國立臺灣大學工學院土木工程學系

碩士論文

Department of Civil Engineering

College of Engineering

National Taiwan University

Master's Thesis

以裂縫自動探索為導向之無人機整合飛控系統研發

Development of an Integrated UAV Flight Control System for Automated Crack Detection and Exploration

陳泓瑋

Hung-Wei Chen

指導教授: 吳日騰 博士

Advisor: Rih-Teng Wu, Ph.D.

中華民國 113 年 7 月

July 2024

doi:10.6342/NTU202403413

Acknowledgements

能夠完成這項研究，首先我要感謝我的指導教授吳日騰老師。這兩年，您給予我許多研究上的幫助，並且提升了我解決新問題的能力。與您對談的過程中，我獲益良多，彌補了許多我未曾考慮到的問題。相信這些能力在未來無論在哪個領域都會對我有很大的幫助。

另外，我也要感謝參與我口試的老師們：仁毓老師、子剛老師和芳耀博士。感謝你們在颱風來襲時仍抽出寶貴時間，幫助我使這個研究變得更為完整。

接下來，我要感謝我研究所的同學們。謝謝 912 研究室歡樂的氛圍，讓我在這兩年中暫時忘卻了辛苦和煩惱。913 的同學們也從不吝嗇地給予我許多幫助及指導。感謝學長們：Jimmy、廷謙、凱哥、和哥、霖哥、淳皓，您們給予我一個很好的研究標竿，讓我能夠向您們看齊，也提供了許多研究上的幫助。感謝泊赫、景諺、政源、簡大哥 MSV 四巨頭這兩年接納我加入您們的歡樂氛圍。感謝亭諺在 coding 方面的指導以及電腦安裝軟體的協助。感謝家文、宥惟這兩年每周一起去國震 meeting 給予的幫助。感謝海威在口試時的互相幫忙，讓一切有了好的結果。謝謝學弟們：彭大哥、豪哥、冠廷、亦德、哲銘、定志，您們給予了我許多幫助。

另外，我也感謝我大學的朋友鈞諺和冠証，雖然在不同學校，但這兩年我們一起讀了很多書，互相交換了資源，也祝您們未來一切順利。感謝我的家人們在

doi:10.6342/NTU202403413

這兩年的支持，沒有您們，我不會有今天的成就。感謝我的女朋友韻璇的陪伴，每當壓力大或心情不好時，您總是陪在我身邊，幫助我度過難關。

最後，我想要感謝這兩年的自己，感謝自己的堅持和信任，使我能夠取得現在的成果。期許未來能夠繼續保持這樣的學習態度和所學的知識，走得更加順遂。

doi:10.6342/NTU202403413

摘要

在鋼筋混凝土建築中，裂縫不僅是結構健康與安全的重要指標，更是地震後評估與補強工作中至關重要的信息來源。地震發生時，結構工程師需迅速評估裂縫的尺寸、形態及位置，以判斷是否存在即時的結構危害，並制定相應的補強策略。因此，裂縫檢測在現代社會中具有重要意義，這項技術不僅影響到建築物的長期穩定性，也直接關乎公眾的安全與福祉。近年來，隨著硬體技術的迅速進步及人力成本的提高，無人機技術開始被廣泛應用於裂縫檢測領域。儘管學術界已有大量相關研究，但許多應用仍限於無人機作為資料收集平台，未能實現裂縫的自動捕捉及自主飛行。本研究針對此問題，致力於開發一種新型無人機飛行系統，專為實現裂縫的自動捕捉和強化自主飛行能力而設計。首先，我們採用深度強化學習方法對模型進行精細訓練，使其能夠基於局部裂縫特徵有效判斷最佳飛行路徑。其次，為實現無人機的自主飛行，我們運用 MAVLINK 無人機通信協議編寫了專屬的飛行控制指令，以確保系統在各種環境條件下能夠靈活適應並執行任務。同時，我們在系統設計中融合了先進的軟體與硬體技術，保證無人機在飛行過程中能夠即時進行邊緣運算，實現高效的全自動操作。實驗結果表明，該系統成功捕捉並分析了 62% 的可見裂縫，檢測準確率為 86%。這些發現突顯了該系統在提高結構檢測的可靠性和效率方面的潛力。

關鍵字：裂縫檢測、無人機、深度強化學習、邊緣計算、自主飛行

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

Abstract

In reinforced concrete structures, cracks serve not only as crucial indicators of structural health and safety but also as vital sources of information for post-earthquake assessment and reinforcement planning. During seismic events, structural engineers must swiftly assess the size, morphology, and locations of cracks to determine immediate structural hazards and devise appropriate reinforcement strategies. Hence, crack detection holds significant importance in modern society, impacting both the long-term stability of buildings and public safety and well-being directly. In recent years, with rapid advancements in hardware technology and the increasing costs of human labor, unmanned aerial vehicle (UAV) technology has found widespread application in the field of crack detection. Despite extensive academic research, many applications still confine UAVs to data collection platforms, failing to achieve automated crack detection and autonomous flight. Addressing this gap, this study focuses on developing a novel UAV flight system designed specifically for automated crack detection and enhanced autonomous flight ca-

doi:10.6342/NTU202403413

pabilities. Firstly, we employ a deep reinforcement learning approach to finely train the model, enabling it to effectively determine optimal flight paths based on local crack features. Secondly, to achieve autonomous flight, we utilize MAVLink UAV communication protocol to develop tailored flight control commands, ensuring the system's adaptability and mission execution across diverse environmental conditions. Concurrently, our system design integrates advanced software and hardware technologies to facilitate real-time edge computing during UAV flight, thereby achieving efficient fully automated operations. Experimental results indicate that the system successfully captured and analyzed 62% of visible cracks, with a detection accuracy rate of 86%. These findings underscore the system's potential in improving the reliability and efficiency of structural inspections.

Keywords: Crack Detection, Unmanned Aerial Vehicle, Deep Reinforcement Learning, MAVLINK, Edge Computing, Autonomous Flight

doi:10.6342/NTU202403413

Contents

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

List of Figures

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

List of Tables

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

Chapter 1 Introduction

1.1 Research Background and Motivation

Structural Health Monitoring (SHM) plays a crucial role in ensuring the integrity and safety of civil infrastructure. Cracks, as potential structural defects, can pose significant safety risks and lead to structural damage, making their accurate and timely detection paramount. Traditional crack detection methods typically rely on manual visual inspections and measurements, which are prone to subjectivity and instability, and may require significant time and cost. For example, during visual inspections, detection personnel may be influenced by lighting conditions, angles, and subjective judgments, leading to inconsistency and inaccuracy in the detection results. Additionally, due to the complexity and scale of civil infrastructure, manual inspections may require substantial human and time resources, especially when conducting inspections over large areas of structures. In this context, the development of efficient and accurate crack detection systems by integrating advanced sensing technologies and automation methods becomes particularly important. Further application of advanced technologies such as remote sensing, digital imaging, and machine learning can effectively capture subtle changes in structures and rapidly identify and locate structural defects such as cracks. These automated detection systems not only improve detection efficiency but also reduce the impact of human factors on detection re-

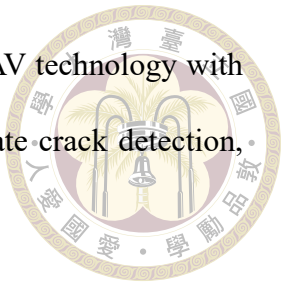
doi:10.6342/NTU202403413

sults, thereby enhancing detection accuracy and consistency. With the increasing demand for infrastructure safety and sustainability, the adoption of advanced crack detection systems in SHM practices is becoming more prevalent.

In recent years, with the rapid advancement of hardware technology, unmanned aerial vehicles (UAVs) have been increasingly utilized in various fields, including commercial, recreational, and military sectors, and have gradually become essential tools for structural health monitoring in civil engineering. Equipped with high-performance cameras and sensing devices, UAVs can conduct comprehensive inspections of structures from different angles and heights, enabling rapid and accurate detection of cracks and other structural defects. Utilizing UAVs for crack detection not only significantly improves detection efficiency but also reduces manpower and time costs. With the rapid development of computer vision and artificial intelligence technologies, UAVs combined with deep learning algorithms are playing an increasingly important role in the field of structural health monitoring. By deploying advanced deep learning models on UAVs, automatic identification and classification of cracks can be achieved, greatly enhancing detection accuracy and reliability. Moreover, combined with the efficient flight capabilities of UAVs, rapid inspections of large-scale structures can be performed, providing a novel solution for structural health monitoring. Furthermore, UAVs offer flexibility and adaptability advantages in structural health monitoring. They can easily access various complex environments and hard-to-reach areas, such as high altitudes, narrow spaces, and steep mountainous regions, to achieve comprehensive coverage and monitoring of structures. This flexibility provides engineers and technicians with more options to better understand the health status of structures and take timely maintenance measures to ensure their safety and stability. In conclusion, the application of UAV technology has brought new possibilities

doi:10.6342/NTU202403413

and opportunities for structural health monitoring. By combining UAV technology with advanced sensing devices, we can achieve more efficient and accurate crack detection, thereby ensuring the safety and sustainability of civil infrastructure.



1.2 Literature Review

In recent years, there has been a surge in research leveraging Artificial Intelligence (AI) and Deep Learning (DL) techniques for crack inspection in civil engineering. Various studies have explored the application of convolutional neural networks (CNNs), recurrent neural networks (RNNs), and other deep learning architectures to automatically detect and classify cracks in structural components. For instance, a novel approach to fast tunnel crack inspection using a mobile tunnel inspection system (MTIS) is presented in a study [1]. By integrating advanced imaging technology and a lightweight convolutional neural network (CNN) for crack detection, the system achieves efficient and accurate detection of tunnel cracks, even at high driving speeds. Additionally, a deep learning framework tailored for crack detection in nuclear power plant inspection videos, known as NB-CNN, is introduced in another study [2]. This framework offers a high accuracy rate of 98.3% and low false positive rates of 0.1 per frame, outperforming existing approaches. Furthermore, an automatic crack detection system employing deep learning and image processing techniques is proposed in a different study [3]. Various CNN architectures, including MobileNetV2, ResNet101, VGG16, and InceptionV2, are explored for crack classification, with MobileNet achieving the highest detection accuracy of 99.59% after 10-fold cross-validation, surpassing other CNN models and existing crack detection methods.

Moreover, predicting rail crack length propagation using machine learning, particu-

doi:10.6342/NTU202403413

larly Recurrent Neural Networks (RNNs), is investigated in a study [4]. Real data from the French rail network is collected, and a Bayesian multi-horizons recurrent architecture is designed to model crack propagation, outperforming traditional methods like LSTM and GRU. Additionally, an image-based crack detection method for concrete structures utilizing Recurrent Neural Network (RNN) classification is presented in another study [5]. It achieves high accuracy (98.7%), precision (98.5%), and recall (99.5%) with a computation time of 0.467 seconds, surpassing existing methods. Finally, a crack detection method for concrete tunnel surfaces employing deep Convolutional Neural Networks (CNNs) and heuristic post-processing techniques is introduced in a study [6]. Demonstrating high accuracy with minimal execution time and hardware resources, the approach shows superiority over existing methods and holds promise for autonomous tunnel-inspection robots.

With rapid advancements in hardware technology, unmanned aerial vehicles (UAVs) have increasingly been employed in the field of crack detection. In recent years, numerous studies have focused on leveraging UAVs for this purpose, aiming to enhance the efficiency and accuracy of structural inspections. Beginning with the development of a UAV-based system for identifying building cracks, researchers aimed to address the time-consuming and risky nature of manual inspections, especially for large-scale structures like skyscrapers [7]. Similarly, DenxiDeepCrack, a novel method utilizing UAV imagery for automated road crack detection, was introduced in another study [8]. This approach, complemented by the UCrack 11 dataset, showcases the potential of UAV technology in streamlining crack detection processes. Addressing challenges in crack detection from UAV-collected images, the Crack Central Point Method (CCPM) is proposed in a different study [9]. This method offers improved adaptability and accuracy compared to traditional methods, demonstrating a significant leap forward in crack identification tech-

doi:10.6342/NTU202403413

niques. Moreover, a UAV-assisted bridge inspection methodology leveraging deep learning for efficient data processing is proposed in another study [10]. Demonstrated on various structures like the Skodsberg bridge in eastern Norway, this methodology showcases its potential in improving infrastructure maintenance practices. In addition, a UAV-based approach for bridge crack recognition, achieving high precision and speed, is introduced in a study [11]. By combining the R-FCN network and Haar-AdaBoost, this method demonstrates the effectiveness of UAV technology in crack detection. Further contributing to the field, a UAV-based method for accurate concrete crack detection and quantification is introduced in another study [12]. With operational feasibility demonstrated through on-site field detection, this method showcases the practical application of UAV technology in structural inspections. Moreover, a UAV road crack detection algorithm using MUENet is proposed in a study [13], effectively addressing challenges in crack morphology and highlighting the adaptability of UAV technology in diverse inspection scenarios. Finally, a Deep Learning approach for bridge crack detection using UAVs is proposed in another study [14]. With the introduction of stable imaging distance selection and verification, this method further enhances the capabilities of UAVs in crack detection tasks.

While these studies demonstrate significant progress in UAV-based crack detection, they often overlook edge computing elements necessary for achieving autonomous flight. This gap underscores the need for further integration of edge computing technologies to enable fully autonomous UAV operations in structural inspection and crack detection tasks.

doi:10.6342/NTU202403413

1.3 Research Objectives

Through the literature review, it has been noted that there are numerous studies in the field of civil engineering focusing on the utilization of drones and machine learning for crack detection and capture. However, most of these studies primarily emphasize data collection using drones, followed by subsequent data processing and model testing upon returning the data to the ground. Nevertheless, this approach still requires a considerable amount of time and human resources, and the use and development of drones are also subject to certain limitations.

To address these challenges and make breakthroughs in this field, this research has several objectives: Firstly, we aim to achieve edge computing on drones for real-time processing, enabling valuable resources to be brought back to the ground station. This will help reduce the time required for subsequent data processing and improve the efficiency of crack detection. Secondly, we aim to achieve automated crack capture through model training techniques and software-hardware integration. Through machine learning and deep learning technologies, our system will be able to autonomously identify and locate structural cracks, reducing human intervention and improving accuracy. Notably, we have extended the model training methods proposed by my senior in his thesis and applied them on the drone platform[15]. Finally, we plan to conduct comprehensive field tests on drones to validate the effectiveness and practicality of the system. This will include tests under different environmental conditions to ensure the robustness and adaptability of the system. These objectives are aimed at streamlining the crack detection process, reducing human intervention, and improving the efficiency and effectiveness of drone-based crack detection methods. With these breakthroughs, we hope to assist structural engineers in

doi:10.6342/NTU202403413

achieving greater efficiency and reducing labor costs during crack detection in the future. Especially when dealing with large areas and linear cracks, our framework can capture them accurately and efficiently.

1.4 Contribution and Scopes

The contribution of this work is succinctly introduced as follows.

- **Establishment of an open-source UAV system**: This study pioneers the development of an open-source UAV system tailored for crack detection applications. Integrating cutting-edge hardware and software technologies, this system offers a novel and accessible solution to address the challenges of crack detection in civil engineering.
- **Integration of Nvidia's Jetson Nano edge computing device**: The integration of Nvidia's Jetson Nano edge computing device onto the UAV platform represents a significant advancement in computational capabilities. This enhancement enables real-time processing of complex algorithms, facilitating efficient crack detection and image analysis.
- **Implementation of reinforced deep learning for crack identification**: Leveraging reinforced deep learning methodologies, this study empowers the UAV system to autonomously identify and localize structural cracks. By training the system to recognize crack patterns and propagation directions, it achieves a high level of accuracy in crack detection, reducing the reliance on manual intervention.
- **Achievement of autonomous UAV flight**: Utilizing the MAVLink communication

doi:10.6342/NTU202403413

protocol, this study enables seamless communication between the Jetson Nano and the UAV, facilitating autonomous flight operations. This capability streamlines the crack detection process, improving efficiency and accuracy while reducing human intervention.

In this research, we will establish an open-source UAV system. Regarding hardware, we will integrate Nvidia's Jetson Nano edge computing device onto our UAV, utilizing AVIX Corporation's UAV platform. For software control, we will build upon the model training techniques, employing deep reinforcement learning to enable the computer to identify the direction of crack propagation. Subsequently, we will utilize the MAVLink communication protocol to facilitate prompt transmission of instructions from the Jetson Nano to the lower-level UAV.

doi:10.6342/NTU202403413

Chapter 2 Methodology

In this chapter, our focus revolves around delineating the intricate process of achieving automated crack detection through the innovative utilization of drones. A pivotal component in this endeavor is empowering drones with the capacity to autonomously discern the direction of cracks. To this end, Section 2.1 elucidates our methodology of training a sophisticated deep reinforcement learning model. This model plays a pivotal role as it enables drones to make informed decisions regarding their maneuvering strategies when detecting cracks. Following this, once the outcomes are discerned, the seamless transmission of flight commands to the drones becomes imperative. Hence, Section 2.2 intricately explores the utilization of the MAVLink communication protocol. This protocol facilitates the transmission of non-control commands to the drones, ensuring efficient reception and execution of instructions. Furthermore, to culminate this intricate process, Section 2.3 delves into the intricacies of software and hardware integration. This encompasses the seamless amalgamation of onboard computers, drones, and relevant accessories, meticulously choreographing the operational workflow. This comprehensive integration lays the groundwork for conducting actual flight tests, poised to be expounded upon in the ensuing chapter. Figure 2.1 serves as a visual aid, encapsulating and elucidating our meticulous approach towards achieving automated crack detection using drones.

doi:10.6342/NTU202403413

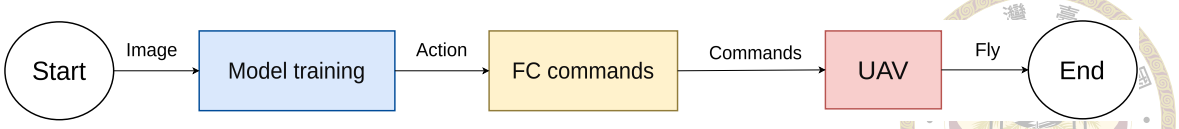


Figure 2.1: Overview of the Proposed Methodology

2.1 Model Training

In this section, we will provide a detailed explanation of how we train our machine learning models to detect crack propagation direction from localized crack images. Firstly, we will discuss the datasets we utilize and how we establish an effective training environment. Next, we will delve into how we employ image segmentation models for crack segmentation. Following that, we will explore our training methodology and the specifics of our deep reinforcement learning algorithm. Finally, we will showcase the results of our model training and discuss how these results are applied to our drone platform. Additionally, Figure 2.2 illustrates the process architecture from training on the server to running on the onboard computer. Our approach builds on the foundational work of previous researchers, specifically extending the methodologies established by our predecessors [15]. This continuity ensures that our model benefits from prior insights while introducing innovations tailored to the unique requirements of crack propagation detection.

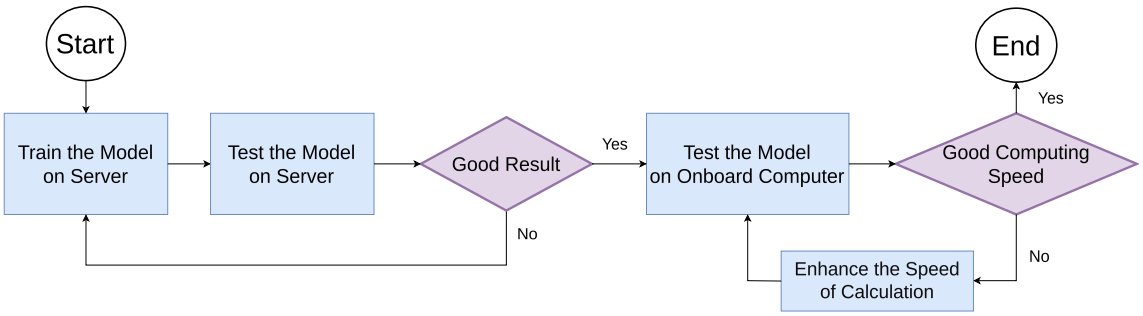


Figure 2.2: The process architecture from server training to onboard computer execution.

doi:10.6342/NTU202403413

2.1.1 Datasets

This study utilizes the DeepCrack dataset established by Liu et al.[16]. This dataset comprises 537 RGB crack images, each with a resolution of 544×384 pixels, along with their corresponding crack segmentation images, shown in Figure 2.3. The images within the dataset exhibit a wide variety of conditions, encompassing asphalt and concrete cracks in various scenarios. The shapes of the cracks range from linear to branched and grid-like. Such diverse data not only enriches our dataset but also aids in constructing a simulation environment that closely resembles real-world scenarios. To facilitate subsequent tasks such as crack segmentation and the establishment of a robust deep learning environment, we undertake preprocessing steps on all crack images, resizing them to a uniform size of 384×384 pixels. This standardized size serves a dual purpose, not only as the visual environment for the robot during model training but also as a baseline for consistent data representation across the dataset. Before delving into crack segmentation, we further partition the 384×384 environment into 16 distinct regions, each measuring 96×96 pixels. These smaller regions emulate the observable field of view for the robot, aiding in spatial awareness and motion planning during training simulations.(Figure 2.4)

This deliberate partitioning strategy yields manifold benefits. Firstly, it provides a more granular perspective of the robot's surroundings, facilitating a detailed analysis of crack patterns and their spatial distribution. Secondly, it enables the generation of a larger volume of training data by effectively disaggregating the original dataset into smaller, more numerous segments. As a result, the dataset expands from its initial 537 images to a significantly larger set comprising 8592 images. Within this augmented dataset, 4320 images are allocated for training, 480 for validation, and 3792 for testing, ensuring a robust

doi:10.6342/NTU202403413

evaluation framework for our models. In addition to data preprocessing, we apply augmentation techniques to further enhance the diversity and richness of our dataset. These techniques include rotation, horizontal flipping, and vertical flipping, which introduce variations in orientation and perspective. By exposing the model to a broader spectrum of crack configurations and orientations, these augmentations foster robust feature learning and reduce overreliance on specific crack orientations, thereby enhancing the model's generalization capabilities.

In summary, our meticulous approach to data preprocessing, partitioning, and augmentation lays a solid foundation for subsequent crack segmentation tasks and deep learning model training. By optimizing the representation and diversity of our dataset, we aim to equip our models with the requisite knowledge and adaptability to effectively detect and analyze cracks in diverse real-world environments.

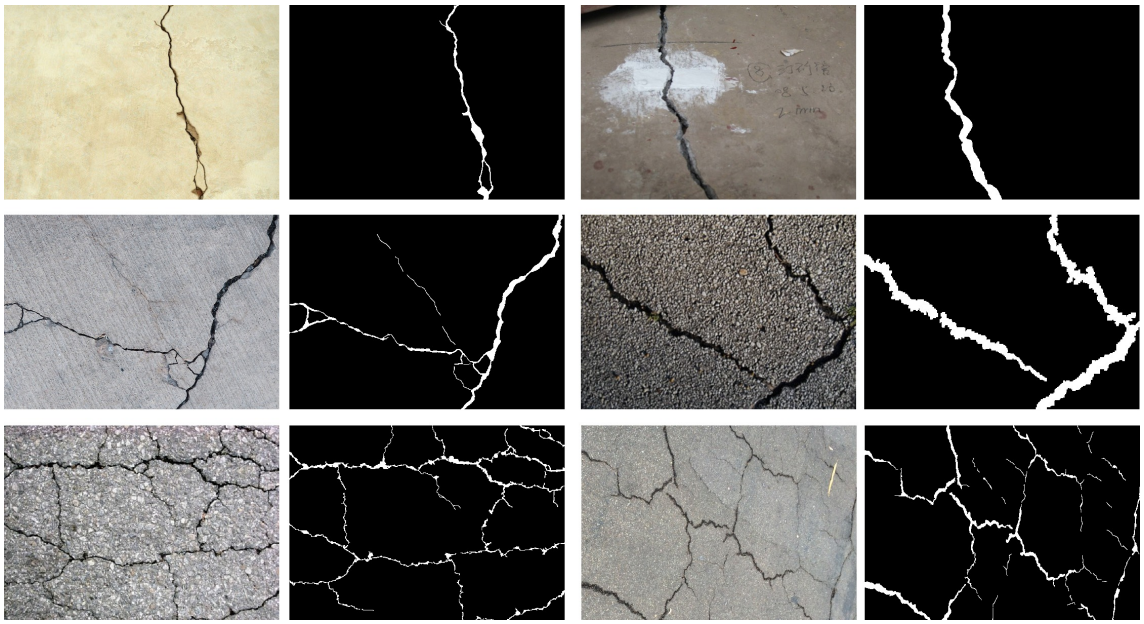


Figure 2.3: The photos in the DeepCrack dataset include cracks in asphalt and concrete, with each crack photo having its corresponding crack segmentation image. This helps us to use them as ground truth for training purposes.

doi:10.6342/NTU202403413

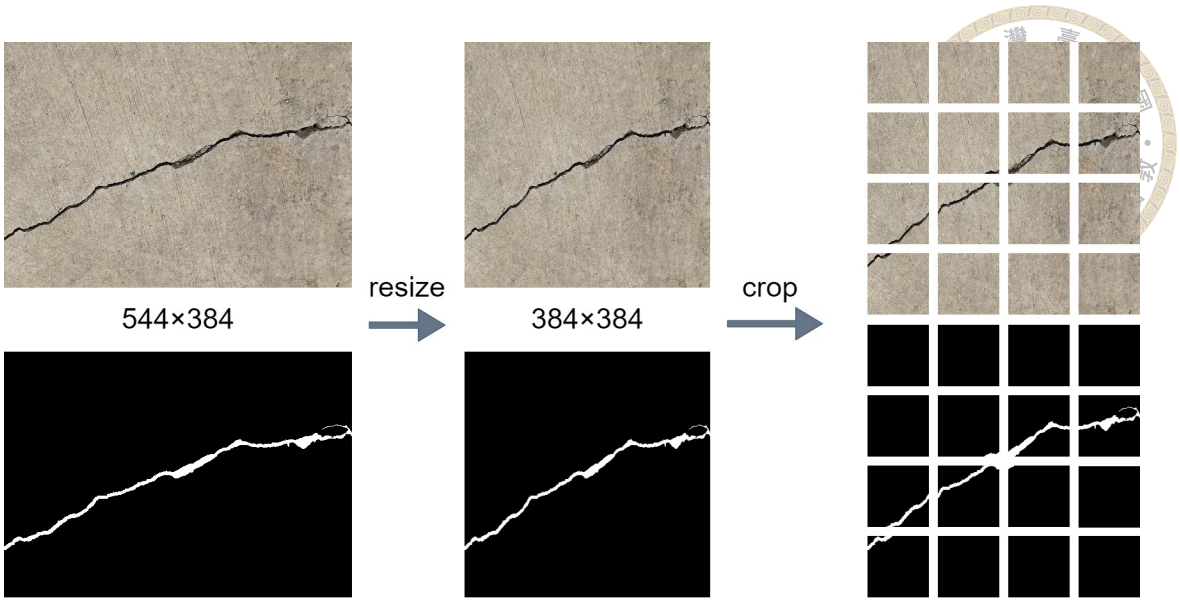


Figure 2.4: Data preprocessing: The original images, initially sized 544×384, are resized to 384×384 to be used in the simulation environment. Following this, the 384×384 images are divided into 16 images, each measuring 96×96, which serves as the viewing units for the robot.

2.1.2 Segmentation

Many studies utilize crack detection methods [17], often employing bounding boxes to identify the regions of interest for subsequent analysis. However, this approach only allows us to determine the presence of cracks within a given area, limiting our ability to assess the direction in which the cracks extend. This is why we chose to use segmentation in our current study. By segmenting the entire crack line, we can subsequently apply deep reinforcement learning techniques to enable a robot to navigate within the segmented area and learn to identify the direction of crack propagation. In order to accurately locate and understand the positions and orientations of cracks, we must perform crack segmentation on our processed RGB images. During this process, areas containing cracks are segmented and displayed in white, while crack-free backgrounds are displayed in black. This allows the computer to clearly understand the locations of cracks, enabling subsequent training and testing tasks. This process provides a crucial foundation for subsequent analysis, en-

doi:10.6342/NTU202403413

suring accurate perception of cracks and thereby ensuring the accuracy and reliability of subsequent tasks. In this segmentation task, we employed the U-Net-ResNet34 model [18][19], which combines the strengths of both U-Net and ResNet architectures. This amalgamation enhances the overall performance. Next, we will provide brief introductions to U-Net and ResNet, followed by highlighting the differences between them and our model.

2.1.2.1 U-Net

U-Net, introduced by Ronneberger et al. in 2015 [20], represents a pioneering convolutional neural network (CNN) architecture primarily employed in biomedical image segmentation applications. Its distinctive structure comprises two interconnected components: an encoder and a decoder. The encoder component serves to distill essential features from input images using a sequence of convolutional and pooling layers, effectively capturing hierarchical representations of the image. Conversely, the decoder component aims to recover spatial details lost during the encoding process by employing upsampling techniques and integrating feature maps from the encoder through skip connections. The architecture of U-Net can be visualized in Figure 2.5. One of the salient attributes of U-Net lies in its remarkable ability to precisely localize objects within images while preserving intricate details. This capability is facilitated by the integration of skip connections, which facilitate the seamless transfer of high-resolution information from the encoder to the decoder. Consequently, U-Net excels in segmenting fine structures such as cracks, ensuring accurate delineation even in scenarios involving subtle features. However, it is pertinent to acknowledge that U-Net may encounter challenges in capturing broader contextual information, potentially limiting its performance in tasks necessitating a holistic understanding

doi:10.6342/NTU202403413

of the image. Despite its inherent limitations, U-Net has garnered widespread adoption and adaptation across diverse segmentation tasks owing to its efficacy and elegant simplicity in design. Its proven track record in achieving superior segmentation results, particularly in biomedical imaging contexts, underscores its enduring relevance and appeal in the realm of deep learning-based image analysis.

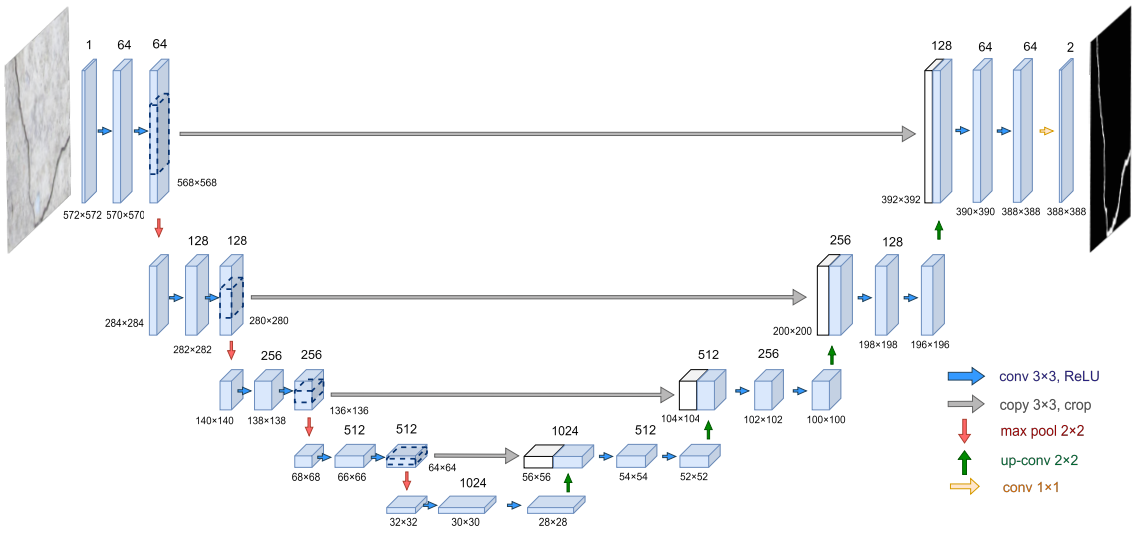


Figure 2.5: U-Net architecture

2.1.2.2 ResNet

ResNet, proposed by He et al. in 2015 [21], stands as a cornerstone convolutional neural network (CNN) architecture prominently utilized for image classification tasks. At its essence lies the innovative concept of residual learning, which addresses the challenge of training exceedingly deep networks by circumventing the vanishing gradient problem. ResNet achieves this by introducing residual blocks, which allow layers to learn residual functions with respect to the input, shown in Figure 2.6. These blocks consist of multiple convolutional layers, followed by a shortcut connection that adds the original input to the output of the convolutional layers. This mechanism facilitates smoother training and enables better performance in deeper networks by alleviating the degradation prob-

doi:10.6342/NTU202403413

lem encountered in traditional deep networks. This novel approach significantly enhances ResNet's efficacy in capturing global features and excelling in tasks necessitating a holistic understanding of the entire image. However, while ResNet demonstrates exceptional prowess in image classification tasks, it may not excel to the same extent as U-Net in tasks requiring pixel-level precise localization and detail capturing. The distinctive ability of ResNet to allow layers to learn residual functions with respect to the input plays a pivotal role in mitigating the challenges associated with training deep networks, making it a preferred choice for a wide array of deep learning applications. Despite its potential limitations in certain localization tasks, ResNet remains a highly acclaimed and widely adopted CNN architecture due to its outstanding performance and versatility. Its remarkable ability to effectively tackle challenges in training deep networks has propelled its widespread application across various domains, cementing its status as one of the most influential contributions to the field of deep learning.

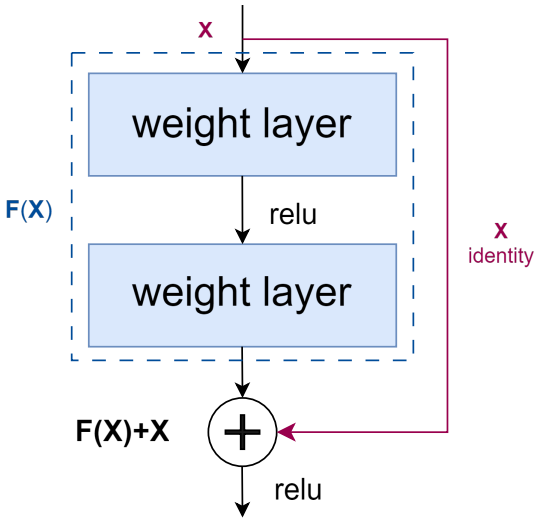


Figure 2.6: Residual Block

doi:10.6342/NTU202403413

2.1.2.3 U-Net-ResNet34

In the preceding section, we have understood that U-Net possesses powerful localizing capabilities, accurately pinpointing objects in images while preserving detailed information. However, it may have limitations in capturing global context, resulting in decreased performance in tasks requiring a comprehensive understanding of the entire image. Conversely, ResNet effectively captures global features, demonstrating excellent holistic understanding capabilities, although it may not excel as much as U-Net in pixel-level precise localization and detail capturing. Therefore, to fully leverage the strengths of both architectures, we adopted a dual architecture, combining U-Net's localizing capability with ResNet's global contextual understanding. Through this amalgamation, our model not only accurately locates cracks but also captures contextual information surrounding the cracks, enhancing segmentation accuracy and clarity.

In this study, we employed the U-Net-ResNet34 model(Figure 2.7), which is included in the open-source Python library created by Yakubovskiy[22]. The model takes 96×96 RGB images as input and outputs 96×96 crack segmentation images. The model begins with a 7×7 convolutional layer, followed by a 2×2 max-pooling layer. The encoder comprises two types of convolutional blocks: ResNet basic blocks and ResNet downsampling blocks. Inside the ResNet basic blocks are two 3×3 convolutional layers with shortcut connections, enabling the direct propagation of features across layers. The downsampling blocks reduce image resolution by increasing stride. The decoder retains the original U-Net structure, consisting of five upsampling blocks, each containing a 3×3 convolutional layer followed by two 2×2 transpose convolutions. The model integrates four skip connections to transmit high-resolution features to each upsampling layer. The final layer is

doi:10.6342/NTU202403413

a 1×1 convolutional layer with softmax activation, yielding pixel-level prediction maps.

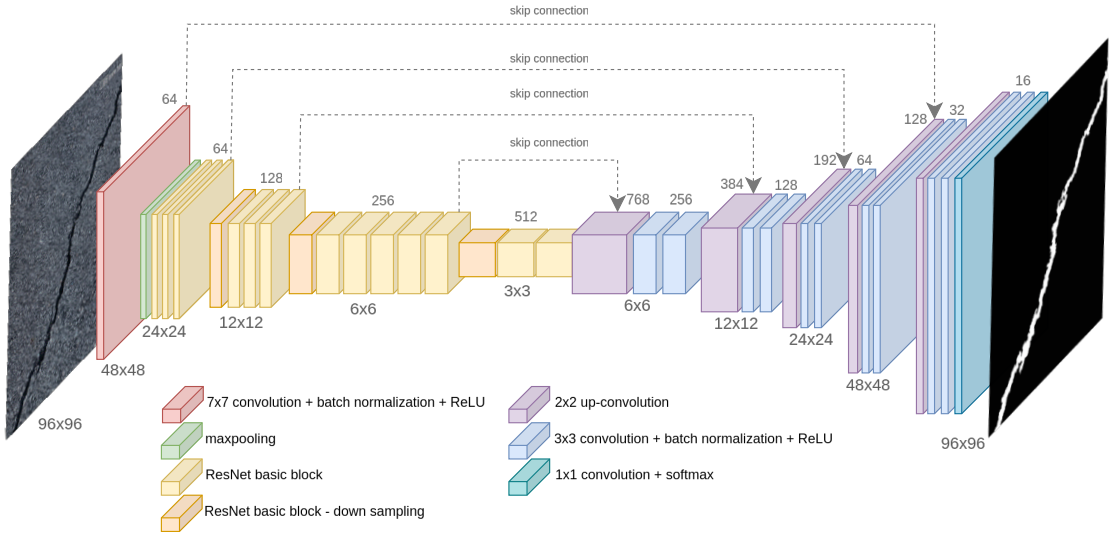


Figure 2.7: U-Net-ResNet34 model architecture, where the numbers below each layer represent the dimensions of the input image and feature maps, and the numbers above indicate the number of channels in that layer.

2.1.2.4 Segmentation Result

By utilizing the model architecture from Section 2.1.2.3, we can effectively segment images with cracks during both the training and testing phases. This segmentation process is crucial for accurately identifying and analyzing cracks in various types of surfaces. As discussed in Section 2.1.1, the input images used are 384x384 pixels. This size allows for detailed analysis and ensures that even small cracks can be detected. However, our approach also allows for the segmentation of areas smaller than 384x384 pixels, providing flexibility and precision in crack detection. During the training and testing processes on the server, we will primarily focus on segmenting images of 384x384 and 96x96 sizes. These specific sizes have been chosen to balance computational efficiency with the need for high-resolution analysis. The results of our crack segmentation on the server are illustrated in Figure 2.8. These results demonstrate the effectiveness of our approach in accurately identifying and segmenting cracks. In the subsequent Section 2.1.5.1, we will delve into

doi:10.6342/NTU202403413

the pixel sizes used for segmentation on the UAV. This discussion will include a detailed explanation of why these specific sizes were chosen, considering factors such as the UAV's operational constraints and the need for real-time processing. Understanding these reasons will provide a comprehensive view of the practical applications and limitations of our segmentation approach in a UAV context.

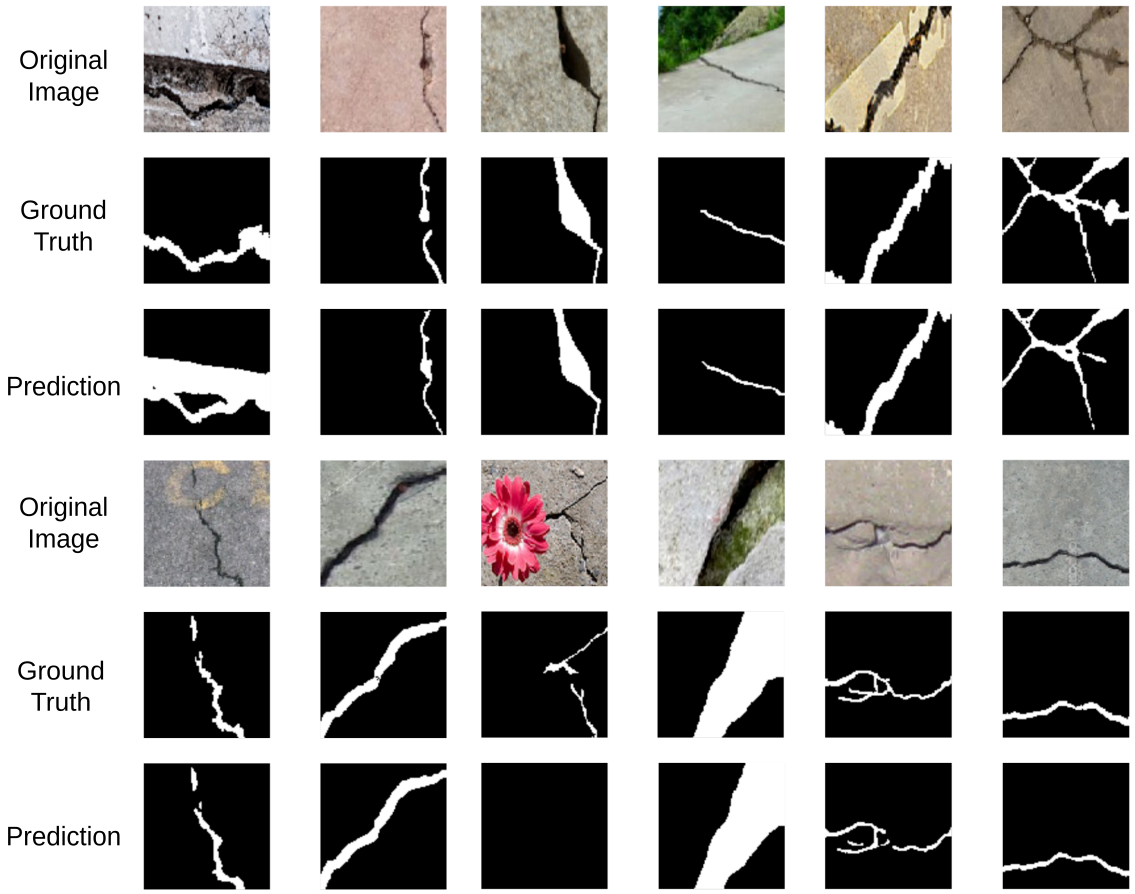


Figure 2.8: The results of crack segmentation using the U-Net-ResNet34 model are shown below from top to bottom: original image, ground truth, segmented result. Through this model, the segmentation results allow us to clearly identify the location and shape of cracks during subsequent training and testing processes.

2.1.3 Deep Reinforcement Learning

In the previous section, we successfully achieved the goal of crack segmentation. Our robot can now accurately identify the location and presence of cracks in images. In this section, we will utilize these segmented crack images to train the robot to determine

doi:10.6342/NTU202403413

the direction of crack extensions based on partial crack images and to identify blocks containing cracks as much as possible. To accomplish this, we are employing a Deep Double Q-learning (DDQN) model, which will assist us in training the robot to handle more complex scenarios.

Next, we will briefly introduce Deep Reinforcement Learning, Q-learning, Deep Q-learning, and the Double Deep Q-learning method we are utilizing. These concepts will provide readers with the necessary background knowledge to better understand our training process and results.

2.1.3.1 Deep Reinforcement Learning

In this section, we will comprehensively explore Deep Reinforcement Learning (DRL), a powerful paradigm in the field of artificial intelligence that enables machines to make decisions by interacting with the environment. Before delving into the next section on Q-learning, let's first establish a thorough understanding of Deep Reinforcement Learning. Deep Reinforcement Learning is a machine learning approach that combines reinforcement learning algorithms with deep learning techniques. It empowers agents to learn how to make decisions by interacting with the environment to maximize expected long-term rewards. Unlike supervised and unsupervised learning, agents in Deep Reinforcement Learning learn through interaction with the environment without explicit guidance. To better understand Deep Reinforcement Learning, let's break down its core components: the agent, environment, actions, and rewards.

- **Agent**: The entity responsible for taking actions in the environment, aiming to maximize expected total reward.

doi:10.6342/NTU202403413

- **Environment**: The external system in which the agent operates, providing feedback in response to the agent's actions.
- **Actions**: The operations or decisions that the agent can take at each time step to influence the environment's state.
- **Rewards**: Feedback signals from the environment after the agent takes actions, used to evaluate the quality of actions.

In our training process, we will have corresponding elements such as agent, environment, action, and rewards. The environment used this time is the previously mentioned 384x384 crack segmentation image, as shown in Figure 2.9. The agent is our protagonist who will decide which direction to capture the crack based on the local crack images observed. The action is the direction he chooses; in this study, we will provide him with eight directions to choose from: up, down, left, right, as well as upper-left, lower-left, upper-right, lower-right, as shown in Figure 2.9. As for rewards, we will define that if the chosen direction leads to the appearance of new cracks in the image, he will be rewarded; otherwise, no reward will be given. With these defined elements, coupled with the model introduced in the next section, we can train an automated robot to capture cracks automatically, as shown in Figure 2.10. It's worth noting that in each step, the agent will move 48 pixels (half of the visible range, which is 96x96). The benefit of this overlap is that we also set constraints to prevent the agent from moving backward in subsequent steps. In conclusion, Deep Reinforcement Learning is a powerful machine learning approach that enables agents to learn optimal decision-making strategies through interactions with the environment. In the following sections, we will delve into specific DRL algorithms and their applications in crack detection and extension prediction tasks.

doi:10.6342/NTU202403413

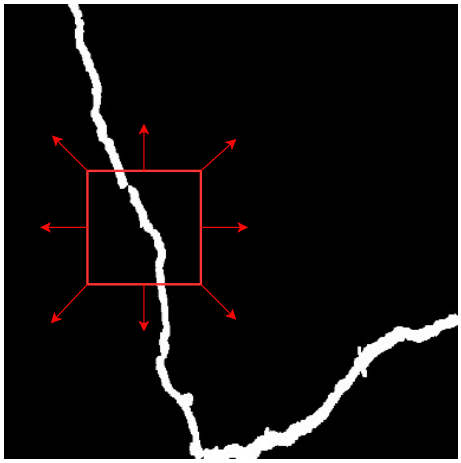


Figure 2.9: The environment is a 384x384 crack segmentation image, and the red arrows represent the eight possible actions that the agent can choose from.

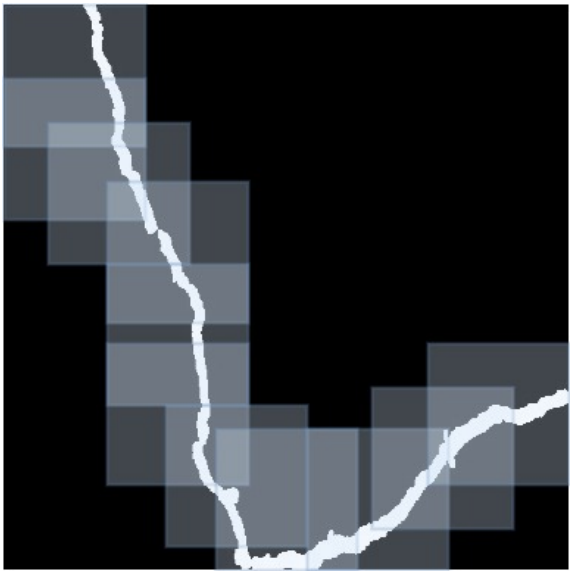


Figure 2.10: The light-colored box represents the size of the area visible to the robot, which is 96x96. The diagram illustrates the path of automated crack exploration by the robot.

2.1.3.2 Q-learning

Q-learning[23] is a model-free, off-policy reinforcement learning algorithm. Reinforcement learning comprises five components: the agent, states, actions, rewards, and the environment. Model-free means that the agent in Q-learning is not trained as a model to predict future states based on probability distributions. Its objective is to learn an optimal value function within the environment, maximizing the agent's cumulative rewards.

doi:10.6342/NTU202403413

Off-policy means that the agent learns from the experiences of the previous policy rather than the current one. Q-learning utilizes the Bellman equation[24] to update its policy. The equation is as follows:

$$\text{NewQ(s,a)} \leftarrow \text{Q(s,a)} + \alpha \cdot [\text{R(s,a)} + \gamma \cdot \max \text{Q'}(s', a') - \text{Q(s,a)}] \tag{2.1}$$

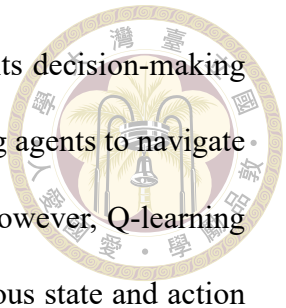
Here, s represents the state, s' the next state, a the action, a' the next action, R the reward, γ the reward discount factor, and α the learning rate. This equation uses temporal difference (TD) to update Q-values, representing the difference between current and new estimations. The current estimation is $Q(s, a)$, and the new estimation is $R(s, a) + \gamma \cdot \max \text{Q'}(s', a')$, which includes the reward and the maximum estimation of the next state. γ is a parameter between 0 and 1 used to adjust the influence of expected future rewards on Q-values. One of the strategies for the agent to select its actions is the epsilon-greedy strategy:

$$A_t = \begin{cases} \text{argmax Q}(\text{s}_t,\text{a}_t) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases} \tag{2.2}$$

Initially, ϵ is often set to a higher probability to allow the agent more chances to choose random actions to explore the environment. As training progresses, ϵ should gradually decrease, allowing the agent to exploit the environment by selecting the optimal action for each state.

In practical applications, traditional Q-learning utilizes a Q-table to store state-action pairs and their corresponding Q-values. The agent iteratively updates and refines this ta-

doi:10.6342/NTU202403413



ble through interactions with the environment, gradually enhancing its decision-making capabilities. This iterative learning process is instrumental in enabling agents to navigate complex environments and develop effective strategies over time. However, Q-learning has limitations, particularly in handling high-dimensional or continuous state and action spaces. As the number of states and actions increases, maintaining a Q-table becomes impractical due to its size and memory requirements. Additionally, Q-learning may struggle with environments where rewards are sparse or delayed, as it relies on immediate rewards to update Q-values effectively. By understanding these principles and limitations, Q-learning provides a robust framework for developing intelligent agents capable of autonomous decision-making across diverse real-world scenarios. Efforts in reinforcement learning continue to address these challenges through advanced algorithms and techniques tailored to specific application domains.

2.1.3.3 Deep Q-learning

Deep Q-Learning addresses the primary challenge of Q-learning arising from the limited memory of the Q-table, particularly in handling high-dimensional state spaces. In large state spaces, memory saturation can occur quickly, resulting in inefficient training and potential hindrance to completion. DeepMind introduced the Deep Q-Learning algorithm (DQN)[25] to tackle this issue. DQN approximates the optimal value function using a Convolutional Neural Network (CNN), which, due to its fixed parameter count, does not suffer from memory limitations. DQN takes the state as input and outputs Q-values for all actions. Similar to Q-learning, DQN also updates its network using the Bellman equation[24]. However, instead of using the Q-table, DQN uses gradient methods to optimize the loss function. The loss function is defined as:

doi:10.6342/NTU202403413

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{2.3}$$

Here, y_i represents the maximum expected value, $Q(s,a;\theta_i)$ denotes the current estimation output by the neural network, and $\rho(s,a)$ is the probability distribution over sequences s and actions a. In addition to the deep neural network, DQN incorporates the following key features:

- **Experience Replay**: DQN employs experience replay to learn from offline policies. Each time the agent interacts with the environment, its experience (state, action, reward, next state) is stored in an experience replay pool with a fixed capacity. When the pool reaches its limit, the oldest experiences are replaced by new ones. Instead of using recent experiences, the algorithm randomly samples a batch of experiences from the replay pool when updating the neural network. This random sampling helps break the temporal correlation between consecutive experiences, reducing the risk of the agent getting stuck in local minima or oscillations during training. Reusing experiences also allows the agent to learn more from each interaction, better utilizing the available data.

- **Epsilon-Greedy Exploration Strategy**: DQN also utilizes an epsilon-greedy strategy to explore the environment. The decay rate of ϵ can be arbitrarily defined. In the DQN paper, a linear decay from 1 to 0.1 was chosen, while in this paper, we opt for an exponential decay function.

- **Target Network**: DQN comprises two neural networks: the online network and the target network. The online network interacts with the environment to approximate the optimal value function, while the target network, with the same initial weights

doi:10.6342/NTU202403413

and structure as the online network, does not interact with the environment. Instead, it periodically updates its weights by copying the weights of the online network. The target network is used to estimate the maximum expected value, stabilizing learning by providing more consistent Q-values during updates. Without the target network, the Q-values used to update the online network would fluctuate rapidly, leading to oscillations or divergence in the learning process.

2.1.3.4 Double Deep Q-learning

In this section, we delve into the theoretical underpinnings and architecture of the Double Deep Q-Network (DDQN). Inspired by the seminal work of Hado van Hasselt [26], we acknowledge a common challenge faced by Deep Q-Networks (DQNs): the tendency to overestimate Q-values, leading to suboptimal action selection and performance inconsistencies across different environments. To address this, the Double DQN (DDQN) algorithm introduces a fundamental modification to the conventional DQN framework, separating the action selection and evaluation processes during target value computation.

$$Y_t^{DQN} = R_{t+1} + \gamma \max Q(S_{t+1}, a; \theta_t^-) \tag{2.4}$$

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max Q(S_{t+1}, a; \theta_t), \theta_t^-) \tag{2.5}$$

In equation (2.4), Y_t^{DQN} is the target value for the DQN, R_{t+1} is the reward received after taking an action, γ is the discount factor, $Q(S_{t+1}, a; \theta_t^-)$ is the estimated Q-value for the next state S_{t+1} and action a, and θ_t^- represents the parameters of the target network.

doi:10.6342/NTU202403413

In contrast, equation (2.5), $Y_t^{DoubleDQN}$ shows that the DDQN target value uses the main network parameters θ_t to select the action that maximizes the Q-value in the next state, while the target network parameters θ_t^- are used to evaluate this action.

The model architecture of DDQN, depicted in Figure 2.11, consists of online and target networks with identical structures. While the online network actively interacts with the environment and undergoes updates using gradient descent methods, the target network periodically synchronizes its weights with the online network. This convolutional neural network (CNN) ingests 96 × 96 crack segmentation images from the U-Net-ResNet34 architecture as inputs and outputs Q-values for each possible action. Each convolutional block encompasses a 96 × 96 convolutional layer, normalization layer, and Rectified Linear Unit (ReLU) activation, followed by flattening and forwarding to a fully connected layer for further processing. Furthermore, our DDQN implementation incorporates additional strategies such as experience replay and epsilon-greedy exploration to enhance training stability and exploration efficiency. These strategies allow the agent to learn from past experiences and strike a balance between exploiting known information and exploring new possibilities in the environment. With a robust theoretical foundation and a well-designed architecture, our DDQN model is poised to undergo rigorous training, as showcased in the subsequent section, where we present and analyze the training results in detail.

doi:10.6342/NTU202403413

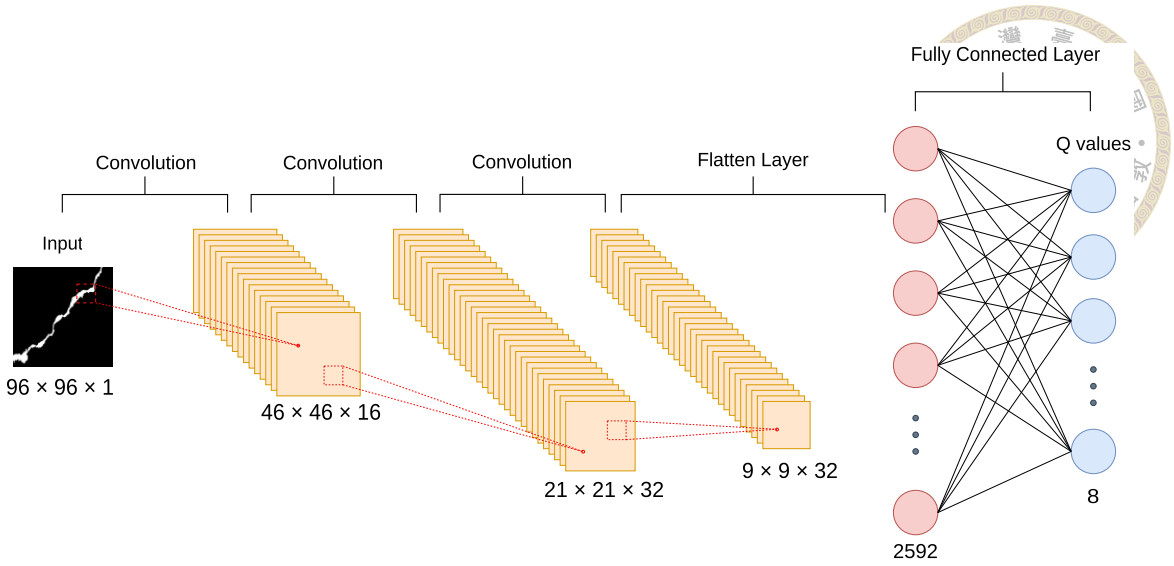


Figure 2.11: Model architecture of the deep double Q neural network for crack segmentation, with identical structures for the online and target networks. Dimensions and channel quantities indicated below. This network inputs segmented crack images and outputs Q-values for each action.

2.1.4 Training Result

In the previous sections, we have extensively discussed how to train our model to enable the robot to identify the locations of other cracks by observing local crack features. In this section, we will discuss how we define training results and what results we expect under our training conditions. First, let's introduce our criterion for assessing the effectiveness of crack detection, which is the capture rate. The capture rate (CR) is calculated as the ratio of the total number of pixels containing cracks that the robot successfully detects to the total number of pixels containing cracks in the entire environment. Mathematically, it can be expressed as:

$$\mathrm{CR} = \frac{\text{Total pixels containing detected cracks}}{\text{Total pixels containing cracks in the entire environment}} \tag{2.6}$$

Following the explanation, we present the crack capture rates for the four versions

doi:10.6342/NTU202403413

of DDQN in Table 2.1 and 2.2. Table 2.1 pertains to environments containing grid-like cracks, while Table 2.2 represents environments without grid-like cracks. In these tables, the DDQN algorithms are listed from top to bottom along with increasing levels of human-controlled factors.

Table 2.1: Average Capture Rates in Training and Testing Environments

Method	Training Capture Rate (%)	Testing Capture Rate (%)
DDQN	66	50
DDQN + Snake scan	72	59
DDQN + Snake scan + Non-repetitive	72	62
DDQN + Snake scan + Non-repetitive + Resample	71	66

Table 2.2: Average Capture Rates in Environments Without Grid-like Cracks

Method	Training Capture Rate (%)	Testing Capture Rate (%)
DDQN	69	51
DDQN + Snake scan	73	58
DDQN + Snake scan + Non-repetitive	73	68
DDQN + Snake scan + Non-repetitive + Resample	73	64

When training the pure DDQN model in the environment, the training results seem promising. However, upon applying the model to the testing environment, the capture rate significantly decreases, indicating overfitting. To address this issue, we introduce snake scanning (Figure 2.12) and selective memory. Selective memory excludes irrelevant features from learning, while snake scanning assists the agent in moving along a predefined path before detecting cracks. In more detail, when the robot is in search of cracks, it first moves along the predefined path, performing snake scanning. The purpose of this scanning pattern is to explore potential crack locations, even when cracks are not detected within the current field of view. If the robot observes cracks during this process, it immediately applies the DDQN algorithm to determine the best action. In other words, snake scanning provides a mechanism for the robot to extensively search potential crack areas before detecting cracks, thereby improving the efficiency and accuracy of crack detection. Results show a notable improvement in the agent's performance in the testing environment with the application of these methods, increasing the crack capture rate from

doi:10.6342/NTU202403413

50% to nearly 60%. Furthermore, by observing the predictions frame by frame, we notice that the agent sometimes gets stuck in repetitive loops. To tackle this problem, we restrict the agent from revisiting its previous state, resulting in a slight increase in the capture rate in the testing environment. Finally, we test the aforementioned methods in a resampled environment. The DDQN with snake scanning and non-repetitive actions achieves the highest crack capture rate, reaching 66% in the resampled environment. However, the resampled dataset does not yield the highest score in environments without grid-like cracks. We speculate that the resampled environment enables the agent to better learn environments with grid-like cracks, whereas agents trained in the original environment may struggle to effectively learn grid-like cracks. Example paths taken by the agent are illustrated in Figure 2.13, demonstrating that the agent can detect a significant portion of cracks by observing small regions.

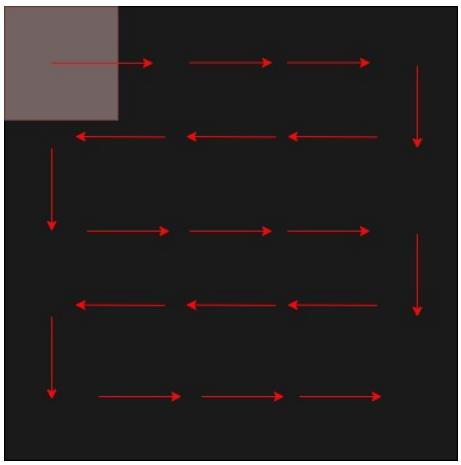


Figure 2.12: Snake Scanning: If the robot does not detect any crack segmentation in its 96×96 field of view, it will perform a snake scanning, as illustrated by the red arrows.

doi:10.6342/NTU202403413

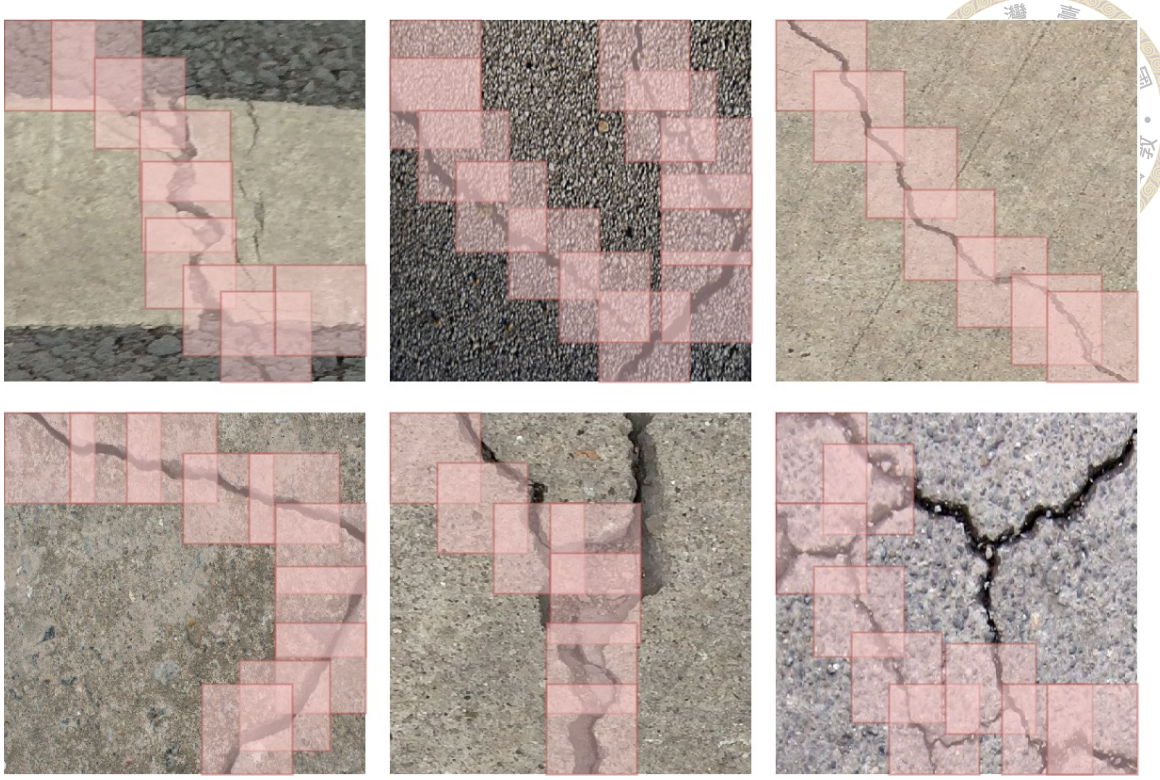


Figure 2.13: The displayed route chosen by the agent in the environment. The red box represents the 96×96 viewable area of the agent, starting from the top left corner for each test.

2.1.5 UAV Platform Implementation

In the previous section, we introduced how we successfully employed crack segmentation and deep reinforcement learning to enable a robot to automatically capture other cracks in the environment using partial crack images, demonstrating a certain level of success in capturing. Next, we will discuss how we achieved this on a drone. Firstly, we will compare the images from the drone with those of the environment we trained on, and explain how we input these images into the model to determine direction. Then, we will discuss the differences between the offboard (typically 384x384 pixels in a server environment) and onboard (on the drone) flowcharts throughout the entire process.

doi:10.6342/NTU202403413

2.1.5.1 UAV Image Input

In the Section 2.1.1, we mentioned that during the training process, the robot operates in an environment with a size of 384x384 pixels, while the observed image size is 96x96 pixels. Therefore, we can utilize the features of the cracks observed in the 96x96-sized images to determine how the robot should move. In our actual flying drone, its role corresponds to that of the robot during the training process, so the size of the observed images is also 96x96 pixels. However, in the drone hardware introduced (Section 2.3.2.2), we use the AXG-AS03F camera, which captures images at a size of 1280x720 pixels. Therefore, to ensure that the images comply with the requirements of our model for directional determination, we need to resize the drone images to 96x96 pixels before inputting them into the model. We will first resize the images to 384x384 and perform crack segmentation, as shown in Figure 2.14. After the segmentation is complete, the segmented images will be resized to 96x96 pixels and fed into the deep reinforcement learning model for direction determination, as shown in Figure 2.15. This approach allows us to more accurately segment the shape of the cracks while also ensuring the images are the appropriate size for the deep reinforcement learning training.

In the process of direction prediction on the drone, we found that we need to significantly reduce the pixel size to meet the specifications of the remote. This method is not the best because compressing pixels to a large extent not only deteriorates image quality but also leads to misinterpretation of some features by the computer, resulting in incorrect direction judgments or failure to detect cracks. However, we still chose to proceed with this approach for the following reasons: Firstly, if we don't compress the pixels, we would have to use the original size photos (1280x720) as the training set. Based on our

doi:10.6342/NTU202403413

standards, we used 8592 images of cracks sized 96x96 for training in this open-source dataset. Gathering the same quantity and quality of crack images sized 1280x720 would be a monumental task. Secondly, as shown in Figure 2.15, in most cases, even with compressed photos, the segmented images of cracks remain clear enough to determine their direction of extension. For these reasons, we have opted for this approach for the entire project. As for how to enhance it in the future, we can discuss that in the Section 5.2.

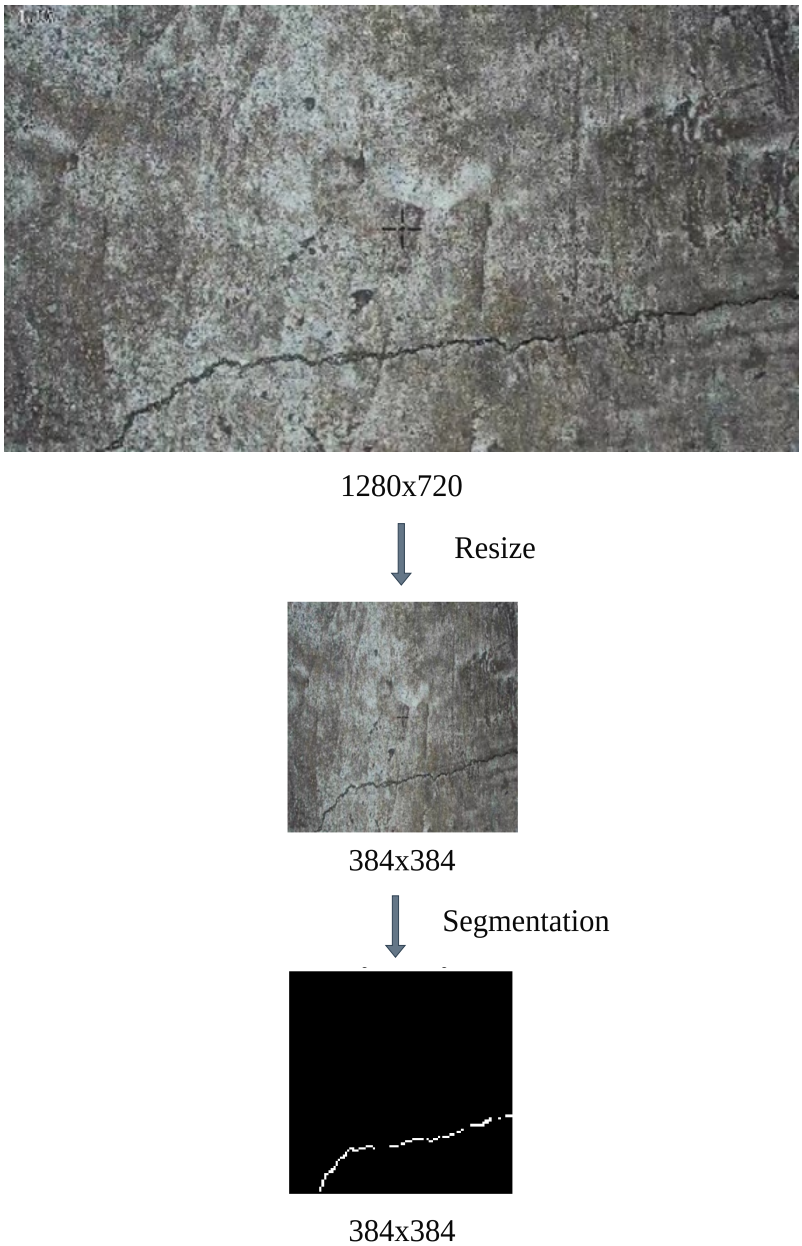


Figure 2.14: The pixels captured by the drone camera are 1280x720. They are converted into 384x384 pixels for crack segmentation.

doi:10.6342/NTU202403413

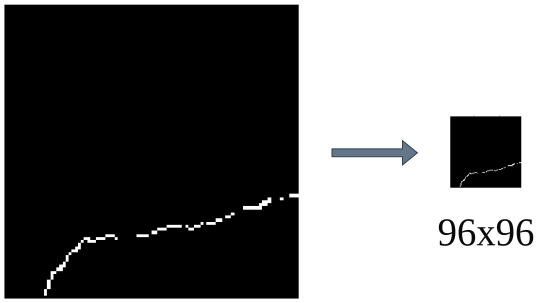


Figure 2.15: Before feeding them into the deep reinforcement learning model, the pixel size needs to be resized to 96x96.

2.1.5.2 Comparison of Offboard and Onboard

In this section, we will introduce the differences between the offboard environment, which is the training and testing environment on the server, and the onboard environment, which is the environment on the UAV, and how we adapt to these differences. Firstly, in the main training process and during offline testing, as depicted in Figure 2.16, we start by establishing a 384x384 environment as the space for robot movement. From this environment, we extract a 96x96 area, which serves as the field of view for the robot. This area is then fed into the U-Net-ResNet for segmentation. If no cracks are detected, the robot conducts regular raster scanning within the 384x384 space. If cracks are present, the process proceeds to DDQN for crack path prediction. Any movements made by the robot are reflected within the 384x384 space, initiating the next cycle.

In the onboard environment of the UAV, as shown in Figure 2.17, there are no defined 384x384 borders in the flying world of the drone. Therefore, its surroundings extend infinitely in all directions, making it impossible to conduct raster scanning in the absence of detected cracks. The camera's view is first compressed from 1280x720 to 384x384, then after crack segmentation, it is further compressed to 96x96 before being input into

doi:10.6342/NTU202403413

the DDQN for crack path prediction. This process yields a directional result and initiates the next cycle. If no cracks are detected during this process, we have a specific protocol to handle it, which will be detailed in Section 2.2. To better illustrate the relationship between the two frameworks, it can be more clearly understood through Figure 2.18.

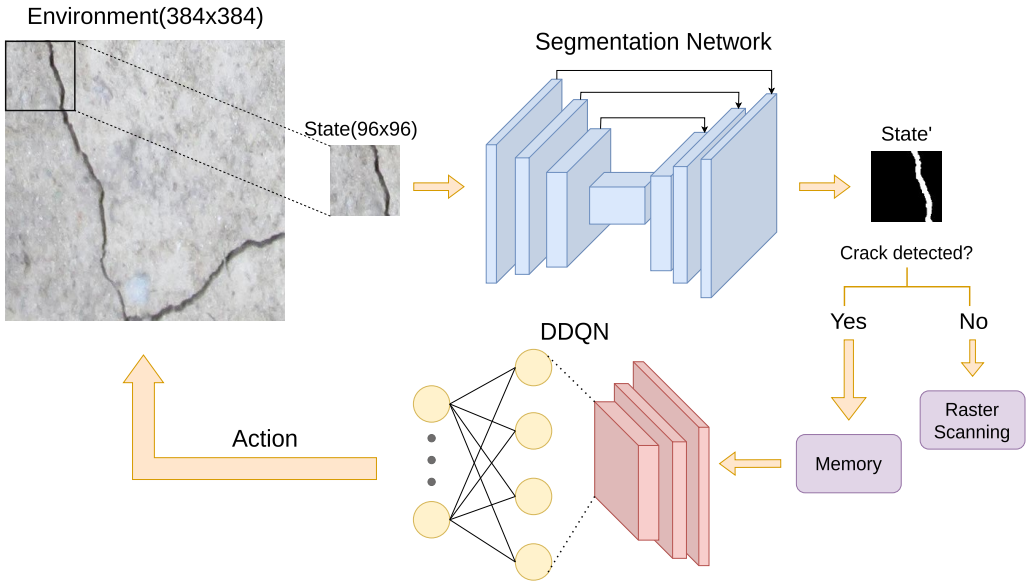


Figure 2.16: Offboard Flowchart

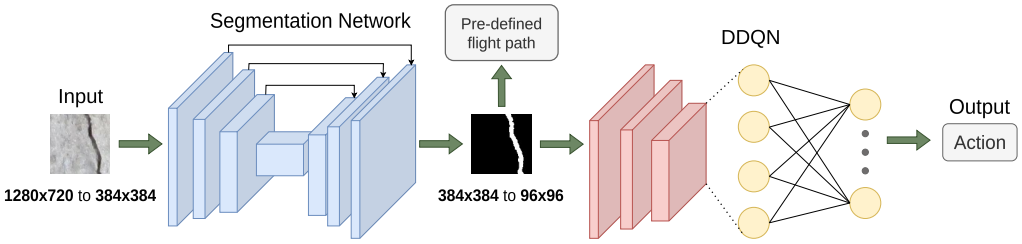


Figure 2.17: Onboard Flowchart

	resize 1	input to segmentation	output from segmentation	**resize 2**	input to DDQN	output from DDQN
offboard test	96x96	96x96	96x96	96x96	96x96	96x96
onboard test	1280x720 to 384x384	384x384	384x384	384x384 to 96x96	96x96	96x96

Figure 2.18: Comparison of Offboard and Onboard

doi:10.6342/NTU202403413

2.2 Flight Control Commands

In the previous section, we successfully developed a system that can identify localized cracks through the imagery captured by drones. This system combines a pre-trained deep reinforcement learning model, enabling the drone to determine the direction it should proceed based on these images, facilitating crack repair or monitoring. However, merely informing the drone of the judgment isn't sufficient to accomplish this task. We require a mechanism to convey these instructions to the drone and ensure it comprehends and executes them accurately.

To achieve this goal, we will introduce a communication protocol to facilitate communication between the ground control system and the drone. Here, we have chosen the MAVLink communication protocol, which is a lightweight protocol specifically designed for data exchange in unmanned aerial vehicle (UAV) systems. This communication protocol needs to support real-time, bidirectional data transmission to ensure the accuracy and timeliness of instructions. Through the MAVLink communication protocol, the ground control system can transmit the instructions generated by the deep reinforcement learning model to the drone and promptly receive feedback from the drone to ensure smooth mission execution. Additionally, Figure 2.19 illustrates the process and test environment architecture for writing flight control commands.

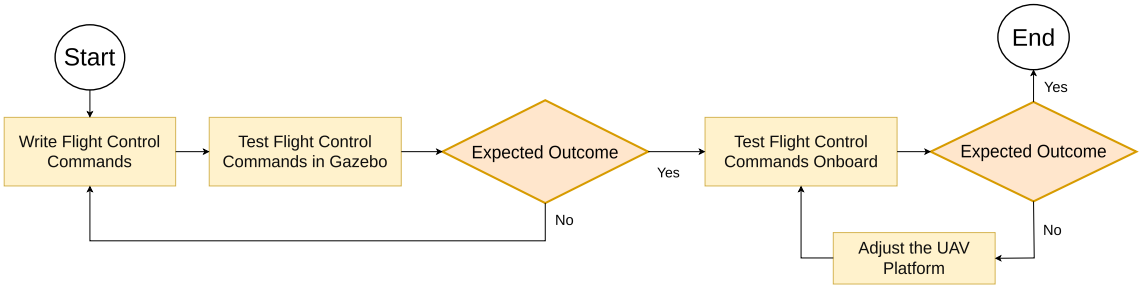


Figure 2.19: Process and Test Environment Architecture for Writing Flight Control Commands.

doi:10.6342/NTU202403413

2.2.1 Mavlink Protocol

MAVLink, short for Micro Air Vehicle Link, is a lightweight communication protocol designed specifically for data exchange between unmanned aerial vehicles (UAVs) and ground control stations (GCS). Developed as part of the PX4 autopilot project by Lorenz Meier and his team, MAVLink has become an industry standard due to its efficiency, flexibility, and widespread adoption. The working principle of MAVLink is as follows:

Packet Format: In MAVLink, whether sending or receiving, operations are performed on the following data structure, shown in Figure 2.20:

- **SYS ID**: Used to identify the sender, where 0 represents broadcasting, and typically numbers 1-255 are used to identify specific systems.
- **COMP ID**: Indicates what component the sender specifically is, for example, it can be set as Autopilot plus Camera.
- **MSG ID**: The packet's identifier, where each packet has a specific ID number, such as 0 representing a heartbeat.
- **Payload**: Contains specific data, and different packets may contain different contents.

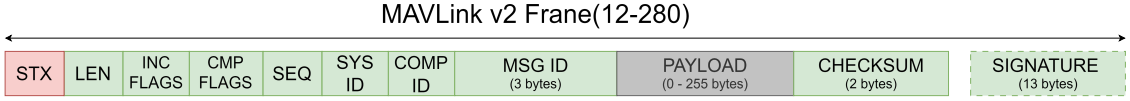


Figure 2.20: MAVLink packet format

Packet Transmission: The MAVLink protocol transmits packets wirelessly from the ground control station to the unmanned aerial vehicle, or vice versa. It can maintain com-

doi:10.6342/NTU202403413

munication stability in unreliable or intermittent connection environments, by supporting error checks, packet sequencing, and retransmission mechanisms.

Command Execution: Upon receiving MAVLink packets sent by the ground control station, the unmanned aerial vehicle parses the instructions in the packet and executes corresponding flight operations, such as changing flight direction, speed, altitude, etc.

Feedback Reception: The unmanned aerial vehicle sends feedback information, such as flight status and image capture conditions, back to the ground control station via the MAVLink protocol. This allows the ground control station to timely understand the status and execution of the unmanned aerial vehicle.

It's worth noting that MAVLink is not exclusive to PX4 autopilot; it is also compatible with ArduPilot, another popular autopilot system used in UAVs. ArduPilot extensively utilizes MAVLink for communication between UAVs and ground control stations. In this study, MAVLink plays a crucial role as a communication protocol. Through MAVLink, we can achieve real-time communication between the unmanned aerial vehicle and the ground control system, and adjust the flight behavior of the unmanned aerial vehicle based on instructions from the ground control system. This mechanism provides the foundation for communication and control in our research, thereby achieving accurate monitoring and repair of cracks.

2.2.2 MAVLink Task Commands

In this section, we will introduce how we use the MAVLink communication protocol to write and send commands to the drone. To align with the eight types of flight behaviors presented in Section 2.1, we have pre-configured the drone to execute these behaviors,

doi:10.6342/NTU202403413

which will ensure a smoother overall test. First, we need to establish a connection to communicate with the drone. We achieve this using the pymavlink library and connect through a USB port. In our example, we have set up the USB port to receive messages from the drone, as shown in Figure 2.21. Once the connection is established, we need to wait for and receive a heartbeat signal from the drone. This heartbeat signal helps us confirm that the connection is established and sets the system and component IDs of the remote system. This step ensures proper communication between the drone and the control system. After receiving the heartbeat signal, we can start sending control commands. To match our model's output, the drone needs to fly in eight directions: up, down, left, right, upper left, upper right, lower left, and lower right, based on its current position. We use the "MAVLink_set_position_target_local_ned_message" command to set the drone's position. We configure the relevant parameters, including the system and component IDs, coordinate frame, bitmask (to specify which bits are enabled), position (x, y, z), as well as speed and acceleration. Finally, after sending the commands, we close the connection to the drone.

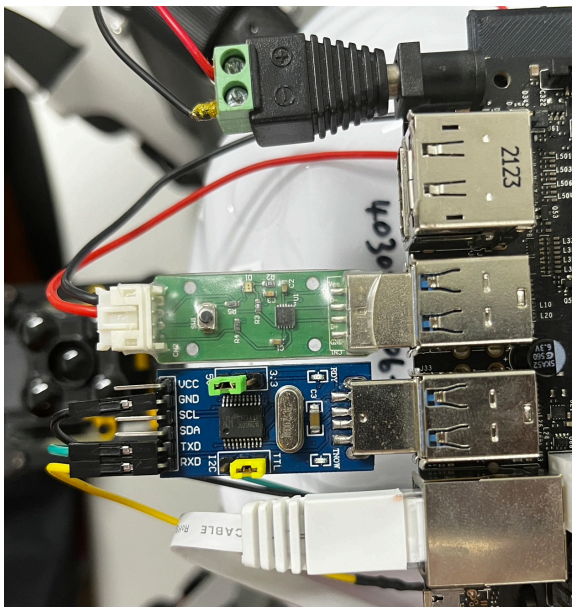


Figure 2.21: GND, TXD, and RXD are consolidated into a USB adapter. (Blue one)

doi:10.6342/NTU202403413

2.2.2.1 Coordinate Frame

In the context of autonomous drone navigation, selecting an appropriate coordinate frame is crucial for ensuring accurate and efficient movement. Various coordinate frames are available in the MAVLink protocol, each with distinct characteristics and use cases. Table 2.3 provides descriptions of the different frame types as documented in the ArduPilot Official Developer Documentation. In this project, we have chosen to use this coordinate frame for several reasons:

1. **Relative Positioning**: The `MAV_FRAME_BODY_OFFSET_NED` frame allows positions to be specified relative to the vehicle's current position and heading. This is particularly useful for dynamic environments where the drone needs to make adjustments based on its current state rather than a fixed origin.

2. **Intuitive Movement**: By using this frame, we can command the drone to move forward, right, and down relative to its current orientation. This simplifies the control logic, as commands can be given in a way that is directly related to the drone's perspective.

3. **Avoiding Repetitive Paths**: The ability to specify movements relative to the current heading helps in avoiding repetitive paths and ensures that the drone can navigate efficiently without getting stuck in loops.

4. **Yaw Control**: This frame allows specifying a yaw rate of zero, preventing unwanted changes in the drone's yaw during movements. This is crucial for maintaining the intended direction of flight, ensuring stable navigation, and avoiding situations where rotation could result in losing sight of the cracks.

doi:10.6342/NTU202403413

Overall, the MAV_FRAME_BODY_OFFSET_NED frame provides the flexibility and control needed for precise navigation in our autonomous drone project. It enables the drone to adapt to real-time changes in its environment while maintaining a clear and consistent movement strategy. This choice is instrumental for conducting pre-flight tests of flight control commands and ultimately simulating the entire project within the Gazebo simulation environment.

Table 2.3: Descriptions of Frame Types from ArduPilot Official Developer Documentation

Frame	Description
MAV_FRAME_LOCAL_NED	Positions are relative to the vehicle's EKF Origin in NED frame. I.e. x=1, y=2, z=3 is 1m North, 2m East and 3m Down from the origin. The EKF origin is the vehicle's location when it first achieved a good position estimate. Velocity and Acceleration are in NED frame.
MAV_FRAME_LOCAL_OFFSET_NED	Positions are relative to the vehicle's current position. I.e. x=1, y=2, z=3 is 1m North, 2m East and 3m below the current position. Velocity and Acceleration are in NED frame.
MAV_FRAME_BODY_NED	Positions are relative to the EKF Origin in NED frame. I.e. x=1, y=2, z=3 is 1m North, 2m East and 3m Down from the origin. Velocity and Acceleration are relative to the current vehicle heading. Use this to specify the speed forward, right and down (or the opposite if you use negative values).
MAV_FRAME_BODY_OFFSET_NED	Positions are relative to the vehicle's current position and heading. I.e. x=1, y=2, z=3 is 1m forward, 2m right and 3m Down from the current position. Velocity and Acceleration are relative to the current vehicle heading. Use this to specify the speed forward, right and down (or the opposite if you use negative values). Specify yaw rate of zero to stop vehicle yaw from changing.

2.2.2.2 Bitmask

In the MAVLink protocol, a bitmask is a technique that uses bits to represent multiple boolean values. Each bit represents a different option or state, and can be manipulated through bitwise operations for setting, clearing, and checking. This method is highly efficient when handling multiple boolean values simultaneously. Bitmasks represent multiple boolean options using a group of bits, usually an integer. Each option corresponds to a

doi:10.6342/NTU202403413

specific bit position. For example, an 11-bit bitmask can represent 11 different options. The parameters represented by each position are shown in Figure 2.22, and the definitions of each parameter are listed in Table 2.4. Here, a digit of 0 indicates the option is selected, while a digit of 1 indicates the option is ignored.

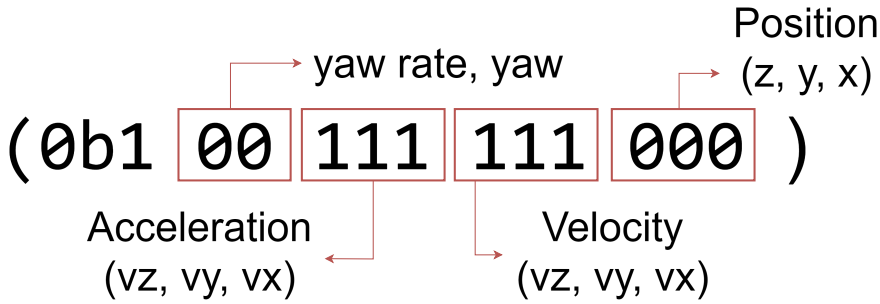


Figure 2.22: The parameters represented by each position in the bitmask, with 0 indicating the parameter is used, and 1 indicating it is not used. In this example, positional parameters are utilized, along with parameters for setting the yaw angle. Under these conditions, it is possible to command the drone to fly a certain distance in a specific direction and instruct it not to rotate.

Table 2.4: Description of Parameters for Drone Flight Control

Parameter	**Description**
x	X Position in meters (positive is forward or North)
y	Y Position in meters (positive is right or East)
z	Z Position in meters (positive is down)
vx	X velocity in m/s (positive is forward or North)
vy	Y velocity in m/s (positive is right or East)
vz	Z velocity in m/s (positive is down)
afx	X acceleration in m/s^2 (positive is forward or North)
afy	Y acceleration in m/s^2 (positive is right or East)
afz	Z acceleration in m/s^2 (positive is down)
yaw	Yaw or heading in radians (0 is forward or North)
yaw_rate	Yaw rate in rad/s

doi:10.6342/NTU202403413

2.3 Integrated Platform

In the preceding two sections, we delved into how we achieved the automation of drones flying along the direction of a localized crack upon detection. In this section, we will consolidate the overall software workflow discussed earlier and introduce the hardware configuration we employed. This will provide readers with a comprehensive understanding of our system design and implementation approach, thus facilitating a better grasp of our research findings.

2.3.1 Software

2.3.1.1 Ubuntu version

For our project, we utilized an NVIDIA Jetson Nano as our upper-level onboard computer. The Jetson Nano operated on Ubuntu 18.04 LTS, a stable and reliable Linux distribution well-suited for development tasks involving artificial intelligence and machine learning. This specific version of Ubuntu provided us with the necessary tools and compatibility to run our deep learning models and communicate effectively with the UAV. The choice of Ubuntu 18.04 LTS was driven by its robustness, long-term support, and the extensive documentation available, which facilitated troubleshooting and development processes.

2.3.1.2 Flowchart

In the preceding sections, we delved into the intricacies of our deep learning and flight control command components, and how we utilized them to achieve our objectives. Now,

doi:10.6342/NTU202403413

we shift our focus to the software workflow during the entire operation. Initiating the process, our upper-level onboard computer directs commands to the unmanned aerial vehicle (UAV) for image capture. Once the UAV captures the images, it transmits them back to the upper-level computer. Subsequently, we first convert the images to 384×384 pixels for crack segmentation. After this, the segmented images are resized to 96×96 pixels and input for extension direction determination. Following these analyses, the computer selects flight control commands based on the decisions made, and then transmits these commands to the unmanned aerial vehicle (UAV) using the MAVLink communication protocol. Finally, the UAV executes the designated flight maneuvers. This iterative process enables us to accomplish our objective of automated crack detection. For a visual representation of this workflow, refer to Figure 2.23. The description of this software workflow will aid readers in better understanding our entire operational process and recognizing the significance of software in our research. Additionally, we will further delve into the details of hardware integration and actual flight in subsequent sections to provide a comprehensive overview. Furthermore, in Section 2.3.3, we will delve into some issues regarding software, hardware, and integration.

doi:10.6342/NTU202403413

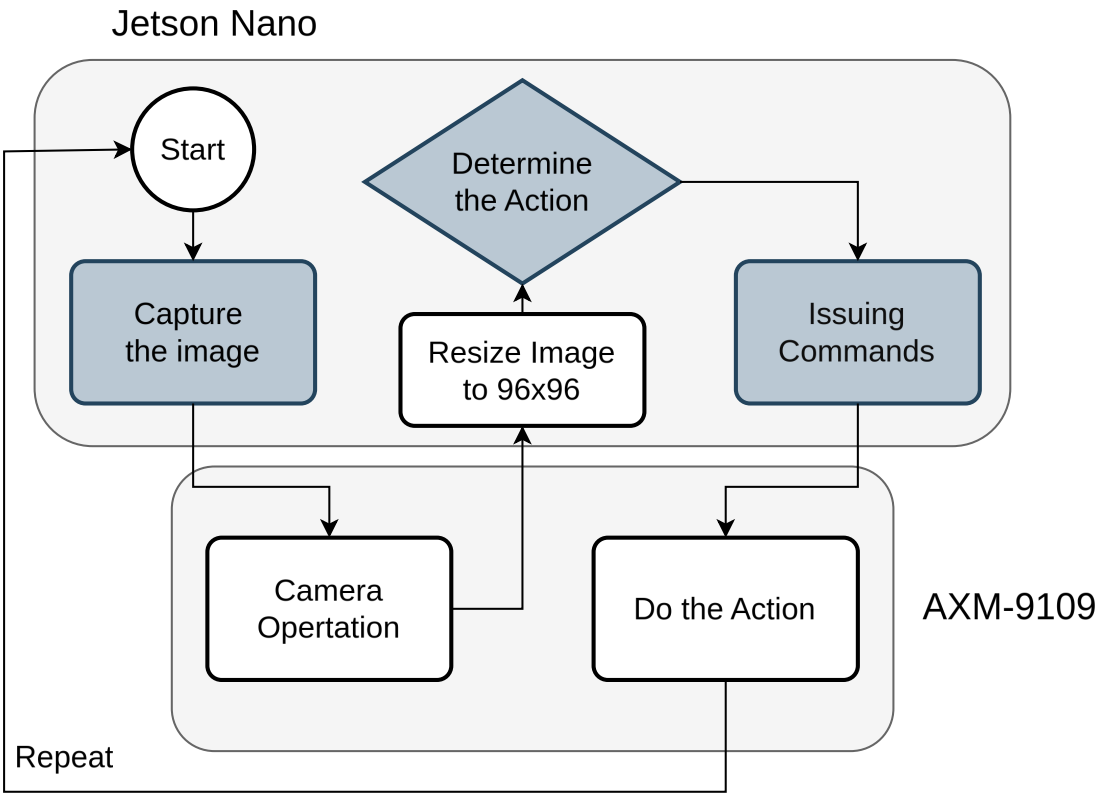


Figure 2.23: Software Workflow Flowchart

2.3.2 Hardware

In this section, we delve into the hardware components crucial to the implementation of our system, which forms the backbone of our research endeavor. Our focus lies in meticulously selecting and introducing the Jetson Nano, a pivotal component in our edge computing infrastructure. Additionally, we provide a comprehensive overview of the drone hardware utilized in our study, highlighting its significance in achieving our research objectives.

2.3.2.1 Jetson Nano

In our research, the installation of an onboard computer atop our unmanned aerial vehicle (UAV) is imperative to enable edge computing capabilities. During the hardware

doi:10.6342/NTU202403413

selection phase, I made the decision to adopt the Jetson Nano. Developed by Nvidia, the Jetson Nano stands as a compact yet powerful edge computing device. It integrates Nvidia's Maxwell architecture GPU with a quad-core ARM Cortex-A57 CPU, endowing it with remarkable computational prowess. It's worth noting that the Jetson Nano finds extensive usage in various real-time robotics applications[27][28], underlining its versatility and reliability in such scenarios. For detailed specifications, please refer to the Table 2.5. Despite its compact size, this device excels in processing graphics and video data and is capable of handling various computational tasks. Moreover, the Jetson Nano offers a variety of input/output interfaces,shown in Figure 2.24, including USB, HDMI, Ethernet, CSI, and DSI, facilitating easy connectivity with external devices and sensors, thus expanding its application range. In addition, the Jetson Nano's low power consumption design surpasses the specifications of other onboard computers. As shown in Table 2.6, the power consumption of the Jetson Nano ranges from only 5 to 10 watts, whereas other onboard computers may consume more power. This low power consumption design is particularly crucial for applications with high power consumption, such as unmanned aerial vehicles (UAVs). UAVs require substantial energy to sustain flight, making the utilization of low power consumption onboard computers essential for efficiently managing limited energy resources. This design not only prolongs the flight time of UAVs but also enhances operational efficiency while reducing reliance on large batteries or charging facilities. Therefore, the Jetson Nano's low power consumption makes it an ideal choice for UAV applications, thereby improving overall system performance and reliability.

Jetson Nano's application is not limited to just crack detection in UAVs. Its powerful computing capability and low power consumption design make it an ideal choice for various UAV applications. For instance, in geological surveys and environmental monitoring,

doi:10.6342/NTU202403413

Jetson Nano can assist UAVs in rapidly analyzing large amounts of collected data, thereby accelerating decision-making processes and improving operational efficiency. Additionally, Jetson Nano is widely used in fields such as smart agriculture, autonomous navigation, and search and rescue missions, where it demonstrates excellent performance and reliability. Overall, the utilization of Jetson Nano opens up new possibilities for the development of UAV technology and showcases significant value across various application scenarios.

Table 2.5: Jetson Nano Hardware Specifications

Component	**Specification**
GPU	NVIDIA Maxwell™ architecture with 128 NVIDIA CUDA® cores
CPU	Quad-core ARM® Cortex®-A57 MPCore processor
Memory	4 GB 64-bit LPDDR4
Storage	16 GB eMMC 5.1 flash memory
Video Encoding	4K @ 30 (H.264/H.265)
Video Decoding	4K @ 60 (H.264/H.265)
Camera	12 channels MIPI CSI-2 DPHY 1.1 (1.5 Gbps)
Connectivity	Gigabit Ethernet
Display	HDMI 2.0 or DP1.2 | eDP 1.4 | DSI (1x2) 2 synchronized
UPHY	1 PCIe x4, 1 USB 3.0, 3 USB 2.0
I/O	1 SDIO, 2 SPI, 6 I2C, 2 I2S, GPIO
Size	69.6 mm x 45 mm
Mechanical	260-pin edge connector

doi:10.6342/NTU202403413

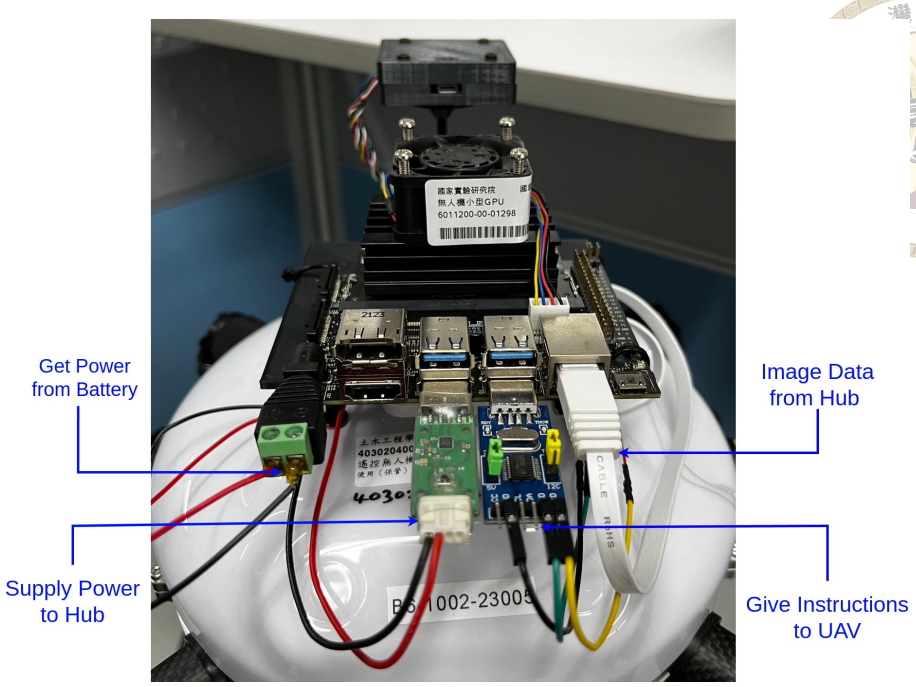


Figure 2.24: Jetson Nano Interface Connection Descriptions

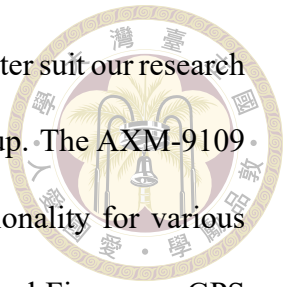
Table 2.6: Onboard Computer Power Consumption Comparison

Computer Model	Power Consumption (W)
Jetson Nano	5-10
Jetson TX2	7.5-15
Jetson Xavier NX	10-20
Jetson AGX Xavier	10-30

2.3.2.2 AXM-9109

In this section, we will introduce the UAV platform we utilized in our research. The UAV selected for this study is the AXM-9109 (Figure 2.25) multirotor UAV developed by AVIX Corporation, featuring a 910mm wheelbase. Notably, this UAV stands out for its open-source architecture, offering enhanced flexibility in development and alignment with our specific research needs. The decision to employ an open-source UAV is pivotal as it allows us to extract data captured by the UAV for analysis on an upper-level computer and transmit flight control commands to execute UAV missions. Leveraging an

doi:10.6342/NTU202403413



open-source system enables us to tailor the UAV's functionalities to better suit our research objectives, facilitating seamless integration with our experimental setup. The AXM-9109 UAV is equipped with essential components, ensuring robust functionality for various aerial applications. These components include ArduPilot Flight Control Firmware, GPS Positioning System, Obstacle Avoidance System, Electronic Speed Controllers (ESC), AXG-AS03F Miniature 3-Axis Gimbal Camera, 2.4GHz Remote Controller, and 5.8GHz Digital Transmission Module. The UAV utilizes ArduPilot firmware for flight control, providing stability and reliability in flight operations. Incorporating a GPS positioning system enables accurate navigation and precise location tracking during flight missions. An obstacle avoidance system is also present to detect and navigate around obstacles in its flight path, enhancing flight safety. Electronic Speed Controllers (ESC) regulate the speed of the UAV's motors, ensuring smooth and responsive flight maneuvers. Additionally, the UAV features a high-quality AXG-AS03F gimbal camera capable of capturing stable and clear aerial footage for observation and analysis. It is controlled remotely using a 2.4GHz radio transmitter, providing reliable communication between the operator and the UAV. Furthermore, a 5.8GHz digital transmission module facilitates real-time transmission of video and telemetry data from the UAV to the ground station, enabling live monitoring and analysis of flight parameters. This hardware configuration (shown in Figure 2.26) provides a solid foundation for various aerial applications, ensuring optimal performance and functionality. Moreover, the AXM-9109 UAV boasts a modular design, allowing for easy customization and integration of additional payloads or sensors to meet specific research requirements. This modular approach enables researchers to adapt the UAV platform for a wide range of applications beyond crack detection, such as aerial mapping, environmental monitoring, or infrastructure inspection. The versatility offered by

doi:10.6342/NTU202403413

the modular design enhances the UAV's capabilities and extends its utility across various research domains, further underscoring its value as a versatile research tool.

Figure 2.25: AXM-9109

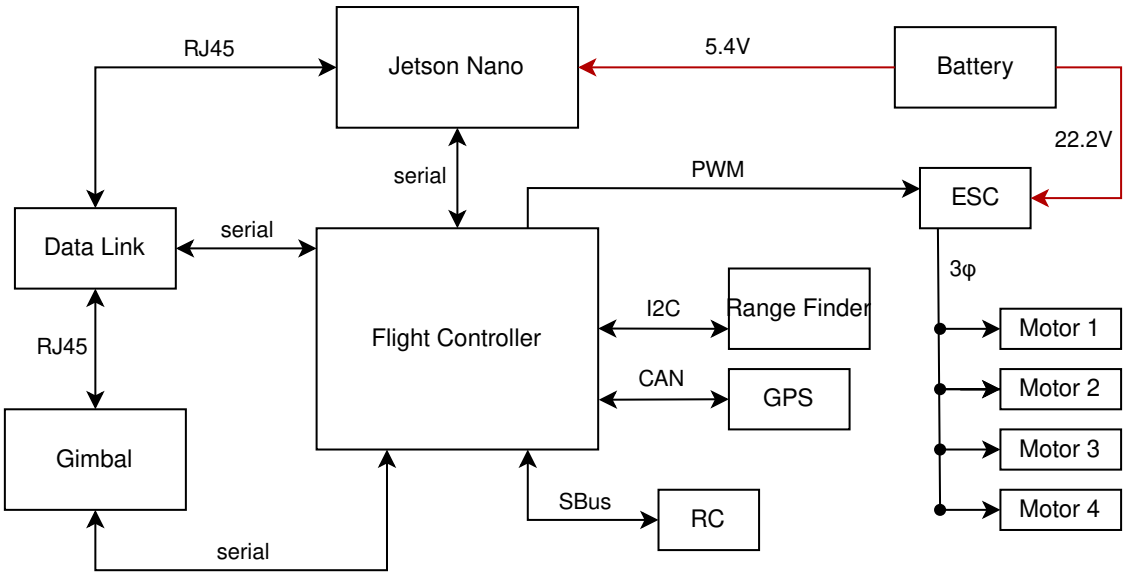


Figure 2.26: The Development of Hardware Configuration

doi:10.6342/NTU202403413

2.3.3 Firmware integration

In a typical system, the interface bridging between hardware and software is commonly referred to as firmware. In unmanned aerial vehicle (UAV) systems, the firmware configuration typically includes components such as flight controllers, electronic speed controllers (ESCs), communication modules, and ground station software. The flight controller firmware manages UAV flight dynamics, stabilization, and navigation control, while ESCs regulate motor speeds to control the aircraft's attitude and flight velocity. Communication module firmware handles data communication between the UAV and ground station, including receiving and transmitting remote control signals and establishing data links. Ground station software is used for mission planning, flight monitoring, and data analysis for ground control operations. In our system, there are several special configurations, which will be elaborated in detail in the following section.

2.3.3.1 Image Transmission Issue

During the flight of the UAV, due to the addition of the onboard computer, the ground station cannot monitor the remote desktop screen of the onboard computer. This will result in the inability of operators to perform necessary operations and monitoring. Therefore, we urgently need to find a solution to this problem. Due to the existing communication module in our UAV system, which facilitates communication with the ground station, we leveraged this capability to achieve our objective. Additionally, this approach also addresses another challenge: how to simultaneously transmit the images captured by the camera back to the ground station for monitoring while sending them to the onboard computer above for edge computing purposes. To facilitate the transmission of images cap-

doi:10.6342/NTU202403413

tured by the onboard computer of our UAV back to the ground station, we implemented an Ethernet hub for cable integration, as depicted in Figure 2.27. This hub allowed us to consolidate three Ethernet cables: one for the camera, one for the communication module, and one for the onboard computer (the corresponding cable connectors can be observed in Figure 2.24). Furthermore, we adjusted the Ethernet segment settings on the ground station to align with those of the onboard computer, ensuring compatibility (specifically, we configured the ground station's address as 192.168.42.87 with a netmask of 255.255.255.0, matching that of the onboard computer). This approach enabled us to seamlessly share image data across all required devices. Additionally, leveraging this setup, we could transmit the remote desktop screen of the onboard computer back to the ground station, as depicted in Figure 2.28.

Through this approach, we can avoid the need for an additional dedicated line for remote screen transmission, making the overall architecture more streamlined. However, it is worth noting that sharing the same line may result in limited bandwidth and congestion. Fortunately, our research primarily focuses on edge computing, significantly reducing unnecessary data transmission. In Figure 2.29, detailed diagrams of the image and data transmission architecture are provided to enhance readers' understanding of our system structure.

doi:10.6342/NTU202403413

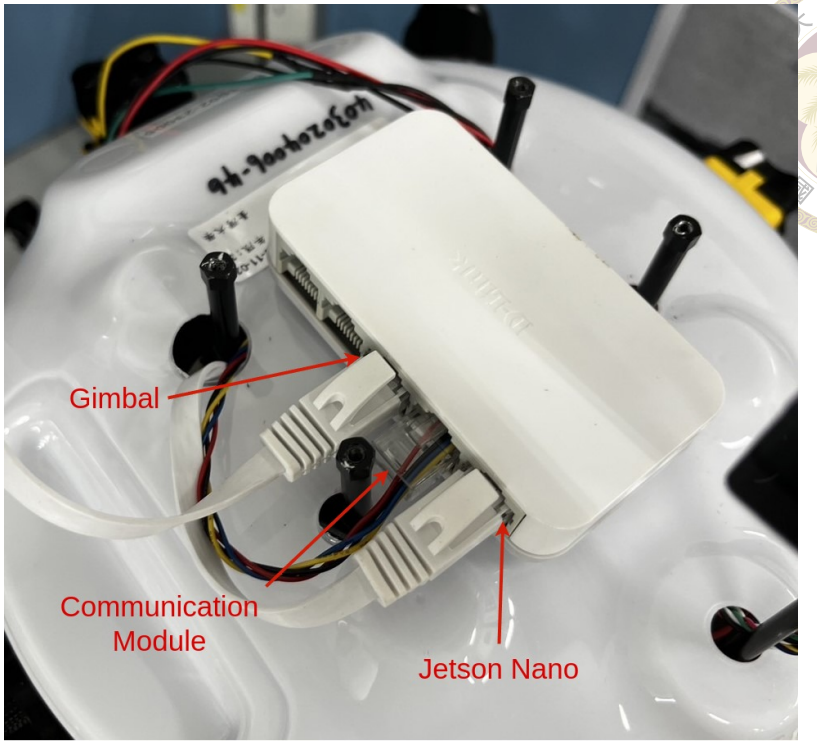


Figure 2.27: Ethernet hub used for cable consolidation

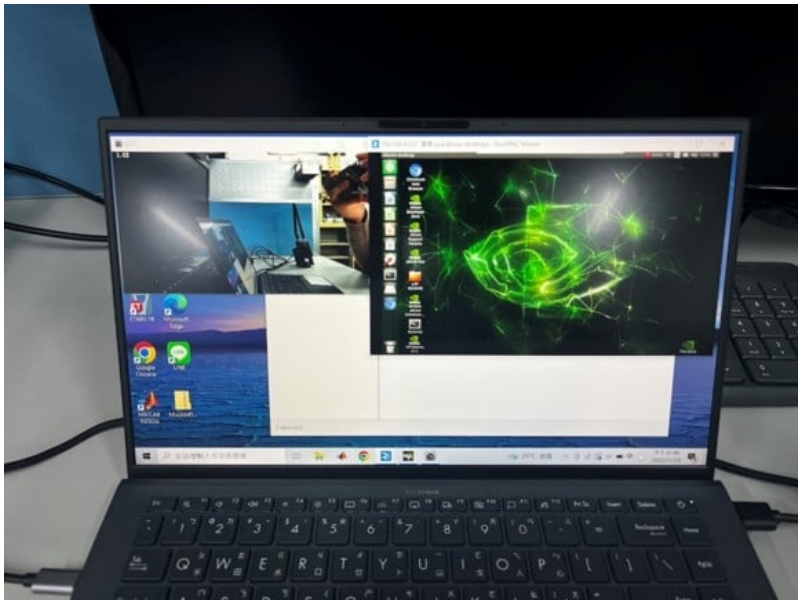


Figure 2.28: The remote desktop screen of the onboard computer and the video feed from the UAV are both transmitted back to the ground station through this architecture.

doi:10.6342/NTU202403413

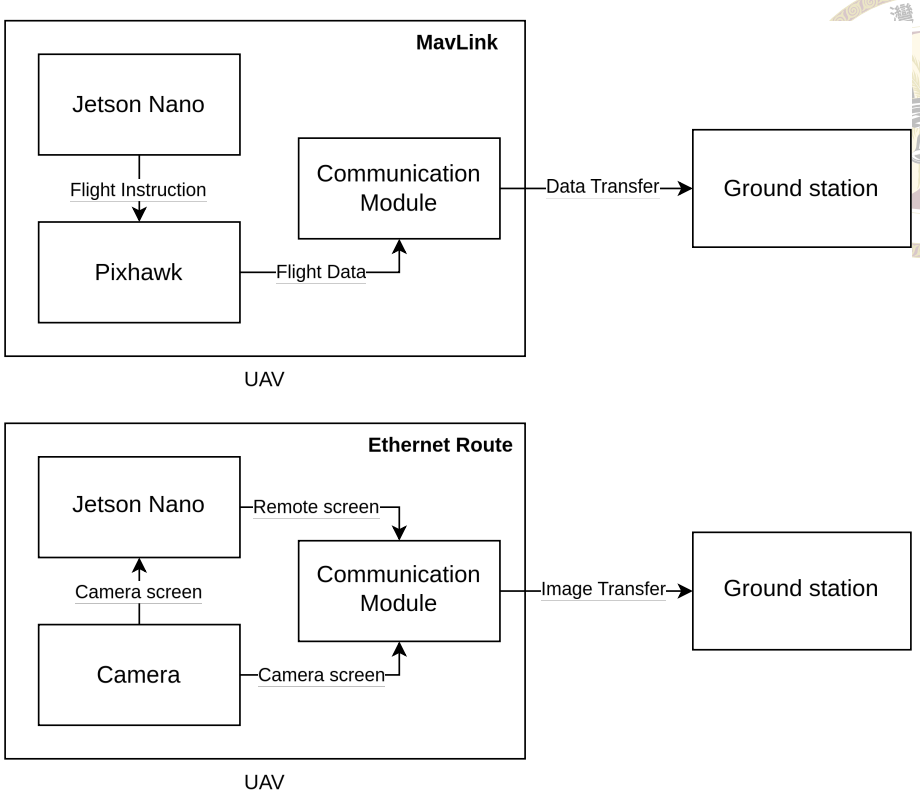


Figure 2.29: Architecture diagrams for image and data transmission

2.3.3.2 Unstable Flight Control Transmission

In Section 2.2, we mentioned that we use MAVLink commands to issue flight instructions to the drone. In most cases, to enable the Jetson Nano to control external hardware, we use its GPIO pins, as shown in Figure 2.30. The GPIO pins of the Jetson Nano offer several advantages, including flexible configuration, high versatility, compatibility with high-level programming languages, and low hardware cost. However, during actual operations, we discovered that directly connecting the main drone control pins (GND, TXD, RXD) to the GPIO pins resulted in signal instability during flight. This issue is likely caused by the electrical characteristics of the GPIO pins and signal jitter due to interference. To address this problem, we decided to integrate these three control pins into a single USB connector and plug it into the USB port of the Jetson Nano, as shown in Figure 2.31. This change not only improved signal stability but also simplified the hardware connec-

doi:10.6342/NTU202403413

tion, reducing the risk of potential contact issues. In this process, we also rewrote the relevant internal commands to adapt to the new USB interface configuration. These improvements have made the drone more stable and safer during flight, effectively reducing the risk of control errors due to signal instability. Overall, integrating the control pins into the USB connector and making the necessary hardware and software adjustments have significantly enhanced the system's reliability and flight performance.

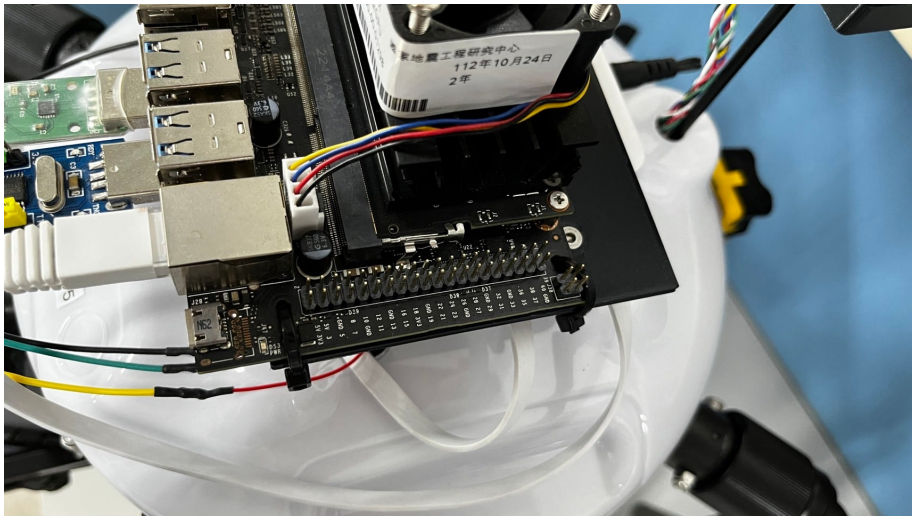


Figure 2.30: Jetson Nano can control external hardware through its GPIO pins, allowing for input and output operations.

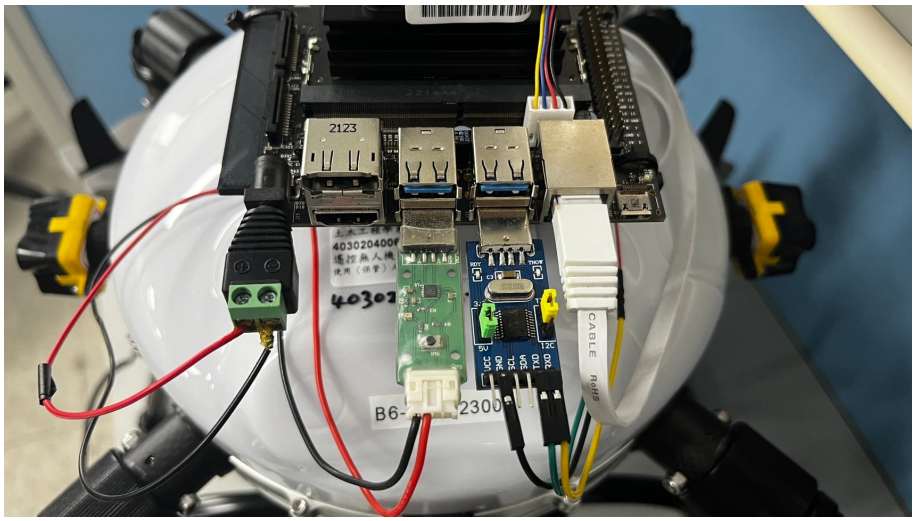


Figure 2.31: Integrating GND, TXD, and RXD into a single USB port to enhance stability.

doi:10.6342/NTU202403413

2.3.3.3 Edge Computing Optimization in Firmware

When performing edge computing, we chose Jetson Nano as the tool for real-time processing. Although Jetson Nano is known for its GPU for parallel processing of large amounts of data, our edge computing tasks mainly focus on segmenting and analyzing individual images of cracks, a processing mode that does not fully leverage the GPU's parallel computing capabilities. In fact, sometimes transferring data to the GPU may result in slower processing speeds. Through testing and comparison, we found that using the CPU for processing individual images is comparable to the GPU in terms of performance, and has advantages in system resource management and power consumption. Therefore, we decided to rely on the CPU of Jetson Nano for efficient and stable operation throughout the entire image processing process. This choice further simplifies the system design and implementation, enhancing overall stability and reliability. Table 2.7 and Figure 2.32 illustrate the performance variation of the CPU and GPU with different numbers of single images. The results show that when processing a small amount of data, the CPU performs better for edge computing; however, when processing a large amount of data, the GPU outperforms. This further highlights the applicability of the CPU and GPU in different situations, providing targeted references for selecting the most suitable computing solution.

Table 2.7: Speed Variation for Different Quantities of Images Processed by CPU and GPU

Quantity of Images	CPU Speed (seconds)	GPU Speed (seconds)
1	3.8	67.4
10	33.9	90.1
237	141.2	110.9

doi:10.6342/NTU202403413

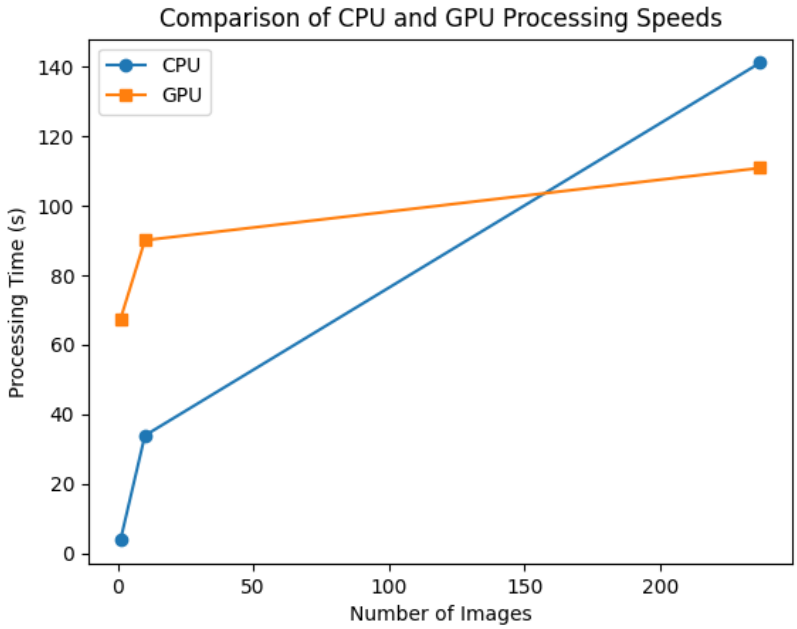


Figure 2.32: Comparison of CPU and GPU Processing Speeds for Different Numbers of Images.

2.4 Calculation of Crack Width in Real World

We can clearly observe from the flight results that although the direction of cracks remains generally consistent between the original images and the segmented images, there is a change in scale. Therefore, when conducting future training or assessments of crack widths, there may be discrepancies with reality. For this reason, factors such as flight altitude and camera zoom level impact the actual size of cracks captured after compression and segmentation. In this section, we will explore these relationships and establish a formula that connects these parameters.

In this section, we outline the camera and custom parameters essential for our imaging system. The camera parameters include a capture size of $D_h \times D_v$ pixels (1280 × 720), and a segmentation size of $D'_h \times D'_v$ pixels (384 × 384). The camera model used is the AXG-AS03F, featuring a CMOS sensor of 1/2.3” with a diagonal measurement of 7.6mm. The

doi:10.6342/NTU202403413

sensor dimensions are 6.17mm in width and 4.55mm in height, with an initial focal length set at 13.4mm. The custom parameters consist of the crack width, denoted as W_h and W_v for horizontal and vertical measurements, respectively. Additionally, the flight altitude is represented by H, and the zoom level by M. The compression ratio is specified as R_H for horizontal and R_V for vertical compression. Finally, the crack width after segmentation is indicated as P_h for horizontal and P_v for vertical measurements. Section 2.4.1 will provide a detailed derivation of the relationship equations.

2.4.1 Formula Derivation

In this section, we will elucidate the relationships between flight altitude, camera zoom level, compression ratios, and the resulting crack widths after segmentation.

We first need to calculate the camera's field of view (FOV), which depends on the camera's lens width (w) and height (h), as well as our initial focal length (f). FOV_h represents the horizontal field of view (FOV) of the camera, while FOV_v represents the vertical field of view (FOV) of the camera:

$$\text{FOV}_h = 2 \times \tan^{-1}\left(\frac{w}{2 \times f}\right) \tag{2.7}$$

$$\text{FOV}_v = 2 \times \tan^{-1}\left(\frac{h}{2 \times f}\right) \tag{2.8}$$

Next, we aim to calculate the unit pixel size of the actual image based on the camera's field of view angle. The altitude H and magnification factor M also influence this calculation. S_h represents the unit pixel size in the horizontal direction of the real image,

doi:10.6342/NTU202403413

while S_v represents it in the vertical direction:

$$S_h = \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_h}{2}\right)}{D_h \times M} \tag{2.9}$$

$$S_v = \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_v}{2}\right)}{D_v \times M} \tag{2.10}$$

where D_h is number of horizontal pixels in capture image, D_v is number of vertical pixels in capture image

Next, we will calculate the compression ratio for width and height, and then determine the unit pixel size for segmentation (S'_h, S'_v):

$$R_h = \frac{1280}{384} = 3.33 \tag{2.11}$$

$$R_v = \frac{720}{384} = 1.88 \tag{2.12}$$

$$S'_h = S_h \cdot R_h = \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_h}{2}\right)}{D'_h \times M} \tag{2.13}$$

$$S'_v = S_v \cdot R_v = \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_v}{2}\right)}{D'_v \times M} \tag{2.14}$$

where D'_h is number of horizontal pixels in segmentation image, D'_v is number of vertical pixels in segmentation image

doi:10.6342/NTU202403413

Finally, we will establish a relationship between the actual crack width and the segmented crack width using the parameters defined above:

$$W_h = P_h \cdot S'_h = P_h \cdot \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_h}{2}\right)}{D'_h \times M} \tag{2.15}$$

$$W_v = P_v \cdot S'_v = P_v \cdot \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_v}{2}\right)}{D'_v \times M} \tag{2.16}$$

$$W \approx \sqrt{W_h^2 + W_v^2} \tag{2.17}$$

where P_h is horizontal crack width in segmentation image, P_v is vertical crack width in segmentation image

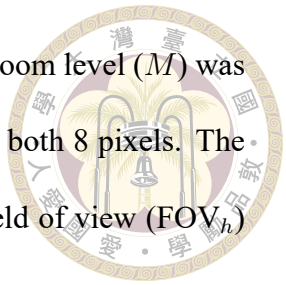
To better reflect the relationship between real-world conditions and ideal values, we introduce a constant term α in our results and use our actual flight data to determine this constant.

$$W_{\text{real}} = \alpha \cdot W \tag{2.18}$$

Where α is the constant term that will be derived based on our actual flight results.

We will use data from one of the flights in Test A to determine the value of α. In Test A, the flight altitude was 2 meters, and the zoom level was 3. The segmentation results of the crack width are shown in Figure 2.33. The crack width in both horizontal and vertical directions in the segmented image is 8 pixels. Using this data, we will perform the

doi:10.6342/NTU202403413



calculations as follows: the flight altitude (H) was 2 meters, and the zoom level (M) was 3. The horizontal pixel count (P_h) and vertical pixel count (P_v) were both 8 pixels. The actual crack width (W_{real}) was 5 mm. Additionally, the horizontal field of view (FOV_h) was $25.9°$, and the vertical field of view (FOV_v) was $19.4°$.

$$W_h = P_h \cdot S'_h = P_h \cdot \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_h}{2}\right)}{384 \cdot M} \tag{2.19}$$

$$W_v = P_v \cdot S'_v = P_v \cdot \frac{2 \cdot H \cdot \tan\left(\frac{\text{FOV}_v}{2}\right)}{384 \cdot M} \tag{2.20}$$

$$W \approx \sqrt{W_h^2 + W_v^2} \tag{2.21}$$

Using the above equations, we calculate:

$$\tan\left(\frac{25.9°}{2}\right) \approx 0.23$$

$$\tan\left(\frac{19.4°}{2}\right) \approx 0.17$$

$$W_h = 8 \cdot \frac{2 \cdot 2 \cdot 0.23}{384 \cdot 3} \approx 0.00638 \text{ meters} \approx 6.38 \text{ mm}$$

$$W_v = 8 \cdot \frac{2 \cdot 2 \cdot 0.17}{384 \cdot 3} \approx 0.00471 \text{ meters} \approx 4.71 \text{ mm}$$

doi:10.6342/NTU202403413

$$W \approx \sqrt{(6.38\ \text{mm})^2 + (4.71\ \text{mm})^2} \approx 7.9\ \text{mm}$$

Next, we introduce the constant α:

$$W_{\text{real}} = \alpha \cdot W \tag{2.22}$$

Given the actual crack width $W_{\text{real}} = 5$ mm:

$$5\ \text{mm} = \alpha \cdot 7.9\ \text{mm}$$

$$\alpha = \frac{5}{7.9} \approx 0.63$$

Therefore, the constant α is approximately 0.63. Thus, we can establish a complete relationship equation. In the next section, we will use another test to verify this constant.

doi:10.6342/NTU202403413

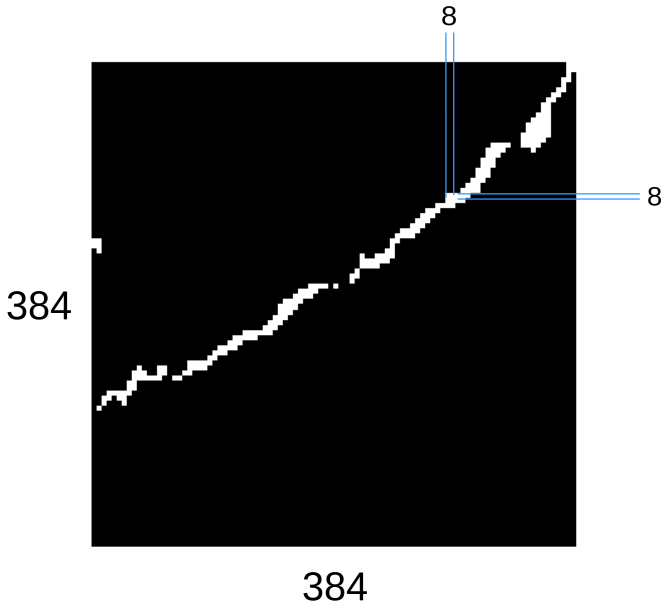


Figure 2.33: The segmented image of the crack along with its width and height measurements after segmentation.

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

Chapter 3 Implementation

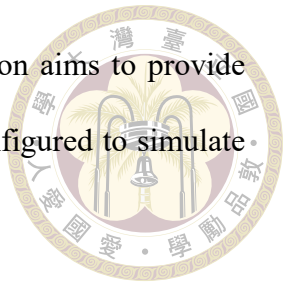
In the previous chapter, we detailed how we utilized deep reinforcement learning and the integration of UAV software and hardware to achieve our goal of automatically exploring cracks. In this chapter, we will implement the entire process to verify the accuracy of our methodology. The chapter is divided into two main parts: simulation and real-world flight. First, we will use Gazebo, a 3D simulation environment, along with various open-source software tools, to test whether our process meets expectations. This simulation will allow us to validate the system's performance in a controlled virtual environment before moving to real-world testing. Next, we will conduct actual flight tests. We will start by setting some basic assumptions to facilitate smooth flight operations. We will then conduct the flights, record the results, and analyze the outcomes. Finally, we will review the entire process to identify any issues and discuss potential improvements. By systematically following these steps, we aim to demonstrate the feasibility and effectiveness of our automated crack exploration system.

3.1 Simulation

In this section, we will first introduce the 3D simulation software Gazebo, which we used extensively in our project. We will also discuss the various simulation tasks

doi:10.6342/NTU202403413

and settings that we employed to validate our approach. This section aims to provide a comprehensive understanding of how Gazebo was utilized and configured to simulate real-world conditions for our UAV's crack exploration tasks.



3.1.1 Gazebo

Gazebo [29][30] is a powerful and flexible open-source simulation tool widely used for developing and testing robotics, drones, vehicles, and other automated systems. It provides a high-fidelity physics engine, rich 3D graphics, and numerous built-in features that allow developers to test and validate their robots and control algorithms in a virtual environment. Gazebo uses advanced physics engines, such as ODE, Bullet, and DART, to accurately simulate real-world physical phenomena, including collision detection, friction, and gravity. This ensures that the simulation results are as realistic as possible. The platform offers extensive 3D graphics support, enabling users to create detailed virtual environments. Users can build custom environments from scratch or use pre-existing models and environments from Gazebo's extensive library, including various terrains, buildings, obstacles, and both static and dynamic objects. One of Gazebo's key strengths is its support for a wide range of sensors, such as cameras, LiDAR, IMU (Inertial Measurement Units), and GPS. These simulated sensors provide data that can be used for robot navigation, environment perception, and decision-making, facilitating the development and testing of algorithms. Additionally, Gazebo's flexible plugin architecture allows users to write custom plugins to extend its functionality, whether to simulate specific hardware behaviors, add new sensors, or control the robot's actions. Gazebo is also tightly integrated with the Robot Operating System (ROS), making it an essential component of the ROS ecosystem. This integration allows users to leverage Gazebo for robot simulation, con-

doi:10.6342/NTU202403413

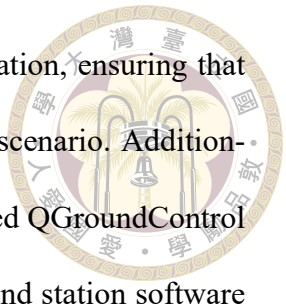
trol, and data exchange within ROS. Algorithms developed and tested in Gazebo can be seamlessly transferred to real robotic systems, ensuring a smooth transition from simulation to real-world deployment. Gazebo supports multi-robot simulation, enabling users to test interactions and cooperation between multiple robots in the same simulated environment. This is particularly useful for developing and testing swarm robotics systems and distributed algorithms. As an open-source platform, Gazebo benefits from an active developer and user community that provides extensive documentation, tutorials, and technical support. Users have access to a wealth of resources, including official documentation, forums, example projects, and third-party plugins.

In this project, Gazebo will be used for pre-flight testing of flight control commands. This involves simulating the drone's flight dynamics, control systems, and sensor outputs to test different flight strategies and algorithms. Additionally, the entire project will be simulated in Gazebo to ensure comprehensive validation before real-world implementation. This approach allows for identifying and addressing potential issues in a controlled virtual environment, thereby enhancing the reliability and performance of the final system.

3.1.2 Testing Architecture

When conducting tests in the Gazebo simulation environment, we needed a drone model to clearly express the overall process. To achieve this, we used an open-source package available online, which includes a drone model with ArduPilot firmware. This software package not only helps us better present our expected results but also ensures that our simulation environment is highly consistent with the actual flight environment. Since we use ArduPilot firmware for flight control in our actual flights, the flight control commands in both the simulation and actual flights only require minor adjustments. This

doi:10.6342/NTU202403413



greatly simplifies the transition from simulation to real-world application, ensuring that the drone's position in the simulation closely matches the actual flight scenario. Additionally, for effective ground monitoring and data management, we utilized QGroundControl (QGC) as our ground station system. QGC is a commonly used ground station software that helps us monitor the drone's flight status during the simulation, simulating the ground station operations. QGC can display real-time status information of the drone, such as position, altitude, and speed, and supports switching between various flight modes, significantly enhancing our control and management efficiency during the simulation tests. Lastly, the transmission of the entire model and commands is executed using Python on a desktop environment, simulating operations on the actual Jetson Nano. Through Python scripts, we can send and receive MAVLink commands to control the drone's flight and status. This setup allows comprehensive testing and debugging in a desktop environment, ensuring smooth deployment on the Jetson Nano. The overall simulation architecture, as shown in Figure 3.1, illustrates the complete process from the simulation environment to real-world application. Figure 3.2 demonstrates the operation of these software tools in practice. This design not only ensures the reliability and accuracy of the simulation results but also provides us with an efficient and flexible testing platform.

doi:10.6342/NTU202403413

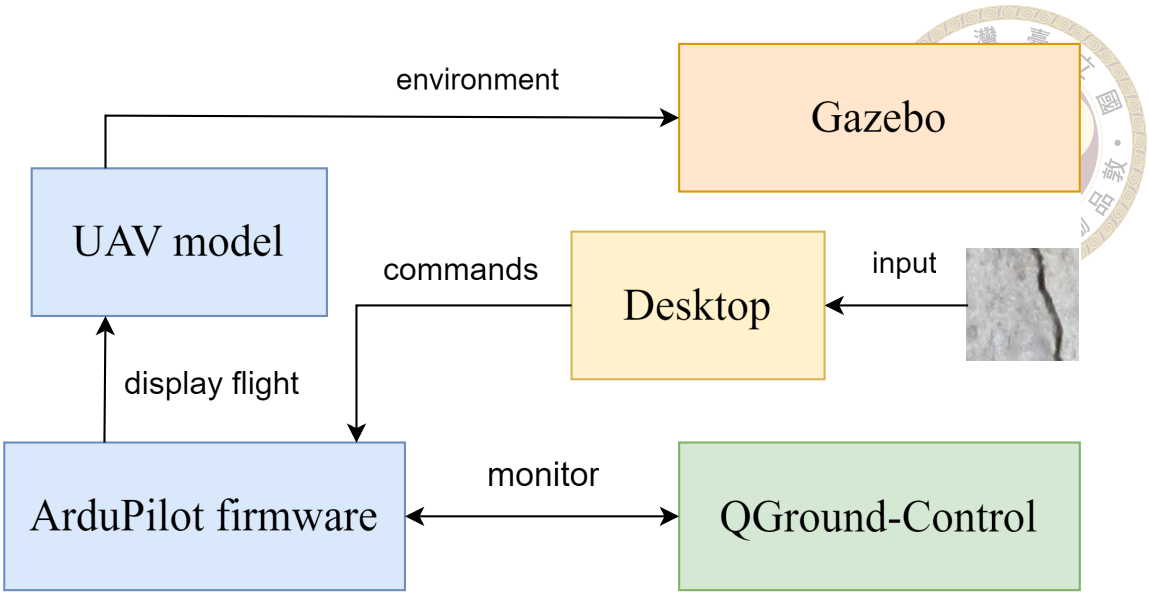


Figure 3.1: The proposal simulation process

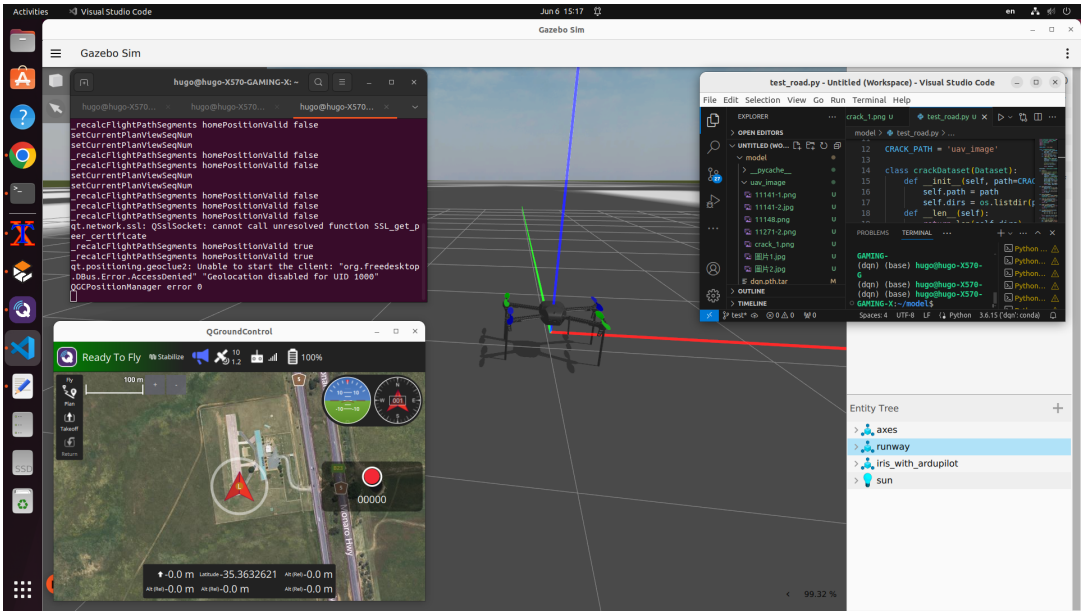


Figure 3.2: The interface for the simulation environment

3.1.3 Simulation Test

In the forthcoming phase, we will undertake process tests within the 3D simulation environment. Adhering to the architecture outlined in Figure 3.1, we will feed diverse local crack images into the computer's deep reinforcement learning model. Subsequently, upon the model's determination of the optimal direction, it will dispatch corresponding flight

doi:10.6342/NTU202403413

control commands to the drone's flight control firmware within the simulation environment, thereby inducing flight in the drone model. Our scrutiny in the Gazebo environment will assess the degree to which actual outcomes align with expectations, encompassing the directional determinations made by the deep reinforcement learning model and the corresponding flight status correlated with the flight control commands. Figures 3.3 and 3.4 offer visual representations of the flight conditions relative to various local crack images inputted. Particularly, the portrayal in the bottom left corner of each image illustrates the local crack scene observed by the drone during flight. It's worth noting that prominent red arrows within the images indicate the direction of autonomous flight determined by the drone after analysis. These visual aids are crucial for understanding how the model interprets different crack patterns and translates that information into navigational decisions. Based on the aforementioned assessments, we discern a notable convergence between the actual outcomes and our initial anticipations. This affirmation underscores the efficacy of the architecture we have devised in expediting communication with the drone. The results demonstrate that our deep reinforcement learning model can effectively process visual data and make real-time decisions that align with our predictive models. Consequently, the subsequent section will witness our transition towards real-world flight tests, where we will further validate our system's performance in more complex and uncontrolled environments. This progression is vital for ensuring the robustness and reliability of our technology in practical applications.

doi:10.6342/NTU202403413

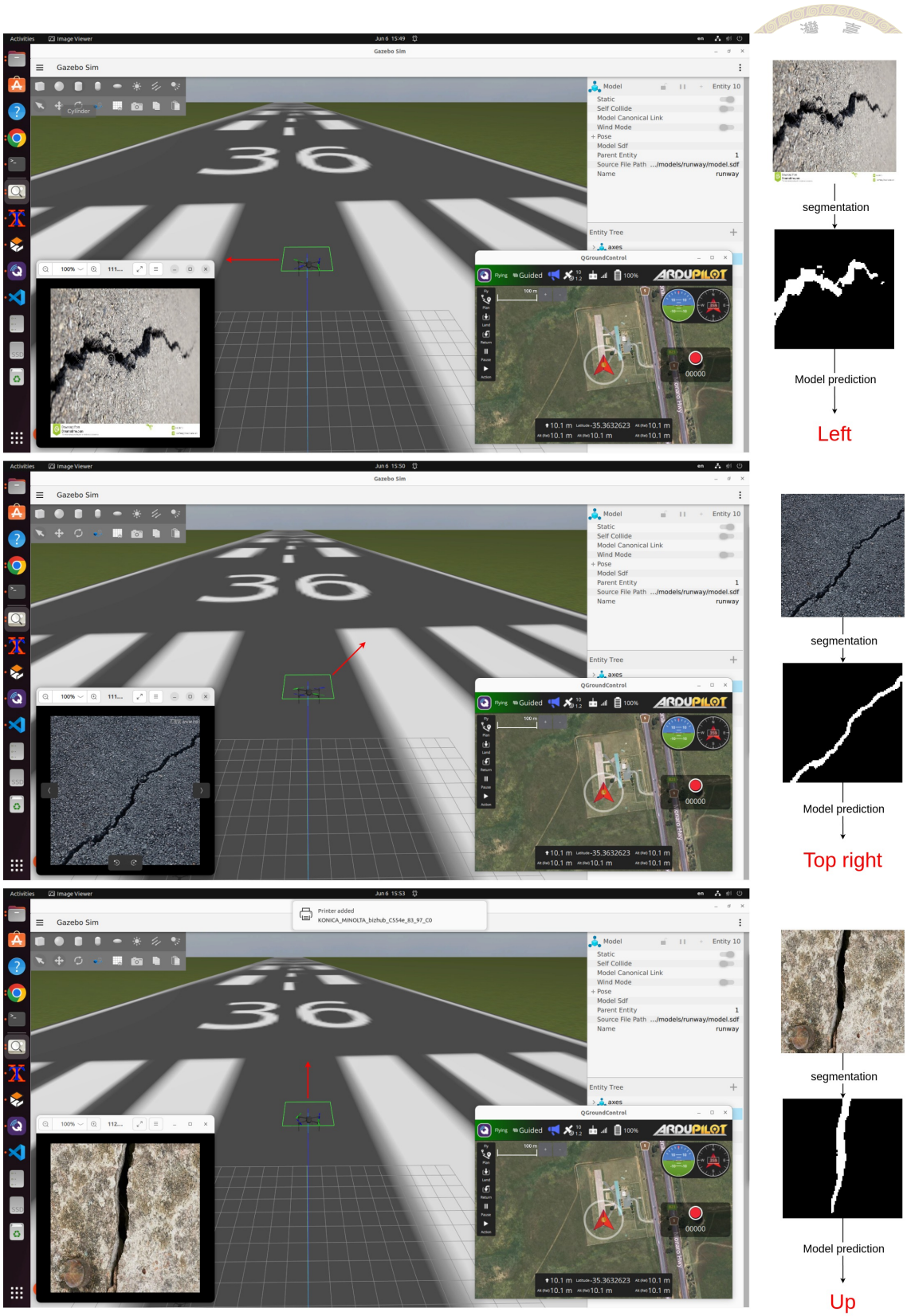


Figure 3.3: Testing Deep Reinforcement Learning Decisions and Flight Trajectories in the Simulation Environment.

doi:10.6342/NTU202403413

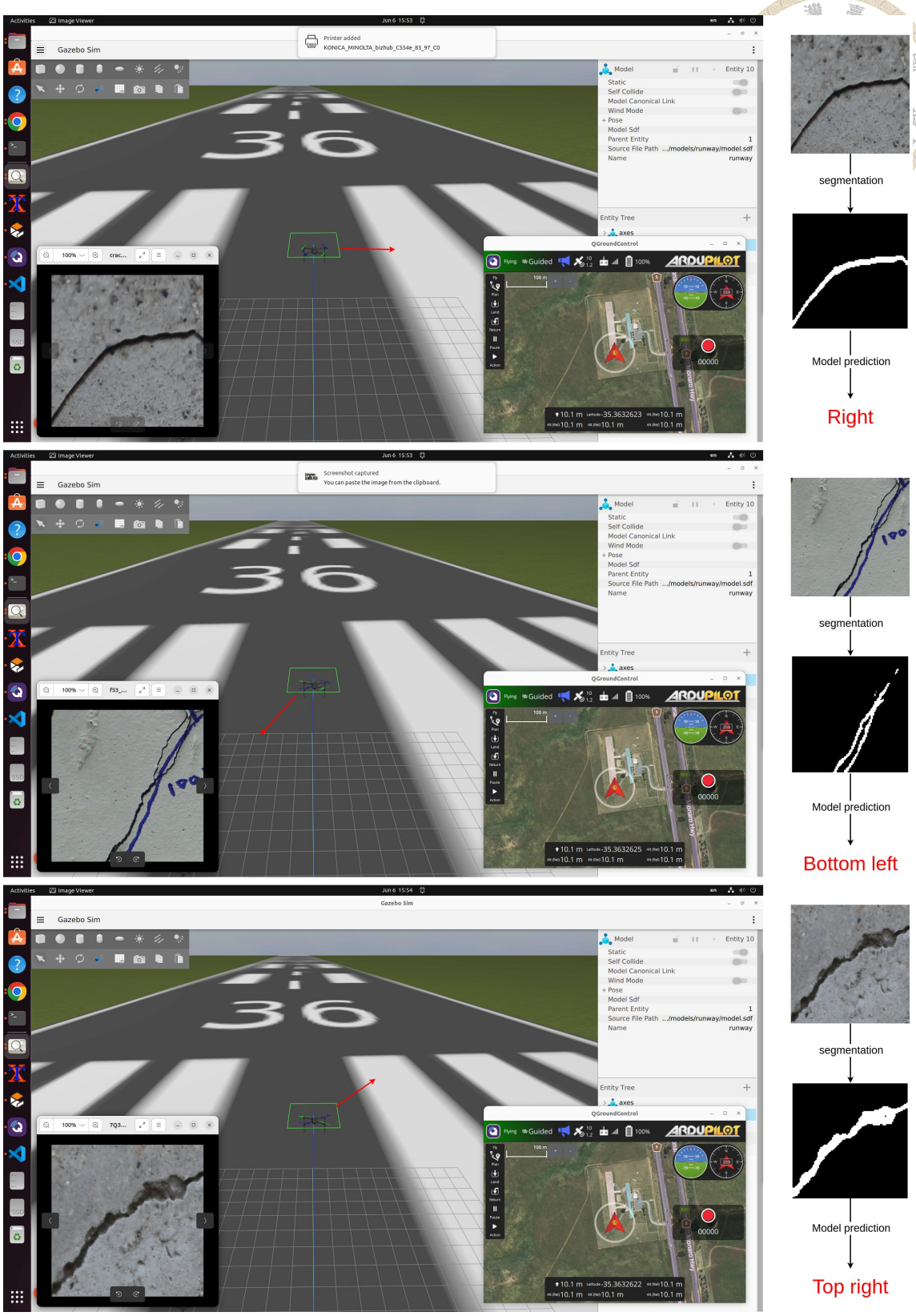


Figure 3.4: Testing Deep Reinforcement Learning Decisions and Flight Trajectories in the Simulation Environment.

doi:10.6342/NTU202403413

3.2 Real Flight

In this section, we will present the results of our real-world flight tests. Before showcasing the results, we will first introduce some of the pre-flight settings.

3.2.1 Camera Orientation for Experiment

In our initial setup, we anticipated capturing cracks in bridge structures or building facades as our ultimate goal. To achieve this, we initially positioned the camera to face forward, capturing the vertical cracks, and then algorithmically predicting directions based on observed targets, outputting directions parallel to the vertical plane, including up, down, left, and right. However, considering that our overall testing is still in its preliminary stages and taking into account safety and operational concerns, we made some adjustments to our setup for the physical flight portion of the experiment. We redirected the camera downward, as illustrated in Figure 3.5, to observe cracks on horizontal surfaces. By assessing these cracks and determining their directions, the output directions were adjusted to include forward, backward, left, and right, as depicted in Figure 3.6. This adjustment facilitated smoother experimentation and yielded comparable results. Additionally, we also took into consideration changes in the camera orientation when adjusting our flight control commands. This ensured that our drone could effectively respond to the altered perspective and navigate accordingly. In future developments, we aim to simultaneously capture cracks on both horizontal surfaces, such as roads, and vertical surfaces, such as walls.

doi:10.6342/NTU202403413

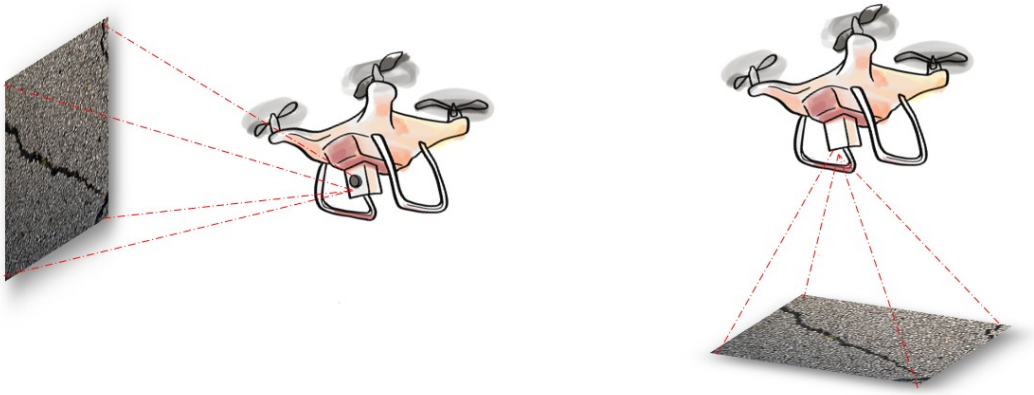


Figure 3.5: On the left is the status of detecting wall cracks, while on the right is the status of detecting ground cracks.

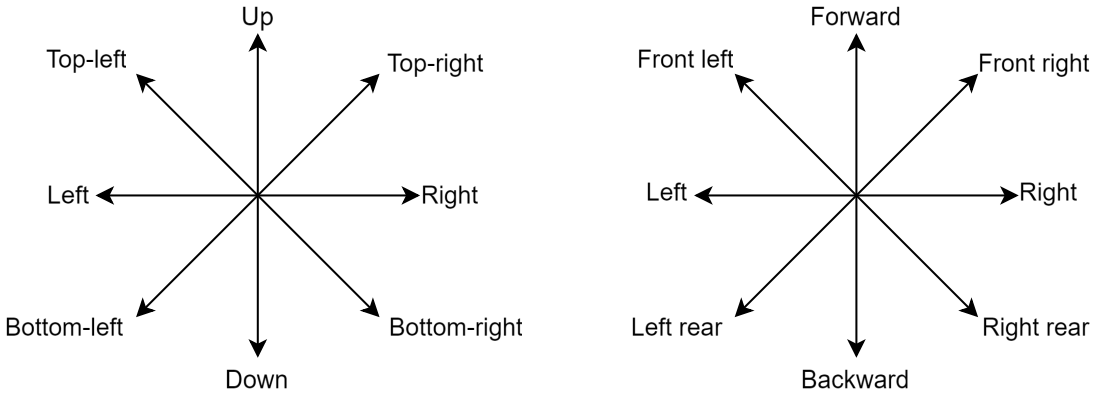


Figure 3.6: On the left, the flight control commands define directions for vertical surfaces; on the right, the flight control commands define directions for horizontal surfaces.

3.2.2 Flight Altitude and Camera Zoom

In the process of actual crack capture, we identified an optimal flight altitude and camera zoom level. This helps us capture clearer crack images and makes our edge computation results more accurate. The variations in flight altitude and camera zoom level have the following features:

1. **Flight Altitude**: Different flight heights affect the image coverage and detail cap-

doi:10.6342/NTU202403413

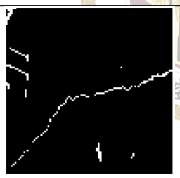
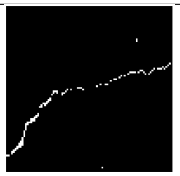
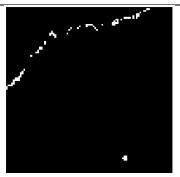
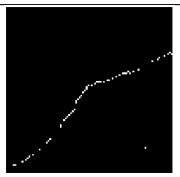
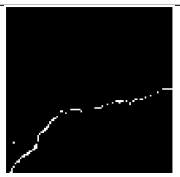
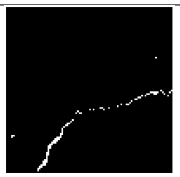
ture. Lower flight heights help obtain higher resolution images, making it easier to identify small cracks. However, this requires more flights to cover the same area. Additionally, lower flight heights may result in clearer images but can compromise flight stability due to increased susceptibility to wind and other environmental factors. Conversely, higher flight heights cover a larger area but may sacrifice some detail in the images.

2. **Camera Zoom Level**: Adjusting the camera's zoom level helps capture clear crack images at different distances. Higher zoom levels capture details from a distance, while lower zoom levels are suitable for close observation, providing a wider field of view. However, excessively high zoom levels can cause issues with lighting, increasing the risk of overexposure and glare, which can affect image quality and crack detection accuracy. Moreover, when zoomed in too much, there may be insufficient aperture for adequate light intake, further compromising image clarity and detail visibility.

We need to find a balance between these two factors. Achieving this balance will help filter out very small cracks and improve the overall accuracy and stability of the detection process. For our testing, we will employ two flight altitudes: 1 meter and 2 meters, and combine them with three zoom levels (×1, ×2, ×3). Through this combination, we aim to identify the optimal setup that achieves the best crack segmentation while maintaining stable flight operations. Comparative results will be illustrated in the following Table 3.1.

doi:10.6342/NTU202403413

Table 3.1: Comparison of Results at Different Flight Altitudes and Camera Zoom Levels

	Altitude (m)	Zoom	Capture Image	Segmentation
A	1	×1		
B	1	×2		
C	1	×3		
D	2	×1		
E	2	×2		
F	2	×3		

Based on the tests above, we can observe that the crack segmentation performance is clear and accurate across all combinations. This indicates that our segmentation model, after adjustments, performs well and demonstrates good robustness in actual flight operations.

doi:10.6342/NTU202403413

3.2.3 Real Flight Test

In this section, we will implement the previously described overall framework on an actual drone for automated crack detection. Before proceeding, several preliminary settings will be established. These include constraining the drone's flight path choices to prevent it from entering a repetitive back-and-forth loop. For instance, if the drone moves forward in one instance, it will not choose to move backward in the next. Similarly, if it moves diagonally leftward in one instance, it will avoid moving rightward immediately afterward. This control mechanism ensures that the drone continuously moves towards new detection areas during crack capture, avoiding unnecessary backtracking and optimizing efficiency. Each flight session will showcase the captured environment, crack width measurements, the flight process itself, segmentation results of identified cracks, and actions taken based on these detections.

3.2.3.1 Test A

In this test, we'll observe a large sloped area focusing on 5mm-wide cracks (Figure 3.7). The sloped terrain challenges the drone's navigation and crack detection. To ensure accuracy despite the slope, we'll fly at 2 meters altitude. This height balances detailed imaging with stable flight. Using a 3x zoom on the camera enhances crack visibility, crucial for detecting fine details. This setup ensures high-resolution images for precise analysis. Flight stability over varying terrain is maintained through tested control strategies, adjusting altitude and speed for consistent ground clearance. This integrated approach optimizes image quality and detection reliability, crucial for evaluating crack detection performance.

doi:10.6342/NTU202403413

(a) Environment

(b) 5mm

Figure 3.7: Figure (a) depicts the environment of our current test: we conducted the experiment on a wide slope with a flight altitude of 2 meters. Concrete cracks are visible on the surface, and our expectation is that the drone can autonomously navigate along them. Figure (b) illustrates the measured scale of the cracks for this test, which is 5 mm.

The crack we are investigating in this test has an inverted U-shape. The capturing process will begin from the lower left corner of the area, following the crack's trajectory as illustrated in Figure 3.8. By systematically following the path of the crack, we aim to ensure comprehensive coverage and precise identification of its features. Figures 3.9 to 3.18 offer a comprehensive visualization of the crack segmentation process and directional decision-making carried out by the onboard platform during the capturing phase. These figures vividly illustrate how the drone accurately segments cracks and makes real-time flight direction decisions to ensure continuous and effective crack detection. The onboard system's adept performance in these tasks is pivotal, underscoring its role in guaranteeing the reliability and efficiency of the entire crack detection framework.

doi:10.6342/NTU202403413

Figure 3.8: The path of an inverted U-shaped crack starts capturing from the bottom-left corner, moving from point A to J, showing each movement trajectory. Using our trained deep reinforcement learning model to determine directions, coupled with the aforementioned constraint to prevent the drone from flying backward, enables automated capturing of the entire inverted U-shaped crack.

Figure 3.9: At point A, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by forward flight.

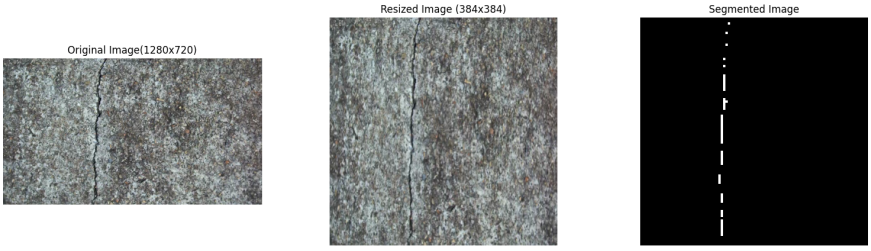


Figure 3.10: At point B, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by forward flight.

doi:10.6342/NTU202403413

Figure 3.11: At point C, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by a flight towards the right.

Figure 3.12: At point D, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by a flight towards the front-right direction.

Figure 3.13: At point E, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by a flight towards the front-right direction.

Figure 3.14: At point F, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by a flight towards the front-right direction.

doi:10.6342/NTU202403413

Figure 3.15: At point G, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by a flight towards the right.

Figure 3.16: At point H, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by a flight towards the back.

Figure 3.17: At point I, the segmentation results of the detected crack are obtained. After detecting the crack, the segmented image data is fed into DDQN (Double Deep Q-Network) for prediction, followed by a flight towards the back.

Figure 3.18: At point J, the segmentation results of the detected crack are obtained. This is the endpoint, so no further flight is necessary.

doi:10.6342/NTU202403413

3.2.3.2 Test B

In this test, we relocated our flight operations to a flat concrete surface specifically chosen to observe a 4mm-wide crack, as shown in Figure 3.19. Given the shaded environment, we utilized a 1x zoom level camera to ensure optimal lighting and clarity, complementing our choice of a 1 meter altitude setting to mitigate potential visibility challenges of the crack. Building on the successful outcomes of our previous test at point A, we employed this setup to assess the versatility and reliability of our system. By testing on diverse surfaces and fine-tuning parameters, our goal was to validate the broader applicability of our automated crack detection framework.

(a) Environment

(b) 4mm

Figure 3.19: For this experiment, we chose a location situated in a shaded area to address lighting issues. To compensate, adjustments were made to the camera focal length and flight altitude. These modifications aimed not only to ensure sufficient lighting and clarity for observing the 4mm-wide crack but also to validate the applicability of our research framework.

The crack pattern in this test presented an irregular and branching structure, initiating crack capture from the bottom right corner of the image, as depicted in Figure 3.20. While our configurations were designed to prevent the drone from retracing its path or entering loops, there remains a possibility of overlooking certain areas of the crack during initial flights. Figures 3.21 to 3.26 provide detailed visual insights into the crack segmentation

doi:10.6342/NTU202403413

process and the dynamic navigation decisions executed by the onboard platform in real-time. These figures illustrate the precision of crack segmentation and the effectiveness of dynamic flight adjustments, ensuring continuous and efficient detection throughout the experiment. The proficiency demonstrated by the onboard system underscores its pivotal role in upholding the robustness and efficiency of our crack detection framework.

Figure 3.20: The crack pattern in this experiment is characterized by irregular branching. However, due to the preceding setup, the drone is prevented from retracing its path. During the process, crack capture will proceed sequentially from points **a** to **f**.

Figure 3.21: At point **a**, upon obtaining the crack segmentation results, the drone determines forward flight direction using DDQN.

doi:10.6342/NTU202403413

Figure 3.22: At point **b**, upon obtaining the crack segmentation results, the drone determines leftward flight direction using DDQN.

Figure 3.23: At point **c**, upon obtaining the crack segmentation results, the drone determines leftward flight direction using DDQN.

Figure 3.24: At point **d**, upon obtaining the crack segmentation results, the drone determines forward flight direction using DDQN.

Figure 3.25: At point **e**, upon obtaining the crack segmentation results, the drone determines forward flight direction using DDQN.

doi:10.6342/NTU202403413

Figure 3.26: At point **f**, once the crack segmentation results are obtained, the drone decides to fly towards the target point and cease flying.

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

Chapter 4 Result and discussion

4.1 Flight Result

In Section 3.2.3, we present our final flight results, which closely align with our initial expectations. We have successfully enabled the drone to perform crack segmentation and determine the presence of cracks using images of localized cracks. Leveraging our prior training, the drone autonomously navigates and executes flight commands, achieving our goal of automated crack exploration. As seen in Figure 3.17, the captured images reveal that cracks are not as simple as initially perceived, often appearing as mesh-like or irregular patches. Despite this variability, our drone demonstrates considerable capability in handling diverse types of cracks, thanks to our carefully designed training methods. This underscores why we opted for deep reinforcement learning training, highlighting its primary advantages. Our training process involves agents continuously improving crack detection and navigation skills through iterative interaction with the environment, ensuring robust performance even in complex scenarios, significantly enhancing the effectiveness of automated crack detection and exploration. Throughout the process, we implemented a specific strategy to prevent the drone from backtracking during crack detection. While this may result in some cracks being overlooked (as shown in Figure 4.1), it prevents the system from getting stuck in repetitive cycles, prioritizing efficiency and operational

doi:10.6342/NTU202403413

stability. The system successfully captured and analyzed 62% of visible cracks, with a detection accuracy rate of 86%. These achievements not only showcase the advancement of our technology but also underscore the potential of deep reinforcement learning in drone-based crack detection. Next, we will explore how to further extend our research and technological capabilities through the equations introduced in the following chapter, addressing more complex engineering applications and challenges.

Figure 4.1: Test A and Test B environments, with red indicating the path where they capture cracks, and blue showing the missed area.

doi:10.6342/NTU202403413

4.2 Validation of Crack Width in Real World

To validate our calculated results, we utilized another dataset from Test B. This validation step was crucial for confirming the accuracy of our formulated relationships and determining the value of α. In Test B, the flight altitude was set at 1 meter with a magnification factor of 1. The actual crack width measured $W_{\text{real}} = 4$ mm, and the segmentation sizes were 4 pixels horizontally (P_h) and 4 pixels vertically (P_v). These specific parameters, along with the crack segmentation image shown in Figure 4.2, provided essential data points to ensure the reliability and consistency of our calculations across varying test conditions.

Recalculating W_h and W_v:

$$W_h = 4 \cdot \frac{2 \cdot 1 \cdot 0.23}{384 \cdot 1} \approx 0.00479 \text{ meters} \approx 4.79 \text{ mm} \tag{4.1}$$

$$W_v = 4 \cdot \frac{2 \cdot 1 \cdot 0.17}{384 \cdot 1} \approx 0.00354 \text{ meters} \approx 3.54 \text{ mm} \tag{4.2}$$

Recalculating W:

$$W \approx \sqrt{(4.79 \text{ mm})^2 + (3.54 \text{ mm})^2} \approx 6.24 \text{ mm}$$

Using $\alpha = 0.63$ to predict W_{real}:

$$W_{\text{real}} = \alpha \cdot W = 0.63 \cdot 6.24 \text{ mm} = 3.93 \text{ mm} \tag{4.3}$$

doi:10.6342/NTU202403413

This calculation closely approximates the updated actual crack width $W_{\text{real}} = 4$ mm for Test B, demonstrating that using $\alpha = 0.63$ provides a reasonably accurate prediction of the actual crack width.

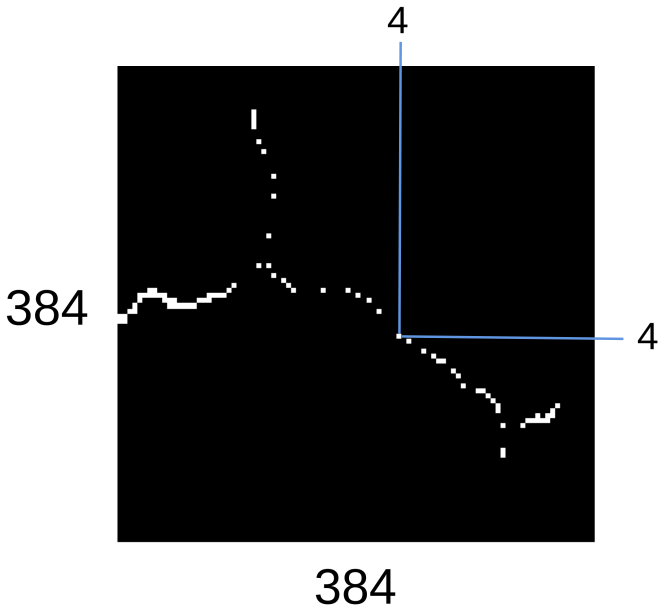


Figure 4.2: The segmented image of the crack along with its width and height measurements after segmentation.

4.3 Limitation

Although our current framework successfully achieves the initial goals—enabling the drone to autonomously navigate and explore cracks through the integration of our model and flight control computer—there are several limitations due to assumptions and system constraints. One major limitation is observed in Figures 4.1, where it becomes evident that when the drone encounters a fork in the path, it can only choose one route to follow. Consequently, any cracks along the unchosen path are ignored. This is a primary reason for the relatively low capture rate observed in physical tests. To prevent the drone from getting stuck in repetitive flight patterns during autonomous exploration, we have implemented a constraint in the flight control system that prevents the drone from back-

doi:10.6342/NTU202403413

tracking. This design choice, while effective in avoiding entrapment, results in missed opportunities to capture cracks on alternate paths. In our server-based model training, we included a feature that addresses this issue by employing a serpentine scanning pattern when the drone has finished inspecting a crack-free path. This approach helps to detect and capture any missed cracks. However, we did not implement this design in the drone's flight environment due to the lack of fixed boundaries in the flight area, which reduces the efficiency of serpentine scanning. The probabilistic nature of this scanning method could result in suboptimal coverage or the drone not returning to previously missed areas. Therefore, our current flight architecture prioritizes overall test efficiency and smooth operation over achieving a higher capture rate, accepting the trade-off of missing some cracks in exchange for avoiding backtracking and ensuring fluid flight operations. In the future, we will attempt to address the current limitations by incorporating different setups before the flight. For example, we can start by zooming out the camera to scan the images for an initial assessment before proceeding with crack capture. Additionally, if the drone encounters a fork in the path during the flight, it can mark the location as a node and return to this node after capturing the cracks to explore the remaining paths.

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

Chapter 5 Conclusion

5.1 Summary

In this study, we extended a deep reinforcement learning model, previously trained by my senior, to enable accurate determination of crack extension directions based on partial crack information and characteristics. Our successful integration of this technology onto a drone platform led to several breakthroughs. We employed the U-Net ResNet model for crack segmentation and utilized the Double Deep Q-Network (DDQN) architecture for iterative training. Additionally, we discussed the nuances between server-based training and testing versus deployment on the drone platform, effectively implementing our model in both scenarios. Furthermore, we developed a flight control system that enabled automated flight of the drone, equipped with an onboard computer platform. This achievement fulfilled our goal of edge computing, enhancing data processing efficiency and addressing issues related to bandwidth and transmission speed during data return. To validate our architecture, we conducted simulations in Gazebo and demonstrated actual flights, confirming the operational feasibility of our system on a drone. These tests underscored the practicality and effectiveness of our approach. By leveraging edge computing, we significantly reduced the time required for subsequent data processing, thereby improving the efficiency of crack detection. Our model autonomously identifies and locates struc-

doi:10.6342/NTU202403413

tural cracks with minimal human intervention, marking a substantial advancement in the field. The successful integration of these technologies into a drone platform demonstrates its potential for real-world applications, ensuring robustness and adaptability across various environmental conditions. This research not only validates the feasibility of real-time crack detection using drones but also emphasizes the synergy between machine learning techniques and hardware-software integration. Extensive field tests and simulations have streamlined the crack detection process, making it more efficient and effective. This study lays the groundwork for future advancements in automated structural health monitoring, contributing to enhanced infrastructure safety and maintenance. Additionally, our drone architecture is adaptable to various research fields, accommodating the integration of different learning models. This flexibility allows it to meet diverse research needs, such as employing various deep learning algorithms for complex image processing and analysis, or applying it to other types of structural health monitoring (e.g., bridges, buildings, roads). By integrating multiple models and techniques, our drone platform holds significant potential for future development, fostering innovation and practical applications. Furthermore, we established an equation describing the relationship between actual crack width and crack segmentation image width. This equation provides a reliable method for accurately estimating real crack dimensions based on our segmentation results.

5.2 Future work

Currently, there are several limitations that this study aims to address through future developments. Moving forward, the study focuses on refining and enhancing several critical aspects. Addressing the pixel disparity between our datasets and actual captured images remains a priority, necessitating more precise image collection using drones to

doi:10.6342/NTU202403413

better align training models with real-world scenarios. Additionally, improving altitude control systems through hardware upgrades is crucial for enhancing flight stability and precision, especially during low-altitude flights vulnerable to air currents. Integrating different machine learning models and optimizing crack detection accuracy further are pivotal for enhancing system performance across varying environmental conditions. These efforts collectively aim to advance the practical applications of drone technology in crack detection and other critical areas.

doi:10.6342/NTU202403413

doi:10.6342/NTU202403413

References

[1] Jianghai Liao, Yuanhao Yue, Dejin Zhang, Wei Tu, Rui Cao, Qin Zou, and Qingquan Li. Automatic tunnel crack inspection using an efficient mobile imaging module and a lightweight cnn. IEEE Transactions on Intelligent Transportation Systems, 23(9):15190–15203, 2022.

[2] Fu-Chen Chen and Mohammad R. Jahanshahi. Nb-cnn: Deep learning-based crack detection using convolutional neural network and naïve bayes data fusion. IEEE Transactions on Industrial Electronics, 65(5):4392–4400, 2018.

[3] Sayyed Bashar Ali, Reshul Wate, Sameer Kujur, Anurag Singh, and Santosh Kumar. Wall crack detection using transfer learning-based cnn models. In 2020 IEEE 17th India Council International Conference (INDICON), pages 1–7, 2020.

[4] Sara Yasmine Ouerk, Olivier Vo Van, and Mouadh Yagoubi. Rail crack propagation forecasting using multi-horizons rnns. In Georgiana Ifrim, Romain Tavenard, Anthony Bagnall, Patrick Schaefer, Simon Malinowski, Thomas Guyet, and Vincent Lemaire, editors, Advanced Analytics and Learning on Temporal Data, pages 260–275, Cham, 2023. Springer Nature Switzerland.

[5] Divya Gupta, Gaurav Goel, Navdeep Kaur, and Dapinder Kaur. Efficient concrete crack detection system using surf and rnn algorithm. 2021.

doi:10.6342/NTU202403413

[6] Eftychios Protopapadakis, Athanasios Voulodimos, Anastasios Doulamis, Nikolaos Doulamis, and Tania Stathaki. Automatic crack detection for tunnel inspection using deep learning and heuristic image post-processing. Applied intelligence, 49:2793–2806, 2019.

[7] Sung-suk Choi and Eung-kon Kim. Building crack inspection using small uav. In 2015 17th International Conference on Advanced Communication Technology (ICACT), pages 235–238, 2015.

[8] Yanxiang Li, Jinming Ma, Ziyu Zhao, and Gang Shi. A novel approach for uav image crack detection. Sensors, 22(9), 2022.

[9] Bin Lei, Ning Wang, Pengcheng Xu, and Gangbing Song. New crack detection method for bridge inspection using uav incorporating image processing. Journal of Aerospace Engineering, 31(5):04018058, 2018.

[10] Yonas Zewdu Ayele, Mostafa Aliyari, David Griffiths, and Enrique Lopez Droguett. Automatic crack segmentation for uav-assisted bridge inspection. Energies, 13(23), 2020.

[11] Xiong Peng, Xingu Zhong, Chao Zhao, Anhua Chen, and Tianyu Zhang. A uav-based machine vision method for bridge crack recognition and width quantification through hybrid feature learning. Construction and Building Materials, 299:123896, 2021.

[12] Wei Ding, Han Yang, Ke Yu, and Jiangpeng Shu. Crack detection and quantification for concrete structures using uav and transformer. Automation in Construction, 152:104929, 2023.

doi:10.6342/NTU202403413

[13] Xinyu He, Zhiwen Tang, Yubao Deng, Guoxiong Zhou, Yanfeng Wang, and Liujun Li. Uav-based road crack object-detection algorithm. Automation in Construction, 154:105014, 2023.

[14] Ruoxian Li, Jiayong Yu, Feng Li, Ruitao Yang, Yudong Wang, and Zhihao Peng. Automatic bridge crack detection using unmanned aerial vehicle and faster r-cnn. Construction and Building Materials, 362:129659, 2023.

[15] 范淳皓. 基於深度強化學習神經網路之自動化裂縫分割與偵測. Master's thesis, Jan 2024.

[16] Yahui Liu, Jian Yao, Xiaohu Lu, Renping Xie, and Li Li. Deepcrack: A deep hierarchical feature learning architecture for crack segmentation. Neurocomputing, 338:139–153, 2019.

[17] Yao Yao, Shue-Ting Ellen Tung, and Branko Glisic. Crack detection and characterization techniques—an overview. Structural Control and Health Monitoring, 21(12):1387–1413, 2014.

[18] P Iakubovskii. Segmentation models pytorch, 2019, github repository, github. URL https://github. com/qubvel/segmentation_models. pytorch.

[19] Edwin Salcedo, Mona Jaber, and Jesús Requena Carrión. A novel road maintenance prioritisation system based on computer vision and crowdsourced reporting. Journal of Sensor and Actuator Networks, 11(1), 2022.

[20] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi, editors, Medical Image Computing

doi:10.6342/NTU202403413

and Computer-Assisted Intervention – MICCAI 2015, pages 234–241, Cham, 2015. Springer International Publishing.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2016.

[22] P Iakubovskii. Segmentation models pytorch, 2019, github repository, github. URL https://github. com/qubvel/segmentation_models. pytorch.

[23] Christopher JCH Watkins and Peter Dayan. Q-learning. Machine learning, 8:279–292, 1992.

[24] Richard Bellman. Dynamic programming. science, 153(3731):34–37, 1966.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[26] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. Proceedings of the AAAI Conference on Artificial Intelligence, 30(1), Mar. 2016.

[27] Sebastián Valladares, Mayerly Toscano, Rodrigo Tufiño, Paulina Morillo, and Diego Vallejo-Huanga. Performance evaluation of the nvidia jetson nano through a real-time machine learning application. In Intelligent Human Systems Integration 2021: Proceedings of the 4th International Conference on Intelligent Human Systems Integration (IHSI 2021): Integrating People and Intelligent Systems, February 22-24, 2021, Palermo, Italy, pages 343–349. Springer, 2021.

doi:10.6342/NTU202403413

[28] Eduardo Assunção, Pedro D Gaspar, Ricardo Mesquita, Maria P Simões, Khadijeh Alibabaei, André Veiros, and Hugo Proença. Real-time weed control application using a jetson nano edge device and a spray mechanism. Remote Sensing, 14(17):4217, 2022.

[29] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In 2004 IEEE/RSJ international conference on intelligent robots and systems (IROS)(IEEE Cat. No. 04CH37566), volume 3, pages 2149–2154. Ieee, 2004.

[30] Johannes Meyer, Alexander Sendobry, Stefan Kohlbrecher, Uwe Klingauf, and Oskar Von Stryk. Comprehensive simulation of quadrotor uavs using ros and gazebo. In Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings 3, pages 400–411. Springer, 2012.

doi:10.6342/NTU202403413