LANDING IN THE LATENT SPACE

BUILDING LABELED SYNTHETIC RUNWAY DATASETS WITH A DATA AUGMENTATION PIPELINE THAT USES DIFFUSION MODELS

BY

GUSTAVO DE PAULA

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

BSC COMPUTER SCIENCE

at the

UNIVERSITY OF LONDON

March 2025

"But test everything; hold fast what is good."

— Saint Paul, First Letter to the Thessalonians 5:21

This work is dedicated

To *Our Lady, Mary*, Mother of God and my mother, in gratitude for her maternal care and intercession throughout this journey.

To *Saint Thomas Aquinas*, the Angelic Doctor, whose love of truth shaped the Christian intellectual tradition.

Among all human pursuits, the pursuit of wisdom is more perfect, more noble, more useful, and more full of joy.

— St. Thomas Aquinas
(Summa Contra Gentiles, Book I, Chapter 2)

ACKNOWLEDGMENTS

Above all, I give thanks and praise to Almighty God, the Most Holy Trinity, without whom nothing has meaning, and without whose grace this journey would not have been possible.

To the Father, source of all life and providence, for sustaining me each day. To Jesus Christ, my Savior and Redeemer, for His mercy and love. To the Holy Spirit, the Sanctifier, for shining light into the twofold darkness I was born into: sin and ignorance.

To my father, Heberth de Paula, who has always been a model of academic excellence and intellectual curiosity, and who has endured great hardships and sacrifices to provide me with the best education possible.

To my dearest, my love, and my sweetheart, Júlia de Paula, my faithful companion, whose love have been the most precious gift I've received in this passing world.

To my family, friends, work colleagues, and teachers, who have supported me in countless ways, and who have always believed in my potential.

To my former employer, Loggi, for taking a chance on me at 17 and investing in my growth—both as an engineer and a student—and whose financial support enabled me to pursue and complete this degree. I'm especially grateful to Davi de Castro Reis, who incentivized me to pursue a degree in Computer Science and whose support and decision enabled that investment on my education.

To all the Angels and Saints who have interceded for me, specially those whom I daily ask for their prayers: Most Holy Mary, Mother of God, St. Joseph, St. Thomas Aquinas, St. Monica, St. Augustine, St. Michael the Archangel, and my Guardian Angel, who never ceased to protect me.

May this work, in whatever good it contains, be for the glory of God and the service of truth.

ABSTRACT

Vision-based landing systems have gained increased attention as cost-effective and autonomous alternatives to traditional radio-based autoland systems, particularly in scenarios where on-ground infrastructure is unavailable or infeasible. A major bottleneck in advancing this technology is the lack of large-scale, diverse, and realistic runway image datasets with pixel-level annotations. While recent works have turned to synthetic data generated from flight simulators, these approaches are limited by their lack of scalability, poor environmental variability, and the high cost of manual annotation.

This paper introduces a novel, open-source, modular data augmentation pipeline, called *Canny2Concrete*, that leverages latent diffusion models with ControlNet to generate high-resolution, labeled runway images. By conditioning image generation on structural features extracted from existing datasets, the pipeline produces realistic synthetic images that preserve critical runway geometry while offering substantial diversity in terms of weather, lighting, and background scenery. A three-stage variant generation process enhances image variety through positional transformations, outpainting, and environmental occlusion effects.

Using this pipeline, a large-scale synthetic runway dataset—*GARD: Gustavo's Awesome Runway Dataset*—has been constructed and evaluated both intrinsically, using the Structural Similarity Index (SSIM), and extrinsically, by training and fine-tuning state-of-the-art detection and segmentation models. Results demonstrate that models trained on the proposed dataset match or outperform those trained on existing synthetic datasets, confirming the viability of diffusion-based augmentation for runway segmentation tasks, and the effectiveness of the *Canny2Concrete* pipeline in generating realistic runway images. The complete dataset, pipeline, and evaluation tools are publicly available to support further research in autonomous aviation and computer vision.

Up to the best of the author's knowledge, and up to date with the literature review conducted in this paper, *GARD* is the largest synthetic runway dataset publicly available, comprising 45,486 images with diverse weather, lighting, background, and runway occlusion conditions. *GARD*, along with trained segmentation model weights and evaluation results, is available at: https://www.kaggle.com/datasets/depaulagu/gard2025

The code for the project, including the implementation of the *Canny2Concrete* pipeline, is available at: https://github.com/gustavo-depaula/GARD

PROJECT INFORMATION

For the project template, I have chosen the *"Gather your own dataset"* template. I have chosen to constrain my project by word limits:

- **Introduction:** 871 out of 1000 words
- **Literature Review:** 2465 out of 2500 words
- **Design:** 1122 out of 2000 words
- **Implementation:** 1716 out of 2000 words
- **Evaluation:** 1168 out of 1500 words
- **Conclusion:** 386 out of 1000 words
- **Total:** 7728 out of 9500 words

I have excluded code, titles, subtitles, page numbers, references, appendices, and all front matter from the word count. Word count was done using the `texcount` tool, with the following command: `texcount -sub=chapter -inc LandingInTheLatentSpace.tex`.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

SSIM Structural Similarity Index Measure

GARD Gustavo's Awesome Runway Dataset

mAP Mean Average Precision

Part I

INTRODUCTION AND BACKGROUND

1

INTRODUCTION

Aviation is regarded as highly safe medium of transportation, and is marked by high degrees of automation, reducing the biggest cause (85% of the general aviation crashes) of accidents: pilot error [23]. Increased automation reduces cognitive, fatigue, and inexperience risks for pilots and is an important factor, alongside better training and safety regulations, for the rapidly decrease (1959-2024) in fatal accidents rate [2].

Although the approach and landing phases are the minority of the flight time, they contribute disproportionately to accidents, corroborated by major commercial airplane companies Boeing (7% in approach, 36% in landing) [8], and Airbus (59% in landing, 11% in approach) [1].

Contributing as a risk-factor, the approach and landing phases are flight phases requiring human intervention. This has led to an increased interest in the development of autonomous landing (autoland) systems, which autonomously navigate civil aircraft or UAVs (e.g. drones) during landing. Currently, most autoland systems are based on radio signals that provide guidance to the system, such as ILS (instrument landing system) and PAR (precision approach radar).

Radio-based autonomous landing systems allow landing in extremely adverse weather conditions and low visibility, but they have a high-cost of deployment and maintenance, can suffer from electromagnetic and radio interference, and require on-the-ground specialized equipment to support the aircraft (e.g. localizer and glideslope).

The recent advancements in the Computer Vision field sparked interest [3] in developing vision-based autoland systems, which use visual navigation to guide the aircraft during the approach and landing phases. In [40], the authors describe key advantages of vision-based autoland systems for UAV: autonomy, low cost, resistance to interference, and ability to be combined with other navigation methods for higher accuracy. Vision based landing is especially attractive for drones often landing in extreme military, environmental, or disaster relief situations, where runways may not have the necessary equipment for radio-based systems.

Two key parts of an autonomous landing system is detection and segmentation of runways, the former detects and locates the runway by drawing a bounding box, and the latter, more granularly, works at the pixel-level, identifying the object's exact shape.

Detection and segmentation methods can be divided into two fields: traditional methods and deep-learning based methods. Traditional

methods employ handcrafted features and mathematical rules to segment images, e.g. texture, line, and shape features of runways [6, 42]. They tend to be faster to run and not require training, but because they require handcrafted rules, they often fail to generalize to out-of-sample images and complex real-world scenarios that involve similar objects like roads and multiple runways and adverse weather conditions.

Compared to traditional methods, deep-learning based ones generalize and perform better on out-of-sample images, with the latest published runway segmentation papers all having used this approach and getting better results than traditional methods [10, 38].

However, because of their data-hungry nature, previous research [10, 11, 16, 38] all note the lack of publicly available, high-quality, large, real-world datasets of aeronautical images for runway segmentation. To close this gap, the researchers have built and published datasets of synthetic images from flight simulators such as X-Plane [11, 38], Microsoft Flight Simulator [10], and images collected from Google Earth Studio [16].

While these synthetic datasets have contributed to advancements in the field, there remains an opportunity for an open-source, realistic, high-quality dataset that has runway image data covering more varied weather, lighting, environments, and structures, and, even better, an open-source image generator that is capable of generating novel images or augmenting an existing dataset.

In recent years, striking advancements were made in the field of image generation models, with many open- and closed-source models being published for general use, such as DALL-E [7], Midjourney [28], Kandinsky [5], and Stable Diffusion [32]. These recently state-of-the-art models are latent diffusion models (LDM), which outperform previous GAN-based techniques. These image-generation models are now so capable of generating realistic images that researchers are studying the impacts of generative AI on the spread of fake news [25].

These recent research advancements opened new opportunities for synthetic data to close the gap on the lack of real images in object detection and segmentation tasks. This idea has progressed in areas such as medical images [34], urban applications [31], and apple detection in orchards [36]. But up to my knowledge of public research, there has been no generative AI-based open-source dataset for runway detection and segmentation.

To address the challenges of lack of data in the context of runway segmentation tasks, this paper introduces a novel, open-source, data augmentation technique based on a multi-step Stable Diffusion pipeline that extracts features from datasets and generates structurally similar, customizable images (scenery, weather, lighting), guided by a text prompt. This data augmentation pipeline is named *Canny2Concrete*. The images generated by the pipeline are already labeled and don't require handcraft labelling.

This technique is then used to construct a novel, large-scale, high-resolution, open-source, dataset of labeled runway images that covers image variants in weather, lighting, background, and occlusion. This dataset is named *GARD: Gustavo's Awesome Runway Dataset*.

This approach greatly increases runway image availability for research, and, by virtue of being open-source, allow other researchers to generate and augment their own synthetic datasets with their own desired characteristics.

This new dataset is evaluated theoretically via similarity metrics like Fréchet Inception Distance (FID) and Structural Similarity Index (SSIM), and practically by training detection and segmentation models and comparing performance of the model when training it with an existing dataset.

2

LITERATURE REVIEW

2.1 DEEP-LEARNING BASED RUNWAY SEGMENTATION

In [4], the authors give a broad literature review of the prior work on runway segmentation using traditional and deep-learning methods. They break down the traditional methods into two categories: template-based and feature-based approaches, the former compares a template image to the actual one pixel-by-pixel, and the latter uses edges, corners, texture, and others to detect and localize the runway. They find that most works that use deep-learning methods are focused on airport detection, and not runway segmentation, highlighting the paper's relevance.

They then propose a two-module pipeline, where the first module is responsible for detecting runway presence and the second module is responsible for localizing it. For the detection module, they fine-tune a pre-trained ResNet50 model, achieving an accuracy of 97%. And for the localization module, they experiment with three approaches: Hough transform, line segment detector, and a CNN, achieving a 0.8 mean IoU (Intersection-over-Union) score.

The study shows that deep-learning methods are a valid and effective pathway to runway detection and segmentation, but there are significant shortfalls with it for practical vision-based landing systems. The top-down perspective from the satellite dataset images (*"Remote Sensing Image Scene Classification"* [12]) is unrealistic for fixed-wing aerial vehicles' approach and landing, because the onboard system needs to detect and segment runways from the perspective of the aircraft. Secondly, there are no discussions on dataset diversity such as adverse weather and lighting conditions and the performance of the pipeline in those cases.

In [11], the authors note the lack of large-scale, publicly available datasets for the field of runway segmentation. Trying to alleviate this problem, they propose *"BARS: A Benchmark for Airport Runway Segmentation"*, a 10,256-image labeled runway dataset, with images collected from X-Plane, a FAA-certified flight simulator. The images were collected from several simulated flights under different weather conditions and at different times, across 40 airports, to generate a diverse dataset far more suitable for the task of vision-based landing than the one used in [4].

To test the efficacy of this new dataset, they experiment with several segmentation methods (e.g., Mask R-CNN, YOLACT, SOLO) and report the trained models' performance, which have a wide range of

reported precisions (AP50 of 90.98% for the best performing one and 62.18% for the worst performing one).

At the same time, using a simulator for generating synthetic images limits scenario diversity (e.g., it is not possible to create scenarios in airports that are not included in the simulator). Also, because it is a closed-source simulator, it is not easy or accessible for other researchers to expand on this dataset by adding more diverse and unseen scenarios. And the manual labeling process using LabelMe [27] also increases the cost of reproducing or expanding the dataset. Another problem of the work is that the authors decided to publish their work on Baidu, which prevents access to the full dataset without installing a third-party program on the computer.

The way the authors decided to test *BARS* experimentally highlights a situation that is similar to the problem encountered by Nobel-winning economist Eugene Fama in his work on Efficient Markets [17]: the joint hypothesis problem. The joint hypothesis problem is the fact that all tests of market efficiency are simultaneously tests of market efficiency and the asset pricing model that defines expected returns. Therefore, anomalous market returns might be due to market inefficiency, an inaccurate model, or both.

Similarly, when proposing new datasets, one has to always be mindful that empirical tests of training models on these new datasets *are always a joint test of dataset quality and model performance*. A model's poor performance might indicate deficiencies in the dataset (e.g., lack of diversity, poor annotation quality, or unrealistic synthetic images), a reflection of the model's limitations in handling a more realistic dataset with more complex tasks, or both. On the other hand, if a model performs really well, it doesn't automatically prove that the dataset is good. It could mean the dataset simply matches the model's strengths, without really testing how well it would work on real-world images.

In [10], the authors propose *"ERFE: efficient runway feature extractor"*, a runway detection model that is able to extract semantic segmentation and feature lines. Also highlighting the difficulties of runway datasets, the authors propose a new synthetic image dataset *"FS2020"*, using images from Microsoft's Flight Simulator 2020. They did have access to BARS but argued that the images from X-Plane were unrealistic, especially in regards to ground texture and lighting conditions. Their proposed dataset contains 5,587 high-resolution (1920x1080) images, sampled from different runways, airplane positions, and lighting and weather conditions.

After image collection, the authors used the LabelMe toolbox to provide two types of annotation for each image: segmentation masks and feature lines with 6 categories (left edge, right edge, center line, aiming point front, threshold rear, and PAPI lights).

The authors highlight the need for fast and accurate inference in the context of a fast-moving airplane. Thus, they chose to build a deep-learning model based on MobilenetV3, a convolutional neural network designed for mobile phone CPUs. They claim that their trained network has the capacity of processing 200 high-resolution images per second.

Their work excels in demonstrating the feasibility of an onboard runway segmentation system, and their FS2020 dataset is a rich contribution to the field, especially as it is accompanied by segmentation and feature lines labels. At the same time, it is a smaller dataset when compared to BARS, and because it is also based on a closed-source simulator, it has the same trade-offs associated with it. The authors also didn't compare their model's performance when trained with another dataset like BARS, which would make it easier to understand how well their dataset generalizes compared to others. Without this comparison, it is unclear whether their model performs well because of the dataset's quality or simply because the dataset aligns well with the model's training conditions. On the other hand, they hosted the dataset on Kaggle, a widely used platform for hosting public datasets.

In [16], the authors still highlight the lack of open-source datasets of aerial images of runways and present, the *"LARD: Landing Approach Runway Detection"* dataset, a novel 17,000-image dataset alongside an image generator. They use Google Earth Studio, positioning a camera inside the studio in the perspective of an airplane nose pointing to the runway. They publicly shared their generator scripts that automatically output labeled images without the need for human intervention. Alongside the generated synthetic data, they manually labeled real videos from airplanes landing.

Their method has considerable advantages over simulator-based ones: it is possible to reproduce and generate new images for virtually free, as Google Earth Studio is a closed-source but free tool, with lower need for manual labeling. On the other hand, the images are less realistic than the simulator-based ones, as the ground texture is worse and it lacks different weather conditions, and night view is simulated by a simple reduction in ambient brightness.

The authors don't train any detection or segmentation models based on the *LARD* dataset in the paper, but [24] did introduce YOLO-RWY, a YOLO-based [30] deep-learning model, and trained it using LARD, reporting that it has *"strong generalization and real-time capabilities"* [24]. The authors highlight how the limited nighttime and adverse weather samples in LARD may affect performance in extreme conditions, and they did include a data augmentation step in their training pipeline.

In [38], the authors built VALNet, a model based on YOLO that uses band-pass filters to be able to handle large-scale changes and input image angle differences in the context of runway segmentation. Among the reviewed papers on runway segmentation models, it is

by far the most advanced, with a novel architecture and extensive comparisons to other models, such as YOLOv8 and Mask R-CNN.

The paper also cited the dataset scarcity challenge and proposes a new dataset called *"RLD (Runway Landing Dataset)"*, with 12,239 images with a resolution of 1280x720. The images are sourced from X-Plane, similarly to the already reviewed BARS. The dataset was also manually labeled using LabelMe, and the dataset is also hosted on Baidu. Although it is the largest simulator-based dataset reviewed in this paper, it has the same advantages and disadvantages as previously reviewed simulator-based datasets.

2.2 IMAGE GENERATION MODELS

GANs (generative adversarial networks) were introduced by [18], and they consist of two neural networks, the Generator and the Discriminator, engaged in adversarial training. The generator is responsible for creating synthetic images (in the context of image GANs) and the discriminator is responsible for evaluating the authenticity of images. During training, the generator gets better at creating realistic images and the discriminator gets better at differentiating real from synthetic images.

In [13], the authors give a survey on the main issues of GANs and their applications. They show how GANs can be effectively used for data augmentation, and with other GAN architectures such as Semi-supervised GAN (SGAN), there can be a model that outputs labeled images. But GANs suffer from a well-known problem called "mode collapse," where the generator learns to produce only one or a few specific patterns that fool the discriminator, making the range of images generated by the model less diverse.

Diffusion models, introduced in [20], are a generative approach based on iterative denoising. Diffusion models work by progressively adding noise to an image (called the forward process) and then training a network that learns how to remove noise from the image (the reverse process).

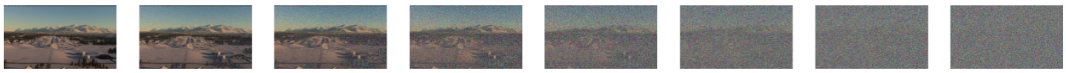

Figure 2.1: Example of Diffusion forward process adding noise to an image

The intuition of Diffusion models is that, if a model can be trained to predict the noise in an image at a *timestep*, we can start at pure noise, then repeatedly call this model and remove the noise from the image, at each step making a less noisy image.

A key architecture used in diffusion models is U-Nets, introduced in [33]. The U-Net is composed of two parts: an encoder and a decoder. The encoder transforms the image into a compressed form that retains

essential features. This compressed data is called a "latent". The decoder can then operate on this latent and output some data related to the input data. In the original paper, they used the U-Net to extract biomedical segmentation data. In Diffusion models, the U-Net is used as the model that predicts the noise from an image.

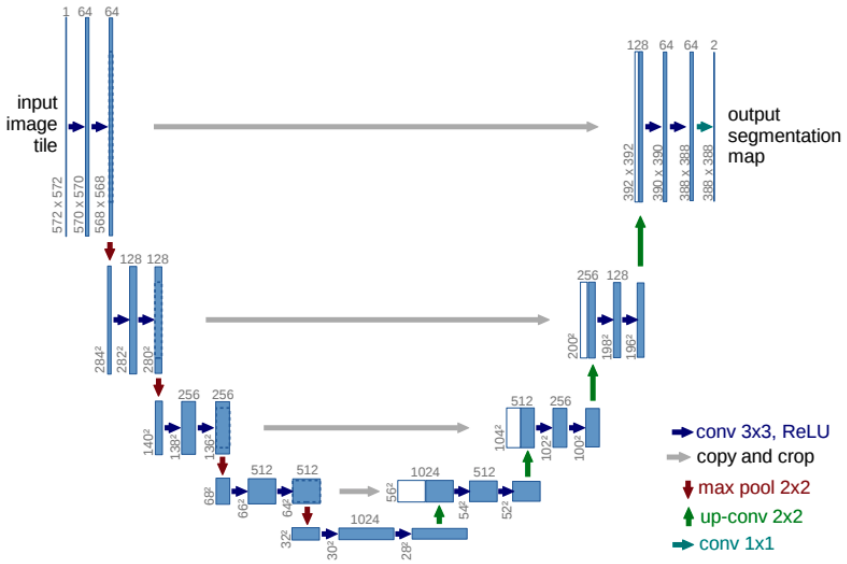

Figure 2.2: U-Net architecture. Figure from [33]

Although recently diffusion models have emerged as the new state-of-the-art architecture for image generation [41], they do have disadvantages. Namely, their inference time is slower than that of GANs, they have a higher computational cost, and by themselves, they lack mechanisms for precise editing of the image aside from the text prompt.

This problem of controlling the generated image was tackled by [43], which introduced ControlNet. ControlNet is a neural network that allows spatial conditioning to pre-trained text-to-image models. With it, it is possible to use canny edges, human poses, and segmentation masks to control the final result of the image. This allows greater control over the generation of synthetic images, such as the positioning of a runway in an image or control over the markings and detailed paintings on the runway.

2.3 SYNTHETIC DATASETS BUILT WITH DIFFUSION MODELS

In [34], the authors cite a similar data scarcity problem in the field of medical images, specifically, gastrointestinal images. To solve this problem, they built a pipeline that used diffusion models to generate labeled gastrointestinal polyp images.

They started by clustering the training images and masks into 20 different clusters, and after that, training a four-channel (3 for RGB and a binary one for the mask) DDPM (Denoising Diffusion Probabilistic Model) for each cluster. The models were trained with a fourth channel so that the model outputs an image and the associated mask along with it, requiring no human labeling. They used the RePaint [26] technique to guide the diffusion process so that the polyp was always generated in a specific area of the images, and they also used styling techniques so that the generated images were realistic.

They compared the diffusion-generated images with GAN model-generated images and found that the diffusion images were closer to real ones. The study trains several diffusion models to address the problem of large variation between images in the dataset, making each model specialize in its cluster. The paper shows promising results that we can augment image datasets with diffusion models, but only low-resolution images were generated (256x256 pixels) and a larger variety of images would require training more independent models.

In [31], the author explored using text-to-image diffusion models to generate urban traffic images for vehicle detection and classification. The author used the trained Kandinsky 2.2 model to generate images using *prompt engineering*. 192 different prompt variants were used to generate 1000 images with different combinations of traffic density, type of vehicle, location, weather condition, time of day, and camera location. The paper shows how far off-the-shelf models are capable of generating realistic images that can be used to train detection algorithms. But it does not compare images generated by different models, and the images have to be manually labeled.

In [36], the authors studied the effectiveness of fine-tuning a pre-trained Stable Diffusion model for the purpose of generating datasets, applied to apple detection in apple orchards. They separated their baseline dataset into two datasets: green apples and red apples. After that, they fine-tuned two Stable Diffusion models with DreamBooth and then generated a whole dataset. They used a trained apple detection model for baseline image annotations and then manually refined these annotations. To experimentally test the effectiveness of their datasets, they trained multiple YOLO [30] object detectors on the baseline and synthetic datasets and compared the results. In their study, object detectors performed similarly when trained on their synthetic dataset and when trained on real image datasets. Their approach shows promising results in dataset generation with Stable Diffusion, although there is a lack of diversity (e.g., weather conditions, lighting conditions, different backgrounds, etc.) in both the baseline dataset that makes it easier to generate a synthetic dataset that is similar to the original.

2.4 EVALUATION OF SYNTHETIC DATA

Across the previously mentioned works, two types of evaluation were common: theoretical similarity and experimental performance. Theoretical similarity was assessed using metrics such as FID (Fréchet Inception Distance) [19] and SSIM (Structural Similarity Index) [39], which measure the similarity between two images. Theoretical similarity was used to evaluate the synthetic datasets in [34]. Experimental performance, on the other hand, is about training a model on the synthetic dataset and comparing the results. Experimental performance was used in all runway segmentation papers, except for LARD, and also in [31, 36].

Both forms of testing are valid and have their trade-offs. Theoretical similarity is faster and easier to measure but heavily depends on what images are included in the test. And if a dataset contains very different image structures than the original, even if they are high-quality, the FID/SSIM give worse values. On the other hand, experimental performance tests the dataset in a realistic setting, evaluating how it will be used by other researchers, but suffers from the joint dataset-model problem covered previously.

Part II

METHODOLOGY AND DEVELOPMENT

3 DESIGN

3.1 PROJECT OVERVIEW

The literature review has shown the importance of large-scale, public, and diverse datasets for deep-learning projects, specifically in the field of runway detection and segmentation, where there is currently a lack of a dataset of real images with these characteristics. To address this challenge, previous work in the field has relied on synthetic data, mainly collected from flight simulators, which poses a significant cost barrier to the expansion of these datasets as these simulators are usually paid and closed-source, and the collected images require manual labelling.

At the same time, the emergence of diffusion models as the new state-of-the-art technique in image generation has opened a new window of opportunity in building synthetic datasets. Although this idea of using diffusion models to build synthetic datasets has been explored in other fields, there is no known work applying it to the runway segmentation research field.

This project's primary research question is *how can a suitable synthetic image dataset be built for computational vision tasks without the need for images extracted from simulators or similar solutions*. To answer this question, the project uses the field of vision-based landing and builds a synthetic runway image dataset.

The primary users of this project are researchers who may use the dataset, Gustavo's Awesome Runway Dataset (GARD), provided with this project to build models for runway detection and segmentation. Secondary users might be researchers interested in synthesizing their own datasets for computational vision tasks, such as classification or segmentation, using the new *Canny2Concrete* pipeline architecture, adapting it for their own needs.

With these end-users in mind, this project delivers a two-fold contribution: a novel modular data augmentation pipeline that can increase the diversity of an existing dataset, while maintaining key features and structures; and using this data augmentation pipeline to generate a fully synthetic runway images dataset based on existing public datasets.

The proposed research question begs the question of what constitutes a "suitable synthetic image dataset". Here, we rank the following characteristics that make an image dataset suitable:

1. **Human-judged realism**: humans subjectively judge the images as credible and realistic. When comparing side-by-side the im-

ages of the generated dataset with other datasets, the perceived quality of the images is the same or better.

2. **Detection by existing models**: fine-tuned models on available datasets can detect the presence and segmentation of the desired data (runways in this project) in the images.
3. **Data diversity**: the dataset has good data diversity, including edge cases and several data variations.
4. **Model training**: an existing architecture can be trained on the dataset and achieve reasonable performance or a pre-trained model can be fine-tuned and have its performance improved.

3.2 DATA AUGMENTATION PIPELINE

In earlier prototypes, several diffusion techniques such as prompt engineering, prompt weighting, unconditional image generation, inpainting, and textual inversion were tried. Most of them failed to, standalone, generate realistic runway images with detailed and accurate markings. The most successful attempt was using ControlNet with an input canny edge, alongside well-crafted prompts to guide the text-to-image model. Because the canny edge was extracted from an existing runway image, the generated image faithfully respected the runway shape, position, markings, and texture. Thus, the data augmentation pipeline is based around using a text-to-image diffusion model with ControlNet.

The data augmentation pipeline is composed of four separate modules, that can each be independently developed and improved, or totally replaced without affecting the overall functioning of the pipeline. This modularity and separation of concerns is good software engineering practice and aids in future research developing on this project.

template image selection module : this step selects the "template images" that will be used in the pipeline. The template images are existing images in public datasets that are used to extract the overall structure of the image containing the runway to be used in the image generation step. Ideal template images have clear visibility of the runway and the surrounding background. The output of this module is a folder with pairs of image files and label files.

edge extraction module : this step uses the template images as input, processes them, and outputs canny edge images, to be used as inputs for ControlNet, alongside the according label for that image.

base image generation module : this step uses as inputs the canny edges, a list of text prompts, and a number of how many

images should be generated for each pair of canny edge and prompt. It then uses a text-to-image model with ControlNet to generate base images.

variant image generation module : this module uses the base images as input and applies a series of transformations to them, such as applying rotations and occlusion effects to the image.

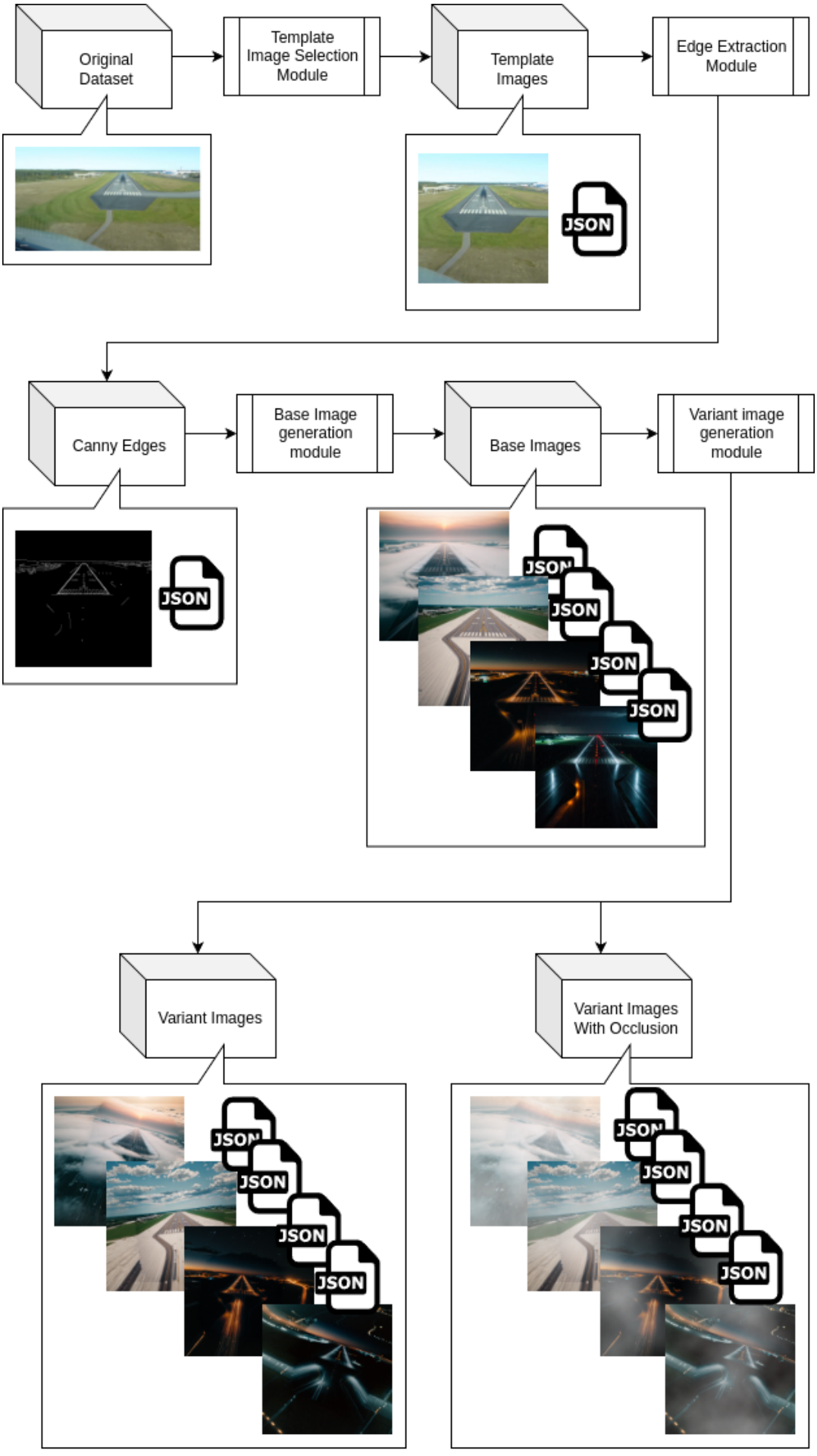

Figure 3.1: Diagram of the *Canny2Concrete* pipeline, with real images as examples.

3.3 EVALUATION

Following the standards of evaluation of synthetic datasets presented in the literature review, and our definition of what constitutes a suitable synthetic dataset, two types of evaluation will be done on the dataset: intrinsic and experimental evaluation.

Intrinsic evaluation measures the quality of a dataset by its intrinsic characteristics, such as image realism, resolution, diversity, and size. Ideally, to judge human-based realism, a group of subject experts would be assembled to judge the images, but because of time constraints, this won't be done.

Instead, it will be replaced by metrics of image similarity between template images and the generated base images. Through metrics such as FID (Fréchet Inception Distance) and SSIM (Structural Similarity Index), it is possible to determine how similar or different the images are from one another, and if they retain similar structures, such as the runway shape, position, markings, and background.

Other metrics such as size and image resolution will be reported. Data diversity will also be reported based on the generated images' text prompts. This doesn't guarantee full accuracy as there is a possibility of the model "hallucinating" and not generating a certain desired characteristic (e.g., rain, snow, runway occlusion), but nonetheless allows a good estimate to be reported.

Experimental evaluation is about using the dataset to train one or a collection of models and evaluating their performance empirically. It is an imperfect measurement, as there are a lot of moving parts when training a model, and, as seen in the literature review, poor performance might be due to the dataset quality or an inherent flaw in the model's architecture. However, this type of evaluation is still widely used and a standard in testing synthetic datasets.

As it is not the focus of this project to build a new architecture for runway segmentation, the experimental evaluation will be restricted to training and fine-tuning existing detection and segmentation models, such as YOLO [21].

Using a pre-trained segmentation model, comparisons will be made between the performance of training on this new synthetic dataset and on existing datasets, and the resulting performance when detecting real images in a dataset. The main metric used will be Mean Average Precision (mAP), at different levels of recall thresholds.

4 IMPLEMENTATION

4.1 PIPELINE

The pipeline is implemented through four Jupyter notebooks, one for each module. By convention, the file names are prefixed with `p_` to indicate they are part of the pipeline.

4.1.1 *Template image selection module*

The template image selection module contains an adapter function that reads images from the public LARD dataset [16], generates JSON label files, and places the image-label file pairs into the output directory. The LARD dataset is composed of several different folders containing both synthetic and real images. It has two real images folders: *Nominal cases* and *Edge cases*, the latter being composed of images with poor runway visibility.

Because the template images are used to extract canny edges, it is better to use images with clear runway and surrounding area visibility. Hence, the *Nominal cases* folder, containing 1500 images, was chosen.

The images are then processed into square format, since Stable Diffusion by default works with square images. The runway is horizontally centered so it will not be out of frame when cropping into a square. The following code does this:

Listing 4.1: Template image selection module, image pre-processing

```
xs = [p[0] for p in pts]
ys = [p[1] for p in pts]
min_x, max_x = min(xs), max(xs)
min_y, max_y = min(ys), max(ys)

# Vertical crop is the entire height
top = 0
bottom = H

# Horizontal positioning for H-wide crop
center_x = (min_x + max_x) / 2.0
ideal_left = int(round(center_x - (H / 2)))
left = max(0, min(ideal_left, W - H))
right = left + H

cropped_img = img.crop((left, top, right, bottom))
```

Then, the new shifted runway corners' keypoints are computed to build a JSON label:

Listing 4.2: Template image selection module, label building

```
shifted_pts = [(p[0] - left, p[1] - top) for p in pts]

# Build label
image_label = ImageLabel(
    dataset="LARD/LARD_test_real_nominal",
    sourceImage=os.path.basename(image_path),
    runwayLabel=shifted_pts
)
```

Finally, the image label is converted to JSON, and both the cropped images and their labels are saved to the output folder.

4.1.2 *Edge extraction module*

The edge extraction module contains a generator function that takes an input directory and outputs a directory with corresponding canny edge images at a 1024×1024 resolution. The generator function uses an image processor from a ControlNet Auxiliary library to generate the canny edges, and then resizes to the desired resolution:

Listing 4.3: Edge extraction module, edge extraction

```

from controlnet_aux.processor import Processor

processor = Processor(
    'canny',
    {
        "detect_resolution": 756,
        "image_resolution": 1024
    }
)

pil_in = Image.open(image_path).convert("RGB")
canny_pil = processor(pil_in, to_pil=True).resize((1024, 1024))
```

To help in the image generation process, a polygon is drawn onto the image to delimit the area of the runway:

Listing 4.4: Edge extraction module, polygon drawing in canny edge image

```
cv2.polylines(
    canny_array,
    [corners_sorted.astype(np.int32)],
    isClosed=True,
    color=(255, 255, 255),
    thickness=2
)
```

Finally, the corners are scaled to this new image resolution and saved to the new label file:

Listing 4.5: Edge extraction module, scaling corners to new resolution

```
orig_h, orig_w = image_bgr.shape[:2]
corners_array = np.array(runway_corners, dtype=np.float32)
corners_array[:, 0] *= (1024.0 / orig_w)
corners_array[:, 1] *= (1024.0 / orig_h)
```

4.1.3 *Base Image generation module*

This is the most important module of the pipeline, as everything depends on the quality of the base images generated in this step. To generate the images, the `diffusers` library [14] is used, providing utilities for generating images with pre-trained diffusion models.

The classes one needs from `diffusers` depend on which model is in use. The DreamShaper model [15] is a general-purpose Stable Diffusion model, meant to compete with other general-purpose models such as DALL-E and Midjourney. Of several tested models and pipelines, DreamShaper was chosen for its image quality, faithfulness to the prompt, and ability to construct diverse scenarios (ranging across different lighting and weather conditions).

From the DreamShaper family, we chose the DreamShaper XL version, capable of producing 1024×1024 images, as it is based on Stable Diffusion XL. This higher resolution was necessary to generate finer details such as runway markings.

Having selected DreamShaper XL, it is required to choose a compatible ControlNet model and scheduler configuration. For the ControlNet model, the pre-trained canny-edge-based ControlNet offered by `diffusers` for Stable Diffusion XL was used, and the recommended scheduler configuration for DreamShaper was applied. Enabling CPU offloading was critical to avoid running out of GPU memory:

Listing 4.6: Base Image generation module, model and scheduler configuration

```
controlnet = ControlNetModel.from_pretrained(
    "diffusers/controlnet-canny-sdxl-1.0", torch_dtype=torch.float16
)

pipeline = StableDiffusionXLControlNetPipeline.from_pretrained(
    "lykon/dreamshaper-xl-v2-turbo",
    torch_dtype=torch.float16,
    variant="fp16",
    controlnet=controlnet
).to("cuda")

pipeline.scheduler = DPMSolverMultistepScheduler.from_config(
    pipeline.scheduler.config,
    algorithm_type="sde-dpmsolver++",
    use_karras_sigmas=True
)

pipeline.enable_model_cpu_offload()
```

A critical part of generative models is *prompt engineering*: crafting an input prompt that achieves the desired result. Diffusion models accept both a *prompt* and a *negative prompt*. The model tries to produce what is mentioned in the prompt and avoid what is mentioned in the negative prompt.

Crafting the prompt is a process of trial and error until satisfactory results are found. In practice, it is helpful to look at prompts used in the model's example pages to see keywords the model responds strongly to, such as `cinematic film still`, `realistic`, `ugly`, or `deformed`.

Since this project needed to create images under diverse scenarios, after trying many different prompts, the following prompts were chosen for their ability to generate realistic runway images with different weather and lighting conditions.

Two base prompts were defined for daylight images with no weather adversity, and a dictionary of modifiers was used to add or remove keywords to achieve weather or time-of-day effects:

Listing 4.7: Base Image generation module, prompt definitions

```
base_prompt = "photo of airport runway, aerial view, 4k, cinematic film
    still, realistic, beautiful landscape around, high-contrast runway
    lines"

base_neg = "airplane, ugly, low-quality, ugly background, ugly airstrip,
    deformed, dark, noisy, blurry, low contrast, missing lines,
    unrealistic, drawing, objects on runway"

modifiers = {
    "rain": (
        "+raining +storm +wet",
        "-dark -noisy -blurry",
    ),
    "fog": (
        "+(harsh fog) +mist +haze",
        "-dark -noisy -blurry",
    ),
    "snow": (
        "+snowing",
        "",
    ),
    "dusk": (
        "+(at dusk)",
        "",
    ),
    "dawn": (
        "+(at dawn)",
        "",
    ),
    "night": (
        "+(at night)",
        "-dark",
    )
}
```

A helper function (e.g., `apply_modifiers`) builds the final prompts (and negative prompts) from the base prompts and the modifiers. It also receives the number of images to generate, returning a data structure that can be passed to `generate_base_images`, which uses the prompt pairs to generate and save images:

Listing 4.8: Base Image generation module, generating base images

```
generate_base_images(
    "p_FilteredCannyEdges",  # input directory
    "p_BaseImages",          # output directory
    prompt_pairs=[           # images to generate
        apply_modifiers("day", [], 5),
        apply_modifiers("night", ["night"], 5),
        apply_modifiers("dusk", ["dusk"], 1),
        apply_modifiers("dawn", ["dawn"], 1),
        apply_modifiers("fog", ["fog"], 1),
        apply_modifiers("fog+night", ["night", "fog"], 1),
        apply_modifiers("rain", ["rain"], 1),
        apply_modifiers("rain+night", ["night", "rain"], 1),
        apply_modifiers("snow", ["snow"], 1),
        apply_modifiers("snow+night", ["night", "snow"], 1),
    ],
    model_name="sdxl-dreamshaperxl",
    # show=True
)
```

4.1.4 *Variant image generation module*

To generate variant images, three pipelines are run: a positional variant augmentation, outpainting the borders, and weather occlusion effects.

The first pipeline uses the *Albumentations* library [9] to run three transformations: (1) random horizontal flip, (2) padding with a border (to simulate a more distant runway), and (3) slight random rotation from $-25°$ to $+25°$:

Listing 4.9: Variant image generation module, Albumentations pipeline

```
pipeline = A.ReplayCompose(
    [
        A.HorizontalFlip(p=0.5),
        A.CropAndPad(
            px=((51, 410),  # top
                (51, 410),  # bottom
                (51, 410),  # left
                (51, 410)), # right
            keep_size=False,
            p=1.0
        ),
        A.Affine(rotate=(-25, 25), p=1.0),
    ],
    keypoint_params=A.KeypointParams(format='xy', remove_invisible=False)
```

```
)
```

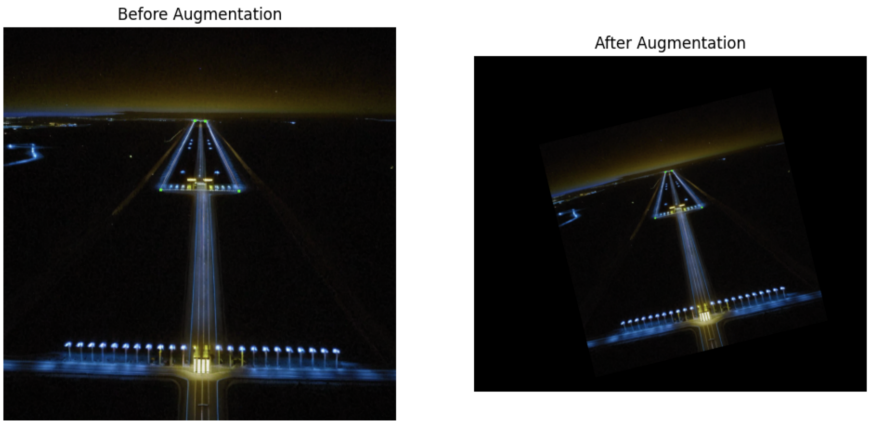

Figure 4.1: Applying the *Albumentations* pipeline to an image.

This first pipeline generates images with black borders. The second pipeline uses *inpainting* to fill in these black borders. First, OpenCV-based inpainting is used, filling the black border with colors from the nearby region. Then, a Stable Diffusion model refines the borders for better blending.

OpenCV-based inpainting is used because Stable Diffusion inpainting works better when the inpainting region is filled with a rough approximation of the missing content.

OpenCV implements Alexandru Telea's inpainting algorithm [35] via `cv2.inpaint` with the `cv2.INPAINT_TELEA` flag:

Listing 4.10: Variant image generation module, OpenCV-based inpainting

```
mask = np.all(image == 0, axis=2).astype(np.uint8) * 255
kernel = np.ones((3, 3), np.uint8)
mask = cv2.dilate(mask, kernel, iterations=4)
image = Image.fromarray(cv2.cvtColor(
    cv2.inpaint(image, mask, 3, cv2.INPAINT_TELEA),
    cv2.COLOR_BGR2RGB
))
```

After this initial inpainting, a Stable Diffusion XL Inpainting pipeline is used to process the image. The original prompts used for the base images are reused, along with a blurred mask for smoother blending:

Listing 4.11: Variant image generation module, Stable Diffusion XL Inpainting pipeline

```
pipe = StableDiffusionXLInpaintPipeline.from_pretrained(
    "lykon/dreamshaper-xl-lightning",
    torch_dtype=torch.float16,
    variant="fp16",
).to("cuda")
pipe.scheduler = DPMSolverMultistepScheduler.from_config(pipe.scheduler.
    config)
```

```

# [...]

mask = pipe.mask_processor.blur(Image.fromarray(mask), blur_factor=75)

result = pipe(
    prompt=data["prompt"],
    negative_prompt=data["negative_prompt"],
    image=image,
    mask_image=mask,
    strength=0.8,
    generator=generator,
    num_inference_steps=30,
    guidance_scale=2,
).images[0]
```

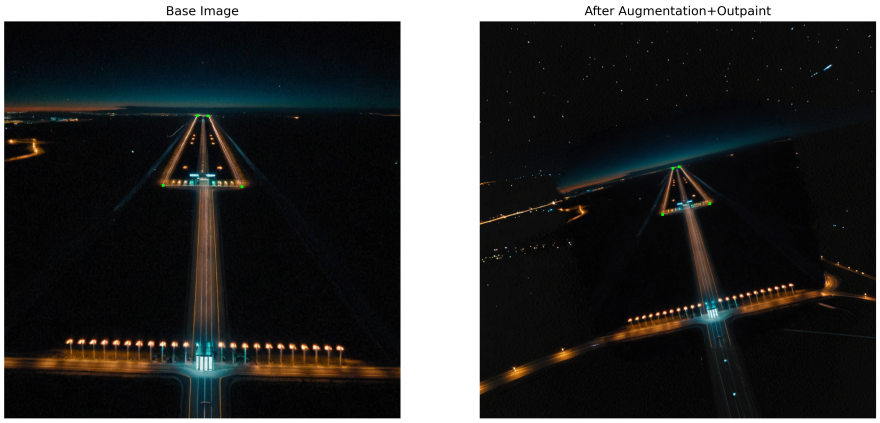

Figure 4.2: Applying the outpainting pipeline to an image.

The third pipeline reads the images output by the second pipeline and creates a new folder with the same images, further augmented with weather occlusion effects done by `imgaug` [22]. Effects are chosen depending on the variant:

- Clouds for *pure* `day/night/dusk/dawn` images
- Fog for `fog` images
- Rain for `rain` images
- Snowflakes for `snow` images

For example:

Listing 4.12: Variant image generation module, `imgaug` pipeline

```
if variant in ["day", "night", "dusk", "dawn"]:
    aug = iaa.SomeOf((1, 2), [
        iaa.CloudLayer(...),
        iaa.CloudLayer(...),
    ])
```

```
    image = aug(image=image)
    applied_effects.append("clouds")
elif variant in ["fog", "fog+night"]:
    aug = iaa.CloudLayer(...)
    image = aug(image=image)
    applied_effects.append("fog")
elif variant in ["rain", "rain+night"]:
    aug = iaa.Rain(drop_size=(0.1, 0.2), speed=(0.01, 0.05))
    image = aug(image=image)
    applied_effects.append("light_rain")
elif variant in ["snow", "snow+night"]:
    aug = iaa.Snowflakes(flake_size=(0.1, 0.3), speed=(0.01, 0.05))
    image = aug(image=image)
    applied_effects.append("snowflakes")
```

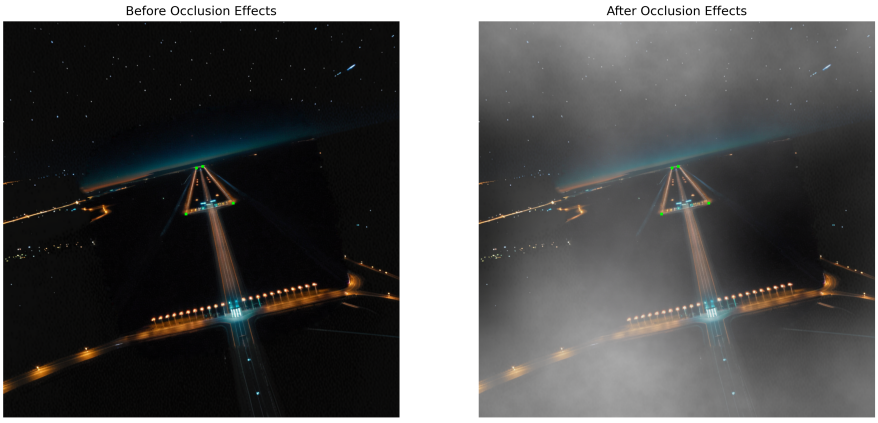

Figure 4.3: Applying the occlusion pipeline to an image.

After each augmentation pipeline, the label JSON file is enriched with data to ensure reproducibility, including random seeds and applied transformations.

4.2 FILTERING TOOL

A manual filtering tool was implemented in Python with OpenCV to assist in selecting template images. It reads images from a directory and allows pressing `Space` to select an image or `X` to discard. Selected images are copied into a new directory, and a log file retains progress so the tool can be closed and resumed later without starting from scratch.

After a first test running the Canny2Concrete pipeline, it was clear that not all canny edges generated from the nominal dataset were suitable for generating realistic runway images. Thus, to filter which images would be used as template images, the filtering tool was used to select only the images that effectively generated a realistic runway image.

Template images were filtered as follows:

1. Use all images from `LARD_test_real_nominal` as template images, pass them through the canny-edge and base-image generation modules, generating one daylight base image per template.
2. Use the filtering tool to select which base images had easily recognizable runways with consistent markings and structure.
3. Then, only use these selected images as template images to generate the datasets.

4.3 DATASETS GENERATION

Three datasets are generated with the Canny2Concrete pipeline:

1. `BaseImages`: 18 variations for each of the canny edge images.
2. `VariantImages`: 3 variations for each base image *before* weather occlusion.
3. `VariantImagesWithOcclusion`: same as `VariantImages` but with weather effects applied.

Each image is associated with four files:

1. `.png` file (the image itself).
2. `.json` file (the image label).
3. `.mask.png` file (a segmentation mask: black background, white runway).
4. `.txt` file (a YOLO-format label for detection/segmentation training).

The JSON label file carries the runway keypoints as well as all data required for reproducing that specific image, including prompts, seeds, variant details, transformations, and so on. This metadata supports image classification (by variant or weather type), label fixes without regenerating images, and easier reproducibility/peer review.

Each dataset also includes `train.txt`, `test.txt`, and `val.txt` splits in 80/10/10 proportions, ensuring images sharing the same source canny edge (and thus similar structure) do not leak from training into testing.

These three datasets are the parts of *GARD: Gustavo's Awesome Runway Dataset*.

4.4 EVALUATION

The evaluation is done in two parts: intrinsic and extrinsic, through five Jupyter notebooks. By convention, the file names are prefixed with `r_` to indicate they are part of the results' evaluation.

4.4.1 *Intrinsic evaluation*

For the intrinsic evaluation, Structural Similarity Index Measure (SSIM) is used to compare the generated images with their original templates. To compute the SSIM, the `pytorch-msssim` [29] was used instead of the more commonly used `scikit-image` [37] implementation, as the former runs on the GPU and is faster.

This function groups SSIM scores by variant type (e.g., `day`, `fog+night`) for later statistical analysis and comparison:

Listing 4.13: Evaluation, SSIM score calculation

```
def calculate_ssim_values(folder):
    image_files = [f for f in os.listdir(folder) if f.lower().endswith(".
        png") and not f.lower().endswith(".mask.png")]

    results = {}

    for img_name in tqdm(image_files, desc="Processing images", unit="img
        "):
        ...

        if not results.get(variant):
            results[variant] = []

        template = cv2.imread(f'p_Templates/{sourceImage}')
        template.resize((1024,1024,3))

        image_tensor = torch.from_numpy(image)
                            .float().permute(2, 0, 1)
                            .unsqueeze(0).to(device) / 255.0
        template_tensor = torch.from_numpy(template)
                               .float().permute(2, 0, 1)
                               .unsqueeze(0).to(device) / 255.0

        with torch.no_grad():
            score = ssim(
                template_tensor,
                image_tensor,
                data_range=1.0,
                size_average=True
            ).item()

        results[variant].append(score)

    return results
```

4.4.2 *Experimental evaluation*

Experimental evaluation was done by training and fine-tuning YOLOv11 models on the generated datasets and on the LARD dataset, serving as a baseline. The YOLO architecture was chosen because it is commonly used for object detection and segmentation tasks, was widely used in

the literature review, and has strong community support. YOLOv11 was chosen because it is the latest and best-performing YOLO model.

4.4.2.1 *Generating Labels and Masks for GARD*

Before training the YOLO model, it is necessary to create labels in the YOLO format. To make it also easier to train other segmentation models, a binary mask is also generated for each image. Both the YOLO TXT labels and the masks are published alongside the images.

Listing 4.14: Evaluation, generating labels and masks

```
def polygon_to_yolo_label_from_image_dict(image: Image.Image, data: dict,
     class_id=0) -> str:
    """
    Converts the 'runwayLabel' polygon from the data dict into YOLO
        segmentation format:
    <class-index> <x1> <y1> <x2> <y2> ... <xn> <yn>

    Coordinates are normalized to [0, 1] based on the image dimensions.
    """
    if "runwayLabel" not in data:
        raise KeyError("'runwayLabel' key not found in data dict.")

    polygon = ensure_clockwise_quad(data["runwayLabel"])
    width, height = image.size

    # Normalize coordinates to 0-1 range
    normalized_points = [(pt[0] / width, pt[1] / height) for pt in
        polygon]

    flat_coords = " ".join(f"{x:.6f} {y:.6f}" for x, y in
        normalized_points)
    return f"{class_id} {flat_coords}"


def create_runway_segmentation_mask(image: Image.Image, data: dict) ->
    Image.Image:
    if "runwayLabel" not in data:
        raise KeyError("'runwayLabel' key not found in data dict.")

    polygon = ensure_clockwise_quad(data["runwayLabel"])
    width, height = image.size
    mask = Image.new("L", (width, height), 0)
    draw = ImageDraw.Draw(mask)
    draw.polygon([(int(x), int(y)) for (x, y) in polygon], fill=255)

    return mask
```

4.4.2.2 *Assembling YOLO datasets*

YOLO requires a very specific dataset folder structure to fine-tune and validate the model. To convert the GARD and LARD datasets into

these formats, symlinks were used to avoid duplicating the images and labels.

First, splits are created for the GARD datasets:

Listing 4.15: Evaluation, GARD train/test/val splits

```
def create_yolo_dataset(folder_path: str):
    """
    Creates train.txt, test.txt, val.txt, and dataset.yaml inside ‘folder_path‘
    based on PNG images in ‘folder_path‘ (excluding *.mask.png files).

    Splits data by original_id (to avoid data leakage) into
    80% train, 10% test, 10% val.

    NOTE: The paths inside train.txt/test.txt/val.txt will be relative
    to ‘folder_path‘.
    """

    n = len(original_ids)
    train_end = int(0.80 * n)
    test_end  = int(0.90 * n)

    train_ids = original_ids[:train_end]
    test_ids  = original_ids[train_end:test_end]
    val_ids   = original_ids[test_end:]

    ...
```

As the comments indicate, the splits are done by the `sourceImage` field, which references the original template image. This ensures that images generated from the same template are not split across train/test/val, avoiding data leakage.

Then, the following code assembles the dataset:

Listing 4.16: Evaluation, assembling GARD dataset in YOLO structure

```
def process_folder_structure_with_labels(folder_path: str):
    """
    Reads train.txt, test.txt, val.txt from ‘folder_path‘ (which was created by create_yolo_dataset).
    Builds a YOLO directory structure in ‘datasets/<folder_name>‘, placing:

      datasets/<folder_name>/
        images/train/ -> symlinks to the train images
        images/test/  -> symlinks to the test images
        images/val/   -> symlinks to the val images
        labels/train/ -> symlinks to the corresponding train labels
        labels/test/  -> symlinks to the corresponding test labels
        labels/val/   -> symlinks to the corresponding val labels
        dataset.yaml  -> references these subfolders

    :param folder_path: The path to your original folder (e.g. "p_BaseImages"),
```

```
                    which must have train.txt, test.txt, val.txt,
                        dataset.yaml
                    already generated by create_yolo_dataset().
    """

    src_dir = Path(folder_path).resolve()
    folder_name = src_dir.name  # e.g. "p_BaseImages"

    # The new dataset directory, e.g. datasets/p_BaseImages
    dst_base = Path("datasets") / folder_name
    dst_base.mkdir(parents=True, exist_ok=True)

    # keep the standard YOLO layout:
    # images/train, images/test, images/val, plus labels/train, labels/
        test, labels/val
    images_dir = dst_base / "images"
    labels_dir = dst_base / "labels"
    train_img_dir = images_dir / "train"
    test_img_dir  = images_dir / "test"
    val_img_dir   = images_dir / "val"
    train_lbl_dir = labels_dir / "train"
    test_lbl_dir  = labels_dir / "test"
    val_lbl_dir   = labels_dir / "val"

    train_img_dir.mkdir(parents=True, exist_ok=True)
    test_img_dir.mkdir(parents=True, exist_ok=True)
    val_img_dir.mkdir(parents=True, exist_ok=True)
    train_lbl_dir.mkdir(parents=True, exist_ok=True)
    test_lbl_dir.mkdir(parents=True, exist_ok=True)
    val_lbl_dir.mkdir(parents=True, exist_ok=True)

    train_file = src_dir / "train.txt"
    test_file  = src_dir / "test.txt"
    val_file   = src_dir / "val.txt"

    ...
    # Symlink the train, test, val sets
    symlink_split(train_file, train_img_dir, train_lbl_dir)
    symlink_split(test_file,  test_img_dir,  test_lbl_dir)
    symlink_split(val_file,   val_img_dir,   val_lbl_dir)

    ...
    dst_dataset_yaml = dst_base / "dataset.yaml"
    with dst_dataset_yaml.open("w") as f:
        f.write(dataset_yaml_content)
```

Similar functions were used to assemble the LARD dataset in YOLO format, and also the LARD test real datasets, which were used for validation.

4.4.2.3 *Fine-tuning YOLOv11*

Once the dataset is assembled in YOLO's format, it is very easy to fine-tune a pre-trained model:

Listing 4.17: Evaluation, fine-tuning YOLOv11

```python
from ultralytics import YOLO

model = YOLO("yolo11n-seg.pt")
results = model.train(
    data="./datasets/p_BaseImages/dataset.yaml",
    epochs=100,
    patience=10,
    project="trained_models/yolo",
    name="gard-BaseImages",
)
```

Epochs and patience are hyperparameters that can be adjusted, with epochs indicating the maximum number of training epochs and patience indicating the number of epochs to wait before early stopping if the validation loss does not improve. YOLO saves the best and latest weights. These weights are also published on the project's Kaggle repository.

4.4.2.4 *Validation*

Using the LARD test real datasets, the trained models are validated using YOLO's own validation function:

Listing 4.18: Evaluation, YOLOv11 validation

```python
import os
from ultralytics import YOLO

def evaluate_all_models():
    models_base_path = "trained_models/yolo"
    properties = {
        "save_json": True,
        "plots": True,
        "imgsz": 640,
        "batch": 16,
        "conf": 0.50,
        "iou": 0.6,
        "device": "0",
        "save_txt": True,
        "save_conf": True,
        "save_crop": True
    }

    results_dict = {}

    for model_dir in os.listdir(models_base_path):
        ...

        model = YOLO(best_weights_path)

        nominal_results = model.val(
            data="./datasets/LARD-real-nominal/dataset.yaml",
            **properties,
        )
        edge_cases_results = model.val(
```

```
            data="./datasets/LARD-real-edge-cases/dataset.yaml",
            **properties,
        )

        results_dict[model_dir] = {
            "nominal": nominal_results,
            "edge_cases": edge_cases_results
        }

    return results_dict
```

The results are both saved to folders that are published on the project's Kaggle repository and returned as a dictionary for easier analysis in the Jupyter notebook itself.

Part III

RESULTS AND DISCUSSION

5 EVALUATION

5.1 GENERATED DATASETS

After filtering the template images manually using the Filtering Tool and the methodology described in Chapter 4, a total of 361 template images were selected.

Using the Canny2Concrete pipeline with these filtered template images, *GARD* was generated, containing a total of 45486 images, separated into three datasets:

1. `BaseImages`: containing 6498 images (18 variations for each of the 361 canny edge images).
2. `VariantImages`: containing 19494 images (3 variations for each base image) *before* weather occlusion.
3. `VariantImagesWithOcclusion`: also 19494 images, same as `VariantImages` but with heavy weather occlusion effects applied.

Table 5.1 shows the distribution of data diversity, presenting the number of images per weather and lighting condition in each dataset.

Table 5.1: Distribution of images per condition in each dataset.

Condition	**Base Images**	**Variant Images**	**Variants w/ Occlusion**
Daylight, no occlusion	1,805	5,415	5,415
Nighttime, no occlusion	1,805	5,415	5,415
Dusk, no occlusion	361	1,083	1,083
Dawn, no occlusion	361	1,083	1,083
Daylight, Fog	361	1,083	1,083
Nighttime, Fog	361	1,083	1,083
Daylight, Rain	361	1,083	1,083
Nighttime, Rain	361	1,083	1,083
Daylight, Snow	361	1,083	1,083
Nighttime, Snow	361	1,083	1,083
Total	6,498	19,494	19,494

Finally, a comparison between GARD and other synthetic datasets is shown in Figure 5.1, such as BARS [11], RLD [38], FS2020 [10], and LARD [16].

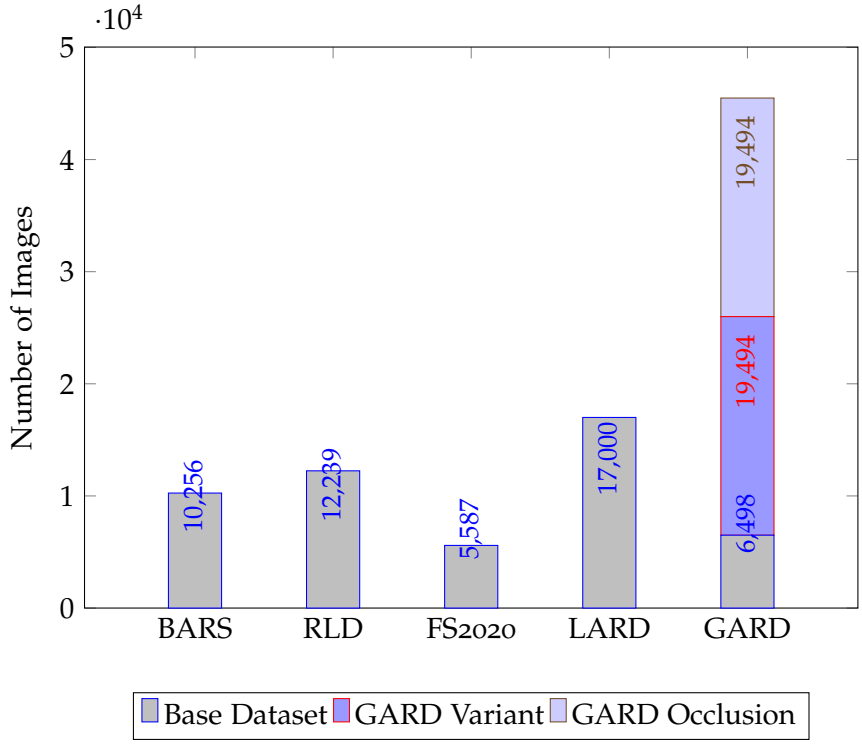

Figure 5.1: Number of images per dataset used in runway segmentation research.

5.2 SAMPLE IMAGES

To demonstrate a sample of generated images, three random template images were chosen. Two are close-up shots and the third is a far-away picture. They are shown as three grids of images of four rows.

The first row contains the template image, the canny edge extracted from it, and the runway label mask. The second row contains three images from the Base Images dataset generated from the template image. The third row contains three images from the Variant Images dataset, generated from the Base Images. The fourth row contains three images from the Variant Images with Occlusion dataset, associated with the shown Variant Images.

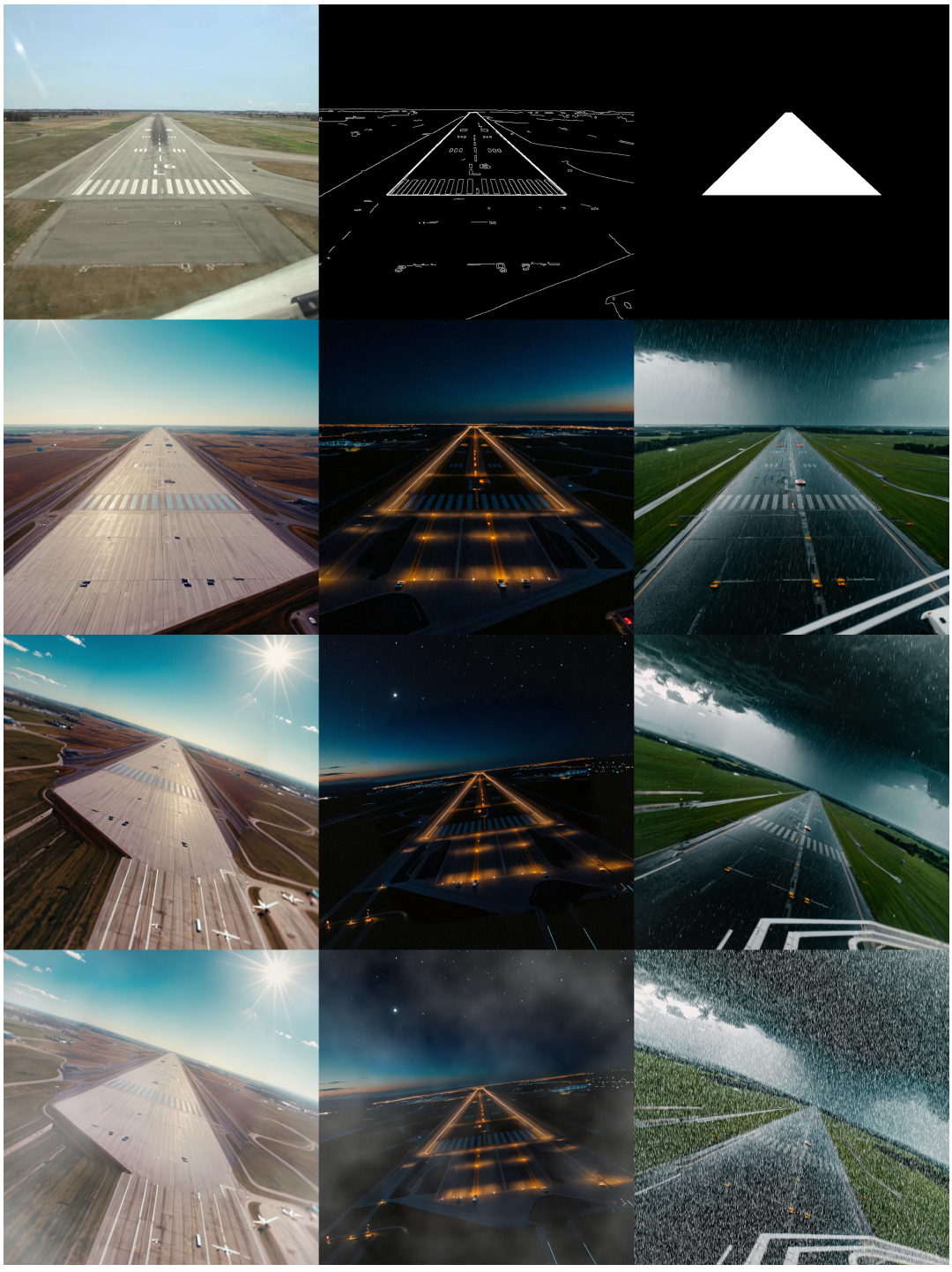

Figure 5.2: Images generated from the template image *"9xGoa8TUdXg_028"*.

Figure 5.3: Images generated from the template image *"oCLSYZPFbTg_110"*.

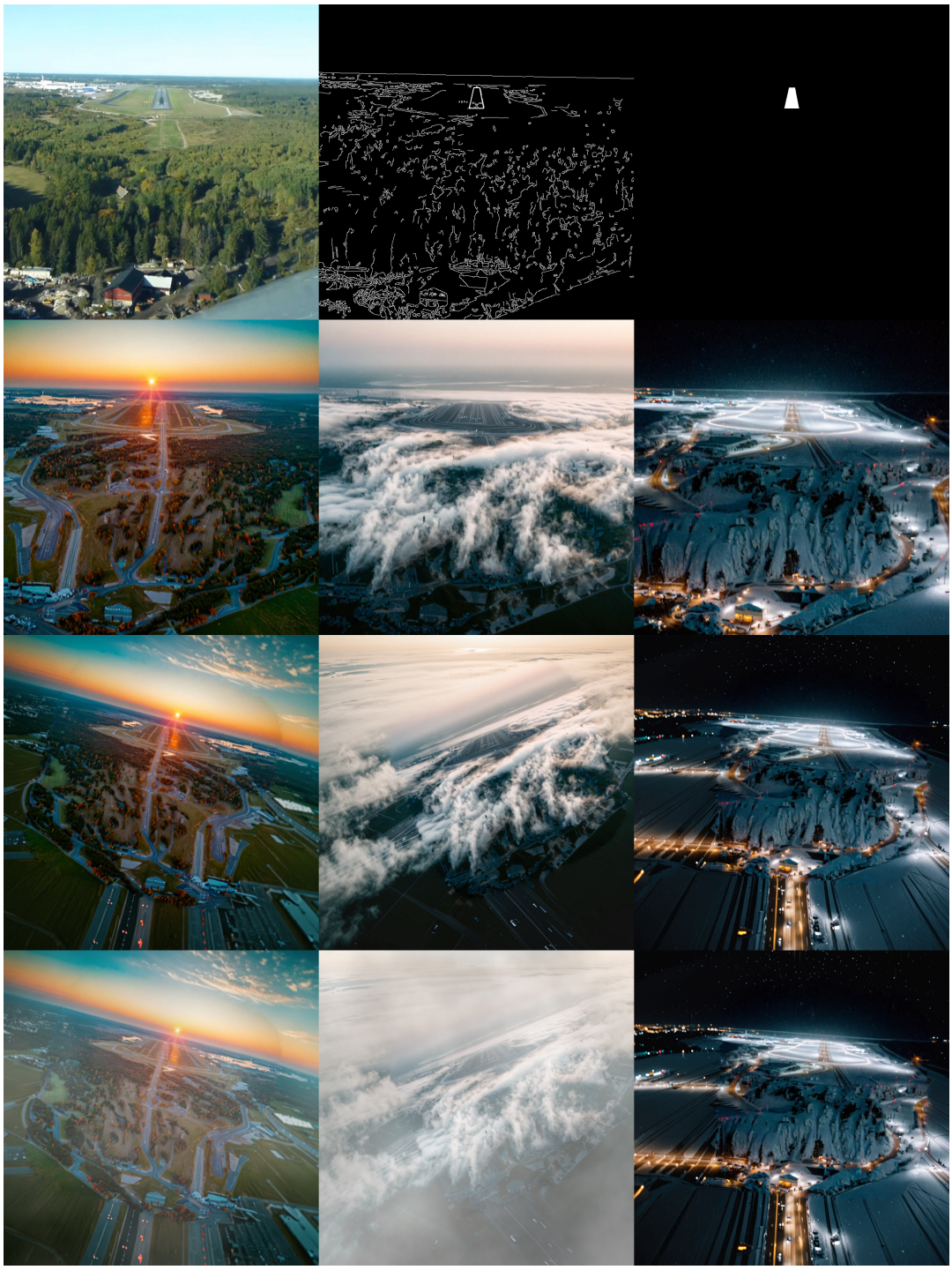

Figure 5.4: Images generated from the template image *"oP-HJgLkZLk_085"*.

5.3 INTRINSIC EVALUATION

The chosen metric for intrinsic evaluation was SSIM (Structural Similarity Index Measure) [39]. SSIM evaluates how similar two images are to each other. An SSIM of 1 indicates perfect similarity (i.e., the image compared with itself), a score of 0 indicates no similarity, and a score of -1 represents perfect anti-correlation.

It is difficult to define what constitutes a good SSIM value. For example, a realistic and high-quality background that significantly differs from the original image would reduce the SSIM score, despite not being inherently negative. On the other hand, since the generated image is based on the canny edge structure of the template image, we should expect a positive correlation if the generation process is functioning as intended.

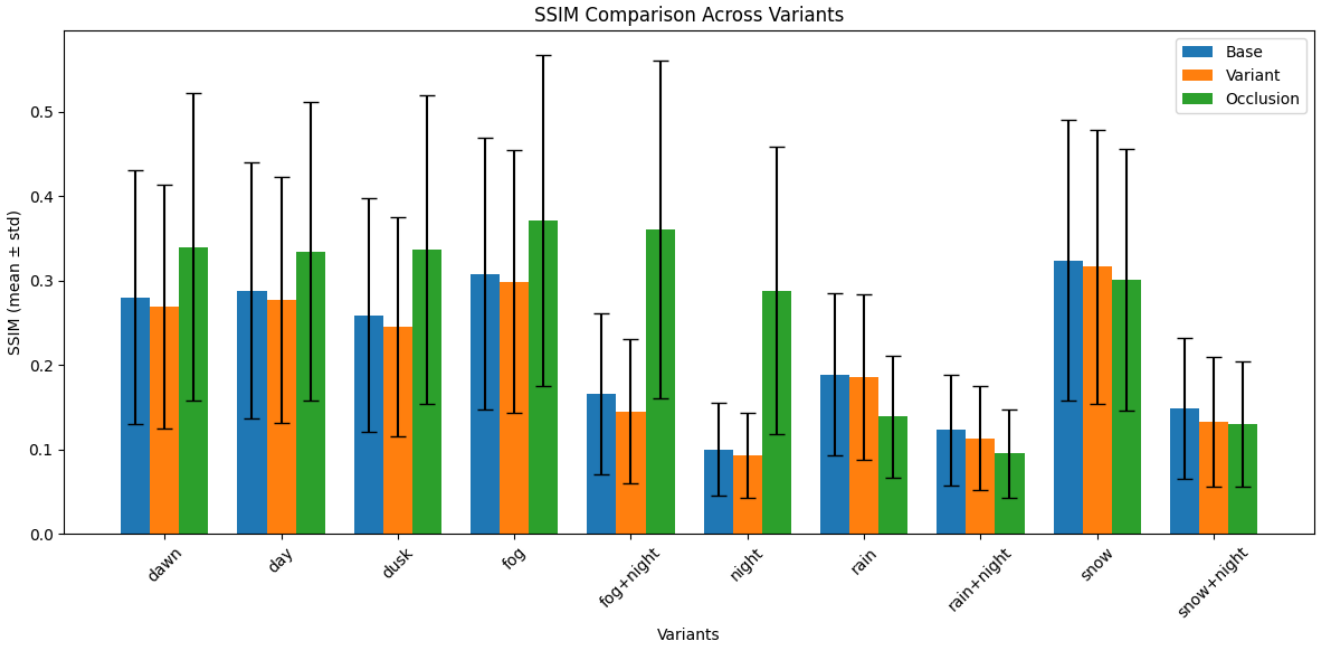

Figure 5.5: SSIM Comparison Across Variants and Datasets

These are positive results, clearly indicating that the data augmentation process is functioning as expected. The generated images are positively correlated with the template images—as expected in a data augmentation pipeline—while the similarity significantly drops as diversity is introduced.

5.4 EXPERIMENTAL EVALUATION

Following the methodology of [36], for extrinsic evaluation, several pre-trained segmentation models are fine-tuned, comparing their per-

formance when trained exclusively on the synthetic images of LARD [16] and when trained on the project's datasets.

To validate the performance of the trained models, the human-labeled real images of the LARD dataset are used. The real images in LARD are divided into two folders: *nominal cases* and *edge cases*, the latter containing images with poor runway visibility.

Three pre-trained models from the YOLO v11 family are fine-tuned: YOLO11n ("n" for nano), YOLO11s ("s" for small), and YOLO11m ("m" for medium). The larger models YOLO11l and YOLO11x were not trained due to hardware constraints.

YOLO reports metrics for two tasks: detection (bounding box evaluation) and segmentation (segmentation mask evaluation). The metric used to compare the performance is mAP (mean Average Precision), which is the average of the precision-recall curve. The mAP is calculated at three different IoU (Intersection over Union) thresholds: 50%, 75%, and 50:95.

Two tables (Tables 5.2 and 5.3) are presented, one for the nominal dataset and another for the edge cases dataset. The tables show the performance of the models when trained on the LARD dataset and the GARD datasets, with the differences in performance highlighted, with a positive value indicating an improvement in performance of the model trained on the GARD dataset over the comparable one with the LARD dataset.

Based on the experimental evaluation metrics, the results are overwhelmingly positive, as the GARD dataset has consistently matched or outperformed the LARD dataset in most cases. Especially in the Edge Cases dataset, which is the most challenging, the GARD dataset shows outperformance across all models, tasks (detection and segmentation), and mAP thresholds.

Table 5.2: Validation with real images from LARD dataset, *Nominal* dataset: YOLO11n/s/m performance comparison between LARD and GARD variants, with per-variant differences (Δ).

Model	**Dataset**	**Detection (mAP)**			**Segmentation (mAP)**		
		@50:95	@50	@75	@50:95	@50	@75
YOLO11n	Baseline (LARD)	0.691	0.862	0.790	0.485	0.833	0.507
	GARD BaseImages	0.722	0.874	0.796	0.452	0.828	0.441
	Δ	**+0.031**	**+0.012**	**+0.006**	**-0.033**	**-0.005**	**-0.066**
	GARD VariantImages	0.735	0.888	0.807	0.471	0.842	0.472
	Δ	**+0.044**	**+0.026**	**+0.017**	**-0.014**	**+0.009**	**-0.035**
	GARD VariantWithOcclusion	0.728	0.899	0.794	0.469	0.841	0.467
	Δ	**+0.037**	**+0.037**	**+0.004**	**-0.016**	**+0.008**	**-0.040**
YOLO11s	Baseline (LARD)	0.709	0.870	0.807	0.498	0.842	0.526
	GARD BaseImages	0.747	0.894	0.819	0.454	0.843	0.432
	Δ	**+0.038**	**+0.024**	**+0.012**	**-0.044**	**+0.001**	**-0.094**
	GARD VariantImages	0.780	0.915	0.846	0.485	0.873	0.482
	Δ	**+0.071**	**+0.045**	**+0.039**	**-0.013**	**+0.031**	**-0.044**
	GARD VariantWithOcclusion	0.771	0.914	0.845	0.481	0.870	0.476
	Δ	**+0.062**	**+0.044**	**+0.038**	**-0.017**	**+0.028**	**-0.050**
YOLO11m	Baseline (LARD)	0.742	0.906	0.841	0.512	0.870	0.529
	GARD BaseImages	0.747	0.889	0.825	0.452	0.836	0.434
	Δ	**+0.005**	**-0.017**	**-0.016**	**-0.060**	**-0.034**	**-0.095**
	GARD VariantImages	0.792	0.923	0.854	0.489	0.874	0.490
	Δ	**+0.050**	**+0.017**	**+0.013**	**-0.023**	**+0.004**	**-0.039**
	GARD VariantWithOcclusion	0.789	0.922	0.860	0.485	0.878	0.481
	Δ	**+0.047**	**+0.016**	**+0.019**	**-0.027**	**+0.008**	**-0.048**

Table 5.3: Validation with real images from LARD dataset, *Edge cases* dataset: YOLO11n/s/m performance comparison between LARD and GARD variants, with per-variant differences (Δ).

Model	**Dataset**	**Detection (mAP)**			**Segmentation (mAP)**		
		@50:95	@50	@75	@50:95	@50	@75
YOLO11n	Baseline (LARD)	0.543	0.708	0.634	0.399	0.700	0.414
	GARD BaseImages	0.608	0.802	0.663	0.425	0.762	0.439
	Δ	**+0.065**	**+0.094**	**+0.029**	**+0.026**	**+0.062**	**+0.025**
	GARD VariantImages	0.627	0.819	0.702	0.452	0.786	0.476
	Δ	**+0.084**	**+0.111**	**+0.068**	**+0.053**	**+0.086**	**+0.062**
	GARD VariantWithOcclusion	0.633	0.850	0.702	0.456	0.805	0.465
	Δ	**+0.090**	**+0.142**	**+0.068**	**+0.057**	**+0.105**	**+0.051**
YOLO11s	Baseline (LARD)	0.530	0.696	0.620	0.395	0.680	0.388
	GARD BaseImages	0.638	0.834	0.689	0.440	0.795	0.446
	Δ	**+0.108**	**+0.138**	**+0.069**	**+0.045**	**+0.115**	**+0.058**
	GARD VariantImages	0.686	0.859	0.772	0.476	0.827	0.490
	Δ	**+0.156**	**+0.163**	**+0.152**	**+0.081**	**+0.147**	**+0.102**
	GARD VariantWithOcclusion	0.681	0.883	0.735	0.470	0.830	0.478
	Δ	**+0.151**	**+0.187**	**+0.115**	**+0.075**	**+0.150**	**+0.090**
YOLO11m	Baseline (LARD)	0.551	0.730	0.653	0.406	0.719	0.396
	GARD BaseImages	0.641	0.826	0.702	0.433	0.791	0.429
	Δ	**+0.090**	**+0.096**	**+0.049**	**+0.027**	**+0.072**	**+0.033**
	GARD VariantImages	0.684	0.863	0.747	0.477	0.813	0.494
	Δ	**+0.133**	**+0.133**	**+0.094**	**+0.071**	**+0.094**	**+0.098**
	GARD VariantWithOcclusion	0.700	0.887	0.770	0.484	0.842	0.492
	Δ	**+0.149**	**+0.157**	**+0.117**	**+0.078**	**+0.123**	**+0.096**

5.5 DISCUSSION

One concern with validating trained models with the Nominal dataset is that images from this same dataset were used as template images for the data augmentation pipeline. This might have introduced a bias in the evaluation, as the generated images are based on the same structure as the template images. But we can be confident that either there is no bias or that the bias is minimal, as the SSIM results show positive but small correlation scores between the template images and the generated images. Another reason why this should not be a concern is that the fine-tuned models performed better in the Edge Cases dataset, of which no images were used in the data augmentation pipeline.

Another limitation of the experimental evaluation setup was that the models were fine-tuned on each isolated GARD dataset, without mixing them. This might have limited the performance of the models, as each dataset has distinct features that could have increased the diversity of a mixed dataset.

The better performance of the LARD-based models in the Nominal dataset might be explained by the fact that the LARD dataset contains low diversity of weather and lighting conditions, and runway occlusion, which matches the characteristics of the Nominal dataset. This highlights the *joint dataset-model problem* that was presented in the literature review section.

This is also corroborated by the better performance of the GARD-based models in the Edge Cases dataset, where the better diversity and realism of the GARD datasets allowed the models to generalize better to the unseen data.

Because of time and hardware constraints, the evaluation was limited to the YOLO11 family of models. Future work could include more models and more complex pipelines dedicated to runway segmentation, such as VALNet [38].

These results are promising, showing that the Canny2Concrete pipeline is effective in generating realistic runway images that can be used to train detection and segmentation models, and that the GARD dataset is a viable alternative to existing synthetic datasets.

The biggest limitation in the Canny2Concrete pipeline is the need for template images. This limits the variety of structures that can be generated and requires a manual selection of images. Future work could be done in either programmatically generating canny edge images without extracting them from any given image. It would also help if the diffusion model understood better the concept of an aerial view of a runway, so that it wouldn't need detailed canny edges to generate a realistic image.

Another limitation is that, due to the chosen Stable Diffusion model, the images don't look like real photos, but more like digital art. Future

work could shrink the "Sim2Real" gap. This could be done by using a different diffusion model, or by adding another module to the pipeline that would convert images to this more realistic style.

The pipeline could also generate better results by improving the outpainting techniques used in the variant image generation module. The current techniques generate noticeable borders between the original image and the outpainted background.

Part IV

CONCLUSION

6 CONCLUSION

To address the challenges of data scarcity in the field of runway detection and segmentation, this paper introduced a novel, open-source data augmentation technique based on a multi-step Stable Diffusion pipeline, called *Canny2Concrete*.

Canny2Concrete extracts features from existing datasets and outputs images that retain a similar structure, especially the runway shape, position, and markings, but are customizable with respect to scenery, weather, and lighting conditions, guided by text prompts. The images generated by the pipeline are already labeled, saving hours of manual labeling.

Canny2Concrete is a modular and easily extensible pipeline, composed of four independent modules: Template Image Selection, Edge Extraction, Base Image Generation, and Variant Image Generation. This design facilitates rapid development and allows easy adaptation to other domains beyond runway imagery.

The pipeline is tested by augmenting real runway images from the LARD [16] dataset, generating GARD. Three datasets are generated: the Base Images dataset, containing 6498 images; the Variant Images dataset, containing 19494 images; and the Variant Images With Occlusion dataset, containing 19494 images. In total, 45486 images, with a diverse range of weather, lighting, background, and runway occlusion conditions and effects were generated and are publicly available on Kaggle. To the best of my knowledge, *GARD* is the largest synthetic runway dataset publicly available.

All the images have a JSON file with metadata allowing any researcher to replicate that exact image, a TXT label ready to be used in training YOLO models, and a simple binary segmentation mask image.

Experimental evaluation was done by training state-of-the-art detection and segmentation models with both the new datasets and a benchmark dataset, and these models were evaluated on a real-world dataset. The results demonstrate the effectiveness of the *Canny2Concrete* pipeline in generating realistic runway images, and the viability of diffusion-based augmentation for runway segmentation tasks.

In summary, this work directly addresses the research question posed at the outset—how to build a suitable synthetic image dataset without using simulators. By leveraging a novel and modular image generation pipeline using edge detection, latent diffusion models, and image augmentation techniques, this work demonstrates a practical and reproducible method for generating high-quality, labeled runway

images at scale. GARD, the resulting dataset, not only bypasses the need for flight simulators but also achieves competitive quality, realism, and utility for training object detection and segmentation models, as evidenced by performance benchmarks.

Part V

APPENDIX

APPENDIX

A.1 HARDWARE AND SOFTWARE SPECIFICATIONS

All the experiments were conducted on a computer with the following specifications:

I use arch btw.

- **CPU:** AMD Ryzen 9 7900X (24) @ 5.733GHz
- **GPU:** NVIDIA GeForce RTX 4070 Ti, 12GB
- **Memory:** 64 GB
- **Operating System:** Arch Linux x86_64
- **Linux Kernel:** 6.13.6-arch1-1

A.2 EXPERIMENTS DURATION

To generate the base images, about 10 hours were needed. To generate the variant images, about 40 hours were needed. To generate the variant images with occlusion, about 2 hours were needed.

A.3 JSON LABEL PROPERTIES

In Table A.1, we summarize the most important top-level fields in the JSON label file describing runway image metadata, annotations, generation, and transformation history.

Property	Description
`dataset`	Name of the dataset the image belongs to (e.g., `LARD`). Useful for classification or filtering.
`sourceImage`	Filename of the original template image.
`runwayLabel`	List of 4 `[x, y]` coordinates representing the annotated runway corners (pre-transform).
`variant`	Visual variant of the scene (e.g., `dawn`, `night`, etc.). Useful for filtering by scene conditions.
`prompt`	Positive text prompt used for image generation (e.g., in Stable Diffusion). Critical for reproducibility.
`negative_prompt`	Negative prompt describing what the model should avoid during image generation.
`seed`	Random seed used for initial image generation — ensures image reproducibility.
`model`	Name of the diffusion model used (e.g., `sdxl-dreamshaperxl`).
`baseImage`	Identifier for the untransformed version of the generated image, before augmentations.
`albumentationsReplay`	Complete record of image transformations (flips, crops, affine) used for augmentation — including parameters and outcomes. Enables full replay of augmentations.
`outpaintingSeed`	Seed used during outpainting generation (if enabled).
`occlusionEffects`	Details of occlusion layers applied (e.g., clouds), including seed used for randomness.

Table A.1: Top-level fields in the JSON label file describing runway image metadata, annotations, generation, and transformation history.

BIBLIOGRAPHY

[1] Airbus. *Accidents by Flight Phase – accidentstats.airbus.com*. en-US. URL: https://accidentstats.airbus.com/accidents-by-flight-phase/ (visited on 02/27/2025).

[2] Airbus. *Fatal Accidents – accidentstats.airbus.com*. en-US. URL: https://accidentstats.airbus.com/fatal-accidents/ (visited on 02/27/2025).

[3] Airbus. *Airbus concludes ATTOL with fully autonomous flight tests | Airbus*. en. Section: Innovation. Oct. 2021. URL: https://www.airbus.com/en/newsroom/press-releases/2020-06-airbus-concludes-attol-with-fully-autonomous-flight-tests (visited on 03/03/2025).

[4] Javeria Akbar, Muhammad Shahzad, Muhammad Imran Malik, Adnan Ul-Hasan, and Fasial Shafait. "Runway Detection and Localization in Aerial Images using Deep Learning." In: *2019 Digital Image Computing: Techniques and Applications (DICTA)*. Dec. 2019, pp. 1–8. DOI: 10.1109/DICTA47822.2019.8945889. URL: https://ieeexplore.ieee.org/document/8945889 (visited on 12/14/2024).

[5] Vladimir Arkhipkin, Andrei Filatov, Viacheslav Vasilev, Anastasia Maltseva, Said Azizov, Igor Pavlov, Julia Agafonova, Andrey Kuznetsov, and Denis Dimitrov. *Kandinsky 3.0 Technical Report*. arXiv:2312.03511 [cs] version: 3. June 2024. DOI: 10.48550/arXiv.2312.03511. URL: http://arxiv.org/abs/2312.03511 (visited on 03/03/2025).

[6] Ö. Aytekin, U. Zöngür, and U. Halici. "Texture-Based Airport Runway Detection." In: *IEEE Geoscience and Remote Sensing Letters* 10.3 (May 2013). Conference Name: IEEE Geoscience and Remote Sensing Letters, pp. 471–475. ISSN: 1558-0571. DOI: 10.1109/LGRS.2012.2210189. URL: https://ieeexplore.ieee.org/abstract/document/6269052 (visited on 03/03/2025).

[7] James Betker et al. "Improving Image Generation with Better Captions." en. In: (). URL: https://cdn.openai.com/papers/dall-e-3.pdf.

[8] Boeing. *Statistical Summary of Commercial Jet Airplane Accidents*. Aug. 2024. URL: https://www.boeing.com/content/dam/boeing/boeingdotcom/company/about_bca/pdf/statsum.pdf (visited on 03/03/2025).

[9] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. "Albumentations: Fast and Flexible Image Augmentations." en. In: *Information* 11.2 (Feb. 2020). Number: 2 Publisher: Multidisciplinary Digital Publishing Institute, p. 125. ISSN: 2078-2489. DOI: 10.3390/info11020125. URL: https://www.mdpi.com/2078-2489/11/2/125 (visited on 03/31/2025).

[10] Mingqiang Chen and Yuzhou Hu. "An image-based runway detection method for fixed-wing aircraft based on deep neural network." en. In: *IET Image Processing* 18.8 (2024). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1049/ipr2.13087, pp. 1939–1949. ISSN: 1751-9667. DOI: 10.1049/ipr2.13087. URL: https://onlinelibrary.wiley.com/doi/abs/10.1049/ipr2.13087 (visited on 03/03/2025).

[11] Wenhui Chen, Zhijiang Zhang, Liang Yu, and Yichun Tai. *BARS: A Benchmark for Airport Runway Segmentation*. arXiv:2210.12922 [cs]. Apr. 2023. DOI: 10.48550/arXiv.2210.12922. URL: http://arxiv.org/abs/2210.12922 (visited on 12/14/2024).

[12] Gong Cheng, Junwei Han, and Xiaoqiang Lu. "Remote Sensing Image Scene Classification: Benchmark and State of the Art." In: *Proceedings of the IEEE* 105.10 (Oct. 2017). Conference Name: Proceedings of the IEEE, pp. 1865–1883. ISSN: 1558-2256. DOI: 10.1109/JPROC.2017.2675998. URL: https://ieeexplore.ieee.org/document/7891544 (visited on 03/04/2025).

[13] Gilad Cohen and Raja Giryes. *Generative Adversarial Networks*. arXiv:2203.00667 [cs]. Mar. 2022. DOI: 10.48550/arXiv.2203.00667. URL: http://arxiv.org/abs/2203.00667 (visited on 03/05/2025).

[14] *Diffusers*. URL: https://huggingface.co/docs/diffusers/en/index (visited on 03/31/2025).

[15] *DreamShaper XL - v2.1 Turbo DPM++ SDE | Stable Diffusion XL Checkpoint | Civitai*. en. Jan. 2025. URL: https://civitai.com/models/112902/dreamshaper-xl (visited on 03/31/2025).

[16] Mélanie Ducoffe, Maxime Carrere, Léo Féliers, Adrien Gauffriau, Vincent Mussot, Claire Pagetti, and Thierry Sammour. *LARD – Landing Approach Runway Detection – Dataset for Vision Based Landing*. arXiv:2304.09938 [cs]. Apr. 2023. DOI: 10.48550/arXiv.2304.09938. URL: http://arxiv.org/abs/2304.09938 (visited on 12/13/2024).

[17] Eugene F. Fama. "Efficient Capital Markets: A Review of Theory and Empirical Work." In: *The Journal of Finance* 25.2 (1970). Publisher: [American Finance Association, Wiley], pp. 383–417. ISSN: 0022-1082. DOI: 10.2307/2325486. URL: https://www.jstor.org/stable/2325486 (visited on 03/04/2025).

[18] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. *Generative Adversarial Networks*. arXiv:1406.2661 [stat]. June 2014. DOI: 10.48550/arXiv.1406.2661. URL: http://arxiv.org/abs/1406.2661 (visited on 03/05/2025).

[19] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium." In: *Advances in Neural Information Processing Systems*. Vol. 30. Curran Associates, Inc., 2017. URL: https://papers.nips.cc/paper/2017/hash/8a1d694707eb0fefe65871369074926d-Abstract.html (visited on 03/05/2025).

[20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising Diffusion Probabilistic Models." In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 6840–6851. URL: https://proceedings.neurips.cc/paper/2020/hash/4c5bcfec8584af0d967f1ab10179ca4b-Abstract.html (visited on 12/15/2024).

[21] Glenn Jocher, Jing Qiu, and Ayush Chaurasia. *Ultralytics YOLO*. Jan. 2023. URL: https://github.com/ultralytics/ultralytics.

[22] Alexander B. Jung et al. *imgaug*. 2020. URL: https://github.com/aleju/imgaug.

[23] Guohua Li, S Baker, Jurek Grabowski, and George Rebok. "Factors associated with pilot error in aviation crashes." In: *Aviation, space, and environmental medicine* 72 (Feb. 2001), pp. 52–8.

[24] Ye Li, Yu Xia, Guangji Zheng, Xiaoyang Guo, and Qingfeng Li. "YOLO-RWY: A Novel Runway Detection Model for Vision-Based Autonomous Landing of Fixed-Wing Unmanned Aerial Vehicles." en. In: *Drones* 8.10 (Oct. 2024). Number: 10 Publisher: Multidisciplinary Digital Publishing Institute, p. 571. ISSN: 2504-446X. DOI: 10.3390/drones8100571. URL: https://www.mdpi.com/2504-446X/8/10/571 (visited on 03/04/2025).

[25] Alexander Loth, Martin Kappes, and Marc-Oliver Pahl. *Blessing or curse? A survey on the Impact of Generative AI on Fake News*. arXiv:2404.03021 [cs] version: 2. Dec. 2024. DOI: 10.48550/arXiv.2404.03021. URL: http://arxiv.org/abs/2404.03021 (visited on 03/03/2025).

[26] Andreas Lugmayr, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. "RePaint: Inpainting using Denoising Diffusion Probabilistic Models." In: *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 2575-7075. June 2022, pp. 11451–11461. DOI: 10.1109/CVPR52688.2022.01117. URL: https://ieeexplore.ieee.org/document/9880056 (visited on 03/05/2025).

[27] MIT. *LabelMe. The Open annotation tool*. URL: http://labelme.csail.mit.edu/Release3.0/ (visited on 03/04/2025).

[28] Midjourney. *Midjourney*. URL: https://www.midjourney.com/website (visited on 03/03/2025).

[29] Jorge Pessoa. *jorge-pessoa/pytorch-msssim*. original-date: 2018-04-09T15:08:13Z. Mar. 2025. URL: https://github.com/jorge-pessoa/pytorch-msssim (visited on 03/31/2025).

[30] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection*. arXiv:1506.02640 [cs]. May 2016. DOI: 10.48550/arXiv.1506.02640. URL: http://arxiv.org/abs/1506.02640 (visited on 03/31/2025).

[31] Ilya Reutov. "Generating of synthetic datasets using diffusion models for solving computer vision tasks in urban applications." In: *Procedia Computer Science*. 12th International Young Scientists Conference in Computational Science, YSC2023 229 (Jan. 2023), pp. 335–344. ISSN: 1877-0509. DOI: 10.1016/j.procs.2023.12.036. URL: https://www.sciencedirect.com/science/article/pii/S1877050923020264 (visited on 03/03/2025).

[32] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. "High-Resolution Image Synthesis With Latent Diffusion Models." en. In: 2022, pp. 10684–10695. URL: https://openaccess.thecvf.com/content/CVPR2022/html/Rombach_High-Resolution_Image_Synthesis_With_Latent_Diffusion_Models_CVPR_2022_paper (visited on 03/03/2025).

[33] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. arXiv:1505.04597 [cs]. May 2015. DOI: 10.48550/arXiv.1505.04597. URL: http://arxiv.org/abs/1505.04597 (visited on 12/15/2024).

[34] Daniel Saragih, Atsuhiro Hibi, and Pascal Tyrrell. *Using Diffusion Models to Generate Synthetic Labelled Data for Medical Image Segmentation*. arXiv:2310.16794 [eess]. May 2024. DOI: 10.48550/arXiv.2310.16794. URL: http://arxiv.org/abs/2310.16794 (visited on 03/03/2025).

[35] Alexandru Telea. "An Image Inpainting Technique Based on the Fast Marching Method." In: *Journal of Graphics Tools* 9.1 (Jan. 2004). Publisher: Taylor & Francis, pp. 23–34. ISSN: 1086-7651. DOI: 10.1080/10867651.2004.10487596. URL: https://www.tandfonline.com/doi/abs/10.1080/10867651.2004.10487596 (visited on 03/22/2025).

[36] Roy Voetman, Maya Aghaei, and Klaas Dijkstra. *The Big Data Myth: Using Diffusion Models for Dataset Generation to Train Deep Detection Models*. arXiv:2306.09762 [cs]. June 2023. DOI: 10.48550/arXiv.2306.09762. URL: http://arxiv.org/abs/2306.09762 (visited on 03/03/2025).

[37] Stéfan J. van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. "scikit-image: image processing in Python." In: *PeerJ* 2 (June 2014), e453. DOI: 10.7717/peerj.453. URL: https://doi.org/10.7717/peerj.453.

[38] Qiang Wang, Wenquan Feng, Hongbo Zhao, Binghao Liu, and Shuchang Lyu. "VALNet: Vision-Based Autonomous Landing with Airport Runway Instance Segmentation." en. In: *Remote Sensing* 16.12 (Jan. 2024). Number: 12 Publisher: Multidisciplinary Digital Publishing Institute, p. 2161. ISSN: 2072-4292. DOI: 10.3390/rs16122161. URL: https://www.mdpi.com/2072-4292/16/12/2161 (visited on 12/14/2024).

[39] Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. "Image quality assessment: from error visibility to structural similarity." In: *IEEE Transactions on Image Processing* 13.4 (Apr. 2004). Conference Name: IEEE Transactions on Image Processing, pp. 600–612. ISSN: 1941-0042. DOI: 10.1109/TIP.2003.819861. URL: https://ieeexplore.ieee.org/document/1284395 (visited on 03/05/2025).

[40] Long Xin, Zimu Tang, Weiqi Gai, and Haobo Liu. "Vision-Based Autonomous Landing for the UAV: A Review." en. In: *Aerospace* 9.11 (Nov. 2022), p. 634. ISSN: 2226-4310. DOI: 10.3390/aerospace9110634. URL: https://www.mdpi.com/2226-4310/9/11/634 (visited on 02/27/2025).

[41] Ling Yang, Zhilong Zhang, Yang Song, Shenda Hong, Runsheng Xu, Yue Zhao, Wentao Zhang, Bin Cui, and Ming-Hsuan Yang. *Diffusion Models: A Comprehensive Survey of Methods and Applications*. arXiv:2209.00796 [cs]. Dec. 2024. DOI: 10.48550/arXiv.2209.00796. URL: http://arxiv.org/abs/2209.00796 (visited on 12/14/2024).

[42] Run Ye, Chao Tao, Bin Yan, and Ting Yang. "Research on Vision-based Autonomous Landing of Unmanned Aerial Vehicle." In: *2020 IEEE 3rd International Conference on Automation, Electronics and Electrical Engineering (AUTEEE)*. Nov. 2020, pp. 348–354. DOI: 10.1109/AUTEEE50969.2020.9315584. URL: https://ieeexplore.ieee.org/abstract/document/9315584 (visited on 03/03/2025).

[43] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. *Adding Conditional Control to Text-to-Image Diffusion Models*. arXiv:2302.05543 [cs]. Nov. 2023. DOI: 10.48550/arXiv.2302.05543. URL: http://arxiv.org/abs/2302.05543 (visited on 03/05/2025).