PERFORMANCE COMPARISON OF STEREO AND RGB SENSORS FOR UAV COLLISION AVOIDANCE

by

ANDRII HLYVKO

A thesis submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Master of Science

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Waheed U. Bajwa,

Francisco Javier Diez

and approved by

_____

_____

_____

New Brunswick, New Jersey

May, 2020

ABSTRACT OF THE THESIS

Performance Comparison of Stereo and RGB Sensors for UAV Collision Avoidance

By Andrii Hlyvko

Thesis Director:

Waheed U. Bajwa,

Francisco Javier Diez

Over the past years there have been many different approaches that have shown significant progress towards solving the challenging problem of collision avoidance for UAVs. These approaches range from SLAM to machine learning. Machine learning approaches are promising because the model learns to perform a complex task using training data instead of someone having to develop a complex and task-specific controller. In machine learning approaches we can choose whether to train in simulation or on real data. Collecting real-world UAV collision data is very time consuming and can result in a damaged UAV. On the other hand, synthetic data and real-world data come from different distributions so training using synthetic data introduces a gap between the learned distribution and the actual one, which can result in poor performance. Even though this distribution gap exists, training in simulation saves time and cost of training,

making these approaches the focus of this study. Usually due to UAV size and weight constrains we can only choose one sensor for performing obstacle avoidance. Therefore, we need to select a sensor that would give the best performance when a model is trained in simulation. Many different sensors can be chosen for performing UAV collision avoidance, such as RGB, stereo, LIDAR, among others. Even if a sensor can be accurately simulated, the data it produces might not contain sufficient information for performing collision avoidance well. For instance, a sonar can be very accurately simulated but it does not contain sufficient information about the state of the environment required to avoid complex shapes. The hypothesis is that a model that is trained entirely in simulation is going to perform differently in the real-world depending on what simulated sensor was used for training. In this thesis we train using different simulated sensors to demonstrate the hypothesis that real-world performance with a model trained entirely in simulation improves when an appropriate sensor is chosen for training. Even though we cannot confirm that one sensor outperforms others for every single machine learning approach, we obtain experimental data for a few methods to support our claim. RGB cameras are one of the simplest and most widely used sensors for drone sense and avoid. On the other hand, stereo sensors are bulky and require high computing power to produce real-time results useful for collision avoidance in drones. This has changed with recent advances in stereo sensors and computing, which has made it possible to use them in micro-aerial vehicles for real-time operation [1]. Therefore, these two sensors are the most suitable for our study. In this thesis we compare how much performance do we gain, if any, by training on a simulated stereo system instead of a simulated RGB camera for obstacle avoidance using machine learning approaches.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Francisco Javier Diez, for his guidance and support during my graduate studies. He encouraged me to work hard even when I wanted to give up. I would not be where I am without his help. I would also like to thank my co-advisor, Professor Waheed U. Bajwa, for his help in guiding this research. His feedback was extremely helpful in guiding this research. When I was stuck he helped me see the problem in a different light. In addition, I would like to thank my family. Thanks to their support I was able to completely focus on my research. A special thanks to Aristides Costeas for his deep insights in quadcopter design and for his support when I was feeling down.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

Collision avoidance remains an active area of research in robotics as it is a basic requirement for robot autonomy and the problem remains needs to be solved. In addition, because of the size and weight limitations of UAVs we can usually only select one small sensor to use for collision avoidance. The constraints are even tighter for micro UAVs which usually weigh under 50g. The question is whether all sensors perform collision avoidance equally well if we keep the collision avoidance algorithm constant. If one sensor outperforms all others, then it would be the one that should be used to perform collision avoidance with.

In order to select the best modality for collision avoidance we need to select a few collision avoidance approaches to perform the comparison. Of course, not every sensor can be used with every kind of algorithm. The approaches we select for our comparison apply to all sensors we compare. Although many approaches can be chosen, we focus on approaches that rely on training a model on synthetic data. Many different approaches to obstacle avoidance have been tried in the past. The most successful approaches used SLAM based techniques to navigate the environment and avoid obstacles. This two-step approach can be computationally costly and requires the UAV to map the environment. Although SLAM approaches showed good performance, they are not the focus of this study as we only want to compare methods that rely training on simulated data. In addition, SLAM is too computationally costly to perform collision avoidance for a very fast flying micro UAV.

Other methods that can be used to solve the problem of obstacle avoidance are supervised learning and imitation learning. Imitation learning suffers from very concentrated data sampling as only either the failure cases or success cases are available. For example, in [2] the drone could not recover if it deviated too much from the center of the forest trail. However, if we utilize imitation learning from simulated data we can avoid this problem. Crashing in simulation is not a problem so negative examples can be obtained much easier. Sensor comparison using imitation learning with a simulated data set could be a promising direction for sensor performance comparison but was not performed due to time constraints.

In supervised learning, data can be hard to collect and label, especially collision data due to high risk of damaged hardware. However, obtaining collision data is easy in simulation. Therefore, in this thesis we also compare sensor performance on a supervised collision classifier. Using reinforcement learning for obstacle avoidance is a promising approach as the agent learns the optimal strategy by trial and error. One reinforcement learning method we can use is deep-Q learning. Deep-Q learning for the purpose of obstacle avoidance for UAVs presents many challenges. Deep-Q learning is very data intensive and takes a long time to train. However, as demonstrated in [3], [4] deep-Q learning is a very powerful method that can achieve human level performance at complex tasks. Thus, we have selected deep-Q learning as another method for comparing sensor performance. We use Monte-Carlo tree search in order to obtain an estimate of the long-horizon rewards. For each action the agent performs an N-step rollout by following an epsilon-greedy policy. This process can be seen in Figure 1.1.

Training is simulation can help us reduce the risks of real-world training since we can generate realistic sensor data and emulate agent's actions faster than real-time. This would cut down the training time tremendously. The problem with training in simulation using synthetic data is the large gap between the synthetic data distribution and distribution that comes from the real-world. It is far more

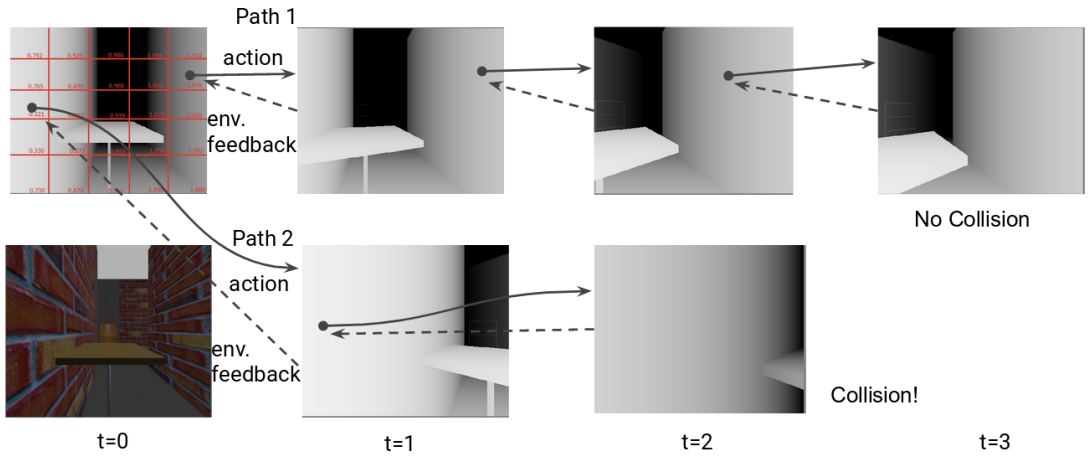

Figure 1.1: Training in a simulated hallway with obstacles using depth data. At time zero we show the current state of the drone inside Gazebo. The state image is partitioned into discrete actions of pitch and yaw. A Monte-Carlo policy evaluation method is used to predict collision probabilities of each action. A rollout of each action is performed until a collision occurs or the maximum rollout depth is reached.

difficult to generate realistic textures for an RGB image than to simulate a depth image. For example, in [5] synthetic RGB images were used for training, but generating realistic textures presents great difficulty and therefore the learned controller could not generalize that well to real-world images. A question that arises is if this gap is sensor dependent or not. If it is, then the training with synthetic data could be more readily applicable in real-world conditions by selecting a modality with low simulation to real-world distribution gap. If so, then using a sensor that performs best for a given approach would help increase the performance of models that were trained entirely in simulations. It should be noted that although beyond the scope of this work, a further improvement on this could be transfer learning on a small real-world data set.

We propose to test the hypothesis that the distribution gap between certain simulated sensors and their real-world counterparts is greater than it is for others. We would then want to show that a sensor with a smaller distribution gap outperforms a similar sensor with a larger distribution gap. In order to show this,

we must select comparable modalities that are sufficient enough for the task of obstacle avoidance. Two such sensors are stereo camera and an RGB camera. Our hypothesis is that an agent that was trained in simulation using stereo data will outperform an agent that was trained using RGB data regardless of the machine learning approach used. Although, it is beyond the scope of this work to verify that this holds for every approach, we plan to test the hypothesis on a few selected approaches thereby obtaining experimental data to support our claim.

Chapter 2
Related Work

In this chapter we contrast different approaches in computer vision and machine learning relevant to our work. The main approaches to robotic navigation considered are traditional computer vision approaches, supervised learning methods, and reinforcement learning approaches.

2.1 Computer Vision Approaches

In traditional computer vision approaches for obstacle avoidance one would first estimate the 3D geometry from motion and then make a decision based on the inferred geometry. We can estimate depth using an RGB camera from motion by using multiple frames. Despite recent progress that was demonstrated in [6], geometry estimation from motion still remains a challenge. In SLAM approaches [7], [8], the agent would have to perform computationally intensive geometry inference and map building, which is not suited for real-time UAV flight. Another approach would be to use stereo methods as in [9], [10]. In earlier years the size of a stereo system was a limiting factor for UAVs. However, with recent developments in stereo systems this is no longer a major concern. In this thesis we demonstrate the use of a stereo system in a UAV without sacrificing flight time.

2.2 Imitation Learning

Imitation learning is a supervised learning approach where a model is trained using labeled data generated by an expert. The expert usually performs the task without making any mistakes or having any failures at performing the task. For example, to collect data to perform self-driving using imitation learning one would have to get the data of real-world driving [11]. Real-world driving data examples of crashes are very scarce. Even examples where the real-world driver deviates from too much from the center of the road are rare. So the training model in never presented with examples of what to do in the case of where it deviates from expert behavior. This makes imitation learning samples very concentrated. In this case the data mostly represents expert behavior without any deviations.

There has been recent success with using imitation learning for autonomy in robotics [2], [11], [12]. The self-driving car in [11] was trained to follow roads using a deep network trained on RGB data. In [2] an imitation learning approach was used to teach a quadcopter to follow forest trails. In both examples the limitation of concentrated data samples was addressed by using additional cameras facing in different directions. This approach is successful with small error corrections but does not work well to recover from unknown bad states. The agent is not presented with cases that deviate from the correct path by larger amounts. Since the data samples are not independent and identically distributed (i.i.d.) and the learned distribution differs from the actual one, this approach results in accumulating error over time, introducing difficulties in recovering from unknown states. Imitation learning in simulation is promising because collision data can be obtained more easily in simulation resulting in a more uniform sampling. Comparing the sensor performance using imitation learning trained on synthetic data remains future work.

2.3 Deep Reinforcement Learning

Deep reinforcement learning has been successfully deployed in many robotics scenarios [5], [13], [14], [15], [16], [17]. In [18] supervised training in simulation was used successfully to estimate depth from RGB, which was later used in a reinforcement learning system to learn a controller for obstacle avoidance. Deep reinforcement learning is suitable for complex tasks but requires training on large data sets. In [5] a deep reinforcement learning approach using synthetic RGB data was taken to learn a collision avoidance controller. The problem with this approach is the large gap between synthetically generated RGB images and real-world images. In [13] a collision avoidance controller was trained using 11500 real-world RGB image dataset of crashes. The issue is that this type of training does not scale for complex environments. To be robust in a complex environment with many obstacles, the network has to be trained on a large amount of data that is impractical to collect. In our approach we address the issue of training the model in the real-world by training with synthetic data that has a small simulated to real-world distribution gap.

Chapter 3
Background

In this chapter our goal is to give the reader some background in the area of reinforcement learning. In reinforcement learning and agent learns to map inputs to actions in order to maximize a reward signal. The agent is the learner and the decision maker in reinforcement learning. The environment in reinforcement learning is what the agent interacts with. It provides the agent with feedback of how well the agent is performing at the task. In this chapter we will go over different types of machine learning and compare them. We will also go over different components of a reinforcement learning system. Then we will introduce the finite Markov decision process which is the foundation for the theory behind reinforcement learning. This chapter is concluded with the introduction of deep-Q learning. The information in this chapter was obtained from [19]. Readers are encouraged to check [19] for an in-depth treatment of reinforcement learning.

3.1 Types of Machine Learning

The main three types of machine learning are supervised, unsupervised, and reinforcement learning. In supervised learning the system is provided with a labeled set of training data and the goal is to generalize so that the system can provide correct labels for data it has never seen before. The problem with supervised learning is that usually it is hard to obtain training examples of all possible situations. Sometimes in supervised learning we can have more examples of one class since some cases might be hard to obtain. In this situation it is hard for the

agent to generalize well to cases it has not already seen. Reinforcement learning tries to solve this problem by letting the agent explore and sample states on its own. Also, supervised learning does not address the problem of maximizing a reward signal. That is, it does not take into account future rewards. Making correct choices requires taking into account indirect delayed consequences of actions.

Another type of machine learning is called unsupervised learning. It is the process of uncovering structure from unlabeled data. An example of unsupervised learning is separating data into clusters. A third type of machine learning is called reinforcement learning. The agent is not told what to do explicitly but must discover optimal decisions by trial and error. In reinforcement learning the agent must be able to sense its environment and to make decisions that alter the state of the environment. The current decisions that the agent makes will not only affect the immediate reward but also all rewards the agent will get in the future. One of the challenges in reinforcement learning is the tradeoff between exploration and exploitation. In order to maximize the reward signal the agent must select the best possible actions it has seen, but in order to find those actions it needs to explore the actions it never took before. On a stochastic task an action must be experienced many times in order to obtain an accurate estimate of its expected reward. The tradeoff between exploration and exploitation has not yet been solved in reinforcement learning.

3.2 Components of a Reinforcement Learning System

The main elements of reinforcement learning system are a policy, reward signal, a value function, and a model of the environment. The model of the environment is optional. The policy defines how the agent acts at a given time. The policy is a mapping between states of the environment to actions when the agent is in those states. A policy π defines a probability distribution between actions in a

state: $\pi(a|s) = Pr(A_t = a|S_t = s)$. A reward signal defines the goal of the reinforcement learning problem. During each time step the environment sends a numerical reward to the agent when an action is taken. Based on the received reward the agent would then alter its policy. A value function determines the total amount of reward the agent expects to receive in the future starting from that state. For example, a state might have a low reward but a high value because that state is followed by states with high reward. The agent makes decisions based on values and not on rewards because the rewards are immediate, and values are long term. Rewards are given directly by the environment, but values must be estimated. Therefore, one of the main problems of reinforcement learning is efficiently estimating values. A model of the environment predicts the next state and the current reward given the current state and action. Models are used for planning. The agent decides on an action by considering future scenarios before they are actually experienced. These reinforcement learning methods are called model based. On the other hand, agent's that learn by trial and error are considered model-free methods.

3.3 Multi-Armed Bandit Problem

In reinforcement learning the agent evaluates different actions and explicitly searches for the best behavior. This is especially helpful for nonstationary actions. In a state the expected reward of an action can shift in the nonstationary case. Then the agent needs to constantly adapt and readjust its policy. First, we consider a simplified case where everything is stationary and associative. Associativity means that there is only one state of the environment. This situation fits into the k-armed bandit problem formulation. In this situation the agent is presented with k different actions. The agent interacts with the environment at discrete time steps t, selects an action a and receives a reward R_t. Since we have

only one state of the environment all the agent has to do is find out which action has the highest expected value and keep choosing that action. The value of an action a at time t is $q_*(a) = E[R_t|A_t = a]$. Since in the beginning the agent does not know the values of each action it has to estimate them. We denote the estimated values as $Q_t(a)$. Greedy actions are actions with the highest estimated value. Exploitation is when the agent selects an action with the highest value. Exploration is when the agent selects an action that doesn't have the highest estimated value. In the stationary k-armed bandit case we can estimate the value of each action as a sample average $Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i * 1_{A_i=a}}{\sum_{i=1}^{t-1} 1_{A_i=a}}$. In this formula $1_{predicate}$ is 1 when the predicate is true and 0 otherwise. When the denominator is 0 we define $Q_t(a)$ as zero. In practice it is better to have an incremental implementation for value estimation. To simplify the notation denote R_i as the reward recieved after the ith selection of that action and Q_n denote the estimate of its action value after that action has been selected $n-1$ times. Then the nth action value estimate is: $Q_n = \frac{R_1+R_2+...+R_{n-1}}{n-1}$. Now we can write an incremental estimation of the value as:

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i = \frac{1}{n}(R_n + \sum_{i=1}^{n-1} R_i) = \frac{1}{n}(R_n + \frac{n-1}{n-1}\sum_{i=1}^{n-1} R_i) = \frac{1}{n}(R_n + (n-1)Q_n) = Q_n + \frac{1}{n}(R_n - Q_n).$$

In the nonstationary case we have the values of actions shifting over time so it does not make sense to consider past rewards equally important as more recent rewards. One way to fix this is to give more weight to recent rewards using a step-size parameter α, $Q_{n+1} = Q_n + \alpha(R_n - Q_n) = (1-\alpha)^n Q_1 + \sum_{i=0}^{n} \alpha(1-\alpha)^{n-i} R_i$. The weights decay exponentially according to the exponent $1-\alpha$. In general, the multi armed bandit algorithm can be described as:

Algorithm 1: Multi-armed bandit algorithm

Result: Q(a) converges to the true value for each action a.

for *a=1, a <= k, a++* **do**

Q(a)=0;

N(a)=0;

end

while *true* **do**

$$A = \begin{cases} argmax_a Q(a) & \text{with probability } 1-\epsilon \\ \text{random action} & \text{with probability } \epsilon. \end{cases}$$

R=bandit(A);

N(A) = N(A) + 1;

Q(A) = Q(A) + $\alpha(R - Q(A))$;

end

3.4 Finite Markov Decision Process

The finite Markov decision process (MDP) is a mathematically idealized form of the reinforcement learning problem. Instead of the simplified k-armed bandits case we consider a nonassociative situation, that is, tasks in which there is no need to associate different actions with different situations. The agent now faces different situations instead of one. Instead of estimating $q_*(a)$ we estimate $q_*(s, a)$, the value of each action in each state. Or we can estimate $v_*(s)$, the value of a state. At each step the agent gets a state of the environment $S_t \in \mathcal{S}$. After the agent makes a decision $A_t \in \mathcal{A}$ at the next time step the agent will receive a reward $R_{t+1} \in \mathcal{R} \subset \Re$ and will be in state S_{t+1}. This interaction can be seen in Figure 3.1. So the value that we are trying to estimate now is $q_*(a, s)$ and it is dependent on the state that the environment is in. In the case of finite MDP the set $(\mathcal{R}, \mathcal{S}, \mathcal{A})$ is finite and the random variables R_t and S_t have well

defined discrete probability distributions that depend on the preceding state and action $p(s', r|s, a) \doteq Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$. This probability distribution completely defines the dynamic of the Markov decision process. It also follows the Markov property since the probability is independent of any past states or actions taken. From the dynamics function we can compute the state-transitions probabilities $p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a)$. We can also compute the expected rewards for each state-action pair as $r(s, a) = E[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a)$.

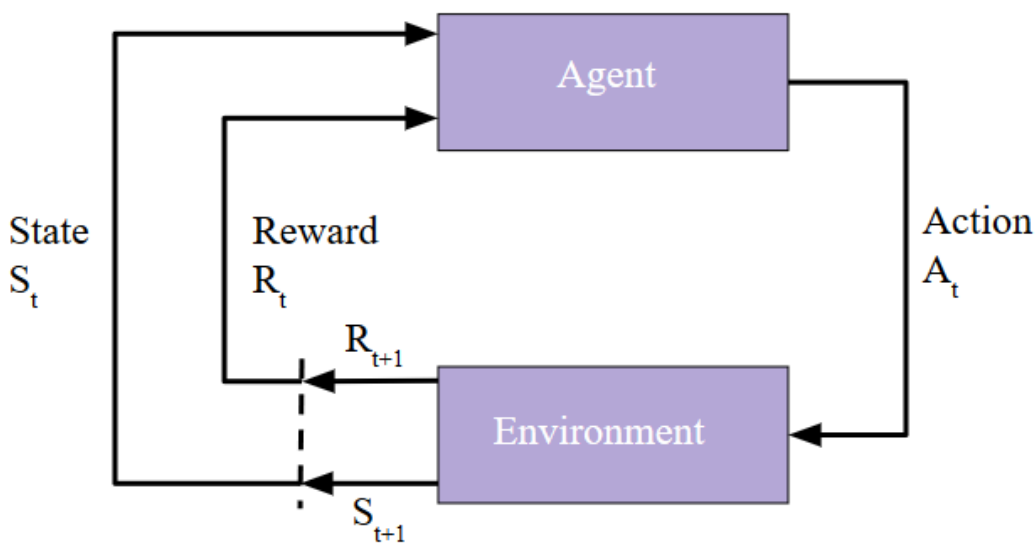

Figure 3.1: The agent-environment interaction in a Markov decision process
.

The goal of the agent is to maximize the expected return, which is the sum of all of the rewards the agent receives. We can write the total return as $G_t = R_{t+1} + R_{t+2} + ... + R_T$. Here T is the final time-step. Sometimes we can break drown the agent-environment interactions into subsequences, which we call episodes. An episode ends in a terminal state from which we reset the agent to a starting state. An example of an episode could be winning a chess game or flying a drone until a collision occurs. On the other hand, some tasks do not break up

naturally into episodes, but are continuously ongoing. In these cases we have $T = \infty$. In reality we want to maximize the discounted expected return $G_t = R_{t+1} + \gamma * R_{t+2} + \gamma^2 * R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. The parameter γ is called the discount rate and determines how much weight the agent places on recent rewards. The infinite sum has a finite value if the reward sequence is bounded and $\gamma < 1$. If $\gamma = 0$ the agent is only concerned with maximizing immediate rewards. As γ approaches 1, the agent becomes more far-sighted. The episodic case and the continuing case can be unified into one framework by introducing a special absorbing case that terminates an episode. This state can only transition to itself and only gives zero rewards. An example of an episode with $T = 3$ can be seen in Figure 3.2. If $T < \infty$ we can allow $\gamma = 1$ since the sum terminates.

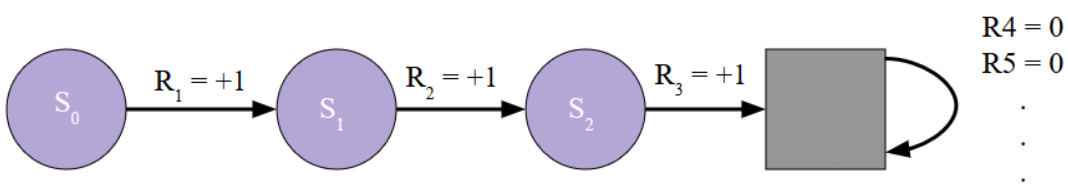

Figure 3.2: Absorbing State of a MDP

.

The state-value function estimates how good it is for the agent to be in a certain state and the action-value function estimates how good is an action in a certain state. These functions are defined with respect to a particular way of acting, called a policy. A policy is a mapping from states to probabilities of selecting each possible action. If the agent follows a policy π, then $\pi(a|s)$ is the probability that $A_t = a$ if $S_t = s$. The value of state s under the policy π is the expected return when starting in s and following π thereafter:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s], \text{ for all } s \in \mathcal{S}.$$

The value of a terminal state is always zero. $v_\pi(s)$ is called the state-value function under policy π. We can also define the value of an action a in state s under a

policy π as the expected return starting from s, taking action a, and following π thereafter:

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi[G_t|S_t = s, A_t = a] =$$
$$E_\pi[\textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a].$$

This function is called the action-value function. Both the state-value and the action-value functions can be estimated from the agent's experience. The agent can keep an average of the returns that follow that state for each state. This way we can estimate the value of each state. If there are too many states then this may be impractical, so it would be better to have the state-value function as a parameterized function and estimate its parameters.

The state-value function satisfies the recursive relationship:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] =$$
$$\textstyle\sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma E_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s]] =$$
$$\textstyle\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')].$$

This equation is known as the Bellman equation for the state-value function and it shows the relationship between the value of a state and the values of its successor states.

To solve a reinforcement learning task we need to find a policy that achieves a lot of reward over the long run. A policy π is defined to be better than a policy π' if its expected return is greater or equal to that of π' for all states. There will always be at least one policy that is better than all others and it is called the optimal policy π_*. Optimal policies share the same optimal state-value function $v_*(s) = \max_\pi v_\pi(s)$ and the same optimal action-value function

$$q_*(s,a) = \max_\pi q_\pi(s,a) = E[R_{t+1} + \gamma v_*(S_{t+1})|S_t = a, A_t = a].$$

Because the optimal value function is not dependent on any particular policy we can write the Bellman equation as:

$$v_*(s) = \max_{a \in \mathcal{A}} q_{\pi_*}(s,a) = \max_a E_{\pi_*}[G_t | S_t = s, A_t = a] =$$
$$\max_a E_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] = \max_a E_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] = \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma v_*(s')].$$

This is known as the Bellman optimality equation for the state-value function. Similarly, we can write the Bellman optimality equation for the action-value function as:

$$q_*(s,a) = E[R_{t+1} + \gamma * \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a] =$$
$$\sum_{s',r} p(s', r | s, a)[r + \gamma * \max_{a'} q_*(s', a')].$$

Thus, we can solve the reinforcement learning problem by explicitly solving the Bellman optimality equation. To do this in practice we assume that we know the dynamics of the environment, we have the computational resources, and the Markov property holds. However, in reality these assumptions rarely hold. Solving the Bellman equation explicitly requires a huge computational cost. Also, it is rarely the case that we fully know the dynamic of the environment.

3.5 Deep-Q Learning

The general policy evaluation (GPE) is the alteration between policy evaluation and policy iteration. In GPE we start with some arbitrary initial policy and repeat the evaluation-iteration method until we reach an optimal policy:

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} ... \xrightarrow{I} \pi_* \xrightarrow{E} V_{\pi_*}.$$

We get the next policy from $V_{k+1}(s) = E_\pi[R_{t+1} + \gamma V_k(S_{t+1}) | S_t = s]$ and do the iterations until $|V_k(s) - V_{k+1}| < \epsilon$. The policy improvement theorem guarantees convergence. Most of the time in practice we cannot compute the value functions directly. This is either because the full dynamics of the MDP is unknown or we do not have enough computational power. Then we estimate the value function $V_\pi(s) = E_\pi[G_t | S_t = s]$ from sample averages.

In Q-learning we learn the action-value function and use temporal difference methods for updates:

$$q(S_t, A_t) = q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a q(S_{t+1}, a) - q(S_t, A_t)].$$

Our goal is to minimize the difference between our current estimate of the action-value function and the target. Let the action-value function be parameterized by some parameter θ. Then define the loss function that we want to minimize as $L(\theta) = \frac{1}{N}\sum_{i \in N}(q_\theta(s_i, a_i) - q'_\theta(s_i, a_i))^2$, where $q'(s, a) = R_t + \gamma \max_{a_i} q_\theta(s'_i, a'_i)$. We perform gradient descent to minimize the error. In deep-Q learning our action value function is parameterized by a deep neural network and we use stochastic gradient descent to update the parameter. In order to estimate the values of our actions we employ a version of Monte-Carlo tree search [20], [21]. Monte-Carlo tree search can be summarized as follows:

1. Selection: Starting at root node R we move down the tree by selecting optimal child node until a leaf node L is reached.

2. Expansion: If L is a not a terminal node then create one or more child nodes according to available actions at the current state and select the first of these new nodes M.

3. Simulation: Run a simulated rollout from M until a terminal state is found. The terminal state contains a result that will be returned to upwards in the is backpropagation phase.

4. Backpropagation: After the simulation phase, a result is returned. All nodes from M up to R will be updated.

Chapter 4
Approach

In order to test the hypothesis that real-world performance with a model trained entirely in simulation improves when an appropriate sensor is chosen for training, we first need to select the algorithms to perform the comparison. For reasons discussed in the introduction we have chosen to use a supervised machine learning algorithm and a deep-Q learning algorithm. This means that we need to develop a deep-Q learning as well as supervised learning frameworks for indoor collision avoidance. Both frameworks were written in Python using Keras [22]. The two frameworks control a drone in simulation, collecting data and learning policies for collision avoidance. During drone flying operations, our learning models select actions for indoor navigation that avoid collisions with obstacles given the current state of the drone in the environment. After the training procedure, we need to implement the agent in the real-world using real hardware and test the performance of all models.

4.1 Deep-Q Model

In this section we explain the implementation of our deep-Q learning model. The deep-Q model uses collision data from simulation and a CNN to learn to predict rewards for agent actions. Let s_i denote the current state of the agent, which corresponds to the camera observation at time i in our case. This observation is an image coming either simulation or the real-world. In the training phase the data will come from simulation, and in the testing phase the images

will come from a real camera. The input images are preprocessed and fed into a convolutional neural network. The input images are rescaled to 90x90 pixels before they are fed into the network. In addition, the for the depth images we copy the depth information is over the 3 channels making the depth input 90x90x3. The depth image pixels are also rescaled to a 0-255 range. The training data is collected according to a Monte-Carlo tree search algorithm [4], [23]. To estimate long-horizon rewards for each action we perform an N-step rollout in the direction of each action by following the current policy. This process can be seen in Figure 1.1. Denote a_i as the action that the agent is taking at time i. Then the q-function that our model will try to learn is $Q(s_i, a_i) = \sum_{j=i+1}^{j=i+1+N} \gamma^{j-i-1} * R(s_j, a_j)$, where γ is the discount factor that makes our model put less value in decisions that were made in the distant future and more weight in the recent decisions. The action at each step are made according to our policy $\pi(s_i) = a_i$, which in our case is the epsilon-greedy policy. This policy will select actions with the highest rewards at each step, but with a small probability ϵ it will select a random action in order to make the model explore different actions during the training phase. In theory to make an accurate estimate of the q-function the number of steps N during rollout should approach ∞, but in practice we limit N to make the estimate during training finish in a reasonable time frame. The reward function that we use is $min(1, d_i)$, where d_i is the distance to the nearest obstacle at time i. This reward function encourages the agent to stay at least 1 meter away from obstacles. If a collision occurs the reward will be zero at that time.

4.2 Supervised Model

The other model we use to test our hypothesis is a supervised machine learning model. We use a neural network to predict the probability of an immediate collision for each action. Since we train in simulation, we can easily obtain training

labels for each image and for each action and train the network in a supervised manner. Essentially, our CNN will try to classify each action as one resulting in an immediate collision on the next step or not. To make our RGB model generalize well we need to train it on a large amount of simulated textures, varying the lighting condition as was demonstrated in [5]. Since training on all possible textures is impractical, we restrict our training in simulation to four different textures and we do no vary lighting conditions. In simulation we demonstrate that testing the model on a texture slightly different from what it has seen before degrades performance significantly. The supervised model collects data during training by following an epsilon-greedy policy, selecting actions with the lowest probability of collision.

4.3 Policy and Action Space

An important factor that affects model performance is the size of the action space. If the action space is too large the deep-Q model would require more time training. A large action space means more rollouts have to be performed by the deep-Q agent to estimate rewards. On the other hand, having a very small action space will result in the model not being very maneuverable. The supervised model does not suffer from this constraint because we can instantly get the labels for all actions, so the training time stays the same. We found a good compromise to partition our input images into a 5x5 grid resulting in 25 possible agent actions in total. An example from simulation of our action space can be seen in Figure 4.1. Each grid represents a yaw angle and pitch angle. Once the agent selects a bin it moves 10cm in that direction. This movement is deterministic inside the Gazebo simulator. However, during real-world testing there is drift that contributes to the drone movements. The Parrot Bebop [24] uses its bottom RGB camera for optical flow which is used for stable flight and in the real-world implementation

our deep-Q module relies on Parrot's optical flow for implementing the real-world actions.

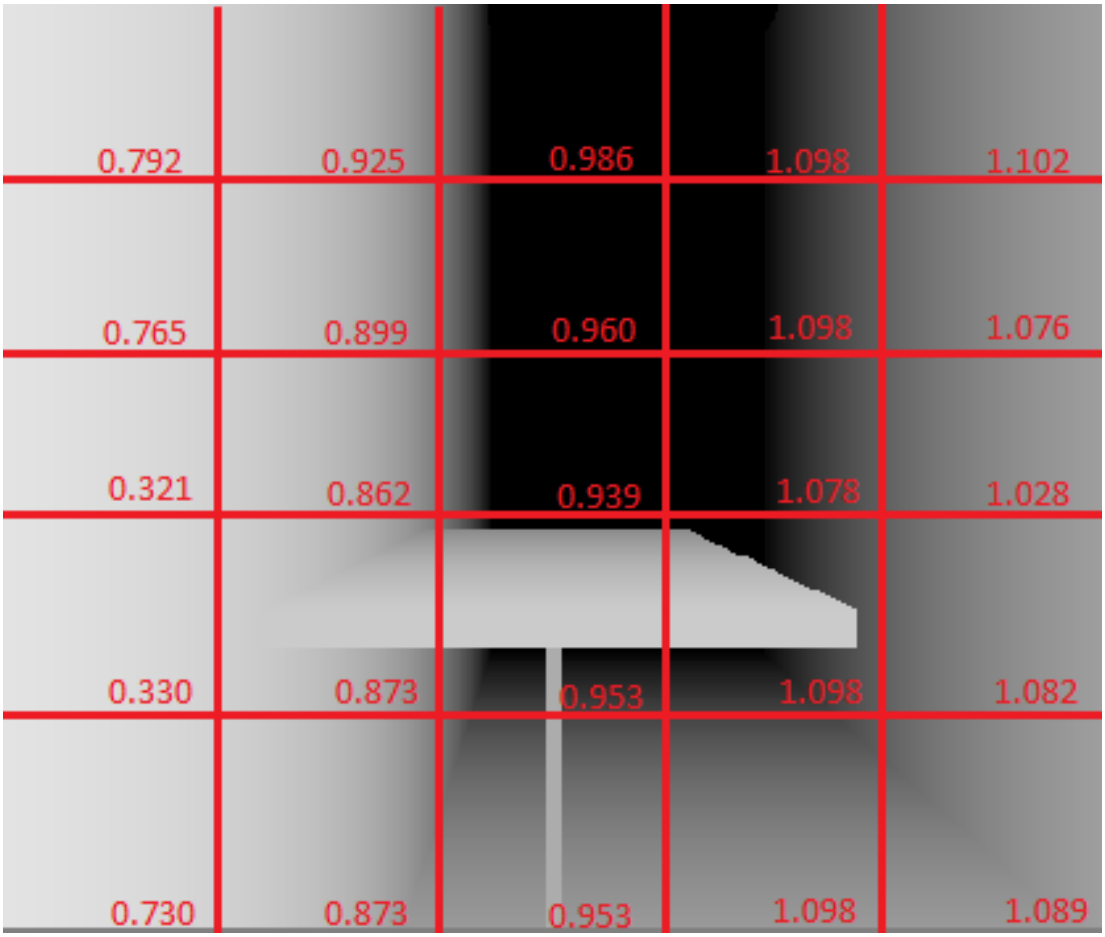

Figure 4.1: A simulated depth image of a hallway with a table as an obstacle. We partition the state image into a grid of yaw and pitch directions. The Monte-Carlo evaluation method is used to estimate the long-horizon future rewards.

4.4 Network Architecture

In order to develop our two learning models for sensor performance comparison we need to choose a CNN architecture to use. A deep neural network is used to learn the q-function and predict long-horizon future rewards in our deep-Q model. Given the state image as input, it estimates the rewards of each action. Because of the success of the VGG16 network architecture on image classification we decided to base our architecture on the VGG16 architecture [25]. Our architecture can be seen in Figure 4.2. Up until the topmost max pooling layer our network is identical to the VGG16 network. But after the last max pooling layer we needed to insert several dense and dropout layers in order to learn the final output of 25 long-horizon rewards for each action. During training the state image is resized to 90x90 before it is fed through our network. The final fully connected layer has linear activation and outputs 25 numbers, which are the cumulative future rewards of each action. This is roughly equivalent to learning the probabilities of collisions for each action since the outputs can be easily be squashed into probabilities using the softmax function i.e the highest reward value would also have the lowest probability of collision. We use the mean squared loss function with Adam optimizer [26] for training the deep-Q model. In the supervised model a VGG16 architecture was used to predict an immediate collision with an object. The difference between the supervised architecture and the one we used for out deep-Q learning model is the activation function in the last layer and the loss function used. For the supervised network we used the sigmoid activation function and we used the binary cross-entropy loss function. For every grid cell in the image the supervised network would output the probability of immediate collision for that action. The training images were processed in the same way as for our deep-Q model. The only difference is that we don't need to perform rollout during training since we have immediate collision data readily available to the

supervised model.

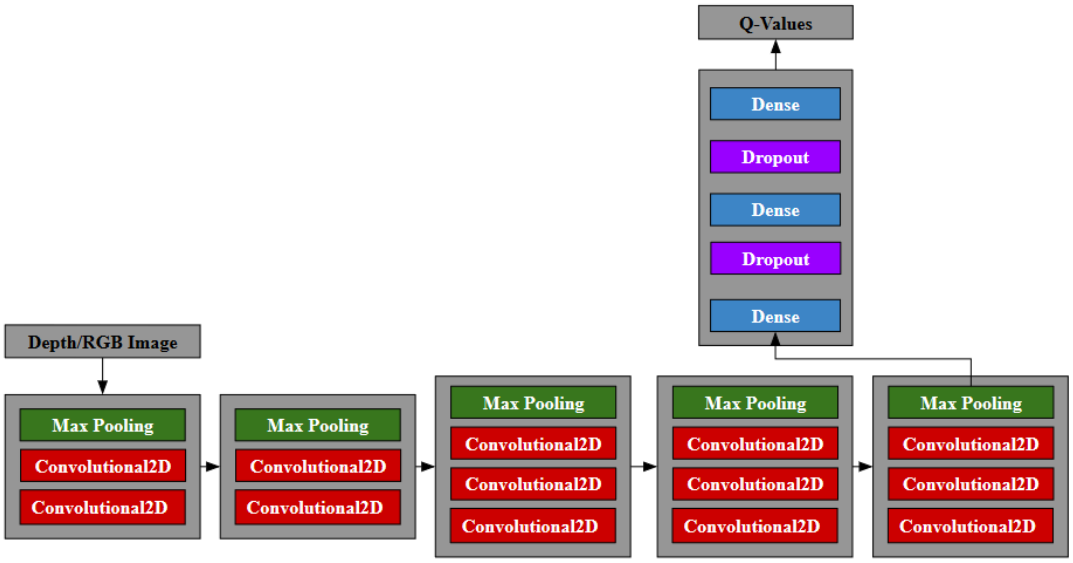

Figure 4.2: Network Architecture used to learn the q-function for collision avoidance. The q-function is used in estimating the long-horizon rewards of each action in each state. The network is based on the VGG16 architecture.

4.5 Gazebo Simulator

Our simulator of choice is Gazebo [27] due to its proven capabilities for machine learning testing [28], [29], [30]. For obstacle avoidance training in Gazebo we selected a number of object models in the Gazebo library to be the obstacles. We made three training maps and one testing map. The training maps are the first three maps in Figure 4.3. The testing map is the rightmost map in Figure 4.3. We tried to make our maps so that they would simulate some common real-world scenarios. For example, the second map in Figure 4.3 simulates the drone trying to move around in an office type area. The last map tests for the ability to follow a long hallway. The objects selected for the obstacles can be seen in Figure 4.4. For our obstacles we tried to select some common object typically seen indoors. To make it more realistic one of the obstacles was a simulated person. For the RGB models training we selected 3 textures for training and 2 for testing. These

textures can be seen in Figure 4.5. The first three textures were used for training the last two for testing. We chose the testing textures so that one of them is very similar to one of the training textures. The second testing texture we selected so that it looks completely different to what the models were trained on.

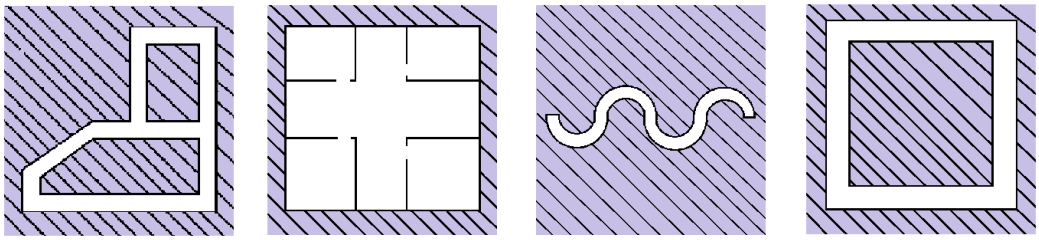

Figure 4.3: Different maps made in gazebo to simulate some real-world scenarios of indoor collision avoidance. First three maps are used for training. The last map is used for testing.

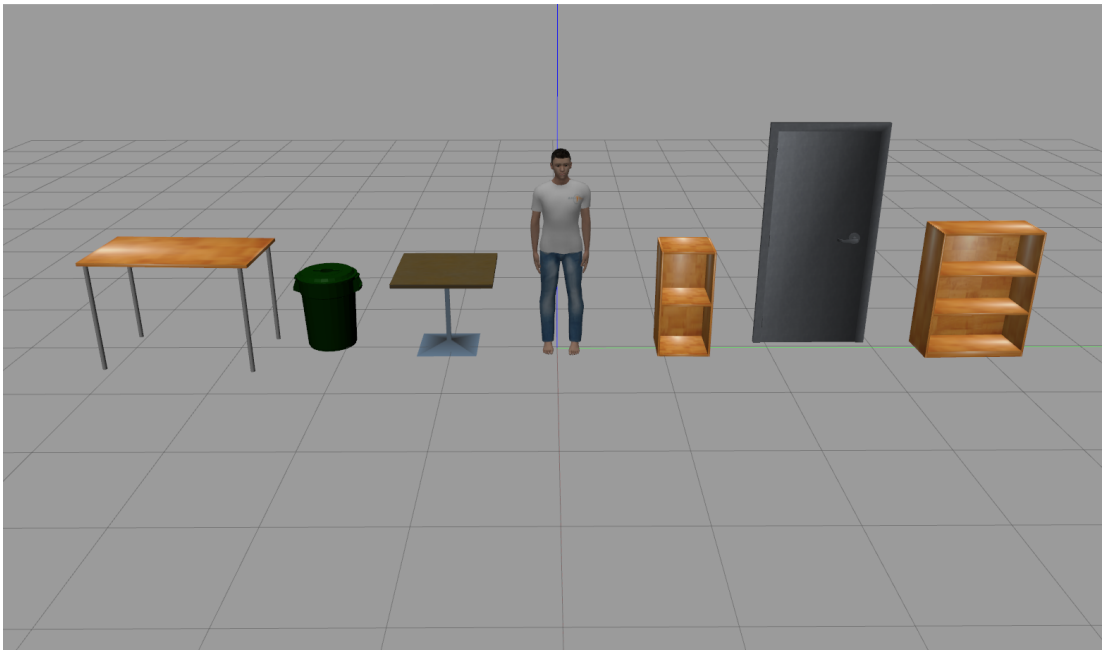

Figure 4.4: Models of common indoor objects available in Gazebo that we used in order to train and test our models.

In order to make our python-written reinforcement learning framework work with Gazebo, we needed to implement a Gazebo plugin as well as a shared library for passing data. Our architecture can be seen in Figure 4.6. In order to transition

Figure 4.5: The simulated textures we used in Gazebo to train and test our models. The first three were used for training the RGB models. The last two were used for testing. The testing textures were selected to test the ability of RGB models to generalize.

between simulation and real-world, our python learning module abstracts the interactions in a general environment class. The environment feedback can then come from either the simulator camera (i.e., Gazebo camera plugin) or from a real-world camera without changing the implementation. To interface Gazebo with our python learning algorithm we implemented a Gazebo plugin in C++ to notify our python environment of state changes and collisions as well as to perform agent actions inside Gazebo. Our Gazebo interface sends a message to the drone plugin depending on the desired action and receives state feedback from Gazebo. A shared library was made in order to pass information between the Gazebo C++ plugin and the python learning module as shown in Figure 4.6.

4.6 Real-World Implementation

In order to test the performance of both modalities in real life we need a drone that has some level of autonomy. The drone has to be able to hover maintaining position and execute high-level vectoring commands. For this purpose we chose the Parrot Bebop 2 drone [24]. In order to obtain stereo information in real-time we selected the ZED mini stereo camera by StereoLabs [31]. The stereo camera is small enough to be placed on UAVs and has good real-time performance. To send commands to the drone and get images back from the camera we implemented a

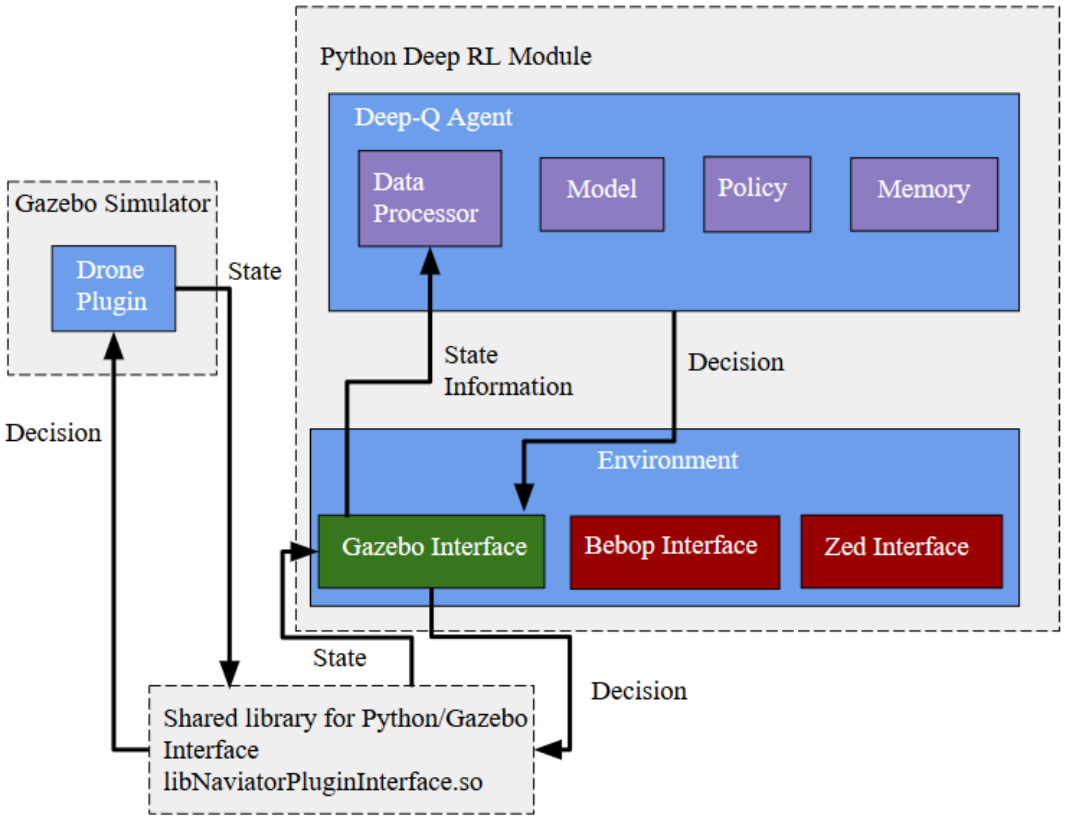

Figure 4.6: The architecture we used to train in simulation. Our Python deep-Q module is communicating decisions to the gazebo simulator, which sends back the environment feedback for evaluation.

Bebop and a ZED interface in Python. These interfaces are abstracted inside the environment class so that the model can be used in simulation or the real-world without any changes to the implementation. This architecture can be seen in Figure 4.6. From the figure we can see that our agent did not require any changes when moved to the real-world and is the same as in Figure 4.7. The companion computer that was selected is the Jetson TX2 board [32], which was reduced in size with the use of the Orbitty Carrier board [33]. Figure 4.8 demonstrates our final testing platform.

Our companion computer and the stereo camera had to be mounted in such a way as to keep the stability of the Parrot drone. Our initial way of mounting introduced oscillations in the drone, so we had to redesign the mounting hardware.

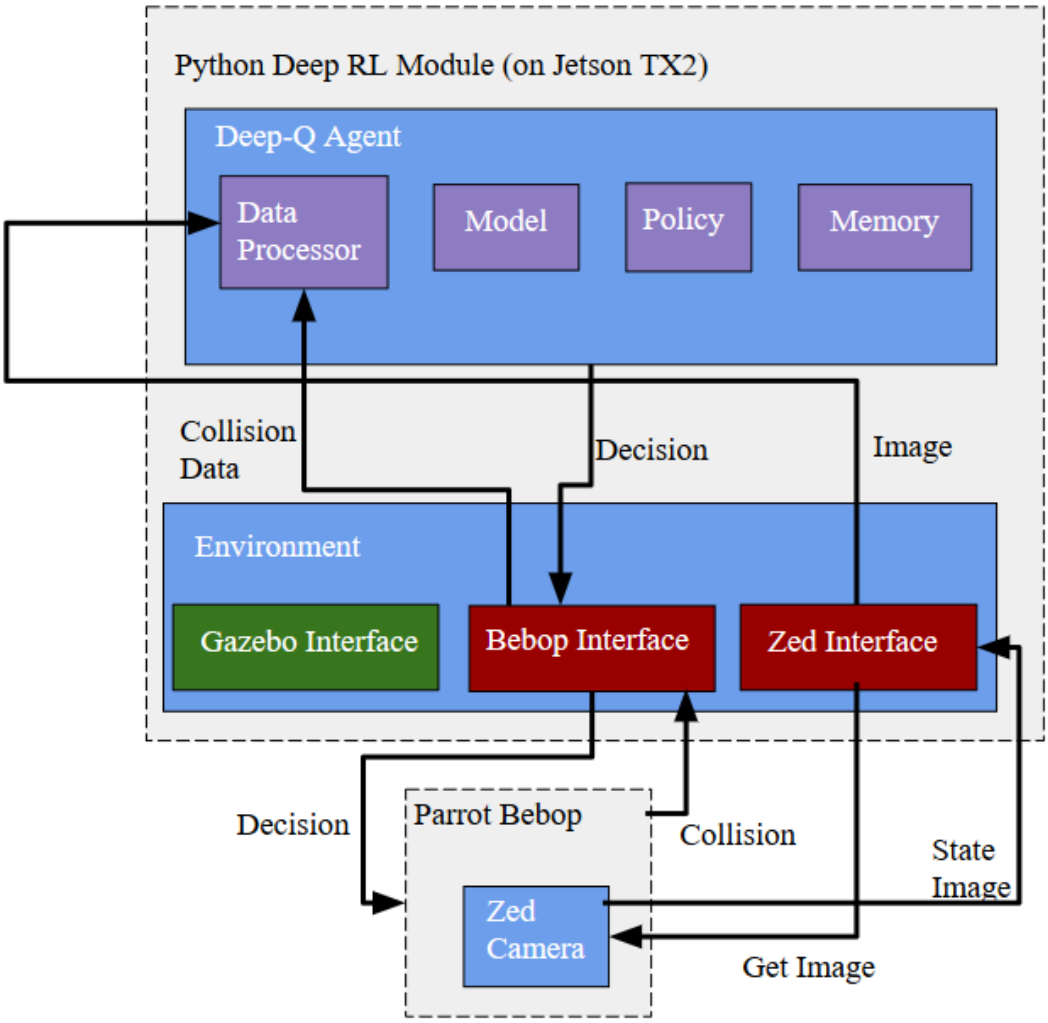

Figure 4.7: The Architecture used for testing in the real-world. The Parrot drone is giving our Python module collision feedback and the stereo camera is giving image data feedback.

Our initial mount was 3D printed, but it ended up not strong enough to survive collisions. Thus, we ended up machining some parts out of aluminum, which provided very good robustness for crashing. The total weight of the Jetson, the ZED camera, and the counting hardware ended up being 160g. The flight time of the Parrot drone was originally approximately 22min, which went to around 11min with our additional hardware.

Figure 4.8: Our real-world testing platform: a Parrot Bebop 2 drone with a Jetson TX2 and a ZED stereo camera.

Chapter 5
Experimental Results

In this chapter we describe our experimental setup and compare the model results. In order to evaluate the performance of the two different sensors we performed several simulated and real-world experiments. First, we describe the testing we performed inside Gazebo. Then, we describe the testing of our models we performed in the real-world. We end the chapter by comparing the model performances in these different testing scenarios.

5.1 Simulated and Real-World Environments

5.1.1 Gazebo Testing Map

We selected a never seen before map in Gazebo and tested the supervised and deep-Q algorithms using both modalities. The testing map is a square shaped 15m x 15m hallway with obstacles and can be seen as the last map in Figure 4.3. The map contains 12 obstacles of type person, door, and cabinet. To see if the RGB models generalize well we selected two never before seen textures: the blue tiles texture, and the dark wood texture.

5.1.2 Real-World Testing Hallway

To test the performance of our models in real-world we selected a small office space. The space can be seen in Figure 5.1. The space has two small rooms that are connected by a hallway. The walls in the testing space were painted white

and did not contain a lot of features. Even in these conditions the stereo camera was able to accurately estimate depth.

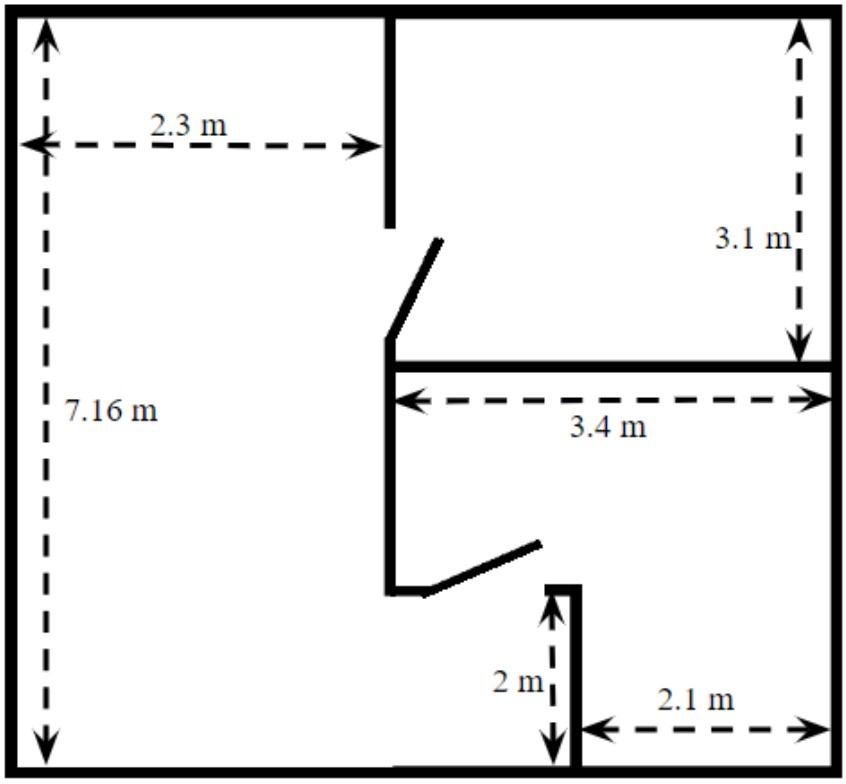

Figure 5.1: The floor plan of the office space where we performed real-world testing. We tested the performance of all of our models by letting the Parrot drone fly through this office space and colliding with obstacles.

In order to test our model we needed a high-level controller that allows small vectorized movements of the drone. The Parrot Bebop 2 supports such movements by using optical flow. We performed several experiments to test the reliability of optical flow and found that in hallways there was significant drift due to lack of features on the ground. The drift of the Parrot drone was resulting in collisions

while making decisions. Since we are only interested in testing our models performance, we chose to try to minimize the drift during decision making. In order to minimize drift of the Parrot Bebop 2 drone we placed tape on the floor to improve optical flow. Even in the presence of more features there were still some collisions due to drift but since all models experienced drift they were compared in equal conditions.

Another challenge we faced during real-world testing was damaging the drone during collisions. The crashes would result in damage to the propellers of the drone. During the duration of our testing we replaced the propellers two times. Some crashes damaged the Jetson mount that we made.

5.2 Comparison of Results

5.2.1 Simulation Results

In order to test the ability of the RGB models to generalize on new textures we selected two new simulated textures. The dark wood texture is very similar to one of the wood textures the RGB models were trained on. On the other hand, the blue tiles texture is very dissimilar to any of the textures we used for training. The depth models ability to generalize was tested using a never before seen hallway. This is in order to obtain depth images that are different than the ones used for training the depth model.

The supervised models results can be seen in Figure 5.2. Here we see that the supervised RGB model that was tested on the dark wood texture slightly outperforms the supervised depth model. This is because a very similar wood texture was present during training the RGB model. On the other hand, the supervised RGB model does worse when tested on the blue tiles texture. The RGB model was not able to generalize well to a new texture. The supervised depth model is independent of texture so its performance did not change with the

change of texture in simulation.

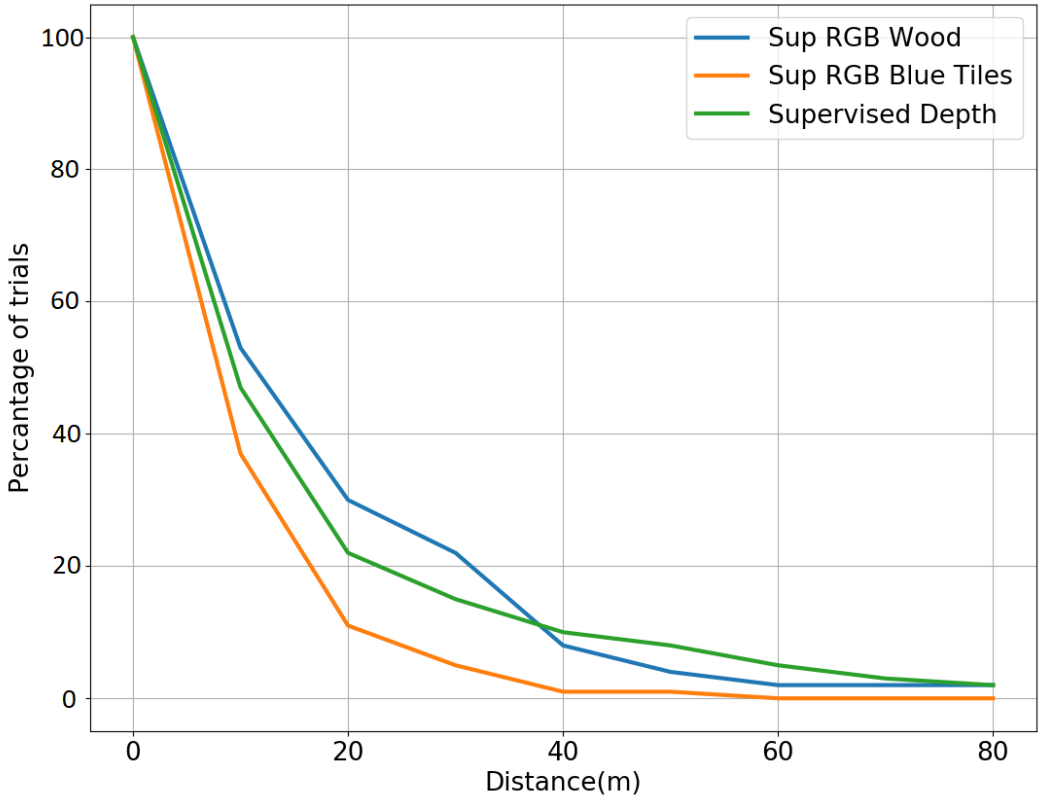

Figure 5.2: Supervised model performance simulated in Gazebo. The supervised RGB model performed much worse on a never-seen before texture (blue tiles) than it did on the same texture it was trained on (wood texture). Supervised depth model performed similarly to the supervised RGB model tested on the wood texture.

The deep-Q models results can be seen in Figure 5.3. The depth models outperform the RGB models. We also tested the depth model in presence of White Gaussian noise in the depth images. It seems like adding the noise during testing improved performance of the depth model. This could be explained by the fact that some points in the depth image will appear closer than they actually are. This is making the depth model more conservative with decisions and make it perform better. Figure 5.4 shows the performance of all models together. In simulation the deep-Q depth models performed the best compared to our other models.

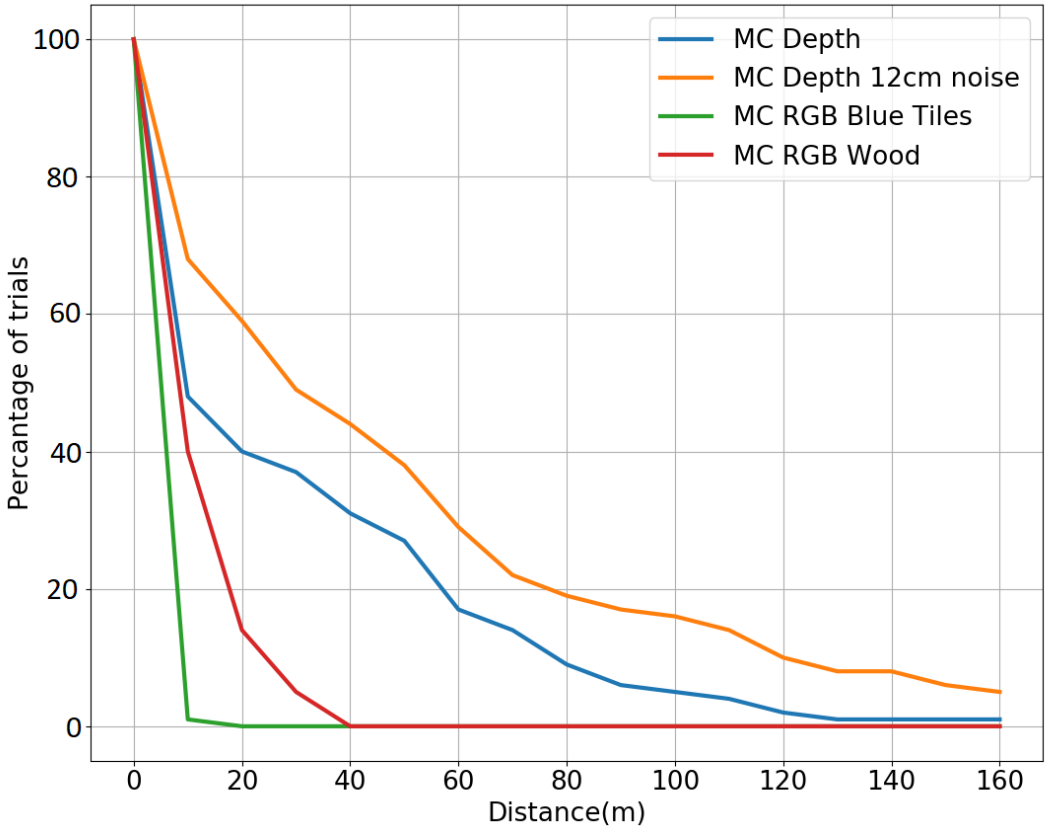

Figure 5.3: Monte-Carlo model performance simulated in Gazebo. The Monte-Carlo models trained on depth data outperform Monte-Carlo models trained on RGB images.

5.2.2 Real-World Results

We tested the RGB and depth models in a small office space that can be seen in Figure 5.1. In addition, we used average distance before collision as a performance metric in Table 5.1. In Figure 5.5 it can be seen that the supervised depth model performed better than the other models.

Unlike in simulation in the real-world the supervised depth model outperformed other models. This is likely because the supervised model is concerned only with predicting the immediate collisions and it is not taking into account future rewards. Normally this would result in worse performance. The reason it did not was because of drone drift during movement. The deep-Q models that

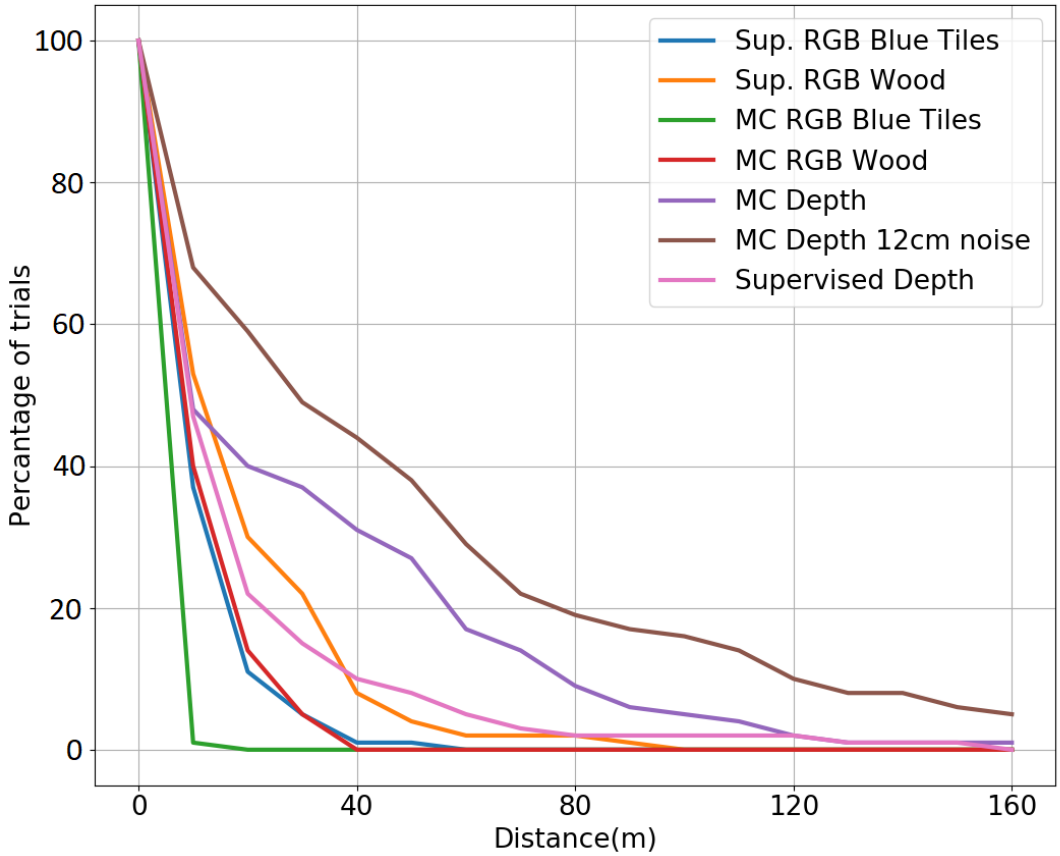

Figure 5.4: Performance of all models simulated inside Gazebo. We can see that in general the Monte-Carlo models trained on depth data perform better.

were trained in simulation learned to predict long-horizon collisions and take that information into account. Once drift comes into the picture in the real-world, those long-horizon probabilities become very inaccurate. So in this case it is better to rely on immediate collision probabilities that the supervised depth model provides. An idea for future work could be trying to simulate drift during training of the simulated models.

During our testing we experienced some collisions that were the result of drift. For example, during some of the decisions when the drone was trying to move left to avoid an obstacle it could drift to the right resulting in a collision. Since all models we tested experienced equal amounts of drift their comparison is still valid.

The real-world results seem to suggest that a model performs better when trained on simulated depth images than simulated RGB images. Whether this remains true for other machine learning approaches remains future work. One could try to see if imitation learning in simulation would perform better when trained on simulated depth. Reinforcement learning methods other than deep-Q learning can also be tried.

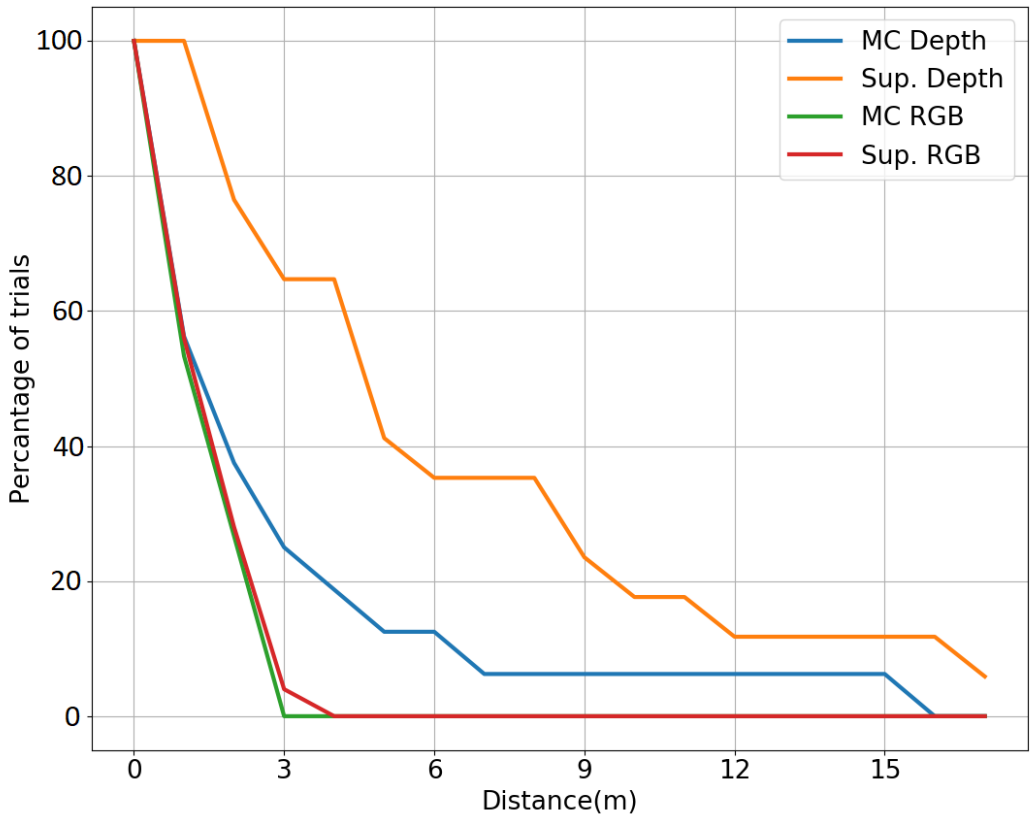

Figure 5.5: Performance of our models on the real-world office space in Figure 5.1. The models that were trained on depth data outperform models trained on RGB images.

Table 5.1: Real-world results

	Travel Dist.	Number of Collisions	Collisions per meter	Avg. Dist
MC Depth	42.9m	16	0.373	2.68m
MC RGB	38.6m	30	0.777	1.29m
Sup Depth	104.4m	17	0.163	6.14m
Sup RGB	36.0m	25	0.694	1.44m

Chapter 6
Conclusion

Training in simulation has many benefits such as reducing time and cost of training. Due to the size and weight constraints of UAVs we can usually select only one sensor to perform collision avoidance. Thus, it is beneficial to select the best performing modality for the purpose of obstacle avoidance. The simulated modalities have a probability distribution that is different from the one in real-world, resulting in poor performance when trained completely in simulation. In this thesis we compared the performance of models trained in simulation on RGB and depth data. The gap between simulated and real-world distributions of these sensors is different resulting in different performance.

In this thesis we showed evidence that deep-Q and supervised models trained on depth can outperform models that were trained on RGB simulated data under certain conditions. Our work presents evidence that performance of models trained in simulation can be improved by selecting an appropriate sensor for training. This evidence can aid in the selection of the sensor to use on a small UAV for collision avoidance. Although we presented evidence of possible advantages of using stereo over RGB for certain cases, further studies need to be made to obtain more certainty what sensor is better and under what conditions. The conditions for which one sensor performs better need to be further studied to have more certainty which sensor is better for UAV collision avoidance.

The results we obtained in simulation show that the deep-Q model trained

on depth performs better than its supervised RGB counterpart. From our simulation results we can see that more trials of the deep-Q depth models achieve a greater distance without collision. Overall we can see that the RGB models performs worse in simulation than depth models. When we move the models over to the real-world we see a different picture. From our real-world results we can see that the depth models still perform better than RGB models, but the supervised depth model performs better than the deep-Q depth model. In simulation the deep-Q depth model was performing better than the supervised depth model. This is an interesting result that needs further investigation. As of now we do not know why this change in performance happened when we moved over the deep-Q models to the real-world. Future work could be investigating how moving over different models to the real-world changes their performance.

Although our results suggest that a model trained on depth could outperform a model trained on RGB, further investigation is required to obtain more confidence. All of our models were trained inside Gazebo simulator and it is possible that if we change the simulator the results will also change. For example, a simulator with better RGB textures could potentially improve the performance of our RGB models. If RGB model performance improves with the use of a better simulator than maybe we would obtain results that show better RGB model performance. In addition, we trained on a limited set of textures and lighting conditions. It could also be that if we trained on more different textures the RGB model would be able to generalize better and its performance would improve. The same reasoning could be applied to the stereo camera. Changing the stereo matching algorithm would result in a change in depth estimation performance and this would effect the performance of our depth models. Furthermore, we never investigated if the model training time has an effect on performance. In our research we trained both models equally long. Perhaps if the RGB models were trained for a longer period of time they would perform better than the depth

models. It is also worth investigating if we obtain the same sensor performance results for other collision avoidance algorithms. It is possible that some other collision avoidance algorithms would work better with RGB instead of stereo.

Another factor that affects model performance in real-life is the difference between simulation and real-world drone flight dynamics. When we trained our models in simulation we assumed that the drone has a perfect trajectory. In real-life this is never the case. The Parrot Bebop 2 that we used for real-world testing relies on optical flow for stable hover and movements. Since the algorithm is not perfect we get drift in the drone movements. This drift affects the performance of our models. Even if the model decides on a collision-free action the drone could still drift in the direction of the obstacle and crashing. A possible fix for this could be training a real drone to move inside of a simulator. In this case we would have to capture the movements of the drone and transfer them into the simulator. The models in this case would learn taking into account drone flight dynamics in addition to learning not to crash.

References

[1] De Wagter, Christophe, et al. "Autonomous flight of a 20-gram flapping wing mav with a 4-gram onboard stereo vision system." Robotics and Automation (ICRA), 2014 IEEE International Conference on. IEEE, 2014.

[2] Giusti, Alessandro, et al. "A machine learning approach to visual perception of forest trails for mobile robots." IEEE Robotics and Automation Letters 1.2 (2016): 661-667.

[3] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.

[4] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[5] Sadeghi, Fereshteh, and Sergey Levine. "(CAD)^{2}RL: Real Single-Image Flight without a Single Real Image." arXiv preprint arXiv:1611.04201 (2016).

[6] Engel, Jakob, Thomas Schöps, and Daniel Cremers. "LSD-SLAM: Large-scale direct monocular SLAM." European conference on computer vision. Springer, Cham, 2014.

[7] Bachrach, Abraham Galton. "Autonomous flight in unstructured and unknown indoor environments." Diss. Massachusetts Institute of Technology, 2009.

[8] Davison, Andrew J., and David W. Murray. "Mobile robot localisation using active vision." European Conference on Computer Vision. Springer, Berlin, Heidelberg, 1998.

[9] Barry, Andrew J., and Russ Tedrake. "Pushbroom stereo for high-speed navigation in cluttered environments." Robotics and automation (icra), 2015 ieee international conference on. IEEE, 2015.

[10] Schmid, Korbinian, et al. "Stereo vision based indoor/outdoor navigation for flying robots." Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on. IEEE, 2013.

[11] Bojarski, Mariusz, et al. "End to end learning for self-driving cars." arXiv preprint arXiv:1604.07316 (2016).

[12] Abbeel, Pieter, Adam Coates, and Andrew Y. Ng. "Autonomous helicopter aerobatics through apprenticeship learning." The International Journal of Robotics Research 29.13 (2010): 1608-1639.

[13] Gandhi, Dhiraj, Lerrel Pinto, and Abhinav Gupta. "Learning to Fly by Crashing." arXiv preprint arXiv:1704.05588 (2017).

[14] Zhu, Yuke, et al. "Target-driven visual navigation in indoor scenes using deep reinforcement learning." Robotics and Automation (ICRA), 2017 IEEE International Conference on. IEEE, 2017.

[15] Pinto, Lerrel, and Abhinav Gupta. "Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours." Robotics and Automation (ICRA), 2016 IEEE International Conference on. IEEE, 2016.

[16] Agrawal, Pulkit, et al. "Learning to poke by poking: Experiential learning of intuitive physics." Advances in Neural Information Processing Systems. 2016.

[17] Levine, Sergey, et al. "Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection." The International Journal of Robotics Research (2016): 0278364917710318.

[18] Michels, Jeff, Ashutosh Saxena, and Andrew Y. Ng. "High speed obstacle avoidance using monocular vision and reinforcement learning." Proceedings of the 22nd international conference on Machine learning. ACM, 2005.

[19] Sutton, Richard S., and Andrew G. Barto. Introduction to reinforcement learning. Vol. 2. No. 4. Cambridge: MIT press, 1998.

[20] Chaslot, Guillaume M. JB, et al. "Progressive strategies for Monte-Carlo tree search." New Mathematics and Natural Computation 4.03 (2008): 343-357.

[21] Browne, Cameron B., et al. "A survey of monte carlo tree search methods." IEEE Transactions on Computational Intelligence and AI in games 4.1 (2012): 1-43.

[22] Keras: high-level neural networks Python API. https://keras.io/

[23] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." nature 529.7587 (2016): 484.

[24] Parrot Bebop 2 Drone: https://www.parrot.com

[25] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[26] Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980 (2014).

[27] Gazebo: Open Source robotics simulator. http://gazebosim.org/

[28] Xie, Linhai, et al. "Towards monocular vision based obstacle avoidance through deep reinforcement learning." arXiv preprint arXiv:1706.09829 (2017).

[29] Rodriguez-Ramos, Alejandro, et al. "A deep reinforcement learning strategy for UAV autonomous landing on a moving platform." Journal of Intelligent and Robotic Systems 93.1-2 (2019): 351-366.

[30] Koch, William, et al. "Reinforcement learning for uav attitude control." ACM Transactions on Cyber-Physical Systems 3.2 (2019): 22.

[31] ZED Stereo Camera: https://www.stereolabs.com/

[32] Jetson TX2 Module: https://developer.nvidia.com/embedded/jetson-tx2

[33] Orbitty Carrier Board: http://connecttech.com/product/orbitty-carrier-for-nvidia-jetson-tx2-tx1/