these works move relatively slow, much slower than any fixed-wing UAV could fly. Therefore, the slow processing time of depth estimation algorithm does not affect the training and performance of their vehicles too much.

3. BACKGROUND KNOWLEDGE

3.1 Reinforcement Learning Algorithm

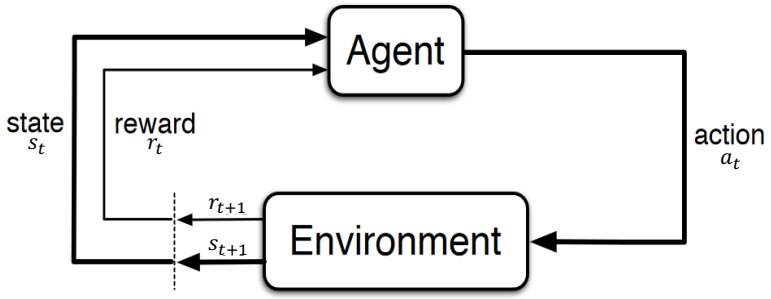

Figure 2: Reinforcement Learning Architecture

The concept of Reinforcement Learning can be described as an agent-environment interaction as illustrated in Figure 2, where Agent is the UAV. At each time step t, the agent observes the environment and takes state $s_t \in \mathbb{S}$ where $\mathbb{S}$ is a finite state list. The agent then takes the an $a_t \in \mathbb{A}$, where $\mathbb{A}$ is a finite set of actions, based on policy π. The action a_t transits the agent into the next state s_{t+1} where the reward r_{t+1} is given to the agent based on the reward function[1]. The reward function plays an important role in the agent's behaviour. The reward function for this project will be discussed later in section 4.7.3. The final purpose of the agent is to maximize the total discounted reward R_t given by equation (1).

$$R_t = \sum_{k=0}^{T} \gamma^k \cdot r_{t+k+1} \tag{1}$$

Where T is the terminal step, γ is discounted factor that determines how the importance of the future reward.

There're various Reinforcement Learning techniques being used nowadays with their own unique advantages or disadvantages. The technique used for this project is a state-of-the-art technique, Deep Q-Learning (DQL). DQL is the technique that integrates Deep Learning into Q-Learning to better account for continuous state environment.

3.1.1 Q-Learning

The main idea of Q-Learning is to map each state-action pair $[s_t, a_t]$ to the expected cumulative discounted reward, known as Q-value [1]. This results in what is called, Q-table with dimension of n x m where n is the number of actions and m is the number of states. The value corresponding to each cell is the Q-value. The Q-value for any state-action pair under policy π is obtained from Q-function as shown in equation (2).

$$Q^{\pi}(s_t, a_t) = E_{\pi}[R_t \mid s_t, a_t]$$
$$Q^{\pi}(s_t, a_t) = E_{\pi}\left[\sum_{k=0}^{T} \gamma^k \cdot r_{t+k+1} \,|s_t, a_t\right] \quad (2)$$

Q-function tells how good it is for the agent to take the action a given a state s while following policy π. Policy is the function that maps a given state to probabilities of selecting each possible action. The goal of DQL (as well as other RL algorithms) is to find the policy that yields the most cumulative reward R_t for the agent as it follows that policy, known as "optimal policy". The optimal policy has an associated optimal Q-function, denoted as $Q^*(s_t, a_t)$ and must satisfy the Bellman equation:

$$Q^*(s, a) = E\left[r_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})\right] \quad (3)$$

Where $\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$ term means the maximum return that can be achieved from any possible next state-action pair.

This Bellman equation is used by Q-learning algorithm to perform value iteration to iteratively updates the Q-values for each (s, a) until the Q-function converges to the optimal Q-function, Q^* [7]. As the agent encounters state s_t and chooses to take action a_t, the agent receives the reward r_{t+1} and progresses into next state s_t. This allows the agent to update the Q-value for (s_t, a_t) on the Q-table by equation (4)

$$Q^{\pi}_{new}(s_t, a_t) = Q^{\pi}(s_t, a_t) + \alpha[Q^*(s, a) - Q^{\pi}(s_t, a_t)] \quad (4)$$

Where $Q^{\pi}(s_t, a_t)$ is the old Q-value of that state-action pair. The term $[Q^*(s, a) - Q^{\pi}(s_t, a_t)]$ is called 'temporal loss', which is the difference between the old Q-value and the maximum Q-value that can be achieved from any possible next state-action pair[7]. This loss term reduces overtime as the agent learn as the Q-function converges to the optimal Q-function, $Q^{\pi}(s, a) \rightarrow Q^*(s, a)$. α is learning rate of the agent, and has the value between 0 and 1.

In order to encourage learning of the agent, the agent must explore different action as it encounters a state, rather than always choosing the action with the highest Q-value. To do this, the epsilon-greedy strategy is introduced[7]. As the agent encounter state s_t the agent can either choose action randomly (exploration), or choose the action with maximum Q-value $Q^*(s_t, a_t)$, which is called exploitation. The probability that the agent would explore or exploit is determined by epsilon ϵ, which has the value between 0 and 1, and reduces gradually as time step increases so that the agent would choose to exploit more over time . The agent chooses the action based on the ϵ-greedy strategy by following the condition:

$$action = \begin{cases} explore & : \quad x < epsilon \\ exploit & : \quad x \geq epsilon \end{cases} \quad (5)$$

where x is a uniformly random number between 0 and 1.

The epsilon will decay by the decay function governed by the parameters ϵ_{decay}. The decay function can be any function that decreases the value of epsilon over time. The decay function used in this experiment will be discussed later in section 4.7.2. One must ensure that epsilon does not decay too quickly otherwise the agent would not have a much chance to explore new strategies and tend to get stuck at non-optimal policy. However, decaying the epsilon too slow will produce a very slow learning rate.

3.1.2 Deep Q-Learning

As the size of state increases, such as in the case of continuous state, Q-table becomes insufficient to account for all of the possible state-action pairs, and thus need to be replaced by neural network. The neural network acts as a regression approximator to estimate the Q-value for each state-action pair. The objective is to approximate the optimal Q-function $Q^*(s, a)$. This network is called Deep Q-Network (DQN). The neural network is also used to predict the Q-value of each action at a given state to perform the action with the highest predicted Q-value[8].

In order for the DQN to become a more accurate approximator and to make better predictions overtime, the DQN must go through training which is done by the technique called "experience replay".

3.1.3 Experience Replay

At each time step, the agent gains an experience e_t defined as a tuple $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. All of the agent's experiences during the training is stored in the replay memory. This replay memory has the capacity to store N_{buffer} number of experiences. When the replay memory is full, the oldest experience will get discarded as the new experience is stored. At each experience replay, an N_{sample} number of experience is randomly sampled from the replay memory to train DQN. For each experience e_t, the state s_t is forward passed into the neural network with weights θ to approximate the Q-value of that state-action pair, $Q^\theta(s_t, a_t)$. Similarly to the Q-Learning, the optimal Q-value $Q^*(s, a)$ needs to be estimated through the Bellman equation. This is done by the second forward pass of s_{t+1} to the neural network to approximate $\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$. Then the temporal loss $Q^*(s, a) - Q^\theta(s_t, a_t)$ can be calculated.

Gradient descent of DQN is then performed to adjust the weights of the network with the goal to minimise the loss. Depending on each experiment, experience replay can occur at every time step, every n steps, or even every episode. Nevertheless, the bottom line is that it brings the Q-function $Q^\theta(s, a)$ closer to the optimal Q-function $Q^*(s, a)$ over time[8].

4. METHODOLOGY

4.1 Simulation of UAV Model in Unity

The environment and the UAV itself are simulated in a gaming engine called Unity3D. It is a common practice for a single UAV to be equipped with multiple controller algorithm, responsible for a different level of control. This project assumes that there's already an algorithm to take care of the low-level control, such as aircraft stability, and the RL algorithm is supposed to only control aircraft's manoeuvre, such as turning left/right or going straight. Therefore, the flight's aerodynamic is completely ignored in this project, and the UAV can transform its position and rotation precisely by the command from the RL algorithm. The UAV model is shown in figure 3 Since the aerodynamic is ignored, the shape of the aircraft is irrelevant except the wingspan. The wingspan should not be too long since the longer the wingspan is, the harder for the UAV to avoid the obstacle due to the wing hitting the obstacle. In this project, the UAV model is set to have both the body length and wingspan of 1 metre.

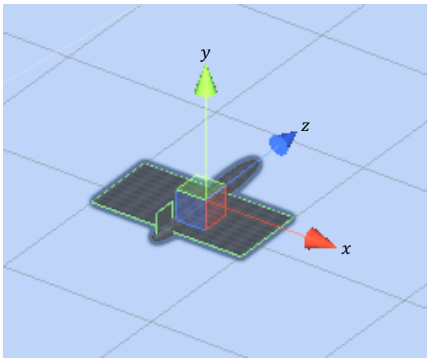

Figure 3: The UAV model

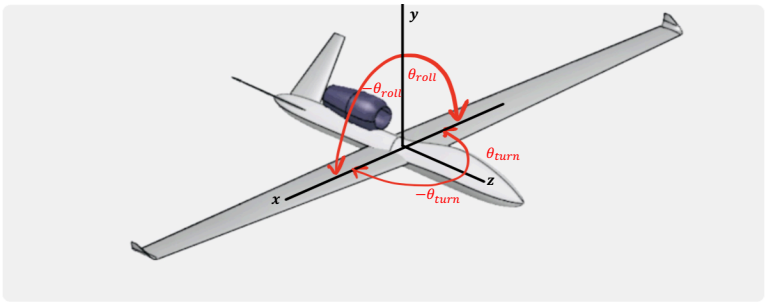

Figure 4: Diagram of θ_{rol} and θ_{turn}

4.1.1 Dynamics of the UAV

For the simplicity in formulating the RL problem, the UAV's manoeuvre is limited to turning left, right, and going straight. This means the UAV only flies on a horizontal xz-plane at a constant altitude. To imitate the actual flight dynamic, as the UAV turns, it also rolls left/right at a non-constant roll angle, $\theta_{roll} = [-\theta_{roll,max}\ ,\ \theta_{roll,max}]$. Since the altitude is constant, the height y is irrelevant and can be ignored. Therefore, the location of the UAV is described by $(x, z, \theta_{roll}, \theta_{turn})$, where (x, z) denotes Cartesian position and $(\theta_{roll}, \theta_{turn})$ denotes orientation. As illustrated in figure 4, θ_{roll} is caused by the rotation around the axial axis of the UAV's body,

while θ_{turn} is caused by the rotation about the y axis that follows the Cartesian position (x, z) of the UAV. Here, it is supposed that the derivative of roll angle $\dot{\theta}_{roll}$ is being controlled by the controller. That is the RL controller applies a certain amount of change of roll angle $\dot{\theta}_{roll}$ to turn the UAV. Then, $\theta_{roll}^{t} = \theta_{roll}^{t-1} + \dot{\theta}_{roll}$. The current roll angle which is θ_{roll}^{t} is used to calculate angular speed of turning $\dot{\theta}_{turn}$ through equation (5).

$$\dot{\theta}_{turn} = \frac{\theta_{roll}^{t}}{\theta_{roll,max}} \times \dot{\theta}_{turn,max} \tag{5}$$

Where $\theta_{roll,max}$ is the maximum possible roll angle, which is a constant. $\dot{\theta}_{turn,max}$ is a pre-defined constant representing the maximum possible angular speed of turning. Then, the current turn angle is calculated by $\theta_{turn}^{t} = \theta_{turn}^{t-1} + \dot{\theta}_{turn}$. The UAV is also supposed to have a constant speed v. Therefore, the dynamics of the UAV are given by:

$$\dot{x} = v \cdot \sin(\theta_{turn}^{t})$$
$$\dot{z} = v \cdot \cos(\theta_{turn}^{t})$$
$$\dot{\theta}_{turn} = \frac{\theta_{roll}^{t}}{\theta_{roll,max}} \times C_{turn}$$
$$\dot{\theta}_{roll} = u$$

Where u has the value between $[-u_{max}, u_{max}]$.

4.1.2 Control of the UAV

Since the only dynamic variable that is being controlled by the controller algorithm is $\dot{\theta}_{roll}$, then the controller algorithm takes the action $a \in \{0, u_{max}, -u_{max}\} = \mathbb{A}$ as an input, and output the current roll angle θ_{roll}^{t} that is calculated by $\dot{\theta}_{roll}$ corresponding to the action. Since θ_{roll} has a range between $[-\theta_{roll,max}, \theta_{roll,max}]$, the controller algorithm needs to account for this constraint. Also, the UAV needs to orient itself back to its normal orientation when the action changes from turning to going straight. Therefore, θ_{roll} needs to move towards 0 when $a = 0$. The Pseudo code for this algorithm is shown on the following page.

Algorithm 1 GetMovement(action)

Input: $action \in \{straight, left, right\}$

Output: u from $\mathbb{A} = \{0\ , u_{max}\ , -u_{max}\}$

if $action = stright$ **then**

    if $\theta_{roll} > 0$ **then**

        $u = -\min(u_{max}\ ,\ \theta_{roll})$

    else if $\theta_{roll} < 0$ **then**

        $u = \min(u_{max}\ ,\ |\theta_{roll}|)$

    else if $\theta_{roll} == 0$ **then**

        $u = 0$

    end if

else if $action = left$ **then**

    $u = \min\left(u_{max}\ ,\ \theta_{roll,max} - \theta_{roll}\right)$

else if $action = right$ **then**

    $u = \max\left(-u_{max}\ ,\ -\theta_{roll,max} - \theta_{roll}\right)$

end if

The actual code written in C# is available in the appendix B, line 245.

4.2 Simulation of the Map

The map is in a squared enclosed area with walls on all four sides representing the boundary of the domain area. The domain's boundary is $[-x_{max}\ , x_{max}]$ and $[\ 0,\ z_{max}]$. Within the domain contains obstacles of cylindrical shape with various radius $r_{obs,S}$, $r_{obs,M}$, and $r_{obs,L}$. The variation of obstacles' sizes is to generalise the training. At the beginning of each episode, the position of each obstacle is randomised for generalised training. At the start of the map, there is an area, free

of obstacles to leave some room for the UAV to start its flight at initial position. As shown in figure 5 the area that the obstacles can be generated is $[-x_{max}, x_{max}]$ and $[z_{min}, z_{max}]$. The number of each type of obstacles per a square unit area is set to a constant at $\rho_{obs,S}$, $\rho_{obs,M}$, and $\rho_{obs,L}$. C# code for obstacles generation is available in appendix C.

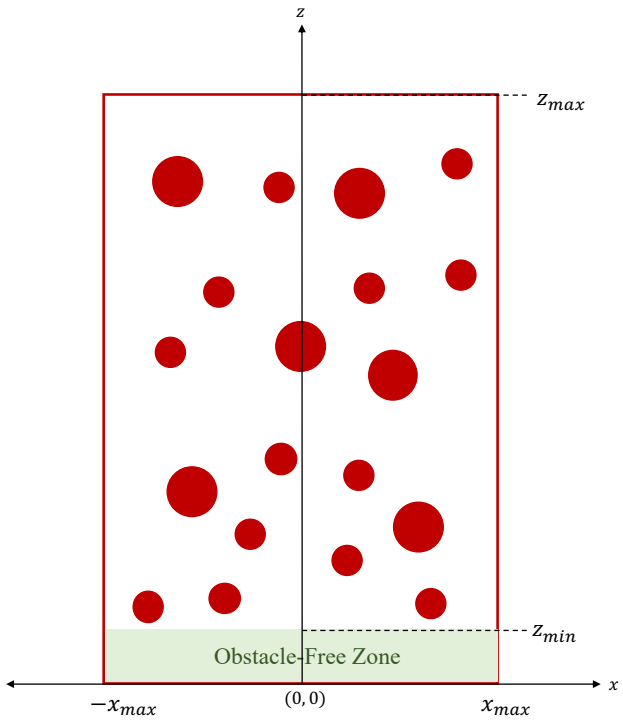

Figure 5: Layout of the map

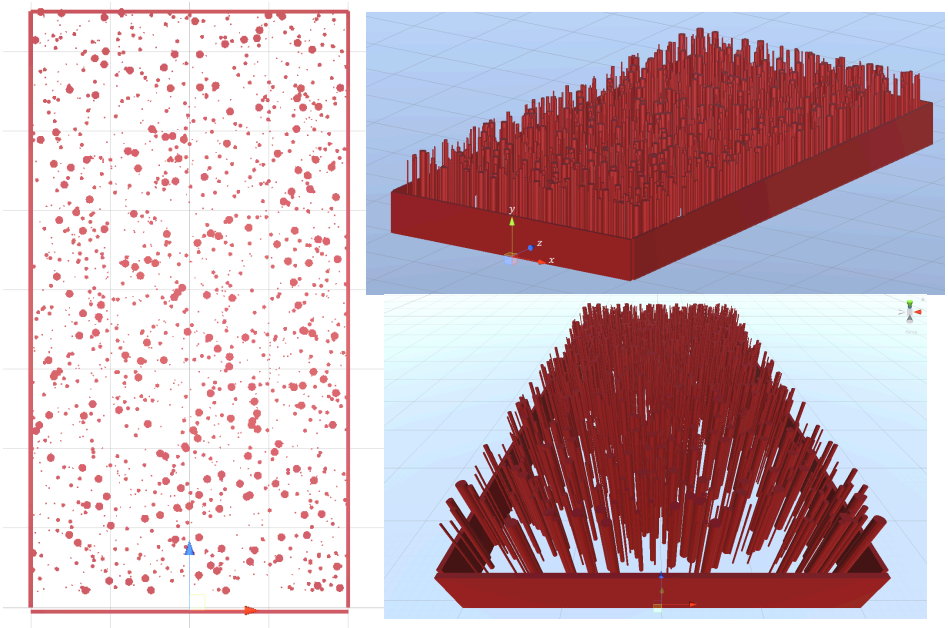

Figure 6: Various views of the map

4.3 Environment Parameters

The selection of the parameters for the environment simulation is very important. The main goal is to balance the difficulty of the task. If the UAV has low speed and the obstacles are too sparsely populated (low ρ_{obs}) then then agent could score a very high cumulative reward even with random action. If the UAV has slow turning speed and high forward speed, then it is not be possible to avoid obstacles in time regardless of how much it has been trained. After trials and errors, the summary of the environment parameters is shown in the table 1 below.

Parameters	Values
Dynamics of UAV	
- forward speed v	$10\ m/s$
- maximum angular turning speed $\dot{\theta}_{turn,max}$	$90°/s$
- roll angle $\theta_{roll} = [-\theta_{roll,max}, \theta_{roll,max}]$	$[-30°, 30°]$
- rate of roll $u = [-u_{max}, u_{max}]$	$[-5°, 5°]$
Map Simulation	
- Domain boundary $[-x_{max}, x_{max}]$ and $[0, z_{max}]$	$[-200m, 200m], [0, 770m]$
- Obstacle-free zone z_{min}	$20m$
- Obstacle radius $(r_{obs,S}, r_{obs,M}, r_{obs,L})$	$(1m, 2.5m, 5m)$
- Obstacle density $(\rho_{obs,S}, \rho_{obs,M}, \rho_{obs,L})$	$\left(\frac{8000}{3}, 2000, \frac{2000}{3}\right)\ unit/km^2$

Table 1: Environment Parameters

4.4 Simulation for LiDAR

LiDAR sensors are simulated by using `Raycast` in Unity, which consists of multiple rays, each can measure the distance from its origin to its hit point. Similar to the actual LiDAR, each ray has a finite detectable range of x_{max}. If the obstacle is not within this range, then the ray will have the reading of $x_i = x_{max}$. In the real world, a depth map from 2D LiDAR is created by having a laser ray to scan side to side horizontally from $-\theta°$ to $\theta°$ at a very high speed[9]. Therefore, theoretically, each column of the LiDAR readings will be the reading at different point in time. However, this project assumes that the scanning speed is very high that this lag in time is neglectable. For simplicity, rather than having the LiDAR ray scanning from side to side, all the rays are simulated all at once to create a 2D LiDAR (see figure 7) that outputs the 1D array of size n representing the distance readings. The angles between each ray are denoted by ϕ. All values in this array are to be normalised into the range of 0 to 1 before getting parsed into the RL algorithm as inputs. This is done by:

$$\vec{x}_{norm} = \frac{\vec{x}}{x_{max}}$$

Commercial LiDAR sensors come with various detachable range depending on the model. The range can be varied from 40m to 200m or more[3]. In this project, the maximum detectable range x_{max} of the simulated LiDAR is set to 100 meters. Since fixed-wing UAV can only fly in forward direction, the field of view of the LiDAR only needs to be $-90°$ to $90°$ In reality, LiDARs can have a very fine angular resolution, usually no more than 1°[3]. However, to reduce the computational workload, this project will set the angular resolution to be 3°. Therefore, this will result in the total number of rays of $n = \frac{90-(-90)}{3} + 1 = 61$ rays. The code to simulate LiDAR is available in appendix B, line 184.

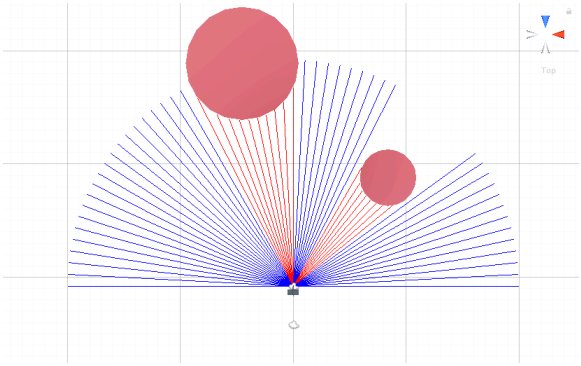

Figure 7: Simulation of LiDAR. Red rays mean the rays are hitting an object

4.5 Simulation for Depth Estimation Algorithm

Rather than implementing real depth estimation algorithm, which is beyond the scope of this project, the depth texture from Unity is used instead. Depth texture allows Unity to display a depth map of the camera view where each pixel corresponds to the distance from the camera to the object in that pixel (see figure 8). The field of view of the camera is set to 60° both vertically and horizontally. To send image from Unity to Python efficiently, the camera view at each frame is encoded into base64 string and sent to Python. Python then decode the base64 sting back to the original image. In order to address the flaw of the algorithm of being slow, each image decoding process in Python is set to run at low speed of 0.1 second. This is number is to reflect that usually the depth estimation algorithm can only run at around 10 frames per second or 0.1 second per frame[21]. This causes the whole training loop in Python to become slower while the simulation in Unity still runs at the same speed of 50fps. To reduce the computational workload, the image at each frame is reduced to the size of 10×10 pixels. The pixel values of the decoded image in Python is to be normalised to the range of [0,1] by dividing them with 255. The 10×10 matrix of normalised pixel values is then flattened to a 1D array of length 100 before being passed to the neural network as a state. The code for converting normal camera display to depth image is available at appendix H.

Figure 8: Depth images from camera when $\theta_{\text{roll}} = 0$ *(left), and when* $\theta_{\text{roll}} = -30$ *(right)*

4.6 Braitenberg Controller

As a baseline against which to compare the RL controller, a simple hand-designed controller from [14] is used as a benchmark. It was inspired by the controllers of Valentino Braitenberg[10]. Their version of controller works with 2D LiDAR. The idea is to divide their LiDAR rays into left side, n_{left}, and right side, n_{right}. Then compute the square inverse of each sensor measurement as its features: $\phi_B(x_i) = \frac{1}{x_i^2}$. The sum of these feature is computed for n_{left} and n_{right}.

$$n_{left} = \sum_{i}^{\frac{N}{2}} \frac{1}{x_i^2}$$

$$n_{right} = \sum_{i=N/2+1}^{N} \frac{1}{x_i^2}$$

Then, the action is selected to turn towards the direction $\min(n_{left}, n_{right})$. In the case where $n_{left} = n_{right} = 0$, the action is set to going straight. However, if $n_{left} = n_{right} \neq 0$, then the action is set to turning left.

This algorithm can be easily applied to the LiDAR in this project by ignoring the middle ray. However, the depth estimated image is a 3D information (2D matrix) and thus cannot simply apply above computation. To produce a 1D array from 2D matrix, the mean of each column is calculated. The result is then used by the above computation to select the action to perform. The Pseudo code for Braitenberg controller is shown below.

Algorithm 2 2-Dimentional Braitenberg Squared Inverse Controller

---

Input: Sensor data $x_{ij} \in \{X\}$, size $n \times m$

Output: Action $a \in \{straight, left, right\}$

Initialize $n_{left}, n_{right} = 0$

Initialize z is empty 1-dimentional array of length m

for $j = 1$ **to** m **do**

$z_j = (\sum_{i=1}^{n} x_{ij})/n$

end for

if length(z) is odd **then**

    delete $z_{(m-1)/2}$

Set $m' =$ length(z)

for $j = 1$ **to** $m'/2$ **do**

    $n_{left} = n_{left} + 1/(z_j)^2$

end for

for $j = 1 + m'/2$ **to** m' **do**

    $n_{right} = n_{right} + 1/(x_j)^2$

end for

if $n_{left} > n_{right}$ **then**

    $a = left$

else if $n_{left} < n_{right}$ **then**

    $a = right$

else if $n_{left} == n_{right}$ **and** $n_{left}, n_{right} \neq 0$ **then**

    $a = left$

else if $n_{left} == n_{right} == 0$ **then**

    $a = stright$

end if

The actual code written in Python is available in the appendix G, line 86.

4.7 Reinforcement Learning: Training Phase

Most of the projects in obstacle avoidance train their agent by letting the agent fly around a map with obstacles (similar to this one) aimlessly with the goal of to fly for as long as possible without crashing. One of the major problems of this set up is that the agent tends to fly around in a circle, or pinning to one particular obstacle and fly around it. One of the solutions is to discourage turning by setting a negative reward for each turn. This approach is used by many works, such as [14], [15], and [19]. However, the result from [14] suggests that agent still occasionally drive around in a circle. For this project, in particular, it was found that this approach leads to the UAV being very "stiff" and to crash very often. The reason might be due to the dynamic of my UAV. Recall equation (5) that the turning speed of the UAV is directly proportional to the current roll angle. Also, the UAV always tries to come back to $\theta_{roll} = 0$ when the action is to go straight. This means that if the agent chooses to turn left for only 1 time-step, the agent will only turn at almost a neglectable amount since θ_{roll}^{t} is very small. In order to experience the full effect of turning, the agent must choose to turn in the same direct for 6 consecutive time-steps. Therefore, at the beginning of the training when the agent choose action mostly randomly, the turning does not seem to help the agent to avoid the obstacle at all. This makes the agent to prefer going straight even more and will take the agent a very long time to learn to avoid the obstacle.

Another approach to overcome the agent circling around is to set the target point for the agent to fly to. Some examples are [16], [17], and [18]. This approach is shown to be working well to fix the circling around the problem. However, the overall performance of the agent depends on how complicated the reward function is since this approach introduces the second task, which is the target point. Solving two tasks at once could lead to a slow learning curve of the agent.

This project takes the second approach with a slight adjustment. Rather than have the target point as another goal, this project only uses the target point as guidance for the agent to keep flying north. This is to prevent the agent from circling around at one place without the need for the negative reward for turning. Reaching the goal or not is not essential and thus the target point is set to be very far away at $(0, z_{max})$. The only primary goal for the agent is to fly for as long as

possible without crashing. However, it is punished for flying away from the target point. The reward function will be explained in detail later.

Since the time of each time-step in LiDAR and depth estimation experiments are different (due to the time delay for depth estimation algorithm), the UAV in Unity travels for a much greater distance in the depth estimation experiment. This is because each simulation loop in Unity is set to run at a fixed rate. Then with the same number of time steps in python, depth estimation experiment takes much more time in real life to complete, and thus Unity undergoes a lot more update loop causing the UAV to fly further. When the agent flies further, it encounters more obstacles and has a higher chance of failing. Therefore, in order to make the experiments fair, the number of training steps per episode of each experiment must be set to so that they result in a relatively similar episodic actual time. In this project, the time it takes to complete a full episode (assuming no crash) is aimed to be around 30 seconds. This is equal to around 220 training steps for depth estimation experiment and 1000 steps for LiDAR experiment. Since the speed of the UAV is $10\ m/s$, then the furthest distance that the UAV can theoretical travel during each episode around $10 \times 30 = 300m$. Each experiment lasts for the total of 300,000 training steps. At the beginning of each episode the UAV resets its position back to the starting position $(0,0)$ and the obstacles' positions are randomised.

4.7.1 State

Apart from outputs from LiDAR and depth image, there are 3 more values needed for the agent to learn properly. First is the current roll angle θ_{roll}, since the degree of turning speed depends on roll angle. θ_{roll} is normalised into $[-1,1]$ by $\theta_{roll,norm} = \frac{\theta_{roll}}{\theta_{roll,max}}$. The second value is the change in distance to the target point, Δd_{goal}. The third value is the current relative angle between the UAV's flying direction and the vector from the UAV to the target point $\theta_{UAV/goal}$. The second and third values are to guide the agent into the right direction. Δd_{goal} is calculated by subtracting the distance from previous time step by the distance of current time step;

$$\Delta d_{goal} = d_{goal}^{t-1} - d_{goal}^{t}$$

Therefore, Δd_{goal} is positive if the agent gets closer to the target point and vice versa. Δd_{goal} is normalised into the range of $[-1,1]$ by $\Delta d_{goal,norm} = \Delta d_{goal} / \Delta d_{goal,max}$. Where $\Delta d_{goal,max}$ is the maximum change in distance. $\Delta d_{goal,max}$ occurs when the agent flies right straight to the target point, thus it can be calculated by $v \times \Delta t$ where v is the forward speed of the UAV and Δt is the time of each frame in Unity. Since Unity run at 50fps and $v = 10$, then

$$\Delta d_{goal,max} = 10 \times \frac{1}{50} = 0.2\ m$$

$\theta_{UAV/goal}$ is not a simple angle between 2 vectors. Rather, it also reflects the clockwise/anti-clockwise direction as illustrated in figure 9. Therefore, it cannot be calculate using simple dot product rule. The formula for $\theta_{UAV/goal}$ is shown as equation (6)

$$\theta_{UAV/goal} = \tan^{-1}\left(\frac{u_x \cdot v_z - u_z \cdot v_x}{u_x \cdot v_x + u_z \cdot v_z}\right) \quad (6)$$

Where $\vec{\boldsymbol{v}}$ is velocity vector of the UAV

$\vec{\boldsymbol{u}}$ is vector from UAV to target point

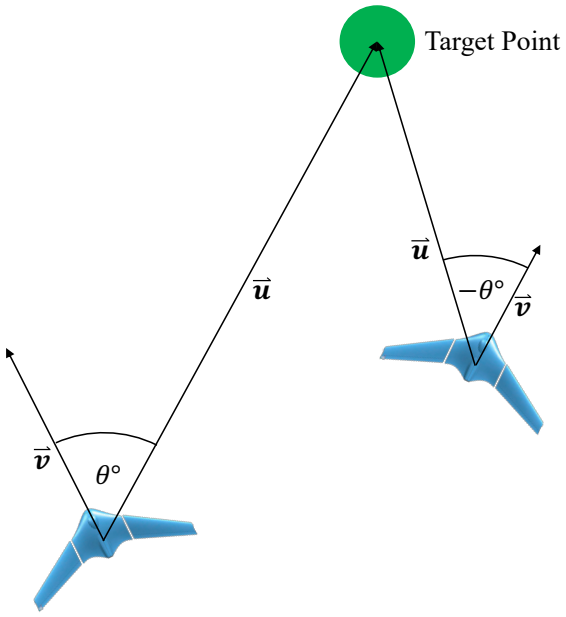

Figure 9: Diagram shows the measurement of $\theta_{UAV/goal}$ *with clockwise and anti-clockwise direction*

However, $\tan^{-1}$ can only output the angle between $\left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$. The function `Mathf.Atan2()` in C# is required to extend the angle range to $[-\pi, \pi]$. $\theta_{UAV/goal}$ is then normalised to be in range $[-1,1]$ by $\theta_{UAV/goal,norm} = \frac{\theta_{UAV/goal}}{\pi}$.

Therefore, at each time step during each episode, the agent observes the state $s = \left(\theta_{roll,norm}, \Delta d_{goal,norm}, \theta_{UAV/goal,norm}, X\right)$, where X is a 1D array output from LiDAR or depth image as discussed earlier. It is worth noting that this set of state is not a complete list of information available ("true state"). In fact, the "true state" also includes the position of the UAV and the obstacles, and the collider state $C_{col} \in \{0, 1\}$, where 0 means no collision is detected and 1 means collision is detected. This collider is a part of Unity feature. However, to make the problem similar to the limitations of a real UAV, the agent is only allowed to access observable state s.

The numbers of LiDAR rays, and pixels in depth image are 61 and 100, respectively. Then the total elements in s for LiDAR and depth image case are $61 + 3 = 64$ and $100 + 3 = 103$ elements, respectively.

4.7.2 Action

Once the agent observes the current state s_t, it will take action $a_t \in \{0, u, -u\} = \mathbb{A}$, which corresponds to straight, left, and right actions. The agent chooses the action based on the ϵ-greedy strategy as mentioned previously. The epsilon at the first episode e^0 is initialised to be 1 and is gradually reduced to the minimum value of ϵ_{min}. The epsilon decay function is governed by equation (7):

$$\epsilon^e = \max\left(\epsilon_{min}, \epsilon_0\left(1 - \epsilon_{decay}\right)^{(e-1)}\right) \tag{7}$$

Where ϵ^e is epsilon value at episode e

ϵ_0 is the initial value of epsilon

ϵ_{decay} is the epsilon decay value

ϵ_{min} is the minimum epsilon value that will anneal to

The purpose of the minimum value of epsilon ϵ_{min} is to prevent the agent from always exploiting in the long run. Since the agent learns new policy by exploring, it is always an ideal to let the agent explore occasionally during training.

If the agent chooses to explore, the action is selected randomly from the action space. If the agent chooses to exploit, the neural network predicts the Q-value of each action based on the current state and the action with highest Q-value is selected.

4.7.3 Reward

In order to encourage the agent to fly towards the target point while avoid the obstacle, the reward would ideally be set in proportion to Δd_{goal} and be heavily negative when the agent crashes. However, having a variety of possible reward at each time step has been proven to cause unstable learning due to the loss function does not converge[11]. Therefore, this project uses "reward clipping" technique to improve learning stability. Reward clipping is when all the positive reward is set to 1 while all the negative reward is set to -1. While the agent is still flying, it receives +1 reward for every time step that it gets closer to the target point and -1 when it gets further away. If the agent crashes, regardless if it is moving closer or away from the target point, it receives -1 reward. Therefore, the reward function can be defined by equation (8):

$$r = \begin{cases} +1 & \text{if } \Delta d_{goal} > 0 \text{ and } C_{col} = 0 \\ -1 & \text{if } \Delta d_{goal} < 0 \text{ and } C_{col} = 0 \\ -1 & \text{if } C_{col} = 1 \end{cases} \quad (8)$$

4.7.4 Training of Deep Q-Network

As previously mentioned that the training of DQN uses the technique called experience replay. In this project, experience replay is set to perform at every time step. It was mentioned that, to compute loss, there are two forward passes. One for $Q^{\theta}(s_t, a_t)$ using the current state as input, another is for $\max_{a_{t+1}} Q^{*}(s_{t+1}, a_{t+1})$ using next state as input. The goal is the minimise the loss $Q^{*}(s_{t+1}, a_{t+1}) - Q^{\theta}(s_t, a_t) = \left(r_{t+1} + \gamma \max_{a_{t+1}} Q^{*}(s_{t+1}, a_{t+1})\right) - Q^{\theta}(s_t, a_t)$. However, if the neural network that are used to approximate $Q^{\theta}(s_t, a_t)$ and $\max_{a_{t+1}} Q^{*}(s_{t+1}, a_{t+1})$ are the same one,

then the optimisation appears to be chasing non-stationary target, since $\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$ is computed with the network whose weights are constantly changing. This can make the optimisation unstable. Therefore, this project separates the network used for $Q^{\theta}(s_t, a_t)$ and $\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$. The network used to compute $Q^{\theta}(s_t, a_t)$ is called "policy network", and the one for $\max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})$ is called "target network". Both have the same architecture (e.g. number of nodes and layers), and initial weights. However, unlike the policy network, the target network will not update its weights every replay. It will only update its weight to be the same as that of policy network once every n_{tg} replays[12].

During the optimisation process, the learning rate α must be specified. At the start of the training, it is ideal to have a relatively high learning rate to accelerate training. To prevent overshooting in the long run, the learning rate should be kept low towards the end of the training. Therefore, it can be concluded that the learning rate should not be kept at a fixed value throughout the training, but rather should be reducing over time. There are various ways to decay the learning rate[13]. Here, the step decay is used as it shows a significant performance against the other type of decay[13]. The step decay is done by scheduling the drop of the learning rate by a factor of α_{decay} at every n_{α} number of replay. This means the learning rate initially starts at $\alpha = \alpha_0$. Then after n_{α} , $2n_{\alpha}$, $3n_{\alpha}$, … , and kn_{α} replays, α be come $\alpha_0(\alpha_{decay})$, $\alpha_0(\alpha_{decay})^2$, $\alpha_0(\alpha_{decay})^3$, … , and $\alpha_0(\alpha_{decay})^k$, respectively. This is illustrated in the figure 10 below.

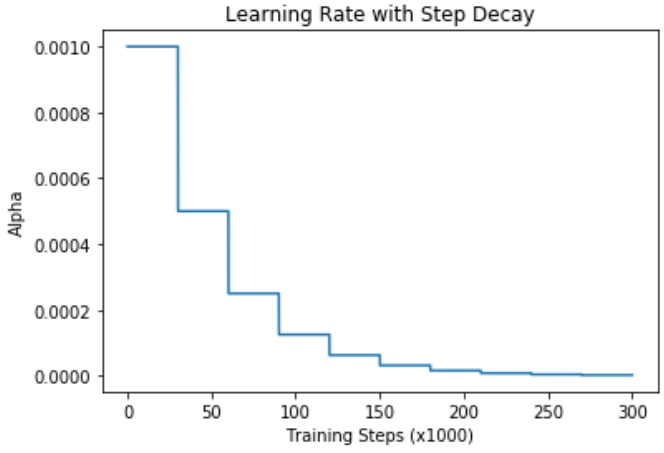

Figure 10: Graph of learning rate vs training steps with step decay

The replay memory is set to have the capacity of N_{buffer} and each replay samples N_{sample} number of experiences from the memory.

4.7.5 Neural Network Architecture and Optimiser

In this project, the neural network is implemented using deep learning package "PyTorch". The neural network architecture consists of 2 fully connected hidden layers. Input layer has 64 and 103 nodes for LiDAR and depth estimation experiment, respectively. The output layer have 3 nodes, representing the size of the action space. The first and the second hidden layers have N_{l1} and N_{l2} nodes, respectively. The activation functions used from the input layer through the first and second hidden layer, is rectified linear function. From the second hidden layer to the output layer, the activation function is linear.

To further increase the learning stability, the temporal loss $\left(Q^*(s,a) - Q^{\pi}(s_t, a_t)\right)$ is calculated by Huber loss[12]. Huber loss is defined by equation (9):

$$L_{\delta}(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta, \\ \delta\left(|a| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases} \tag{9}$$

Where a is temporal loss $\left(Q^*(s,a) - Q^{\pi}(s_t, a_t)\right)$. This means loss function grows in quadratic only for small temporal loss before becomes linear after a certain point. Therefore, if the temporal loss is big $(> \delta)$, then Huber loss will prevent loss function from exploding, unlike mean square error loss (see figure 11).

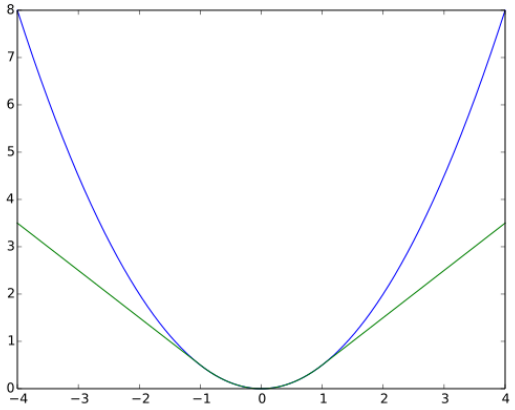

Figure 11: Graph of mean square error loss function (blue) vs Huber loss function (green)

The Huber loss in PyTorch is defined by the class `smooth_l1_loss(x,y)`, which set $\delta = 1$.The loss function is minimised by the optimizer called "Adam" with learning rate α.

5.7.6 Training Parameters

The training parameters for this project are shown in the table 2 below

Parameters	LiDAR	Depth Estimation
ϵ_0	1	1
ϵ_{decay}	0.005	0.003
ϵ_{min}	0.01	0.01
γ	0.95	0.95
α_0	0.001	0.001
α_{decay}	0.5	0.5
n_α	30,000	30,000
n_{tg}	3,000	3,000
N_{buffer}	300,000	300,000
N_{sample}	64	64
$[N_{l1}, N_{l2}]$	[64 , 64]	[128 , 128]

Table 2: Learning parameters of the two experiments

The Pseudo code for the training phase of DQL agent is shown in the following page.

Algorithm 3 Training of DQL Agent

Input: State S

Output: Action $a \in \{straight, left, right\}$

Initialize replay memory D to capacity N

Initialize the policy network Q with random weights θ

Initialize the target network $\hat{Q}$ with policy network's weights $\hat{\theta} = \theta$

Initialize $Tstep_{total} = 0$

Initialize $\epsilon = 1$

Initialize $\alpha = 1$

while $Tstep_{total} < Tstep_{total,max}$ **do**

Initialize $Tstep_{episodic} = 0$

while not crash **and** $Tstep_{episodic} < Tstep_{episodic,max}$ **do**

Observe current state s_t

if random(0,1) $> \epsilon$ **then**

Select a random action a_t

else

Predict $a_t = \mathbf{argmax}_a Q(s_t, a;\ \theta)$ with policy network

end if

Execute action a_t based on ϵ-greedy strategy

Observe reward r_{t+1} and next state s_{t+1}

Store experience $(s_t, a_t, r_{t+1}, s_{t+1})$ in D

Sample random minibatch of batch of $(s_j, a_j, r_{j+1}, s_{j+1})$ from D

Set $y_i = \begin{cases} r_j & \text{if } j \text{ is terminal episode} \\ r_j + \gamma \max\limits_{a_{j+1}} \hat{Q}\ (s_{j+1}, a_{j+1};\ \hat{\theta}) & \text{otherwise} \end{cases}$

Set $a = y_i - Q(s_j, a_j; \theta)$

Calculate Huber loss $L(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta, \\ |a| - \frac{1}{2}\delta, & \text{otherwise.} \end{cases}$

Perform gradient descent on $L(a)$ with respect to θ with learning rate α

Every t_θ steps, set $\hat{\theta} = \theta$

Every t_α steps, set $\alpha = 0.5\alpha$

Set $Tstep_{episodic} = Tstep_{episodic} + 1$

Set $Tstep_{total} = Tstep_{total} + 1$

end while

$\epsilon = \max\left(\epsilon_{min}, \epsilon\left(1 - \epsilon_{decay}\right)\right)$

end while

The actual code written in Python is available in the appendix D, line 151.

4.8 Reinforcement Learning: Testing Phase

During training, the neural network models at every 10,000 training steps are recorded to be later used for the testing phase. During the testing phase, each of the neural network models is used to guide the agent through the map with the same setting as training. Each test lasts for 100 episodes. The length of each episode is the same as in training. Epsilon value is also set to be 0 to prevent the agent from exploring. During the testing phase, there is no experience replay. The model that has the best performance in the testing phase from each experiment is used as the 'master' model for a more in-depth analysis of the agent's behaviour. Python code for testing phase is available in appendix I.

4.9 Connection Between Python and Unity's C#

Since the simulation is written in C#, while the Reinforcement Learning algorithm is written in Python, the connection between these two programs needs to be establish. This is done by using "`socket`" packages available in both Python and C# [2]. This allows the information to be sent and received back and forth between Python and Unity's C#. Once the Python code starts while Unity is running, the connection is established, and Python sends the command to Unity to reset the environment. The Unity then sends the current state of the agent back to Python for the learning algorithm and waits for Python to send the action to be executed by the agent to Unity. During this wait, Unity simulation still keeps on running. This means Unity undergoes a lot more update loops than Python. The simulation in Unity is set to run at a fixed speed of 50fps or 0.02s second per frame. Each loop in Python is varied in execution time. For the LiDAR experiment, each loop can run at a speed of around 0.03 seconds. For depth estimation experiment, with a time delay of 0.1 seconds, each loop then takes around 0.13 seconds to run. This process iterates until the terminal condition is reached, which is when Python send the reset command again, ready for the new episode. Python code for connect is available in appendix F, and C# code for connection is available in appendix B, line 293.

5. RESULT & DISCUSSION

5.1 Benchmark Controller (Braitenberg controller)

	Normalised Episode Length	Crash	Straight	Left Turn	Right Turn
LiDAR	0.888	19.00%	3.21%	48.77%	48.01%
Depth Estimation	0.941	22.00%	0.00%	49.85%	50.15%

Table 3: Performance statistics of Braitenberg controller with the two types of input

According to table 3, On average, the controller with depth estimation can survive significantly longer than the controller with LiDAR. However, when considering the crash percentage, the controller with depth estimation performs worse. This is due to the limited field of view of the camera in the depth estimation experiment, which causes the tendency to fail when the UAV encounters a corner as shown by flight paths in figure 14. This is because the controller is programmed to fly towards the direction with the highest value of average distances. As the UAV encounters a corner, the direction which has the highest average distances is the straight direction since it cannot see the entire surrounding (see figure 12). This limitation of the benchmark controller is later tested in the RL controller in section 5.3.3

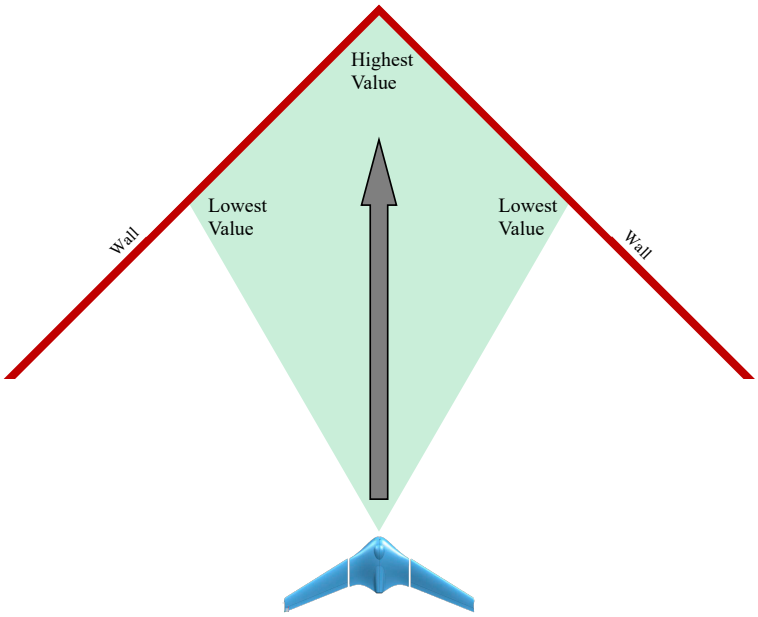

Figure 12: Situation where the UAV encounters a corner of the wall

Since the corners are quite far from the origin, the UAV would have flown for quite some time before crashing in those episodes that fail at wall or corner, and thus the average episode length does not drop as much. In fact, if the episodes that the controller fails at a corner are neglected, the crash percentage would be even better to that of the controller with LiDAR (18% vs 19%), while the average episode length is still higher. Therefore, in the environment with no walls or corners, the performance of the controller with depth estimation would be better than that of the controller with LiDAR.

The main reason for the inferior performance if LiDAR case is that LiDAR has the limitation of measuring distances on the horizontal plane as mention in section 1.2. Thus, when the UAV has a non-zero roll angle, the output distances from LiDAR is greater than the true distances to the surrounding object since each ray is measuring at an angle (see figure 1). It needs to take the roll angle into account to be able to calculate the true distance. Unfortunately, unlike RL algorithm, the Braitenberg controller is too simple, and the roll angle is not taken into account. Therefore, the controller with LiDAR overestimates the distances to the surrounding obstacles and crashes. With depth images, on the other hand, the controller always knows true distances regardless of the roll angle since it has 3D information, and thus fails less often.

The controller with both inputs is suffered to high sensitivity since the controller is set to turn whenever it detects any obstacles. This causes the agent to make a lot of unnecessary turns. The result is that the UAV has a very jiggly or swaying movement. This can be seen by the high percentage of turning in table 3.

The controller with LiDAR is also suffered from another problem, which is overturning. Since the span angle of LiDAR is very wide, if the agent encounters an obstacle in front of it, it would turn until that obstacle is out of the LiDAR range. This means that the agent could be turning for almost 90° just to avoid a single obstacle. With multiple obstacles in its way, the agent sometimes does a full 180° turn instead of flying through the gap between obstacles. Therefore, there's a lot of time when the agent only flies around within the same area or gets stuck among obstacles. This is illustrated by the flight paths in figure 13 below. The flight path of the LiDAR experiment has a lot of loops, whereas the path in the depth estimation experiment does not.

All of these drawbacks of the benchmark controller will be assessed against the RL controller in section 5.3.2.

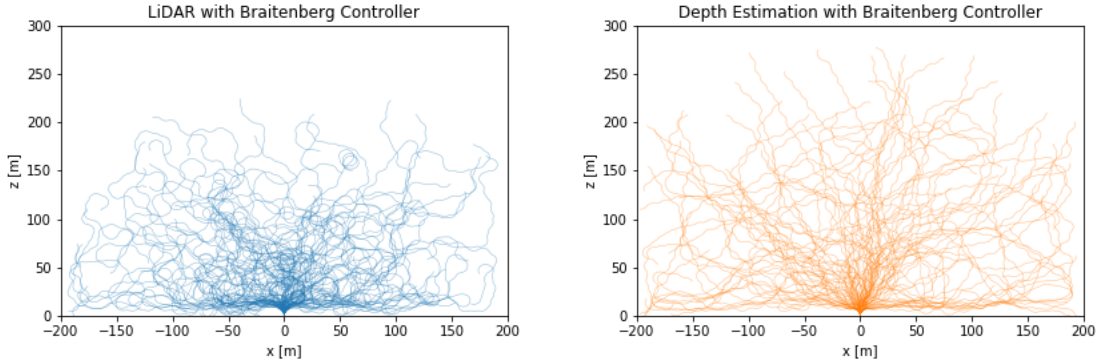

Figure 13: Flight paths of each episode from Braitenberg controller with LiDAR (left), and with depth estimation (right)

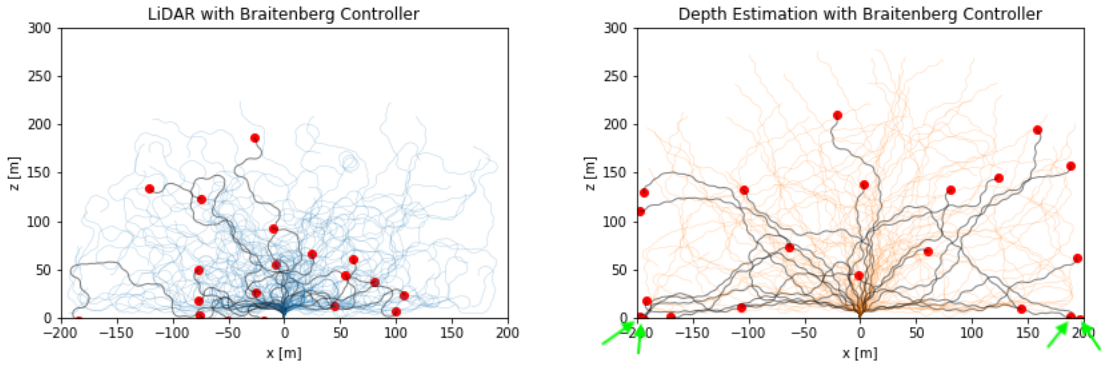

Figure 14: Flight paths of fail episodes (black lines) and crashing point (red dots) from Braitenberg controller with LiDAR (left), and with depth estimation (right). Green arrows point the crashes that happen at corners.

5.2 RL Controller: Training

There are two main indicators in measuring the learning progress of the agent, episodic cumulative reward and loss value. Since the numbers of training steps per episode of each experiment are different, then the episodic cumulative rewards of both experiments need to be normalized by dividing the rewards with maximum possible rewards. Also, due to a large difference in the number of training steps per episode of two experiment, the episodic reward is plotted against the number of training step instead. This shows a better comparison in the learning progress since the "learning" occurs at every time step instead of every episode. This results in the graph in figure 15 below.

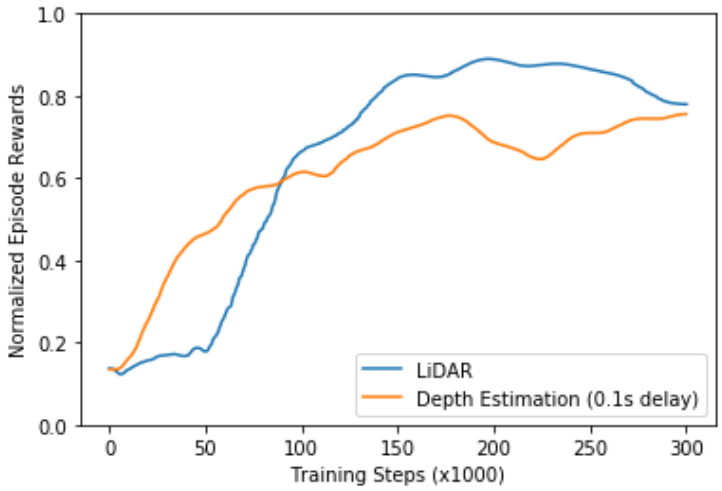

Figure 15: Learning curve of the RL agents in each experiment over the whole training period

As figure 15 shows, the depth estimation experiment exhibits a much steeper learning curve than LiDAR experiment. However, the average normalized episodic reward of depth estimation experiment reaches its highest value of about 0.78 at around 175,000th training step. On the other hand, the average normalized episodic reward of LiDAR experiment reaches the value of around 0.9 before drops slightly. The drop in reward could be caused by various factors such as the learning rate at the very end of the training may be too high. The superior learning progress of depth estimation experiment is also supported by the loss graph in figure 16.

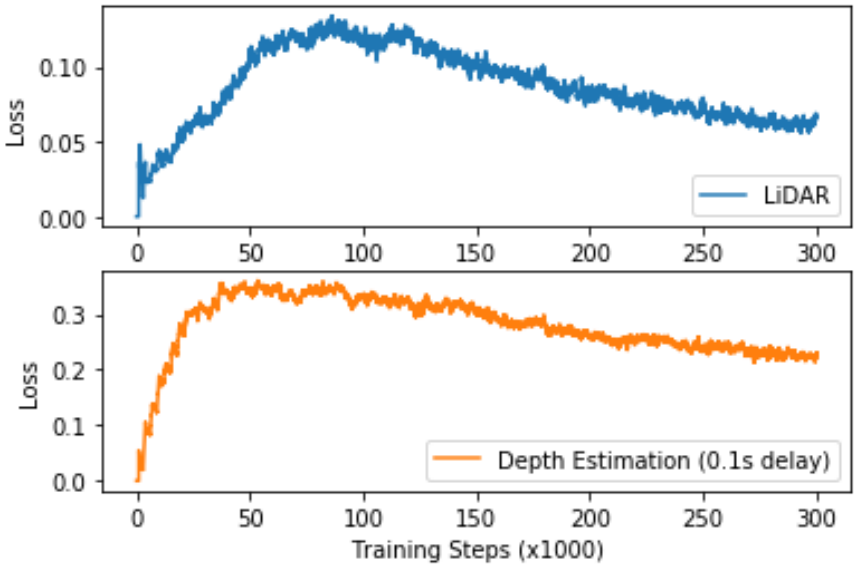

Figure 16: Loss values at each training step of both experiments

As graphs in figure 16 shows, the loss of LiDAR experiment takes much longer to start converging. This could be the source of the slower learning progress of LiDAR experiment. This results strongly contradict a former believe that depth estimation experiment would take much longer to converge due to a larger size of input and neural network. However, the time delay that is implemented on the depth estimation experiment actually plays a vital role in learning rate. According to [23], a technique called "frame skipping" could significantly accelerate the learning rate of an agent. Frame skipping is when the agent reacts to the environment only after k frames, rather than every single frame. The general trend is that the more frames are skipped, the faster the learning rate. When the agent does not skip any frame, the difference between the current state s_t and next state s_{t+1} is small. With more frames skipped, each (s_t, s_{t+1}) pair becomes more differentiable. Thus, According to [24], frame skipping prevents the agent from populating the replay memory with experiences that almost imperceptibly differ from one another. However, with too many frames skipped, the performance of the agent could be deteriorated.

The time delay in the depth estimation experiment is very similar to the frame skipping. Although LiDAR experiment also skips some frames as well (since Python loop always run slower than Unity), the number of frames skipped is a lot less than in the depth estimation experiment. The larger number of skipped frames could be the main reason for the faster learning

rate in the depth estimation experiment. However, this faster learning rate may not actually be an advantage, as one might think. So far, the learning rate that has been discussed is based on the number of training step. However, one must recall that each training step does not take the same amount of time in each experiment. In the LiDAR experiment, each training step takes about 0.03 seconds, while in the depth estimation experiment, each training step takes about 0.13 seconds. Therefore, for a certain amount of training steps, depth estimation experiment would take more than 4 times amount of actual time to run. Therefore, in real-life time, the agent in the depth estimation experiment would take much longer real-life time to learn.

As mentioned that the performance of the agent is also affected by the frame skipping. This is because the agent cannot react as quickly when it encounters obstacles. The agent in the depth estimation experiment also fails many times when it has to fly through narrow gaps, where precise sets of actions are required to navigate the UAV safely.

In order to validate this claim about the effect of frame skipping, another depth estimation experiment was conducted but with less time delay. The result compared to the previous experiments are shown below in figure 17.

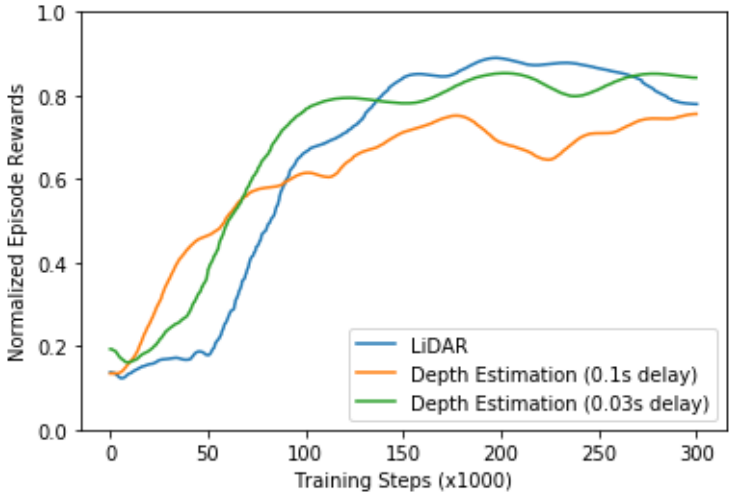

Figure 17: Learning curve of the RL agents in each experiment over the whole training period

It can be seen that with less time delay, the learning rate of the depth estimation experiment becomes slower, while the performance significantly improves to be comparable to the one in LiDAR experiment. This means that the main factor that reduces the performance of the depth

estimation experiment comes from the time delay, while other factors such as low resolution and limited field of view may not have any noticeable effect. However, in the previous section, it was mentioned that the limited field of view does have a huge negative impact when the benchmark controller encounters a corner. It is still too early to assume that this limitation does not affect the RL controller. The trained RL controller would never have to come across any corner as it would always fly towards the target goal, rather than to the side corner (see flight paths in figure 21). Therefore, the RL controller needs to go through further testing to see how the limited field of view affects the performance of the agent when encountering a corner. This will be discussed in section 5.3.3.

Unlike the benchmark controller, having 3D information does not seem to help the agent with depth estimation to outperform the agent with LiDAR, which has 2D information. This is due to two main reasons. Firstly, the RL algorithm has a roll angle as one of the input. Thus, even with 2D information, the algorithm could adjust its measurements according to the current roll angle. Secondly, the maximum roll angle is not too high that it would make the obstacles unreachable by the rays. Thirdly, the size of each obstacle is identical from top to bottom.

Another parameter that is worth looking at during the training is the percentage of each action. Since the reward function does not punish the agent for any turning, the agent is actually not expected to favour the "go straight" action. It is possible of the agent to always be turning while keeping its path relatively straight, such as by alternating between left and right. However, the graphs in figure 18 below show that the agent would favour the "go straight" action in both experiments more and more as it learns.

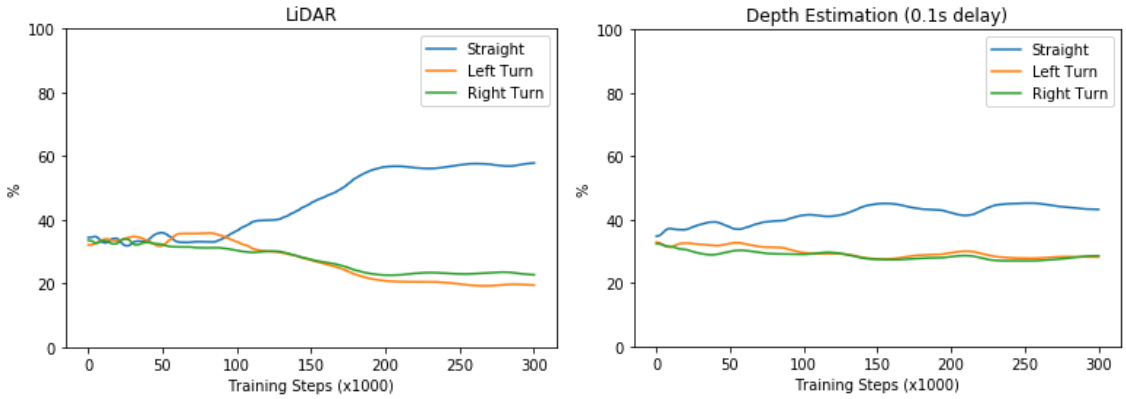

Figure 18: Percentage of each action over the whole training of LiDAR experiment (left) and depth estimation experiment (right)

It's worth noting that once the agent in LiDAR experiment has learned, it chooses to go straight a lot more than the agent in the depth estimation experiment. This means the agent in LiDAR experiment flies more stable. One possible explanation is the shorter range of LiDAR rays when compared to the camera. In this project, each LiDAR rays can detect an object at a maximum range of 20m, while the camera can see the object as far as 100m away. Thus, at a certain moment, the agent in the depth estimation experiment tends to detect a lot more obstacles than the agent in the LiDAR experiment. Being able to detect too many obstacles at once could be the source of noise for the agent that causes the agent to react more sensitively.

5.3 RL Controller: Testing

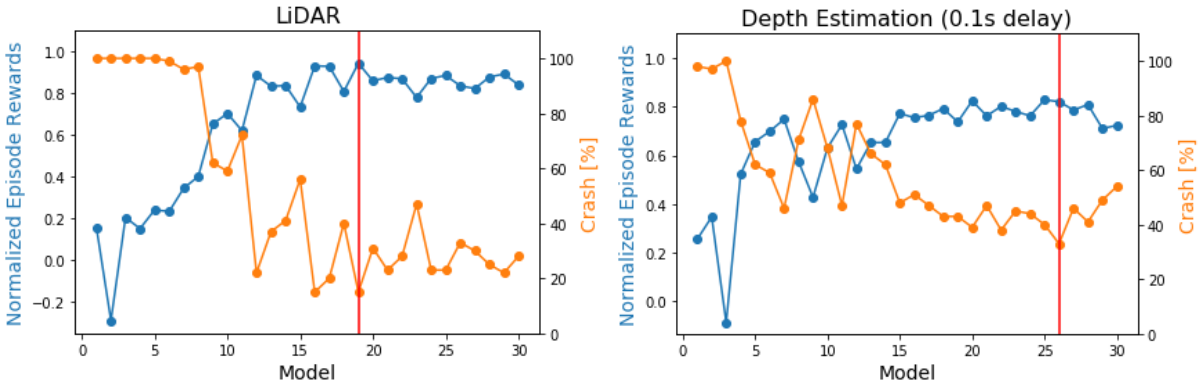

Figure 19: Normalised average episode reward during testing of models from LiDAR experiment (left) and models from depth estimation experiment (right)

According to the graphs in figure 19, the model 19 of LiDAR experiment and model 26 of the depth estimation experiment are the best performers (lowest crash percentage) among their classes. The graphs also support the claim about a faster learning rate in the depth estimation experiment. Furthermore, according to the flight paths in figure 20, the agent in the depth estimation experiment shown an unbelievable steep learning curve. With model 1 (model at 10,000th training step), the agent already shows the behaviour of flying towards the target point while the agent LiDAR experiment starts to fly towards the target point after 30,000 training steps. From observation during testing, the agent in the depth estimation experiment with model 2 already shows the sign of being able to avoid obstacles while the agent in LiDAR experiment does not exhibit this behaviour until model 4.

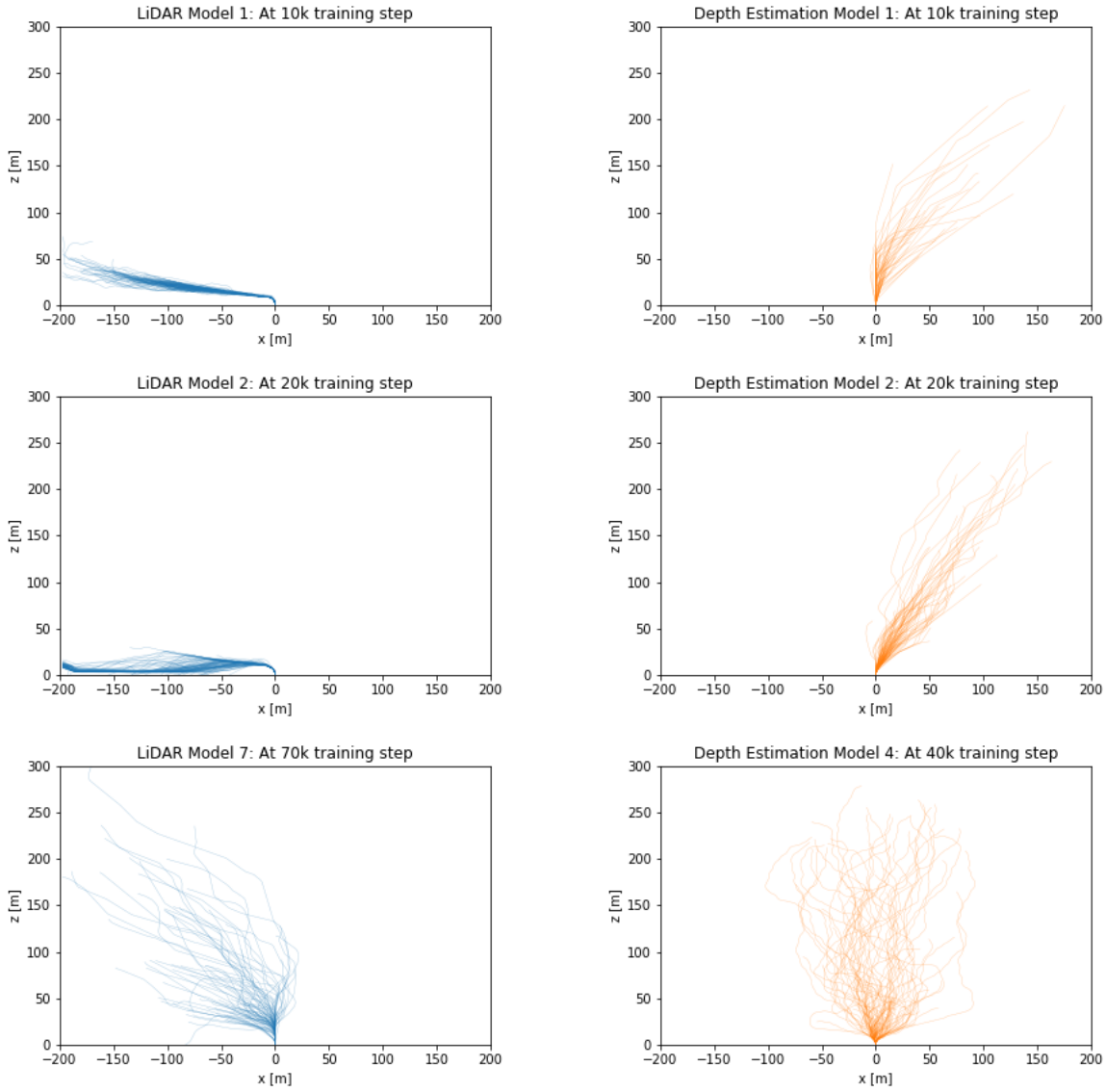

Figure 20: Flight paths of model 1, 2, and 7 from LiDAR experiment (left), and model 1, 2, and 4 from depth estimation experiment (right)

When looking at flight paths of some of those models when the learning has been stable in figure 21, it can be clearly seen that some model would favour flying to one side of the map, while some model does not favour to any side. Although ideally the model that does not favour any side would be preferred, they are all acceptable since the reward function only encourages the agent to closer to the target point at each time step.

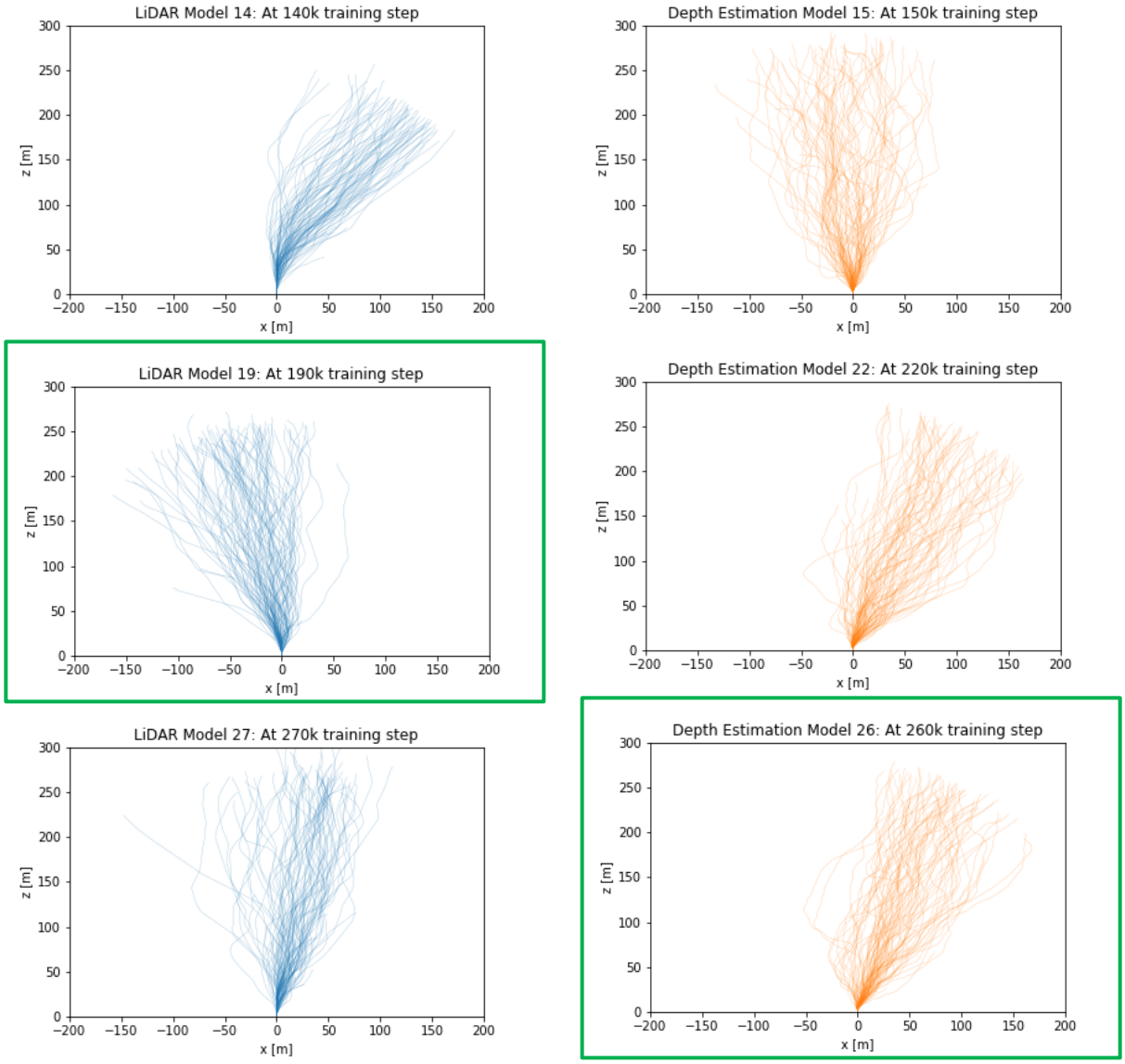

Figure 21: Flight paths of model 14, 19, and 27 from LiDAR experiment (left), and model 15, 22, and 26 from depth estimation experiment (right). Flight paths with green box are the path from best performer models

5.3.1 Flight Behaviour of the Best Performer RL Models

This section will qualitatively review and analyse the performance of the two best performer models in details by carefully observing the flight behaviour of the agent. RL agent with LiDAR shows a very steady and stable flight while the agent with depth estimation is more swaying. The cause of this has already been mentioned in section 5.3 and supported by graphs in figure 18. The clear difference between the two models is the ability to safely navigate through narrow paths or paths with a lot of obstacles. As mentioned in section 5.3, the slow reaction time of the agent with depth estimation significantly deteriorates the precision of the agent. It leads to many crashes when it encounters narrow or obstacle-dense paths.

On the other hand, the agent with LiDAR has a much faster reaction time and can navigate through tight situation relatively easy. This trend still stands during the testing phase and in the best performer models. This is the main reason why the performance of the agent with depth estimation is much lower than that of the agent with LiDAR.

5.3.2 Best Performer Models vs Benchmark Controller

	RL Controller		**Braitenberg Controller**	
	LiDAR (model 19)	Depth Estimation (model 26)	LiDAR	Depth Estimation
Episodic Reward	0.944	0.821	-	-
Episode Length	0.945	0.825	0.888	0.941
Crash	15.00%	33.00%	19.00%	22.00%
Straight	62.04%	48.18%	3.21%	0.00%
Left Turn	17.99%	25.91%	48.77%	49.85%
Right Turn	19.97%	25.92%	48.01%	50.15%

Table 4Overall performance statistics of the best performer models and that of benchmark controller. "Episodic Reward" and "Episode Length" are normalised average episode reward and length, respectively

According to table 4, the performance of the RL agent in LiDAR experiment is better than the one in the depth estimation experiment as expected. In fact, when compared to the benchmark controller, while the RL agent with LiDAR performs the best, the RL agent with depth

estimation has the worst performance. Since the benchmark controller is not programmed to obey the reward system, the normalised average episode length is used to compare the performance instead of a reward for a fair comparison.

As mentioned that the benchmark controller with depth estimation is suffered from the limited field of view that causes it to fail as it encounters a corner. If the episodes the benchmark controller fails at a corner is ignored, then the crash percentage would be very similar to that of RL controller with LiDAR (18% vs 15%), while the average episode lengths are relatively similar. Therefore, in the environment with no walls or corners, the performances of the RL controller with LiDAR and the benchmark controller with depth estimation would be relatively the same.

This excellent performance of the benchmark controller with depth estimation raises a question that challenges the previous claim. Previously (section 5.3), it was mentioned that the inferior performance of the RL controller with depth estimation, in comparison to RL controller with LiDAR, is caused mainly by the slow reaction time due to the time delay. If this is the case, then the performance of the benchmark controller with depth estimation should have been noticeably lower than that of the RL controller with LiDAR, and relatively similar to that of RL controller with depth estimation. The explanation is that, unlike RL controller, the benchmark controller is unconstraint by the flying direction. Since the RL controller is highly encouraged to fly towards the target point at all time, there are many situations where it is forced to fly into narrow gaps or the path with a lot of obstacles, rather than the other directions that have fewer obstacles.

On the other hand, the benchmark controller is set to fly towards the direction with the highest value of average distances. Therefore, the benchmark controller is very much less likely to fly through difficult paths. Since the time delay deteriorates the performance of the agent mostly in a tight situation like flying through difficult paths, the performance of the benchmark controller with depth estimation is not really compromised by the time delay. With this information, it seems like the RL controller would not decide to fly away from the target point at all cost, even though it means a chance of surviving to receive more cumulative reward goes up. However, to confirm this claim, a further test is required, which is conducted in section 5.3.3.

When considering the percentage of turning, it can be clearly seen that the RL controllers in both experiments fly a lot more stable than the Braitenberg controllers due to a much higher percentage of "straight" movement. This is one of a clear difference that separates the learning algorithm from a rule-based algorithm. Even though the RL agent is not discouraged from turning by the reward function, it stills converge to the policy that leads to a more stable flight than the benchmark controller. This is the sign of intelligence which the benchmark controller does not have

5.3.3 Test for Robustness of the Best Performer Models

In this section, various small experiments is conduct to test the robustness of the agents against changes in environment.

Square Cross-Section Obstacles

In this experiment, the cylindrical obstacles are changes to square cross-section prism. There are 3 sizes of the obstacles as before, and the length of the side of each obstacle is equal to $r_{obs,S}, r_{obs,M}, r_{obs,L}$ and the number of each obstacle per unit area is $\rho_{obs,S}, \rho_{obs,M}, \rho_{obs,L}$. The tests were run for 100 episodes for each input.

	Episodic Reward	Crash
LiDAR (model 19)	0.805	39%
Depth Estimation (Model 26)	0.747	51%

Table 5: Performance statistics of the best performer models against map with square cross-section obstacles

According to table 5, the performance of both models drops significantly with the change in the obstacles' shape. Thus, this means that the models are not generalised very well. However, the agent with LiDAR still shows superior performance over the agent with depth estimation as before.

Dead-End, Concaved Path

In a way, a corner is basically a dead-end, concaved path. In this experiment, the agents with each model are set to fly in a dead-end concaved path constructed by two walls, as shown in figure 22 and 23. The ability of the agent to avoid flying into the dead-end is heavily dependent on how deep the dead-end is and how far away head can the agent detect obstacles.

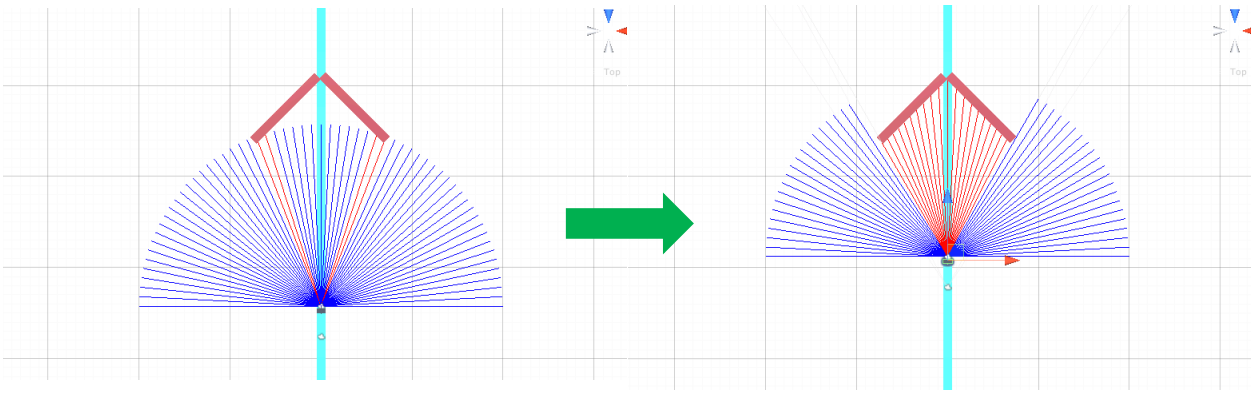

Figure 22: UAV is flying into a shallow, dead-end, concaved path

Consider the situation in figure 22, where the UAV is flying towards the corner. When the UAV first detects the obstacles, the agent chooses to go straight since what it sees is two obstacles side by side with a big gap in between that it can fly through. As it flies further in, the gap appears to be a dead-end. Both agents with LiDAR and depth estimation show that they can steer away from obstacles to the left or right side as soon as it detects the dead end. This is because both agents could detect the dead-end early, so they have enough time and space to steer away. However, if the dead-end is too deep inside, the agents would fail.

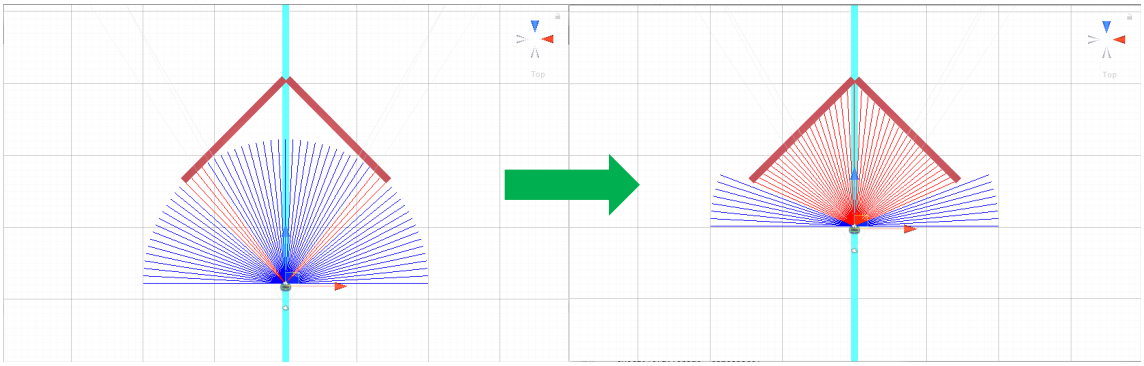

Figure 23: UAV is flying into a deep, dead-end, concaved path

Figure 23 shows the same situation with the dead-end is much deeper inside. By the type the agent detects the dead end, it is too far inside that if it were to steer away, it must sacrifices some reward by flying away from the target goal. However, both agents do not have this high-level thinking to be able to sacrifices some reward for longer survival time. Therefore, both agents would fly straight into the dead-end if the dead-end is too far inside. The only difference is that the agent with dept estimation can get out of this situation even with a much deeper dead end.

This is all due to the much longer detectable range of the camera. This is the only one and major drawback of RL controller with LiDAR in comparison to the benchmark controller with LiDAR. With benchmark controller, the agent with LiDAR would be able to survive this situation easily no matter how deep inside the dead-end is. This is because the agent is very sensitive to any detection and is programmed to turn straight away as soon as it detects the edge of the concaved path. This is why the benchmark controller with LiDAR would hardly fly through any gap. The very wide angle of LiDAR also guides the benchmark controller to turn to the side, unlike the depth image that guides the agent towards the middle due to the limited field of view as mentioned before. From this observation, another major disadvantage of the RL controller has been confirmed. That is, given a circumstance where the only way to survive longer to receive more cumulative reward is to fly away from the target point and sacrifices some reward, the agent would not choose to do it. This may be the result of the reward discount factor, γ, not being high enough, and therefore, the agent is not looking far enough in the future. Or it could be the result of the oversimplified reward function that it gives the same negative weight for the crash and for flying towards the target point. Another possible cause is that the punishment of crashing is too low that it might be better for the agent to crash and receive -1 reward once, rather than turn away from target goal to receive -1 reward at multiple time steps.

6. CONCLUSION

In term of the overall performance, RL agents with both types of inputs have shown adequate learning progress. They can survive the whole episodes most of the time while keeping their flight paths towards the target point. Using LiDAR gives a much better obstacle avoidance ability to the RL agent in comparison to using the depth estimation algorithm. The only main reason is due to the slower reaction time caused by the time delay in the depth estimation algorithm. However, this slow reaction time deteriorates the agent's performance mostly when it encounters a tight situation, such as flying through narrow gaps or obstacles-dense paths. This was proven by how the benchmark controller hardly undergoes through tight situations and with depth estimation algorithm, it can perform equally well as RL controller with LiDAR. Nevertheless, it was found that the time delay of the depth estimation algorithm helps the agent to learn much faster in term of the number of training step. This is due to the effect of frame skipping technique.

Having 3D information does not seem to improve the performance of the agent with depth estimation when compared to the agent with LiDAR that has 2D information. This is due to three main reasons; 1. the size of each obstacle is identical from top to bottom, 2. The RL algorithm knows the current roll angle of the UAV and thus, be able to adjust its measurement accordingly, 3. The maximum roll angle is not too high.

After all, the RL agents still have some flaws that are caused by the learning environment. RL agents with both types of inputs have a high tendency to fail when they encounter a dead-end, concaved path if the dead-end is too deep inside. This is when the longer detection range of camera increases the survival chance the RL agent with depth estimation to avoid a much deeper dead-end that, which the agent with LiDAR would fail due to having short range. However, the true root cause of this problem may originate itself in the reward function and learning parameter. Having the punishment for crashing equal to the punishment of flying away from the target point discourage the agent from turning away from the target point much more strongly than it encourages the agent to turn away from the dead-end. Having the value of reward discount factor, γ, being too low could limit the agent to look only a few time steps ahead, and thus fail to make a scarification of some reward in order to survive for longer. When tested for

robustness against the obstacles with square cross-section, the agents with both inputs can still survive an episode most of the time, but their chance of surviving drops significantly.

6.1 Recommendation

For any real-life application, if the environment that the UAV would be flying in consist of obstacles that do not have much variety of cross-section from top to bottom, such as buildings in the city, then LiDAR input would be enough to make the agent safely navigate through the environment. On the other hand, using a camera with depth estimation algorithm would make the agent much more versatile and robust to any environment due to the 3D information. However, if the environment is dense in obstacles, such as forest, then there is a need for a much faster depth estimation algorithm to increase the reaction time of the agent. Using the camera also gives an edge to the agent against the dead-end, concaved path. To completely eliminate the problem of the dead-end, concaved path, it might help by trying to increase the reward discount factor, γ. Also, the reward function could be set to punish the agent more heavily for crashing. However, this will break the reward clipping, which might destabilise the learning process of the agent.

6.2 Limitations

There are two major limitations about this experiment that could be the source of the inaccuracy of the result.

1. One of the assumptions that this whole project replies on is the perfect manoeuvrability of the UAV. In the real world, aerodynamics plays a very important role even when the RL controller is implemented with a PID controller for aircraft stabilisation. This project assumes that the PID controller can accurately control the aircraft at perfection, which might not be the case in the real world. With imperfect stability control, the performance of the agent to avoid obstacle could drop significantly.
2. Another assumption is about the simulation of the depth estimation algorithm. This project assumes that the depth estimation algorithm can predict the depth by any given images at the same accuracy. In the real world, one of the potential harm to a depth estimation algorithm, especially those that use machine learning, could be the tilt of images. When any fixed-wing aircraft rolls, the image from the camera will start to tilt.

When an image is tilted, some depth estimation algorithm that uses supervised learning could make a very inaccurate prediction if it has never been trained with tilted images. Therefore, in the real world, it may be possible that the depth estimation algorithm will not work at all when the UAV turns.

7. FUTURE WORK

To continue this project, there are many aspects that could be changed and improve to make the simulation closer to the real world.

1. To make the UAV's manoeuvrability fully 3 dimensional by introducing pitch up and down movement. However, this will increase the size of action space and the complexity of the learning process. It can lead to unstable learning and slow convergence time. This may be counteracted by changing the RL technique from DQL to others that can handle larger action space.
2. To increase the variety of obstacles. Obstacles with different cross-sectional shape, such as square, triangle, or thing rectangle could be implemented to generalise the trained model better. If the UAV has been modified to be able to move in 3D, the obstacles could be generated to have different heights and varied cross-section from top to bottom. This will force the UAV to make a more complicated decision. With the old setting, the UAV only needs to choose between a left or right to avoid an obstacle. Now, the UAV could choose to pitch up or down to fly over or under the obstacles as well.
3. In order to improve the stability of the learning process, or overall performance of the agent, prioritised experience replay technique could be implemented. Prioritised experience replay is when the agent sample the experience based on how big the temporal loss of each experience is. It will help the agent to sample more experiences that the neural network has not yet become certain to predict the Q-values, and thus improve the stability and, potentially, the performance of the agent.

8. REFERENCES

[1] CHOUDHARY, A. (2019). *Introduction to Deep Q-Learning for Reinforcement Learning (in Python)*. [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/ [Accessed 4 Nov. 2019].

[2] TCP socket connection to Unity with Python. (2017). [video] Available at: https://www.youtube.com/watch?v=eI3YRluluR4 [Accessed 15 Oct. 2019].

[3] Corrigan, F. (2019). *12 Top Lidar Sensors For Uavs, Lidar Drones And So Many Great Uses*. [online] Dronezon. Available at: https://www.dronezon.com/learn-about-drones-quadcopters/best-lidar-sensors-for-drones-great-uses-for-lidar-sensors/ [Accessed 18 February 2020].

[4] Tan, D. (2020). Depth Estimation: Basics And Intuition. *Medium*. Available at: https://towardsdatascience.com/depth-estimation-1-basics-and-intuition-86f2c9538cd1 [Accessed 3 Apr. 2020].

[5] Fu, H., Gong, M., Wang, C., Batmanghelich, K. and Tao, D. (2018). *Deep Ordinal Regression Network For Monocular Depth Estimation*. [online] p.1. Available at: https://arxiv.org/pdf/1806.02446v1.pdf [Accessed 3 April 2020].

[6] Pinke, R. (2018). *Buying Guide: Comparing Field Of View When Buying A Conference Room Video Camera*. [online] Video Conference Gear. Available at: https://www.videoconferencegear.com/blog/buying-guide-comparing-field-of-view-when-buying-a-conference-room-video-camera/ [Accessed 16 March 2020].

[7] Shyalika, C. (2019). *A Beginners Guide To Q-Learning*. Medium. Available at: https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c [Accessed 21 November 2019].

[8] Heberer, R. (2019). *Why Going From Implementing Q-Learning To Deep Q-Learning Can Be Difficult*. Medium. Available at: https://towardsdatascience.com/why-going-from-implementing-q-learning-to-deep-q-learning-can-be-difficult-36e7ea1648af [Accessed 9 November 2019].

[9] Mazzari, V. (2019). *What Is Lidar Technology?*. [online] Génération Robots - Blog. Available at: https://www.generationrobots.com/blog/en/what-is-lidar-technology/ [Accessed 10 December 2019].

[10] Kovan.ceng.metu.edu.tr. (2010). *Braitenberg Vehicles*. Available at: http://kovan.ceng.metu.edu.tr/software/Braitenberg/BraitenbergEN/Vehicles.html [Accessed 20 March 2020].

[11] Hasselt, H.V., Guez, A., Hessel, M., Mnih, V. and Silver, D. (2016). *Learning Values Across Many Orders Of Magnitude*. [online] Google DeepMind. Available at: https://arxiv.org/pdf/1602.07714.pdf [Accessed 7 April 2020].

[12] Hui, J. (2019). *RL — DQN Deep Q-network*. Medium. Available at: https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4 [Accessed 10 Feb. 2020].

[13] Lau, S. (2017). *Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning*. Medium. Available at: https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1 [Accessed 16 Jan. 2020].

[14] Manuelli, L. and Florence, P. (2017). *Reinforcement Learning for Autonomous Driving Obstacle Avoidance using LIDAR*. [online] *Github*. Available at: https://github.com/peteflorence/Machine-Learning-6.867-homework/blob/master/project/docs/project-writeup.pdf [Accessed 11 Feb. 2020].

[15] Xie, L., Wang, S., Markham, A. and Trigoni, N. (2017). *Towards Monocular Vision based Obstacle Avoidance through Deep Reinforcement Learning*. [online] *arxiv*. Oxford: University of Oxford. Available at: https://arxiv.org/pdf/1706.09829.pdf [Accessed 5 Mar. 2020].

[16] Wu, T.-C., Ho, C.-Y., Tseng, S.-Y., Lai, Y.-H. and Lai, C.-F. (2018). *Navigating Assistance System for Quadcopter with Deep Reinforcement Learning*. [online] *arxiv*. Available at: https://arxiv.org/pdf/1811.04584.pdf [Accessed 2 Nov. 2020].

[17] Madaan, R., Saxena, M.D., Bonatti, R., Mukherjee, S. and Scherer, S. (2016). *Deep Flight: Autonomous Quadrotor Navigation with Deep Reinforcement Learning*. [online] *Carnegie Mellon University School of Computer Science*. Pittsburgh: Carnegie Mellon University. Available at: https://www.cs.cmu.edu/~rbonatti/files/rss17.pdf [Accessed 14 Oct. 2020].

[18] Shin, S.-Y., Kang, Y.-W. and Kim, Y.-G. (2019). *Obstacle Avoidance Drone by Deep Reinforcement Learning and Its Racing with Human Pilot*. [online] *MDPI*. MDPI. Available at: https://www.mdpi.com/2076-3417/9/24/5571/htm#cite [Accessed 29 Jan. 2020].

[19] Singla, A., Padakandla, S. and Bhatnagar, S. (2018). *Memory-Based Deep Reinforcement Learning for Obstacle Avoidance in UAV With Limited Environment Knowledge*. [online] *arxiv*. Available at: https://arxiv.org/pdf/1811.03307.pdf [Accessed 22 Jan. 2020].

[20] Michels, J., Saxena, A. and Ng, A. (2005). *High Speed Obstacle Avoidance using Monocular Vision and Reinforcement Learning*. [online] *Stanford Artificial Intelligence Laboratory*. Available at: http://ai.stanford.edu/~asaxena/rccar/ICML_ObstacleAvoidance.pdf [Accessed 7 Feb. 2020].

[21] Revuelta, P., Ruiz, B. and Snchez Pe, J.M. (2012). Depth Estimation - An Introduction. *Current Advancements in Stereo Vision*. [online] Available at: https://cdn.intechopen.com/pdfs/37767/InTech-Depth_estimation_an_introduction.pdf [Accessed 25 Apr. 2020].

[23] Khan, A., Feng, J., Liu, S. and Asghar, M.Z. (2019). Optimal Skipping Rates: Training Agents with Fine-Grained Control Using Deep Reinforcement Learning. *Journal of Robotics*, [online] 2019. Available at: https://www.hindawi.com/journals/jr/2019/2970408/ [Accessed 25 Apr. 2020].

[24] Schulze, C. and Schulze, M. (2018). *ViZDoom: DRQN with Prioritized Experience Replay, Double-Q Learning, & Snapshot Ensembling**. [online] *arxiv*. Available at: https://arxiv.org/pdf/1801.01000.pdf [Accessed 20 Mar. 2020].

APPENDICES

Appendix A. PROJECT MANAGEMENT

As soon as the project brief was given by Prof. Crowther in the first week of the first semester, the project started right away. The very early stage of the project was mostly about familiarising with reinforcement learning (specifically, deep Q-learning), and selecting the research goal and environmental setups, such as UAV model and its movement.

Adjustment in Research Goal

The initial research goal was slightly different than the one on this project. Instead of comparing LiDAR with depth estimation algorithm, it was to compare LiDAR with object recognition algorithm. The change was made quite late (around 1 month before submission). The main reason for the change was that in reality, there is hardly anyone that process images with object recognition algorithm for obstacle avoidance. Most of the researches being done nowadays are working on obstacle avoidance using a depth estimation algorithm on camera images. Therefore, the initial research goal would be quite obsolete and would add no values to the industry. However, the change of research goal did not significantly affect the progress of the project since the implementation of the simulation of object recognition algorithm could be easily converted to the simulation of the depth estimation algorithm.

C# and Python

After the environmental setup has been planned, the software to simulate the environment was chosen to be Unity3D for 2 main reasons; 1. Its user interface allows for a quick and easy setup of the 3D environment. 2. Its realistic physics allow the research to be carried on further with real aerodynamics if there's time available. However, many problems also risen by using Unity3D. First, the programming language of Unity is C#, while the RL algorithm has to be written in Python. C# was also very new to me at that time. Therefore, a lot of time and effort was put into learning C# and establishing communication between Python and Unity's C#.

Change of UAV's Dynamic and Control

At first, the environment was set up to be in a fully 3D environment with realistic aerodynamics. That is the agent controls the UAV by moving elevons up and down to roll or pitch the plane. However, this setting was too complicated for the agent to learn since the agent must also learn to stabilize the plane alongside with the main goal. Therefore, it was later changed to the one on this project.

The Evolution of the Map and UAV's Task

The environment was first created to be a forest full of obstacles (Environment #1). This was only to test the very early version of DQL algorithm. Later on, when the research goal had not been changed yet, it was changed to nothing but a couple of obstacles that would keep respawning in front of the UAV as it flies past the previous ones (Environment #2). The reason of this map was that if the UAV were to fly in a map with a lot of static obstacles, the agent would fail to learn with object recognition experiment since no matter how far the objects are, it would appear in the camera view and be detected. Therefore, the agent would always see a screen which almost every pixel contains a part of the object in it. Having only a couple of objects respawning in front of the UAV once at a time would allow the agent to see which pixels do not contain obstacles and thus, be able to learn. However, the flaw of this setup is that the agent tends to fly around in a circle, which was fixed by setting negative reward for turning. However, setting a negative reward for turning, even only a very slight amount also caused the agent to be very stiff in turning and thus perform poorly. This was counteracted by setting a heavily negative reward when crashing. After countless attempts to achieve a stable learning curve and performance with this setup by fine-tuning the learning parameters and reward function, it was found that having such a wide range of possible reward at each time step actually causes the divergence of the loss function which is the reason of the inconsistency in the agent's performance[11] and reward clipping is required to achieve stable performance. Therefore it was necessary to find a suitable combination of map and UAV's task that allows reward clipping to be implemented without making the agent flies around in circle nor becoming too stiff to turn. Then, when the research goal changed from object recognition to the depth estimation, this means that the map can be filled with a lot of static obstacles rather than a couple of respawning obstacles. However, this did not completely eliminate the problem of the agent flying around in

circle since the agent would occasionally find an empty space that is big enough for it to circle around. The agent could sometime fly along the side to the walls to minimize the interaction with obstacles. Therefore, the subtask of flying towards the target point is implemented to guide the agent into a certain direction rather than letting it fly around aimlessly (Environment #3).

The Progression of DQN Training Process

After the environmental setup has been finalised, various improvements of the training process were being introduced more and more. The main aim was to improve the stability of the learning progress even further. These improvements include target Q-network, step decay of learning rate, and Huber loss function. Throughout the experiments, the learning parameters were always being tuned to optimise the learning process.

Reflection

Since the beginning of this project, a huge amount of time and effort has been put into experimenting with the right setup of the environment. This also includes familiarising myself with Unity and C#. More importantly, a large amount of time was wasted into creating realistic aerodynamics of the UAV that was proven to be too complicated for the agent to learn. Thus, making the environment as simple as it can be is probably one of the most important things at the early stage of the project to quickly see the progress of the RL algorithm. Perhaps, the part that consumed the most amount time found the right combination of the map, agent's task, and reward function that would lead to stable learning, excellent performance, and desired behaviour of the agent. Therefore, all of these left only a little time for experimenting with the training process and parameters. I believe that with more time available for the training process, the performance of the agent would be much better with a better set of training parameters and implementation of other techniques.

Gantt Charts

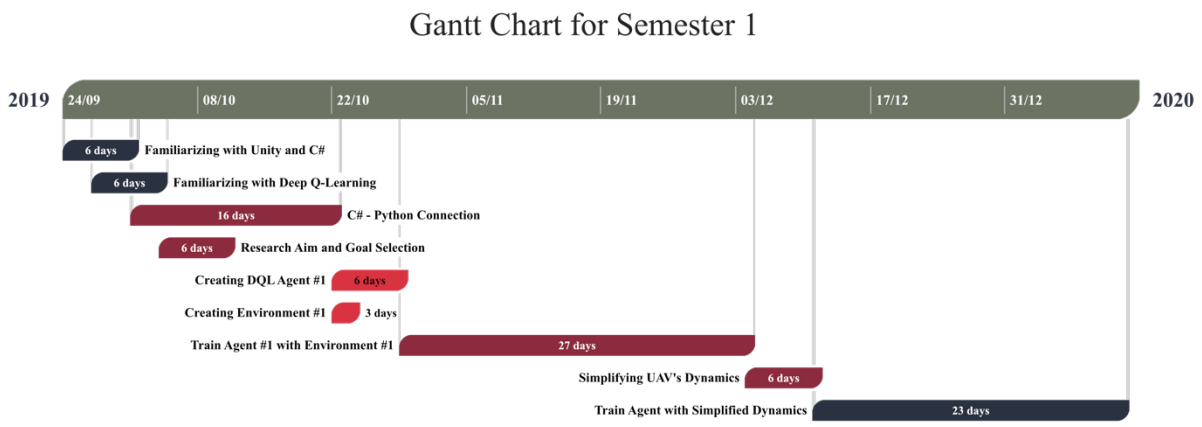

Figure 24: Gantt Chart for Semester 1

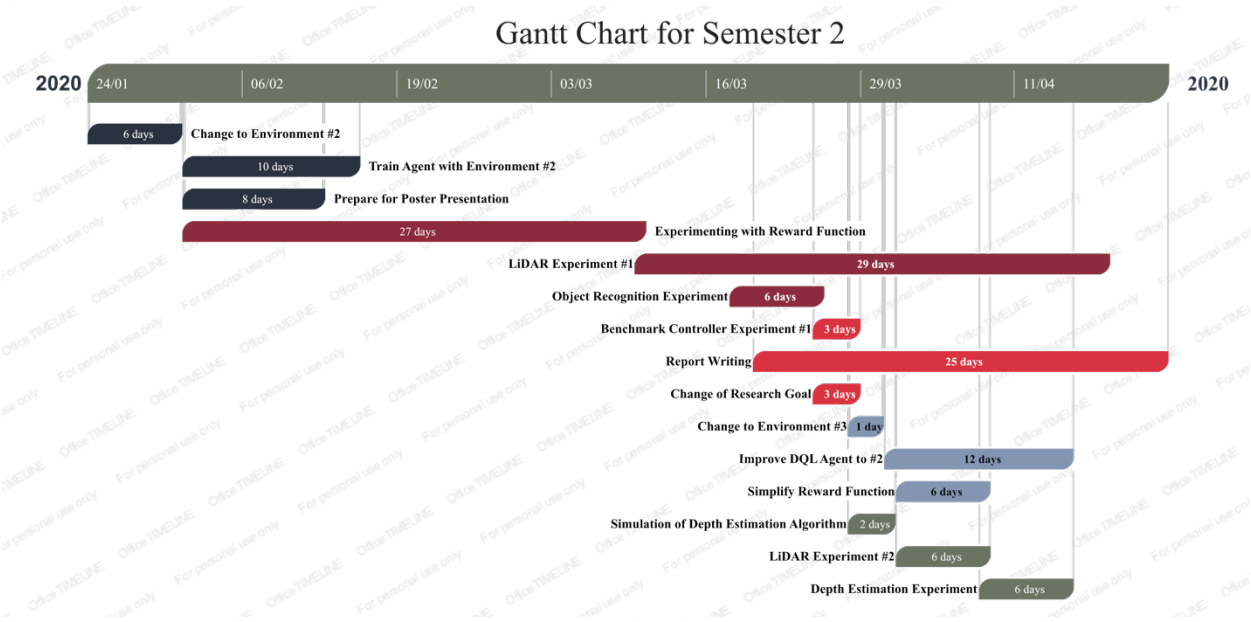

Figure 25: Gantt Chart for Semester 2

Appendix B: Main C# Code for UAV Control and Connection with Python

```
1.  //============================================================================
2.  //Simulation Inputs
3.  //============================================================================
4.  [Header("UAV Specification")]
5.  public float Speed;
6.  public float TurnSpeed;
7.  public float MaxRollAngle;
8.  public float MaxRollSpeed;
9.  public Vector3 InitialPosition;
10.
11. public enum PlayMode { Human, AI };
12. public PlayMode playMode;
13. public enum InputType { LiDAR, Visual };
14. public InputType inputType;
15.
16. [Header("Sensors")]
17. public float sensorLength;
18. public float SensorPosition;
19. public float SensorAngleIncHor;
20. public float SensorAngleIncVer;
21. public int NumSensorsHor;
22. public int NumSensorsVer;
23. //----------------------------------------------------------------------------
24.
25.
26. //============================================================================
27. //Declarations
28. //============================================================================
29. Thread mThread;
30. string connectionIP = "127.0.0.1";
31. int connectionPort = 25001;
32. IPAddress localAdd;
33. TcpListener listener;
34. TcpClient client;
35. bool running;
36. [Header("Reset State")]
37. public bool Reset;
38. string state_string;
39. bool GateOpen;
40. float IsCollided = 0.0f;
41. int command;
42. string encodedImage;
43. Vector3 pos;
44. float[] info;
45. float[] DistanceArray;
46. float RollAngle;
47. float Dist2TargetCurrent;
48. private bool left;
49. private bool right;
50. private float rollSpeed;
51.
52. //----------------------------------------------------------------------------
53.
54.
55. void Start()
56. {
57.     ///Write specifications into a text file for Python to read
58.     if (inputType.Equals(InputType.LiDAR))
```

```
    {
        info = new float[]{NumSensorsHor, NumSensorsVer, SensorAngleIncHor,
            SensorAngleIncVer, sensorLength, Speed, TurnSpeed,
            MaxRollAngle, rollSpeed};

    }
    else if (inputType.Equals(InputType.Visual))
    {
        info = new float[]{ 0, 0, Camera.main.fieldOfView,
            Camera.main.farClipPlane, Speed, TurnSpeed, MaxRollAngle,
            rollSpeed};
    }
    string Path = Directory.GetCurrentDirectory();
    var MainPath = Directory.GetParent(@Path);
    string info_string = string.Join(" ", info);
    info_string = inputType + " " + info_string;
    System.IO.File.WriteAllText(@MainPath + "/spec.txt", info_string);

    //Start connection with Python
    ThreadStart ts = new ThreadStart(GetInfo);
    mThread = new Thread(ts);
    mThread.Start();
}

void FixedUpdate()
//Main update loop for simulation. This loop runs at 50FPS.
{
    Vector3 GoalPos = Goal.pos;
    Vector3 Vec2TargetInitial = new Vector3(GoalPos.x - InitialPosition.x,
        GoalPos.y - InitialPosition.y, GoalPos.z - InitialPosition.z);
    //=========================================================================
    //Reset & Re-initialise the simulation
    //=========================================================================
    if (Reset)
    {
        Rigidbody.transform.position = InitialPosition;
        Rigidbody.velocity = new Vector3(0.0f, 0.0f, 15.0f);
        Rigidbody.transform.rotation = Quaternion.identity;
        Rigidbody.angularVelocity = new Vector3(0.0f, 0.0f, 0.0f);
        IsCollided = 0.0f;
        Dist2TargetCurrent = Mathf.Sqrt(
                    Vec2TargetInitial.x * Vec2TargetInitial.x
                    + Vec2TargetInitial.z * Vec2TargetInitial.z);
           }

           //=========================================================================
           /*Calculate Difference in distance to the target goal between
            * the previous time step and current time step. */
           //=========================================================================
           pos = Rigidbody.transform.position;
           Vector3 Vec2Target = new Vector3(GoalPos.x - pos.x,
                                            GoalPos.y - pos.y,
                                            GoalPos.z - pos.z);
           float Dis2TargetNew = Mathf.Sqrt(Vec2Target.x * Vec2Target.x
                                           + Vec2Target.z * Vec2Target.z);
           float Dis2TargetDiff = Dist2TargetCurrent - Dis2TargetNew;
           Dist2TargetCurrent = Dis2TargetNew;
```

```
            //========================================================================
            /*Calculate angle between the flying direction of the UAV and
             * the vector from the UAV to the goal. */
            //========================================================================
            float Angle2Target = Mathf.Atan2((Vec2Target.x * transform.forward.z
                                             - Vec2Target.z * transform.forward.x),
                                              (Vec2Target.x * transform.forward.x
                                             + Vec2Target.z * transform.forward.z));

            //========================================================================
            //Gather states
            //========================================================================
            if (inputType.Equals(InputType.LiDAR))
            {
                //Get distanes from raycasts
                DistanceArray = Sensor();
            }
            float[] state1 = { IsCollided,
                               pos.x,
                               pos.z,
                               RollAngle/MaxRollAngle,
                               Dis2TargetDiff/(Time.deltaTime*Speed),
                               Angle2Target/Mathf.PI};

            if (inputType.Equals(InputType.LiDAR))
            {
                float[] state = new float[state1.Length + DistanceArray.Length];
                Array.Copy(state1, state, state1.Length);
                Array.Copy(DistanceArray, 0, state, state1.Length,
                           DistanceArray.Length);
                state_string = string.Join(" ", state);
            }
            else if (inputType.Equals(InputType.Visual))
            {
                state_string = string.Join(" ", state1);
                StartCoroutine(Encode());
                state_string = state_string + " " + encodedImage;
            }
            //========================================================================
            //Compute UAV's movement
            //========================================================================
            Rigidbody.velocity = transform.forward * (Speed);
            float RollSpeed = GetMovement(command, RollAngle); //Get Roll Speed
            RollAngle += RollSpeed;
            RollAngle = Mathf.Clamp(RollAngle, -MaxRollAngle, MaxRollAngle);
            Rigidbody.transform.RotateAround(pos, Vector3.up,
                        -Time.deltaTime * (RollAngle / MaxRollAngle) * TurnSpeed);
            var rot = RollAngle * Convert.ToInt32(RollAngle > 0) +
                     (RollAngle + 360) * Convert.ToInt32(RollAngle < 0);
            transform.localEulerAngles = new Vector3(transform.localEulerAngles.x,
                                                     transform.localEulerAngles.y, rot);
            //Draw line to show action
            Debug.DrawLine(pos,
                new Vector3(pos.x - RollAngle * transform.right.x / 2,
                pos.y - RollAngle * transform.right.y / 2,
                pos.z - RollAngle * transform.right.z / 2),
                Color.black);
            GateOpen = true; //Allow connection
        }
        //------------------------------------------------------------------------

```

```
181.        //========================================================================
182.        //Function that generates raycasts and output their readings
183.        //========================================================================
184.        private float[] Sensor()
185.        {
186.            RaycastHit hit;
187.            float Distance;
188.            Vector3 sensorStartPos = transform.position;
189.            sensorStartPos.x += SensorPosition * transform.forward.x;
190.            sensorStartPos.z += SensorPosition * transform.forward.z;
191.            float[] Distances = new float[NumSensorsVer * NumSensorsHor];
192.            var c = 0;
193.
194.            for (int i = 0; i < NumSensorsVer; i++)
195.            {
196.                var phi = (SensorAngleIncVer * i
197.                    - SensorAngleIncVer * (NumSensorsVer - 1) / 2);
198.                Vector3 direction_pre = Quaternion.AngleAxis(phi, transform.right)
199.                                                            * transform.forward;
200.
201.                for (int j = 0; j < NumSensorsHor; j++)
202.                {
203.                    var theta = (SensorAngleIncHor * j -
204.                        (SensorAngleIncHor * (NumSensorsHor - 1) / 2));
205.                    Vector3 direction = Quaternion.AngleAxis(theta, transform.up)
206.                                                            * direction_pre;
207.
208.                    Vector3 sensorEndPos = new Vector3(
209.                                    sensorStartPos.x + sensorLength * (direction.x),
210.                                    sensorStartPos.y + sensorLength * (direction.y),
211.                                    sensorStartPos.z + sensorLength * (direction.z));
212.
213.                    if (Physics.Raycast(sensorStartPos, direction,
214.                        out hit, sensorLength))
215.                    {
216.                        Distance = hit.distance;
217.                        Distances[c] = Distance / sensorLength;
218.                        Debug.DrawLine(sensorStartPos, hit.point, Color.red);
219.                    }
220.                    else
221.                    {
222.                        Distances[c] = 1;
223.                        Debug.DrawLine(sensorStartPos, sensorEndPos, Color.blue);
224.                    }
225.                    c += 1;
226.                }
227.            }
228.            return Distances;
229.        }
230.        //------------------------------------------------------------------------
231.
232.
233.
234.
235.
236.
237.
238.
239.
240.
241.
```

```
242.        //============================================================================
243.        //Get Roll Speed from command
244.        //============================================================================
245.        public float GetMovement(int command, float RollAngle)
246.        {
247.            if (playMode.Equals(PlayMode.Human))
248.            {
249.                left = Input.GetKey("left");
250.                right = Input.GetKey("right");
251.            }
252.            else if (playMode.Equals(PlayMode.AI))
253.            {
254.                left = command == 1;
255.                right = command == 2;
256.
257.            }
258.
259.            if (left)
260.            {
261.                rollSpeed = MaxRollSpeed;
262.            }
263.
264.            else if (right)
265.            {
266.                rollSpeed = -MaxRollSpeed;
267.            }
268.
269.            else
270.            {
271.                rollSpeed = -Mathf.Sign(RollAngle) *
272.                    Mathf.Min(MaxRollSpeed, Mathf.Abs(RollAngle));
273.            }
274.            return rollSpeed;
275.        }
276.        //----------------------------------------------------------------------------
277.
278.
279.        //============================================================================
280.        //Get collision state
281.        //============================================================================
282.        private void OnCollisionEnter(Collision collision)
283.        {
284.            IsCollided = 1.0f;
285.        }
286.        //----------------------------------------------------------------------------
287.
288.
289.
290.        //============================================================================
291.        //Prepare the connection with Python
292.        //============================================================================
293.        public static string GetLocalIPAddress()
294.        {
295.            var host = Dns.GetHostEntry(Dns.GetHostName());
296.            foreach (var ip in host.AddressList)
297.            {
298.                if (ip.AddressFamily == AddressFamily.InterNetwork)
299.                {
300.                    return ip.ToString();
301.                }
302.            }
```

```
303.            throw new System.Exception(
304.                "No network adapters with an IPv4 address in the system!");
305.        }
306.
307.        void GetInfo()
308.        {
309.            localAdd = IPAddress.Parse(connectionIP);
310.            listener = new TcpListener(IPAddress.Any, connectionPort);
311.            listener.Start();
312.
313.            client = listener.AcceptTcpClient();
314.
315.
316.            running = true;
317.            while (running)
318.            {
319.                Connection();
320.            }
321.            listener.Stop();
322.        }
323.
324.        void Connection()
325.        {
326.            if (GateOpen)
327.            {
328.                NetworkStream nwStream = client.GetStream();
329.                byte[] buffer = new byte[client.ReceiveBufferSize];
330.
331.                byte[] ToPy = Encoding.ASCII.GetBytes(state_string);
332.                nwStream.Write(ToPy, 0, ToPy.Length);
333.
334.                int bytesRead = nwStream.Read(buffer, 0, client.ReceiveBufferSize);
335.                string dataReceived = Encoding.UTF8.GetString(buffer, 0, bytesRead);
336.                print(dataReceived);
337.
338.                if (dataReceived != null)
339.                {
340.                    command = (int)StringToArray(dataReceived)[0];
341.                    Reset = Convert.ToBoolean(StringToArray(dataReceived)[1]);
342.
343.                }
344.                GateOpen = false; //Close the connection
345.            }
346.        }
347.
348.        public float[] StringToArray(string sArray)
349.        {
350.            if (sArray.StartsWith("[", System.StringComparison.Ordinal) &&
351.                sArray.EndsWith("]", System.StringComparison.Ordinal))
352.            {
353.                sArray = sArray.Substring(1, sArray.Length - 2);
354.            }
355.            string[] output = sArray.Split(',');
356.            float[] result = {
357.            float.Parse(output[0]),
358.            float.Parse(output[1])
359.            };
360.            return result;
361.        }
362.        //-----------------------------------------------------------------------
363.
```

```
364.
365.        //========================================================================
366.        //Encode current display image to base64 string
367.        //========================================================================
368.        IEnumerator Encode()
369.        {
370.            yield return new WaitForEndOfFrame();
371.            encodedImage = GetFrameEncoded();
372.        }
373.
374.        private string GetFrameEncoded()
375.        {
376.            var width = Screen.width;
377.            var height = Screen.height;
378.            var tex = new Texture2D(width, height, TextureFormat.RGB24, false);
379.            tex.ReadPixels(new Rect(0, 0, width, height), 0, 0);
380.            tex.Apply();
381.
382.            var bytes = tex.EncodeToPNG();
383.            encodedImage = Convert.ToBase64String(bytes);
384.            Destroy(tex);
385.            return encodedImage;
386.        }
387.        //------------------------------------------------------------------------
```

Appendix C: C# Code for Generating an Obstacle and Randomizing Its Position

```
1.  using UnityEngine;
2.
3.  public class RSAll : MonoBehaviour
4.  {
5.      /* This code is for generating and randomising an obstacle's position.
6.       * This code needs to be attached to each single one of every obstacles
7.       * to generate all of them together */
8.      public Airplane airplane;
9.      public float z_min;
10.     public float z_max;
11.     public float x_min;
12.     public float x_max;
13.     public float range1;
14.     public float range2;
15.
16.     void Start()
17.     {
18.         transform.position = new Vector3(Random.Range(x_min, x_max), 0, Random.Range(z_
    min, z_max));
19.     }
20.
21.     void Update()
22.     {
23.         // Regenerate the obstacle at the begining of each episde
24.         if (airplane.Reset)
25.         {
26.             transform.position = new Vector3(Random.Range(x_min, x_max), 0, Random.Rang
    e(z_min, z_max));
27.         }
28.     }
29. }
```

Appendix D: Main Python Code for Training the RL Agent

```
# ================================================================================
# Import required dependencies
# ================================================================================
from __future__ import print_function
import socket
import numpy as np
from collections import namedtuple
import pandas as pd
from datetime import datetime
import os
import shutil
from pathlib import Path
import torch
import torch.optim as optim
from transfer_data import transfer_data
from DQL import update_learning_rate, ReplayMemory,\
                ExperienceReplay, Agent, DQN
from OtherFunc import read_spec, plot_progress, save_result, \
                      save_model, choose_action_HC
# ================================================================================

"""Choose whether to start a new trainng or continue from the previous one"""
StartNewSim = True

"""Choose the type of controller.
'RL' => Reinforcement Learning controller,
'HC' => Hand controller (Breaitenberg controller) """
Controller = 'RL'

# ================================================================================
# Learning Parameters
# ================================================================================
gamma = 0.95          #Reward discount factor
epsilon_initial = 1   #Initial value of epsilon
epsilon_min = 0.01    #Minimum possible epsilon (the value it will anneal to)
epsilon_decay = 0.003 #Epsilon decay factor. (Higher -> faster decay rate)
alpha_initial = 0.001 #Initial value of learning rate
alpha_decay = 0.5     #Learning rate decay factor (Hgiher -> slower decay)
batch_size = 64       #The size of the batch of sampled experiences
target_update =3000   #Interval to update target network
lr_update = 30000     #Interval to step down learning rate
memory_size = 300000  #Capacity of replay memory
layers = [128,128]    #Number of nodes of 1st and 2nd hidden layers of NN
max_env_steps = 1000  #Maximum training steps of each episode
n_episodes = 10000    #Maximum number of episodes for one training
replay_epoch = 1      #Number of epoch for an experience replay

# ================================================================================
# Other Parameters
# ================================================================================
NumPixelHor = 10        #No. of horizontal pixels to resize the image to
NumPixelVer = 10        #No. of vertical pixels to resize the image to
DeptEstSpeed = 0.1      #Time delay for depth estimation algorithm
truestate = 3           #No. of elements in observable state excluding LiDAR/image
action_space_size = 3   #Size of action space
model_save_interval = 10000 #Interval to save NN model
sma_period_reward = 100 #period to caculate moving average of reward graph
sma_period_loss = 1000  #period to caculate moving average of loss graph
```

```
# ================================================================================
# Get Specifications from Unity
# ================================================================================
mainpath = str(Path(__file__).parents[1])
SpecPath = mainpath + '/spec.txt'
spec = read_spec(SpecPath,NumPixelHor,NumPixelVer)
InputDim = [spec.values[0,2], spec.values[0,1]]
InputType = spec.values[0,0]

# ================================================================================
# Start New Training or Continue from what was left off
# ================================================================================
if StartNewSim == True:
    """Initialise parameters, result tanks, and replay memory"""
    epsilon = epsilon_initial
    alpha = alpha_initial
    memory = ReplayMemory(memory_size)
    loss_array = []
    positions = []
    result = pd.DataFrame(columns=['Cumulative Reward', \
                'Time Step', 'Crash','Straight', 'Left Turn', 'Right Turn'])
    e_c = 0
    total_tstep = 0
    image_old = np.ones([NumPixelVer,NumPixelHor])
    Experience = namedtuple('Experience',
                        ('state','action','next_state','reward', 'crash'))

    """Create new file path for new experiment"""
    ID = datetime.now().strftime("%d%m%Y%H%M")
    newpath = mainpath + '/History'
    if not os.path.exists(newpath):
        os.makedirs(newpath)
    newpath = newpath + '/' + spec.values[0,0] + '_' + Controller + '_' +ID
    if os.path.exists(newpath):
        shutil.rmtree(newpath)
    os.makedirs(newpath)
    modelpath = newpath + '/model'
    if not os.path.exists(modelpath):
        os.makedirs(modelpath)

    """Create text file containing learning parameters for this experiment"""
            inputs_list = ['gamma = ' + str(gamma) + '\n',\
                           'epsilon_initial = ' + str(epsilon_initial) + '\n',\
                           'epsilon_min = ' + str(epsilon_min) + '\n',\
                           'epsilon_decay = ' + str(epsilon_decay) + '\n',\
                           'alpha_initial = ' + str(alpha_initial) + '\n',\
                           'alpha_decay = ' + str(alpha_decay) + '\n',\
                           'target_update = ' + str(target_update) + '\n',\
                           'lr_update = ' + str(lr_update) + '\n',\
                           'memory_size = ' + str(memory_size) + '\n',\
                           'batch_size = ' + str(batch_size) + '\n',\
                           'NN = ' + str(layers[0]) + ',' + str(layers[1]) + '\n',\
                           'Outcome:']
            file = open(newpath + '/info.txt', 'w')
            file.writelines(inputs_list)
            file.close()
            spec.to_csv(newpath + '/spec.csv', index=None, sep=',', mode='a')

            """Initialise neural network model"""
            device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
            policy_net = DQN(InputDim,layers).to(device)
```

```
            target_net = DQN(InputDim,layers).to(device)
            target_net.load_state_dict(policy_net.state_dict())
            target_net.eval()
            optimizer = optim.Adam(params = policy_net.parameters(), lr = alpha, )

        elif StartNewSim == False:
            """Get values and models from what was left off"""
            e_c = len(result)
            checkpoint = torch.load(modelpath + '/model_recent.h5')
            policy_net = DQN(InputDim,layers).to(device)
            target_net = DQN(InputDim,layers).to(device)
            policy_net.load_state_dict(checkpoint['state_dict'])
            optimizer = optim.Adam(params = policy_net.parameters(), lr = alpha)
            optimizer.load_state_dict(checkpoint['optimizer'])
            target_net.load_state_dict(policy_net.state_dict())

        """Create connection with Unity"""
        host, port = "127.0.0.1" , 25001 #Must be identical to the ones in Unity code
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.connect((host,port))

        """initialise classes"""
        transfer_data = transfer_data(sock, InputType,
                                      InputDim, DeptEstSpeed, truestate)
        ExperienceReplay = ExperienceReplay(Experience, batch_size,
                                            replay_epoch, gamma, target_update)
        Agent = Agent(Controller, action_space_size, InputDim)

        # ================================================================================
        # Begin the Training
        # ================================================================================
        for e in range(e_c,n_episodes):

            """Initialise episodic results"""
            reward_cumu = 0
            tstep = 0
            act = [0,0,0]
            loss_tmt = []
            positions_tmt = []

            """Initialise the simulation"""
            data_received = transfer_data.ReceiveData(image_old)
            transfer_data.SendData([0,1]) #Reset the environment

            """Get first set of state"""
            data, state, image, rem = transfer_data.ReceiveData(image_old)
            crash = data[0]

            while not crash and tstep < max_env_steps:
                """Choose and send action to Unity"""
                if InputType == "LiDAR":
                    action = Agent.choose_action_RL(
                            torch.tensor([state]), epsilon_initial, policy_net, image)
                else:
                    action = choose_action_HC(image, InputDim)
                transfer_data.SendData([action,0])

                """Get next state and other information from Unity"""
                data, next_state, image, rem = transfer_data.ReceiveData(image_old)
                crash = data[0]
                reward = Agent.get_reward(data)
```

```
181.
182.                 """Store experience in replay memory"""
183.                 memory.push(Experience(torch.tensor([state]), torch.tensor([action]),
184.                                 torch.tensor([next_state]), torch.tensor([reward]),
185.                                 torch.tensor([crash])), rem)
186.
187.                 """Perform experience replay & update target network by the schedule"""
188.                 ExperienceReplay.replay(memory, policy_net, target_net, optimizer,
189.                                         loss_tmt, total_tstep)
190.
191.                 """Trainsition to the next state"""
192.                 state = next_state
193.                 image_old = image   #Save previous image in case error occurs
194.                                     #during image dceoding in the next loop
195.
196.                 """Append or cumulate results"""
197.                 reward_cumu += reward
198.                 tstep += 1
199.                 act[action] += 1
200.                 total_tstep += 1
201.                 positions_tmt.append(data[2:4])
202.
203.             """Reset the environment"""
204.             transfer_data.SendData([0,1])
205.
206.             """Store episodic results in result tank"""
207.             loss_array += loss_tmt
208.             positions += positions_tmt
209.             result = result.append({'Cumulative Reward': reward_cumu.item(),\
210.                                 'Time Step': tstep,\
211.                                 'Crash': int(crash),\
212.                                 'Straight': act[0],\
213.                                 'Left Turn': act[1],\
214.                                 'Right Turn': act[2]}, ignore_index=True)
215.
216.             """Save results and model"""
217.             save_result(positions, loss_array, result, newpath)
218.             save_model(policy_net, optimizer, modelpath,
219.                        total_tstep, model_save_interval)
220.
221.             """Plot progress"""
222.             plot_progress(result, loss_array, crash, e, reward_cumu,
223.                           newpath, [sma_period_reward, sma_period_loss])
224.
225.             """Update learning rate"""
226.             update_learning_rate(alpha_initial, alpha_decay,
227.                                  total_tstep, lr_update, optimizer)
228.
229.             """Update epsilon"""
230.             epsilon = max(epsilon_min, epsilon_initial * ((1 - epsilon_decay) ** e))
231.
232.         sock.close()
```

Appendix E: Python Code with All Necessary Classes and Functions for Deep Q-Learning

```
from __future__ import print_function
import random
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

def update_learning_rate(alpha_initial, alpha_decay,
                         total_tstep, lr_update, optimizer):
    """Update learning rate"""
    alpha = alpha_initial * ((alpha_decay)**np.floor(total_tstep/lr_update))
    for param_group in optimizer.param_groups:
        param_group['lr'] = alpha

class Agent():
    def __init__(self, Controller, action_space_size, InputDim):
        self.Controller = Controller
        self.action_space_size = action_space_size
        self.InputDim = InputDim

    def get_reward(self, data):
        """Calculate reward at current time step"""
        distdiff = data[3]
        crash = bool(data[0])
        reward = np.sign(distdiff)*(1-crash) - crash
        return reward

    def choose_action_RL(self, state, epsilon, policy_net, image):
        """e-greedy strategy"""
        if np.random.random() <= epsilon:
            action = random.randrange(self.action_space_size)
        else:
            with torch.no_grad():
                predict = policy_net(state)
                action = predict.argmax(dim=1).item()
        return int(action)

class DQN(nn.Module):
    def __init__(self, InputDim,layers):
        super().__init__()
        """create nn layers"""
        self.fc1 = nn.Linear(in_features = np.prod(InputDim) + 3,
                             out_features = layers[0])
        self.fc2 = nn.Linear(in_features = layers[0], out_features = layers[1])
        self.out = nn.Linear(in_features = layers[1], out_features = 3)

    def forward(self, t):
        """feed forward"""
        t = F.relu(self.fc1(t))
        t = F.relu(self.fc2(t))
        t = self.out(t)
        return t

class ReplayMemory():
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.push_count = 0
```

```
59.
60.     def push(self, experience, rem):
61.         if rem:
62.             if len(self.memory) < self.capacity:
63.                 """add new experience"""
64.                 self.memory.append(experience)
65.             else:
66.                 """push the oldest one out if memory is full"""
67.                 self.memory[self.push_count % self.capacity] = experience
68.             self.push_count += 1
69.
70.     def sample(self, batch_size):
71.         """sample a batch of experience"""
72.         return random.sample(self.memory, batch_size)
73.
74.     def can_provide_sample(self, batch_size):
75.         """Check if there's enough experiences in memory to be sampled"""
76.         return len(self.memory) >= batch_size
77.
78. class ExperienceReplay():
79.
80.     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
81.
82.     def __init__(self, Experience, batch_size,
83.                  replay_epoch, gamma, target_update):
84.         self.Experience = Experience
85.         self.batch_size = batch_size
86.         self.replay_epoch = replay_epoch
87.         self.gamma = gamma
88.         self.target_update = target_update
89.
90.     def extract_tensors(self, experiences):
91.         batch = self.Experience(*zip(*experiences))
92.         t1 = torch.cat(batch.state)
93.         t2 = torch.cat(batch.action)
94.         t3 = torch.cat(batch.reward)
95.         t4 = torch.cat(batch.next_state)
96.         t5 = torch.cat(batch.crash)
97.         return (t1,t2,t3,t4,t5)
98.
99.             @staticmethod
100.            def Q_current(policy_net, states, actions):
101.                """Get Q(s_t,a_t)"""
102.                return policy_net(states).gather(dim = 1, index=actions.unsqueeze(-1))
103.
104.            @staticmethod
105.            def Q_next(target_net, next_states, crashes):
106.                """Get max(Q(s_t+1,a_t+1))"""
107.                final_state_locations = crashes.eq(1).type(torch.bool)
108.                non_final_state_locations = (final_state_locations == False)
109.                non_final_state = next_states[non_final_state_locations]
110.                batch_size = next_states.shape[0]
111.                values = torch.zeros(batch_size).to(ExperienceReplay.device)
112.                values[non_final_state_locations] = \
113.                                    target_net(non_final_state).max(dim=1)[0].detach()
114.                return values
115.
116.            def replay(self, memory, policy_net, target_net,
117.                           optimizer, loss_tmt, total_tstep):
118.                if memory.can_provide_sample(self.batch_size):
119.
```

```
                    """Update target network if it's time for update"""
                    if total_tstep % self.target_update == 0:
                        target_net.load_state_dict(policy_net.state_dict())
                        print("target network updated")

                    """Perform experience replay"""
                    for epoch in range(self.replay_epoch):
                        experiences = memory.sample(self.batch_size)#Sample experiences
                        states,actions,rewards,next_states,crashes = \
                                    ExperienceReplay.extract_tensors(self, experiences)
                        current_q_values = ExperienceReplay.Q_current(
                                                    policy_net, states, actions)
                        next_q_values = ExperienceReplay.Q_next(target_net,
                                                            next_states, crashes)
                        target_q_values = (next_q_values * self.gamma) + rewards

                        """Get Huber loss and perform gradient descend"""
                        loss = F.smooth_l1_loss(current_q_values,
                                                    target_q_values.unsqueeze(1))
                        optimizer.zero_grad()
                        loss.backward()
                        optimizer.step()
                        loss_tmt.append(loss.item())
```

Appendix F: Python Code with Class Required for Python – Unity Connection

```
from __future__ import print_function
import numpy as np
from base64 import b64decode
import cv2
import time

class transfer_data():
    """Class for transfering data to and from Unity"""
    def __init__(self, sock, InputType, InputDim, DeptEstSpeed, truestate):
        self.sock = sock
        self.InputType = InputType
        self.InputDim = InputDim
        self.DeptEstSpeed = DeptEstSpeed
        self.truestate = truestate

    def ConvertDataReceived(self, data_received):
        """Convert string of data from Unity to processable list"""
        split_list = data_received.split(" ")
        if self.InputType == "Visual":
            return list(map(float, split_list[:-1])) + [split_list[-1]]
        elif self.InputType == "LiDAR":
            return list(map(float, split_list))

    def decode_image(self, base64_img):
        """Decode base64 from Unity to image"""
        a = b64decode(base64_img)
        np_arr = np.frombuffer(a, np.uint8)
        img_np = cv2.imdecode(np_arr, cv2.IMREAD_COLOR)
        image = np.round(transfer_data.rgb2gray(img_np))
        image = cv2.resize(image,(self.InputDim[1],self.InputDim[0]),
                           interpolation = cv2.INTER_AREA)

        """Time delay to simulate slow computational time
        of depth estimation algorithm"""
        time.sleep(max(0,(self.DeptEstSpeed)))
        return image/255 #Standardise the pixel value into 0 - 1

    def rgb2gray(rgb):
        """Turn RGB type of image into gray-scale image"""
        return np.dot(rgb[...,:3], [0.2989, 0.5870, 0.1140])

    def ReceiveData(self, image_old):
        """Recieve raw data from Unity and process into state"""
        data_received = self.sock.recv(1024).decode("utf-8")
        data = transfer_data.ConvertDataReceived(self, data_received)
        img_start = len(data) - np.prod(self.InputDim)
        if self.InputType == "Visual":
            """
            If data contains image, try to decode it.
            If error occurs, continue by using previous image and do not store
            this experience in the replay memory. The error does not occur
            often.
            """
            try:
                image = transfer_data.decode_image(self, data[-1])
                rem = True
            except:
                print('error')
```

```
                image = self.image_old
                rem = False

        elif self.InputType == "LiDAR":
            image = np.array(data[img_start:])
            image = image.reshape([self.InputDim[0],self.InputDim[1]])
            rem = True

        state = data[(img_start - self.truestate):img_start] + \
                list(image.flatten())
        return data, state, image, rem

    def SendData(self, data):
        """
        Send action and command to whether to reset the environment to Unity
        """
        data_send = str(data)
        self.sock.sendall(data_send.encode("utf-8"))
```

Appendix G: Python Code with Other Important Functions

```
from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import torch

def read_spec(SpecPath,NumPixelHor,NumPixelVer):
    """Read specification from Unity"""
    spec = pd.read_csv(SpecPath, sep=" ", header=None)
    if spec.values[0,0] == 'Visual':
        spec_header = ['InputType','NumPixelHor', 'NumPixelVer', \
                       'CamFieldOfView', 'CamFarPlane',\
                       'thrust', 'TurnSpeed', 'maxRoll', 'rollSpeed']
        spec.columns = spec_header
        spec['NumPixelHor'] = NumPixelHor
        spec['NumPixelVer'] = NumPixelVer

    elif spec.values[0,0] == 'LiDAR':
        spec_header = ['InputType', 'NumSensorsHor', 'NumSensorsVer', \
                       'SensorAngleIncHor', 'SensorAngleIncVer', \
                       'sensorLength', 'thrust', 'TurnSpeed', \
                       'maxRoll', 'rollSpeed']

        spec.columns = spec_header

    return spec

def get_moving_average(period, values):
    """Get simple moving average"""
    values = torch.tensor(values, dtype=torch.float)
    if len(values) >= period:
        moving_avg = values.unfold(dimension=0, size=period, step=1) \
            .mean(dim=1).flatten(start_dim=0)
        moving_avg = torch.cat((torch.zeros(period-1), moving_avg))
        return moving_avg.numpy()
    else:
        moving_avg = torch.zeros(len(values))
        return moving_avg.numpy()

def plot_progress(result, loss_array, crash, e, reward_cumu, newpath, arg):
    """Get simaple moving average of reward and loss"""
    reward_sma = get_moving_average(arg[0], result['Cumulative Reward'])
    loss_sma = get_moving_average(arg[1], loss_array)

    """assign different marker color depending on who the episode ends"""
    if crash == True:
        marker = 'ro'
    else:
        marker = 'go'

    """Plot reward and loss graphs"""
    if len(loss_array) > 0:
        plt.subplot(2,1,1)
        plt.plot(result['Cumulative Reward'], linewidth=0.3)
        plt.plot(reward_sma)
        plt.plot(e,reward_cumu,marker)
        plt.subplot(2,1,2)
```

```
        plt.plot(loss_array, linewidth=0.1)
        plt.plot(loss_sma)
        plt.ylim(0, max(np.array(loss_sma).flatten())*1.1)
        plt.savefig(newpath + '/result.png')
        plt.pause(0.0001)

def save_result(positions, loss_array, result, newpath):
    """Save result, loss, and flight path to files"""
    positionsdf = pd.DataFrame.from_records(positions, columns = ['x','z'])
    lossdf = pd.DataFrame(loss_array, columns = ['loss'])
    result.to_csv(newpath + '/result.csv', index = False, header=True)
    positionsdf.to_csv(newpath + '/positions.csv',index = False, header=True)
    lossdf.to_csv(newpath + '/loss.csv',index = False, header=True)

def save_model(policy_net, optimizer, modelpath,
               total_tstep, model_save_interval):
    """Save the most recet neural network model"""
    policy_net_state = {
            'state_dict': policy_net.state_dict(),
            'optimizer': optimizer.state_dict(),
            }
    torch.save(policy_net_state, modelpath + '/model_recent.h5')

    """Save neural network model at a specific interval of training steps"""
    s = np.floor(total_tstep / model_save_interval)
    torch.save(policy_net.state_dict(), modelpath + '/model_' +\
               str(int(s+1)) +'.h5')

def choose_action_HC(image, InputDim):
        """Braitenberg controller"""
        ray = image.flatten()
        middle_index = int((len(ray)/InputDim[0])/2)

        ray = np.array(ray)
        ray = ray.reshape([InputDim[0],InputDim[1]])
        ray = ray.tolist()

        if (len(ray) % 2) != 0:
            for row in ray:
                del row[middle_index]

        ray = np.array(ray)
               ray = np.mean(ray,0)

               N = [0,0]
               for i in range(len(ray)):
                   N_index = i >= middle_index
                   N[N_index] += 1/(ray[i]**2)
               action = (N.index(min(N)) + 1) * np.sign(1 - np.average(ray))

               return int(action)
```

Appendix H: C# Code for Depth Image in Unity

```
using UnityEngine;

[ExecuteInEditMode]
public class RenderDepth : MonoBehaviour
{
    /*Attach this code to any camera in Unity to turn the display
     * from that camera into depth image */
    public bool grab;
    [Range(0f, 3f)]
    public float depthLevel = 0.5f;

    private Shader _shader;
    private Shader shader
    {
        get { return _shader != null ? _shader : (_shader = Shader.Find("Custom/RenderDepth2")); }
    }

    private Material _material;
    private Material material
    {
        get
        {
            if (_material == null)
            {
                _material = new Material(shader);
                _material.hideFlags = HideFlags.HideAndDontSave;
            }
            return _material;
        }
    }

    private void Start()
    {
        if (!SystemInfo.supportsImageEffects)
        {
            print("System doesn't support image effects");
            enabled = false;
            return;
        }
        if (shader == null || !shader.isSupported)
        {
            enabled = false;
            print("Shader " + shader.name + " is not supported");
            return;
        }

        var mainCam = GetComponent<Camera>();
        mainCam.depthTextureMode = DepthTextureMode.Depth;
    }

    private void OnDisable()
    {
        if (_material != null)
            DestroyImmediate(_material);
    }

    private void OnRenderImage(RenderTexture src, RenderTexture dest)
```

```
    {
        if (shader != null)
        {
            material.SetFloat("_DepthLevel", depthLevel);
            Graphics.Blit(src, dest, material);
        }
        else
        {
            Graphics.Blit(src, dest);
        }
    }
}
```

Appendix I: Python Code for Testing Phase

```
from __future__ import print_function
import socket
import numpy as np
import pandas as pd
import os, glob
from pathlib import Path
import torch
import re
from transfer_data import transfer_data
from DQL import Agent, DQN
from OtherFunc import choose_action_HC

def numericalSort(value):
    """Function for sorting file name by number"""
    numbers = re.compile(r'(\d+)')
    parts = numbers.split(value)
    parts[1::2] = map(int, parts[1::2])
    return parts

"""Choose the type of controller.
'RL' => Reinforcement Learning controller,
'HC' => Hand controller (Breaitenberg controller) """
Controller = 'RL'
# ================================================================================
# Testing Parameters
# ================================================================================
Folder_Name = '**LiDAR_RL_200420201534' #File name of the experiment folder
max_env_steps = 1000    #Max episode steps
n_episodes = 100        #Max number of episode for each test
DeptEstSpeed = 0.1
epsilon = 0             #Set to 0 for testing
action_space_size = 3
truestate = 3


# ================================================================================
# Initialsing
# ================================================================================
"""Get paths"""
mainpath = str(Path(__file__).parents[1])
path = mainpath + '/History/' + Folder_Name
```

```
modelpath = path + '/model'

"""Read info and spec of the experiment"""
info = pd.read_csv(path + '/info.txt', sep=" ", header = None)
NN = info.loc[info[0] == 'NN'].values[0,2].split(',')
layers = [int(NN[0]), int(NN[1])]
spec = pd.read_csv(path + '/spec.csv', sep=",")
InputDim = [spec.values[0,2], spec.values[0,1]]
InputType = spec.values[0,0]

"""Initialise parameters, result tanks, and replay memory"""
epsilon = 0

total_tstep = 0
image_old = np.ones(InputDim)

"""Create new file path in current experiment's folder for test result"""
testpath = path + '/test'
if not os.path.exists(testpath):
    os.makedirs(testpath)

"""Initialise neural network model"""
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
policy_net = DQN(InputDim,layers).to(device)

"""Create connection with Unity"""
host, port = "127.0.0.1" , 25001 #Must be identical to the ones in Unity code
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host,port))

"""initialise classes"""
transfer_data = transfer_data(sock, InputType,
                              InputDim, DeptEstSpeed, truestate)
Agent = Agent(Controller, action_space_size, InputDim)

# ================================================================================
# Begin the Testing
# ================================================================================
for filename in sorted(glob.glob(os.path.join(modelpath, '*.h5')),
                       key=numericalSort):
    """Iterate the models in the model folder to test one by one"""
    file = filename.split("/")[-1]
    file = file[:-3]
    print(file)
    policy_net = DQN(InputDim,layers).to(device)
    policy_net.load_state_dict(torch.load(filename))
    policy_net.eval()

    """Initialise result tanks for each model"""
    positions = []
    result = pd.DataFrame(columns=['Cumulative Reward', 'Time Step',
                                   'Crash','Straight', 'Left Turn',
                                   'Right Turn'])
    for e in range(0,n_episodes):
        print(e)
        """Initialise episodic results"""
        reward_cumu = 0
                tstep = 0
                act = [0,0,0]
                positions_tmt = []

```

```
                """Initialise the simulation"""
                data_received = transfer_data.ReceiveData(image_old)
                transfer_data.SendData([0,1]) #Reset the environment

                """Get first set of state"""
                data, state, image, rem = transfer_data.ReceiveData(image_old)
                crash = data[0]

                while not crash and tstep < max_env_steps:
                    """Choose and send action to Unity"""
                    if InputType == "LiDAR":
                        action = Agent.choose_action_RL(
                                torch.tensor([state]), epsilon,
                                policy_net, image)
                    else:
                        action = choose_action_HC(image, InputDim)
                    transfer_data.SendData([action,0])

                    """Get next state and other information from Unity"""
                    data, next_state, image, rem = transfer_data.ReceiveData(image_old)
                    crash = data[0]
                    reward = Agent.get_reward(data)

                    """Trainsition to the next state"""
                    state = next_state
                    image_old = image   #Save previous image in case error occurs
                                        #during image dceoding in the next loop

                    """Append or cumulate results"""
                    reward_cumu += reward
                    tstep += 1
                    act[action] += 1
                    total_tstep += 1
                    positions_tmt.append(data[2:4])

                """Reset the environment"""
                transfer_data.SendData([0,1])

                """Store episodic results in result tank"""
                positions += positions_tmt
                result = result.append({'Cumulative Reward': reward_cumu.item(),\
                                        'Time Step': tstep,\
                                        'Crash': int(crash),\
                                        'Straight': act[0],\
                                        'Left Turn': act[1],\
                                        'Right Turn': act[2]}, ignore_index=True)

                """Save results"""
                positionsdf = pd.DataFrame.from_records(positions, columns = ['x','z'])
                result.to_csv(testpath + '/result_test_' + \
                              file + '.csv', index = False, header=True)
                positionsdf.to_csv(testpath + '/positions_test_' + \
                                   file + '.csv',index = False, header=True)

        sock.close()
```