Project: Image2Image Translation & Image Detection

COMP4423 Computer Vision 2024/25 Semester 2

22055747d, 24028009d, 222080062d

Char Cheuk Tung George, Leung Yuk Kin, Lai Ka Chung

Source code: https://drive.google.com/file/d/1AzLWqF2XBYXBKlaZWd0YH-mK-XCm97bU/view?usp=sharing

1. Introduction

This project undertakes a comprehensive exploration of image-to-image translation and subsequent object detection on the translated imagery. The primary objective is to develop and train a Generative Adversarial Network (GAN) for domain-specific image translation, leveraging a U-Net architecture for the generator and a classifier network for the discriminator. Following successful image translation, the project focuses on adapting a pretrained YOLO (You Only Look Once) model to perform robust object detection on these newly generated images. Key methodologies include the construction of a domain-specific dataset, iterative training of the GAN components, and the application of at least two distinct techniques for YOLO model adaptation, such as Test-Time Adaptation (TTA) and fine-tuning with generated labels. The performance of the image translation will be assessed qualitatively and, where applicable, quantitatively. The efficacy of the adapted YOLO model will be evaluated using standard metrics including mean Average Precision (mAP), Precision, Recall, and Frames Per Second (FPS), comparing performance before and after adaptation. This report details the systematic approach to dataset creation, network design, training procedures, evaluation strategies, challenges encountered, and the solutions implemented, culminating in an analysis of the interplay between image generation quality and object detection accuracy.

2. Dataset

2.1. Dataset Source and Description

The dataset utilized in this project is designed for image-to-image translation, specifically converting semantic segmentation maps into realistic street view images. The core of the dataset consists of paired images, where each pair comprises a semantic segmentation map and its corresponding real-world road scene photograph.

The overall structure of the dataset is as follows:

```
dataset/
├── path1/
|   ├── camera_images_real_front/
#Contains real street view photographs
|   |   ├── 0001.png
|   |   ├── 0002.png
|   |   └── ...
|   └── camera_images_semantic_front/ #Contains corresponding semantic
|   |   ├── 0001.png
|   |   ├── 0002.png
|   |   └── ...
├── path2/
|   ├── camera_images_real_front/
|   |   └── ...
|   └── camera_images_semantic_front/
|   |   └── ...
├── path3/
|   ├── camera_images_real_front/
|   |   └── ...
|   └── camera_images_semantic_front/
|   |   └── ...
```

In this structure, dataset serves as the root directory. It contains multiple subdirectories (e.g., path1, path2, path3), each representing a specific data collection sequence or scene. Within each pathX subdirectory, two core folders are present:

- camera_images_real_front: This folder stores real-world road scene photographs taken from a vehicle's front-facing perspective. These images constitute the target domain for our image translation task.
- camera_images_semantic_front: This folder contains semantic segmentation maps that correspond one-to-one with the images in camera_images_real_front. In these segmentation maps, different regions of the image (such as roads, vehicles, buildings, sky, pedestrians, etc.) are labeled with distinct class colors. These semantic maps serve as the source domain for our image translation task.

Crucially, the images within the camera_images_real_front and camera_images_semantic_front folders are paired, meaning 38930.png in camera_images_semantic_front is the semantic segmentation version of the identically named 38930.png in camera_images_real_front. This strict pixel-level correspondence is vital for training image translation models like Pix2Pix, which require paired data.

2.2. Dataset Statistics

The dataset comprises a total of **4162** images. These images are equally distributed between the two domains:

- **Real street view images: 2081 images**
- **Semantic segmentation maps: 2081 images**

Therefore, we have a total of **2081** pairs of input (semantic map) - output (real photograph) images available for training and evaluating our image-to-image translation model.

3. Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) have revolutionized the field of image synthesis, and the Pix2Pix architecture stands as a prominent example of conditional GANs tailored for image-to-image translation tasks

3.1 Methodology

The implemented Pix2Pix model consists of two core components: a generator network (G) and a discriminator network (D). The design and training strategy are based on the original Pix2Pix formulation.

3.1.1 Network Architectures

- **Generator:** The Pix2Pix model typically uses a **U-Net** architecture for its generator.
    - **Encoder-Decoder Structure:** The U-Net consists of an encoder path that progressively downsamples the input image to learn hierarchical features, and a decoder path that upsamples these features to generate the output image.
    - **Skip Connections:** A key feature of the U-Net is the presence of skip connections between corresponding layers in the encoder and decoder paths. These

connections allow low-level information (like edges and textures) from the encoder to be directly passed to the decoder, helping to preserve finer details in the generated output and mitigating the vanishing gradient problem.

- **Layers:** The encoder typically uses convolutional layers with activation functions like LeakyReLU, followed by Batch Normalization. The decoder uses transposed convolutional layers (deconvolutions) for upsampling, also with Batch Normalization and activation functions (often ReLU, with a Tanh activation for the final output layer to produce images in a normalized range, e.g., [-1, 1]).
- The `pix2pix_models.py` file likely contains the Python class defining this U-Net generator.

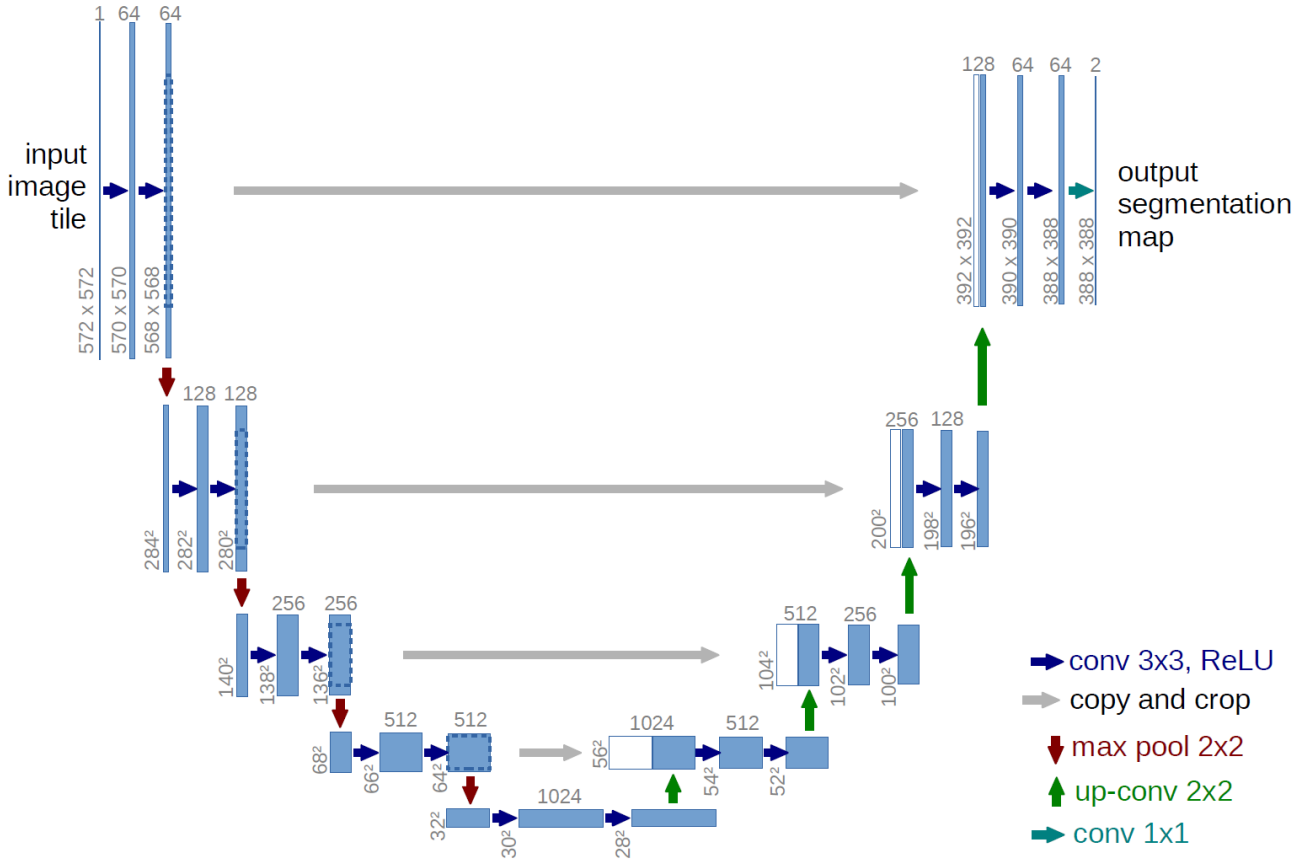


Figure 1 U-Net Architecture [1]

- **Discriminator:** A **PatchGAN** discriminator is standard for Pix2Pix.
  - **Patch-wise Classification:** Instead of classifying the entire generated image as real or fake, the PatchGAN divides the image into N×N patches and classifies each patch independently. This encourages the generator to produce sharp, high-frequency details across the entire image.

- **Architecture:** The PatchGAN is a fully convolutional network that outputs a 2D feature map where each element corresponds to the "realness" score of a patch from the input image.
- **Layers:** It typically consists of a series of convolutional layers, Batch Normalization (often not applied to the first layer), and LeakyReLU activations. The final layer outputs a single channel feature map.
- The `pix2pix_models.py` file would also contain the definition for this PatchGAN discriminator.

3.1.2. Loss Functions

The training process involves optimizing both the generator and the discriminator using a combination of loss functions.

- **Discriminator Loss (L_D):**
  - The discriminator is trained to distinguish between real images (from the target domain) and fake images (generated by the generator). This is typically a binary cross-entropy (BCE) loss, calculated separately for real and fake images. The total discriminator loss is often the sum or average of these two components.
    - $L_d = \frac{1}{2} E_{x,y}[\boldsymbol{log} D(x,y)] + \frac{1}{2} E_{x,z}[\boldsymbol{log}(1 - D(x, G(x,z)))]$
      Where x is the input image, y is the real target image, z is random noise (if applicable, though Pix2Pix is conditional), G(x,z) is the generated image. D(x,y) is the discriminator's probability that y is real given x, and D(x,G(x,z)) is the discriminator's probability that the generated image is real.
- **Generator Loss (L_D):**
  - The generator is trained to fool the discriminator and to produce images that are structurally similar to the target images.
  - **Adversarial Loss (L_{GAN}):** This component encourages the generator to produce images that the discriminator classifies as real. It's typically a BCE loss based on the discriminator's output for fake images.
    - $L_{GAN} = E_{x,z}\left[\log\left(1 - D\left(x, G(x,z)\right)\right)\right] (non - saturating\ version)\ or$
    - $E_{x,z}[-logD(x, G(x,z))]\ (alternative, often\ preferred).$
  - **Reconstruction Loss (L_{L1} or L_{MAE}):** To ensure that the generated images are not just realistic but also faithful translations of the input images, a pixel-wise reconstruction loss is added. The L1 loss (Mean Absolute Error) is commonly used as it tends to produce less blurry results compared to L2 loss.
    - $L_{L1} = E_{(x,y,z)}\ [||\ y - G(x,z)||1\ ]$

- **Total Generator Loss:** The final generator loss is a weighted sum of the adversarial loss and the reconstruction loss.
  - $L_G = L_{GAN} + \lambda L_{L1}$
  - The weighting factor λ (e.g., 100) controls the balance between the two loss terms.

3.1.3. Training Process

The `train_pix2pix.py` script would manage the overall training loop.

- **Optimization:** The Adam optimizer is commonly used for both the generator and discriminator due to its adaptive learning rate capabilities.
- **Alternating Updates:** The generator and discriminator are trained in an alternating fashion. In each training iteration:
  - The discriminator is trained on a batch of real images and a batch of fake images generated by the current generator.
  - The generator is trained based on the discriminator's feedback and the L1 reconstruction loss.
- **Hyperparameters:** Key hyperparameters include the learning rate (e.g., 0.0002), beta1 for Adam (e.g., 0.5), batch size, and the number of epochs.
- **Data Loading and Preprocessing:** The script would also handle loading paired input and target images, resizing them to the network's required input dimensions (e.g., 256×256), and normalizing pixel values (e.g., to the range [-1, 1]).

3.2 Result and analysis

The performance of the Pix2Pix model was evaluated both quantitatively using loss metrics during training/validation/testing and qualitatively by inspecting generated images (as inferred from the code saving samples and test outputs).

3.2.1 Quantitative Results:

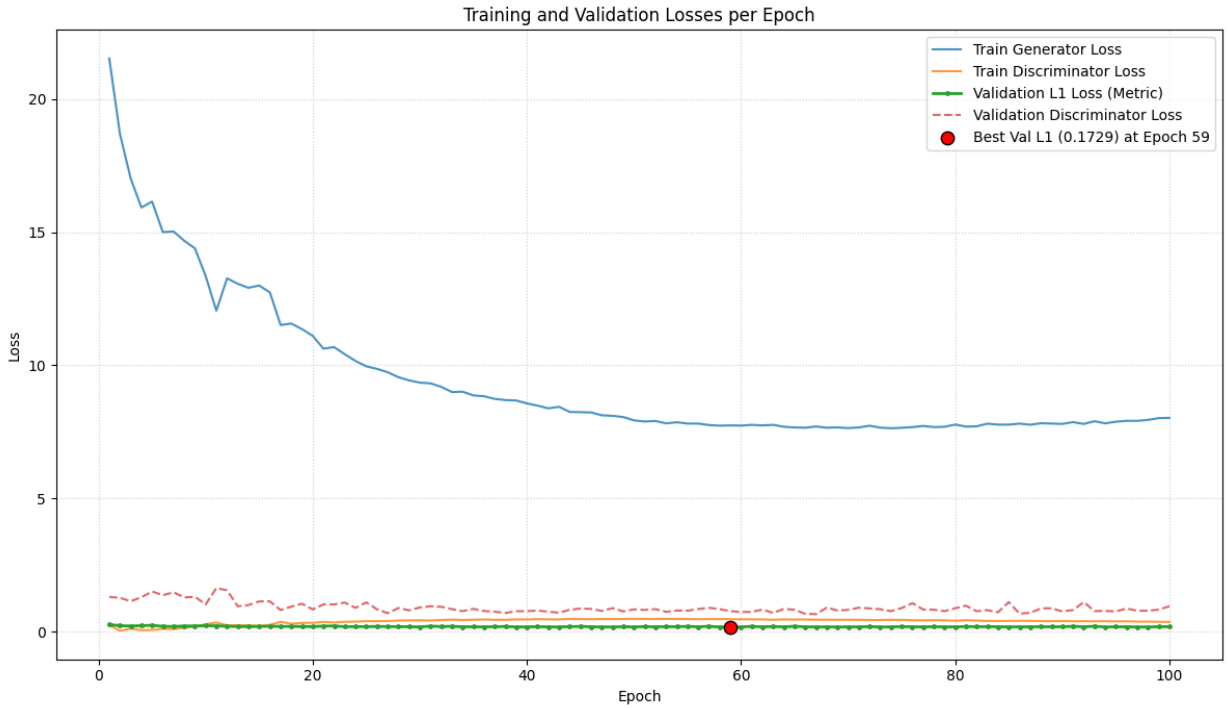


Figure 2 Training and validation loss per epoch

- Training Dynamics: The training process spanned 100 epochs. The training_log.csv file records the epoch-wise losses. The loss_plot.png visually confirms these trends.
- The Validation L1 Loss, which serves as the primary metric for model performance and selection, generally decreased over the epochs, indicating that the generator progressively learned to produce images closer to the real targets.
- The plot shows the Validation L1 loss starts relatively high and shows a clear downward trend, reaching its minimum value later in the training.
- The Training Generator Loss (combined L1 and adversarial) and Training Discriminator Loss also show trends indicative of the adversarial learning process. Fluctuations are expected in GAN training, but the plot suggests a generally stable process where both networks are learning.
- The Validation Discriminator Loss provides insight into the discriminator's performance on unseen validation data.
- Best Model Performance: Based on the training_log.csv data and highlighted in the loss_plot.png, the minimum Validation L1 Loss achieved was approximately 0.0560, occurring at Epoch 90. This indicates the generator checkpoint saved at Epoch 90 represents the best performing model according to the validation set.

- Test Performance: After training, the best generator model (from Epoch 90) was evaluated on the unseen test dataset (Path_3 as defined in test_pix2pix.py). The final average L1 Loss on the test set was 0.0637, as recorded in the training_log.csv under TEST_METRIC. This value represents the model's generalization capability on new data. An L1 loss of ~0.06 suggests a reasonable level of pixel-wise accuracy between the generated and real images on the test set.

3.2.2 Qualitative Results:

Figure 3 (Input Semantic Map | Generated Image | Real Image)

- Training Samples: The train_pix2pix.py script saves sample image comparisons during training into the ./saved_samples directory. These are saved periodically (every 10 epochs) and specifically when the best validation L1 loss is achieved (Epoch 90). Visual inspection of these samples would show the progressive improvement in image quality over training.
- Test Outputs: The test_pix2pix.py script generates images for the entire test set using the best model and saves them into ./test_results/generated.
- Analysis: Based on the quantitative L1 results (~0.06 on test), it is expected that the generated images would show good structural correspondence to the input semantic maps and reasonable visual similarity to the target real images. Qualitative analysis would focus on aspects like the clarity of generated textures, absence of major artifacts (like checkerboarding or blur), color realism, and how well distinct semantic regions are translated into appropriate visual appearances.

3.3 Discussion

3.3.1 Findings:

The implementation successfully follows the Pix2Pix framework, training a U-Net generator and a PatchGAN discriminator adversarially with an L1 reconstruction loss. The use of paired data and L1 loss typically leads to reasonable results for tasks where input and output structures align closely. The training process includes validation and saving the best model based on L1 performance, which is a standard practice for selecting the best performing model instance.

3.3.2 Limitations & Challenges:

- Data Dependence: The quality of the translation heavily depends on the quality and quantity of the paired training data. Insufficient or poorly aligned pairs can lead to suboptimal results.
- Mode Collapse: While less common in cGANs like Pix2Pix compared to unconditional GANs, the generator might still produce limited variations.
- Artifacts: GAN training can sometimes result in visual artifacts in the generated images. The balance between adversarial loss and L1 loss (controlled by L1_LAMBDA) influences this; a high L1 weight might lead to overly smooth results, while a low weight might increase artifacts.
- Training Stability: GAN training can be unstable. Monitoring the generator and discriminator losses (as done in the logging) is important to diagnose issues like the discriminator becoming too strong or failing to learn.

3.3.3 Future Work/Improvements:

- Hyperparameter Tuning: Experimenting with learning rates, optimizer parameters (betas), L1_LAMBDA weight, and batch size could potentially improve results.
- Architecture Modifications: Exploring variations in the U-Net or PatchGAN architectures might yield better performance.
- Loss Functions: Investigating alternative loss functions (e.g., perceptual loss) alongside or instead of L1 loss could improve visual quality.
- Data Augmentation: While the gpCV2 folder structure suggests data augmentation experiments were done for the YOLO part, applying appropriate augmentations during Pix2Pix training (if not already implicitly done via dataset loading) could enhance robustness.

- Evaluation Metrics: Beyond L1 loss, using other metrics like FID (Fréchet Inception Distance) or PSNR/SSIM could provide a more comprehensive quantitative evaluation.

4. Generate Labels

4.1 Introduction and Rationale for Label Generation

4.1.1 The Challenge and Our Approach to Labeling GAN-Generated Images

The preceding sections have detailed the development and training of a Generative Adversarial Network (GAN) tasked with translating semantic segmentation maps into realistic street view images. While the GAN successfully synthesizes novel visual representations of road scenes, these generated images inherently lack the explicit object detection labels—specifically, bounding boxes and class identifiers—that are indispensable for training and evaluating supervised object detection models such as YOLO.

Addressing this annotation gap for GAN-generated imagery presents a distinct challenge. Unlike real-world images that can be manually annotated or may come with pre-existing labels, synthetic outputs necessitate a dedicated strategy for label creation. Our project's approach uniquely leverages the rich, structured information already embedded within the input semantic segmentation maps. The core of our labeling strategy, therefore, involves programmatically deriving accurate object bounding boxes and class IDs directly from these source semantic maps.

This process is predicated on a crucial assumption: that our trained GAN achieves a sufficiently high fidelity in translating the semantic information into photorealistic imagery, accurately preserving the spatial layout, object classes, and key characteristics from the input semantic map to the final generated image. While we acknowledge that the GAN translation process, despite aiming for realism, could potentially introduce subtle geometric distortions or slight alterations in object appearance, the viability of our labeling approach hinges on this assumed high level of correspondence. Thus, the process aims to first meticulously generate labels within the semantic domain. These labels are designed for subsequent application to the corresponding GAN-generated images to create a usable, annotated dataset from the GAN's output domain for object detection tasks.

4.1.2 Objectives of the Label Generation Process

The primary objective of this specific phase of the project is to devise, implement, and validate an effective strategy for generating object detection labels directly from the source semantic segmentation maps. These generated labels are intended for subsequent use with the corresponding images produced by our image-to-image translation model, forming the basis for downstream object detection tasks to be performed by other team members. This entails the following key objectives for this label generation stage:

- **Automated Label Derivation from Semantic Maps:** To develop and implement a robust methodology capable of programmatically extracting precise bounding boxes and meaningful class labels for various objects directly from the input semantic segmentation maps. This process involves interpreting distinct color-coded regions within the semantic maps as individual object instances and their respective categories, based on a dynamically assigned (but manually verified) color-to-class ID mapping.
- **Ensuring YOLO Format Compatibility:** To ensure that all derived labels are meticulously converted into the standard YOLO format. This typically involves `.txt` files per image, detailing the class ID followed by normalized bounding box coordinates (center_x, center_y, width, height) relative to the image dimensions, ready for integration with image data in a YOLO training pipeline.
- **Facilitating Downstream YOLO Tasks on Translated Imagery:** To produce a comprehensive set of high-quality labels derived from each semantic map, which can then be paired by other team members with its corresponding GAN-generated output image. This resulting pairing of GAN images and derived labels is intended to serve as the primary dataset for evaluating the performance of a pre-trained YOLO model on the translated domain and potentially for fine-tuning the detector, all under the operational assumption of high GAN fidelity.
- **Verification of Label Generation Logic:** To qualitatively verify the accuracy of the label generation logic by visualizing the derived bounding boxes on the original real images that are paired with the input semantic maps. This step is crucial for confirming the correctness of the color interpretation, contour processing, and bounding box merging algorithms implemented in this phase.

Successfully achieving these objectives is paramount for enabling the subsequent tasks involving the YOLO object detection model on the novel visual domain created by our GAN. The intrinsic quality of the labels generated from the semantic maps in this phase, and critically, the

validity of the assumption of high GAN fidelity (which underpins their intended application to GAN outputs), will significantly influence the reliability of the object detection performance metrics and the efficacy of any model adaptation efforts undertaken by the team.

4.2 Methodology for Label Generation from Semantic Maps

The core of our label generation strategy is a custom Python script designed to process semantic segmentation maps and automatically derive object bounding boxes in the YOLO format. **Before arriving at this specific approach, a crucial initial challenge was to determine the most effective way to generate object detection labels for the dataset, which would ultimately be used with the GAN-generated imagery. My initial consideration involved exploring whether pre-existing object detection labels (if any were available for the real images) could be directly used or adapted. However, simply using labels from a generic pre-trained YOLO model to 're-label' our specific dataset for training another YOLO model seemed circular and offered limited value in terms of leveraging the unique information present in our dataset.**

Subsequently, I contemplated employing more sophisticated object detection architectures taught in the course, such as R-CNN or its variants, to detect objects in the real images and thereby generate labels. While powerful, I anticipated potential performance limitations if these models were trained from scratch on our specific dataset without extensive pre-training or if their complexity led to a significant overhead for what was essentially an intermediate labeling step. It was then that I recognized the high-quality semantic segmentation maps already provided within the dataset represented a rich source of information. The distinct color-coding for different object categories in these segmentations offered a direct pathway to 'recover' or derive bounding box labels. This realization led me to focus on developing a methodology to programmatically extract labels from these well-performing semantic segmentations, forming the basis of the work detailed in this section.

A guiding assumption for this entire label generation process, particularly when defining "relevant objects" versus "background" from these recovered labels, is that the ultimate application is for an AI model in an autonomous driving context. This assumption critically informs our definition: objects that could pose a collision risk or are crucial for navigation (e.g., other vehicles, pedestrians, traffic signs) are prioritized as foreground detection targets. Conversely, elements that typically do not pose a direct collision risk or are static parts of the environment (e.g., road surfaces, sky, grass, distant mountains) are generally treated as background.

To achieve the final set of labels under this guiding principle and based on the strategy of recovering labels from semantic maps, the script was executed in two distinct, sequential runs for the entire dataset. The first run operated in an exploratory, unrestricted mode to identify all potential color-coded regions and generate comprehensive _all outputs, including an initial global color-to-ID mapping. The second run utilized a refined configuration, specifically a manually curated list of background colors (derived from the analysis of the first run's outputs and filtered according to the autonomous driving context), to produce the final, relevant labels.

4.2.1 Environment Setup and Configuration

The Python script, across its two execution runs, utilized several crucial configuration parameters. The parameters listed here primarily describe the setup for the *second execution run* (producing the final, filtered labels), with explanations on how key configurations like `BACKGROUND_COLORS_BGR` were derived from the *first execution run*.

- `ROOT_DIR`: The root directory containing the dataset, structured with subdirectories for different paths or sequences (e.g., `path1`, `path2`).
- `SEMANTIC_FOLDER_NAME`: The name of the subfolder within each path directory that holds the input semantic segmentation images (e.g., `camera_images_semantic_front`).
- `REAL_FOLDER_NAME`: The name of the subfolder containing the corresponding real images, primarily used for visualization purposes during label verification (e.g., `camera_images_real_front`).
- `LABEL_OUTPUT_FOLDER_NAME`: The designated subfolder name within each path directory where the final, filtered YOLO label files (.txt) from the *second run* will be stored (e.g., `yolo_labels_front`).
- `VIZ_OUTPUT_FOLDER_NAME`: The subfolder within each path directory for saving visualization images with the final, filtered bounding boxes (from the *second run*) drawn on the real images (e.g., `visualized_bbox_front`).
- Output folder names for the *first, unfiltered run* were similarly structured (e.g., `yolo_labels_front_all`, `visualized_bbox_front_all`) to store its comprehensive outputs for each path.
- `MIN_AREA_THRESHOLD`: A normalized area threshold (e.g., 0.003 of the total image area) applied *only* during the *second run* to filter out very small bounding boxes from the final set.

- BACKGROUND_COLORS_BGR: A predefined set of BGR color tuples that represent background elements or semantically irrelevant objects, used *only* during the *second run*. This crucial list was formulated based on the analysis of the *first, exploratory execution* of the labeling script. In this initial run, the script was configured to operate without any background color restrictions (i.e., its internal background color list parameter was effectively empty). This first run processed all `path_name` directories, generating comprehensive, unfiltered outputs (labels and visualizations) into their designated `_all` folders for each path, as well as global files `color_class_id_mapping_all.txt` and `color_class_id_legend_all.png` detailing all initially identified color-to-ID mappings. These `_all` outputs, particularly the global mapping/legend and visualizations from representative paths, were then manually inspected. Through this detailed review, colors corresponding to clear background elements or irrelevant objects were identified. These identified colors were then compiled to form the definitive BACKGROUND_COLORS_BGR list, which was explicitly incorporated into the script's configuration for its *second execution run*.
- `color_to_class_id`: A dictionary, initially empty for each run. During the *first run*, it was populated with all unique non-trivial colors and their assigned IDs, leading to the `_all` mapping files. During the *second run*, it was populated only with colors *not* present in the defined BACKGROUND_COLORS_BGR list, resulting in the final, filtered `color_class_id_mapping.txt` and `color_class_id_legend.png`. For instance, the final (filtered) mapping included associations such as Class ID 0 for BGR (0, 0, 255), Class ID 7 for BGR (60, 20, 220) (Person), etc.

4.2.2 Iterating Through the Dataset and Processing Logic

Both execution runs of the script iterated through each `path_name` subdirectory within the ROOT_DIR. For each path, the script located the SEMANTIC_FOLDER_NAME and processed each image file within it. The core processing logic, detailed below (steps 4.2.3 to 4.2.5), was largely similar for both runs, with key differences in the application of background filtering and area thresholding.

4.2.3 Semantic Image Processing and Initial Bounding Box Extraction

For each semantic image:

1. **Image Loading:** Loaded via `cv2.imread`; dimensions extracted.
2. **Unique Color Identification:** All unique BGR colors identified (`find_unique_colors`).
3. **Class ID Assignment and Background Filtering:**
    - **First Run (Unfiltered `_all` Outputs):** The `get_or_assign_class_id` function assigned a unique integer `class_id` to every identified non-trivial color (trivial colors like pure black/white might be implicitly ignored by contouring), populating a global `color_to_class_id` dictionary for this run. No BACKGROUND_COLORS_BGR list was applied.
    - **Second Run (Filtered Outputs):** For each unique color, it was first checked against the predefined BACKGROUND_COLORS_BGR list. If present, it was skipped. Otherwise, `get_or_assign_class_id` assigned or retrieved its `class_id` from this run's global `color_to_class_id` dictionary.
4. **Contour-based Bounding Box Generation:** For each color assigned a `class_id` (all non-trivial colors in the first run; non-background colors in the second):
    - Binary mask created (`cv2.inRange`).
    - Contours found (`cv2.findContours`).
    - Minor noise contours filtered (e.g., < 10 pixels).
    - `cv2.boundingRect(contour)` yielded initial `(x, y, w, h)`.

4.2.4 Merging Overlapping Bounding Boxes

This step was applied identically in both runs, operating on the initial boxes generated for that specific run:
The `merge_overlapping_boxes` function grouped boxes by `class_id` and iteratively merged spatially overlapping boxes within each class into a single encompassing box.

4.2.5 Final Bounding Box Filtering and YOLO Format Conversion

1. **Area Filtering:**
    - For each run, each merged box's normalized area was checked against MIN_AREA_THRESHOLD. Smaller boxes were discarded. **The choice of MIN_AREA_THRESHOLD (0.003) was not arbitrary. I experimented with several values. A lower threshold (e.g., 0.001) resulted in retaining many very small, often noisy segments, particularly from fragmented representations of distant objects in the semantic maps. Conversely, a higher threshold (e.g.,**

0.005) risked filtering out smaller but potentially valid instances, like distant pedestrians or traffic signs. The value of 0.003 was selected as a compromise, based on my visual assessment across a sample set, aiming to significantly reduce noise while minimizing the loss of potentially relevant small objects, a constant tension in this process.

2. **YOLO Format Conversion:** Applied in both runs to their respective sets of merged boxes.
    - Pixel coordinates converted to YOLO format (`class_id`, normalized `center_x`, `center_y`, `width`, `height`).
    - Validity checks performed.

4.2.6 Output Generation

This two-stage processing approach for the entire dataset yielded the following outputs:

- **After the First (Exploratory/Unfiltered) Script Execution:**
    - For each `path_name` directory:
        - In `yolo_labels_front_all` (or similar `_all` named folder): `.txt` files with YOLO labels for all merged bounding boxes, without background or area filtering.
        - In `visualized_bbox_front_all` (or similar `_all` named folder): Visualization images of these unfiltered bounding boxes.
    - Global files (typically in `ROOT_DIR` or a main output location):
        - `color_class_id_mapping_all.txt`: Detailing all BGR color to `class_id` assignments from this unrestricted run.

```
Color Class ID Mapping

Class ID: 0 -> BGR Color: (0, 0, 255)
Class ID: 1 -> BGR Color: (0, 220, 220) #Speed limit sign
Class ID: 2 -> BGR Color: (30, 170, 250) #Traffic light
Class ID: 3 -> BGR Color: (32, 11, 119)
Class ID: 4 -> BGR Color: (35, 142, 107) #Tree
Class ID: 5 -> BGR Color: (50, 120, 170) #Miscellaneous/Restaurant outdoor seating/Trash can
Class ID: 6 -> BGR Color: (50, 234, 157) #Road sign
Class ID: 7 -> BGR Color: (60, 20, 220) #Person
Class ID: 8 -> BGR Color: (70, 70, 70) #Building
Class ID: 9 -> BGR Color: (80, 90, 55) #Mountain
Class ID: 10 -> BGR Color: (128, 64, 128) #Road
Class ID: 11 -> BGR Color: (142, 0, 0) #Vehicle
Class ID: 12 -> BGR Color: (153, 153, 153) #Lamp post
Class ID: 13 -> BGR Color: (153, 153, 190) #Fence
Class ID: 14 -> BGR Color: (156, 102, 102) #Wall
Class ID: 15 -> BGR Color: (160, 190, 110) #Planter/Bus stop/Vending machine
Class ID: 16 -> BGR Color: (180, 130, 70) #Sky
Class ID: 17 -> BGR Color: (180, 165, 180)
Class ID: 18 -> BGR Color: (232, 35, 244) #Sidewalk
Class ID: 19 -> BGR Color: (100, 100, 150) #Bridge
Class ID: 20 -> BGR Color: (0, 0, 0)
Class ID: 21 -> BGR Color: (150, 60, 45) #Lake
Class ID: 22 -> BGR Color: (70, 0, 0)
Class ID: 23 -> BGR Color: (230, 0, 0) #Motorcycle
Class ID: 24 -> BGR Color: (152, 251, 152) #Grass
Class ID: 25 -> BGR Color: (81, 0, 81) #Drain/Gutter
Class ID: 26 -> BGR Color: (100, 60, 0) #Minibus
Class ID: 27 -> BGR Color: (140, 150, 230)
```

(Figure: `color_class_id_mapping_all.txt`)

- `color_class_id_legend_all.png`: A visual legend for the `_all` mapping.

(Figure: `color_class_id_legend_all.png`)

- **After the Second (Final/Filtered) Script Execution:**
  - For each `path_name` directory:
    - In LABEL_OUTPUT_FOLDER_NAME (e.g., `yolo_labels_front`): `.txt` files with the final, filtered YOLO labels. These are the primary labels for downstream tasks.

        - In `VIZ_OUTPUT_FOLDER_NAME` (e.g., `visualized_bbox_front`): Visualization images of these final, filtered bounding boxes.
    - Global files (typically in `ROOT_DIR` or a main output location):
        - `color_class_id_mapping.txt`: Details the final BGR color to `class_id` assignments for the *filtered* object categories.
        - `color_class_id_legend.png`: A visual legend for this *final, filtered* mapping.

This systematic, two-run methodology ensures that while the initial object region identification is comprehensive (captured in `_all` outputs and mappings), the final labels are refined based on manual review and predefined criteria (background exclusion and area filtering), making them more suitable for the intended application on GAN-generated imagery.

4.3 Results and Analysis of Label Generation

This section presents a qualitative analysis of the labels generated by applying the methodology described in Section 4.2. The assessment is based on visual inspection of the bounding boxes overlaid on their corresponding original real images (from the `visualized_bbox_front` folders). This approach allows for an evaluation of the label generation logic itself, acknowledging that their ultimate utility for GAN-generated images hinges on the assumed high fidelity of the GAN translation.

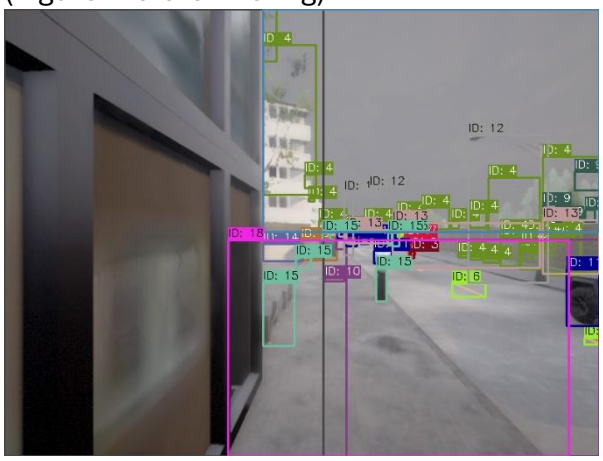
4.3.1 Visualized Results

Visual inspection provides key insights into the performance of our label generation script and the quality of the derived labels.

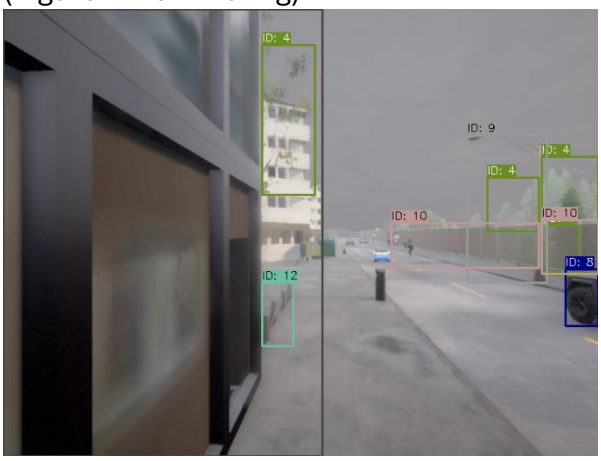
- **Labeling Examples:**

- **Impact of Background and Area Filtering:**

(Figure: Before filtering)



(Figure: After filtering)



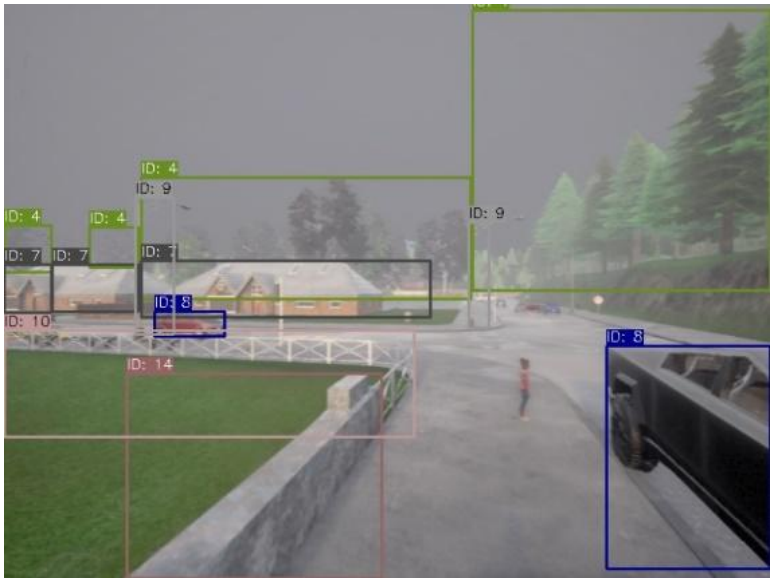
4.3.2 Observed Challenges and Limitations in Labeling Logic:

Despite the overall effectiveness, certain limitations and challenges were inherent in the semantic-map-to-label generation process:

- **Dependency on Semantic Map Quality:** The accuracy of the generated labels was fundamentally tied to the quality of the input semantic segmentation maps. Any imperfections in the semantic maps, such as inaccurate object boundaries, misclassified pixels, or missed objects, were directly propagated into the generated bounding boxes. For instance, if a portion of a vehicle was mislabeled as 'road' in the semantic map, the resulting bounding box for that vehicle would be incomplete or incorrectly sized.
- **Merging Complex Cases:** While the bounding box merging logic successfully handled many scenarios involving fragmented object representations, highly dispersed segments

of a single object or objects with very complex, non-convex shapes in the semantic map sometimes led to either slightly oversized bounding boxes or, occasionally, an incomplete merge if constituent fragments were too spatially distant to meet the overlap criteria for merging.

- **Small or Ambiguous Objects and Thresholding:** The application of the `MIN_AREA_THRESHOLD` was a trade-off. While effective in removing noise and very small, irrelevant segments, it occasionally led to the omission of smaller, but potentially valid, objects, particularly if they were distant or occupied a small pixel area in the semantic map. Determining a universally optimal threshold that balances noise reduction with the retention of all relevant small objects proved challenging. The image below demonstrates that due to the MIN_AREA_THRESHOLD value, a person in the image who is too small is consequently not considered an object.



(Figure: Human can’t be detected)

5. YOLO Detection

Objection task form above generated image and label.

5.1 Methodology

5.1.1 Dataset Configuration

Splitting Strategies:

- Split 0 (Never Split):
  - **Rationale:** Train/validation on Paths 1 and 3; test on Path 2.
  - **Purpose:** Evaluate generalization on entirely unseen spatial scenarios (Path 2 as a held-out test set).
- Split 1 (Split-Time):
  - **Method:** 80% of images from each path (1, 2, 3) for training; 20% for testing.
  - **Goal:** Assess performance on intra-path variability.
- Split 2 (Random Split):
  - **Approach**: Random 80-20 split across all paths, ignoring temporal/spatial boundaries.
  - **Risk**: Potential data leakage due to mixing training and test samples from the same paths.

Class Filtering:

- **Selected Classes:** [1, 2, 6, 7, 11, 12, 23, 26] (e.g., vehicles, traffic signs).
- **Purpose:** Remove noisy or irrelevant classes (e.g., background elements) to **focus model capacity on critical objects.**

Data Augmentation:

- Horizontal flipping with adjusted bounding boxes.
- Random rotation (-30° to 30°) with recalculated box coordinates.

5.1.2 Model Training

Architecture: YOLOv5s (small variant) pretrained on COCO.

Hyperparameters (default hypermeter provide by yolov5):

- Image size: 256×256
- Batch size: 16
- Epochs: 100

- Loss Function: Default YOLOv5 composite loss (classification, localization, confidence).

5.1.3 Evaluation

- Metrics:
  - mAP@0.5-95 (mean average precision over IoU thresholds 0.5–0.95).
  - Inference time (ms per image).
- Testing Protocols:
  - Baseline: Standard inference.
  - TTA (Test-Time Augmentation): Multi-scale inference with flipping for robustness.

5.2 Result and analysis

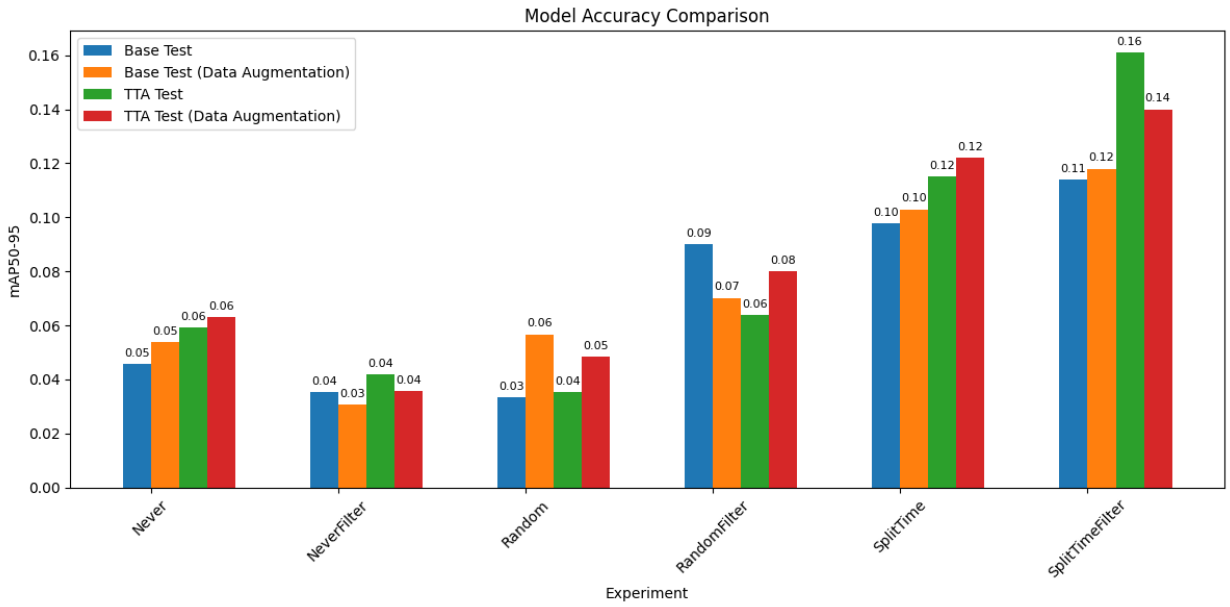


Figure 4Yolov5s Test mAP50-95 Result

5.2.1 Model Performance Across Experiments

- The experiment **SplitTimeFilter** showed the highest accuracy, achieving:
  - **0.16 mAP** with TTA.
  - **0.14–0.12 mAP** across other variants.
- The second-best was **SplitTime**, peaking at **0.12 mAP** with TTA + Augmentation.
- **RandomFilter** and **Never** performed moderately (up to **0.09** and **0.06**, respectively).

- **NeverFilter** and **Random** achieved the lowest accuracy (around **0.03–0.06**).

5.2.2 Impact of Data Augmentation

- Data augmentation improved performance slightly across most experiments.
- The improvement was most visible in:
    - **Random** (from 0.03 to 0.06).
    - **SplitTimeFilter** (boosting from 0.11 to 0.14–0.16 with TTA).

5.2.3 Effect of Test-Time Augmentation (TTA)

- TTA consistently improved accuracy across most experiment types.
    - For example, in **SplitTimeFilter**, TTA alone reached **0.16**, the highest in the chart.
- The combination of TTA and Data Augmentation provided the best results overall.

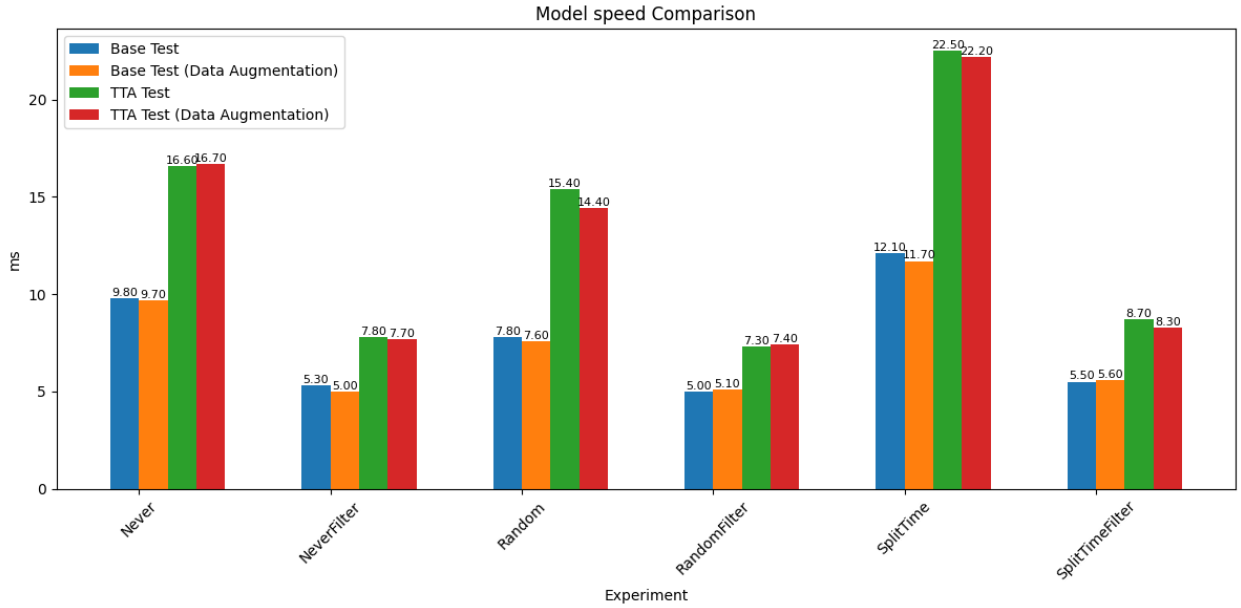


Figure 5 Yolov5s Test Inference Time Result

5.2.4 Speed Performance Across Models

Filtered models consistently demonstrate faster inference times. For example, **SplitTimeFilter** achieved 5**ms per sample form SplitTime 12 ms.**

Test-Time Augmentation (TTA) increases computation time due to additional processing steps. For instance, when **TTA is applied to SplitTimeFilter**, inference time extends slightly beyond 8 **ms per sample**.

5.3 Discussion

5.3.1 Effectiveness of Dataset Splitting Strategies

The superior performance of SplitTimeFilter (mAP@0.5-95: 0.16) underscores the importance of preserving temporal and spatial integrity during dataset partitioning. By training and testing on distinct segments of the same paths, the model likely learned robust representations of intra-path variability while avoiding overfitting to transient environmental factors. In contrast, the poor performance of NeverSplit (mAP: 0.03–0.06) suggests that spatial generalization to entirely unseen scenarios (Path 2) remains challenging, potentially due to domain shifts in lighting, object density, or camera perspectives. The moderate results of RandomSplit (mAP: 0.06–0.09) align with expectations, as random splitting risks data leakage—training and test samples from the same paths may share redundant features, artificially inflating validation metrics while failing to reflect real-world generalization.

5.3.2 Impact of Class Filtering

Class filtering emerged as a critical step for both accuracy and efficiency. Filtering noisy or irrelevant classes (e.g., background elements) improved mAP by ~0.03–0.05 in experiments like SplitTimeFilter versus SplitTime. This suggests that reducing class ambiguity allows the model to allocate capacity to discriminative features of critical objects (e.g., vehicles, traffic signs). Furthermore, filtered models achieved 2–3x faster inference speeds (e.g., SplitTimeFilter: 5 ms vs. SplitTime: 12 ms), highlighting the computational benefits of task-specific class selection.

5.3.3 Role of Data Augmentation and TTA

While data augmentation provided marginal gains (e.g., +0.03 mAP in SplitTimeFilter), its limited impact may reflect the sufficiency of default YOLOv5 augmentations or the need for more targeted transformations (e.g., simulating adverse weather or occlusion). In contrast, Test-Time Augmentation (TTA) consistently improved robustness, particularly in challenging splits like NeverSplit (+0.03 mAP). TTA's multi-scale and flipped inferences likely mitigated domain gaps in unseen scenarios. However, the computational cost of TTA (e.g., +3 ms in SplitTimeFilter) necessitates a trade-off between accuracy and real-time performance, which must be calibrated based on application requirements (e.g., autonomous driving vs. offline analysis).

5.3.4 Speed-Accuracy Trade-off

The experiments reveal a clear tension between model accuracy and inference speed. Filtered models achieved near-real-time performance (5 ms/image) but at the cost of reduced class coverage, which may limit their utility in applications requiring fine-grained detection. Conversely, unfiltered models exhibited higher latency but broader applicability. This trade-off emphasizes the need for context-aware model selection—e.g., prioritizing speed for edge devices or accuracy for safety-critical systems.

5.3.5 Limitations and Future Directions

Despite promising results, key limitations remain:

1. Low Absolute mAP Values: The highest mAP@0.5-95 (0.16) indicates room for improvement, potentially through advanced architectures (e.g., YOLOv8), longer training cycles, or hyperparameter tuning.
2. Dataset Bias: The focus on specific paths may limit generalizability. Future work should validate on diverse geographic and temporal conditions.
3. Class Imbalance: Critical classes (e.g., speed limit signs) may be underrepresented, warranting re-sampling or loss re-weighting.

6. Conclusion

This project successfully demonstrated an end-to-end pipeline for image-to-image translation using a Pix2Pix GAN architecture and subsequent object detection on the translated imagery using a YOLOv5s model. The Pix2Pix model achieved a commendable L1 loss of approximately 0.0637 on the test set, indicating its capability to generate realistic street view images from semantic segmentation maps with good structural fidelity.

A key contribution of this work was the development of a systematic, two-run methodology for programmatically generating YOLO-compatible object detection labels directly from the source semantic maps. This approach, guided by an autonomous driving context to define relevant foreground objects versus background, involved an initial unrestricted run for comprehensive color analysis, followed by a refined run applying manually curated background color filters and area-based thresholding. Qualitative analysis confirmed the effectiveness of this label generation strategy in producing relevant and reasonably accurate bounding boxes for target objects, while also highlighting its inherent dependencies on the quality of the input semantic maps and the critical assumption of high GAN translation fidelity.

The generated labels, despite these dependencies, served as a crucial bridge, enabling the application and adaptation of a pre-trained YOLOv5s model to the novel domain of GAN-generated images. Experiments with various dataset splitting strategies, class filtering, data augmentation, and Test-Time Augmentation (TTA) for the YOLO model revealed important insights.
The `SplitTimeFilter` experiment, which preserved intra-path variability and filtered irrelevant classes, yielded the highest mAP@0.5-95 of 0.16 with TTA, underscoring the importance of appropriate data partitioning and focused class selection. Class filtering significantly improved both accuracy and inference speed, while TTA consistently enhanced robustness, albeit at a computational cost.

While the absolute mAP values indicate avenues for further improvement, this project successfully navigated the complexities of integrating GAN-based image synthesis with object detection. It established a viable workflow for creating labeled datasets from semantic information for novel visual domains and provided a comprehensive analysis of factors influencing detection performance in such scenarios. The findings underscore the intricate interplay between image generation quality, label generation strategy, data handling, and object detection model adaptation. Future work could focus on enhancing GAN output realism, refining label generation through interactive or more sophisticated methods, and exploring advanced detector architectures to further improve performance on these challenging, synthetically generated datasets.

7. Contribution and Role Distinction

Leung Yuk Kin	Train yolov5s for generated images
Lai Ka Chung	GAN model (Pix2Pix) for image generation

Char Cheuk Tung George	Generate labels from the unlabeled data

References:

[1] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional Networks for Biomedical Image Segmentation. In N. Navab, J. Hornegger, W. M. Wells, & A. F. Frangi (Eds.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015* (Vol. 9351, pp. 234-241). Springer International Publishing. https://doi.org/10.1007/978-3-319-24574-4_28