Comparison Between CNNs and Transformers for the Occupant Detection Task in Vehicle Interior Monitoring Systems

Abdul Bari
Matrikelnummer: 00146949
Technische Hochschule Ingolstadt
Email: abb0233@thi.de

Declaration

I, Abdul Bari, hereby declare that the seminar report, represents my ideas in my own words and where ideas or work of others have been included, I have adequately and accurately cited and referenced the original sources.

I also declare that I have adhered to the ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not been previously formed the basis for the award of any degree, diploma or similar title of any other University.

Contents

Abstract

Interior monitoring systems play a crucial role in improving the safety and comfort of passengers in modern vehicles. One key task in this domain is **occupant detection**, which involves identifying and localizing individuals inside the vehicle cabin. With the rise of deep learning, **Convolutional Neural Networks (CNNs)** gained popularity for their efficiency and hierarchical feature learning [1] while **Transformer-based models** which were originally designed for natural language processing, have demonstrated strong global reasoning capabilities in vision tasks, leading to their rapid adoption in computer vision [2, 3, 4]. Despite their success, limited comparative studies exist evaluating these architectures specifically for **interior occupant detection**. This study addresses the gap by conducting a focused analysis by comparing a **CNN-based model (YOLOv5l)** [1] and a **Transformer-based model (Swin-T)** [4], both chosen for their similar parameter sizes and specialization in object detection tasks, allowing for a fair comparison using standard object detection metrics including **mean Average Precision (mAP)** and **generalization ability** across unseen occupants and lighting conditions. The models are fine-tuned and evaluated on the **SVIRO** [5] dataset, an open-source benchmark designed for realistic vehicle interior scenes. Through this comparative study, our aim is to investigate the practical trade-offs between **CNNs** and **Transformers** in the context of vehicle interior monitoring, providing insights into their strengths, limitations, and suitability for real-world applications.

1 Introduction

Computer vision has seen many advances since advances in deep learning [6], particularly with the introduction of convolutional neural networks and transformers, which have significantly improved the performance of computer vision models. Transformers, which were initially designed for NLP use cases, have recently shown remarkable skills in capturing long-range dependencies in images while traditional CNNs, with their ability to extract features in a hierarchical manner, have been the fundamental in many state-of-the-art image processing pipelines for many years now. Together, these architectures have taken computer vision tasks to an entirely new level.

One of the key computer vision tasks is object detection which identifies a single or multiple objects in an image or a video and localizes it by providing both the class labels and the bounding boxes for each detected object, unlike image classification that labels the entire image. This then allows systems to recognize not only what is in an image, but also locate its exact position. This ability is important for certain usecases that require a deeper understanding of the scene, allowing systems not just to recognize objects but also to figure out their exact locations. Such detailed visual information is important for various fields, including self-driving cars, robotics and security monitoring.

One of the main object detection applications is for vehicle interior monitoring systems, this can allow us to detect and track occupants as well as their activities inside the vehicle. Furthermore, this technique can support advanced driver-assistance systems (ADAS)

by ensuring driver attentiveness, preventing distracted driving, and enabling smart features like automatic climate control based on passenger locations allowing us to improve the user experience as well as the safety of the vehicle.

As vehicle interiors increase in complexity and expectations continue to increase, the need and requirement for reliable occupant detection models is more important than ever. Selecting the right architectures whether whether it is CNNs, which are great at extracting spatial features, or transformers, which can capture broader context, can really impact the system's performance. This research aims to compare these approaches specifically for occupant and object detection in vehicles, showcasing their different strengths and weaknesses in this emerging field.

2 Foundational Architectures

2.1 CNNs (Convolutional Neural Networks)

CNNs are deep learning models that specifically deal with grid-like data (i.e. visual data) [6]. First, why did we need them instead of traditional artificial neural networks (ANN)? Because, images are made up of pixels and inputting each pixel as an input data to an ANN is very resource intensive, moreover using an ANN is in this situation can lead to overfitting because inputting each an every pixel into the network can cause it to learn the image specific data instead of the general pattern in all the images, lastly spatial info is lost because we flatten down the entire image to input it into the ANN. Because of these reasons we use a CNN as it addresses all these issues.

CNNs consist of several key layers that extract key features from the images, of which the most important one is the convolutional layer which basically applies multiple filters that slide across the entire image applying a convolution operation on that patch. A convolution operation basically computes the dot products between the filter and the region the filter is placed on top of to produce a feature map, which highlights specific visual patterns such as edges, textures, or shapes.

Following the convolution layer we have an activation function that introduces non linearity in the model, this is a common thing to add throughout deep learning. The most common activation function used in a CNN is ReLU, the main reason for choosing this is to avoid vanishing gradients in deeper networks.

Next we have the pooling layer which helps shrink the spatial dimensions (height and width) of the feature maps. The most widely used method is max pooling, where usually a 2×2 window moves across the feature map, keeping only the highest value from each section. Pooling has several benefits: it decreases the computational load, by introducing a degree of spatial invariance (so the model is less sensitive to small translations or distortions in the image), and helps reduce overfitting by compressing the representation.

After going through multiple convolutional and pooling layers, CNNs typically finish up with one or more fully connected layers. At this point, the multi-dimensional feature maps get flattened into a single vector, which then flows through these dense layers that learn to blend the extracted features for the final prediction. In classification tasks, this process usually ends with a softmax layer, which outputs a probability distribution over the possible classes, allowing the model to make a decision [7].

2.2 CNNs to YOLO

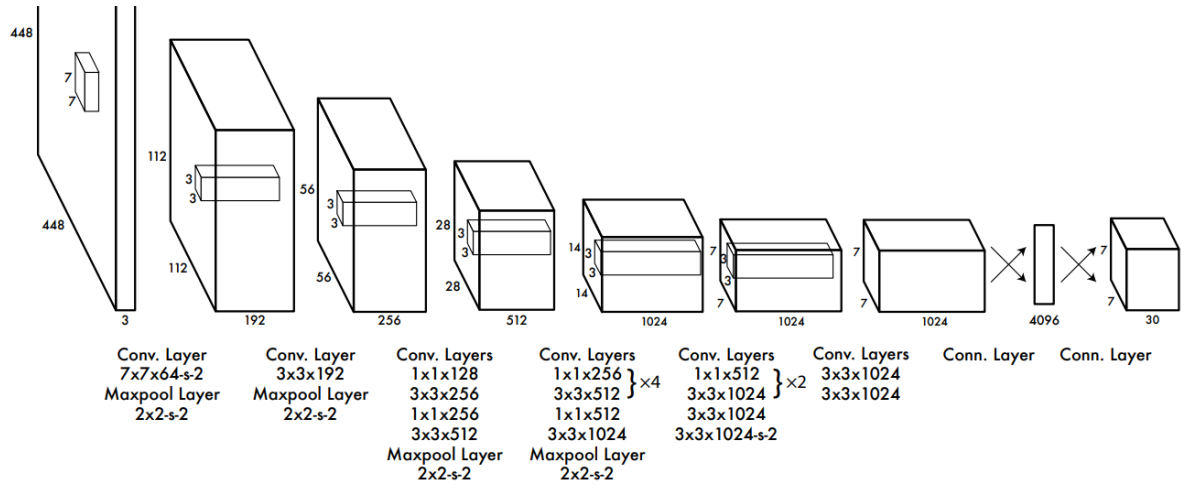

Figure 1: YOLO architecture [1]

Earlier methods of object detection used multi-stage pipelines [8] which involved separate models to predict the class labels and bounding boxes but YOLO combined them by treating object detection as a regression problem, directly predicting the bounding boxes and the class labels from an image in a single forward pass, hence the name You Only Look Once (YOLO). This resulted in significant speed improvements compared to previous methods.

The YOLO algorithm is foundationally based on CNNs. An image of fixed size is passed through multiple convolutional layers which extracts spatial features and high level patterns. These layers form the backbone of the entire model which are responsible for generating the feature maps. Instead of flattening these feature maps and using dense layers, which is commonly done in classification tasks, YOLO retains the spatial structure by applying additional convolutional layers that directly output detection results.

YOLO takes an input image and breaks it down into an S × S grid. Each cell in this grid is tasked with predicting a set number of bounding boxes in the format (x, y, w, h) where x and y are the coordinates of the center of the box and w and h are width and height relative to the image dimensions (see Figure 1). Each prediction also includes a confidence score which tells us how likely it is that the box contains an object and class probabilities for object classification [1].

2.3 Transformers

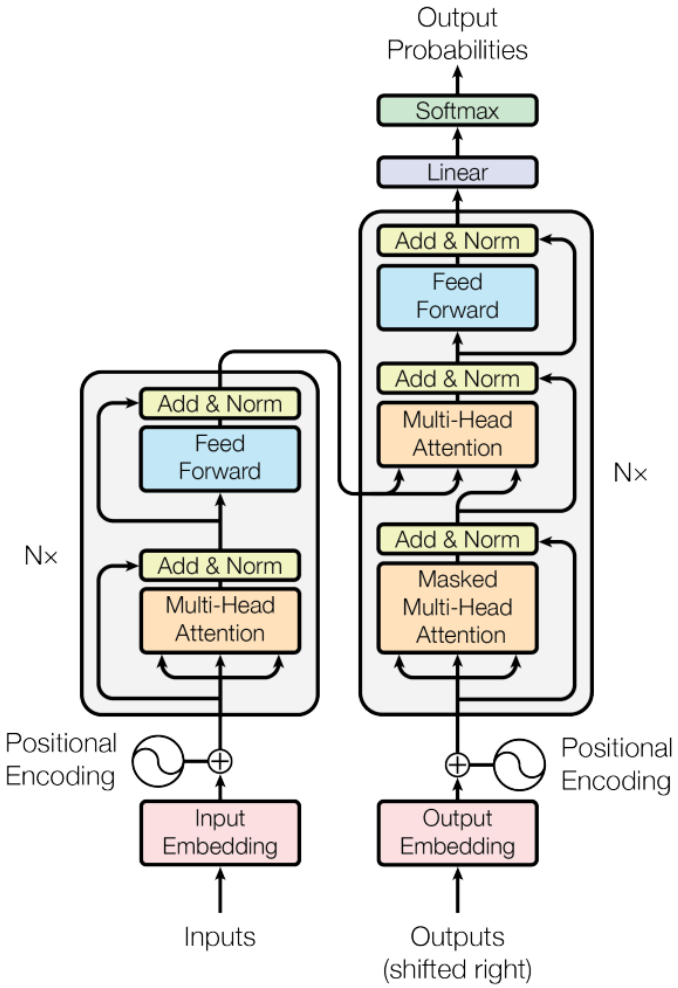

Figure 2: Transformer architecture [9]

The transformer model is a commonly used architecture in NLP applications because of its ability to learn sequential data much more efficiently than previous state of the art models such as LSTMs. Transformers basically entirely relies on the attention mechanisms particularly self attention [10], this then allows it create global dependencies with the added advantage of this being done parallelly allowing it to utilize modern GPUs thus significantly improving training speed and performance.

Transformers consist of two important parts, the encoder and the decoder with each of them composing multiple stacks of identical layers. The basic task of the encoder is to learn and produce rich contextual embeddings. However for the words to be inputted in the encoding layer, some key operations must be performed on them with the first one being performed by the embedding layer which transforms our input tokens into high dimensional vectors that preserve meaning of the token, moreover a positional encoding is added to each of the word embeddings which tell us about the relative position of the word in the sentence since transformers usually don't process data sequentially.

Processing in the encoder layer is done through mainly two layers, the multihead attention block which performs the most important operation in the entire architecture, self-attention, and the feedforward neural network. Residual connections and layer normalization is applied on the output of both of these layers (see Figure 2).

The fundamental component of the transformer architecture is the self attention mechanism [10], which transforms the previously static word embeddings into contextual embeddings. This basically tells us how much each token is related to each other token from the input sequence. This is done by deriving three further vectors from the input embedding, the query, key and the value vectors. These are derived through linear transformation which are learned from the training data. After deriving these vectors we use the following formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Here d_k is the dimension of the key vectors and the division by $\sqrt{d_k}$ is done to help stabilize the gradients and improve the training performance.

Instead of using one self attention block usually multiple blocks are used, this is done to capture multiple perspectives and patterns that might be a bit nuanced, from the training data. This is done by passing our original word embedding through multiple matrices that generate multiple query, key and value vectors. Hence the name of the first block in the encoder layer being a multi-head self attention block.

After getting the contextual embeddings for each of the tokens, we pass them through the feed forward network consisting of two layers and the activation function for each of them being ReLU which introduces the non linearity in the data allowing the model to learn more complex mappings.

Coming to the decoder part, it generates the output sequence one token at a time making use of both the encoder's output and its own generated tokens, which it does similarly to the encoder by first making word embeddings and adding a positional encoding to each token in the sequence. This is then passed through a masked multihead attention which does self attention but with one change, each token in the output sequence can only attend to only previous tokens but not the future ones, this is done by applying a mask that blocks access to tokens ahead in the sequence. This is done to prevent data leakage since we are preventing the model to look ahead during training which wouldn't be possible in actual text generation.

Following this we perform cross attention using the output from the masked attention layer and the encoder. Here we try to link the input sequence to the output being generated. This is done by generating the query vector from the output of the masked attention and key and value vectors come from the output of the encoder. We then perform attention using these query, key and value vectors and the result of this effectively allows the decoder to focus on relevant parts of the input sequence while generating each token, enabling strong contextual associations between the input and output sequences.

These new contextual embeddings are then input in a feed forward network just like in the encoder consisting of two layers, each having the ReLU activation allowing the model to learn the non linearity in data. Moreover, just like in the encoder each sublayer is wrapped in a residual connection followed by a layer normalization operation.

Finally, the output layer applies a linear transformation followed by a softmax to the decoder's final output, converting it into a probability distribution over the vocabulary (see Figure 2). This allows the model to predict the next token in the sequence during training or generation [9].

2.4 Vision Transformers (ViTs)

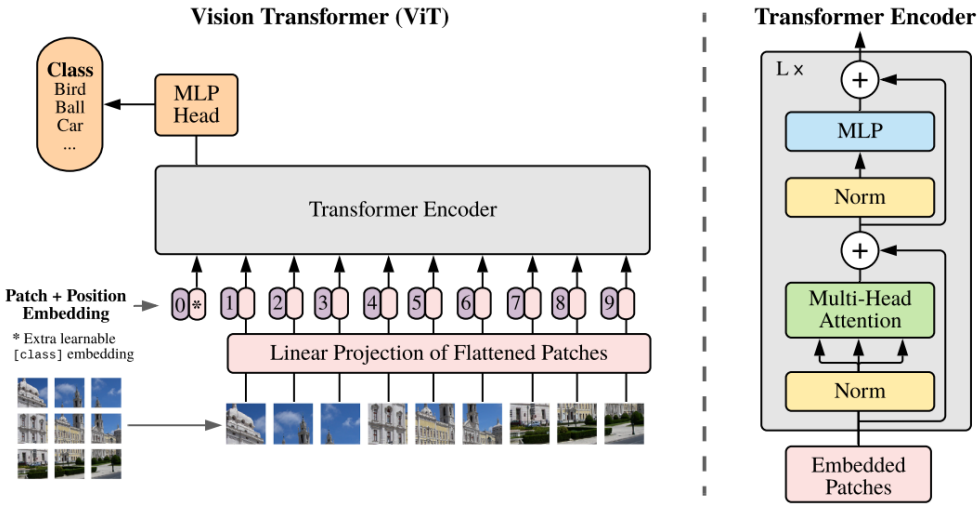

Figure 3: Vision Transformer architecture [3]

Motivated by the success of transformers in NLP tasks, researchers proposed the vision transformer [3, 11] which uses the self attention mechanism to extract feature maps from an image similar to that of CNNs. ViTs treat images in the same way Transformers treat text, by splitting them into a sequence of tokens and learning relationships among them without relying on convolutional operations (see Figure 3).

First we need to convert the input image into numbers for the model to understand it, similar to normal transformers that create word embeddings for each token, we need to create similar embeddings of our input image, this is done by first splitting our image into patches and flattening that patch into a 1D vector. The paper proposed an image of size 224×224 pixels to be divided into 16×16 patches, yielding 196 non-overlapping patches and the size of the flattened patch being $16 \times 16 \times 3 = 768$ [3]. Each flattened vector is then passed through a trainable linear projection layer to map it into a fixed-dimensional embedding space. Just like the original transformer model we then add a positional encoding to each vector to tell us about the relative position of the patch since transformers usually don't process data

sequentially. These patch embeddings with positional encoding serve as the input tokens to the Transformer model.

The Transformer encoder used in ViTs is similar to the one used in NLP models. It consists of multiple identical layers, each composed of two main sub-layers: a multi-head self-attention block and a feed-forward network. Layer normalization and residual connections are applied around both sub-layers to stabilize training (see Figure 3). The self-attention mechanism allows each image patch to attend to every other patch, capturing both local and global interactions in a single layer. Additionally, a special [CLS] token is prepended to the patch sequence, whose representation at the final encoder layer is intended to capture a global summary of the image. For classification tasks, this representation is passed through a fully connected head and a softmax layer to produce class probabilities.

Even though Vision Transformers have been quite successful [3], they do come with their own set of challenges. Unlike CNNs, they lack some built-in assumptions, such as the ability to focus on nearby pixels or effectively manage shifts in images (translation invariance). This can make them less efficient, especially when working with smaller datasets. Plus, ViTs require significantly more data and computing power to really shine. On top of that, the attention mechanism scales quadratically as the image size increases, which can pose issues for larger or more intricate tasks like object detection.

2.5 Architectures Adopted for This Study

2.5.1 YOLOv5l

YOLOv5l is one of the larger variants in the YOLOv5 family of object detection models, created and maintained by Ultralytics. Just like its siblings, YOLOv5l utilizes a fully convolutional architecture that allows it to detect objects in a single forward pass, which makes it incredibly fast and perfect for real-time applications.

It adheres to the standard YOLO workflow: an image is processed through a CSPDarknet53 backbone, based on Cross Stage Partial connections, which helps in reducing computations while maintaining a strong gradient flow resulting in an improved learning capability. This backbone is responsible for extracting high-level visual features from the input image.

Next is the PANet (Path Aggregation Network) neck, which is important for creating feature pyramids which allow the model to combine and refine features at different scales, improving its ability to detect objects of different sizes in the same image. This is especially useful in scenarios like vehicle interior monitoring, where both large (e.g., a human body) and small (e.g., hands, mobile devices) objects may need to be detected simultaneously.

Finally, we have a detection head that produces bounding box coordinates, objectness scores, and class probabilities. To refine these outputs, we apply a technique called non-maximum suppression (NMS). This helps us remove overlapping predictions and keep only the reliable ones.

We chose YOLOv5l for this study because it strikes balance between detection accuracy and inference speed. Its fully convolutional design enables real-time object detection, which is important for occupant monitoring systems that require a fast response. Morevoer, when compared to the smaller YOLO variants, YOLOv5l's deeper and wider architecture really enhances its ability to spot objects of different sizes and in various lighting conditions that are often encounter inside vehicles.

2.5.2 Swin Transformer

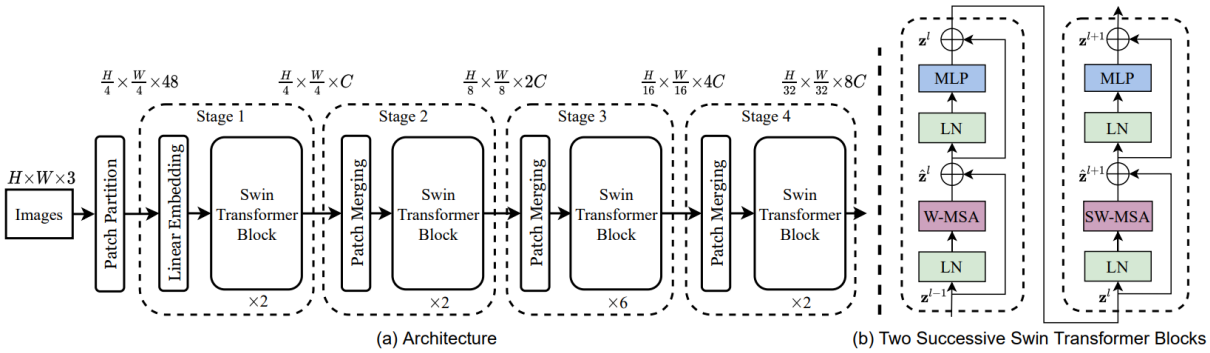

Figure 4: Swin Transformer architecture [4]

The Swin Transformer (Swin-T) addresses key drawbacks of the original Vision Transformer (ViT), such as its inability to focus on local features and high computational cost from global self-attention. ViTs perform well on classification tasks, but struggle with object detection tasks which require dense predictions, as discussed previously.

Swin-T introduces shifted window-based self-attention, which alternates across layers (see Figure 4). In the first layer of each stage, self-attention is computed within distinct, non-overlapping windows. The following layer then has the window partitioning shifted by a small offset (typically half the window size), allowing the model to connect with nearby windows. This strategy maintains the efficiency of local attention while enabling cross-window communication, effectively increasing the context without the cost of full global attention.

This architectural flexibility is further enhanced by a hierarchical design where feature maps are progressively downsampled using patch merging, similar to CNNs, making it more suitable for multi-scale vision tasks. This works well as a backbone when combined with RetinaNet or Mask R-CNN, resulting in rich, multi-scale features for accurate object detection.

The Swin Transformer, when paired with RetinaNet, was chosen for its ability to model both multi-scale and contextual information. Its hierarchical, shifted window self-attention mechanism allows for efficient extraction of global and local features, making it particularly effective at detecting partially occluded occupants or small objects in messy vehicle interiors. When combined with RetinaNet detection head, this setup provides accurate and dependable object detection even in complex situations, complementing the performance of YOLOv5l.

The following table summarizes the key differences, trade-offs, and characteristics of the two selected object detection models: YOLOv5l and Swin Transformer with RetinaNet.

Aspect	**YOLOv5l**	**Swin Transformer (with RetinaNet)**
Model Type	CNN-based single-stage detector	Transformer-based backbone with RetinaNet head (two-stage-like)
Backbone	CSPDarknet53	Swin-T (Shifted Window Transformer)
Head	YOLO detection head (anchor-based)	RetinaNet head (anchor-based, focal loss)
Feature Fusion	PANet (Path Aggregation Network)	FPN (Feature Pyramid Network)
Architecture Nature	Fully convolutional	Hierarchical transformer with local-global attention
Training Complexity	Easier and faster to train	More complex; higher memory and time requirements
Data Efficiency	Performs well on medium datasets	Requires more data for generalization
Augmentation Support	Strong built-in augmentation (Ultralytics)	Requires external pipeline (e.g., MMDetection)

Table 1: Comparison between YOLOv5l and Swin Transformer with RetinaNet

Model	**mAP@[.5, .95]**	**Params (M)**	**GFLOPs**	**Ref**
CNN (ResNet-101)	27.2	~ 44.5	7.6	[12]
ViT (DETR)	42.0	41.3	86	[2]
YOLOv3	33.0	61.9	65.9	[13]
YOLOv5l	49.0	46.5	109.1	[14]
Swin-T + Mask R-CNN	50.5	86	745	[4]

Table 2: Benchmark comparison of the discussed architectures on COCO dataset.

3 Experiment

The research focuses on two models: YOLOv5l and Swin Transformer paired with a RetinaNet head. The YOLOv5l model comes from the official Ultralytics GitHub repository. It has been pretrained on the COCO (Common Objects in Context) dataset [15], featuring over 200,000 labeled images across 80 different object categories. These pretrained weights enable the model to perform well on a variety of object detection tasks, especially when it's fine-tuned on a smaller, more specific dataset like SVIRO [5]. To set up the model, the yolov5 repository [14] was cloned, dependencies were installed via requirements.txt, and the training was initiated using a custom YAML config file specifying the dataset paths, class labels, and hyperparameters. The training was done using the provided train.py script.

On the other hand, the Swin Transformer, which was used alongside a RetinaNet detection head, was obtained from the MMDetection model zoo by OpenMMLab. This model has also been pretrained on the COCO dataset. Starting with strong feature representations from COCO allows the model to be fine-tuned effectively for tasks related to monitoring vehicle interiors, specifically tailored to the SVIRO dataset. To set up the Swin-T + RetinaNet model, we used the MMDetection toolbox from OpenMMLab. The process started with cloning the MMDetection repository [16] and installing all the necessary packages, including MMCV. We also had to tweak the existing Swin Transformer configuration file to match the details of the SVIRO dataset, such as the number of classes, the dataset format, and the file paths. Training was carried out using the tools/train.py script from MMDetection, and we evaluated the results with the built-in COCO evaluation tools that come with the framework.

3.1 Training Setup

3.1.1 Dataset

The models described above are fine-tuned on the SVIRO dataset [5] (specifically for the object detection use case using the bounding box labels), which is a synthetic dataset specifically designed for vehicle interior monitoring tasks. Our training dataset contained **1732** grayscale training images, and our test set contained **500** grayscale testing images and we use it specifically for occupant and object detection in the back seat of the vehicle.

The dataset contains the following classes as shown in Figure 5.

1 = Infant seat
2 = Child seat
3 = Person
4 = Everyday object

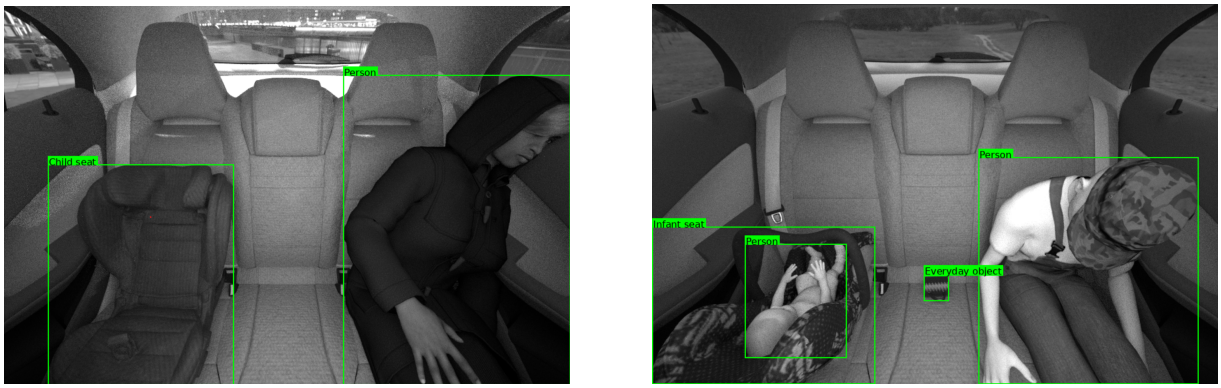

Figure 5: Example images from our dataset with their bounding box labels

Original Format

The bounding box labels in the original dataset are in the format:

`[class_id, x_min, y_min, x_max, y_max]`

where:

- `class_id`: Integer representing the class
- `x_min`: x-coordinate of the top-left corner of the bounding box
- `y_min`: y-coordinate of the top-left corner of the bounding box
- `x_max`: x-coordinate of the bottom-right corner of the bounding box
- `y_max`: y-coordinate of the bottom-right corner of the bounding box

YOLOv5l Format

For the YOLOv5l model, these labels were converted to the YOLO format:

`[class_id, x_center, y_center, width, height]`

where:

- `class_id`: Integer representing the class

- `x_center`: x-coordinate of the center of the bounding box, normalized by image width
- `y_center`: y-coordinate of the center of the bounding box, normalized by image height
- `width`: Width of the bounding box, normalized by image width
- `height`: Height of the bounding box, normalized by image height

Swin Transformer Format (COCO)

For the Swin Transformer model, labels were converted to the COCO format:

```
{
  "image_id": int,
  "category_id": int,
  "bbox": [x_min, y_min, width, height],
  "area": float,
  "iscrowd": 0,
  "id": int
}
```

where:

- `image_id`: Unique identifier for the image
- `category_id`: Integer ID representing the object class
- `bbox`: Bounding box in [x_min, y_min, width, height] format (absolute pixel values)
- `area`: Area of the bounding box (width $\times$ height)
- `iscrowd`: Indicates whether the object is part of a crowd (set to 0)

3.1.2 Model Hyperparameters

Hyperparameter	**YOLOv5l**	**Swin-T + RetinaNet**
Epochs	200	50
Batch Size	8	16
Image Size	640×640	448×448
Optimizer	SGD	SGD
Initial Learning Rate	0.01	0.005
LR Schedule	Linear decay	Cosine annealing
Warmup Strategy	3 epochs (momentum 0.8 to 0.937, warmup_bias_lr = 0.1)	500 iters to 10% of initial LR
Momentum	0.937	0.9
Weight Decay	0.0005	0.0001
Gradient Clipping	–	Max norm = 35
Cls Loss Type	BCE with logits	Focal Loss (γ=2.0, α=0.25)
Backbone	CSPDarknet	Swin-Tiny (96-dim, [2,2,6,2], win=7)
Drop Path Rate	–	0.2
Data Augmentations	HSV jitter, scaling, translation, flipping, mosaic	Resize, flip, affine (scale, rotate, shear), photometric distortions
Dataloader Workers	8	4

Table 3: Comparison of the training setups and architectural differences between YOLOv5l and Swin-T + RetinaNet. YOLOv5l emphasizes speed and simplicity using mosaic augmentation, CIoU and objectness loss, and a CNN-based CSPDarknet backbone. In contrast, Swin-T + RetinaNet focuses on semantic depth and contextual awareness with a transformer-based backbone, cosine annealing learning rate, and focal loss to address class imbalance.

3.2 Model Evaluation

Metric	YOLOv5l	Swin-T+RetinaNet
Precision	0.952	0.119
Recall	0.862	0.473
mAP@0.5	0.948	0.203
mAP@0.5:0.95	0.829	0.119
Training Epochs	56	50
Parameters (Millions)	46.5	52.8

Table 4: Performance comparison of YOLOv5l and Swin-T+RetinaNet models on test data

YOLOv5l

To assess how well the YOLOv5l model performed, we used a test set made up of 500 grayscale images of interiors. Even though according to Table 3 we set the training to run for 200 epochs, the model actually converged early and wrapped up at just 56 epochs, thanks to the early stopping rule we applied. When we evaluated the best model checkpoint on the test set, it delivered impressive results: an overall Precision (P) of 0.952, a Recall (R) of 0.862, and a mean Average Precision at IoU=0.5 (mAP@0.5) of 0.948, along with a mAP@0.5:0.95 of 0.829 (see Table 4) [17]. These results indicate that YOLOv5 performs reliably and efficiently for real-time object detection in grayscale interior environments.

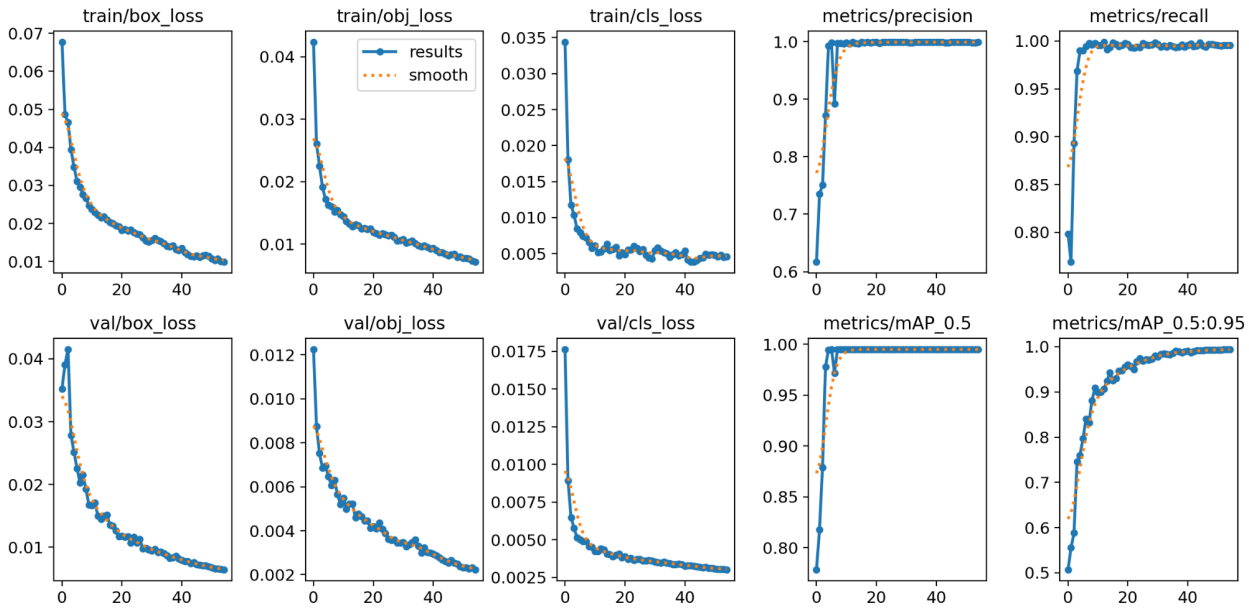

Figure 6: The results clearly show a consistent drop in all loss components, which points to stable training. At the same time, the precision, recall, and mAP metrics demonstrate a quick boost during the initial epochs before leveling off, indicating that the YOLOv5l model is effectively learning and converging.

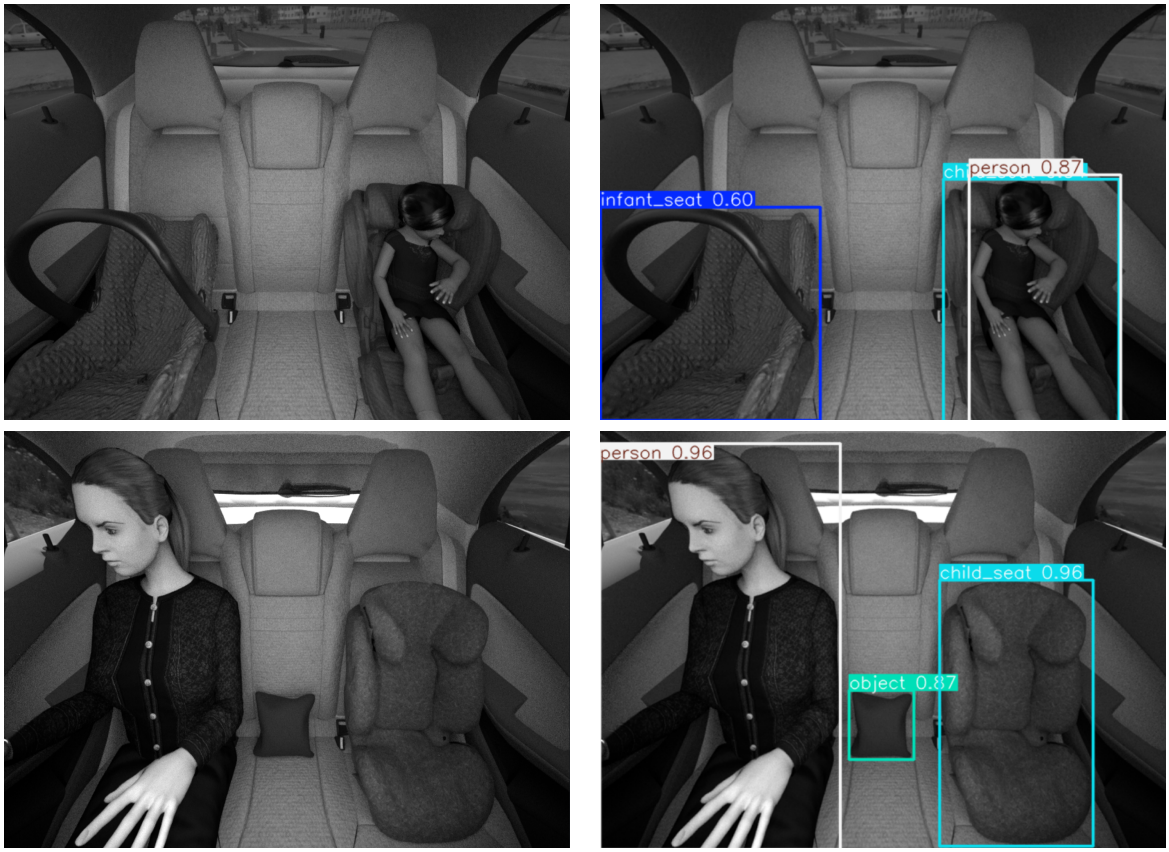

Figure 7: Detection results comparison: original images (left) with YOLO outputs (right)

Figure 7 shows the detection results from the YOLO model alongside the original input images. In the left column, we have the unannotated synthetic images of the interior, while the right column features the YOLO predictions, complete with bounding boxes and confidence scores for detected classes described above. These results really showcase the model's impressive ability to accurately locate and classify important interior elements and occupants.

Swin Transformer

The Swin Transformer model was trained for 50 epochs and evaluated on the same test set. It managed to achieve an Average Recall of 0.473, which means it successfully detected nearly half of the objects in the dataset. However, there was a noticeable precision-recall imbalance, leading to an overall mAP@0.5 of 0.203 and mAP@0.5:0.95 of 0.119. The model's performance was primarily focused on large objects, with a `mAP_large` of 0.124, and it completely missed detecting small or medium-sized objects. This aligns with the dataset's characteristics, where most of the target objects (infant seats, child seats, persons) are relatively large within the grayscale interior scenes. While the model shows a strong capability for detection, it still needs some optimization to reduce false positives before it can be considered ready for practical deployment.

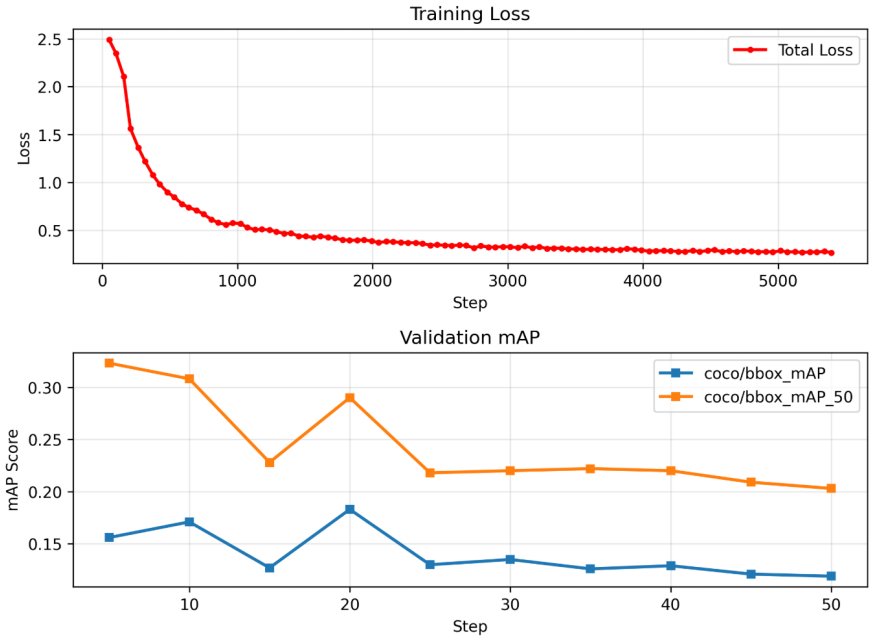

Figure 8: The results show a consistent drop in total loss, which shows that the model is converging nicely. On the other hand, the validation mAP scores are low and scattered. This implies that while the Swin-T model is good at cutting down training loss, but struggles to maintain steady detection performance when it comes to validation data.

Figure 9: Detection results comparison: original images (left) with Swin-T outputs (right)

Figure 9 shows the detection results from the Swin-T model alongside the original input images. In the left column, we have the same unannotated synthetic interior images, while the right column features the Swin-T predictions, complete with red bounding boxes and confidence scores. The model shows strong performance for person detection. However, it takes a broader approach to classification, labeling objects simply as "everyday object" instead of specifying seat types like the YOLO model does. Even with this difference, the Swin-T model consistently and accurately pinpoints larger elements within the vehicle interior scenes.

4 Conclusion

While the Swin Transformer has shown impressive results in various vision tasks, it didn't quite match the performance of YOLOv5l in this particular case. There are a few reasons that might explain this difference, such as the relatively small training dataset and the use of grayscale images, which can restrict the model's ability to fully utilize its architectural advantages. On the other hand, YOLOv5l, with its convolutional neural network (CNN) backbone and robust data augmentation techniques, demonstrated better generalization and more efficient inference. These findings indicate that for tasks where data is limited and input domains are specialized, tried-and-true CNN-based models like YOLOv5l may still have the edge over newer transformer-based approaches, highlighting their ongoing relevance in real-world applications.

A Configuration Files

A.1 YOLOv5l

```

    # YOLOv5 Custom Dataset Configuration
    path: ../yolo_dataset  # dataset root dir
    train: images/train/grayscale_wholeImage
    val: images/test/grayscale_wholeImage

    # Classes
    nc: 4
    names: ['infant_seat', 'child_seat', 'person', 'object']

    # Class weights
    class_weights: {1: 1.5202492211838006, 3: 2.585430463576159, 2:
       1.0, 0: 2.0699893955461293}
```

A.2 Swin Transformer

```
    auto_scale_lr = dict(base_batch_size=16, enable=False)
custom_imports = dict(
    allow_failed_imports=False,
    imports=[
        'mmdet.evaluation.metrics.coco_metric',
    ])
data_root = '/home/abdulbari/uni/seminar/swin_transformer/
   transformer_dataset/'
dataset_type = 'CocoDataset'
default_hooks = dict(
    checkpoint=dict(
        interval=5, max_keep_ckpts=3, save_best='auto', type='
           CheckpointHook'),
    logger=dict(interval=50, type='LoggerHook'),
    param_scheduler=dict(type='ParamSchedulerHook'),
    sampler_seed=dict(type='DistSamplerSeedHook'),
    timer=dict(type='IterTimerHook'),
    visualization=dict(type='DetVisualizationHook'))
default_scope = 'mmdet'
env_cfg = dict(
    cudnn_benchmark=False,
    dist_cfg=dict(backend='nccl'),
    mp_cfg=dict(mp_start_method='fork', opencv_num_threads=0))
launcher = 'none'
load_from = None
log_level = 'INFO'
```

```
log_processor = dict(by_epoch=True, type='LogProcessor', window_size
    =50)
metainfo = dict(
    classes=(
        'infant seat',
        'child seat',
        'person',
        'everyday object',
    ))
model = dict(
    backbone=dict(
        attn_drop_rate=0.0,
        convert_weights=True,
        depths=[
            2,
            2,
            6,
            2,
        ],
        drop_path_rate=0.2,
        drop_rate=0.0,
        embed_dims=96,
        init_cfg=dict(
            checkpoint=
            'https://github.com/SwinTransformer/storage/releases/
               download/v1.0.0/swin_tiny_patch4_window7_224.pth',
            type='Pretrained'),
        mlp_ratio=4,
        num_heads=[
            3,
            6,
            12,
            24,
        ],
        out_indices=(
            1,
            2,
            3,
        ),
        patch_norm=True,
        qk_scale=None,
        qkv_bias=True,
        type='SwinTransformer',
        window_size=7,
        with_cp=False),
    bbox_head=dict(
        anchor_generator=dict(
            octave_base_scale=4,
            ratios=[
```

```
                0.5,
                1.0,
                2.0,
            ],
            scales_per_octave=3,
            strides=[
                8,
                16,
                32,
                64,
                128,
            ],
            type='AnchorGenerator'),
        bbox_coder=dict(
            target_means=[
                0.0,
                0.0,
                0.0,
                0.0,
            ],
            target_stds=[
                1.0,
                1.0,
                1.0,
                1.0,
            ],
            type='DeltaXYWHBBoxCoder'),
        feat_channels=256,
        in_channels=256,
        loss_bbox=dict(loss_weight=2.0, type='L1Loss'),
        loss_cls=dict(
            alpha=0.25,
            gamma=2.0,
            loss_weight=1.0,
            type='FocalLoss',
            use_sigmoid=True),
        num_classes=4,
        stacked_convs=4,
        type='RetinaHead'),
    data_preprocessor=dict(
        bgr_to_rgb=True,
        mean=[
            123.675,
            116.28,
            103.53,
        ],
        pad_size_divisor=32,
        std=[
            58.395,
```

```
            57.12,
            57.375,
        ],
        type='DetDataPreprocessor'),
    neck=dict(
        add_extra_convs='on_input',
        in_channels=[
            192,
            384,
            768,
        ],
        num_outs=5,
        out_channels=256,
        start_level=0,
        type='FPN'),
    test_cfg=dict(
        max_per_img=100,
        min_bbox_size=0,
        nms=dict(iou_threshold=0.5, type='nms'),
        nms_pre=1000,
        score_thr=0.05),
    train_cfg=dict(
        allowed_border=-1,
        assigner=dict(
            ignore_iof_thr=-1,
            min_pos_iou=0,
            neg_iou_thr=0.4,
            pos_iou_thr=0.5,
            type='MaxIoUAssigner'),
        debug=False,
        pos_weight=-1,
        sampler=dict(type='PseudoSampler')),
    type='RetinaNet')
optim_wrapper = dict(
    clip_grad=dict(max_norm=35, norm_type=2),
    optimizer=dict(lr=0.005, momentum=0.9, type='SGD', weight_decay
       =0.0001),
    type='OptimWrapper')
param_scheduler = [
    dict(begin=0, by_epoch=False, end=500, start_factor=0.1, type='
       LinearLR'),
    dict(
        begin=0,
        by_epoch=True,
        end=50,
        eta_min=1e-06,
        type='CosineAnnealingLR'),
]
pretrained = 'https://github.com/SwinTransformer/storage/releases/
```

```
    download/v1.0.0/swin_tiny_patch4_window7_224.pth'
resume = False
test_cfg = dict(type='TestLoop')
test_dataloader = dict(
    batch_size=16,
    dataset=dict(
        ann_file='annotations/instances_test.json',
        data_prefix=dict(img='test/grayscale_wholeImage'),
        data_root=
        '/home/abdulbari/uni/seminar/swin_transformer/
           transformer_dataset/',
        metainfo=dict(
            classes=(
                'infant seat',
                'child seat',
                'person',
                'everyday object',
            )),
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(keep_ratio=True, scale=(
                448,
                448,
            ), type='Resize'),
            dict(type='LoadAnnotations', with_bbox=True),
            dict(size_divisor=32, type='Pad'),
            dict(
                meta_keys=(
                    'img_id',
                    'img_path',
                    'ori_shape',
                    'img_shape',
                    'scale_factor',
                ),
                type='PackDetInputs'),
        ],
        test_mode=True,
        type='CocoDataset'),
    drop_last=False,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(shuffle=False, type='DefaultSampler'))
test_evaluator = dict(
    ann_file=
    '/home/abdulbari/uni/seminar/swin_transformer/
       transformer_dataset/annotations/instances_test.json',
    format_only=False,
    metric='bbox',
    type='CocoMetric')
```

```
test_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(keep_ratio=True, scale=(
        448,
        448,
    ), type='Resize'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(size_divisor=32, type='Pad'),
    dict(
        meta_keys=(
            'img_id',
            'img_path',
            'ori_shape',
            'img_shape',
            'scale_factor',
        ),
        type='PackDetInputs'),
]
train_cfg = dict(max_epochs=50, type='EpochBasedTrainLoop',
    val_interval=5)
train_dataloader = dict(
    batch_sampler=dict(type='AspectRatioBatchSampler'),
    batch_size=16,
    dataset=dict(
        ann_file='annotations/instances_train.json',
        data_prefix=dict(img='train/grayscale_wholeImage'),
        data_root=
        '/home/abdulbari/uni/seminar/swin_transformer/
           transformer_dataset/',
        filter_cfg=dict(filter_empty_gt=True, min_size=32),
        metainfo=dict(
            classes=(
                'infant seat',
                'child seat',
                'person',
                'everyday object',
            )),
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(type='LoadAnnotations', with_bbox=True),
            dict(keep_ratio=True, scale=(
                448,
                448,
            ), type='Resize'),
            dict(prob=0.5, type='RandomFlip'),
            dict(
                border=(
                    -32,
                    -32,
```

```
                ),
                max_rotate_degree=5,
                max_shear_degree=0,
                scaling_ratio_range=(
                    0.8,
                    1.2,
                ),
                type='RandomAffine'),
            dict(
                brightness_delta=32,
                contrast_range=(
                    0.8,
                    1.2,
                ),
                hue_delta=10,
                saturation_range=(
                    0.8,
                    1.2,
                ),
                type='PhotoMetricDistortion'),
            dict(size_divisor=32, type='Pad'),
            dict(type='PackDetInputs'),
        ],
        type='CocoDataset'),
    num_workers=4,
    persistent_workers=True,
    sampler=dict(shuffle=True, type='DefaultSampler'))
train_pipeline = [
    dict(type='LoadImageFromFile'),
    dict(type='LoadAnnotations', with_bbox=True),
    dict(keep_ratio=True, scale=(
        448,
        448,
    ), type='Resize'),
    dict(prob=0.5, type='RandomFlip'),
    dict(
        border=(
            -32,
            -32,
        ),
        max_rotate_degree=5,
        max_shear_degree=0,
        scaling_ratio_range=(
            0.8,
            1.2,
        ),
        type='RandomAffine'),
    dict(
        brightness_delta=32,
```

```
        contrast_range=(
            0.8,
            1.2,
        ),
        hue_delta=10,
        saturation_range=(
            0.8,
            1.2,
        ),
        type='PhotoMetricDistortion'),
    dict(size_divisor=32, type='Pad'),
    dict(type='PackDetInputs'),
]
val_cfg = dict(type='ValLoop')
val_dataloader = dict(
    batch_size=16,
    dataset=dict(
        ann_file='annotations/instances_test.json',
        data_prefix=dict(img='test/grayscale_wholeImage'),
        data_root=
        '/home/abdulbari/uni/seminar/swin_transformer/
           transformer_dataset/',
        metainfo=dict(
            classes=(
                'infant seat',
                'child seat',
                'person',
                'everyday object',
            )),
        pipeline=[
            dict(type='LoadImageFromFile'),
            dict(keep_ratio=True, scale=(
                448,
                448,
            ), type='Resize'),
            dict(type='LoadAnnotations', with_bbox=True),
            dict(size_divisor=32, type='Pad'),
            dict(
                meta_keys=(
                    'img_id',
                    'img_path',
                    'ori_shape',
                    'img_shape',
                    'scale_factor',
                ),
                type='PackDetInputs'),
        ],
        test_mode=True,
        type='CocoDataset'),
```

```
    drop_last=False,
    num_workers=4,
    persistent_workers=True,
    sampler=dict(shuffle=False, type='DefaultSampler'))
val_evaluator = dict(
    ann_file=
    '/home/abdulbari/uni/seminar/swin_transformer/
       transformer_dataset/annotations/instances_test.json',
    format_only=False,
    metric='bbox',
    type='CocoMetric')
vis_backends = [
    dict(type='LocalVisBackend'),
]
visualizer = dict(
    name='visualizer',
    type='DetLocalVisualizer',
    vis_backends=[
        dict(type='LocalVisBackend'),
    ])
work_dir = 'work_dirs/final1'
```

B Inference Code

B.1 YOLOv5l

```
import os
import torch
from PIL import Image

model = torch.hub.load('ultralytics/yolov5', 'custom', path='yolov5/
    runs/train/exp11/weights/best.pt', force_reload=True)

image = os.path.join('yolo_dataset', 'images', 'test', '
    grayscale_wholeImage', 'aclass_test_imageID_489_GT_5_0_2.png')

result = model(image)
result.print()

Image.fromarray(result.render()[0]).show()
```

B.2 Swin Transformer

```
from mmdet.apis import init_detector, inference_detector
```

```
import mmcv
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.patches import Rectangle

config_file = 'mmdetection/configs/swin/retinanet_swin-t-p4-w7_fpn_1x_coco.py'
checkpoint_file = 'work_dirs/final1/epoch_50.pth'
device = 'cuda:0'

model = init_detector(config_file, checkpoint_file, device=device)

class_names = ['infant seat', 'child seat', 'person', 'everyday object']

img = 'transformer_dataset/test/grayscale_wholeImage/aclass_test_imageID_489_GT_5_0_2.png'

result = inference_detector(model, img)

# Visualize results
def visualize_detection(img_path, result, score_threshold=0.3):
    img = mmcv.imread(img_path)
    img = mmcv.bgr2rgb(img)

    plt.figure(figsize=(12, 8))
    plt.imshow(img)

    # Access predictions from the DetDataSample object
    pred_instances = result.pred_instances

    # Get bounding boxes, scores, and labels
    bboxes = pred_instances.bboxes.cpu().numpy()
    scores = pred_instances.scores.cpu().numpy()
    labels = pred_instances.labels.cpu().numpy()

    # Draw detections
    for i in range(len(bboxes)):
        if scores[i] >= score_threshold:
            bbox = bboxes[i]
            label = labels[i]
            score = scores[i]

            # Create a rectangle patch
            x1, y1, x2, y2 = bbox
            width = x2 - x1
            height = y2 - y1
            rect = Rectangle((x1, y1), width, height, linewidth=2,
                             edgecolor='r', facecolor='none')
```

```
            plt.gca().add_patch(rect)

            # Add label
            plt.text(x1, y1-5, f'{class_names[label]}: {score:.2f}',
                     color='red', fontsize=12, bbox=dict(facecolor='
                        white', alpha=0.7))

    plt.axis('off')
    plt.tight_layout()
    plt.show()

# Print detection results
def print_detections(result, score_threshold=0.3):
    # Access predictions
    pred_instances = result.pred_instances

    # Get bounding boxes, scores, and labels
    bboxes = pred_instances.bboxes.cpu().numpy()
    scores = pred_instances.scores.cpu().numpy()
    labels = pred_instances.labels.cpu().numpy()

    # Create a dictionary to group detections by class
    class_detections = {}
    for i in range(len(bboxes)):
        if scores[i] >= score_threshold:
            label = int(labels[i])
            if label not in class_detections:
                class_detections[label] = []
            class_detections[label].append((bboxes[i], scores[i]))

    # Print results for each class
    for class_id in sorted(class_detections.keys()):
        print(f"\nDetections for class '{class_names[class_id]}':")
        for i, (bbox, score) in enumerate(class_detections[class_id
            ]):
            x1, y1, x2, y2 = bbox
            print(f"  Detection {i+1}: confidence={score:.2f}, bbox
                =[{x1:.1f}, {y1:.1f}, {x2:.1f}, {y2:.1f}]")

# Call the visualization function
visualize_detection(img, result, score_threshold=0.6)

# Print detection details
print_detections(result, score_threshold=0.6)
```

References

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 779–788, 2016.

[2] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, "End-to-end object detection with transformers," in *European conference on computer vision*, pp. 213–229, Springer, 2020.

[3] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[4] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 10012–10022, 2021.

[5] S. D. D. Cruz, O. Wasenmuller, H.-P. Beise, T. Stifter, and D. Stricker, "Sviro: Synthetic vehicle interior rear seat occupancy dataset and benchmark," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pp. 973–982, 2020.

[6] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, vol. 25, 2012.

[8] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, vol. 28, 2015.

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, Curran Associates, Inc., 2017.

[10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014.

[11] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, *et al.*, "A survey on vision transformer," *IEEE transactions on pattern analysis and machine intelligence*, vol. 45, no. 1, pp. 87–110, 2022.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[13] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[14] G. Jocher *et al.*, "Yolov5 - object detection models." `https://github.com/ultralytics/yolov5`, 2020.

[15] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*, pp. 740–755, Springer, 2014.

[16] M. Contributors, "Mmdetection: Openmmlab detection toolbox and benchmark." `https://github.com/open-mmlab/mmdetection`, 2018.

[17] R. Padilla, S. L. Netto, and E. A. Da Silva, "A survey on performance metrics for object-detection algorithms," pp. 237–242, 2020.

[18] C. Shorten and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," vol. 6, pp. 1–48, SpringerOpen, 2019.