Assessment #5

Research Assessment

Name: Aryan Patel

Date: October 16, 2026

Subject: Machine Learning

MLA Citation:

Martínez Otero, Pablo, et al. "Performance Analysis of the YOLO Object Detection Algorithm in Embedded Systems: Generated Code vs. Native Implementation." *Computation*, vol. 14, no. 3, 12 Mar. 2026, article 67, https://doi.org/10.3390/computation14030067.

Analysis:

The authors ran the same pretrained YOLOv8 and YOLOv9 weights two ways on NVIDIA Jetson Orin Nano and AGX Orin boards: through the standard Ultralytics and PyTorch pipeline, which they call native, and through a MATLAB and Simulink code generation pipeline. They tested on 400 frames from one simulated driving sequence in CARLA with a single object class, the traffic light. The native pipeline gave better accuracy and a better balance of accuracy and speed overall, and the generated code fell further behind as the models got larger. The authors say this is a controlled micro-benchmark and not a validation of a real driving system.

What I keep coming back to is that the weights never changed. Both pipelines used the same COCO-pretrained models at 640 by 640 pixels, so every difference came from how the models were run. In Assessment #3 I wrote that a detector and a deployed system are different things because mAP never sees the tracking and smoothing around it. This paper moves that gap earlier in the chain, since the detector alone gives different answers depending on the toolchain that runs it. The confusion matrices in Tables 4 and 5 show where the accuracy went. For

YOLOv9s, both pipelines matched 188 of the 254 labeled traffic lights at an intersection over union (IoU) threshold of 0.5, with the same 18 false positives and 66 misses. At 0.8 the native pipeline still had 145 correct boxes and the generated code had 72. At 0.9 it was 43 against 5. The generated code found just as many objects but placed the boxes less precisely, and an averaged mAP would hide that.

This applies directly to my own project. In Assessment #3 I said I would fill in the evaluation data section of a model card based on the Mitchell and co-authors paper and record the split I used, but I was only thinking about the data. Table 2 of this paper is a reproducibility checklist that lists the confidence threshold (0.3), batch size (1), precision (FP32), the board's power mode, and the JetPack and PyTorch versions. Any of those could change my numbers. A model card that leaves them out describes a model nobody else could rebuild, so mine needs a deployment section next to the data section.

I would split the results into accuracy, latency, and startup time. On accuracy, the native pipeline beat the generated code on all eight models that ran in both, by roughly 0.05 to 0.10 mAP@0.5:0.95 (Tables 6 and 7). Latency depended on size. On the AGX, generated code was more than twice as fast for the smallest model (YOLOv9t at 19.94 ms against 43.08 ms) and tied on YOLOv8n (20.71 against 20.96). On the Nano, the largest models ran about five times slower in generated code, with YOLOv8l at 730.50 ms against 142.72 ms. Startup was the number I did not expect to care about. YOLOv8n took about 2.6 seconds to initialize natively on the Nano and 0.12 seconds in generated code, and for YOLOv8l it was 5.7 seconds against 0.69 seconds. The tables report this, but the text never discusses it. So the abstract's claim that the automated pipeline is less competitive in both latency and accuracy holds for the bigger models and not for

every row. For a robot that restarts its vision process, startup could matter more than a few milliseconds per frame.

This changed two things I believed. After Assessment #3 I thought the evaluation protocol decided which architecture won, and now the pipeline joins the protocol. Among the eight shared models, the best one was YOLOv9s natively (0.5108) but YOLOv8m in generated code (0.4528). I also assumed bigger models were slower. On the AGX, native YOLOv9m ran in 35.94 ms, faster than YOLOv9t at 43.08 ms and YOLOv9s at 44.50 ms, and the authors can only guess why. I would not lean hard on the ranking anyway, because the top four native scores sit within 0.0098 of each other. The most surprising part was that the fastest result in the paper, YOLOv9t at 19.94 ms, comes from the least accurate model (0.2585), so speed alone told me nothing.

I have doubts about how far these results go. The test set is 400 consecutive frames from a single simulated sequence, the tables add up to only 254 labeled boxes, and no confidence interval is given for mAP (the authors say a bootstrap would be needed and leave it out). The frames are correlated, so even the latency intervals are called approximate. That is the parking lot problem from Assessment #3 again, except here it shrinks the effective sample size instead of leaking into training. There is also a confound. Each pipeline used its own framework's default pre-processing and non-maximum suppression settings, so I cannot tell whether the accuracy gap comes from code generation itself or from different defaults. Two details looked off as well. The text says native YOLOv8n is about 180% slower on the Nano than on the AGX and generated YOLOv9t about 315% slower, but Tables 8 to 11 give 36.64 ms against 20.96 ms and 63.00 ms against 19.94 ms. Those are about 75% and 216% slower, so ratios seem to have been written as percentages. The paper also calls the Nano unsuitable for real time without stating a time budget,

even though native YOLOv8n runs at about 27 frames per second there. None of this changes the main result, but it reminded me to recompute numbers before I repeat them.

My plan has three parts. When I run the two-split experiment from Assessment #3, I will score each model at an IoU of 0.5 and again at 0.8 instead of reporting only the averaged mAP, since this paper showed a gap can hide at the strict thresholds. I will bootstrap my test images to get a confidence interval, so I know whether a difference of 0.01 means anything. Then I will write the deployment section of my model card using Table 2 as the template. I still have questions. Would TensorRT or FP16 change the comparison? Table 2 lists FP32, and I did not find TensorRT mentioned anywhere in the paper. Does the 0.8 IoU gap show up on real camera footage, or is it a product of a clean simulator? And what time budget counts as real time for my own vision project, since the paper never defines one?

Annotated article: Annotated Article - Assessment #5.pdf