CNN-Based Autonomous Navigation for Micro Air Vehicles

Group 8: Danish Ansari, Damian Bhawan, Patrick Kostelac, David Schep, Manu Singh, Gijs Zijderveld
Code: https://github.com/TU-DAnsari/paparazzi
Delft University of Technology
April 3, 2024

ABSTRACT

The goal of the MAV course's (AE4317) project was to cover as much distance as possible with the Parrot Bebop Drone (Zoë, our drone's name) in the CyberZoo, while avoiding random obstacles. For obstacle avoidance, various techniques such as optic flow, mapping, edge detection and monocular depth estimation were explored. The final chosen approach contains a control strategy based on a convolutional neural network model. This CNN model uses depth estimation for labelling a self captured dataset. The CNN model predicts the presence of the obstacle in three segments of the image, with the drone being controlled according to these probabilities while trying to maximise continuous flight. Using this approach, our drone covered a distance of 130.3 metres in the CyberZoo in the allotted 10 minutes.

1 Introduction

Many different types of sensors can be used with the purpose of machine perception to aid a Micro Air Vehicle (MAV) in autonomous flight. The problem is that the sensors that provide high-fidelity information about the environment are active sensors, e.g. LIDAR, which increases cost, weight and energy consumption, ultimately leading to shorter flight time. Onboard video cameras are passive sensors that are relatively cheap and light, which make them a more suitable option. This project aims to make use of the onboard mono-camera on a commercially available drone to develop a vision-based obstacle avoidance algorithm.

2 Explored Approaches

This section goes over some of the approaches that were initially explored for obstacle avoidance and path planning but were ultimately left out due to various reasons.

2.1 Orange Avoidance

Since a part of the obstacles were guaranteed to be orange, the first implementation of obstacle avoidance was made specifically to avoid orange poles while trying to maximise continuous flight. By calculating the fraction of orange pixels in different regions of the camera images, the weighted difference between the left regions and right regions could be used to set the yaw rate. A frontal obstacle is detected when the fraction of orange pixels exceeds a certain threshold, which results in stopping and turning to search for a safe heading.

When solely avoiding orange poles, this method worked quite reliably, and allowed for prolonged continuous flight. However, an additional method of obstacle detection and avoidance would be necessary to ensure even obstacles that are not orange are detected as well.

2.2 Optic Flow

By calculating optical flow vectors between consecutive camera images, the magnitude of the optical flow vectors can be used to determine how crowded different areas of an image are with obstacles. Using the difference between the magnitude of optical flow between the left side and right side of the image, the heading rate of the drone can be controlled to move away from the obstacles [1].

While optical flow can provide information about obstacles on the sides of the image, it does not directly provide any information about obstacles in the direction of where the drone is headed. Additionally, both sparse flow and dense flow calculations are quite noisy and would require additional processing to make them usable for fast-paced obstacle avoidance. In the interest of time, optical flow was ultimately not used.

2.3 Edge Detection

The edge detection feature was already implemented in one of the paparazzi files, which means that it only had to be implemented in the `orange_avoider` module. The strategy we planned for this was quite simple. The edge detection feature outputs a grayscale image that depicts edges in white. Once we get the edges, the image would then be split up into five segments, and based on the sum of the intensity of the pixels in each segment of the image, the drone would choose its heading. The drone would then turn in the direction where the least amount of edges through the sum of pixel intensities are [2]. The main issue with this method is that the output of the image was quite noisy. This is because the feature detects edges at objects, and also at the edges of the CyberZoo. This makes it hard to detect obstacles in the CyberZoo. For these reasons, this feature was excluded from the final approach.

2.4 Monocular Depth Perception

Monocular Depth Estimation is the task of estimating the depth value (distance relative to the camera) of each pixel given a single (monocular) RGB image. For this, a general pre-trained monocular depth perception CNN model was used. The concept and code were taken from Boosting Light-Weight Depth Estimation Via Knowledge Distillation [3]. This paper proposed a lightweight network that can accurately estimate depth maps using minimal computing resources. After implementing this, we got the performance of 33 milliseconds per frame or 30fps on a high-end i7 (forced single core). However, the performance of the drone would be

significantly worse. Pruning was explored to enhance performance. We got an 8 % increment in performance but this was still slow for real-time applications in the drone. Quantization was also explored. However, quantizing for older ARM hardware is less effective as the ARM version does not have the right instructions for int8 inference [4].

We could have continued refining but we encountered two problems that made us abandon this approach: First, since the model is in Pytorch, we need to convert the model to C for deployment. However, using onnx2c [5] did not work as one of the operators was not implemented. Second, the code available for this method is only for inference and not for training. However, to make it work in CyberZoo, it was needed to fine-tune with new data and writing custom training code would have taken too long.

2.5 Mapping

The drone maps the area, selects an obstacle-free path, and updates its map based on sensor data. A 2D map is used due to computational limits. Map cells are categorized into three groups: empty, unknown, and obstacle cells. Unknown cells can have both empty and obstacle cells. The updating process includes two methods: First, utilizing the drone's precise location obtained from the OptiTrack system in the CyberZoo as a periodic update to mark safe areas on the map. Secondly, responding to alerts from the local path planner indicates the need to deviate from the global path due to detected obstacles. In such instances, the drone adjusts its global path by removing waypoints and marking cells between the drone and the waypoint as obstacle cells.

These methods are adaptable but may lack real-world practicality due to issues in fine-tuning interaction between local and global planners. Although this enhancement to the drone was prepared for competition, there wasn't adequate time to fine-tune the interaction between the local and global planners, especially concerning the circumstances under which the local path planner should supersede the global one.

3 Chosen Approach

Section 2 goes over some promising methods for the purposes of obstacle avoidance, however, most of them have their limitations. The methods are either subject to noise, e.g. optical flow and edge detection, or are either too computationally intensive to run onboard the drone, e.g. monocular depth estimation.

For this reason, a different approach is taken, which uses CNNs to estimate the probability of obstacle presence within certain regions, directly from the provided camera images. With enough data, this method can have a broader range of applications and would not be limited to obstacle avoidance within the confines of the cyberzoo, which is the main reason this method was chosen as the main approach for obstacle avoidance.

3.1 CNN architecture

We used the MobileNetV3 model [6] which is specially designed for computer vision applications in devices with limited computational resources. MobileNetV3 incorporates efficient building blocks such as depthwise separable convolutions and linear bottlenecks to reduce the computational cost while maintaining high accuracy.

3.1.1 MobileNetV3 Configuration

MovileNetV3 comes with 2 configurations, $small$ and $large$. These configurations define the specifics of the MobileNetV3 blocks that are stacked on top of each other. The 3 most important configuration values are shown in Table 1. The kernel size defines the convolution kernel size used in that block. The Expansion size defines how much the block will expand and then project back (see the MobileNetV3 paper for more details [6]). And the channels define the number of channels of the block.

The small configuration contains 1.5 million trainable parameters spread over 194 layers. For the low-powered CPU on the Bebop Drone, it will be too resource intensive. For this reason we created the tiny configuration which only has 108 thousand trainable parameters over 93 layers.

There are 3 important changes from the small config. First, the MobileNetV3 blocks. As shown in table 1c the number of blocks is significantly reduced. The expansion size is also reduced and the maximum number of channels is also lowered.

The second change is to the first layer. Profiling the model (on a laptop high-end CPU) showed that the first convolution layer took approximately 60 milliseconds to run. While the total execution time, including all other 92 layers, was 90 milliseconds. The bottle-neck was caused by the first layer which immediately increased the channels to 16 while the width and height were still quite large. To remedy this, the first layer only increases the channels to 4. After which the first MobileNetV3 block increases to 8 and then 16.

The last change was to the classifier, there is a final fully connected layer with a hidden dimension. For $large$ the dimension is 1280, for $small$ it is 1080. And for $tiny$ this was chosen as 520.

Including the last two optimisations the model took 120 milliseconds to run on the drone, instead of 180 milliseconds with only the first optimisation.

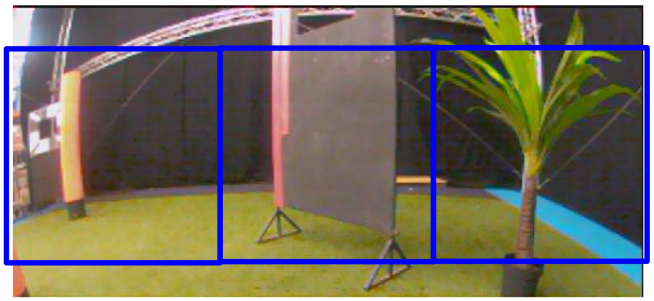

Figure 1: Patch split

Table 1: MobileNetV3 Configurations

(a) Large

Kernel Size	Expansion Size	Channels
3	1	16
3	4	24
3	3	24
5	3	40
5	3	40
5	3	40
3	6	80
3	2.5	80
3	2.3	80
3	2.3	80
3	6	112
3	6	112
5	6	160
5	6	160
5	6	160

(b) Small

Kernel Size	Expansion Size	Channels
3	1	16
3	4.5	24
3	3.67	24
5	4	40
5	6	40
5	6	40
5	3	48
5	3	48
5	6	96
5	6	96
5	6	96

(c) Tiny (our)

Kernel Size	Expansion Size	Channels
3	1	8
3	1	16
3	4.5	24
3	3.67	24
5	3	40
5	3	40

3.1.2 Dataset Creation

We collected data by flying the drone around and recording 8000 images. These images represented the various obstacles and backgrounds. After collecting the dataset, the RGB images were transformed into depth images using a large Deep Learning model. This model can be seen as the teacher model. DepthAnything [7] was chosen for it's state-of-the-art performance.

We segment the RGB images and corresponding depth images into three segments by converting the original images 520×240 pixels into 3 segments of 170×170 after removing some pixels from top and bottom, as shown in Figure 1. The segments are then downscaled to 85×85. Then we used these segmented depth images for labelling our dataset. We chose to create labels based on the presence of obstacles nearby. For this, we used a histogram filled with the depth estimation per pixel, ranging from 0 to 255. We are only interested in the

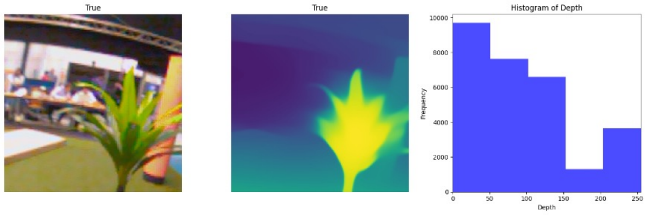

Figure 2: Dataset creation through Depth estimation

rightmost bin, as this includes all the pixels that are estimated to be close to the drone. If that frequency crosses the threshold of 1500, then we label it as True for a nearby obstacle.

The resulting dataset is imbalanced, the final dataset has 1.9 times more negative samples than positive samples. Since, the task is binary classification, we will have to consider this imbalance while training.

3.1.3 Model Training

We used the custom MobileNetV3 tiny model as described in section 3.3.1. For training, we used Adam Optimiser with a learning rate of 0.001 in 8 batches and 40 epochs. We used Binary Cross Entropy as a loss function. This loss function can handle class disparity effectively. The code for the dataset creation, model architecture, model training and converting to C, is all available on our github repo.

3.2 Control Strategy

As the goal of the competition was to cover as much distance as possible while avoiding obstacles, maximising continuous flight time is of utmost importance. In the provided `orange_avoider` module, the drone would stop and turn when it detected an obstacle. This resulted in a large portion of time being spent hovering in place.

An optimal control scheme would allow for continuous control, which means the drone should fly forward continuously while the heading rate is used to steer away from obstacles. This cuts down on the amount of time the drone is not covering any distance. To allow for this direct control over linear and angular velocities, the drone is controlled using the guided mode rather than the navigation move.

Using the output of the neural network, continuous control can be implemented by using the difference in probabilities between the left and right regions to apply a heading rate that steers the drone away from the region with the higher probability. With a sufficiently low forward velocity, the CNN can accurately provide probabilities for the presence of obstacles within these different regions. To mitigate the effects of noisy outputs from CNN, the Exponentially Weighted Moving Average (EWMA) was taken over the last 5 outputs

for each region of the image.

Additionally, to counteract forward drift when applying a heading-rate, an additional inward velocity in the y-direction of the drone is applied to steer into the bend.

When continuous flight is no longer feasible, i.e. there is a probability of there being an obstacle in the middle of the camera images above a certain threshold, the drone stops and turns to find a safe heading direction. When a frontal obstacle is encountered, the current position p_o of the drone is stored. To avoid collisions with frontal obstacles, a backward velocity that is proportional to the distance between the drone's current position and the saved position p_o, is applied. Moreover, this backward velocity ensures that there is enough space between the drone and the frontal obstacle to allow for a turn in place. The position p_o is reset when a new frontal obstacle is detected.

To ensure that the drone stays within the bounds of the CyberZoo, the OptiTrack motion capture system was used to keep track of the drone's xy-position. When the drone reaches the edge, it stops and searches for a safe heading direction. Additionally, when the drone approaches the edge of the CyberZoo, the forward velocity of the drone is scaled down to ensure the drone does not drift outward when stopping at the edge.

4 Testing

We tested the code developed for autonomous navigation first in the Gazebo simulator and then in the CyberZoo. In this section, the test phases will be discussed in more detail.

4.1 Simulation

Testing in simulation first is required to save time and resources due to the limited allocated time to test in the CyberZoo. Different obstacle detection approaches were tested in simulation, and those with subpar performance were abandoned. Testing in simulation also helped in bringing forward some software issues which could have been over-sighted if directly tested in CyberZoo. Some issues, however, need real-world testing. First and foremost, the CNN used to detect obstacles could not be tested in simulation, as the obstacles appearing in simulation appear different in real life. One problem that was detected in simulation testing is the potential danger of continuous turning when the drone is close to the edge. In such cases, the drone would enter the close-to-edge case, which turned the drone towards the centre of the CyberZoo. In this case, however, the drone did not perform any obstacle detection, which meant that the drone would crash into nearby obstacles when turning inwards. As this was realised very late, the solution was to remove the continuous turning and stop the drone when close to an edge.

4.2 CyberZoo

Every week during the course, the drone could be tested in real life. Every technique described in section 3 and section 2, which could also be successfully implemented in simulation, was then tested in the CyberZoo. It was important to test in the CyberZoo to refine our control strategy. The control strategy, as described in section 3.2 had to be adjusted after every test day in the CyberZoo. We started with the basic control strategy and fine-tuned it by doing the gain tuning, as gains in the simulation model were not enough for the CyberZoo.

Testing in the CyberZoo also helped us in improving our CNN model. The more data is feed to a CNN, the better the CNN will do its job. Data was recorded while flying around and testing, but the best data was gathered while holding the drone in our hands while walking in the CyberZoo. Different obstacles could then be scanned more thoroughly. During the testing of the CNN, it was then important to fly the drone at the same height as the trained data was recorded.

5 Results

We got >80% accuracy after training several CNN models and this accuracy was on real-world data. However, since there is a class imbalance, accuracy is not the desired metric for choosing the model. We trained multiple models with different architectures and hyper-parameters and the best model was chosen based on a confusion matrix. We desired the confusion matrix with more false positives than false negatives for the obstacle detection task. The deployed model (v8 accuracy on the test images is 86.157% and the confusion matrix is closest to what we desired:

	Predicted Positive	Predicted Negative
Actual Positive	1235	306
Actual Negative	296	2512

In the competition, our drone covered a distance of 130.3 meters in the CyberZoo. There was an operator error in keeping the drone at an altitude different from the one where the CNN model was trained. This though did not affect the initial phase of the flight where there were fewer obstacles, but its performance deprecated soon when more obstacles were introduced. Once we corrected the altitude, the drone performance was increased and the drone was able to avoid obstacles and navigate smoothly. There was an issue of battery dying and slower prediction for some obstacles that made us lose some time.

6 Discussion

Though we chose accuracy as our main metric along with the confusion matrix, we need to look for other metrics such as F1 score which is better at class imbalances and Recall (sensitivity) which considers False Negative as a higher concern. Our CNN model was trained between 40 to 80 epochs due to time constraints. However, we could try with a larger number of epochs to get better results. Moreover, tiny configuration CNN was designed empirically with little data. More data could lead to better performance. Furthermore, we did not do hyper-parameter optimisation which could give us parameters for best performance. Our learning rate was kept once but we can also employ learning rate scheduling. In the future, we would also like to add other explored approaches such as orange avoidance and edge detection to the CNN model to make obstacle detection and avoidance more robust.

References

[1] G. Cho, J. Kim, and H. Oh, "Vision-based obstacle avoidance strategies for mavs using optical flows in 3-d textured environments," *Sensors*, vol. 19, no. 11, 2019, ISSN: 1424-8220. DOI: 10.3390/s19112523. [Online]. Available: https://www.mdpi.com/1424-8220/19/11/2523.

[2] C. Harris and M. Stephens, "A combined corner and edge detector," in *Alvey Vision Conference*, vol. 15, Aug. 1988, p. 50.

[3] J. Hu, C. Fan, H. Jiang, X. Guo, X. Lu, and T. L. Lam, "Boosting light-weight depth estimation via knowledge distillation," *CoRR*, vol. abs/2105.06143, 2021. arXiv: 2105.06143. [Online]. Available: https://arxiv.org/abs/2105.06143.

[4] *Quantize onnx models*. [Online]. Available: https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html.

[5] Kraiskil, *Kraiskil/onnx2c: Open neural network exchange to c compiler.* Apr. 2024. [Online]. Available: https://github.com/kraiskil/onnx2c.

[6] A. Howard, M. Sandler, G. Chu, *et al.*, *Searching for mobilenetv3*, 2019. arXiv: 1905.02244 [cs.CV].

[7] L. Yang, B. Kang, Z. Huang, X. Xu, J. Feng, and H. Zhao, *Depth anything: Unleashing the power of large-scale unlabeled data*, 2024. arXiv: 2401.10891 [cs.CV].