11

Edge Deployment of Multi-Task Vision Models for Smart City Infrastructures

Carmelo Scribano[1,3], Mohammad Mahdi[2], Filippo Muzzini[1], Nedyalko Prisadnikov[2], Yuqian Fu[2], Micaela Verucchi[1], Ignacio Sanudo Olmedo[1], Danda Pani Paudel[2], and Luc Van Gool[2]

[1]HiPeRT, University of Modena and Reggio Emilia, Italy
[2]INSAIT, Sofia University "St. Kliment Ohridski", Bulgaria
[3]Institute of Informatics and Telematics, National Research Council, Italy

Abstract

The ability to deploy complex vision pipelines at the edge is crucial for next-generation smart city infrastructures. Running inference locally avoids transmitting raw images, thereby reducing bandwidth usage, preserving privacy, and improving system robustness through a distributed paradigm. Achieving this, however, requires models that are both powerful and efficient. In this context, the multi-task paradigm emerges as a key enabler, supporting concurrent prediction across diverse tasks such as object detection, panoptic segmentation, and depth estimation. We introduce a novel pipeline built around a multi-task model explicitly designed to balance expressive power with inference efficiency under resource-constrained conditions. In parallel, we propose an optimization workflow that minimizes inference costs for edge deployment. Experimental results in a real-world setting demonstrate the effectiveness of our approach and establish a strong baseline for future smart city applications.

Keywords: smart city, edge inference, multi-task learning, model compression.

11.1 Introduction

Camera-based smart-city systems leverage computer vision, machine learning, and data anonymization techniques to monitor public areas in real time, detecting and recognizing vehicles and pedestrians. Processing camera streams locally mitigates network bottlenecks and enhances privacy preservation. However, this edge-side processing also introduces significant computational challenges, as modern perception pipelines rely on deep neural networks that are typically resource-intensive. Achieving low-latency inference is crucial to enable real-time interaction between the smart city infrastructure and connected vehicles, thereby extending vehicle perception capabilities with contextual environmental information.

11.1.1 Use Case Definition

The HAura device, previously introduced in [1], is a smart roadside unit designed for safety management and data analytics in urban and industrial environments. The system continuously processes image data from two cameras. The resulting metadata is transmitted to a central server that can implement various urban monitoring and management policies based on the extracted information. At the core of HAura lies the NVIDIA Jetson Orin Nano embedded platform. This platform is a popular choice for edge computer vision applications due to its integrated NVIDIA GPU, which delivers a favorable balance between performance, power efficiency, and programmability within a compact form factor.

11.1.2 Multi-Task Perception Model Deployment

This work focuses on the deployment and optimization of a multi-task perception pipeline on the HAura edge device. The perception model, based on a DINOv2 backbone, jointly performs semantic segmentation, instance segmentation, object detection, and depth estimation within a unified architecture (see Figure 11.1). This paradigm significantly reduces the computational overhead compared to the use of multiple single-task models, leading to a more efficient and scalable deployment on resource-constrained hardware. The primary contribution of this work lies in the integration, adaptation, and optimization of the perception stack for real-time execution on the NVIDIA Jetson Orin Nano platform. The goal is to achieve an end-to-end processing latency below 100ms per frame, including all post-processing stages, while maintaining task performance. The paper details the software

Figure 11.1 Sample outputs for the Multi-task perception pipeline deployed in a smart-city scenario. Right to Left: object detection, instance segmentation, semantic segmentation, and monocular depth estimation.

stack (Section 21.3), runtime configuration (Section 21.4), and performance evaluation methodology used to reach this target on a resource-constrained embedded platform (Section 21.5).

11.2 Related Work

Traffic and urban monitoring have been widely addressed in the literature. The needs, perspectives, and enabling technologies of an urban monitoring system are discussed in [2]. In [3] the authors propose a method for counting vehicles at an intersection using recorded or live video. In [4] a deep neural network is proposed to identify emergencies in the streets. Similarly, in [5] the model recognizes human abnormal activity in an urban area. In [6] computer vision is used to retrieve changes in the urban environment.

The cited approaches require huge computation or a centralized server. This scales poorly, as the number of monitoring areas in a city can be very large, eventually reaching computational and latency limits. A distributed approach, in which each intersection has its own embedded system, can overcome this problem. To achieve this situation is important to use smart cameras or sensors equipped with an embedded GPU, as in [1]. In this work, we show the trade-off between the accuracy and latencies on an embedded system for a model that addresses most of the tasks required in traffic and urban monitoring.

11.2.1 Vision Models for Urban Monitoring

In the considered scenario, multi-task models are the most promising solution [7]. They leverage the Vision Transformers (ViTs) [8, 9] that can extract rich features from images. One of the most popular models that use ViTs

is Dino-v2 [10, 11]. Using Dino-v2 makes it possible to train a multitask model since it was used in different tasks such as Object Detection [12, 13] or Segmentation [14]. In this work, we show that the multitask models can be used on an embedded system for urban monitoring, and we investigate how to reduce the latencies of these models to meet the real-time requirements.

11.3 Model Description

The multi-task perception model analyzed in this work is derived from an extension of the architecture presented in [15]. This architecture combines the powerful Dino-V2 foundation transformer as a shared feature extractor with a shallow upscaling decoder and per-task projection layers to produce the task-specific outputs. The rationale behind this design is multifold: leveraging Dino-v2 as the shared deep feature encoder provides a strong feature representation that has been extensively shown to achieve state-of-the-art performance on several downstream tasks. The shared shallow decoder upscales the patch-level Dino-V2 embeddings to pixel-level, keeping the projection shared across the tasks. Finally, the per-task projection is used to map the shared pixel-space to produce the correct feature dimension required by each task (see Figure 11.2).

Specifically, the Dino-V2 encoder is based on the ViT architecture [16] with a patch size of 14×14, the authors provide four different architectural variations which are compared in Table 11.1. The model analyzed in this paper is based on the Large backbone, which was chosen for its superior performance. The analysis presented can easily be extended to different

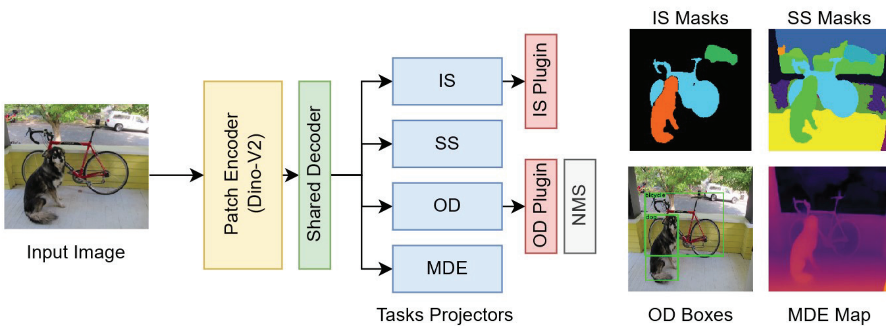

Figure 11.2 Schematized architecture of the considered Multi-modal model.

Table 11.1 DINOv2 model variants and core architecture specs

Variant	Architecture	Emb dim	# Parameters	# Attention heads	# Blocks
SMALL	ViT-S/14	384	21M	6	12
BASE	ViT-B/14	768	86M	12	12
LARGE	ViT-L/14	1024	300M	16	24
GIANT	ViT-L/14	1536	1,100M	24	40

variants to satisfy different latency-performance tradeoffs. Given an input image $I \in \mathbb{R}^{3\times H\times W}$ each encoder block produces an intermediate output $z_b \in \mathbb{R}^{\frac{H}{p}\times\frac{W}{p}\times d}$ there p denotes the patch size and d denotes the embedding dimension.

The upscaling decoder takes as input the intermediate patch-level embeddings Dino-V2 ViT blocks, concatenated along the embedding dimension $z_c = [z_{b1}|z_{b_2}|z_{b3}|z_{b4}] \in \mathbb{R}^{\frac{H}{p}\times\frac{W}{p}\times 4d}$. First, it projects to the decoder embedding dimension e with a 1×1 convolution, then it gradually up-scales to the pixel-space with four layers of transposed convolution. This yields an $8\times$ upscaling of the patch space, which corresponds to $\frac{8}{p}$ of the original input resolution.

11.3.1 Multi-Task Design

The output of the upscaling decoder, described in the previous section, is used as input by the Task Projectors. The considered Task Projectors are: **Object detection (OD)**, **Instance Segmentation (IS)**, **Semantic Segmentation (SS)**, and **Monocular Depth estimation (MDE)**. All these tasks are trained simultaneously using labels from the COCO dataset [17], except MDE, which is pseudo-labeled on COCO images, leveraging state-of-the-art zero-shot depth estimation DepthAnything-V2 [18]. The design of the task-specific projectors is detailed hereafter.

Semantic Segmentation: The SS task is cast as per-pixel multi-class classification. The projection for this task is a single 3×3 convolution to project from the decoder embedding dimension e to the number of semantic classes C_S. The semantic class is trivially decoded with a per-pixel argmax operation.

Instance Segmentation: Conversely, IS uses the most complex encoding, proposed in [15]. The IS projection head produces an output $O_{IS} \in \mathbb{R}^{H_O\times W_O\times 4L}$, each location (i, j) regresses the coordinate of the mask centroids it belongs to, or a void centroid for unlabeled regions. Denoting as $O(u, v)$ the set of pixel locations belonging to the instance mask with centroid

(i, j), the regression target is formalized as:

$$O_{IS}(i,j) = \begin{cases} \text{concat}\left(\gamma(u), \gamma(v)\right) & \text{if} \quad (i,j) \in O(u,v) \\ \overrightarrow{0} \in \mathbb{R}^{4L} & \text{otherwise} \end{cases} \tag{21.1}$$

where the positional encoding $\gamma(p)$ derived from [19] is defined as:

$$\gamma(p) = \left(sin\left(2^l \pi p\right), cos\left(2^l \pi p\right)\right)_{l=0}^{L-1} \tag{22.2}$$

This encoding, while achieving strong performance on the task, poses significant challenges for deployment. Post-processing O_{IS} to obtain the decoded instance mask poses a substantial overhead on end-to-end execution latency. In the next section, we detail the implementation of an efficient CUDA-accelerated post-processing stage that can be built in the TensorRT inference engine, achieving substantial speedup over the baseline Pytorch implementation.

Object Detection: The object detection is based on anchor-free YoLo [20]. The input image is mapped to a $S \times S$ grid, at each grid location B candidate bounding boxes are regressed, each encoded as (c_x, c_y, w, h, p) with (c_x, c_y) denoting the normalized center point coordinates expressed in cell-relative coordinates, (w, h) box width and height normalized with respect to the image dimensions and s a confidence score. In addition, a single C_o class classification is regressed at each cell location. Therefore, the OD projection produces an output:

$$O_{OD} \in \mathbb{R}^{S \times S \times (5B+C)} \tag{22.3}$$

Decoding YOLO-style predictions requires mapping grid-relative box coordinates to image coordinates, followed by Non-Maximum Suppression (NMS) to remove overlaps. While TensorRT's INMSLayer efficiently handles NMS, it lacks the preceding coordinate transformation. To enable end-to-end inference without breaking the TensorRT execution graph, we implement a custom CUDA plugin that performs on-device box decoding and filtering, keeping all postprocessing within the optimized graph.

Monocular Depth Estimation: The depth estimation is cast as a raw per-pixel regression problem. In this case, no additional post-processing is required. Similar to SS, the MDE projection simply up-scales the decoder features to the output resolution, producing the single-channel depth map.

11.4 Deployment

Our target application involves distributed smart city monitoring, which imposes two key constraints: the model must run on embedded hardware and operate in real time. Achieving these requirements depends on an appropriate deployment configuration, including the backbone size, input resolution, batch size, and arithmetic precision (FP32, FP16, or INT8). Each of these parameters directly affects both accuracy and task performance. A critical step is defining a latency deadline: the maximum acceptable inference time per frame. In practice, processing results that arrive too late (e.g., one second after capture) become irrelevant, as the scene has already changed. The deadline should not exceed the camera's frame period; otherwise, the system remains idle between frames. For our use case, we adopt a 100ms deadline, which ensures timely scene updates without perceptible information loss. Driven by this observation, in the remainder of this section we detail the main technical solution experimented with to meet the latency constraint. Later, Section 21.5 presents experimental results used to identify the optimal deployment settings

11.4.1 Mixed-Precision Inference

Out of the box, TensorRT supports mixed-precision inference with FP32, FP16, and INT8 arithmetic, all natively accelerated on the NVIDIA Orin Nano via Ampere Tensor Cores. In addition, the TensorRT model compiler applies aggressive graph-level optimizations; in our case, self-attention layers are automatically fused and replaced with FlashAttention-V2 [21] at FP16 and INT8 precision, yielding more efficient memory access and higher throughput. While FP16 arithmetic can be enabled without additional steps, INT8 arithmetic requires an additional calibration stage to determine the scaling values that map floating-point activations and weights into the discrete 8-bit integer range.

Post Training Quantization: TensorRT's explicit quantization workflow represents INT8 computation using Quantize (Q) and Dequantize (DQ) operator pairs inserted directly into the network graph, explicitly defining the quantization boundaries for each tensor. During Post-Training Quantization (PTQ), calibration data is used to collect activation statistics and determine the scaling factors that map FP32 tensors into the INT8 domain, typically through entropy or percentile-based range estimation. Weight tensors are quantized

offline, while activation quantization parameters are computed dynamically during calibration. For this work, we leverage the Post Training Quantization (PTQ) pipeline provided by Nvidia TensorRT-Model-Optimizer (ModelOpt); specifically, leveraging ModelOpt's implementation of the SmoothQuant calibration method [22] which redistributes activation ranges into the corresponding weights to mitigate outliers and improve quantization robustness. We employ per-channel quantization for weights and per-tensor quantization for activations. In Section 21.5, we compare the performance-speed tradeoff under different arithmetic.

11.4.2 Processors Plugins

Instance Segmentation: Since the entire model, from input to output, is subject to real-time constraints, we accelerated the processors using Nvidia CUDA. The encoded output of this task is defined in Eq. 1 each element $O_{IS}(i,j)$ stores a tuple representing an estimate of the coordinates of the centroid (i.e. the mask) associated with pixel (i,j). The proximity between this tuple and the actual centroid coordinates serves as a confidence score: the closer the value, the higher the likelihood that the corresponding point is a true centroid.

To compute these scores, we calculate the distance between each pixel and every potential centroid. This is performed by launching a CUDA kernel in which each thread is associated with a pixel of O_{IS} and a candidate centroid. Each thread computes the two-dimensional distance and stores the corresponding score derived from this distance. Since multiple pixels may contribute to the same centroid, their scores are accumulated. To minimize memory accesses, we exploit CUDA shared memory and the `syncthreads()` primitive to coordinate partial sums efficiently. Next, a second kernel is launched in which each candidate centroid is assigned to a thread. Each thread examines neighboring centroids and, if it holds the highest accumulated score, it designates itself as the centroid of a distinct mask. Finally, a third kernel is executed, where each thread corresponds to a pixel–mask pair. If a pixel is associated with a specific mask (recall that, as defined in Eq. 1, some pixels may correspond to a null mask), the thread writes a value of 1 at the corresponding position for that mask.

Since all these phases operate at the pixel level, they can be executed in parallel, resulting in significant speedup. Some operations require coordination between neighboring pixels to prevent race conditions, but the

use of shared memory and the `syncthreads()` primitive effectively reduces synchronization overhead.

Object Detection: To post-process the object detection output defined in Eq. 3, a single CUDA kernel is employed. Each thread is assigned to a cell of size $S \times S$ and selects the optimal bounding box among the available candidates based on the corresponding confidence score. The thread then computes the bounding box coordinates using the parameters (c_x, c_y, w, h) of the selected box. Similarly, the object class is determined according to the class confidence score, which is also stored in the output.

The final result consists of three arrays: (i) the bounding box coordinates, (ii) the associated class for each box, and (iii) the corresponding class confidence values. The CUDA implementation significantly accelerates this computation, as each cell is processed independently, enabling full parallelization across threads.

11.5 Experiments

11.5.1 Experimental Setup

Performance Assessment: To comprehensively evaluate the effectiveness of our multi-task model, we assess the performance of each task-specific head using standard benchmarks commonly adopted in the literature: mean Average Precision at 0.5 IoU threshold (**mAP@50**) [23] for object detection, mean Intersection over Union (**mIoU**) [24] for semantic segmentation, Panoptic Quality (**PQ**) [25] for instance segmentation, and Root Mean Square Error (**RMSE**) [26] depth estimation. We report task metrics on the COCO validation split, using the pseudo-labels for the MDE task.

Benchmarking: The multi-task model is implemented in PyTorch, following the standard deployment pipeline of exporting the network to the ONNX format via JIT tracing. This export process preserves the Quantize/Dequantize (Q/DQ) operators introduced during Post-Training Quantization (PTQ), as well as any custom CUDA plugins, which are automatically recognized and integrated by the TensorRT builder with their corresponding device-level implementations. All latency and memory measurements are collected on an NVIDIA Jetson Orin Nano running JetPack 6.1, which includes TensorRT 10.3 and CUDA 12.x. Benchmark results are obtained under identical inference conditions across precision modes (FP32, FP16, and INT8) to ensure fair comparison of performance and efficiency. Latency measurements

are obtained using the TensorRT trtexec utility with real-time scheduling to ensure deterministic timing:

```
$ trtexec –loadEngine=model.trt –useCudaGraph –noDataTransfers –useSpinWait
–iterations=100 –avgRuns=100 –exportTimes=measure.json
```

The reported latency corresponds to end-to-end inference time, averaged over multiple runs after warm-up. Memory footprint is assessed using the TensorRT Python inference API, implementing a custom CUDA memory allocator to track dynamic allocations and measure peak device memory consumption during inference. All benchmarks are performed under identical runtime conditions across precision modes (FP32, FP16, and INT8) to ensure fair and reproducible comparison.

11.5.2 Post-Processing Acceleration

First, we discuss the impact of the proposed implementation on the efficient decoding of the Instance Segmentation and Object Detection outputs. To this end, we benchmark and compare only the post-processing layer, comparing side-by-side the TensorRT and reference Pytorch implementations.

Instance Segmentation: Instance segmentation represents the most computationally demanding component of the multi-task pipeline, primarily due to the high cost of mask post-processing. We evaluate two implementations under two feature map resolutions: $192{\times}192$ (corresponding to an image resolution of $336{\times}336$) and $256{\times}256$ (corresponding to an input resolution of $448{\times}448$). By default, the proposed TensorRT plugin aggregates centroid candidates over a coarse grid of $32{\times}32$ buckets and explicitly decodes all 1024 candidate masks. In contrast, the baseline PyTorch implementation accumulates candidates on a finer $80{\times}80$ grid but decodes only a subset of selected masks. As summarized in Table 11.2, at an input resolution of ($336{\times}336$), the plugin achieves $\sim 3\times$ speedup when using a coarser bucket grid $32{\times}32$ and a $\sim 2.5\times$ speedup when both implementations use the same grid configuration. While the plugin exhibits a higher memory footprint, caused by the CUDA buffer allocations, it substantially reduces inference latency, demonstrating the benefit of GPU-resident post-processing for instance segmentation.

Object Detection: We perform a similar evaluation for the post-processing stage of the object detection task (see Table 11.3). compares the PyTorch baseline with our custom TensorRT plugin implementation, including both cases with and without Non-Maximum Suppression (NMS). The TensorRT

Table 11.2 Benchmarking of Instance Segmentation Post-processor

Framework	Buckets	Feature res	Infer (ms)	Memory (MB)
Pytorch	80×80	192	134.39	30.72
		256	150.85	50.04
	32×32	192	100.70	29.82
		256	117.28	48.98
Plugin	32×32	192	34.89	162.81
		256	62.07	289.42

Table 11.3 Benchmarking of Object Detection Post-processor

Framework	NMS	Infer (ms)	Memory (MB)
Pytorch	No	4.014	0.029
	Yes	43.22	8.17
Plugin	No	0.017	0.02
	Yes	0.451	0.03

version leverages the optimized INMSLayer for NMS execution, which is fully integrated into the inference graph. As shown in Table 11.3, the plugin achieves a substantial latency reduction compared to the PyTorch implementation, yielding up to two orders of magnitude speedup in both configurations. When NMS is enabled, the TensorRT INMSLayer adds minimal overhead (0.451 ms vs. 0.017 ms), whereas the PyTorch implementation incurs a significant slowdown due to host-side execution. Memory usage remains nearly constant across settings, confirming that the proposed plugin and INMSLayer enable fully GPU-resident post-processing with negligible additional footprint.

11.5.3 Performance Assessment

We conducted different experiments varying: (a) The input resolution (b) The precision format (FP32, FP16, INT8) (c) post-processors implementation. We measured both the task metrics (mAP@50, mIoU, PQ, RMSE) and their runtime performance and memory footprints. The results, summarized in Table 11.4, show that lower precision formats (FP16 and INT8) introduce negligible degradation in the task metrics while offering substantial reductions in inference latency.

For the Object detection tasks (mAP@50 metric) and Instance Segmentation (PQ metric), we compare PyTorch-based post-processing (reported in braces) with custom CUDA-optimized runtime plugins. Interestingly, the mAP@50 consistently improves when using the efficient decoding implementation. Conversely, the PQ metric shows a slight degradation with

Table 11.4 Multi-task model performance assessment under different configurations of input resolution and arithmetic precision. map@50 and PQ are evaluated both using the proposed post-processors plugin and with the Pytorch reference implementation (value in brackets).

Res.	Precision	mAP@50	mIoU	PQ	RMSE	Infer (ms)	Mem (MB)
336	FP32	23.22 (17.28)	63.65	43.27 (46.62)	0.56	272.0 (377.8)	1585.7
	FP16	23.21 (17.35)	63.65	43.26 (46.63)	0.56	126.3 (232.7)	864.4
	INT8	22.58 (16.99)	63.60	42.94 (46.41)	0.56	112.0 (218.3)	556.0
448	FP32	35.54 (26.80)	64.35	44.06 (46.18)	0.47	544.0 (638.8)	1795.9
	FP16	35.58 (26.73)	64.35	44.06 (46.19)	0.47	225.8 (322.0)	1029.8
	INT8	35.53 (26.05)	64.10	43.86 (46.02)	0.47	200.0 (295.3)	725.8

the efficient post-processing, likely due to the reduced number of accumulation buckets (32×32 compared to 80×80), which limits the precision of instance-level mask aggregation.

11.5.4 End-To-End Results

The analyzed optimizations collectively nearly enable end-to-end perception pipeline execution under 100ms per frame at a resolution of 336×336, demonstrating promising results for fulfilling the real-time processing requirement for deployment on the HAura platform. In Figure 11.3 we

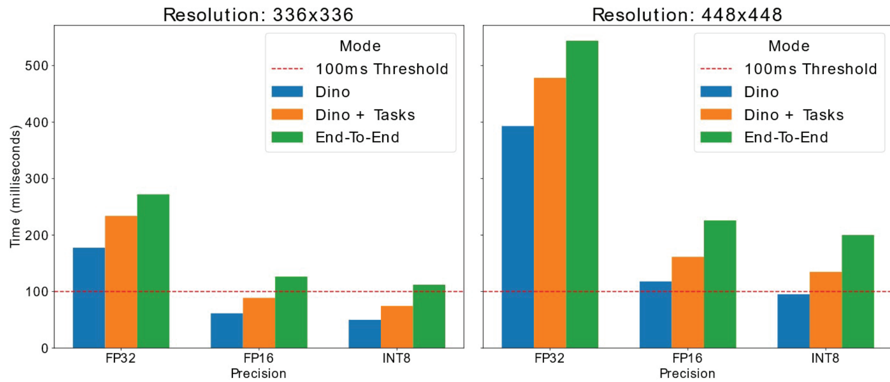

Figure 11.3 Latency breakdown of the perception pipeline for backbone only ("Dino"), full model ("Dino+Tasks"), and end-to-end execution.

further dissect the source of latency, comparing the execution of the backbone alone (“Dino”), the complete model of decoders and task projectors (“Dino+Tasks”), and the end-to-end execution including post-processing (“End-To-End”). In the most optimized INT8 configuration, post-processing time accounts for approximately 1/3 of the end-to-end execution time. Considering this, future developments will focus further on improving decoding efficiency, especially for the task of instance segmentation.

11.6 Conclusions

In this paper, we investigated the deployment of a multi-task deep neural network in an urban monitoring scenario. We first discussed how system-level constraints, such as real-time deadlines and the number of video streams to process, affect the overall design and performance. We then identified several tunable parameters that can be adjusted to meet timing requirements. Furthermore, we proposed an accelerated implementation of the post-processing modules used in multi-task models, which significantly improves end-to-end latency.

Our experimental evaluation demonstrated how the selected parameters influence overall performance, confirming that optimizing post-processing can yield substantial latency reductions. As future work, we plan to further enhance the post-processor implementation and investigate additional tuning parameters, such as batch size, to generalize our approach and meet a wider range of system requirements.

Acknowledgements

This research was partially funded by the dAIedge project (HORIZON-CL4-2022-HUMAN-02-02, Grant Agreement Number: 101120726) and the Ministry of Education and Science of Bulgaria (support for INSAIT, part of the Bulgarian National Roadmap for Research Infrastructure). C. Scribano work was partly funded by the Partenariato Esteso PE00000013 - “FAIR”, funded by the European Commission under the NextGeneration EU program, PNRR - M4C2 Investimento 1.3.

References

[1] C. Scribano, I. Sanudo Olmedo, M. Verucchi, M. Bertogna e others, ≪ On-the-Edge Inference Enabled Vision System for Smart Cities, ≫

in SMART 2025, The Fourteenth International Conference on Smart Cities, Systems, Devices and Technologies, 2025.

[2] C. Scribano e F. Muzzini, ≪ Emergency vehicles in the Smart Cities: challenges and enabling technologies, ≫ in 2024 IEEE Symposium on Computers and Communications (ISCC), 2024.

[3] G. M. Lingani, D. B. Rawat e M. Garuba, ≪ Smart traffic management system using deep learning for smart city applications, ≫ in 2019 IEEE 9th annual computing and communication workshop and conference (CCWC), 2019.

[4] N. T. Ha, N. P. D. Tuan, T. H. Khanh, L. T. Du, T. C. Dinh, D. N. Sang, N. Van Son, N. L. D. Luan e N. T. Le, ≪ Leveraging Deep Learning Model for Emergency Situations Detection on Urban Road Using Images from CCTV Cameras, ≫ in 2022 International Conference on Engineering and Emerging Technologies (ICEET), 2022.

[5] R. Nouisser, S. K. Jarraya e M. Hammami, ≪ A Review of Vision-based Abnormal Human Activity Analysis for Elderly Emergency Detection, ≫ in 2023 International Conference on Innovations in Intelligent Systems and Applications (INISTA), 2023.

[6] N. Naik, S. D. Kominers, R. Raskar, E. L. Glaeser e C. A. Hidalgo, ≪ Computer vision uncovers predictors of physical urban change, ≫ Proceedings of the National Academy of Sciences, vol. 114, p. 7571–7576, 2017.

[7] K.-H. Thung e C.-Y. Wee, ≪ A brief review on multi-task learning, ≫ Multimedia Tools and Applications, vol. 77, p. 29705–29725, 2018.

[8] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan e M. Shah, ≪ Transformers in vision: A survey, ≫ ACM computing surveys (CSUR), vol. 54, p. 1–41, 2022.

[9] R. Ranftl, A. Bochkovskiy e V. Koltun, ≪ Vision transformers for dense prediction, ≫ in Proceedings of the IEEE/CVF international conference on computer vision, 2021.

[10] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski e A. Joulin, ≪ Emerging properties in self-supervised vision transformers, ≫ in Proceedings of the IEEE/CVF international conference on computer vision, 2021.

[11] M. Oquab, T. Darcet, T. Moutakanni, H. V. Vo, M. Szafraniec, V. Khalidov, P. Fernandez, D. HAZIZA, F. Massa, A. El-Nouby, M. Assran, N. Ballas, W. Galuba, R. Howes, P.-Y. Huang, S.-W. Li, I. Misra, M. Rabbat, V. Sharma, G. Synnaeve, H. Xu, H. Jegou, J. Mairal, P. Labatut, A. Joulin e P. Bojanowski, ≪ DINOv2: Learning Robust Visual Features

without Supervision, ≫ Transactions on Machine Learning Research, 2024.

[12] S. Liu, Z. Zeng, T. Ren, F. Li, H. Zhang, J. Yang, Q. Jiang, C. Li, J. Yang, H. Su e others, ≪ Grounding dino: Marrying dino with grounded pre-training for open-set object detection, ≫ in European conference on computer vision, 2024.

[13] T. Ren, Q. Jiang, S. Liu, Z. Zeng, W. Liu, H. Gao, H. Huang, Z. Ma, X. Jiang, Y. Chen e others, ≪ Grounding dino 1.5: Advance the" edge" of open-set object detection, ≫ arXiv preprint arXiv:2405.10300, 2024.

[14] T. Ren, Y. Chen, Q. Jiang, Z. Zeng, Y. Xiong, W. Liu, Z. Ma, J. Shen, Y. Gao, X. Jiang e others, ≪ Dino-x: A unified vision model for open-world object detection and understanding, ≫ arXiv preprint arXiv:2411.14347, 2024.

[15] N. Prisadnikov, W. Van Gansbeke, D. P. Paudel e L. Van Gool, ≪ A simple and generalist approach for panoptic segmentation, ≫ arXiv preprint arXiv:2408.16504, 2024.

[16] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit e N. Houlsby, ≪ An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale, ≫ ICLR, 2021.

[17] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár e C. L. Zitnick, ≪ Microsoft coco: Common objects in context, ≫ in European conference on computer vision, 2014.

[18] L. Yang, B. Kang, Z. Huang, Z. Zhao, X. Xu, J. Feng e H. Zhao, ≪ Depth anything v2, ≫ Advances in Neural Information Processing Systems, vol. 37, p. 21875–21911, 2024.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser e I. Polosukhin, ≪ Attention is all you need, ≫ Advances in neural information processing systems, vol. 30, 2017.

[20] J. Redmon, S. Divvala, R. Girshick e A. Farhadi, ≪ You only look once: Unified, real-time object detection, ≫ in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016.

[21] T. Dao, ≪ FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning, ≫ in International Conference on Learning Representations (ICLR), 2024.

[22] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth e S. Han, ≪ Smoothquant: Accurate and efficient post-training quantization for large language models, ≫ in International conference on machine learning, 2023.

[23] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn e A. Zisserman, ≪ The pascal visual object classes (voc) challenge, ≫ International journal of computer vision, vol. 88, p. 303–338, 2010.
[24] A. Garcia-Garcia, S. Orts-Escolano, S. Oprea, V. Villena-Martinez e J. Garcia-Rodriguez, ≪ A review on deep learning techniques applied to semantic segmentation, ≫ arXiv preprint arXiv:1704.06857, 2017.
[25] A. Kirillov, K. He, R. Girshick, C. Rother e P. Dollár, ≪ Panoptic segmentation, ≫ in Proceedings of the IEEE/CVF conference on computer vision and pattern recognition, 2019.
[26] D. Eigen, C. Puhrsch e R. Fergus, ≪ Depth map prediction from a single image using a multi-scale deep network, ≫ Advances in neural information processing systems, vol. 27, 2014.