Real-Time Object Detection and On-Track Verification for Autonomous Toy Cars*

Manda Andriamaromanana
Université Paris-Saclay
France
manda.andriamaromanana@universite-paris-saclay.fr

Abstract

This project presents a real-time object detection and tracking system designed to provide input to a reinforcement learning agent controlling a toy car. A YOLOv5 model is trained on custom Mario/Yoshi data and optimized for deployment using ONNX and NCNN. The system includes centroid tracking and a mask-based on-track verification module using HSV thresholding. While initial inference was tested on a Raspberry Pi 4, performance constraints necessitated a switch to a GPU-based platform. The pipeline supports modular integration for future reinforcement learning applications.

1 Introduction

Designing vision pipelines for embedded autonomous systems remains a central challenge in robotics. In this project, we build a system to detect, track, and validate whether a toy vehicle remains within a race track, using real-time video input. Intended as a perception module for an RL-based autonomous controller, the system relies on deep learning for object detection and classical computer vision for path validation. While the Raspberry Pi 4 was used for early-stage experimentation, final integration and testing took place on a GPU desktop.

2 Project Development and Implementation

2.1 Exploring State-of-the-Art Methods

The project began with an exploration of modern computer vision tasks such as classification, segmentation, and object detection. In reviewing the literature and open-source community tools, I focused on well-established architectures like Faster R-CNN, SSD, and the YOLO family. YOLOv5, in particular, stood out for its strong performance and modularity, as well as its seamless export to ONNX—key for embedded deployment.

2.2 First Experimentations and Raspberry Pi Limitations

Initial experiments were conducted using OpenCV on a Raspberry Pi 4 to assess real-time video capture and display. Performance was disappointing—basic display operations ran below 10 FPS, and the addition of image processing steps like color masking or object detection made it unusable for live inference. This experience confirmed that additional optimization or a fallback plan would be necessary.

Figure 1: The two toy cars (Mario and Yoshi) used for training.

2.3 Dataset Creation and Training

The system required a custom dataset. I recorded a short video of Mario and Yoshi toy figures on a

*TER report for Master 1 ISD, Université Paris-Saclay

racetrack and extracted every other frame, producing around 400 samples. Annotation was done via Roboflow, which streamlined dataset formatting, augmentation, and export to YOLOv5.

The initial training yielded good results on the training data, but failed on unseen clips—showing overfitting. To improve generalization, I expanded the dataset with additional clips recorded in different lighting and backgrounds. Subsequent training rounds resulted in a much more robust model.

2.4 Training Configuration and Hyperparameters

The YOLOv5 model used in this project was trained over **50 epochs** on a dataset of around 400 annotated frames, using a batch size of **16** and image resolution fixed at **640×640 pixels**. The training process was carried out using the **SGD optimizer**, with an initial learning rate (`lr0`) of **0.01**, and a learning rate factor (`lrf`) of **0.01** for cosine decay.

To enhance generalization, several data augmentation strategies were applied:

- **HSV-based color jittering**: `hsv-h = 0.015`, `hsv-s = 0.7`, `hsv-v = 0.4`
- **Horizontal flipping**: applied with a probability of `0.5`
- **Translation and scaling**: `translate = 0.1`, `scale = 0.5`

The warmup phase lasted **3 epochs**, using `warmup-bias-lr = 0.1` and `warmup-momentum = 0.8` to stabilize early training dynamics. **Weight decay** was set to `5e-4`, and the momentum parameter to `0.937`.

The final model was trained with the `yolov5n.pt` backbone—a lightweight version suitable for real-time inference—before conversion to `ONNX` and optimization with `NCNN` for embedded use.

The data configuration was managed through a custom `data.yaml` file, and model checkpoints were stored under `runs/train/exp5`.

2.5 Inference on Raspberry Pi and Transition to GPU

With the model trained, I converted it to ONNX and tested inference using ONNX Runtime and NCNN. While NCNN reduced latency significantly, inference on Raspberry Pi was still slow and unsuitable for real-time. Frame skipping and resolution reduction helped, but not enough. The development focus shifted to a desktop with GPU, enabling 30+ FPS inference and full-resolution overlays.

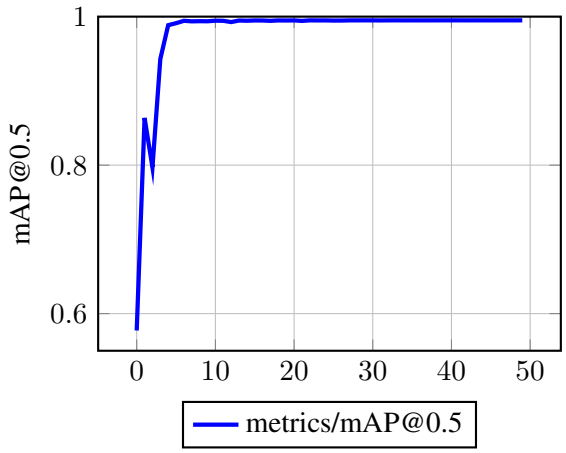

Figure 2: Evolution of mAP@0.5 over 50 epochs

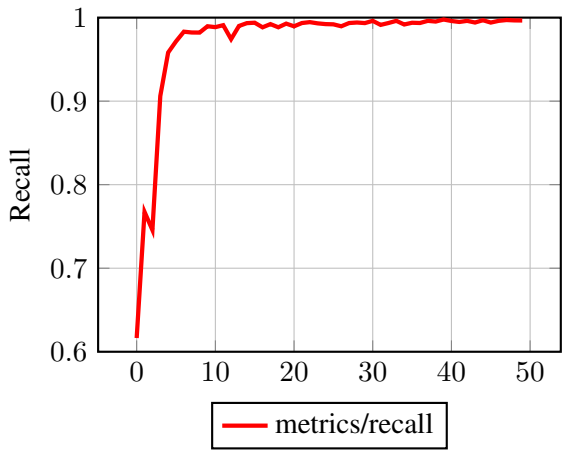

Figure 3: Evolution of recall over 50 epochs

2.6 Tracking and Final System Integration

A critical challenge was to determine whether the object (e.g., Yoshi) remained within the race track throughout its motion. To approximate the valid track area, a dark zone detection module was developed using HSV thresholding. Specifically, the frame is converted to the HSV color space and thresholded to extract regions of low brightness and saturation, which typically correspond to the physical track. This binary mask is then smoothed using morphological operations (openings and Gaussian blur), and valid contours are filtered based on area. A function get_dark_mask() encapsulates this process.

Metric	Min	Max	Final (Step 10399)
Total Loss	3.2430	0.4211	0.5329
mAP@0.5	0.5773	0.9950	0.9950
mAP@0.5:0.95	0.2158	0.8271	0.8205
Precision	0.6227	0.9994	0.9970
Recall	0.6164	0.9977	0.9966
Val Box Loss	0.0504	0.0163	0.0165
Val Cls Loss	0.0065	0.0003	0.0004
Val Obj Loss	0.0123	0.0047	0.0047
Train Cls Loss	0.0239	0.0008	0.0008
Train Obj Loss	0.0277	0.0116	0.0118

Table 1: Key training metrics of the YOLOv5 model across 50 epochs.

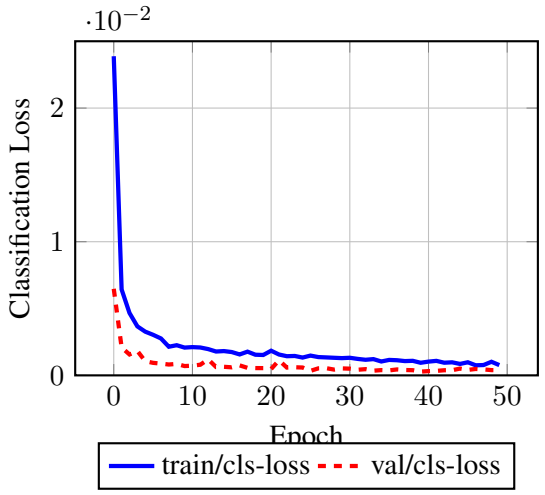

Figure 4: Training and validation classification loss over 50 epochs.

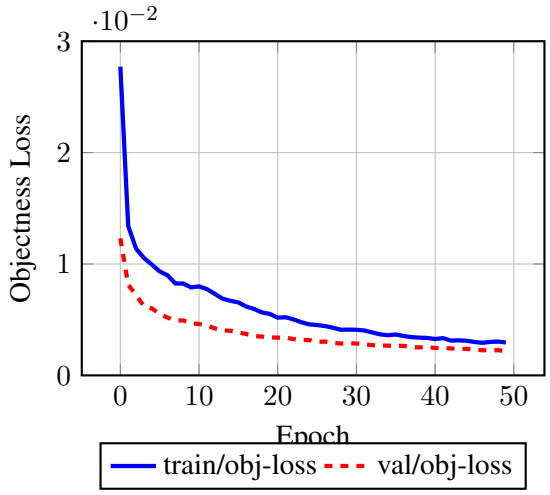

Figure 5: Training and validation objectness loss over 50 epochs.

To increase temporal robustness and reduce noise, a history buffer of binary masks is maintained (default: 10 frames). Masks are averaged and thresholded to produce a stable segmentation. Every few frames (default interval: 10), the contours are re-evaluated and spatially merged using a distance-based k-d tree (cKDTree), implemented in merge_contours_by_distance(). This helps to simplify multiple nearby segments into a unified mask representing the entire track zone.

For object identity, a lightweight CentroidTracker class maintains object IDs across frames based on centroid proximity. For each detected object, its center is checked against the merged dark mask using OpenCV's pointPolygonTest() to determine if it is still on track. This test is performed within the is_on_track() function, which returns a boolean indicating track compliance.

The overall pipeline is orchestrated in main.py, where each frame goes through: preprocessing, dark area segmentation, YOLOv5-based detection (via ONNX inference), centroid-based tracking, and track verification. Results are rendered in real-time with bounding boxes colored green (on track) or red (off track), and optionally sent via HTTP to an external server for future reinforcement learning integration.

In terms of performance, the system achieves approximately 30 FPS when the display window is enabled using –show (via cv2.imshow()), and up to 45–50 FPS when the video output is not shown. Timing metrics are recorded per module (inference, tracking, segmentation, FPS computation) to support profiling and further optimization.

3 Related Work

Real-time object detection on embedded systems has seen increasing research attention. The YOLO family of models, particularly YOLOv5 and its

Figure 6: Example of the racetrack setup seen from above.

variants, have demonstrated successful deployment in embedded systems using NCNN or TensorRT (Jocher et al., 2020). Lightweight detectors such as MobileNet-SSD (Howard et al., 2017) and Tiny-YOLO (Redmon and Farhadi, 2018) have been optimized for devices like Jetson Nano and Raspberry Pi.

In the field of autonomous vehicles, works such as PilotNet (Bojarski et al., 2016) and DAVE-2 have demonstrated end-to-end steering control from raw image data. For lane and track detection, traditional approaches like Canny edge detection or HSV thresholding have been widely used, as well as semantic segmentation with CNNs. For tracking, Deep SORT (Wojke et al., 2017) provides high-accuracy ID assignment, though it is too computationally expensive for lightweight platforms.

4 Results

The final system, running on a GPU-powered desktop, achieved over 30 FPS with full-resolution input and live annotations. Detection accuracy on Mario/Yoshi targets remained consistent across lighting variations and angles. The dark area mask proved reliable in identifying when the toy car left the expected path. Color-coded bounding boxes (green for ”on track”, red for ”off track”) offered intuitive visual feedback.

Real-time performance on Raspberry Pi 4 was attempted but remained limited. Despite optimizations such as NCNN inference and reduced resolution, the hardware could not sustain consistent frame rates suitable for live tracking.

5 Future Work

The project opens the door to several promising extensions:

- **Reinforcement Learning Integration:** Use detection and track verification outputs as feedback for training an RL agent to drive the car autonomously.
- **Learned Track Segmentation:** Replace dark area segmentation with a learned segmentation model to adapt to varied environments.
- **Temporal Prediction:** Add Kalman filters or sequence models to predict object motion and trajectory.
- **HTTP Communication:** Extend the optional HTTP module to integrate with an RL backend in real-time.
- **Edge Deployment:** Port the system to platforms like Jetson Nano or Coral TPU for onboard inference.

6 Conclusion

This project demonstrates a complete, real-time perception pipeline capable of supporting reinforcement learning control for autonomous miniature vehicles. While Raspberry Pi served as a testbed, final performance was only achieved using a GPU system. The trade-offs between inference complexity and embedded feasibility were thoroughly explored, offering practical insights into edge AI deployment and the modular development of hybrid systems.

The full codebase and documentation are available at: `https://github.com/Arou03/carrera_go_raspberry`

Acknowledgments

Thanks to the Deep Learning teaching team at Université Paris-Saclay, and to Roboflow and the open-source CV community. Special thanks to Steven Martin and Thomas Gerald for their guidance and technical advice throughout the development.

References

Aho, A. V., and Ullman, J. D. (1972). *The Theory of Parsing, Translation and Compiling*, Vol. 1. Prentice-Hall.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press.

Glenn Jocher et al. (2020). YOLOv5 by Ultralytics. `https://github.com/ultralytics/yolov5`

Howard, A. et al. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861.

Redmon, J., Farhadi, A. (2018). YOLOv3: An Incremental Improvement. arXiv:1804.02767.

Bojarski, M. et al. (2016). End to End Learning for Self-Driving Cars. arXiv:1604.07316.

Wojke, N., Bewley, A., Paulus, D. (2017). Simple Online and Realtime Tracking with a Deep Association Metric. IEEE ICIP.

`https://roboflow.com/`

`https://github.com/Tencent/ncnn`

`https://opencv.org/`

`https://onnxruntime.ai/`