UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING,
ELECTRICAL ENGINEERING AND APPLIED
MATHEMATICS

Project Report

Artificial Vision

Artificial Vision Contest 2025
People Detection, Tracking, Classification and Behavior Analysis

prof. Vento Mario
prof. Greco Antonio

Group 06

Students	Id number
Alberti Andrea	0622702370
Attianese Carmine	0622702355
Capaldo Vincenzo	0622702347
Esposito Paolo	0622702292

Academic Year 2024 – 2025

CONTENTS

ABSTRACT

The project is part of the *Artificial Vision Contest 2025*, aimed at developing an advanced system for automatic video analysis. The problem addressed requires the implementation of software capable of *detecting and tracking in real-time all people present* in a given input video. Furthermore, the system must be able to *recognize specific attributes of pedestrians*, including gender and the presence of bags and hats. In parallel, the software must *analyze pedestrian behavior* by monitoring the number of crossings over virtual lines within the scene.

The proposed solution must also address various technical challenges, including managing environmental variations such as lighting, scale and perspective, as well as operating in complex scenarios characterized by occlusions and interactions among people. Another fundamental requirement is to ensure high precision in both detection and attribute classification, minimizing identification and tracking errors.

The system, designed to process videos recorded in real-time, will present the data in an interactive graphical user interface and will produce a results file structured according to a standardized format.

CHAPTER 1

SYSTEM ARCHITECTURE

To address the contest problem, the solution is divided into different modules. Figure 1.1 illustrates the overall system architecture.

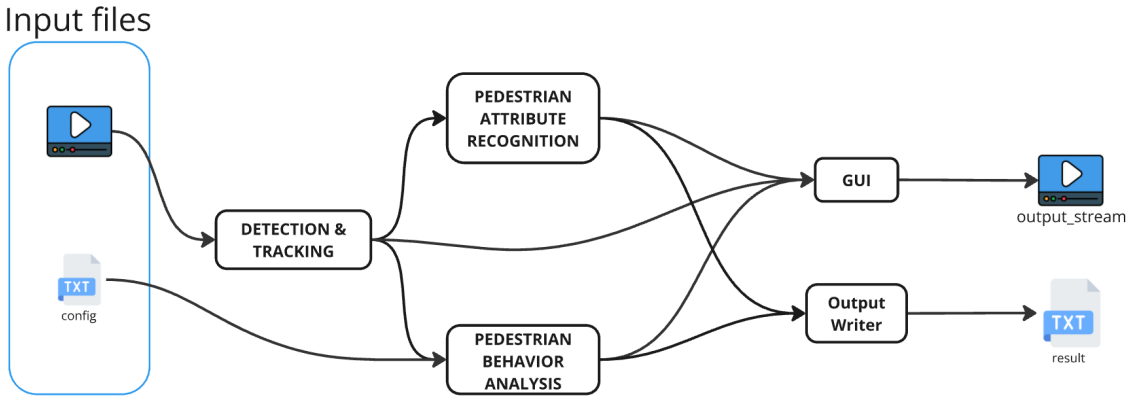


Figure 1.1: System architecture overview.

Below is a description of the modules that compose the architecture:

- **Input Files**: these are all the files input into the system. The file *config.txt* contains all the camera parameters and the positions of the virtual lines, necessary for pedestrian behavior analysis. For example:

```
{
    "thyaw": 0,
    "throll": 0,
    "thpitch": -0.558,
    "xc": 0,
    "yc": 0,
    "zc": 7.20,
    "f": 0.003,
    "sw": 0.00498,
    "sh": 0.00374,
    "U": 1280,
    "V": 720,
    "lines": [
        {
            "id": 1,
            "x1": 0.5,
```

```
            "y1": 13,
            "x2": -2.5,
            "y2": 13.41
        },
        {
            "id": 2,
            "x1": 0.5,
            "y1": 8,
            "x2": 4.6,
            "y2": 10.91
        }
    ]
}
```


where:

- *thyaw*: yaw angle of the camera, in radians.
- *throll*: roll angle of the camera, in radians.
- *thpitch*: pitch angle of the camera, in radians.
- *xc*: X-coordinate (width) of the camera position in meters.
- *yc*: Y-coordinate (depth) of the camera position in meters.
- *zc*: Z-coordinate (height) of the camera in meters.
- *sh (sensor height)*: camera sensor height in meters.
- *sw (sensor width)*: camera sensor width in meters.
- *f (focal length)*: distance between lens and sensor in meters.
- *U (image width)*: image resolution in pixels (horizontal).
- *V (image height)*: image resolution in pixels (vertical).

With this information, it is possible to precisely map the points in the real world to the pixels of the image captured by the camera. Each line is identified by an ID and its definition requires two coordinates: the start point $(x1, y1)$ and the end point $(x2, y2)$, that are positioned on the $z = 0$ plane. The second input file is the video to be analyzed, which must be recorded with the same properties defined in the configuration file.

- **Detection & Tracking**: this module processes the video file to detect and track people in the scene. The output consists of bounding boxes for the detected people and associated IDs to maintain tracking.
- **Pedestrian Attribute Recognition**: this module analyzes people detected by the Detection & Tracking module. It takes the scene portions defined by bounding boxes for each ID and classifies the attributes required by the contest, specifically: gender recognition and the presence of bag and hat.
- **Pedestrian Behavior Analysis**: this module is responsible for analyzing the behavior of people within the scene, specifically observing when a person crosses a virtual line. In order to display the virtual lines in the scene, is necessary to convert the line's real world coordinates to image coordinates (mapping $3D/2D$).
- **GUI**: this module overlays necessary information on each frame, including virtual lines, general information about the scene (number of people detected and the number of times a line has been crossed), bounding boxes around detected people with their ids and attributes. All this information is drawn frame by frame and displayed in real-time (represented as *output_stream* in Figure 1.1).
- **Output Writer**: this module generates the output file containing information for each person (represented as *result* in figure 1.1). An example of the output format is as follows:

```
{"people": [
    {
        "id": 1,
        "gender": "male",
        "hat": true,
        "bag": false,
        "trajectory": [1,2,3,4]
    },
    {
        "id": 2,
        "gender": "female",
        "hat": false,
        "bag": true,
        "trajectory": [4,1,3,1,2]
    }
]}
```

where:

- *id* represents the unique identifier assigned by the tracker;
- *gender, hat, bag* are the attributes recognized by the classifier for that person;
- *trajectory* is a list of the lines crossed by the person during their presence in the scene.

CHAPTER 2

DETECTION & TRACKING

The foundation of the system is built on the integration of detection and tracking. Detection is used to locate individuals within the scene, while tracking ensures the consistent assignment of identities to each person across the video frames.

2.1 Detection

The detection system used in this project is *YOLO11*, the latest evolution in object detection technology developed by *Ultralytics* [4]. The architecture is shown in the figure 2.1.

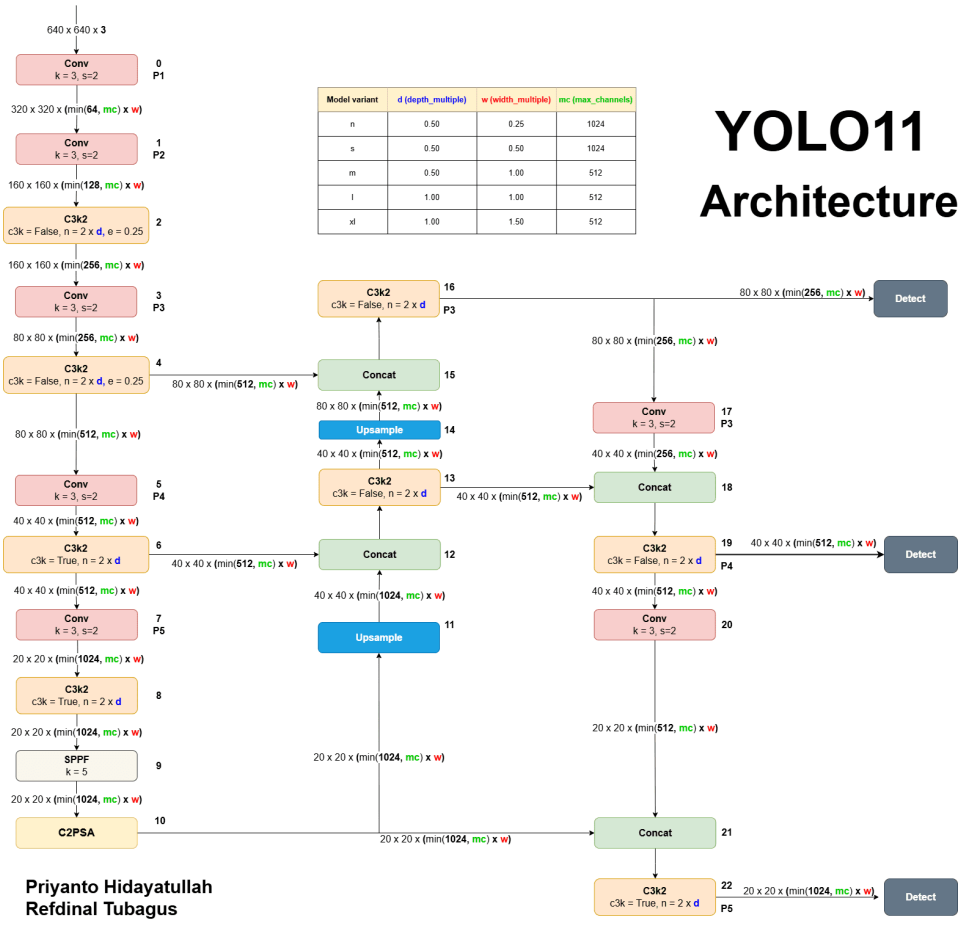


Figure 2.1: YOLO11 architecture

The main advantages of *YOLO11* are: its ability to detect subtle details even in challenging conditions, faster inference times and high adaptability to various deployment platforms. Moreover, its refined architectures, both in the backbone and neck, enhance detection accuracy across a wide range of computer vision tasks, from object recognition to instance segmentation.

In particular, the *YOLO11m* model was chosen for its balance between high precision and computational efficiency, ensuring accurate detection in complex scenarios. Smaller models in the *YOLO11* family, although faster in terms of inference time, lacked the required precision to consistently detect multiple objects in crowded environments. Conversely, larger versions provided only slight performance improvements but at the cost of significantly increased inference time, which was incompatible with the project's real-time requirements. *YOLO11m* addresses these challenges by combining advanced feature extraction capabilities with reduced parameter complexity, ensuring precise detection without requiring excessive computational resources.

Compared to *YOLOv8m*, *YOLO11m* achieves higher mean Average Precision (mAP) on the COCO dataset while using 22% fewer parameters, demonstrating a significant improvement in efficiency and performance (as shown in Figure 2.2)[1]. The choice of *YOLO11m* allows the project to leverage a model that not only provides state-of-the-art detection capabilities but also meets the system's requirements for real-time performance and scalability.

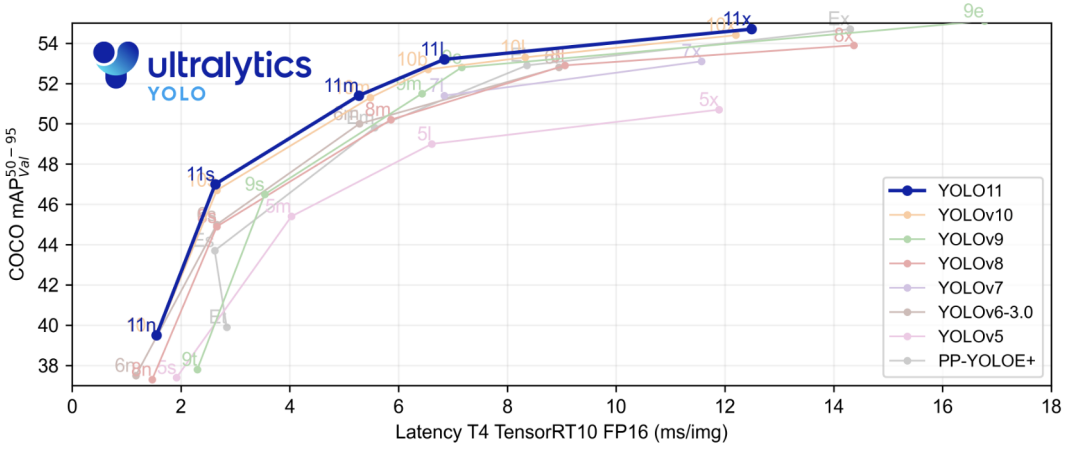


Figure 2.2: Benchmarking *YOLO11* against Previous Versions

2.2 Tracking

Ultralytics YOLO supports the following tracking algorithms: *BoT-SORT* and *ByteTrack*. These trackers can be enabled by passing the corresponding YAML configuration file during execution. The trackers use the detections generated by the YOLO model to associate objects across consecutive frames, implementing advanced strategies to handle complex scenarios. In the context of *Pedestrian Attribute Recognition (PAR)*, **BoT-SORT** was chosen over ByteTrack for its robustness in occlusions and tracking, accuracy in crowded scenes and computational efficiency.

In figure 2.3 are compared the performances of different state-of-the-art trackers on the **MOT17** and **MOT20** datasets. The axes represent two key metrics:

- **IDF1** (x-axis), which measures ID consistency:

$$\text{IDF1} = \frac{2 \cdot \text{IDTP}}{2 \cdot \text{IDTP} + \text{IDFP} + \text{IDFN}}$$

- **MOTA** (y-axis), which evaluates overall tracking accuracy:

$$\text{MOTA} = 1 - \frac{\sum_t (\text{FN}_t + \text{FP}_t + \text{IDSW}_t)}{\sum_t \text{GT}_t},$$

The size of the circles reflects the **HOTA** metric, a balanced combination of accuracy and association:

$$\text{HOTA} = \sqrt{\text{DetA} \cdot \text{AssA}}$$

where:

- IDTP: True Positives, IDFP: False Positives, IDFN: False Negatives.
- FN_t: False Negatives, FP_t: False Positives, IDSW_t: ID switches, GT_t: Ground Truth.
- DetA: Detection Accuracy, AssA: Association Accuracy.

BoT-SORT and *BoT-SORT-ReID* emerge as the top-performing trackers, with high performance across all metrics[5].

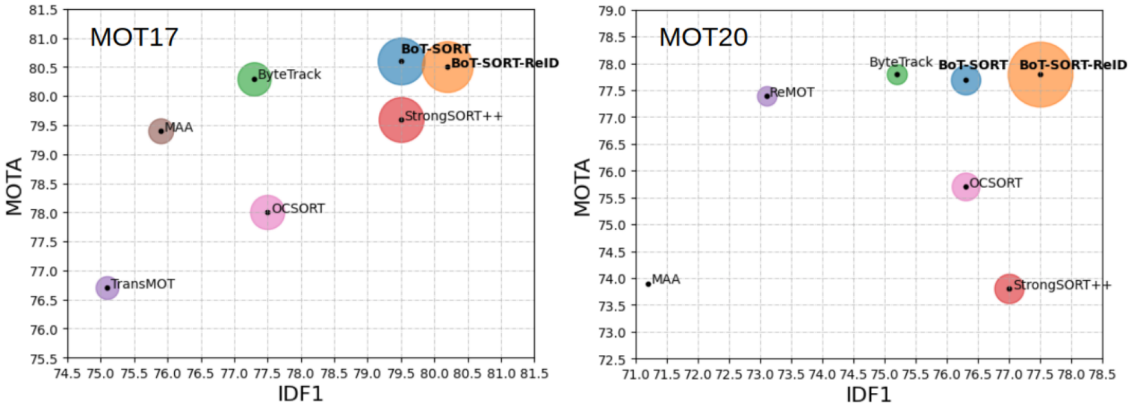


Figure 2.3: IDF1-MOTA-HOTA comparisons of state-of-the-art trackers

Finally, testing *BoT-SORT* in the real environment of the project (*ATRIO CUES*) revealed that *BoT-SORT* significantly minimized ID-switches compared to *ByteTrack*. This behavior proved particularly advantageous in scenarios with frequent occlusions or complex movements.

After several tests conducted within the project environment, the version of BOT-SORT without Re-ID was selected. Despite the generally superior performance of the variant with Re-ID, this decision was driven by specific project requirements and constraints. BOT-SORT without Re-ID offers a faster and lighter solution, as it does not require the integration of an additional model for re-identification. This feature significantly reduces computational load and simplifies the implementation within the operational pipeline. Furthermore, the project did not require the ability to recognize subjects exiting and re-entering the scene. Despite the presence of occlusions, BOT-SORT proved to be sufficiently effective in managing tracking, providing an optimal balance between simplicity, efficiency, and performance, without introducing unnecessary complexity.

2.2.1 Configurable Parameters in `botsort.yaml`

The `botsort.yaml` file allows customization of the tracker's behavior to adapt it to specific needs. Below are the main configurable parameters:

- *tracker_type*: botsort → tracker type, ['botsort', 'bytetrack']
- *track_high_thresh*: 0.2 → threshold for the first association (detections with a confidence higher than this value are considered for direct tracking).
- *track_low_thresh*: 0.1 → threshold for the second association (helps to keep track when detection is less certain).
- *new_track_thresh*: 0.7 → threshold for init new track if the detection does not match any tracks.
- *track_buffer*: 40 → buffer to calculate the time when to remove tracks, useful to deal with occlusions. It is a reasonable number of frames ($4 \cdot frame\_rate$) that can be used to address issues with occlusions and people entering or leaving the scene.

- *match_thresh*: 0.9 → threshold for matching tracks (the higher the value, the stricter the criterion for considering two objects as matching).
- *fuse_score*: True → whether to fuse detector confidence scores with IoU distances before matching (if set to True, the detection confidence influences the association).

These parameters were set by observing the performance of the detector and tracker on example videos recorded in the environment where the contest will take place.

2.2.2 Tracker Limitations

BoT-SORT represents an optimal compromise between efficiency and precision; however, some limitations persist. In crowded areas, cases of *identity switch* can occur, and individuals who are either too distant or static might not be consistently tracked. More advanced solutions could address these limitations, such as employing more efficient detectors (e.g., the large or extra-large versions of *YOLO11*), but this would come at the cost of increased computational time, making such approaches less suitable for the real-time requirements of the project.

Additionally, the integration of the Re-ID version of *BoT-SORT* could further improve tracking robustness by reducing identity switches and enhancing the ability to re-identify individuals during occlusions or exiting and re-entering the scene. However, this improvement would come at the expense of higher computational complexity and longer processing times, potentially compromising the real-time performance critical to the project's objectives.

CHAPTER 3

PEDESTRIAN ATTRIBUTE RECOGNITION

Pedestrian Attribute Recognition is a problem of significant importance in the field of computer vision. The goal is to recognize the attributes of detected people from an image. In the specific context of the contest, the attributes to be recognized are gender and the presence or absence of a bag and a hat. The dataset used to train the classification model is the one presented during the course: MIVIA PAR Dataset 2023[3].

3.1 Dataset

The MIVIA PAR Dataset consists of 105.244 images (93.082 in the training set and 12.162 in the validation set) annotated with the following labels (or a subset of them):

- **Color of the clothes** (upper and lower): the considered values are black, blue, brown, gray, green, orange, pink, purple, red, white, and yellow, represented in this order with the labels [1,2,3,4,5,6,7,8,9,10,11]. Since these attributes are not part of the contest, they are ignored.
- **Gender**: the considered values are male and female, represented in this order with the values [0,1].
- **Bag**: the absence or presence of a bag is represented with the values [0,1].
- **Hat**: the absence or presence of a hat is represented with the values [0,1].

The unavailability of a specific annotation is indicated with the value -1.

One common challenge, also present in this dataset, is the disparity in sample distribution within the same class and between different classes. This is referred to as an imbalanced dataset and can be observed in Figure 3.1. Having an imbalanced dataset introduces challenges during the training phase, making it crucial to take this aspect into consideration.

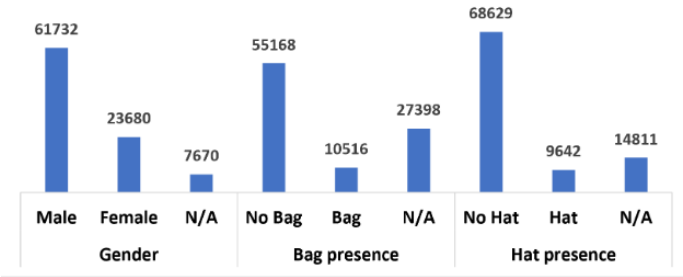


Figure 3.1: Data distribution for each class

3.2 Model Description

The model used for pedestrian attribute recognition is a multitask neural network. Its structure is shown in Figure 3.2. The network takes the preprocessed image as input, which passes through the backbone to extract a feature vector. This vector is then fed into three branches, each with the same structure but solving different tasks: the first branch performs gender recognition, the second one identifies the presence or absence of a bag and the third one detects the presence or absence of a hat.

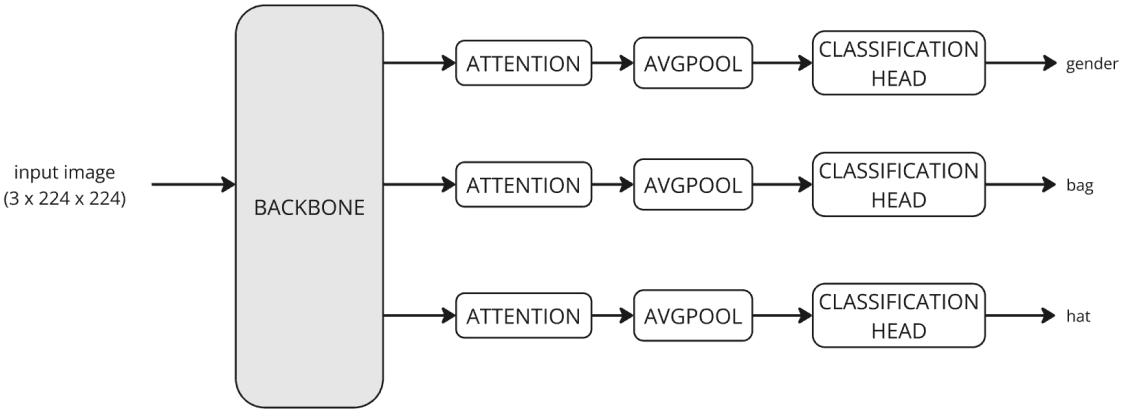


Figure 3.2: General model architecture.

The network uses ResNet50 as its backbone, requiring the input image to be a tensor of dimensions $(3 \times 224 \times 224)$.

3.2.1 Attention Module

Each branch starts with an attention module, comprising a CBAM (*Convolutional Block Attention Module*) and an SEBlock (*Squeeze-and-Excitation Block*). The structure of the CBAM module is shown in Figure 3.3. This module refines the features extracted by the backbone, emphasizing the channels and spatial areas most relevant to the task of the respective branch.

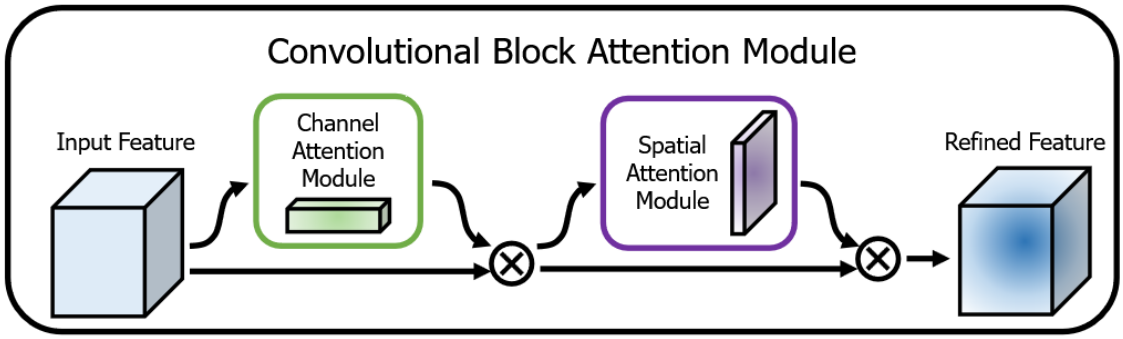


Figure 3.3: CBAM structure.

The *Squeeze-and-Excitation Block* enhances the network representational power by enabling dynamic channel-wise feature recalibration. The process involves:

1. Taking a convolutional block as input.
2. *Squeezing* each channel into a single numeric value using average pooling.
3. Applying a linear layer followed by a ReLU activation to introduce non-linearity and reduce output channel complexity by a ratio.
4. Using another linear layer followed by a sigmoid activation to provide each channel with a smooth gating function.

The structure of the SEBlock is shown in Figure 3.4.

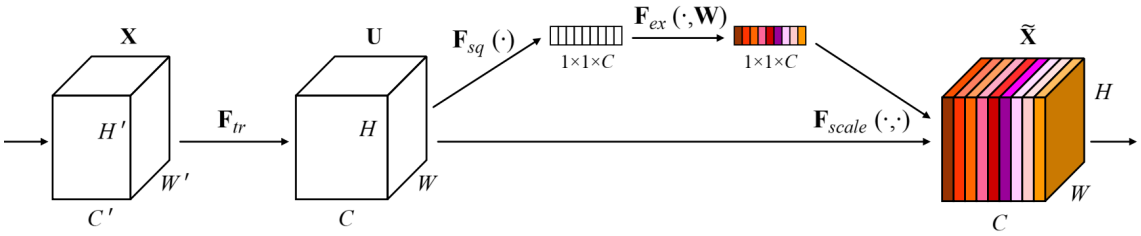


Figure 3.4: SEBlock structure.

By first testing with only CBAM and then combining both modules, an improvement in performance on the test set was observed. This enhancement can be attributed to the following factors:

- emphasis on channel-specific attention and post-CBAM refinement: while CBAM already applies attention to channels, the addition of the SE module could further refine the weights on more critical channels, improving the model's ability to learn meaningful representations. Furthermore, after CBAM has enhanced both channel-wise and spatial features, SE serves as an additional step to consolidate or further refine the channel-wise information, optimizing the effectiveness of the learned representations.
- model capacity increase: SE acts as a complementary module, enhancing the model's ability to capture global patterns.

3.2.2 Classification Head

The output from the attention module passes through an average pooling layer to produce a fixed-dimension tensor, which is then fed into fully connected layers. The classification head consists of the following layers:

- a fully connected layer that reduces the input dimensionality;
- a normalization layer to stabilize the learning process, which its output is passed through a ReLU activation function;
- a dropout layer to prevent overfitting.

This structure is repeated once more (except for the normalization layer), culminating in a final fully connected layer that produces the output for the respective task.

3.3 Data Preprocessing and Augmentation

The training set was split into two subsets: a training split (comprising 80% of the original samples) and a validation split (comprising the remaining 20%). This approach allowed the use of the provided validation set as test set.

As previously mentioned, the training set suffers from class imbalance among the labels within the same class. To address this issue, a weighted sampler was used to create consistently balanced batches. Before training, the weights associated with each dataset sample are calculated. The weights for each class are computed as the inverse of the class frequency, scaled by a factor:

$$Weight(class) = \frac{1}{Number\_Samples(class)} \times scale\_factor$$

with $scale\_factor = 1000$, to assign a higher weight to less frequent samples.

The weights for each sample are combined by averaging the weights of all related classes. For samples labeled with -1 for all tasks, a combined weight of 0 is assigned, as these samples provide no useful information during training. The calculated weights are passed to a *WeightedSampler*, which uses them to ensure the generation of balanced batches during training.

Once the class imbalance issue is addressed, the input images must be transformed to ensure compatibility with the backbone's expected input format. Additionally, random transformations were applied to present the model with different images. Specifically, the following transformations were applied:

- Conversion of the image to RGB format.
- Resizing the image to dimensions of $(3 \times 224 \times 224)$ for compatibility with the backbone.
- Random horizontal flip with a 50% probability, useful for data augmentation.
- Adjusting image parameters within defined ranges: brightness (± 0.3), contrast (± 0.3), saturation (± 0.3) and hue (± 0.1), to simulate varying lighting conditions.
- Converting the image representation from numpy format to tensor.
- Normalizing the channel values to align the image characteristics with those expected by the backbone. Since ResNet50, pretrained on ImageNet, was used as the backbone, images are normalized with respect to the mean values $[0.485, 0.456, 0.406]$ and standard deviations $[0.229, 0.224, 0.225]$, each of them for each image channel.

3.4 Training procedure

The training process for a multitask neural network involves addressing multiple objectives simultaneously, requiring careful consideration of task-specific challenges and dataset characteristics. Unlike single-task networks, multitask training must balance learning priorities between tasks with varying levels of difficulty and importance. In addition, special techniques, such as loss weighting and batch balancing, are used to handle label imbalances and ensure that all tasks contribute effectively to the overall learning process. This approach aims to optimize the network performance for each task while leveraging shared features to enhance generalization.

During the design of the training phase, two main training strategies were chosen to be tested, and after a final analysis, the model with the best performance was selected. Specifically, the two strategies chosen were:

- **Strategy 1**: training with dataset balancing and combining the losses by averaging the individual losses.
- **Strategy 2**: training with dataset balancing and GradNorm.

3.4.1 Training Cycle

The training process is divided into epochs, each consisting of a training phase and a validation phase.

Training Phase: Strategy 1

The training phase was conducted with the following parameters:

- Number of epochs: 50.
- Batch size: 64.
- Backbone Learning rate: 0.0001.
- Classification Learning rate: 0.001.
- Early stopping with a patience of 7 epochs.
- Dataset balancing: True.
- Optimizer: SGD.

Regarding the loss function, a masked loss was implemented to ignore predictions for samples without assigned labels. The chosen loss function was *BCEWithLogitsLoss*[2]. For each task, after creating the mask, the loss for each valid pair (prediction-sample) is calculated as:

$$l_n = -[y_n \cdot log\ \sigma(x_n) + (1 - y_n) \cdot log\ (1 - \sigma(x_n))]$$

where l_n is the loss for the n-th sample in the batch, $\sigma(x_n)$ is the sigmoid applied to the output of the network, and y_n is the ground truth. This computation is performed for all samples in the batch that belong to the mask. Let M denote the number of valid samples in the batch; the vector is:

$$l(x, y) = \{l_1, ..., l_M\}^T$$

The mean of these values is taken to compute the batch loss for the specific task:

$$L = mean(l(x, y))$$

This loss is computed for all tasks in the network. The final loss is the average of individual task losses:

$$final\_loss = \frac{1}{task} \sum_{i \in task} L_i$$

where $task = 3$ is the number of tasks covered by the network.

The model was trained for a maximum of 50 epochs but stopped at epoch 35 due to early stopping to avoid overfitting.

Training Phase: Strategy 2

The training phase was conducted with the following parameters:

- Number of epochs: 30.
- Batch size: 64.
- Backbone Learning rate: 0.0001.
- Classification Learning rate: 0.001.
- Early stopping with a patience of 5 epochs.
- Dataset balancing: True.
- Optimizer: Adam.

The second training strategy uses **GradNorm**, a technique introduced to address training difficulties in multitask networks, mainly caused by disparities in the gradient magnitudes of each task, which can lead to unbalanced training to the detriment of some tasks. Specifically, GradNorm aims to normalize and balance the gradients of each task, keeping them on a common scale and dynamically adjusting their magnitudes based on the observed training rate.
The loss used during training remains the same:

$$loss = \sum_{i \in task} w_i L_i$$

with the difference that, using *GradNorm*, an adaptive strategy is proposed, where the weights w_i can vary at each training step t: $w_i = w_i(t)$.

This linear approach simplifies gradient balancing since w_i directly influences the magnitudes of the gradients backpropagated from each task. The main challenge then becomes understanding how to choose $w_i(t)$ to ensure optimal training for all tasks: if one task trains faster than the others (e.g., its loss decreases too quickly), its weight should decrease, allowing the other tasks to catch up.

The proposed algorithm explicitly penalizes excessively large (or excessively small) gradients of a task, thus maintaining learning rates on a comparable scale. In this sense, it is conceptually similar to batch normalization, but instead of normalizing across data batches, it normalizes gradients across different tasks, taking the learning rate as a balancing objective.

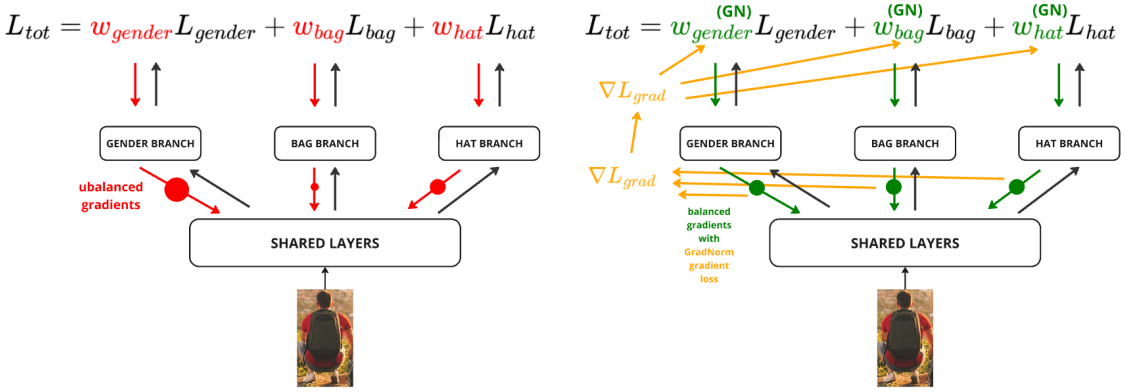


Figure 3.5: Gradient normalization.

In summary, *GradNorm* aims to learn the function $w_i(t)$ with the following objectives:

1. bring the gradient norms of the different tasks to a common scale, enabling a reasonable comparison of their relative magnitudes;
2. dynamically adjust the gradient norms so that the different tasks train at similar rates.

To achieve these goals, we define the following quantities:

- The subset of the overall network weights $\mathcal{W}$ where *GradNorm* is applied. W is chosen as the *network's last shared layer* to reduce computational costs.
- $G^{(i)}_W(t) = \left\|\nabla_W\left(w_i(t)L_i(t)\right)\right\|_2$: the L_2-norm of the gradient of the weighted loss of a single task $w_i(t)L_i(t)$ with respect to the selected weights W.
- $\overline{G}_W(t) = \mathbb{E}_{\text{task}}\left[G^{(i)}_W(t)\right]$: the average gradient norm, computed by averaging $G^{(i)}_W(t)$ across all tasks at time t.

Additionally, the following metrics are defined to measure the training speed of each task:

- $\tilde{L}_i(t) = \frac{L_i(t)}{L_i(0)}$: the loss ratio for task i at time t. This quantity measures the inverse learning speed of task i. Lower values of $\tilde{L}_i(t)$ indicate faster learning.
- $r_i(t) = \frac{\tilde{L}_i(t)}{\mathbb{E}_{\text{task}}\left[\tilde{L}_i(t)\right]}$: the relative inverse training rate of task i. This metric compares the rate of loss reduction of a single task to the average rate across all tasks.

The average norm of the gradient $\overline{G}_W(t)$ establishes a common scale for the gradients, while $r_i(t)$ represents the relative learning rate of task i compared to the others.

The desired gradient norm for task i is defined as:

$$G^{(i)}_W(t) \mapsto \overline{G}_W(t) \times [r_i(t)]^\alpha,$$

where α is a hyperparameter that controls the strength with which GradNorm tries to bring tasks to similar training rates. A higher value of α is useful when tasks have very different learning dynamics, while lower values are suited for tasks with more symmetric dynamics. After several trials, the chosen value is $\alpha = 0.6$, which corresponds to the model with the best performance.

To implement this balancing, GradNorm introduces a loss function L_{grad}, defined as:

$$L_{\text{grad}}(t; w_i(t)) = \sum_i \left| G^{(i)}_W(t) - \overline{G}_W(t) \times [r_i(t)]^\alpha \right|_1,$$

which penalizes the difference between the actual gradient norm and the target one.

During optimization, L_{grad} is differentiated only with respect to the weights w_i, preventing the weights from drifting towards insignificant values.

After each update, the weights $w_i(t)$ are renormalized so that:

$$\sum_i w_i(t) = T,$$

to decouple the gradient normalization from the global learning rate[6].

The following figure 3.6 shows the evolution of the weights during the training phase, initialized to a value of $\frac{1}{3}$, to ensure the same initial weight for each task and that $T = 1$. The best model was saved at the end of the third epoch, which corresponds to iteration number 3276.

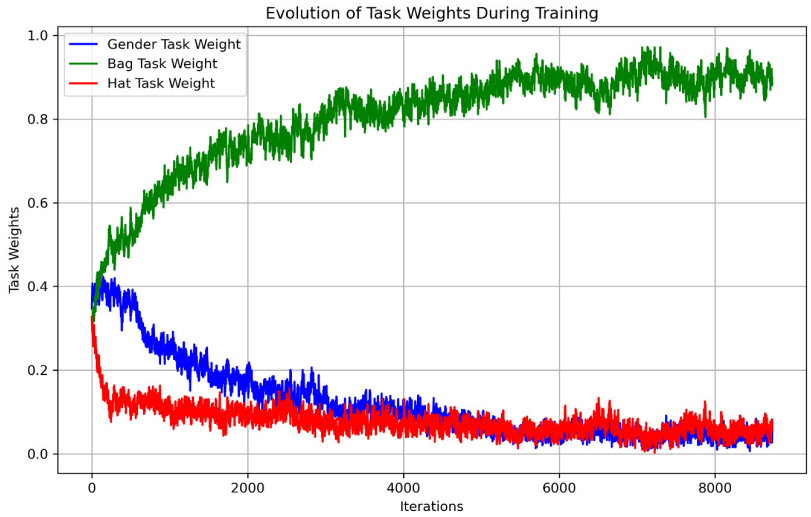


Figure 3.6: Evolution of the weights during the GradNorm training phase

In figure 3.7, the pseudocode of the algorithm just described is presented.

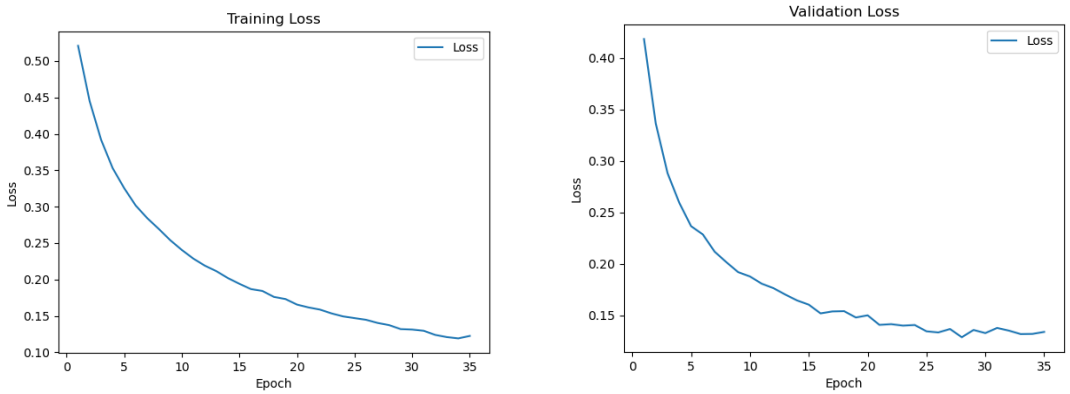
Algorithm 1 Training with GradNorm

Initialize $w_i(0) = 1\ \forall i$
Initialize network weights $\mathcal{W}$
Pick value for $\alpha > 0$ and pick the weights W (usually the final layer of weights which are shared between tasks)
for $t = 0$ **to** max_train_steps **do**
  Input batch x_i to compute $L_i(t)\ \forall i$ and $L(t) = \sum_i w_i(t)L_i(t)$ [standard forward pass]
  Compute $G^{(i)}_W(t)$ and $r_i(t)\ \forall i$
  Compute $\overline{G}_W(t)$ by averaging the $G^{(i)}_W(t)$
  Compute $L_{\text{grad}} = \sum_i |G^{(i)}_W(t) - \overline{G}_W(t) \times [r_i(t)]^\alpha|_1$
  Compute GradNorm gradients $\nabla_{w_i} L_{\text{grad}}$, keeping targets $\overline{G}_W(t) \times [r_i(t)]^\alpha$ constant
  Compute standard gradients $\nabla_{\mathcal{W}} L(t)$
  Update $w_i(t) \mapsto w_i(t+1)$ using $\nabla_{w_i} L_{\text{grad}}$
  Update $\mathcal{W}(t) \mapsto \mathcal{W}(t+1)$ using $\nabla_{\mathcal{W}} L(t)$ [standard backward pass]
  Renormalize $w_i(t+1)$ so that $\sum_i w_i(t+1) = T$
end for

Figure 3.7: Gradnorm pseudocode.

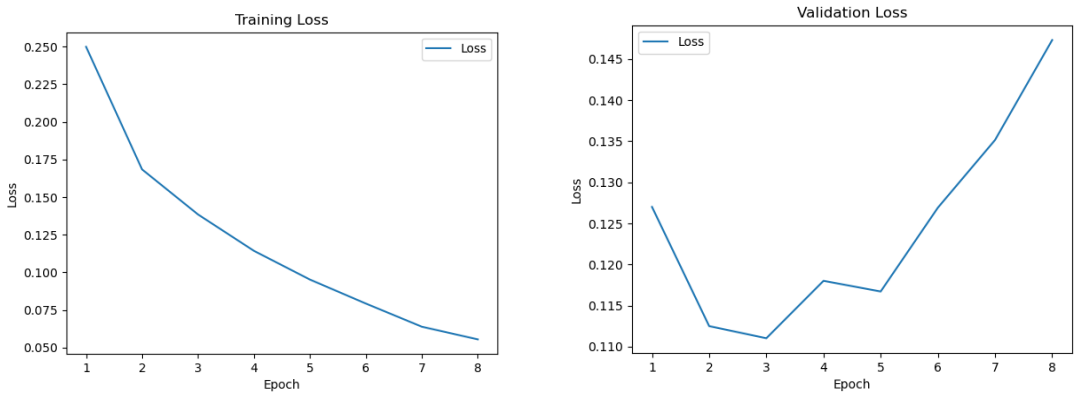
Validation Phase

During the validation phase, the model was switched to evaluation mode. Similar to the training phase, the masked loss was used to compute the combined loss function. Validation samples underwent the same transformations as training samples, except for random horizontal flipping and random light adjustments. The results are shown in Figure 3.8.



(a) (b)

Figure 3.8: Figure 3.8a shows the training loss trend, while Figure 3.8b shows the validation loss trend. Both of them perdormed on the network trained with the first strategy.



(a) (b)

Figure 3.9: Figure 3.9a shows the training loss trend, while Figure 3.9b shows the validation loss trend. Both of them perdormed on the network trained with the second strategy.

This process allowed the model with the minimum loss and the best generalization to be saved.

3.5 Testing procedure

To evaluate the classification models, the following metrics were calculated on the test set:

1. **Accuracy**: the percentage of correct classifications.
2. **Precision**: the percentage of correct predictions among all positive predictions, calculated as:
$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$
3. **Recall**: the percentage of correct predictions relative to all actual positive samples, calculated as:
$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$
4. **F1-Score**: the harmonic mean of precision and recall, calculated as:
$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Table 3.1 shows the results (of the model trained with the strategy 1) obtained for each task on the test set, which consists of $12,162$ samples.

Attribute	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Gender	0.9278	0.8751	0.8747	0.8732
Bag	0.9099	0.7594	0.7650	0.7622
Hat	0.9758	0.9422	0.8005	0.8656

Table 3.1: Performance metrics for different attributes.

Table 3.2 shows the results (of the model trained with the strategy 2) obtained for each task on the test set.

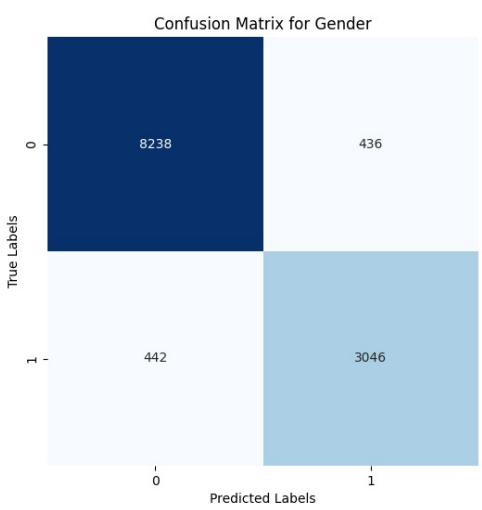
Attribute	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Gender	0.9408	0.8885	0.9076	0.8980
Bag	0.9158	0.8127	0.7214	0.7637
Hat	0.9716	0.9355	0.7607	0.8391

Table 3.2: Performance metrics for different attributes.

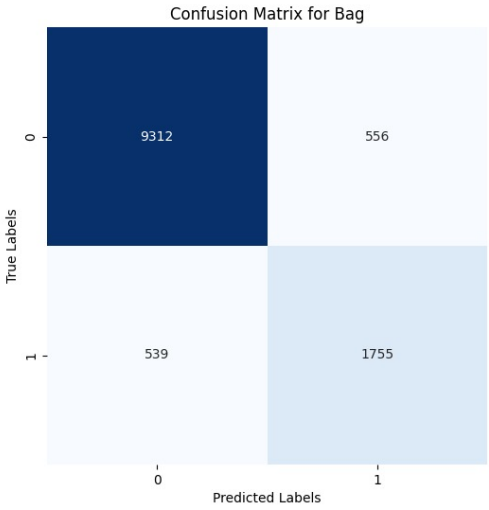
Figure 3.10 shows the confusion matrices for each task.

3.6 Selected model

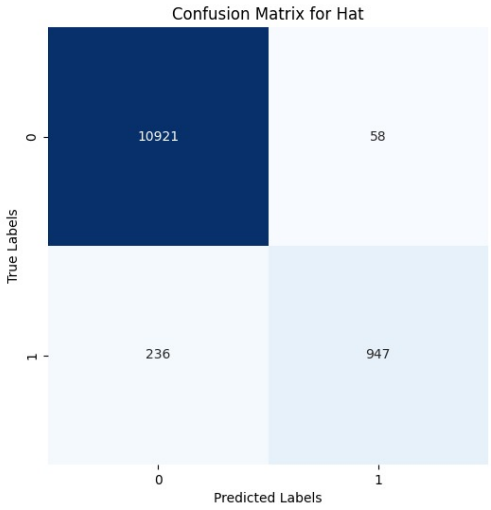
Although the performance measured on the test set is similar between the two models analyzed, further testing in the environment where the contest will take place revealed that the model trained using strategy 2 demonstrates better generalization capabilities, correctly classifying the majority of attributes. For this reason, the model trained with strategy 2 was selected for the contest.



(a)

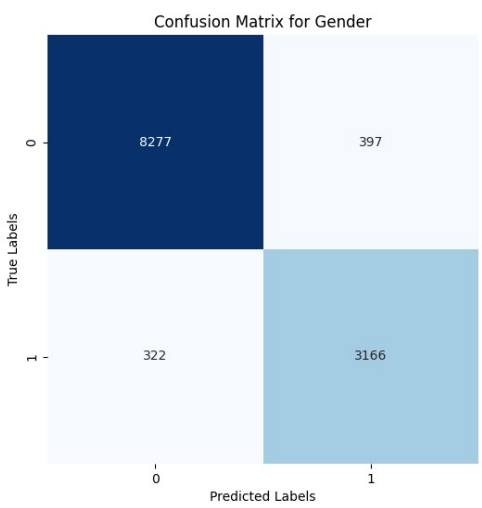


(b)

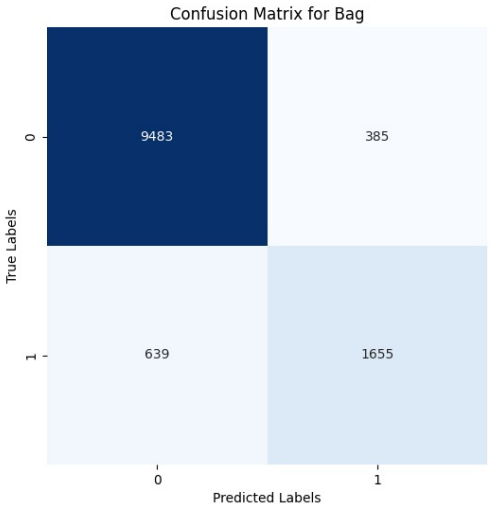


(c)

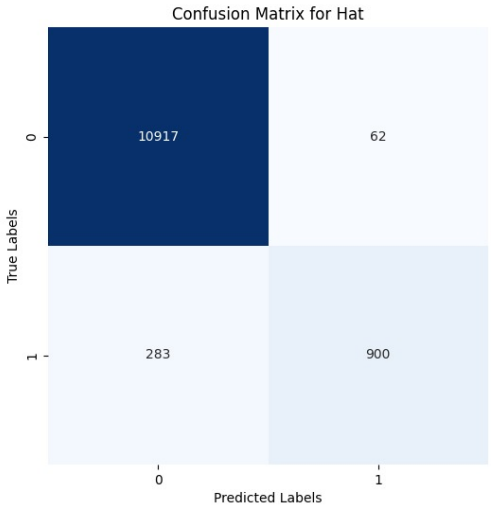
Figure 3.10: Confusion Matrices with first training strategy.



(a)



(b)



(c)

Figure 3.11: Confusion Matrices with second training strategy.

CHAPTER 4

PEDESTRIAN BEHAVIOR ANALYSIS

In this project, the **Pedestrian Behavior Analysis** focuses on monitoring the movement of individuals within a defined area and analyzing their interaction with predefined virtual lines. The ultimate goal is to identify when people cross these lines and, importantly, determine whether they cross them in the correct direction.

To analyze pedestrian behavior, virtual lines are defined in the environment. The starting point coordinates and the ending point coordinates of these lines are provided in the configuration file in *real-world coordinates* and these real-world coordinates are transformed into *pixel coordinates* using mapping 2D/3D.

Then, by using object detection, tracking algorithms and geometric methods, the system is able to check whether individuals cross the predefined virtual lines in the correct direction.

4.1 Mapping 3D/2D

This section explores the process of mapping between the coordinates of the three-dimensional reference system of the real world and the two-dimensional coordinate system of the image captured by the camera. The 3D/2D mapping is essential for drawing lines on the image, where each line is represented by two distinct points in the real world: the starting point and the ending point. Thus, this process allows for the projection of lines, originally defined in a three-dimensional space, into the two-dimensional plane of the image.

To understand the process of coordinate transformation, it is important to first clarify the different reference systems involved.

4.1.1 Camera Coordinate System (3D)

The **camera coordinate system (3D)** is a coordinate system used to describe the position and orientation of the camera relative to the surrounding world. The origin of this coordinate system is located at the optical center of the camera. The axes of this system are defined as follows:

- $\hat{x}$**-axis**: the $\hat{x}$-axis is oriented horizontally, positive to the right from the camera's point of view.
- $\hat{y}$**-axis**: the $\hat{y}$-axis is oriented along the line of sight of the camera, positive forwards from the camera, perpendicular to the $\hat{x}$-axis.
- $\hat{z}$**-axis**: the $\hat{z}$-axis is oriented vertically, positive upwards from the camera's point of view, perpendicular to the $\hat{x}$-axis and $\hat{y}$-axis.

In the camera coordinate system, the camera itself is located at the origin. When the camera moves or rotates, the scene is viewed from a new angle. The camera's orientation is defined by three rotation angles relative to the real-world coordinate system:

- **yaw** (θ_{yaw}): this represents the rotation of the camera around the $\hat{z}$-axis. This angle describes the movement of the camera from left to right or vice versa (horizontal rotation).
- **pitch** (θ_{pitch}): this represents the rotation around the $\hat{x}$-axis. This angle defines the camera's inclination upwards or downwards (vertical rotation).
- **roll** (θ_{roll}): this represents the rotation around the $\hat{y}$-axis. This angle describes the camera's tilting from side to side (lateral rotation).

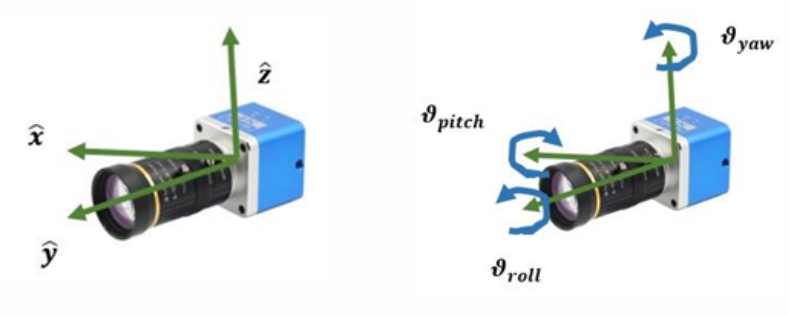


Figure 4.1: Camera reference system

4.1.2 Real World Coordinate System (3D)

The **real-world coordinate system (3D)** is the coordinate system used to represent the position of physical points in the world. The origin of this reference system is positioned at the same location as the origin of the camera coordinate system, but it is translated vertically so that it coincides with the floor plane of the atrium (ground floor). Coordinates in this system are expressed in meters and are used to describe the position of points in the physical context. The axes of this system are defined as follows:

- x**-axis**: the x-axis is oriented horizontally, positive to the right from the camera's point of view.
- y**-axis**: the y-axis is oriented along the line of sight of the camera, positive forwards from the camera, perpendicular to the x-axis.
- z**-axis**: the z-axis is oriented vertically, positive upwards from the camera's point of view, perpendicular to the x-axis and y-axis.

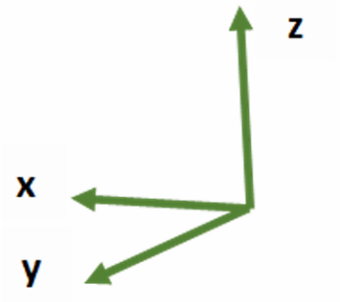


Figure 4.2: World reference system

4.1.3 Image Coordinate System (2D)

The **image coordinate system (2D)** is the coordinate system used to represent images captured by the camera. In this system, the 3D points from the real world are projected into the 2D image plane so that they can be displayed on a screen or processed by computer vision algorithms. The origin of this system is located at the top-left corner of the image and the coordinates are expressed in pixel units, which represent the image resolution. The axes of this system are defined as follows:

- x**-axis**: the x-axis is oriented horizontally and increases to the right.
- y**-axis**: the y-axis is oriented vertically and increases downwards.

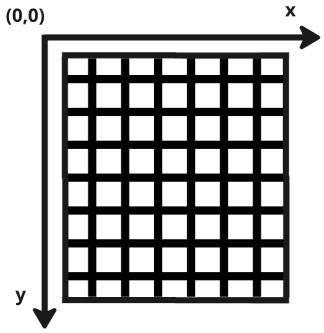


Figure 4.3: Image reference system

4.1.4 Operations for Mapping from 3D to 2D

The process of mapping from 3D to 2D involves a series of mathematical transformations that allow for the accurate projection of the scene from the real-world coordinate system to the camera's coordinate system and finally onto the image's coordinate system. These transformations are crucial for image acquisition and interpretation by computer vision systems, as they allow the 3D scene to be adapted into a format that can be displayed and analyzed on a two-dimensional device, such as a screen. In particular, in the mapping process, three main operations are applied: translation, rotation and projection.

Translation

Translation is necessary to shift the origin of the real-world coordinate system to the position of the camera. In other words, the 3D points from the real world must be moved so that the camera is at the center of these points. Mathematically, this is done by subtracting the camera's coordinates (x_c, y_c, z_c) from each point in the real-world coordinates. The formula for translation is:

$$(x'_{real}, y'_{real}, z'_{real}) = (x_{real}, y_{real}, z_{real}) - (x_c, y_c, z_c)$$

Rotation

Once the coordinates are translated, a rotation must be applied to align the points with the camera's coordinate system. This is because the camera does not necessarily have the same orientation as the real-world coordinate system. Rotation can be performed using rotation matrices for each of the rotation angles: *yaw*, *roll* and *pitch*. Each angle corresponds to a rotation around one of the axes (respectively $\hat{z}$, $\hat{y}$, and $\hat{x}$). The rotation matrices for each angle are as follows:

- **Yaw** (rotation around the $\hat{z}$-axis):

$$R_{yaw} = \begin{bmatrix} \cos(\theta_{yaw}) & \sin(\theta_{yaw}) & 0 \\ -\sin(\theta_{yaw}) & \cos(\theta_{yaw}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **Roll** (rotation around the $\hat{x}$-axis):

$$R_{roll} = \begin{bmatrix} \cos(\theta_{roll}) & 0 & -\sin(\theta_{roll}) \\ 0 & 1 & 0 \\ \sin(\theta_{roll}) & 0 & \cos(\theta_{roll}) \end{bmatrix}$$

- **Pitch** (rotation around the $\hat{y}$-axis):

$$R_{pitch} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_{pitch}) & \sin(\theta_{pitch}) \\ 0 & -\sin(\theta_{pitch}) & \cos(\theta_{pitch}) \end{bmatrix}$$

The total rotation is obtained by combining these matrices, multiplying them in the correct order:

$$R = R_{roll} \cdot R_{pitch} \cdot R_{yaw}$$

This total rotation matrix is applied to the translated coordinates to obtain the coordinates in the camera's system.

Projection

After the coordinates have been translated and rotated, the next step is the **projection** of these 3D coordinates onto the 2D image plane. The 3D image is projected in a 2D image plane using a *central projection* with the following formulas:

$$u = \frac{U}{2} + \frac{f_x \cdot dx}{dy} \qquad v = \frac{V}{2} + \frac{f_x \cdot dz}{dy}$$

Where:

- u and v are the coordinates in pixels on the image plane (2D reference system).
- dx, dy, dz are the coordinates of the point in the camera's system, after translation and rotation.
- f_x is the focal length in pixels along the x-axis of the image, calculated as:

  $$f_x = \frac{f \cdot U}{s_w}$$

  where f is the focal length and s_w is the width of the camera's sensor.
- U and V are the width and height of the image in pixels.

4.1.5 Examples of Mapping 3D/2D

```
image_file = "test20241212.png"
f = 0.003
U = 1280
V = 720
thyaw = 0*pi/180
throll = 0*pi/180
thpitch = -32*pi/180
xc = 0
yc = 0
zc = 7.20

x_real = [-2.5, 0.5, 0.5, 4.6]
y_real = [13.41, 8.00, 13.00, 10.91]

s_w = 0.00498
s_h = 0.00374
```

Figure 4.4: Information about example 1

```
image_file = "test.png"
f = 0.00325
U = 1920
V = 1080
thyaw = 12*pi/180
throll = 10.5*pi/180
thpitch = -36*pi/180
xc = 0
yc = 0
zc = 6.92

x_real = [-3.5, 3.6, -2.8, 2]
y_real = [13.11, 10.61, 4.11, 5.81]

s_w = 0.00498
s_h = 0.00374
```

Figure 4.5: Information about example 2

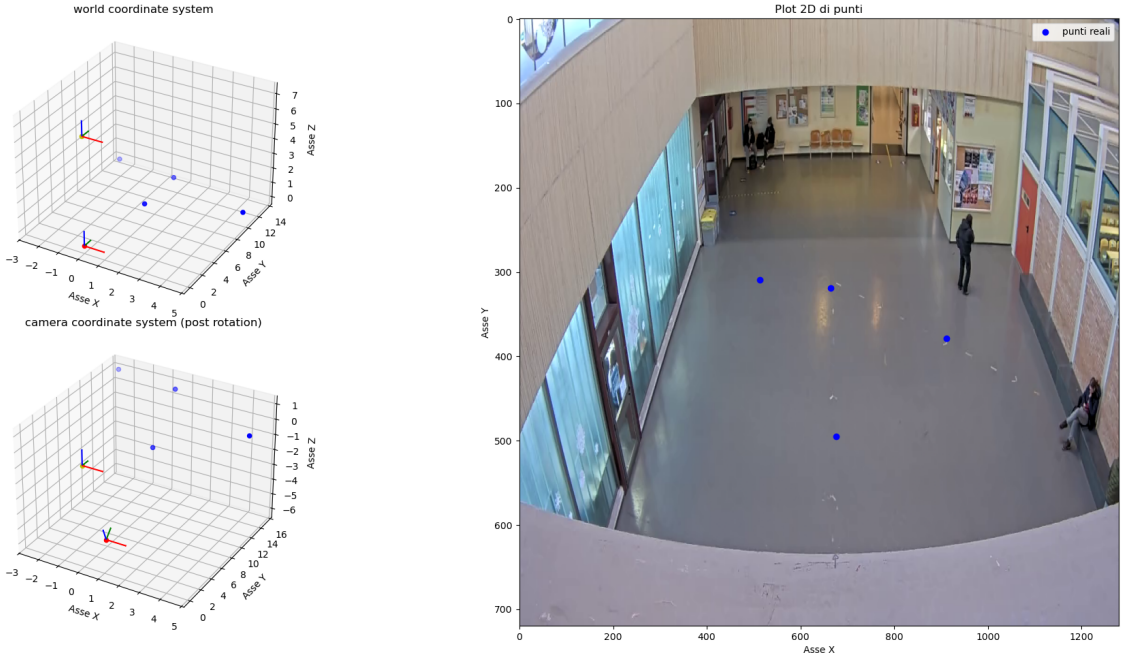


Figure 4.6: Example 1 of mapping 3D/2D

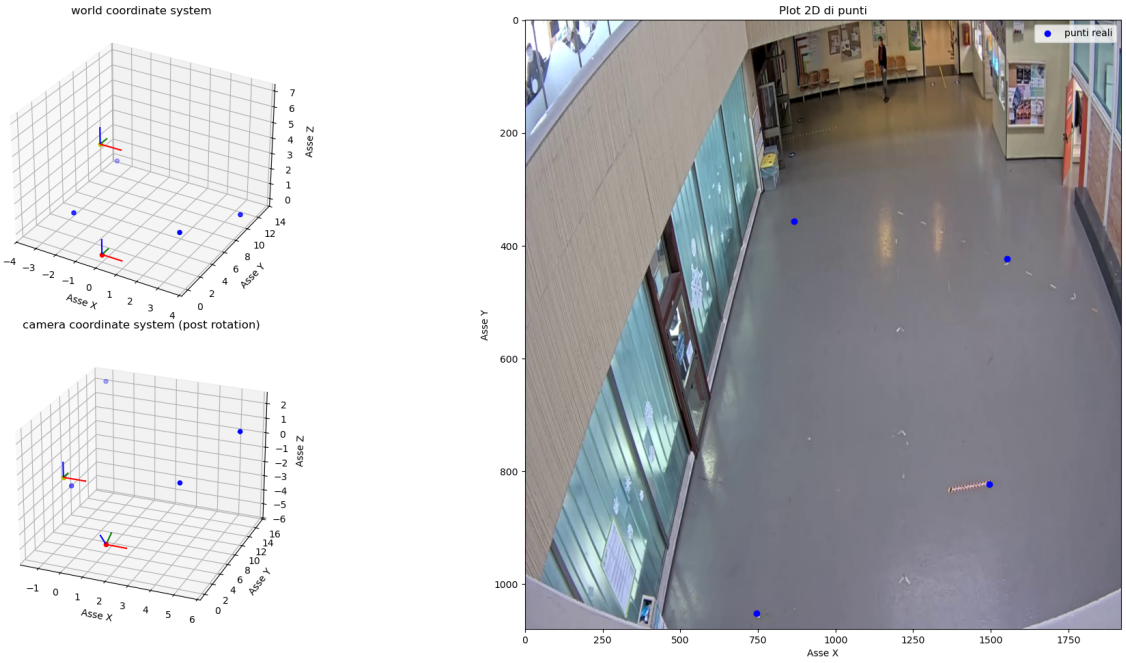


Figure 4.7: Example 2 of mapping 3D/2D

4.2 Crossing Detection

Once the virtual lines are defined in pixel space, the system needs to check if any person crosses these lines. To detect this, the system monitors the movement of each person, identified by their unique ID, and records their trajectory (the sequence of lines they have crossed).

The key step is checking whether the trajectory of a person intersects with any of the virtual lines. This is achieved by calculating the **intersection** between the trajectory of the person (represented by their movement across consecutive frames) and the lines.

To determine if a crossing has occurred, the system calculates whether the last two positions of the person (the starting and ending points of their path in the last two frames) intersect with any line in the scene. This is done using geometric algorithms that calculate the orientation of points and check if the lines (representing the pedestrian's path and the virtual line) intersect.

4.3 Direction of Crossing

Detecting the crossing is not enough; it is also necessary to determine if the person has crossed the line in the **correct direction**. Each virtual line has a defined direction (a person should cross from left to right, from right to left, from top to bottom or from bottom to top), which is determined by the start and end points of the line: from the start point to the end point, the direction of the line is perpendicular to the line and upwards. If the person crosses from the correct side, the crossing is valid. To determine if the crossing is valid, it is computing the dot product between the direction vector of the person's movement and the direction of the arrow placed on the line:

- if the dot product is positive, the person is moving in the correct direction;
- if it's negative, the person is moving in the opposite direction.

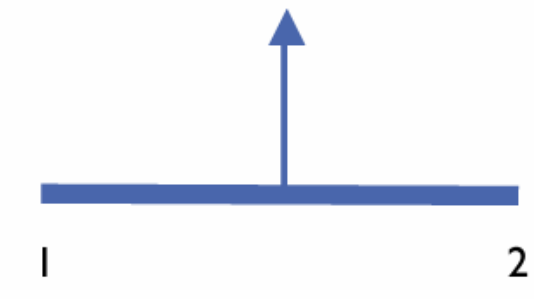


Figure 4.8: The virtual lines have a direction of crossing, which can be obtained from the order of the points, as in the figure.

4.4 Real-time Analysis

At each analyzed frame, the system performs the checks to detect whether any person crosses any of the predefined virtual lines in the correct direction. This is achieved by examining the last two positions of the person, which represent the starting and ending points of their movement. The system checks whether this movement intersects any of the virtual lines defined in the scene (in the correct direction).

For every person, a list is maintained that tracks the sequence of lines they have crossed. This list essentially represents the trajectory of the person, indicating the path they have taken in the monitored area. As the person moves through the scene, their trajectory is continuously updated with the lines they cross. This allows the system to analyze in real time the interactions of people with the defined virtual lines.

CHAPTER 5

SYSTEM OVERVIEW AND METHODOLOGY

In this chapter, the integration of various components for real-time pedestrian tracking and behavior analysis is discussed. The system combines object detection, trajectory tracking, and attribute classification to monitor and analyze pedestrians in video footage. The approach includes processing video frames in real time, classifying attributes such as gender, bag, and hat, and analyzing pedestrian movements based on virtual line crossings. The chapter also highlights how these components are combined to ensure real time processing. It also displays the total number of people in the scene in real time.

5.1 Detection and Tracking

The system starts with detecting people in the video using the pre-trained YOLO model, which detects individuals in each frame and assigns them a unique ID for tracking. Each person is tracked across consecutive frames using BoT-SORT, which ensures correct association between people and their trajectories even in the case of temporary occlusions. Bounding boxes are drawn on the frames, with each person's ID displayed next to their respective box, allowing for easy tracking of pedestrians in the video.

To optimize computational resources and enable real-time processing, a frame-skipping mechanism is implemented. This approach allows the system to process a specific number of frames per second (the processing frame rate is set to 10) by skipping some intermediate frames (the number of frames to skip is calculated based on the frame rate of the video to be processed). This reduces the computational load while maintaining real-time performance.

5.2 Pedestrian Attribute Recognition

Once people are detected and tracked, the system performs attribute classification for features such as gender, bag, and hat. The multi-task classification model uses a cropped image of the person and applies transformations to fit the model's input format. Inference for each attribute occurs at regular intervals.

To improve classification reliability, the system uses the average of probabilities obtained from all frames. This approach helps reduce errors and optimize the final prediction. Attributes are displayed below each person's bounding box, allowing real-time visualization of probabilities for each attribute.

5.3 Behavior Analysis

Behavior analysis is based on tracking virtual lines and recording when people cross them. As pedestrians move through the scene, the system constantly monitors their trajectory and records when they cross the virtual lines in the correct direction. In real time, the system updates and displays the crossing count for each line. Additionally, lines crossed by a pedestrian are displayed next to the attributes under the bounding box.

5.4 Result Generation

At the end of the video processing, the results are saved in the result file, which includes the ID of each pedestrian, their attributes (gender, bag, hat) and the virtual lines they have crossed.

5.5 Conclusions and Future Developments

The developed system has proven effective in tracking individuals and analyzing their behavior in real time. By integrating detection models (YOLO), tracking (BoT-SORT), and multi-task classification, the system is able to detect and analyze pedestrian attributes such as gender, bag, and hat, while monitoring their movement through predefined virtual lines. The results are visualized in real time on a graphical interface, providing a comprehensive overview of people's behavior in the video. Overall, the system has demonstrated good performance in various scenarios, successfully capturing and visualizing complex pedestrian interactions in a dynamic environment.

However, there are several areas where the system can be improved. One potential future development is the inclusion of a re-identification mechanism in tracking. This would improve tracking reliability in situations where people are temporarily occluded or change positions quickly, thus reducing association errors between pedestrians.

Additionally, another area for improvement is pre-processing the images provided to the classifier. Currently, the system uses cropped images from the frames in a generic manner, but it could be beneficial to apply a filter to avoid images containing person intersections. A possible solution could be the use of Intersection over Union (IoU) to select images with a low occlusion, improving the accuracy of the classifier's inferences.

Integrating these and other advanced techniques could further enhance the system's performance, allowing it to handle more complex environments and scenarios with higher pedestrian density.

LIST OF FIGURES

BIBLIOGRAPHY

[1] Anonymous. Yolov11: State-of-the-art real-time object detection, 2025. Disponibile online: https://arxiv.org/html/2410.17725v1#S2.

[2] PyTorch Contributors. Binary cross-entropy with logits loss (bcewithlogitsloss) - pytorch documentation, 2025. Disponibile online: https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html.

[3] MIVIA Lab. Mivia dataset 2023. 2023. Disponibile online: https://mivia.unisa.it/datasets/video-analysis-datasets/.

[4] ultralytics. ultralytics documentation. ., . Disponibile online: https://docs.ultralytics.com/.

[5] Zhongkui Wang et al. Bytetrack: Multi-object tracking by associating every detection box. *arXiv preprint arXiv:2206.14651*, 2022. Disponibile online: https://arxiv.org/abs/2206.14651.

[6] Amos Yu et al. Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. *arXiv preprint arXiv:1711.02257*, 2017. Disponibile online: https://arxiv.org/abs/1711.02257.