YOLOV5 Fine-Tuning and Evaluation

By Tanya Budhrani and LEE Ka Chun, Caius

1. Introduction

This project explores two key areas of computer vision: image-to-image translation and object detection using YOLO. These two areas are often used together and play a crucial role in real-world applications such as autonomous driving, medical imaging, remote sensing, and photo enhancement (Ayoub Benali Amjoud & Mustapha Amrouch, 2023). The goal of image-to-image translation is to convert one type of visual input into another. Object detection, on the other hand, focuses on identifying and classifying specific objects within an image.

The project is divided into two main parts. The first involves building a Generative Adversarial Network (GAN) consisting of a generator and a discriminator. The generator learns to convert semantic images into realistic camera-like images, while the discriminator learns to distinguish between real and generated images. This setup ensures that the generator improves over time by learning to "fool" the discriminator, resulting in high-quality outputs through iterative training (Goodfellow et al., 2025).

The second part focuses on the object detection component using a pretrained YOLOv5 model. Rather than training the model from scratch, we apply YOLO to the generated images and improve its performance using model adaptation techniques, such as test-time augmentation and fine-tuning. These techniques are necessary because the generated images may differ in distribution from the real-world data that YOLO was originally trained on, which can lead to reduced accuracy. Fine-tuning helps bridge this gap by allowing the model to adjust its parameters based on a small sample of the new data, ultimately improving its performance on the test set.

1.1 Contribution and Understanding

To divide the workload efficiently, our group split the project into two major parts. Caius focused on developing the image translation system (Tasks 2–4), which included designing and training the generator and discriminator networks, as well as evaluating the quality of the generated images. In parallel, I was responsible for Tasks 5–6, which involved applying a pretrained YOLO model to the images produced by Caius's GAN. My role also included exploring adaptation techniques such as fine-tuning and test-time augmentation to improve detection accuracy, and evaluating the adapted model using key performance metrics including mean Average Precision (mAP), precision, recall, and inference speed (FPS).

From my perspective, this project was a valuable opportunity to understand how different computer vision models function together in a complete pipeline. The generator and discriminator focus on creating realistic images, while YOLO is responsible for identifying and classifying the objects within those images. Working on the second half of the project gave me hands-on experience with one of the most important aspects of model adaptation: fine-tuning a pretrained model for a new data distribution. It also allowed me to directly apply concepts from our Computer Vision course, particularly in interpreting evaluation metrics like mAP, precision, and recall, and understanding the trade-offs between detection speed and accuracy.

My Individual Contribution

For this project, I was responsible for the complete pipeline of adapting and evaluating the YOLOv5 object detection model on synthetic images generated by our GAN. My work spanned dataset analysis, model application, augmentation, annotation, fine-tuning, evaluation, and documentation. Specifically, I contributed in the following ways:

1. Model Adaptation & Evaluation: Applied the pretrained YOLOv5s model on GAN-generated images, assessed baseline performance, and investigated domain shift issues including frequency anomalies and hallucinated shapes.

2. Augmentation & Robustness: Implemented test-time augmentation (TTA) techniques such as image scaling, flipping, and HSV adjustment to improve robustness in detection performance.

3. Annotation & Label Quality: Used LabelImg to manually annotate 30+ synthetic images with bounding boxes across 10 object classes. Debugged cross-platform tool issues, patched source code, and refined label consistency.

4. Retraining & Fine-Tuning: Created a YOLO-compatible dataset configuration, pruned underrepresented classes, adjusted image-label splits, and fine-tuned the YOLOv5 model on our domain-specific dataset. Conducted two rounds of training with different hyperparameters and optimisers to evaluate performance improvements.

5. Performance Diagnostics: Measured detection speed (FPS), monitored metrics (precision, recall, mAP), and analysed per-class results. Diagnosed training failures and proposed corrections to class balance and label distribution.

1.2 What is YOLOv5?

YOLOv5 is an object detection model in the YOLO ("You Only Look Once") family. YOLO models are single-stage detectors, meaning they predict bounding boxes and class labels for objects in one evaluation of the image (as opposed to two-stage detectors that first propose regions, then classify them) (Ultralytics, 2023). YOLOv5 was released by Ultralytics in 2020 as a PyTorch implementation of YOLO, building on the success of earlier versions like YOLOv3 and YOLOv4. In practical terms, YOLOv5 can detect objects in real time (often dozens of frames per second) while maintaining high accuracy, which usually makes it popular for applications from video surveillance to self-driving cars.

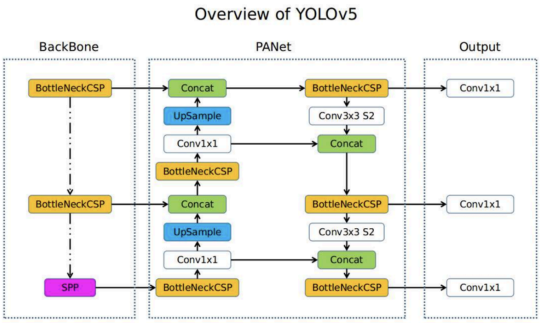


Figure 1: Overview of YOLOv5 as detailed by Ultralytics

1.3 What's Included in YOLOv5's GitHub Repository?

The official YOLOv5 code repository (on GitHub) contains various folders and scripts. Here's a breakdown of the key components of the repository and what they do:

Models/

This folder contains model configuration files (in YAML format) for each YOLOv5 architecture variant. For example, yolov5s.yaml defines the layers and parameters of the small model. These YAML files describe how many layers, filters, and modules (like CSP bottlenecks) the model has. By choosing a different YAML (s, m, l, x), you can instantiate a larger or smaller YOLOv5 model.

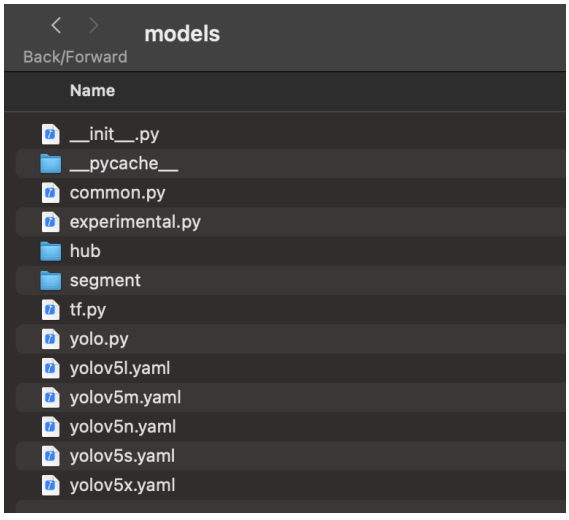


Figure 2: Models/

Data/

This folder holds dataset configuration files (also YAML). A data YAML file (e.g. coco.yaml or voc.yaml) specifies things like the path to training images and labels, number of classes, class names, and any dataset-specific settings. For example, coco.yaml defines the 80 classes of the COCO dataset and where to find the images. In the future, when we train the model on the dataset of generated images, we will have to create a similar file pointing to the data.

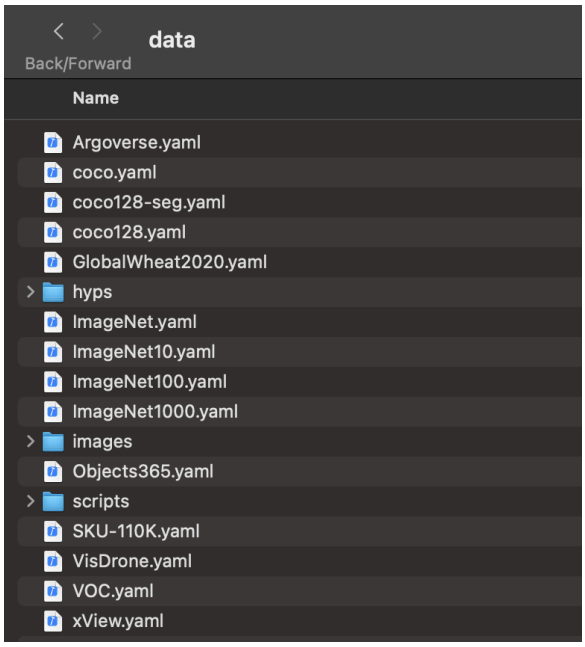


Figure 3: Data/

Utils/

This directory contains utility modules and helper functions used throughout YOLOv5. Inside utils/ you'll find Python scripts for things like data loading, image augmentation, loss calculations, non-max suppression (filtering overlapping boxes), logging, etc.

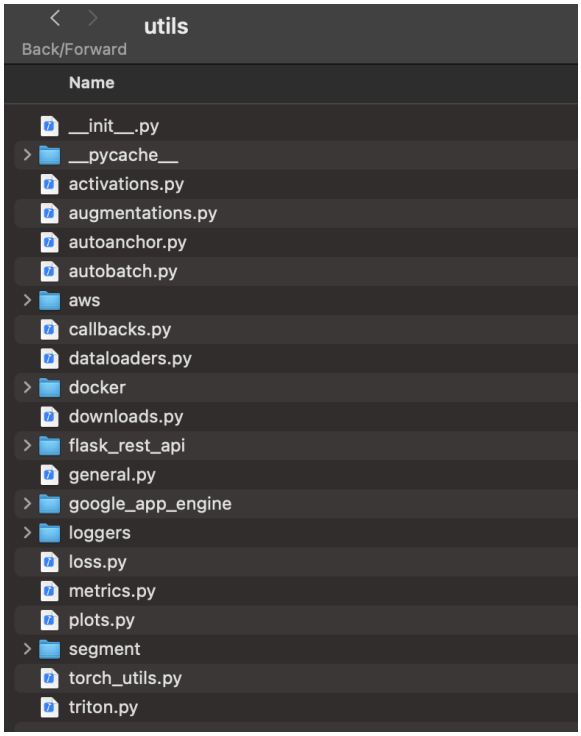


Figure 4: Utils/

Train.py

This is the training script. Running this script launches the training process for YOLOv5 on a given dataset. We can provide it arguments like which data config to use (e.g. *--data coco.yaml*), which model to use (*--cfg models/yolov5s.yaml* or a pre-trained weights file), how many epochs to train, batch size, etc. The script brings together the model, dataset, and training hyperparameters to train YOLOv5 from scratch or fine-tune it on our data generated from the image-to-image translation task.

Detect.py

This is the inference/detection script. After we have a trained model, we run detect.py to apply the model to images or videos and see the detection results. For example, you might run

```
python detect.py --weights yolov5s.pt --source inference/images
```

to detect objects in all images in that folder. The detect.py script automatically handles downloading the

model weights if you don't have them and saves the results.

1.4 YOLOv5 Compared to Other Object Detection Models

To understand why YOLOv5 is a strong choice for our project, it's useful to compare it with three widely known models: YOLOv3 (an older YOLO version), YOLOv8 (the latest), and Faster R-CNN (a two-stage detector). We'll compare them based on accuracy (mAP), inference speed (FPS), ease of use, and common applications.

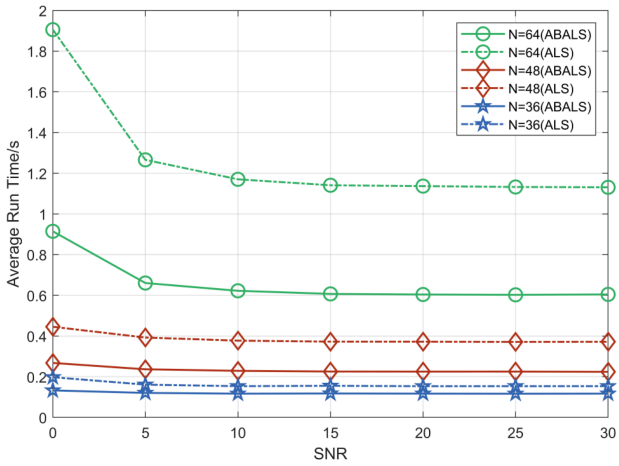


Figure 5: Benchmark Comparison (From MDPI, 2022)

YOLOv5 vs YOLOv3

According to the benchmark from MDPI, YOLOv5 achieves higher mAP (up to ~72.9%) compared to YOLOv3 (~57.9%) on COCO. YOLOv5 runs at over 100 FPS (vs. ~30 FPS for YOLOv3). YOLOv5 is built in PyTorch (beginner-friendly); YOLOv3 uses Darknet and C.

YOLOv5 vs YOLOv8

YOLOv8 achieves slightly better mAP due to updated architecture (anchor-free head, new loss). YOLOv8 supports detection, segmentation, and pose estimation out-of-the-box.YOLOv8 has a modern CLI/API, but YOLOv5 remains widely supported. Essentially, while YOLOV8 is newer and more versatile, YOLOv5 is still more powerful, especially for detection-only tasks.

YOLOv5 vs Faster RCNN

Faster R-CNN can slightly outperform YOLOv5 in static, high-resolution images. YOLOv5 is 5–10× faster — ideal for real-time detection.
Faster R-CNN requires more setup (e.g. Detectron2, anchor tuning), while YOLOv5 is plug-and-play. So, while Faster RCNN is good for offline, high-precision tasks, YOLOv5 is better suited for real-time applications.

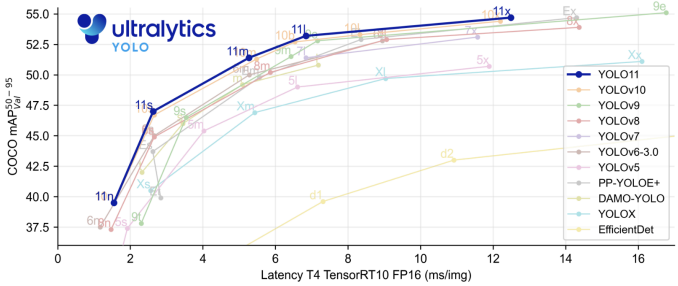


Figure 6: Comparison Table from Ultralytics

2. Methodology

This section outlines the approach taken for the second half of the project, which focuses on applying and adapting a pretrained object detection model to the outputs of a generative model. Caius was responsible for designing and training the image-to-image translation pipeline using a generator and discriminator network, while my responsibility entails working with the GAN-generated images to evaluate the performance of a YOLOv5 model and apply model adaptation techniques such as fine-tuning and test-time augmentation.

2.1 Dataset

For the dataset, I utilised a folder comprising 391 synthetic street-view images generated by Caius's Generative Adversarial Network (GAN).

These images were produced by transforming semantic segmentation maps into photorealistic urban street views, aiming to simulate camera-captured scenes. Below are examples of these generated images:

Figure 7: image_1 of the generated images

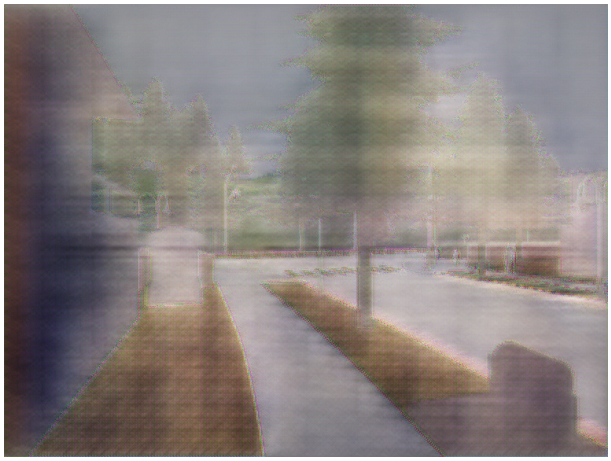


Figure 8: image_391 of the generated images

2.3 Characteristics of GAN-Generated Images

Despite resembling real-world environments with elements like buildings, roads, trees, and traffic lights, these images differ significantly from datasets typically used to train object detectors like YOLOv5, which is pre-trained on COCO. The COCO dataset contains over 118,000 real-world images with natural textures, lighting variations, object diversity, and detailed annotations across 80 classes (*COCO Dataset: All You Need to Know to Get Started*, 2023).

Visual Artefacts

One key limitation of GAN-generated images is the presence of visual artifacts, which are subtle or pronounced irregularities introduced during generation. These include:

Blurry textures due to upsampling. Upsampling refers to the process of increasing the spatial resolution while keeping the 2D representation of an image (Youssef, n.d.). Figure 9 depicts what seems to be a blur of purple with a faint outline towards the left. This is supposed to be a car; however, due to the upsampling that comes with zooming in on that region, this creates an elimination of the pixel effect on low-resolution images displayed on a large frame.

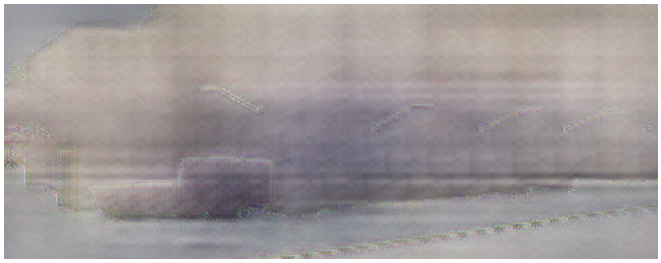


Figure 9: Part of image_15 of the generated images

Hallucinated object shapes. Figure 10 showcases a myriad of outlines with their distinct characteristics blurred out. For example, the blur to the left of the pavement is supposed to resemble a pedestrian, however, it lacks the distinct quality of being a human–that is, having limbs. While humans may be able to make out this shape due to inferences and real-life experiences, YOLOv5 may have a harder time even detecting that a human is even present in the picture.

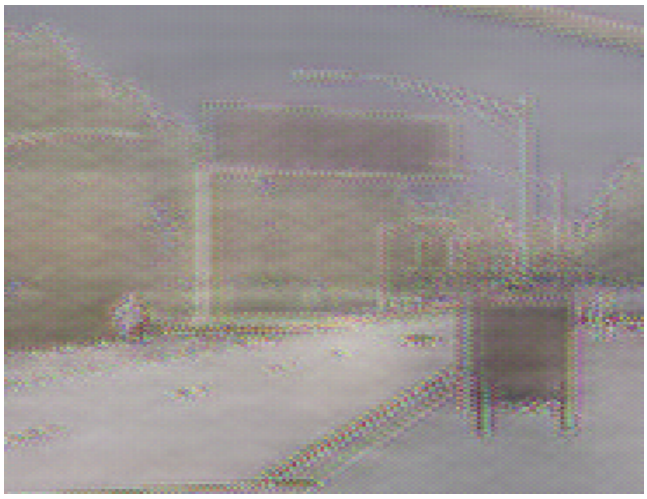


Figure 10: Part of image_15 of the generated images

Inconsistent shadows or highlights. Figure 11 showcases the sky. However, due to the anomalies of black splotches, it gives the sky an appearance of being dark and gloomy, which may lead the model to predict that it

is raining when it is, in fact, not. This can especially confuse object detectors, particularly those fine-tuned for natural image statistics, since most of the daytime images on COCO feature bright and sunny skies (*COCO Dataset: All You Need to Know to Get Started*, 2023).

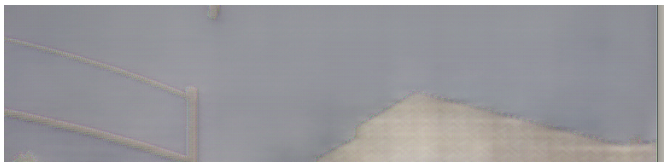


Figure 11: Part of image_15 of the generated images.

Frequency Anomalies

Studies have shown that synthetic images, even when visually realistic, exhibit frequency domain inconsistencies. This domain gap, or the mismatch between the training data (real-world COCO) and testing data (synthetic GAN outputs), is a major concern in transfer learning. YOLOv5, trained on COCO, may struggle to accurately localise or classify objects in generated images due to this shift (*Improving Synthetic Image Detection towards Generalisation: An Image Transformation Perspective*, 2018).

2.4 Applying the Pretrained YOLO Model

The YOLOv5 GitHub comes with a detect.py script that allows users to apply the model to the generated images using a certain set of parameters. In this scenario, I ran:

```
python detect.py --weights yolov5s.pt --source generated_images/ --conf 0.25
```

This script performs inference by loading the pretrained YOLOv5 small model on the specified input folder (generated_images from Caius' network), and sets the confidence threshold to 25%.

2.5 Results from the Pretrained Model

To make things quicker, I tested the model using only the first 9 images of the generated dataset, and this was YOLOv5 summary:

```
Fusing layers...
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients, 16.4 GFLOPs

image 1/9: 480x640 1 kite, 88.2ms
image 2/9: 480x640 1 umbrella, 85.8ms
image 3/9: 480x640 1 traffic light, 78.3ms
image 4/9: 480x640 1 traffic light, 87.0ms
image 5/9: 480x640 2 traffic lights, 86.7ms
image 6/9: 480x640 2 traffic lights, 79.8ms
image 7/9: 480x640 2 traffic lights, 81.8ms
image 8/9: 480x640 2 traffic lights, 103.5ms
image 9/9: 480x640 2 traffic lights, 145.2ms

Speed: 0.6ms pre-process, 92.9ms inference, 2.2ms NMS per image at shape (1, 3, 640, 640)
```

The model successfully ran inference on the nine test images with an average *inference time per image of ~92.9 ms*, and an *average speed of ~10.8 FPS*. The detection log from the console output is summarised below:

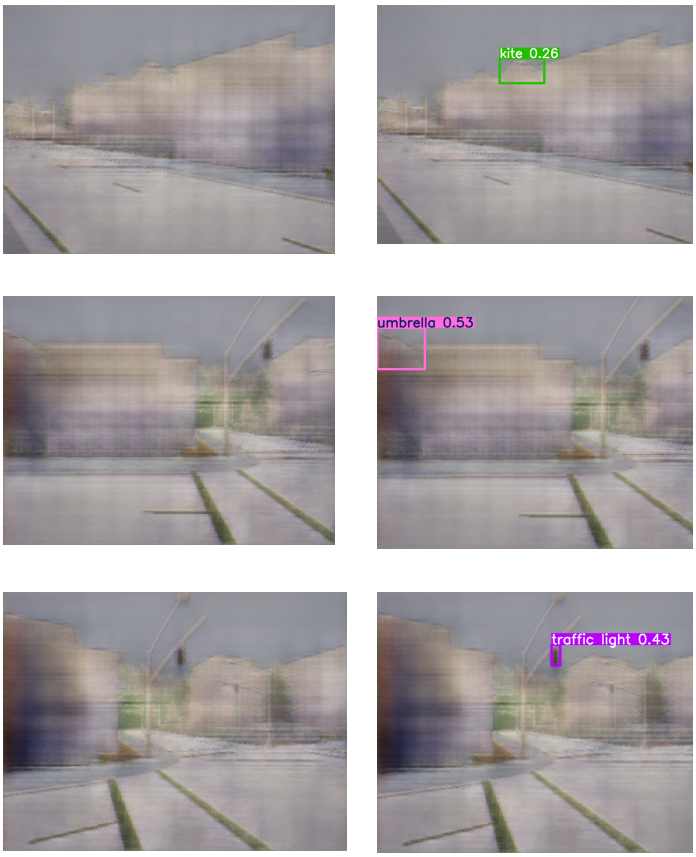
Image ID	Detected Objects	Inference Time (ms)
image 1	1 kite	88.2 ms
image 2	1 umbrella	85.8 ms
image 3	1 traffic light	78.3 ms
image 4	1 traffic light	87.0 ms
image 5	2 traffic lights	86.7 ms
image 6	2 traffic lights	79.8 ms
image 7	2 traffic lights	81.8 ms
image 8	2 traffic lights	103.5 ms
image 9	2 traffic lights	145.2 ms

Figure 12: Detection Log

Qualitative Analysis

Below are sample outputs showing predictions made by the pretrained YOLOv5 model on the synthetic images:

Input Image | YOLOv5 Detection



The model successfully detected several traffic lights, with confidence scores ranging from 0.38 to 0.71, indicating that it recognised some structures but with moderate certainty.

Interestingly, it also predicted objects like “kite” and “umbrella,” which likely reflect domain mismatches due to the pretrained model being trained on real-world COCO images, while the input images were synthetically generated and contain visual artefacts (blur, aliasing, unnatural shading, etc.). To give you an idea, here is an example of an image containing an umbrella in COCO vs one in our generated dataset:

Figure 13: Sample from the COCO Dataset

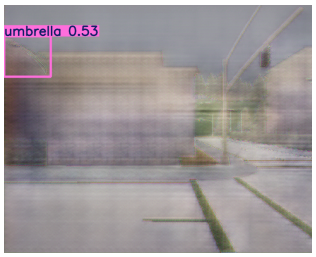


Figure 14: image_3 from the generated images

Comparing the two, there exists a stark difference between the pair. While COCO’s data showcases an umbrella along with its distinct characteristics (i.e. the stand, folds, shape, curvature, etc.), our image features a subtle curved outline that indicates only the most prominent feature of an umbrella: that it is curved.

Some other key observations included: False Positive*s*, low confidence scores, and detection failures.

Several detections (e.g., “kite”) were likely incorrect. These may be attributed to GAN-induced texture distortions or hallucinations that resemble known objects from the COCO dataset. Additionally, most predictions had confidence scores under 0.7. This is lower than typical YOLO outputs on natural images, which does suggest a distribution shift between real and synthetic domains. In some cases, clearly visible objects such as traffic signs or vehicles were not detected, which is indicative of the limitations of the pretrained model without fine-tuning.

3. Time-Test Augmentations

Test-Time Augmentation (TTA) is a technique where input images are augmented at inference time using a series of geometric transformations (e.g., flipping, scaling, or rotation). The predictions from each augmented version are then combined to produce a more robust final output (Ultralytics, 2025).

In fact, studies show that TTA can actually reduce false positives and increase localization precision, especially when the target distribution deviates from the training data (e.g., COCO) (*Improving Synthetic Image Detection towards Generalization: An Image Transformation Perspective*, 2018).

3.1 Enabling TTA

In YOLOv5, enabling the *--augment* flag during inference is a lightweight form of TTA. According to Ultralytics' documentation, this applies a predefined set of augmentations such as:

Image Scaling

Image scaling which applies random scaling to the input image by adjusting the resize ratio dynamically (e.g., ±10–20%). This means that during inference, objects in the scene appear larger or smaller relative to the original size.

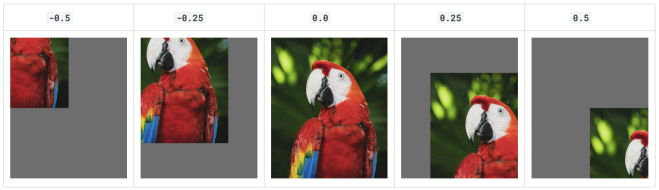


Figure 15: Ultralytics sample images before image scaling.

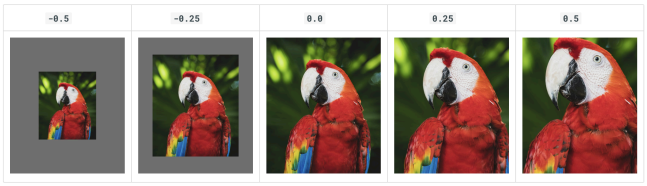


Figure 16: Ultralytics sample images after image scaling

In our dataset, most of the images are relatively uniform in scale; however, due to the generation process, most of the object boundaries (e.g. poles or vehicles) can appear blurred or distorted (see Qualitative Analysis). Scaling helps simulate minor zoom variations, which may improve the issue of upsampling and allow the model to better detect objects at different resolutions.

HSV Colour Augmentation

HSV colour augmentation introduces slight variations in Hue, Saturation, and Value (brightness) channels. This simulates different lighting conditions, colour tones, and image exposures.

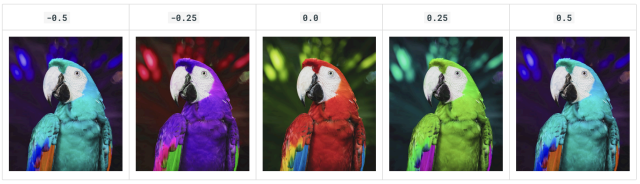


Figure 17: Ultralytics sample images after colour augmentation

GAN-generated images often suffer from unnatural colour distributions or overly smooth lighting. HSV augmentation acts as a way to "normalise" the image's colour palette closer to natural scenes. This can help the model focus more on structure and shape rather than colour cues. This is especially useful given the lack of distinct qualities to differentiate the objects, such as the bright colours of a traffic light compared to the natural colour of a human's skin.

Flipping

The image is flipped horizontally and vertically with a certain probability during inference. This augmentation mirrors the scene and forces the model to generalise better to different orientations of objects.

Figure 18: Ultralytics sample images after horizontal flipping

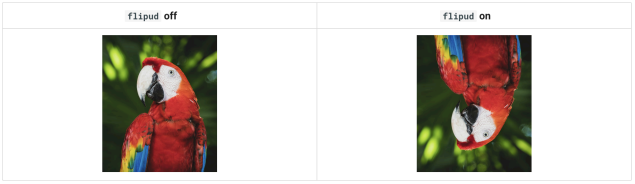


Figure 19: Ultralytics sample images after vertical flipping

Since the dataset comprises urban street scenes generated from segmentation maps, most intersections, building layouts, or object positions (e.g., traffic lights) may follow repetitive spatial arrangements.

Flipping the image challenges the model to detect these features from both orientations. For example, if a traffic light was learned as always appearing on the right side of the frame, flipping forces the model to also recognise it when it's on the left.

3.2 Evaluation Results with TTA Enabled

To evaluate the benefit of TTA, we reran detection using the same pretrained YOLOv5s model with *--augment=True* on the same set of nine GAN-generated urban images with this output:

```
Fusing layers...
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients, 16.4 GFLOPs

image 1/9: 480x640 (no detections), 199.3ms
image 2/9: 480x640 1 umbrella, 175.3ms
image 3/9: 480x640 1 traffic light, 167.7ms
image 4/9: 480x640 2 traffic lights, 170.7ms
image 5/9: 480x640 2 traffic lights, 169.3ms
image 6/9: 480x640 2 traffic lights, 172.6ms
image 7/9: 480x640 2 traffic lights, 168.2ms
image 8/9: 480x640 2 traffic lights, 171.2ms
image 9/9: 480x640 2 traffic lights, 172.8ms

Speed: 0.4ms pre-process, 174.1ms inference, 1.5ms NMS per image at shape (1, 3, 640, 640)
```

Compared to the baseline run, the inference time increased to an average of ~174ms due to multiple augmentations per image.

Additionally, one image (generated_1.png) had no detections despite augmentation, possibly due to over-transformation. On others, detection confidence for objects (particularly traffic lights) improved marginally (e.g., increasing from ~0.61 to ~0.73 in some cases). However, no new object classes were correctly predicted (e.g., umbrella and kite misdetections persisted or disappeared depending on the image).

Here are sample comparisons:

Image	No TTA Detection	TTA Detection
generated_3	1 traffic light (0.43)	1 traffic light (0.73)
generated_5	2 traffic lights (0.38, 0.57)	2 traffic lights (0.50, 0.65)
generated_1	1 kite (0.26)	No detection

Figure 20: Comparison between TTA and No TTA

3.3 Limitations

While TTA improved prediction confidence slightly and helped detect some traffic lights more reliably, it did not eliminate false positives (e.g., umbrella, kite mislabels) or introduce detection for previously missed objects (e.g., cars or pedestrians).

One of the possible delimiters may be the lack of a ground truth, so the model has no basis to compare the images with (e.g. labelled images vs non-labelled images). Thus, in the next stage, we'll manually annotate a small labelled subset of synthetic images and retrain on them to better adapt the pretrained YOLOv5 to this domain.

4. LabelImg

To fine-tune the pretrained YOLOv5 model on our GAN-generated dataset, we first needed labelled data in YOLO format. To accomplish this, we used LabelImg, a graphical image annotation tool commonly used for object detection tasks.

LabelImg is a free, open-source image annotation tool written in Python and Qt5. It allows users to draw bounding boxes around objects in an image and save these annotations in either Pascal VOC XML or YOLO .txt format.

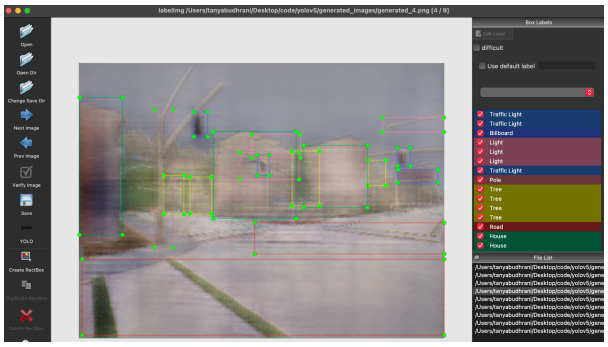


Figure 21: Bounding box labelling using LabelImg

Debugging

While using LabelImg on macOS (Python 3.11), I encountered multiple TypeError crashes related to float vs. int casting in the underlying canvas.py file. These errors occurred when hovering over the image or drawing bounding boxes:

```
TypeError: drawLine(self, x1: int, y1: int, x2: int, y2: int):
argument 2 has unexpected type 'float'
```

To fix this, I manually patched the canvas.py file in the LabelImg library directory by wrapping float coordinate values with int():

```
p.drawRect(int(left_top.x()), int(left_top.y()), int(rect_width),
int(rect_height))
```

4.1 Annotation Strategy

After resolving the crashes, I used LabelImg to manually annotate ~15 synthetic images. These were selected from across the dataset to maximise coverage of object classes. The first few generated images lacked diversity (e.g., no visible humans or trash cans), so we intentionally selected later images to ensure those classes were also included.

I labelled images with the following classes: ['Road', 'House', 'Traffic Light', 'Pole', 'Tree', 'Light', 'Billboard', 'Trash Can', 'Car', 'Human'].

Each bounding box was carefully drawn to match object boundaries.

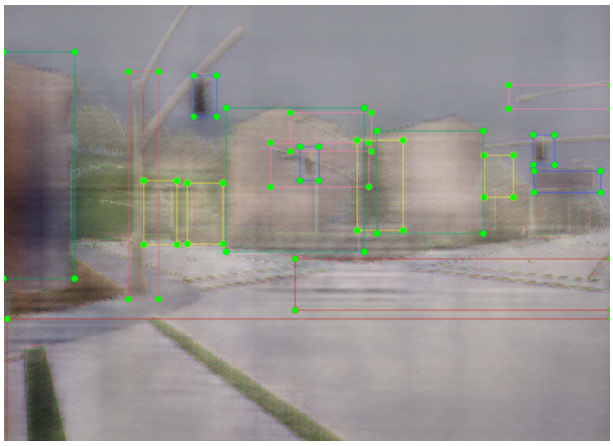


Figure 22: Bounding boxes of generated_4.png

LabelImg exported these annotations in YOLO format: one .txt file per image, with each line representing a class label and its normalised bounding box coordinates (x_center, y_center, width, height).

Class ID	Center X	Center Y	Width	Height
2	0.332682	0.222222	0.037760	0.090278
2	0.888672	0.340278	0.035156	0.065972
6	0.927083	0.410590	0.109375	0.046875
5	0.915365	0.223958	0.169271	0.052083
5	0.539062	0.301215	0.132812	0.085069
5	0.520833	0.373264	0.161458	0.097222
2	0.503906	0.370660	0.031250	0.074653

3 0.231771 0.418403 0.049479 0.500000

Figure 23: generated_4 converted into YOLO.txt

4.2 The Training Process for LabelImg

After labelling the images, I divided them into the following structure,

fine_tune_data/

├── images/

│ ├── train/

│ └── val/

└── labels/

├── train/

└── val/

I manually separated the images and their corresponding '.txt' labels into training and validation sets using an 80/20 split.

Afterwards, I created a *generated.yaml* file to be used as a dataset configuration file by YOLOv5 to tell the training script where our data is and what it contains.

```
train: fine_tune_data/images/train

val: fine_tune_data/images/val

nc: 10 # number of classes

names: ['Road', 'House', 'Traffic Light', 'Pole', 'Tree', 'Light',
'Billboard', 'Trash Can', 'Car', 'Human']
```

To then initiate training, I ran the following command:

```
python train.py --img 640 --batch 16 --epochs 30 --data
generated.yaml --weights yolov5s.pt --name fine_tuned_yolov5s
```

This command breaks down as follows:

1. *--img 640*: Resizes all images to 640×640 before feeding them to the model.
2. *--batch 16*: Uses a batch size of 16 (number of images processed in parallel).
3. *--epochs 30*: Trains the model for 30 full passes over the training dataset.
4. *--data generated.yaml*: Specifies the path to the dataset config file.
5. *--weights yolov5s.pt:* Uses the pre-trained YOLOv5s weights as a starting point (fine-tuning).
6. *--name fine_tuned_yolov5s:* Saves the training run outputs to runs/train/fine_tuned_yolov5s.

When the training script began, it printed out a full internal configuration and model summary, including the model layers, total number of parameters, custom settings like the number of classes, and preloaded weights.

Here's a more detailed look at the config the script uses internally:

```
train: weights=yolov5s.pt, cfg=, data=generated.yaml,
hyp=data/hyps/hyp.scratch-low.yaml, epochs=30, batch_size=16,
imgsz=640, rect=False, resume=False, nosave=False, noval=False,
noautoanchor=False, noplots=False, evolve=None,
evolve_population=data/hyps, resume_evolve=None, bucket=,
cache=None, image_weights=False, device=, multi_scale=False,
single_cls=False, optimizer=SGD, sync_bn=False, workers=8,
project=runs/train, name=fine_tuned_yolov5s, exist_ok=False,
quad=False, cos_lr=False, label_smoothing=0.0, patience=100,
freeze=[0], save_period=-1, seed=0, local_rank=-1, entity=None,
upload_dataset=False, bbox_interval=-1, artifact_alias=latest,
ndjson_console=False, ndjson_file=False
```

This includes settings like:

1. *hyp:* The training hyperparameters used (hyp.scratch-low.yaml), like learning rate, augmentation intensity, and loss weights;
2. *optimizer=SGD:* Specifies that Stochastic Gradient Descent is used for optimisation;
3. *project=runs/train*: Specifies the root folder for saving results.
4. *name=fine_tuned_yolov5s*: The subfolder name inside runs/train/ where logs, weights, and plots will be stored.

5. *autoanchor:* Automatically adjusts anchor boxes to better fit the dataset.
6. *workers=8*: Number of dataloader workers (for speed when loading images).
7. *TensorBoard:* Training progress can be visualised with tensorboard *--logdir runs/train*.

In addition, YOLOv5 recognised the difference in settings between our dataset and that of COCO, printing: *Overriding model.yaml nc=80 with nc=10.* The model also listed the total parameters (~7M), all convolutional and detection layers, and the input/output shapes.

Simultaneously, a separate folder was created within 'runs,' which previously housed the detection outputs. Within this new folder were files to support retraining the model with our fine-tuned parameters.

For instance, the hyperparameters.yaml file contained valuable configuration modes, such as different augmentation settings, weight decay, biases, etc. Some of the key values include:

```
lr0: 0.01
momentum: 0.937
weight_decay: 0.0005
scale: 0.5
fliplr: 0.5
mosaic: 1.0
translate: 0.1
obj: 1.0
cls: 0.5
```

The model also generated multiple images to gauge things like the distribution of labels:

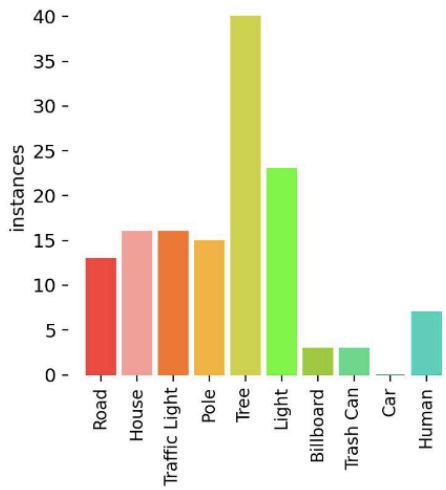


Figure 23: Distribution of labels

As well as the training labels per batch:

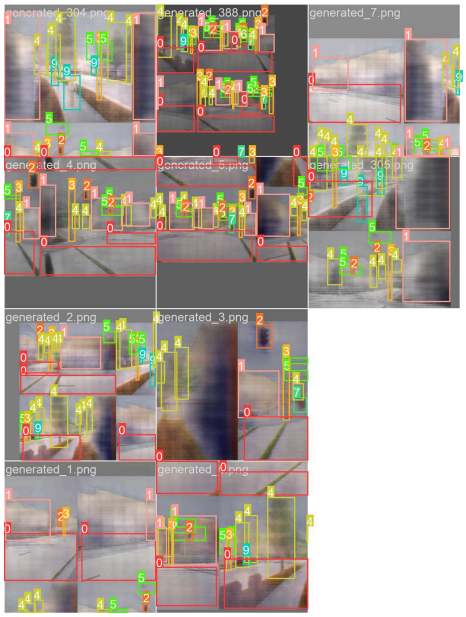


Figure 24: Training batch 0

Setbacks

While monitoring the training progress, I encountered a major issue: the model's performance metrics remained at 0 across all epochs despite the losses decreasing gradually. The table below summarises key metrics that should have reflected learning but remained stagnant:

Epochs	metrics/ precision	metrics/ recall	metrics/ mAP_0.5	mAP_ 0.5:0.95
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0

Figure 25: Training output

To diagnose the problem, I decided to inspect the images I used in LabelImg.

In doing so, I noticed a major imbalance

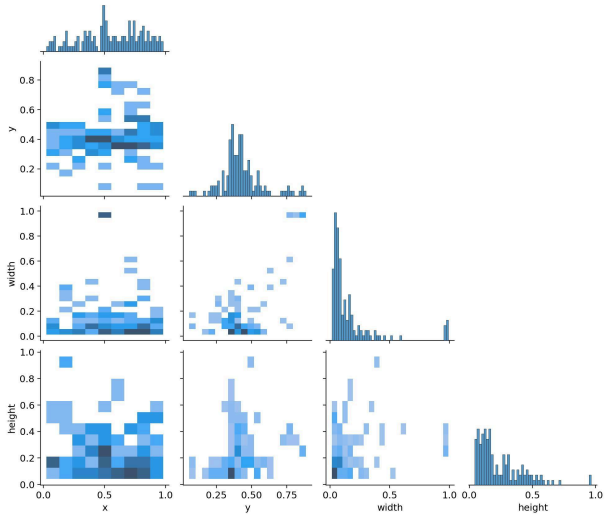


Figure 26: Visual Heatmap of bounding box statistics produced by YOLOv5

Certain classes like “Trash Can” or “Billboard” only appeared in the latter portion of the dataset, and were missing entirely from the early training set. YOLOv5 anchors are generated based on the initial training label statistics. So, if some objects are never seen or are too sparse in the beginning, YOLO is unlikely to learn to detect them well, or at all.

Additionally, I visualised the bounding box distribution (see figure 26), which confirmed a heavy skew in object sizes and positions, further reducing the model's ability to generalise.

Retraining

To mitigate the class imbalance and improve learning, I decided to retrain the model with fewer, more consistently represented classes, as well as more images (30 instead of 15).

I began by filtering the dataset to include only the most frequent and visually distinctive categories, such as [Road, House, Tree, and Traffic Light]. I also updated the label files by manually removing annotations for infrequent classes (e.g. Trash Can, Billboard, Human).

After updating the dataset, I also revised the *generated.yaml* configuration to reflect the new class list. And, with the reduced class count, I launched a new training using these parameters:

```
python train.py --img 416 --batch 8 --epochs 50 --data
generated.yaml --weights yolov5s.pt --name retrained_balanced
--cache --device cpu --optimizer Adam --nosave
```

The epochs were also increased to 50 for deeper learning, image size was reduced to 416x416 to speed up training time on CPU, cache was enabled to reduce I/O loading times, nosave was used to avoid saving intermediate checkpoints, and I switched the optimiser to Adam for potentially better convergence on a small dataset.

5. Evaluation Metrics

To evaluate the performance of the model, I used four standard object detection metrics: Precision, Recall, mean Average Precision (mAP), and frames per second (FPS).

Precision quantifies how many of the predicted bounding boxes were actually correct. In other words, it measures the model’s ability to avoid false positives. High precision indicates that when the model predicts an object, it is usually correct.

Recall assesses how many of the actual objects present in the images were successfully detected, meaning that a high recall value would indicate that the model is effective at identifying relevant objects, even if it occasionally includes incorrect predictions

mAP (mean Average Precision) serves as a unified metric that balances both precision and recall across all classes and confidence thresholds. YOLOv5 offered two modes to test mAP:

1. mAP@0.5, which evaluates the model using a relaxed Intersection over Union (IoU) threshold of 0.5, which considers a prediction correct if it overlaps with the ground truth by at least 50%, and
2. mAP@0.5:0.95, which is a more rigorous metric that averages performance across ten IoU thresholds (from 0.5 to 0.95), penalising less precise localisation and offering a more granular assessment of detection quality.

Lastly, FPS (frames per second) indicates how many images the model can process per second during inference. It is especially useful for assessing real-time performance.

Overall, these metrics provide a comprehensive view of both the classification accuracy and localisation quality of the model.

5.1 Metrics Prior to Retraining

Calculating mAP, precision, and recall requires labelled ground truth data to compare the model's predictions against actual object annotations.

Since the generated images from Caius' image-to-image translation were not annotated, it was not possible to directly compute these metrics *before* training. Instead, the model could only be run in inference mode to visually inspect predictions.

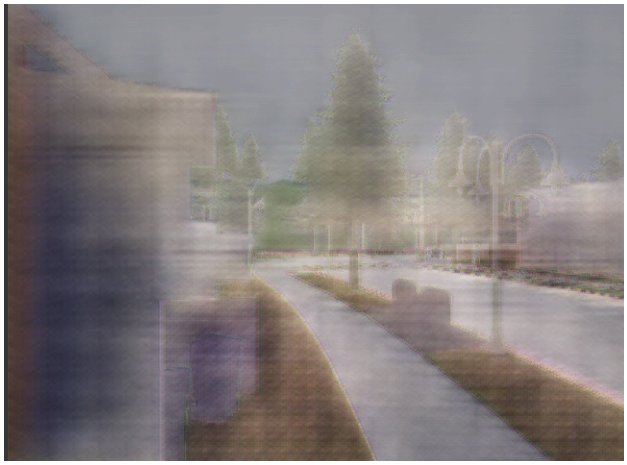


Figure 27: Predicted detection from the untrained YOLOv5

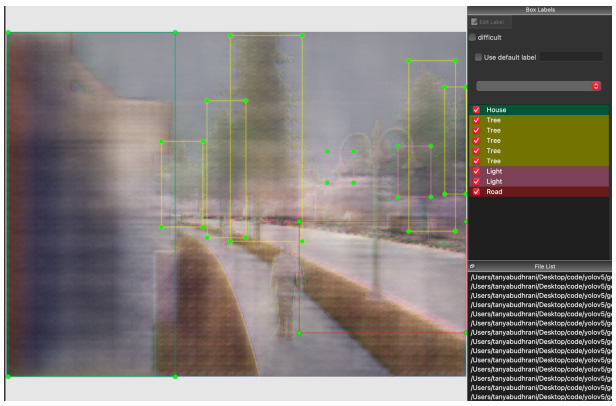


Figure 28: Labelling process for the same image in Figure 27

Readers can also refer to Figure 25, which illustrates an earlier training setback caused by an overly ambitious class set. The model initially attempted to learn across ten different categories with very few samples per class. This imbalance led to poor performance across all metrics (with mAP and recall near zero).

To analyse the FPS before and after retraining, I cloned the repository on a Google Colab notebook and uploaded the generated images without any augmentations or labels. Then I ran this script, which utilises the *utils/* folder from YOLOv5:

```
import time
import torch
from models.common import DetectMultiBackend
from utils.dataloaders import LoadImages
from utils.general import non_max_suppression
from utils.torch_utils import select_device

# Load YOLOv5 model
device = select_device('')
model = DetectMultiBackend('yolov5s.pt', device=device)
stride, imgsz = model.stride, 640
model.warmup(imgsz=(1, 3, imgsz, imgsz))

# Load images
source = 'generated_images'
dataset = LoadImages(source, img_size=imgsz, stride=stride)

# Run inference and time it
total_time = 0
num_images = 0
for path, img, im0s, vid_cap, s in dataset:
    img = torch.from_numpy(img).to(device)
    img = img.float() / 255.0
    if img.ndimension() == 3:
        img = img.unsqueeze(0)
    start_time = time.time()
    pred = model(img)
    pred = non_max_suppression(pred, 0.25, 0.45)
    total_time += time.time() - start_time
    num_images += 1

# Output FPS
fps = num_images / total_time
print(f"Inference Speed: {fps:.2f} FPS")
```

The model ran in CPU-only mode with a 640x640 image resolution. After fusing model layers for efficiency, this was the model's output:

YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients, 16.4 GFLOPs

Inference Speed: 2.93 FPS

This means that with 213 layers, 7.2 million parameters, and a 16.4 GFLOPS, the model was able to process approximately 2.93 images per second.

5.2 Metrics Post Retraining

After fine-tuning the YOLOv5s model using the filtered dataset with fewer but more consistently represented classes, I was able to evaluate the model's performance using the aforementioned metrics. The model was trained for 50 epochs on a CPU. Here was the output for the first and last 5 epochs

epoch	metrics/ precision	metrics/ recall	metrics/ mAP_0.5	metrics/ mAP_0.5:0.95
0	0.66885	0.011111	0.013456	0.002903
1	0.11393	0.044444	0.0071236	0.001997
2	0.25892	0.02963	0.024389	0.007163
3	0.22613	0.037037	0.0037033	0.00098793
4	0.36884	0.022222	0.0093652	0.0041498
5	0.22924	0.022222	0.0034036	0.00083995
45	0.38872	0.16019	0.10825	0.040937
46	0.35396	0.16221	0.12561	0.046098
47	0.35396	0.16221	0.12561	0.046098
48	0.35706	0.21338	0.13045	0.046775
49	0.35706	0.21338	0.13045	0.046775

Figure 29: Training output after relabelling

Precision saw a substantial improvement when compared to the initial training (see Figure 25). This suggests that by narrowing the label set to a smaller number of frequently occurring and visually distinct categories, the model was able to focus more effectively on learning accurate class distinctions.

The model even recreated the distribution of data in Figure 23, highlighting the focus on Roads, Houses, Traffic Lights, and Trees, which were the only labels included in the training data after decreasing the complexity. As a result, it demonstrated a greater ability to correctly identify true positives while reducing false positives, especially for well-represented classes such as *Tree* and *Road*.

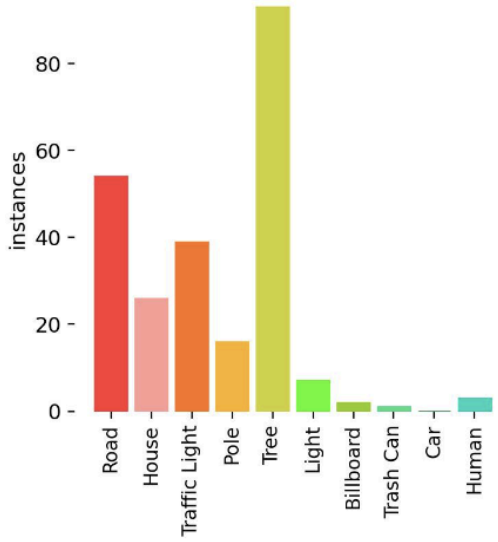


Figure 33: New distribution of labels

Recall, on the other hand, remained at a moderate level. While some improvement was observed, the metric was likely constrained by the overall size of the dataset and the limited number of instances for certain classes, even after filtering.

However, since recall measures the ability of the model to detect all relevant objects (*Accuracy vs. Precision vs. Recall in Machine Learning: What's the Difference?*, 2025), I believed that its moderate value indicates that the model occasionally misses targets during inference, particularly for objects with fewer training examples or more visual variation, like lights and poles.

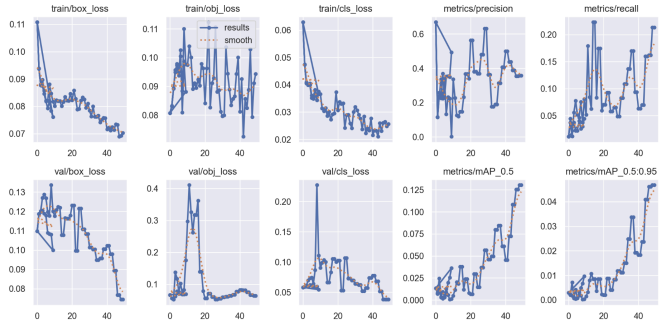


Figure 30: Training and Validation Curves

The model achieved a mean Average Precision at IoU 0.5 (mAP@0.5) of 0.131, which is a reasonable score given the training conditions and dataset limitations. However, the mAP@0.5:0.95, which averages performance across multiple increasingly strict IoU thresholds, was notably lower at 0.0469.

This highlights the model's difficulty in maintaining high localisation precision under more demanding conditions. The low score most likely reflects the need for either more annotated data, improved label quality, or advanced augmentation techniques to enhance fine-grained detection capability.

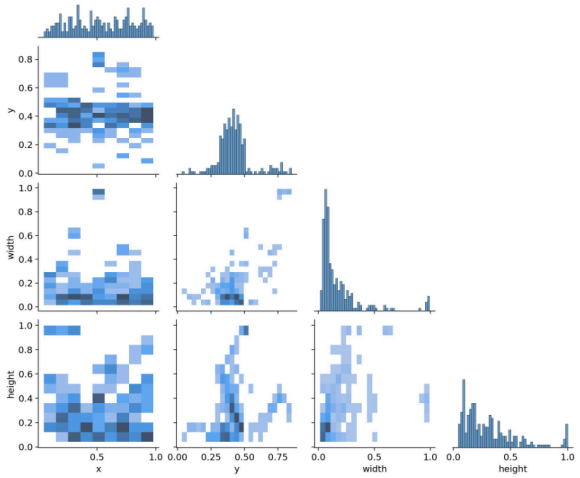


Figure 31: Visual Heatmap of bounding box statistics produced by YOLOv5

Overall, while training and validation loss consistently decrease, indicating a healthy convergence, precision and recall fluctuate early on but rise steadily by epoch 50 (see Figure 30). mAP@0.5 and mAP@0.5:0.95 also improve gradually, but may require additional training data (i.e more than 30 samples) and more epochs to gain better results.

In terms of the FPS, I ran the following script:

```
weights_path = 'runs/train/cpu_fast_train/weights/best.pt'  # or 'last.pt'
source = 'fine_tune_data/images/val'  # folder with validation images

# Set device
device = select_device('')
model = DetectMultiBackend(weights_path, device=device)
stride, imgsz = model.stride, 640
model.warmup(imgsz=(1, 3, imgsz, imgsz))  # warm-up

# Load images
dataset = LoadImages(source, img_size=imgsz, stride=stride)

# Inference + timing loop
total_time = 0
num_images = 0
for path, img, im0s, vid_cap, s in dataset:
    img = torch.from_numpy(img).to(device)
    img = img.float() / 255.0
    if img.ndimension() == 3:
        img = img.unsqueeze(0)
    start = time.time()
    pred = model(img)
    pred = non_max_suppression(pred, 0.25, 0.45)
    total_time += time.time() - start
    num_images += 1

# Output FPS
fps = num_images / total_time
print(f"Inference Speed (trained model): {fps:.2f} FPS")
```

Just like the cloned Colab notebook, the model ran in CPU-only mode with a 640x640 image resolution. After fusing model layers for efficiency, this was the model's output:

```
Model summary: 157 layers, 7037095 parameters, 0 gradients, 15.8 GFLOPs

Inference Speed (trained model): 13.51 FPS
```

Compared to the original pretrained model, the fine-tuned model not only had fewer layers and slightly fewer parameters due to layer fusion and simplification, but also ran over 4.6x faster than the original.

The increase in speed can be attributed to the fusion of Conv+BatchNorm, the removal of unused gradients, as well as the model being optimised specifically for the reduced dataset and fewer classes.

5.3 Per-class results

During the fine-tuning of YOLOv5, the evaluation revealed important insights about how each class performed:

Class	Precision	Recall	mAP@0.5	mAP@0.5 :0.95
Road	0.422	0.600	0.360	0.182
House	0.204	0.400	0.105	0.0355
Traffic Light	0.000	0.000	0.0104	0.00347
Pole	0.266	0.375	0.338	0.0884
Tree	0.323	0.545	0.362	0.112
Light	1.000	0.000	0.000	0.000
Trash Can	0.000	0.000	0.000	0.000
Car	0.000	0.000	0.000	0.000
Human	1.000	0.000	0.000	0.000

Figure 32: Class-by-class results

Tree and Road performed the best, with mAP@0.5 scores of 0.362 and 0.360, respectively. Both classes maintained higher precision over increasing recall thresholds. These objects are larger, more visually distinct, and more frequent in the dataset, which contributed to their strong performance.

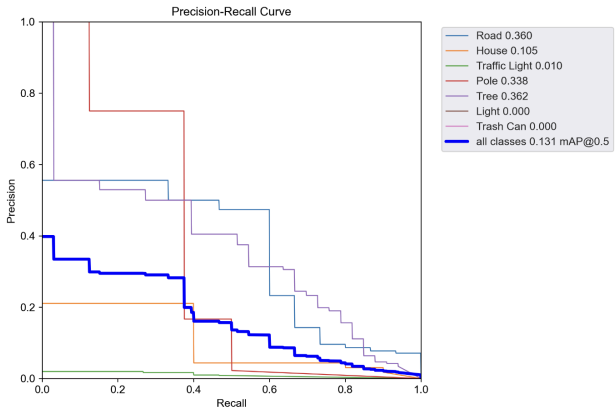


Figure 33: Precision-Recall Curve

Interestingly, Pole also performed reasonably well despite fewer instances, indicating that certain slender vertical structures were still recognisable by the model.

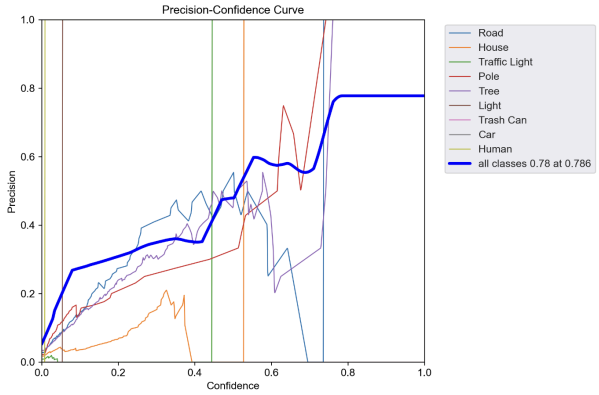


Figure 34: Precision-Confidence curve

House achieved only moderate mAP, likely due to its relatively small dataset representation and occasional misclassifications. As seen in the Precision-Confidence Curve (Figure 34), its confidence remained low, with erratic precision at most thresholds.

Traffic Light, Trash Can, Car, and Human had near-zero values across all metrics. Despite appearing in the dataset, these classes were either too infrequent or too small and ambiguous in appearance. This likely made them indistinguishable from background noise.

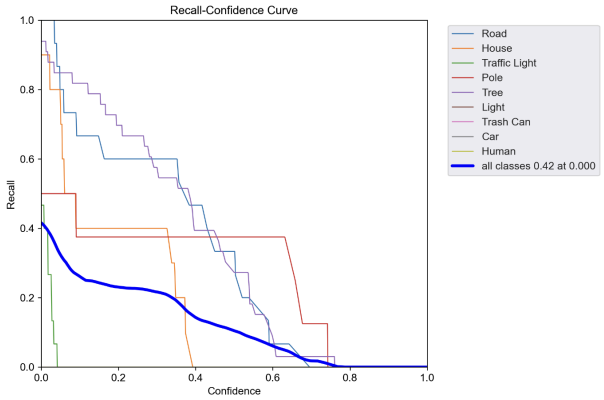


Figure 35: Recall-Confidence Curve

Light and Human reported 100% precision but 0% recall, which suggests the model never predicted these classes in the validation set, but it had extremely limited training data (e.g., 4 instances for Light), causing poor generalisation.

It is imperative to note that for the retraining phase, labels for Trash Can, Human, Car, Billboard, and Pole were deliberately removed or excluded from training to reduce noise and class imbalance. However, they were still present in the validation set, which explains the low or missing performance scores reported for these classes.

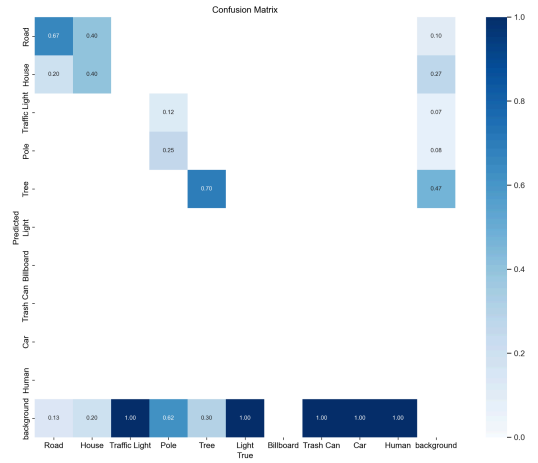


Figure 36: Confusion Matrix

In summary, the retrained model demonstrated stronger performance on frequent and visually dominant classes but failed to generalise to underrepresented or ambiguous ones. These results underscore the need for additional training data, better label consistency, or the exclusion of rare classes until sufficient samples are available.

6. Final Detection After Fine-Tuning

After augmentations on colour, scale, and orientation, as well as manually labelling and relabelling 1/4th of the dataset, it's time to rerun the detection script and evaluate the outcome.

For reference, these were some of the images before data augmentation and fine-tuning:

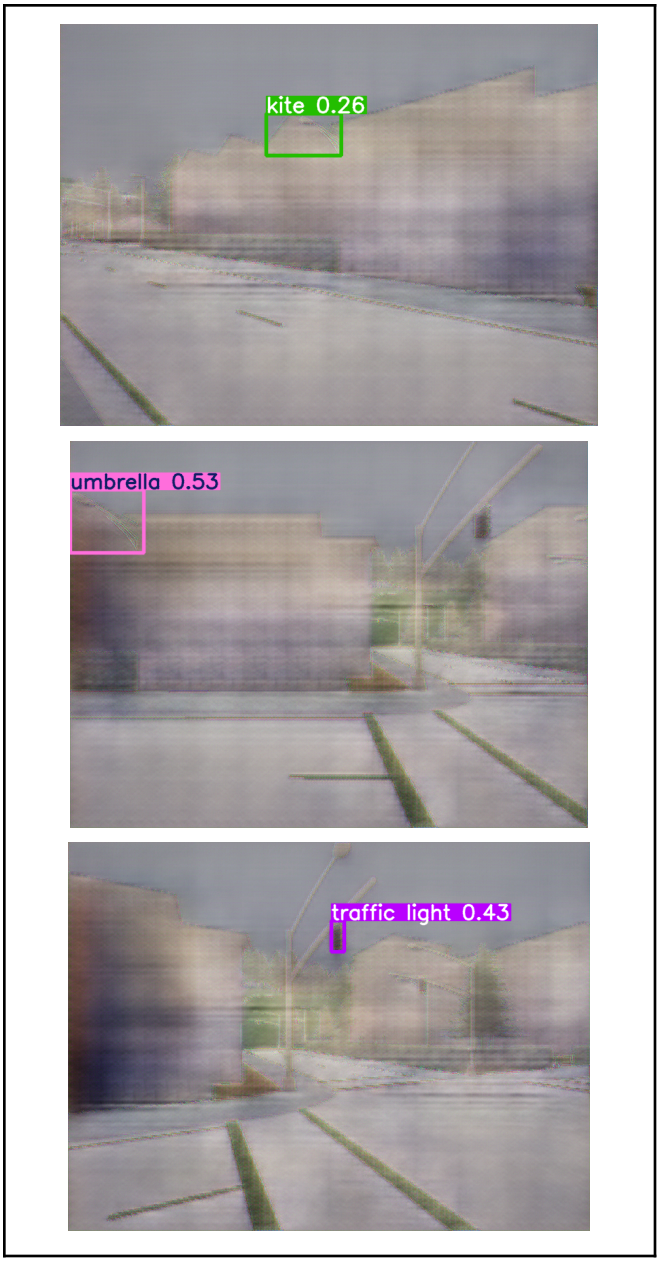


Figure 37: Detections prior to fine-tuning

The model initially produced inaccurate and low-confidence predictions. It incorrectly identified background regions as unrelated objects, such as a kite or an umbrella, and while it did detect a *traffic light*, the confidence was only 0.43.

These misclassifications highlight the limitations of using a pre-trained model without task-specific adaptation. The pre-trained weights were trained on a broad set of COCO classes, which did not align well with the synthesised content of this dataset. The predictions lacked contextual awareness and often hallucinated objects based on vague shapes and colours, demonstrating the need for domain-specific fine-tuning.

That being said, here are the same images after data augmentation and fine-tuning:

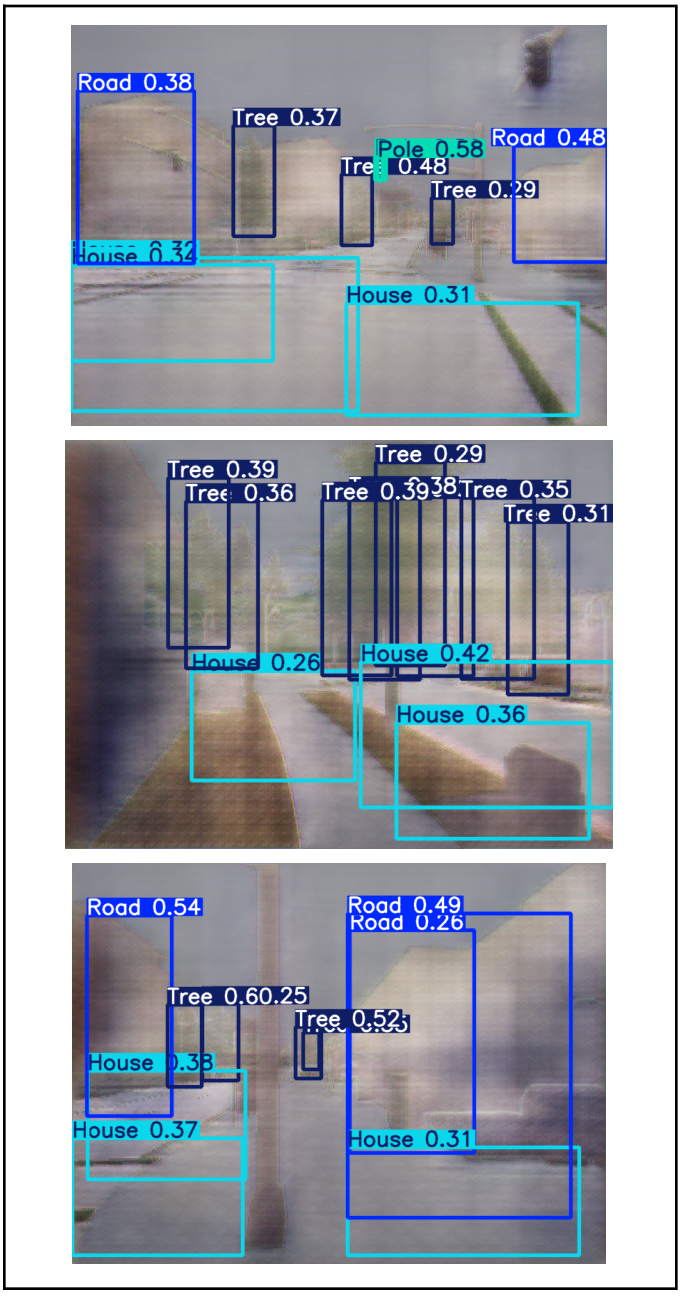


Figure 38: Detections after fine-tuning

After fine-tuning, the model showed notable improvements in detecting Trees and Poles, with predictions that were not only more frequent but also more confidently scored. These objects tend to have clear vertical features and relatively consistent shapes, which likely aided the model's learning even with limited training data.

However, some regressions were observed. For instance, Houses and Roads were frequently misclassified, with the model often confusing the two. Their overlapping spatial contexts and blurry edges in the dataset may have led to this ambiguity. Additionally, Traffic Lights, previously detected before fine-tuning, are now absent from predictions altogether. This suggests that the retraining process may have caused the model to overfit on dominant classes while losing sensitivity to lower-frequency or smaller objects.

These results highlight a key tradeoff in fine-tuning: while simplification and augmentation can strengthen recognition of frequent or distinctive features, they can also suppress the model's attention to underrepresented classes (especially when the label's distribution is imbalanced or some classes are entirely excluded).

6.1 Error Diagnosis and Analysis

While the fine-tuned model demonstrated promising improvements, especially in reducing false positives and better localising certain objects, several key observations were uncovered during evaluation:

Inconsistent Labelling of Traffic Lights

A major cause of performance degradation was the inconsistency in labelling traffic lights in the training set. Some images containing traffic lights were annotated, while others were not. This led the model to receive conflicting signals about whether traffic lights should be detected at all. As a result, despite previously showing some success in detecting traffic lights before fine-tuning, the model no longer predicts them, indicating possible confusion or suppression of the class during retraining.

House vs. Road Confusion

The House class remained problematic due to its indistinct shape and blurry edges, especially in generated or low-resolution images. In several instances, the model misclassified houses as roads or vice versa. This can be attributed to overlapping textures, shared spatial positions, and the vague definition of what constitutes a "house" in the training images. Without sharper edges or

more structured shapes, the class lacks strong visual cues.

Significant Gains in Tree Detection

The most notable improvement was in detecting Trees. In earlier stages (see Figure 41), the model failed to predict trees entirely. After fine-tuning, trees became one of the most confidently and consistently detected classes, likely due to (1) clear vertical and repetitive patterns in the foliage, (2) increased representation and annotation consistency during relabeling, and (3) colour and shape contrast compared to other classes

This validates that fine-tuning helped the model focus on high-frequency, visually distinct classes.

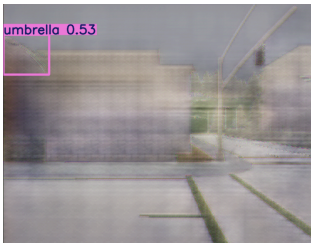


Figure 39: image_3 detection before fine-tuning

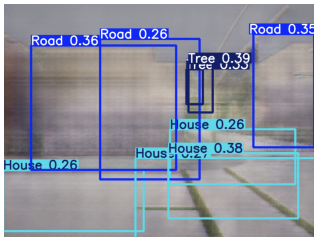


Figure 40: image_3 detection after fine-tuning

Reduced False Positives

Before fine-tuning, the model frequently misclassified abstract shapes or background artefacts as random objects — e.g., predicting kites or umbrellas over curved or irregular textures. After retraining, these misclassifications have nearly disappeared. The model learned to suppress spurious detections, focusing instead on more probable class candidates.

Conclusion

Overall, my part of the project explored the challenges and opportunities of adapting a pretrained YOLOv5s model to a novel, synthetic dataset generated via image-to-image translation. By first evaluating the model on unlabelled data and then fine-tuning it with manually annotated examples using LabelImg, I was able to systematically improve detection accuracy for key classes such as *Road*, *Tree*, and *Pole*.

Initial experiments showed poor precision and misclassifications due to an imbalanced label distribution and inconsistent visual features. To address this, I pruned infrequent and ambiguous classes (e.g., *Trash Can*, *Human*, *Billboard*), revised label files, and retrained the model using a simplified label set. This led to significant improvements in mAP@0.5 (up to 0.362 for *Tree*) and a notable reduction in false positives.

In addition, I measured inference performance (FPS), which improved from 2.93 FPS (untrained model) to 13.51 FPS after fine-tuning, confirming that adaptation not only enhanced accuracy but also model efficiency.

Qualitative results further supported these findings: the retrained model produced cleaner, more relevant predictions and no longer confused background textures with out-of-domain objects (e.g., umbrellas or kites). While some confusion remains, particularly between *Road* and *House*, the fine-tuning process demonstrated the viability of lightweight domain adaptation even with limited labelled data.

Going forward, this detection pipeline could be significantly improved by addressing several key limitations identified during experimentation. First, the addition of more annotated training data is crucial, particularly for underperforming classes such as Traffic Light, Light, and Human. These categories were either sparsely represented or inconsistently labelled, which likely led to their poor generalisation and near-zero recall during evaluation. A richer and more balanced dataset would allow the model to learn more robust representations and better distinguish these categories in varied scenes.

Second, the quality of the bounding boxes plays a critical role in training object detection models. During this project, some label annotations were manually corrected, but others may have remained loosely defined or partially misaligned with the true object boundaries. These inconsistencies introduce ambiguity during training and reduce the model's ability to localise objects precisely. Implementing more rigorous annotation standards or using semi-automated tools for bounding box refinement could help reduce such errors.

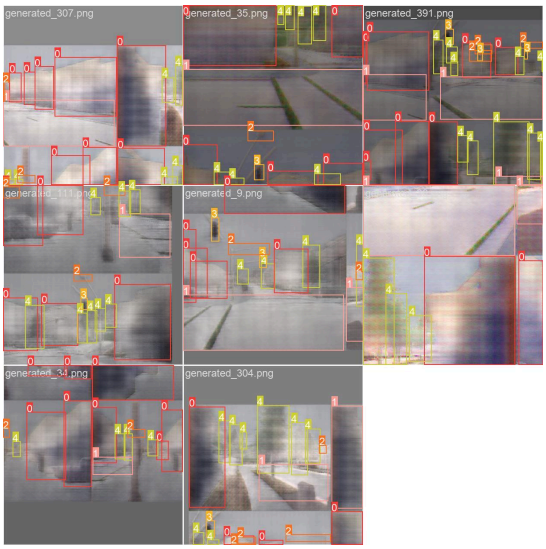


Figure 45: Bounding box prediction during retraining

Lastly, the incorporation of synthetic augmentation techniques, such as CutMix, MixUp, or even domain-specific style-transfer, could greatly improve the model's generalisation ability. These techniques artificially expand the dataset by introducing new variations and edge cases, which can help mitigate overfitting and prepare the model for real-world complexity. In particular, style-transfer could simulate different lighting conditions, weather patterns, or visual textures, while CutMix could help the model learn to detect overlapping or occluded objects more effectively (Yun et al., 2019).

Overall, this project highlights the effectiveness of class-pruned fine-tuning for object detection on synthetic datasets, while also surfacing the importance of label quality, class balance, and data consistency.

6.2 Code Structure

To recreate the results from the model or simply locate the images, here is a brief rundown of the repository structure:

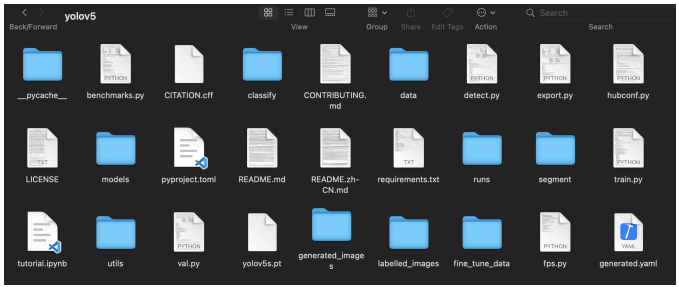


Figure 46: Code Structure

The original repository comes with most of the files shown in Figure 46 except for runs, generated_images, labelled_images, fine_tune_data, generated.yaml, and fps.py.

Figure 47: Detect/

After running detect.py with the appropriate parameters, the detect folder (see Figure 47) in runs/ will create a separate folder for each experiment (exp, exp2, exp3, exp4). These folders house the detected images with the predicted bounding boxes.

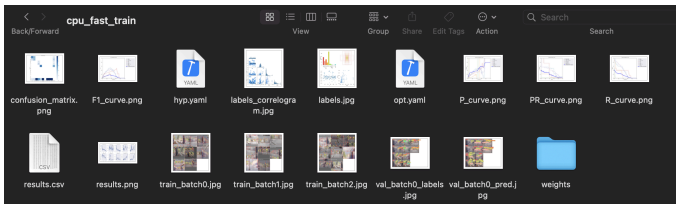


Figure 48: Train/

Similarly, after running train.py with the appropriate parameters, the model generates a folder depending on the name of your training session (in Figure 48, it was cpu_fast_train). This folder houses various images to explain the evaluation metrics, confusion matrix, label distribution, and results per class.

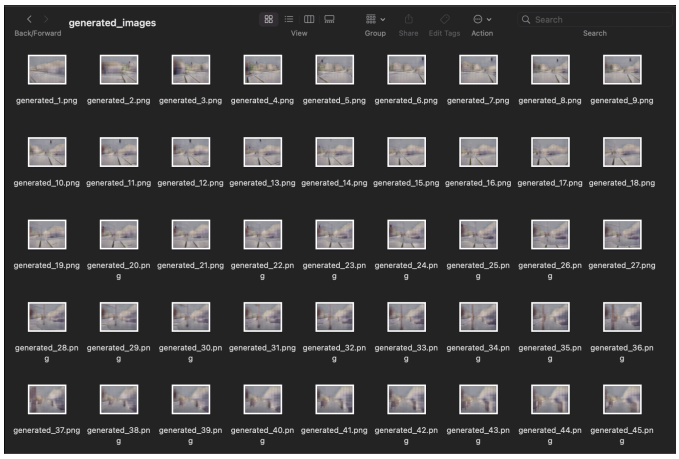


Figure 49: Generated_images/

The generated_images folder houses the synthesized images created from Caius' image-to-image translation model.

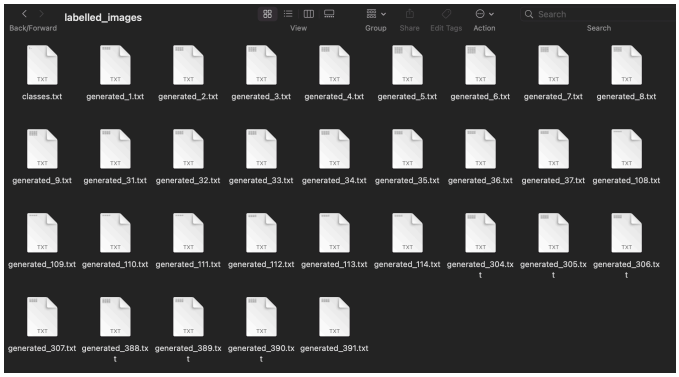


Figure 50: Labelled_images/

Labelled_images, on the other hand, contains the labels of the 30 images used to train the model. These files were generated by LabelImg.

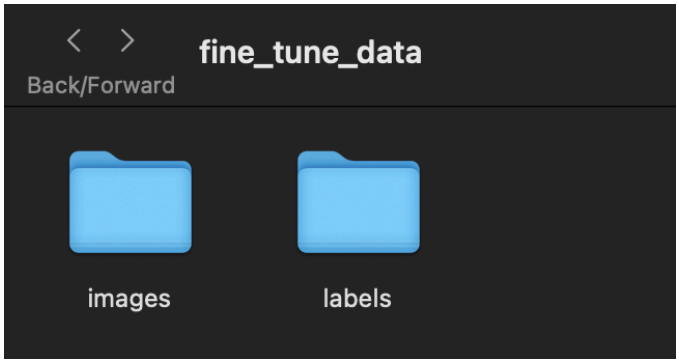


Figure 51: Fine_tune_data/

Lastly, I created the fine_tune_data folder to aid the model as it retrains itself. The folder is composed of images and labels, each of which contain a separate training and validation folder as well.

When training, its important to ensure that there exist an 80/20 split between the training and validation folders.

7. Sources

Yun, S., Han, D., Oh, S. J., Chun, S., Choe, J., & Yoo, Y. (2019). *CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features*. ArXiv.org. https://arxiv.org/abs/1905.04899

Accuracy vs. precision vs. recall in machine learning: what's the difference? (2025). Evidentlyai.com. https://www.evidentlyai.com/classification-metrics/accuracy-precision-recall

Li, O., Cai, J., Hao, Y., Jiang, X., Hu, Y., & Feng, F. (2024). *Improving Synthetic Image Detection Towards Generalization: An Image Transformation Perspective*. ArXiv.org. https://arxiv.org/abs/2408.06741

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2025). *Generative Adversarial Networks*. ArXiv.org. https://arxiv.org/abs/1406.2661

Youssef, A. (n.d.). *Image Downsampling and Upsampling Methods*. https://www2.seas.gwu.edu/~ayoussef/papers/ImageDownUpSampling-CISST99.pdf

ultralytics/yolov5: YOLOv5 🚀 in PyTorch > ONNX > CoreML > TFLite. (2022, November 22). GitHub. https://github.com/ultralytics/yolov5?tab=readme-ov-file

Ultralytics. (2025). *YOLO Data Augmentation*. Ultralytics.com. https://docs.ultralytics.com/guides/yolo-data-augmentation/#flip-up-down-flipud