energies

MDPI

Article

Counting People and Bicycles in Real Time Using YOLO on Jetson Nano

Hugo Gomes [1,2], Nuno Redinha [2], Nuno Lavado [1,3] and Mateus Mendes [1,4,*]

[1] Polytechnic Institute of Coimbra, Coimbra Institute of Engineering, Rua Pedro Nunes—Quinta da Nora, 3030-199 Coimbra, Portugal
[2] Geologic Information Systems, Rua Pero Vaz de Caminha, 99, R/C, 3030-200 Coimbra, Portugal
[3] Research Group on Sustainability Cities and Urban Intelligence (SUScita), Polytechnic Institute of Coimbra, Rua Pedro Nunes—Quinta da Nora, 3030-199 Coimbra, Portugal
[4] Institute of Systems and Robotics, University of Coimbra, Rua Silvio Lima- Polo II, 3030-290 Coimbra, Portugal
* Correspondence: mmendes@isec.pt

Abstract: Counting objects in video images has been an active area of computer vision for decades. For precise counting, it is necessary to detect objects and follow them through consecutive frames. Deep neural networks have allowed great improvements in this area. Nonetheless, this task is still a challenge for edge computing, especially when low-power edge AI devices must be used. The present work describes an application where an edge device is used to run a YOLO network and V-IOU tracker to count people and bicycles in real time. A selective frame-downsampling algorithm is used to allow a larger frame rate when necessary while optimizing memory usage and energy consumption. In the experiments, the system was able to detect and count the objects with 18 counting errors in 525 objects and a mean inference time of 112.82 ms per frame. With the selective downsampling algorithm, it was also capable of recovering and reduce memory usage while maintaining its precision.

Keywords: real-time object counting; YOLO; edge AI; Jetson Nano

Citation: Gomes, H.; Redinha, N.; Lavado, N.; Mendes, M. Counting People and Bicycles in Real Time Using YOLO on Jetson Nano. *Energies* **2022**, *15*, 8816. https://doi.org/10.3390/en15238816

Academic Editor: Fausto Pedro García Márquez

Received: 29 September 2022
Accepted: 10 November 2022
Published: 22 November 2022

1. Introduction

Counting objects automatically with good precision and accuracy using non-invasive methods such as a video cameras is a long sought-after goal. In structured environments, the problem is difficult, and in unstructured environments, namely outdoor scenarios, it is even more difficult because of the nature of the environment.

In recent years, different approaches have been proposed, using different algorithms and techniques, for object detection and tracking. Fachrie et al. [1] proposed the use of a YOLOv3 [2] detection model without a proper frame-by-frame object tracker. Bharadhwaj et al. [3] proposed the improved Tiny-YOLO with the ensemble Knowledge Distillation [4] and also using the SORT [5] tracker. OpenDataCam [6] is an open-source suite to detect objects in real-time video. Bochkovskiy et al. [7] used YOLOv4 and Tiny-YOLOv4 [8] with a modified version of the IOU tracker [9].

Aside from the referred approaches, more recently, YOLOv5 [10] was published by Ultralytics, providing different architectures of different sizes and mAP scores. The IOU tracker mentioned in the OpenDataCam project has also been extended with a visual tracker, originating the V-IOU tracker [11]. Finally, the SORT tracker was also followed by DeepSORT [12], where the distance metric was replaced by a convolutional neural network (CNN). One challenge that all of these approaches mentioned above faced was that the performance, when running these systems on low-power edge AI devices, was difficult to maintain without draining large amounts of power. This issue sometimes prevented them from using heavier but better models and tracker algorithms in challenging situations.

The present work proposes a system that can be deployed on an edge AI device, namely the Nvidia Jetson Nano 2 GB [13]. The system was optimized to detect, track and count people and bicycles in real time in situations of variable frequency. It was tested on a real-world scenario dataset using the YOLOv5 detection algorithm with the IOU and V-IOU trackers. We also propose the use of a velocity-based frame-downsampling algorithm that enables the system to easily adjust the frame rate at which the detection module processes frames based on the velocity of the objects being detected by either reducing or increasing the frame rate when slower or faster objects are detected. This allows the system to ignore some frames when they are not needed, maintaining performance in real time and without having a big impact on the performance of the tracking. This feature also makes it possible to use cheaper and lower-power devices. The object-counting tests executed achieved a mean absolute error of 0.0435 without downsampling and 0.0457 with downsampling. The downsampling tests showed that the system can easily maintain real-time execution while reducing memory usage and maintaining its precision.

Section 2 briefly reviews the relevant literature in the field of object detection and tracking, as well as previously proposed work tackling the same or similar problems. Section 3 describes the architecture, methods and algorithms used in this work and how they were implemented. Section 4 presents the different experiments conducted and their results.

2. Related Work

Some relevant works have been reviewed, and a summary is presented below.

2.1. Non-Invasive Systems for Counting Objects

Different methods have been proposed to solve the problem of detecting and counting unique objects in real-time video.

Fachrie et al. [1] proposed a method that consists of a system to count vehicles using a YOLOv3 [2] network. The network was pretrained on the MSCOCO [14] dataset. Instead of using a proper object tracker to track the objects frame by frame, Fachrie used a simplified method consisting of calculating the distance from the centroid of a detection to the counting line, and if it was less than a certain threshold value, then it was counted. To prevent the same vehicle from being counted more than once, they analyzed three consecutive frames to identify the same vehicles by measuring their centroids to the previous frame. They achieved a high accuracy of 97.72% for a video with a frontside view of a road.

Bharadhwaj et al. [3] proposed a robust method to count vehicles using edge devices. The work used the detect-track-count framework, where they used an improved Tiny-YOLO network with the ensemble Knowledge Distillation [4], consisting of three different detection networks (Faster-RCNN [15], YOLOv3 [2] and SSD [16]), with the learned model being optimized with TensorRT [17,18]. They used the SORT [5] tracker with a modified version of the Kalman filter to circumvent sudden changes in the speed and position of the vehicles during high traffic density situations. With this work, they achieved a 0.584 weighted average for the normalized weighted root mean squared error (nwRMSE) score on the 2021 AI City Challenge Track 1 evaluation server.

OpenDataCam [6] is an open-source project able to count different classes of objects, including cars, people, motorcycles and other objects. The direction of movement is also detected. It was built to run on CUDA devices, including the NVIDIA Jetson Nano. It uses the darknet framework to run YOLOv4 [7] and Tiny-YOLOv4 [8] as the detection model and a modified version of the IOU tracker [9]. It features a web interface, where the user can configure and interact with the system to define the counting lines and watch the detections in real time. Additionally, this project contains a web API, making it possible for third parties to perform some operations on the system.

Kumar et al. [19] proposed an object tracking and counting system in a zone. This system uses the YOLOv4 model with pretrained weights for the object detection phase and the DeepSORT tracker for the tracking phase. Different from the other methods proposed, this work focuses on counting objects that are specifically inside a defined zone, being able

to detect when a certain amount of objects has been reached inside it. This system was tested on the MOT17 dataset on the NVIDIA Jetson Xavier, achieving a MOTA score of approximately 60%.

Oltean et al. [20] approached the object-counting problem in a similar way to the other methods mentioned above. This system is able to count vehicles in real time. It uses Tiny-YOLO for the vehicle detection and fast motion estimation for tracking. The motion estimation considers all the present vehicles and estimates the most likely position for the next frame, considering just the previous positions and the dynamic average movement. This system was able to properly count vehicles for both one-way and two-way traffic, achieving a speed of 33.5 GPS with a GeForce GTX 950M on an Ubuntu OS system.

2.2. Object Detection Models

During the last few years, there have been very important advances in the area of object detection. The R-CNN [21] deep neural network was proposed, aiming to be a simple and scalable detection algorithm while improving substantially the mAP compared with the other algorithms up to then. This network runs in three consecutive steps. First, it extracts about 2000 region proposals from the image. Then, it computes the features of every region proposal extracted. Finally, those features are fed into pretrained linear SVMs for each class in order to classify those regions. The main problem with this algorithm is that since it has to extract features and classify about 2000 region proposals, the R-CNN becomes computationally very expensive. Afterward, new variations of the algorithm were proposed in order to improve the computing time and minimize the problem. Some improved architectures are the Fast R-CNN and the Faster R-CNN [15,22,23].

In 2016, Liu et al., proposed the SSD [16] algorithm. The SSD is a simple algorithm divided into two components: the backbone and the head. The backbone's model is usually a pretrained image classification model that is used as a feature extractor. Typically, it is a network such as ResNet [24] trained on the ImageNet dataset [25], in which the classification layer is removed. The head of the SSD is composed of one or more convolutional layers added to the backbone, whose outputs are interpreted as the objects' bounding boxes and classes.

The YOLO [26] algorithm proposed by Redmon et al. changed the way object detection was performed until then, transforming it into a single regression problem. This algorithm divides the input image into an $S \times S$ grid, and if the center of an object belongs to a grid cell, then that cell is responsible for identifying that object. That cell then predicts B bounding boxes and their confidence scores as well as C class probabilities. Finally, after the bounding boxes, confidence scores and class probabilities are generated, the model calculates the final detections. One of the problems with the YOLO algorithm was that it had some difficulty detecting smaller objects, and therefore, its precision was lower than some of the other models at the time. YOLO was further improved the following years, with the most recent versions being YOLOv4 [7] proposed by Bochkovskiy et al. and YOLOv5 [10] proposed by Ultralytics.

Table 1 shows a summary of different object detection models, namely the most suitable ones for real time operation. As the table shows, both the YOLO and SSD models achieved good precision and allowed very high frame rates on powerful GPUs. Notice that these results were extracted from the literature. Nonetheless, on edge AI devices, the performance was much worse, as detailed in Section 4.

Table 1. Comparison of object detection models.

Model	Image Size	Backbone	mAP	GPU	FPS
Faster R-CNN [7]	-	ResNet-50	39.8%	GTX 1080 Ti	9
SSD [7]	300	VGG-16	25.1%	GTX Titan X	43
SSD [7]	512	VGG-16	28.8%	GTX Titan X	22
YOLOv4 [7]	416	CPSDarknet-53	41.2%	GTX Titan X	38
YOLOv4 [7]	512	CPSDarknet-53	43.0%	GTX Titan X	31
YOLOv4-tiny [7]	416	CPSDarknet53-tiny	38.1%	GTX 1080 Ti	270
YOLOv5n [10]	640	New CSPDarknet53	28.0%	V100 b1	158
YOLOv5m [10]	640	New CSPDarknet53	45.4%	V100 b1	121
YOLOv5x [10]	640	New CSPDarknet53	50.7%	V100 b1	82
YOLOv5n6 [10]	1280	New CSPDarknet53	36.0%	V100 b1	123
YOLOv5m6 [10]	1280	New CSPDarknet53	51.3%	V100 b1	90
YOLOv5x6 [10]	1280	New CSPDarknet53	55.0%	V100 b1	38

2.3. Tracking Algorithms

In 2017, Bochinski et al. proposed the IOU tracker [9]. This tracker had a very low computational cost, but it was still able to compete with the top tracking algorithms at the time. The IOU tracker uses the intersection over union (IoU) to associate a detection to a track. It looks for the track of its last detection which has the highest IoU value and associates with the new detection if it satisfies a given σ_{iou} value. If no association is made, then a new track is created. Additionally, the tracker's performance is improved by filtering the tracks which are smaller than a length t_{min} and have a detection with a confidence value bigger than σ_h. This also helps with filtering out false positives. The main problems with this tracker are the high rate of track fragmentation and ID switches.

Later in 2018, Bochinski et al. proposed an extension of the IOU tracker, denominated as the V-IOU tracker [11]. This new algorithm improves the previous IOU tracker problems by using a visual single-object tracker. This visual single-object tracker is started when a new detection is not associated with any track. The visual tracking is performed both forward and backward. The visual tracker is initialized in the last known position of an object and used to track the object for a maximum of *ttl* frames. If an association is made, then the visual tracker is stopped, and the IOU tracker is continued; otherwise, the track is terminated. The visual tracking is also performed backward through the last *ttl* frames for each new track. Performing the visual tracking both forward and backward makes it possible to close $(2 \times ttl)$-frame-in-length gaps, while the single visual object trackers only track for a maximum of *ttl* frames.

DeepSORT [12] was proposed by Wojke et al. with the goal of improving and fixing some problems of its predecessor, the SORT [12] tracker. The most important problem is the high rate of ID switches due to the difficulty of tracking objects through occlusions. The DeepSORT tracker replaced the previous association metric with a CNN. After the detections are made, the Kalman filter is used to propagate them from the current frame to the next one. Then, to associate the detections to the tracks, it uses the Mahalanobis distance to quantify the association and the Hungarian algorithm to perform the association. The distance metric used for the association, as stated before, is based on a CNN classifier, where the classification layer is removed, producing a feature vector that is also known as the object's appearance descriptor. This way, the DeepSORT tracker is able to follow objects for a longer occlusion time while keeping it simple and able to run in real time.

Table 2 shows a summary of the performance of the V-IOU and DeepSORT trackers compared with the IOU tracker. The metrics used to compare these algorithms are the following:

- Multi-object tracking accuracy (MOTA): Measures three types of tracking errors: false positives, false negatives and ID switches. It does not take into account the localization error or the detection performance as a significant influence on this measure.

- Multi-object tracking precision (MOTP): Measures the localization accuracy by averaging the overlap between all the correctly matched predictions and their ground truth.
- Mostly tracked (MT): Percentage of ground truth tracks that have the same label for at least 80% of their lifespans.
- Identity switches (IDs): Number of times an object with the same identity ground truth changes its ID.
- Fragmentations (FMs): Number of times a track is interrupted by a missing detection.

From Table 2, it is possible to conclude that the IOU tracker was the worst performing one between the three. Even though it achieved the same MOTA and MOTP values on the UA-DETRACT-test benchmark as the V-IOU tracker, it had the lowest MT percentages and the highest ID switch and fragmentation counts, showing how much the V-IOU tracker improved in comparison with the IOU tracker. When comparing the IOU and DeepSORT trackers on the MOT17 and MOT16 benchmarks, respectively, since they were very similar, DeepSORT showed itself to be much better than the IOU tracker in every metric evaluated. Unfortunately, the V-IOU and DeepSORT trackers were not compared directly due to the lack of results on the same (or a similar) dataset.

Table 2. Tracker comparison.

Tracker	Detector	MOTA	MOTP	MT	IDs	FM	Benchmark
IOU [11]	Mask R-CNN	30.7%	37.0%	30.3%	668	733	UA-DETRACT-test
V-IOU [11]	Mask R-CNN	30.7%	37.0%	32.0%	162	286	UA-DETRACT-test
IOU [11]	Faster R-CNN	45.5%	76.9%	15.7%	5988	7404	MOT17
DeepSORT [12]	Faster R-CNN	61.4%	79.1%	32.8%	781	2008	MOT16

3. System Architecture and Implementation

The research for this project was divided into different steps. The first step was the literature review performed in order to understand which approaches had already been tried to solve the same problem and get to know different approaches and algorithms previously used. The second step consisted of the implementation of the system. The last step was focused on testing the system with different algorithms, both for object detection and tracking on a test dataset based on a real-world scenario where this application will be used, in order to select the best algorithms and parameters.

3.1. System Architecture

The architecture of the system followed a method similar to that of Bharadhwaj et al. [3], which was based on the following steps: detect, track and count. A representation of this architecture is depicted in Figure 1.

The system is divided into three different modules. The first module deals with video input in real time. The second module deals with object detection for detecting objects in the video stream. The last module deals with object tracking, following the objects along consecutive frames. These three modules are executed in different processes, and each one performs a different operation. The data flow sequentially from one to another through message queues. The processes run in parallel, thus achieving better performance than in series. This was achieved by using the Python multiprocessing library, which helped bring true parallel execution, as opposed to using threads, which after an initial experimentation showed that the performance of the system was affected due to its concurrent execution and, more specifically, the detection module because of the way the YOLOv5 model was implemented. Additionally, the Queue class from the multiprocessing library was used for communication between processes. This class already takes care of synchronization by resorting to locks and semaphores, thus preventing any synchronization problems from occurring. The video input module takes the frames and sends them to the detection module at 15 FPS. The detection module is then responsible for extracting the relevant objects which are found in each frame and sending the list of objects detected to the tracker

module. The tracker module associates the objects received with existing tracks or new tracks which are created if no matching tracks are found. After associating the objects to tracks, the tracks are then counted if they overlap with the counting line which defines the area of interest of the image. The modules forward the data to each other using queues. This architecture and implementation of the project were developed in a way that left open the possibility of implementing, testing and using new object detection models and tracking algorithms in the future, if deemed appropriate. The code of this project was developed from scratch, with the exceptions of the YOLOv5 model, which used the official implementation in the repository [10], and the object tracker, whose code came from the official repository [27] and was adapted to better fit this project.

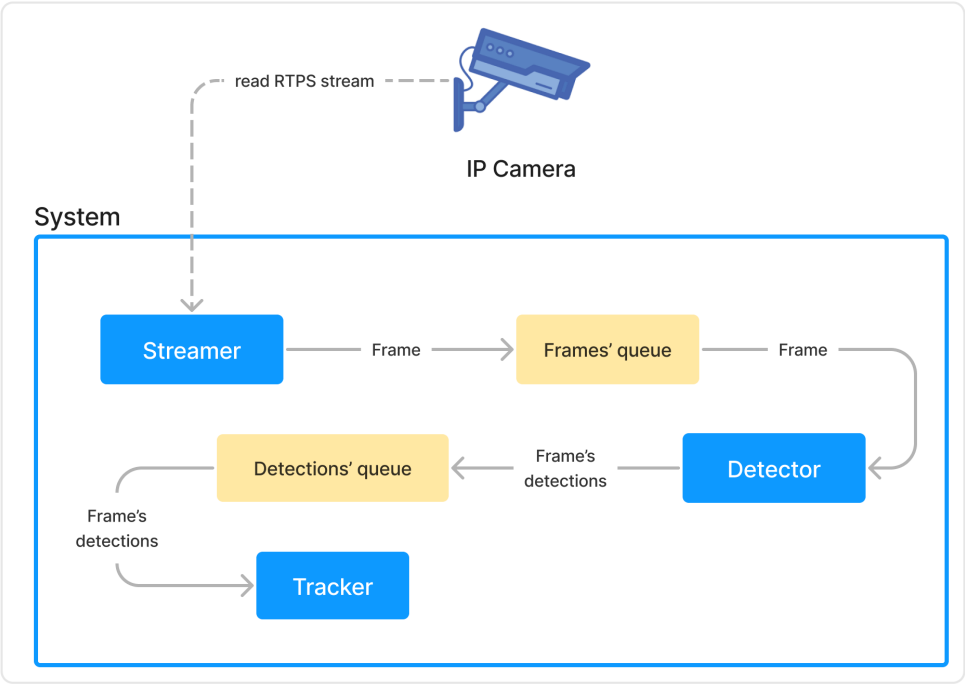

Figure 1. Diagram of the system architecture, showing the three main modules and data queues.

3.2. Detection Module

The detection module uses the YOLOv5 [10] network optimized by TensorRT [18] to detect people and bicycles, although it is also possible to use the original PyTorch framework version. This model was chosen over the others previously researched because of its good results, variety of architectures and ease of implementation.

One problem that arose when using the YOLOv5 architecture was that the queue with the frames from the video input module was always increasing in size due to the detection process taking more than the system was able to handle. Naturally, this would eventually lead to a system overflow when the Jetson Nano ran out of memory. This problem is an important limitation of the system. Preliminary results showed that a solution was not possible without significant adjustments, which could be one of the following:

- Reduce the frame rate. This would decrease the number of frames to process per second. However, this also means that the difference between frames is larger and can thus compromise the performance of the tracker, especially when faster objects move through the frame.
- Use faster algorithms, such as a faster YOLO implementation or object detection model. This would decrease the processing time and thus increase the frame rate. The cost, however, is lower performance by the algorithms, for a faster object detector has lower accuracy and precision, and a faster tracker also has poorer performance.

- Use a faster edge AI device. While this is also a simple solution, the main problem with this solution is a significant cost increase for the solution, which is not desirable for the product being developed. The increase in cost is not only due to the edge AI device but also the power requirements. The device is expected to run on solar panels and batteries, and additional power requirements can significantly increase the cost of the system.
- Make modifications to the algorithm to optimize the processing time. This was the preferred solution, and it is described below.

The developed system is, by customer requirements, designed to operate in remote areas with low traffic and a limited energy supply. Therefore, the frequency of objects to track is expected to be low, and thus not all frames need to be tracked. Based on this information, the detection and tracking algorithm can be adjusted so that the time- and energy-consuming algorithms are only activated when necessary. A velocity-based frame-downsampling algorithm was implemented, optimizing the usage of the detector and tracker modules. This algorithm is able to adjust the frame rate at which the detection module processes the frames by ignoring some of them, depending on the velocity of the fastest object in a frame. This way, the system can use a lower frame rate when there are no objects or slower ones in the frame, such as people walking or standing still, and use a higher frame rate when faster objects are detected, such as moving bicycles. This method helps the system by reducing the frame's queue size, maintaining its execution in real time while reducing energy and memory consumption without having a big impact on the tracker's performance and consequently the counting error.

The algorithm starts by generating a frame batch with the size of the max FPS the system is able to process, consisting of a list of Boolean values that map whether a frame should be processed or not. The frame batch content is generated as a function of the velocity of the fastest object in a frame. To prevent sudden changes in velocity, the value considered in the frame batch generation is the mean of the max velocity values in the last 10 frames. Given the max velocity value, the frame rate to be used is calculated by the function in the Figure 2. This function outputs a lower frame rate for slower objects and a higher frame rate for faster ones. With the frame rate value calculated, the frame batch is finally generated, with the frames that will not be processed by the system evenly distributed across the batch to reduce the distance of a moving object between two frames. While the frame batch is not empty, the system will remove its first element and use it to decide whether the current frame will be ignored or not. When it gets empty, a new frame batch is generated. Additionally, for every frame tracked, the velocity of the fastest object in it is saved for later usage.

After the detections are extracted from the model, they are fed into a non-maximum suppression algorithm with a lower IoU threshold to try and reduce duplicate detections. After that, these detections are passed through a class agnostic non-maximum suppression algorithm but with a higher IoU threshold. This is due to the fact that there were many situations where, when detecting a bicycle, the model would detect a bounding box of a person very similar to the bounding box of the bicycle in addition to the normal and correct bounding box of the person, which led to an over-counting situation.

3.3. Tracking Module

This module is responsible for both tracking and counting. In this project, the IOU and V-IOU trackers were used.

The system starts by retrieving a list of detections from the queue that were sent by the detection module. Then, these detections are fed through the tracker—either the IOU or V-IOU tracker—so these detections can be associated with the tracks. After this, each track's direction is updated and obtained by calculating the vector from the oldest frame to a limit of 20 to the last frame. After the tracks are fully updated, these are then counted. A track is counted when the last detection's bounding box overlaps a defined counting line, and it will only happen the first time it overlaps with the line. For every track counted,

there will be a new registry inserted in a local SQLite database containing the track's ID, direction, class and highest confidence score, as well as the timestamp of the moment it was counted.

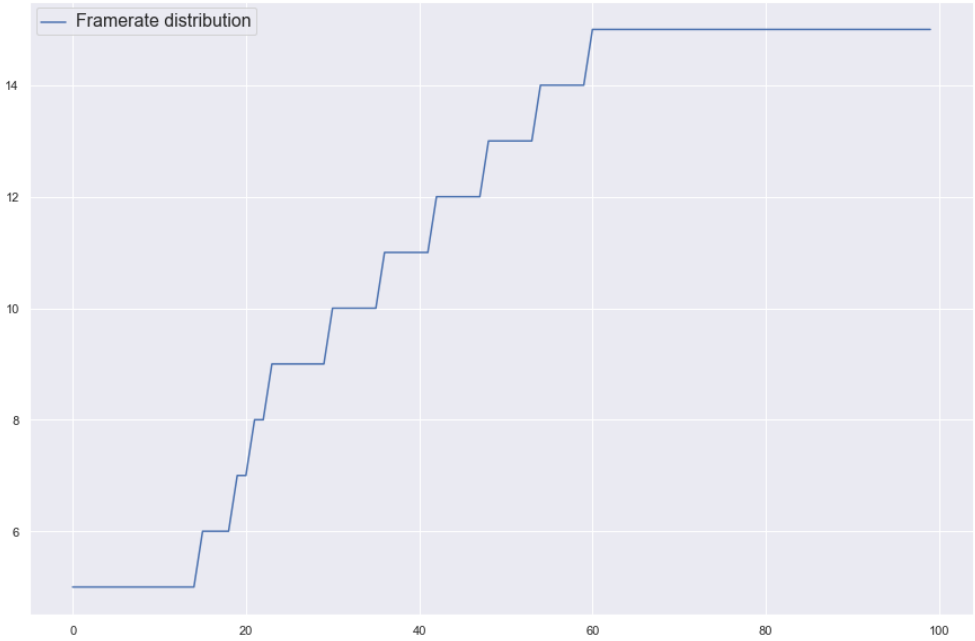

Figure 2. Frame rate distribution function.

3.4. Implementation

For this project, we selected the YOLOv5 network due to it being able to run at a faster frame rate compared with the Faster R-CNN and SSD models, and because it is supported by PyTorch and TensorRT, making it easier to implement than YOLOv4 and YOLOv4-Tiny. The YOLOv5 network has a wide number of architectures of different sizes and mAP scores available, but due to the hardware restrictions presented by the Jetson Nano, we selected the following architectures for experimentation: YOLOv5n, YOLOv5n6, YOLOv5s and YOLOv5s6. The models were optimized by TensorRT specifically for the Jetson Nano with the goal of reducing their inference time. The optimized model was obtained by running one script from the official YOLOv5 repository [10], and then it was imported and used in the system with the use of the respective class in the Computer Vision Utils [28].

The tracker algorithms chosen were the IOU and V-IOU trackers [9,11], since they run very quickly compared with DeepSORT, which turned out to be inapplicable on the Jetson Nano due to its low FPS count. The code of the trackers used in this project was based on the original repository [27], with the only change being some code restructuring and refactoring for real-time detections.

4. Results

Multiple tests and experiments were conducted with the final goal of obtaining the best combinations of algorithms and their best parameters. These tests were performed on a private dataset of 339 videos at 4 s each, with people and bicycles passing by. The videos are examples of real-world scenarios of the use case this project was intended to solve. Figure 3 shows how the data were distributed, showing the number of videos where x people or bicycles passed through the area of interest and therefore should have been counted. The figure shows the number of objects distributed by direction of movement, namely from left to right or vice versa.

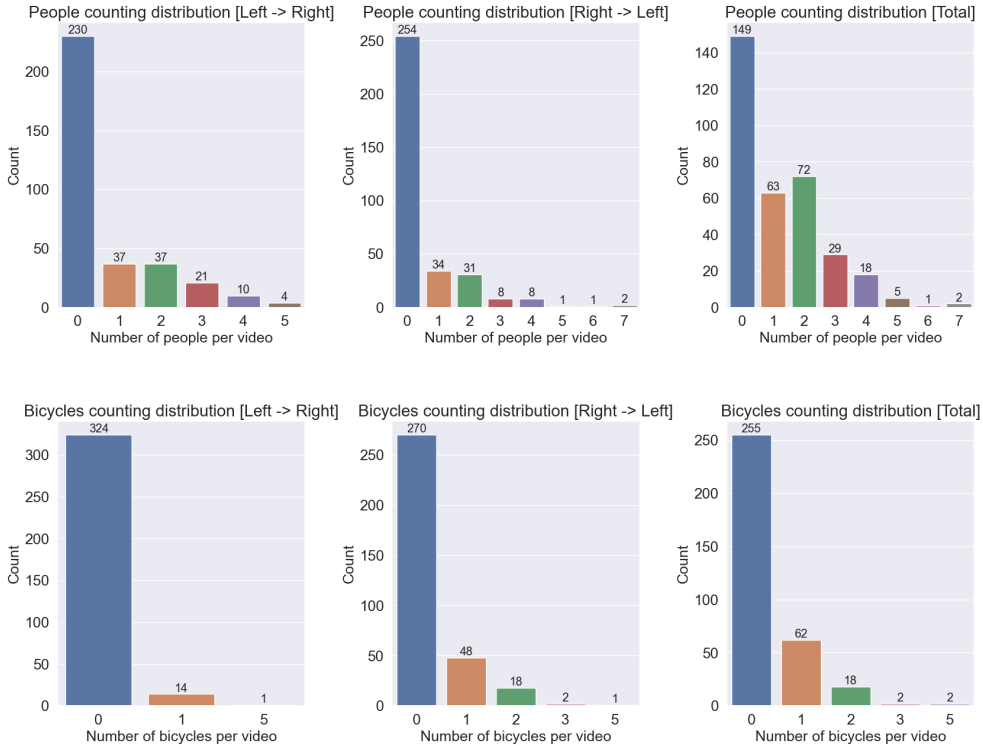

Figure 3. Characteristics of the dataset used to test the algorithms.

4.1. Object Detection

In this set of experiments, the main goal was to compare different YOLOv5 architectures, the performance of the PyTorch framework and the TensorRT optimized models. To test these architectures together, we extracted several frames from the videos of the testset, obtaining a total of 630 images at a size of 640 × 640 pixels properly annotated with the respective bounding boxes for people and bicycles. The annotations were made by using Roboflow's annotation tools (https://roboflow.com/, accessed on 9 November 2022). In this experiment, we measured the $AP^{IoU=0.50:0.05:0.95}$ for both people and bicycles individually and the final mAP combining the last two results. Additionally, the mean inference time (MIT) per image in milliseconds (ms) was also evaluated in order to compare the different architectures and the difference in performance between PyTorch and TensorRT.

The results of this experiment are summarized in Table 3. They show that the models optimized by TensorRT outperformed the ones implemented in PyTorch in terms of the inference time, which was reduced by approximately 50%. One other interesting result that can be observed is the improvement in the mAP when using the TensorRT-optimized models. Another interesting result is that the architectures YOLOv5n and YOLOv5s outperformed their bigger versions (YOLOv5n6 and YOLOv5s6, respectively). This could be due to the fact that the input images had a size of 640 × 640 pixels, which was smaller than the image size the worst-performing versions of these architectures were trained on (1280 ×1280 pixels). Given these results, the selected architectures for the next experiments were YOLOv5n and YOLOv5s optimized by TensorRT.

4.2. Tracking and Counting

In this group of tests, the main goal was to see how the trackers performed in comparison with each other and then how the best of the two performed when using a better object detection model. First, both trackers were tested with the lighter YOLOv5n architecture, and then in the best experience, the detection model would be replaced by YOLOv5s.

These tests would evaluate the mean inference time (MIT) that a frame took to be processed by the detection model and the tracker, the number of ID switches (IDs) and the

mean absolute error (*MAE*). The *MAE* was calculated by averaging the counting error for each individual video, taking in consideration the classes of the objects and their directions.

Table 3. Results of object detection. The best results of each experiment are highlighted in bold type.

		$AP^{IoU=0.50:0.05:0.95}$			
	Model	**People**	**Bicycles**	**mAP**	**MIT (ms)**
PyTorch	YOLOv5n	50.76%	42.08%	46.79%	114.20
	YOLOv5n6	52.26%	39.23%	45.75%	117.67
	YOLOv5s	57.81%	**61.20%**	59.50%	216.37
	YOLOv5s6	58.11%	56.43%	57.27%	221.39
TensorRT	YOLOv5n	55.36%	43.13%	49.25%	66.99
	YOLOv5n6	56.04%	38.60%	47.33%	**62.79**
	YOLOv5s	60.05%	60.98%	**60.51%**	100.06
	YOLOv5s6	**60.18%**	54.51%	57.35%	108.26

In these experiments, the values of the parameters were the same for both trackers and had fixed values, as shown in Table 4. Additionally, in this group, the class-agnostic nms was npt used due to not being implemented yet.

Table 4. System parameters.

Tracker	σ_l	σ_h	σ_{iou}	**tmin**	**ttl**	**nms**	**Class-Agnostic nms**
IOU	0.2	0.5	0.05	10	-	0.4	-
V-IOU	0.2	0.5	0.05	10	8	0.4	-

Analysing the results shown in Table 5, the V-IOU tracker appeared to greatly outperform the IOU tracker. The biggest improvement was in the ID switches, which were reduced by 90.8%. This may also be responsible for the *MAE* reduction of 8.85%. The only parameter where the IOU tracker was better than the V-IOU tracker was the mean inference time, which was slightly higher, although the benefits outweighed this higher value. Using a YOLOv5s detection model also influenced the number of ID switches, although surprisingly, the *MAE* was 0.15% higher than when using the YOLOv5n. One problem that was detected when analysing these results was the very high error for the the people moving in the left-to-right direction.

Table 5. Tracker and counting results.

		Left to Right		**Right to Left**				
Model	**Tracker**	**People (*MAE*)**	**Bicycles (*MAE*)**	**People (*MAE*)**	**Bicycles (*MAE*)**	***MAE***	**MIT (ms)**	***IDs***
Expected		234	19	177	95	-	-	-
YOLOv5n	IOU	185 (0.2035)	38 (0.0796)	261 (0.2832)	141 (0.2005)	0.1917	67.46	1369
YOLOv5n	V-IOU	249 (0.1091)	21 (0.0294)	238 (0.2153)	103 (0.0590)	0.1032	82.14	125
YOLOv5s	V-IOU	241 (0.0973)	22 (0.0206)	239 (0.2242)	115 (0.0767)	0.1047	111.82	100

4.3. Parameter Optimization

With these tests, we tried to solve the problems encountered in the previous tests while trying to find the optimal parameters for the system by performing various experiences. The optimization of the parameters was carried out manually while analyzing the results and creating a new hypothesis for better results based on the previous results.

Going back to the last test group results, the biggest problem encountered was the very high error rate when counting people in the direction from right to left. The first step to know the origin of this problem was to observe the confusion matrix of the people counter in this same direction (Figure 4). It is clear that the main source of the problem was the system counting people when it was not supposed to. Another interesting aspect is that this happened more in this particular direction.

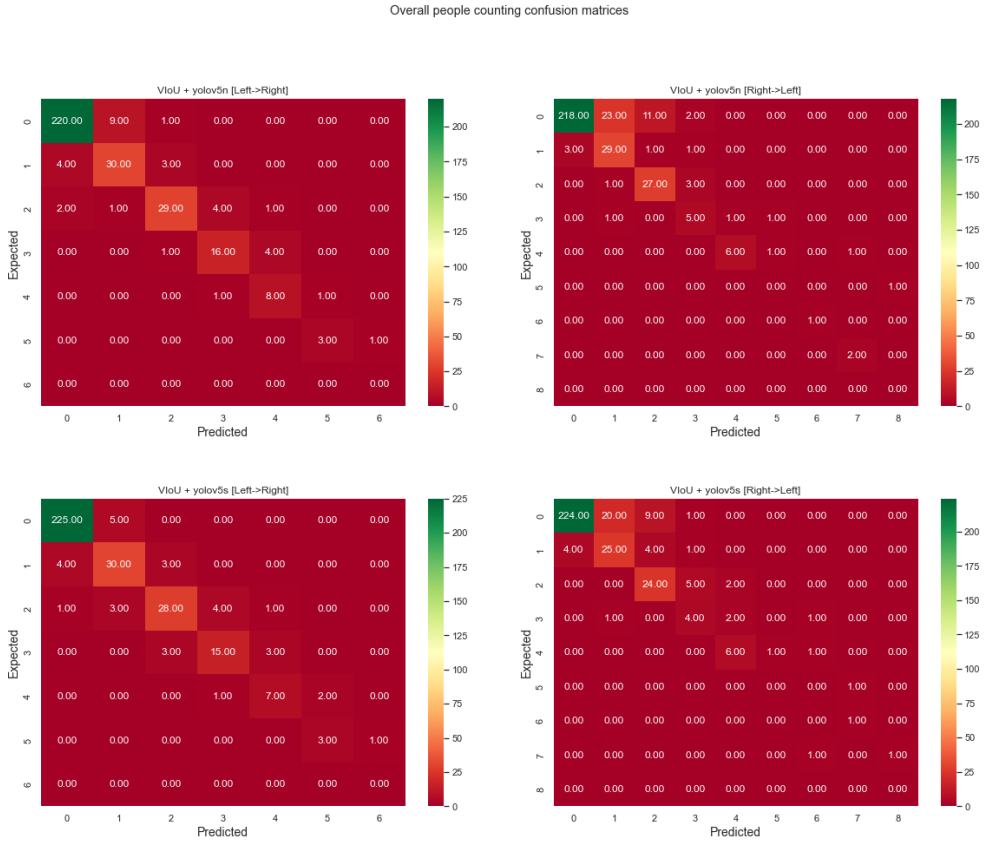

Figure 4. Confusion matrices of people counting overall.

Next, we observed the confusion matrix of the people count but only in the videos where there were also bicycles counted this time (Figure 5), which showed that there were more people being counted when they should not have been in these videos, as shown in Table 6.

Looking at the bounding box output of some of these videos, we noticed that there were situations where there was a bounding box for the person riding the bicycle, with one for the bicycle and another one for the person. This last one was very similar to the bounding box of the bicycle. To try and correct this, we implemented the additional class-agnostic nms algorithm right after the first one with the goal of filtering these out by eliminating the one with a lower score when two or more boxes overlapped with a higher IoU threshold. Figure 6 presents the results before and after applying this class-agnostic nms algorithm.

4.4. Downsampling Test

The YOLOv5s architecture was proven to give better results than the YOLOv5n architecture, although for real-time detection, it raised the problem of its frame queue being constantly filled with new frames, deeming it inappropriate for the present project. YOLOv5n worked at approximately 15 FPS in the configuration used. YOLOv5s worked at approximately 10 FPS only. If the system passed the retrieved frames by the detector, using YOLOv5n, it could easily empty out the queue, although with YOLOv5s, this queue would

increase its size by roughly five frames every second. When analyzing a long video or real-time footage, the Jetson Nano will eventually run out of memory, which is something that needs to be prevented from happening.

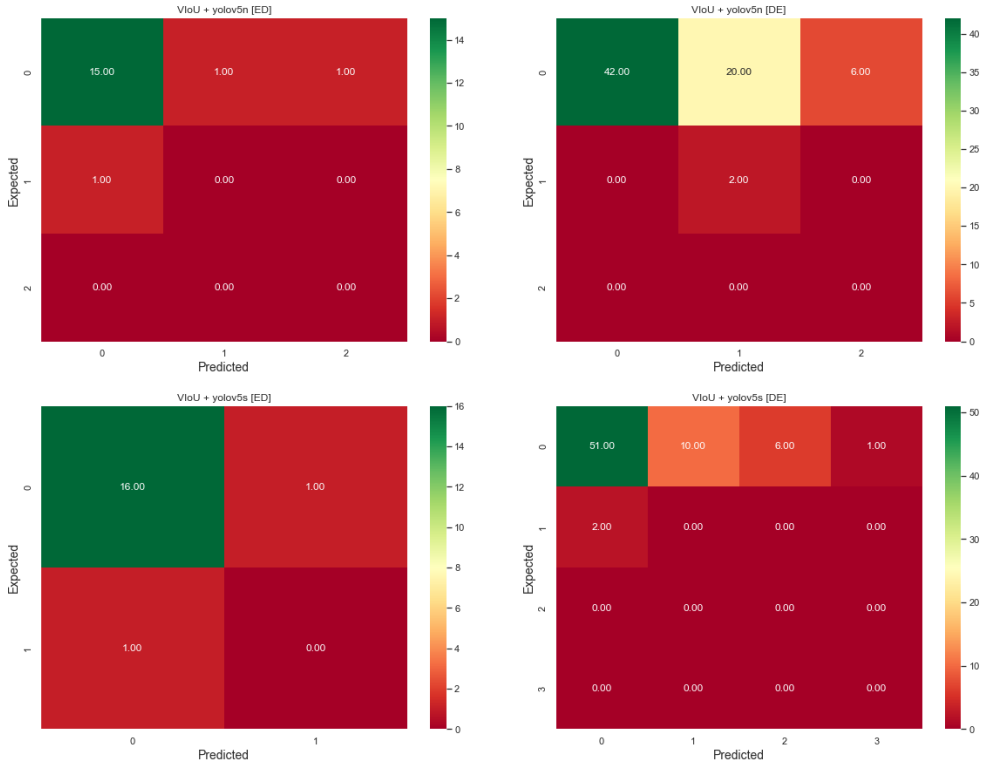

Figure 5. Confusion matrices of people counting in videos with bicycles.

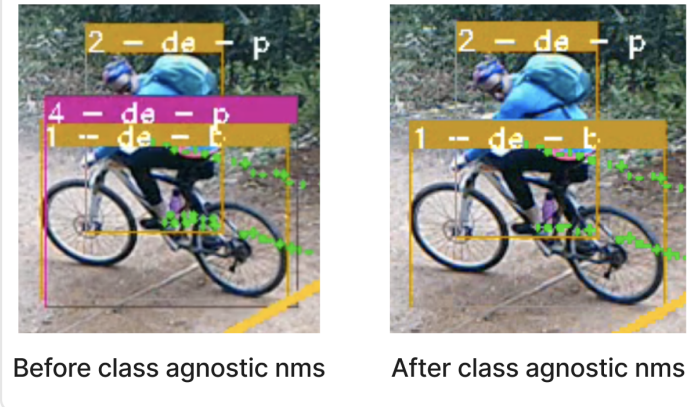

Figure 6. Before and after applying the class-agnostic nms.

To solve this problem, we proposed a selective downsampling algorithm capable of adjusting the frame rate at which the system operated based on the velocity of the fastest objects in the previous frames without having a big impact on the tracker's performance. In this group of tests, we evaluated the usage of this downsampling algorithm, comparing it to the results obtained when not using this algorithm.

Table 6. Tracker and counting results.

Model	Parameters					Left to Right		Right to Left		*MAE*	MIT (ms)
	σ_l	σ_h	tmin	ttl	nms_agnostic	People (*MAE*)	Bicycles (*MAE*)	People (*MAE*)	Bicycles (*MAE*)		
Expected				-		234	19	177	95	-	-
YOLOv5n	0.2	0.5	10	8	1.0	249 (0.1091)	21 (0.0294)	238 (0.2153)	103 (0.0590)	0.1032	82.14
YOLOv5s						241 (0.0973)	22 (0.0206)	239 (0.2242)	115 (0.0767)	0.1047	111.82
YOLOv5n	0.2	0.5	10	8	0.8	247 (0.1032)	21 (0.0295)	232 (0.1976)	102 (0.0560)	0.0966	97.32
YOLOv5s						241 (0.0914)	21 (0.0177)	226 (0.1858)	102 (0.0472)	0.0855	110.34
YOLOv5n	0.2	0.5	10	8	0.6	248 (0.1062)	21 (0.0295)	219 (0.1593)	102 (0.0560)	0.0878	81.24
YOLOv5s						241 (0.0914)	21 (0.0177)	224 (0.1740)	102 (0.0383)	0.0804	109.93
YOLOv5n	0.2	0.5	30	20	0.6	248 (0.1062)	19 (0.0236)	210 (0.1327)	101 (0.0472)	0.0774	86.54
YOLOv5s						241 (0.0914)	21 (0.0177)	221 (0.1652)	100 (0.0324)	0.0767	113.60
YOLOv5n	0.35	0.5	30	20	0.6	239 (0.0914)	19 (0.0236)	201 (0.1121)	94 (0.0383)	0.0664	84.39
YOLOv5s						233 (0.0678)	21 (0.0236)	192 (0.0796)	94 (0.0295)	0.0501	111.96
YOLOv5n	0.45	0.5	30	20	0.6	238 (0.0767)	17 (0.0236)	192 (0.1091)	91 (0.0472)	0.0642	85.34
YOLOv5s						233 (0.0619)	21 (0.0177)	184 (0.0737)	94 (0.0265)	0.0450	112.06
YOLOv5n	0.5	0.5	30	20	0.6	240 (0.0885)	16 (0.0265)	189 (0.1062)	91 (0.0531)	0.0686	86.02
YOLOv5s						230 (0.0531)	21 (0.0177)	188 (0.0737)	95 (0.0295)	0.0435	112.82

To test the evolution of the frames' queue throughout the execution of the system, we put together an 18 min and 7-s video with various situations of people and bicycles passing through at different speeds, as well as some situations where there were people standing still in front of the camera for longer periods of time. To mimic the real-time camera, the video frames were retrieved at 15 FPS. Figure 7 shows the big difference this algorithm made in maintaining real-time performance, since the bigger the frames' queue, the bigger the delay was. This algorithm not only prevented the frames' queue from constantly increasing its size, which eventually led the system to crash at the size of 8199 frames when not using the downsampling algorithm, but it also helped to keep its size very small.

Figure 8 presents a more detailed view of the evolution of the frames' queue when using the downsampling algorithm. We can see some spikes throughout the execution, where the biggest one peaked at 139 frames, meaning that faster objects were detected, resulting in an increase in the frame rate, as opposed to when the queue size was closer to 0.

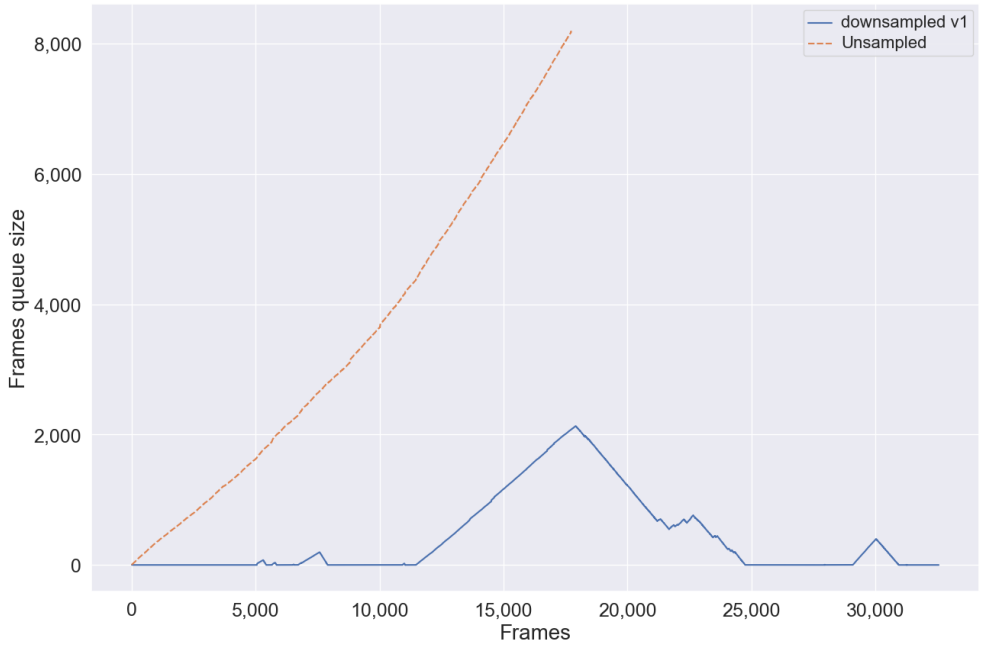

Figure 7. Size of the frame queue with and without the downsampling algorithm.

Since there were some frames being ignored when objects were being detected, the tracker might have had some difficulties tracking some of them—especially the faster ones—due to their distance between frames being bigger. For this reason, the downsampling algorithm increases the frame rate at which the detection module processes these frames when faster objects are detected in order to decrease the impact of this algorithm on the counting error. Table 7 presents and compares the results of object counting with and without using the downsampling algorithm. When analyzing the results, it is possible to see a slight increase in the mean absolute error, as expected, with an error variation of only 0.0022.

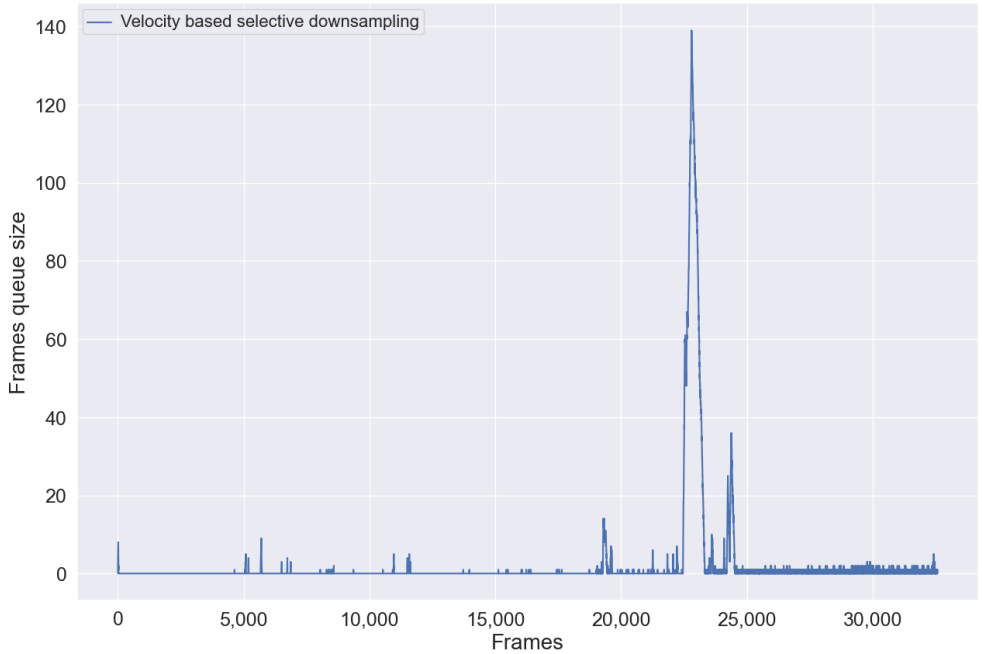

Figure 8. Detailed view of the size of the frames' queue using the downsampling algorithm.

Table 7. Counting results comparison of downsampled vs. unsampled architectures.

Model	Parameters					Left to Right		Right to Left		*MAE*
	σ_l	σ_h	tmin	ttl	nms_agnostic	People (*MAE*)	Bicycles (*MAE*)	People (*MAE*)	Bicycles (*MAE*)	
Expected	-					234	19	177	95	-
YOLOv5s (Unsampled)	0.5	0.5	30	20	0.6	230 (0.0531)	21 (0.0177)	188 (0.0737)	95 (0.0295)	0.0435
YOLOv5s (Downsampled)						230 (0.0530)	20 (0.0147)	187 (0.0884)	92 (0.0265)	0.0457

5. Discussion

A new approach was proposed for counting people and bicycles in an edge AI system using the Jetson Nano board. The V-IOU tracker turned out to be a good approach for the tracking problem, as expected, providing a good solution for the fragmentation and ID switching problem presented in its predecessor by using a visual tracker and still maintaining a very low frame rate. YOLOv5 also showed itself to be a good approach for a small and fast object detector while achieving good *mAP* values for people and bicycles. Using TensorRT to optimize this model specifically for the Jetson Nano set-up improved its performance by approximately 50%.

The most important contribution of the present work was the velocity-based downsampling algorithm, which allowed the system to adjust the frame rate at which the detection module processed the frames and ignore some of them based on the velocity of the objects moving through the frame. This algorithm helps the system maintain real-time performance with heavier detection models without having a big impact on the counting error. This also helps the system reduce energy consumption and therefore operate it on budget hardware. The method was validated on a real-world scenario dataset, achieving a mean absolute error of 0.0457 with downsampling applied and a mean absolute error of 0.0435 without downsampling being applied.

The focus of the present study was to provide a stable solution for running inferences on a Jetson Nano device without any energy restrictions and with high model performance. Too many parameters result in massive memory usage for the model and a long computation time for inference. These are not appropriate to implement into all types of microprocessors or with energy restrictions. The proposed solution can easily run inferences on a Jetson Nano device. The results can also be used as a benchmark for further studies that take into account efficiency regarding energy consumption and also the trade-off between model performance and the computing resources required.

Future work may include the improvement of the *mAP* value of the detection model by training it specifically for people and bicycles or by applying the knowledge distillation method, as Bharadhwaj et al. proposed. The new YOLOv7 architecture [29], which apparently shows some improvements in performance without sacrificing the mAP, may also be tried. The YOLO model may also be adapted to incorporate the changes necessary for edge AI object detection, thus improving the present frame rate or decreasing power needs.

Additional experiments may also be performed in order to determine the maximum and minimum distances of the objects to the camera, as they impact the size of the objects in the images.

Author Contributions: Conceptualization, N.R., N.L. and M.M.; methodology, N.R., N.L. and M.M.; software, H.G. and N.R.; validation, N.R., N.L. and M.M.; data curation, H.G.; writing—original draft preparation, H.G.; writing—review and editing, N.L., M.M. and N.R.; supervision, N.L., M.M. and N.R.; project administration, N.R. All authors have read and agreed to the published version of the manuscript.

Funding: This work received financial support from the Polytechnic Institute of Coimbra within the scope of Regulamento de Apoio à Publicação Científica dos Professores e Investigadores do IPC (Despacho n.º 12598/2020).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fachrie, M. A simple vehicle counting system using deep learning with YOLOv3 model. *J. Resti (Rekayasa Sist. Dan Teknol. Inf.)* **2020**, *4*, 462–468. [CrossRef]
2. Redmon, J.; Farhadi, A. YOLOv3: An incremental improvement. *arXiv* **2018**, arXiv:1804.02767.
3. Bharadhwaj, M.; Ramadurai, G.; Ravindran, B. Detecting Vehicles on the Edge: Knowledge Distillation to Improve Performance in Heterogeneous Road Traffic. In Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition, New Orleans, LA, USA, 1–20 June 2022; pp. 3192–3198.
4. Allen-Zhu, Z.; Li, Y. Towards understanding ensemble, knowledge distillation and self-distillation in deep learning. *arXiv* **2020**, arXiv:2012.09816.
5. Bewley, A.; Ge, Z.; Ott, L.; Ramos, F.; Upcroft, B. Simple online and realtime tracking. In Proceedings of the 2016 IEEE International Conference on Image Processing (ICIP), Phoenix, AZ, USA, 25–28 September 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 3464–3468.
6. OpenDataCam. An Open Source Tool to Quantify the World (Version 3.0.2). 2021. Available online: https://github.com/opendatacam/opendatacam (accessed on 9 November 2022).
7. Bochkovskiy, A.; Wang, C.Y.; Liao, H.Y.M. YOLOv4: Optimal Speed and Accuracy of Object Detection, 2020. *arXiv* **2004**, arXiv:cs.CV/2004.10934.
8. Jiang, Z.; Zhao, L.; Li, S.; Jia, Y. Real-time object detection method based on improved YOLOv4-tiny. *arXiv* **2020**, arXiv:2011.04244.
9. Erik Bochinski, V.E.; Sikora, T. High-Speed Tracking-by-Detection Without Using Image Information. In Proceedings of the International Workshop on Traffic and Street Surveillance for Safety and Security at IEEE AVSS 2017, Lecce, Italy, 19 April 2017.
10. Ultralytics. YOLOv5. 2022. Available online: https://github.com/ultralytics/yolov5 (accessed on 9 November 2022).
11. Bochinski, E.; Senst, T.; Sikora, T. Extending IOU Based Multi-Object Tracking by Visual Information. In Proceedings of the IEEE International Conference on Advanced Video and Signals-Based Surveillance, Auckland, New Zealand, 27–30 November 2018; pp. 441–446.
12. Wojke, N.; Bewley, A.; Paulus, D. Simple online and realtime tracking with a deep association metric. In Proceedings of the 2017 IEEE International Conference on Image Processing (ICIP), Beijing, China, 17–20 September 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 3645–3649.
13. Nvidia. Nvidia Jetson Nano. Available online: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/product-development (accessed on 9 November 2022).
14. Lin, T.Y.; Maire, M.; Belongie, S.; Hays, J.; Perona, P.; Ramanan, D.; Dollár, P.; Zitnick, C.L. Microsoft coco: Common objects in context. In Proceedings of the European Conference on Computer Vision, Zurich, Switzerland, 6–12 September 2014; Springer: New York, NY, USA, 2014; pp. 740–755.
15. Ren, S.; He, K.; Girshick, R.; Sun, J. Faster r-cnn: Towards real-time object detection with region proposal networks. *Adv. Neural Inf. Process. Syst.* **2015**, *28*, 1137–1149. [CrossRef] [PubMed]
16. Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.Y.; Berg, A.C. Ssd: Single shot multibox detector. In Proceedings of the European Conference on Computer Vision, Honolulu, HI, USA, 21–26 July 2016; Springer: New York, NY, USA, 2016; pp. 21–37.
17. Vanholder, H. Efficient inference with tensorrt. In Proceedings of the GPU Technology Conference, Edinburgh, UK, 29 March–1 April 2016; Volume 1, p. 2.
18. Developer, N. TensorRT Open Source Software. 2022. Available online: https://github.com/NVIDIA/TensorRT (accessed on 9 November 2022).
19. Kumar, S.; Sharma, P.; Pal, N. Object tracking and counting in a zone using YOLOv4, DeepSORT and TensorFlow. In Proceedings of the 2021 International Conference on Artificial Intelligence and Smart Systems (ICAIS), Coimbatore, India, 21–25 March 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1017–1022.
20. Oltean, G.; Florea, C.; Orghidan, R.; Oltean, V. Towards real time vehicle counting using yolo-tiny and fast motion estimation. In Proceedings of the 2019 IEEE 25th International Symposium for Design and Technology in Electronic Packaging (SIITME), Cluj-Napoca, Romania, 23–26 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 240–243.
21. Girshick, R.; Donahue, J.; Darrell, T.; Malik, J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Columbus, OH, USA, 23–28 June 2014.
22. Gandhi, R. R-CNN, Fast R-CNN, Faster R-CNN, YOLO—Object Detection Algorithms. 2018. Available online: https://www.datasciencecentral.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms/ (accessed on 9 November 2022).
23. Yelisetty, A. Understanding Fast R-CNN and Faster R-CNN for Object Detection. 2020. Available online: https://towardsdatascience.com/understanding-fast-r-cnn-and-faster-r-cnn-for-object-detection-adbb55653d97 (accessed on 9 November 2022).
24. Targ, S.; Almeida, D.; Lyman, K. Resnet in resnet: Generalizing residual architectures. *arXiv* **2016**, arXiv:1603.08029.

25. Deng, J.; Dong, W.; Socher, R.; Li, L.J.; Li, K.; Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In Proceedings of the 2009 IEEE Conference on Computer Vision and Pattern Recognition, Miami, FL, USA, 20–25 June 2009; IEEE: Piscataway, NJ, USA 2009; pp. 248–255.
26. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You only look once: Unified, real-time object detection. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 779–788.
27. Bochinski, E.; Eiselein, V.; Sikora, T.; Senst, T. Python Implementation of the IOU/V-IOU Tracker. 2022. Available online: https://github.com/bochinski/iou-tracker (accessed on 9 November 2022).
28. BlueMirrors. CVU: Computer Vision Utils. 2022. Available online: https://github.com/BlueMirrors/cvu (accessed on 9 November 2022).
29. Wang, C.Y.; Bochkovskiy, A.; Liao, H.Y.M. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv* **2022**, arXiv:2207.02696.