

Building An Autonomous Robot In Simulation: HTTP 418 Final Report

ENPH 353
December 6th 2025
Instructor: Ioan (Miti) Isbasescu
Team 2

Authors:
Joshua Himmens
George Sleen

Vision Training Repository: <https://github.com/enph353-2025-team2/yolo-vision>
Simulation and Control Repository: https://github.com/enph353-2025-team2/ENPH353_HTTP_418
Final Report Repository: <https://github.com/enph353-2025-team2/final-report>
Competition Music Server: <https://github.com/enph353-2025-team2/music-server>
Robot Operating Video: <https://youtu.be/4jBRHqV6Ss8>

418 I'm a teapot:

Any attempt to brew coffee with a teapot should result in the error code "418 I'm a teapot". The resulting entity body MAY be short and stout.

Contents

1. Introduction	3
1.1. System overview	3
1.2. Division of labour	4
1.3. Artifact tracking and versioning	4
2. Sign Recognition	4
2.1. You Only Look Once (YOLO)	4
2.2. Training	5
2.2.1. Cloud computing	5
2.3. Classical image processing	6
2.3.1. Original strategy	6
2.3.2. Simpler approach	6
2.4. Results	6
2.4.1. More cloud computing	7
2.5. Histogram Voting System	7
3. Line following	7
3.1. Reinforcement Learning	7
3.1.1. Training and the reward function	7
3.2. Imitation Learning	8
3.2.1. IL Data Collection	8
3.2.2. IL Model Training	9
3.2.3. IL Model Use	10
3.3. Obstacle Detection	10
3.3.1. Pedestrian Detection	10
3.3.2. Truck Detection	11
3.3.3. Baby Yoda Avoidance	11
4. Conclusion	11
4.1. Issues at competition	11
4.1.1. OCR and numbers	11
4.1.2. Driving	12
4.1.3. Respawn logic	12
4.2. Future improvements	12
4.2.1. Reinforcement Learning	12
4.2.2. YOLO Optical Character Recognition (OCR) training	12
4.2.3. Optimized code	12
Bibliography	13
Glossary	14
5. Appendix A - Results of RL	15
6. Appendix B - Steering model architectures	16
6.1. Imitation Learning Model (TensorFlow/Keras)	16
6.2. Discrete Q-Network (PyTorch) – Double DQN	17
6.3. Continuous Steering Policy (PyTorch) – Actor-Critic	18
6.4. RLlib Distributed PPO (Default CNN)	19
6.5. Model Comparison Summary	20
7. Appendix C - Conversations with ChatGPT, Codex, and Claude	21
7.1. Large Language Models (LLMs) as knowledge gathering tools	21
7.2. LLMs as coding agents	21
8. Appendix D - YOLO OCR data	25
9. Appendix E - The music server	26

1. Introduction

The HTTP 418 project was to develop a control system for an autonomous clue-finding robot in a simulated world. The robot's goal was to drive around the world while recognizing clues from various clue boards around the map while avoiding pedestrians, vehicles, and Baby Yoda (illustrated in Figure 1). Furthermore, the robot had to remain within the lines of the road while completing the course using only visual information derived from the robot's on-board camera.

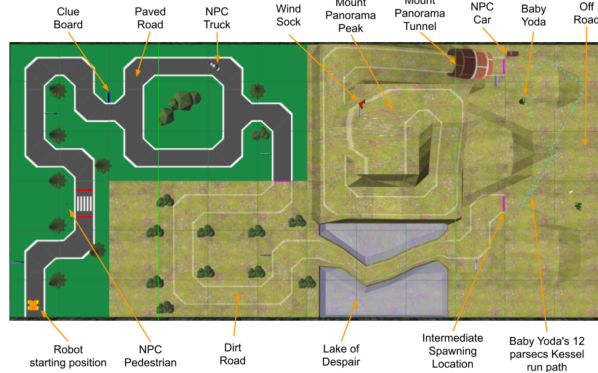
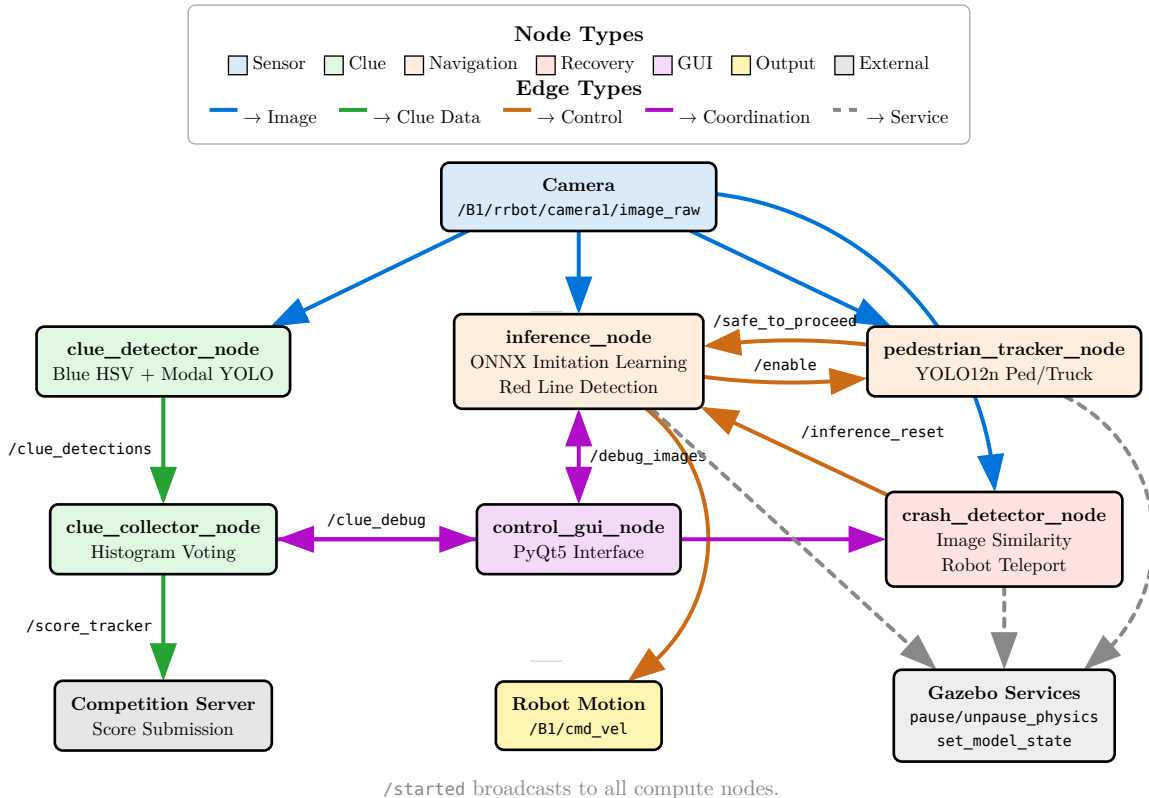


Figure 1: Overhead view of competition playing area
Source: M. Isbasescu [1, Figure 1]

1.1. System overview

The control system is a collection of Robot Operating System (ROS) nodes connected by ROS topics and services with the goal of providing accurate and timely information to the robot's steering system and scoring system. This architecture developed from the goal of semantically segmenting the system so that each node could be tested individually with the `control_gui_node`. Furthermore, this model allows us to decouple time dependencies between nodes with varying compute constraints. For instance our `inference_node` uses a compute-optimized small network capable of running inference at the camera's fps for maximum driving fidelity, while `pedestrian_tracker_node` and `clue_detector_node` both run large models incapable of being run in real-time.



Listing 1: Control Package ROS System Architecture

At its core, the `inference_node` drives the robot using an ONNX model trained via imitation learning, processing camera images to predict steering commands published to `/B1/cmd_vel`. When the robot approaches crosswalks, it transitions into a waiting state and enables the `pedestrian_tracker_node`, which uses a YOLO12n model to track pedestrian movement and detect blocking vehicles. The system only proceeds once the pedestrian has been observed moving, then stopped or left the frame (in that order). Concurrently, the `clue_detector_node` scans for blue clue signs in camera images, crops them, and sends the crops to a Modal-hosted Graphics Processing Unit (GPU) service running a custom trained YOLO OCR model to extract text. The `clue_collector_node` aggregates these detections using histogram voting, publishing the most-seen value for each clue type to the competition scoring node every 2 seconds.

System coordination is managed through the `/started` topic, a latched boolean published by the `control_gui_node` (a PyQt5 interface) that enables all processing nodes simultaneously. The Graphical User Interface (GUI) also provides real-time visualization of camera feeds, YOLO detection overlays, and pedestrian tracker state for debugging.

The `crash_detector_node` provides fault tolerance by monitoring image similarity to detect when the robot is stuck, automatically teleporting it back to a spawn position via Gazebo Simulators (Gazebos) `set_model_state` service. The run terminates when the final “BANDIT” clue reaches sufficient confidence, at which point the `clue_collector` publishes `/run_ended` to signal all nodes to stop and end the timer.

1.2. Division of labour

There are two large problems to be tackled for competition: driving, and clue recognition. As such we felt that the most natural way to divide the work was for each member to take on one of these problems. Josh primarily focused on driving, while George focused on OCR and training. Both members worked on getting networking set up for the final competition, which involved multiple cloud compute providers, home servers, laptops, and even Bluetooth speakers. Our project ended up being just as much a battle with reproducibility on different systems as it was training a robot to solve the mystery.

The driving training done by Josh explored a lot of different approaches with both RL and IL. Joshua also came up with the idea to repurposed YOLO for OCR, and architected much of the training schemes for both models.

George did training for the character recognition model, which was largely training a YOLO model to recognize both the clue-boards, as well as the individual letters on the signs. George also did a lot of training for the IL driving.

1.3. Artifact tracking and versioning

We made significant use of Weights & Biases (W&B), Git Version Control (Git), and The uv Python Package Manager (uv), for version tracking and to ensure repeatable builds. During a run, the custom trained YOLO models and driving models would be downloaded from W&B from the latest version with the `comp` tag. Furthermore, this decoupled development so that while George worked on model training, Joshua could work on crash detection while still using the best models.

We also made significant use of uv for package management because it allowed us to inter-op modern python versions running current and supported versions of libraries from the ROS python environment and installation.

2. Sign Recognition

2.1. You Only Look Once (YOLO)

The most common approach for OCR was to train a Convolutional Neural Network (CNN) much like the labs leading up to competition had shown. We went a different direction and wanted to try and get one YOLO model to detect *everything*: pedestrians, cars, signs, characters, etc.

2.2. Training

Our goal with training YOLO was to update the current weights without having to completely rebuild the model from the ground up.

Our motivation was ease of development, using YOLO for all detections would limit the need for classical algorithms to identify signs, segment letters, classify letters, avoid pedestrians and the truck. Unfortunately, our focus on YOLO to recognize the letters forced us to overfit the model, which has the side effect of reducing the models efficacy on other classes like the pedestrian and truck. Similarly, while we found that the model was highly effective at detecting the letters, it struggled with the signs, which forced us to use a classical algorithm for sign detection.

For signs specifically our diagnostic data seemed to point to the model being very good, but in vivo it ended up being not good at all. We ended up needing to run multiple YOLO models, and classical algorithms to pick out other objects and the signs respectively.

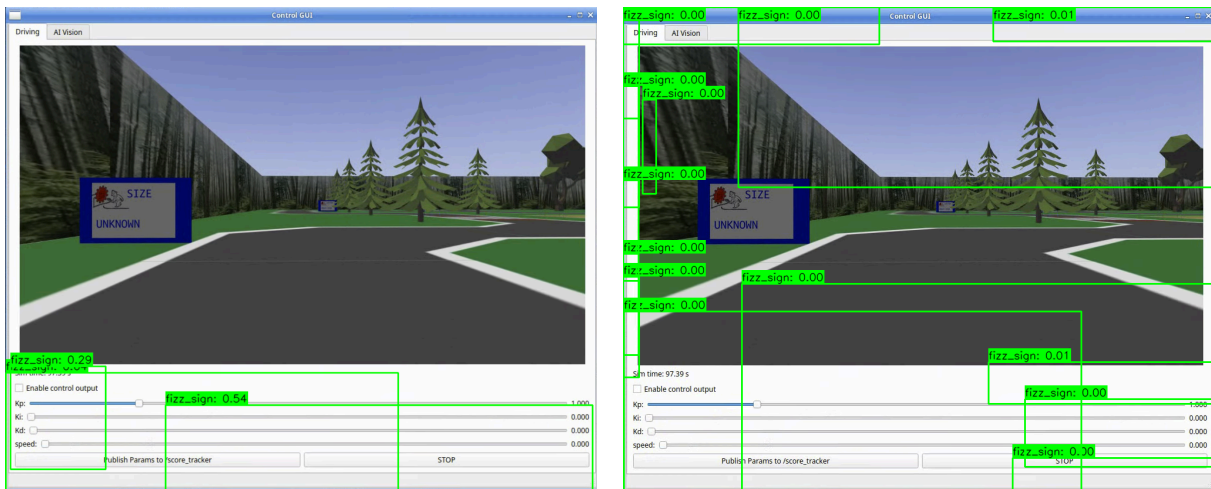


Figure 2: Examples of YOLO struggling to recognize the fizz sign

From the figure you can see that it doesn't come even close to detecting the fizz sign. One of the issues that I suspected was that there was no blue border in the training data, however even after adding it in our performance was similar, and we ended up abandoning the one YOLO model.

2.2.1. Cloud computing

Training YOLO on our laptops with any reasonable amount of data was *slow*. Fortunately for us, cloud computing is extremely affordable and reasonably easy to deal with. A computer with hundreds of cores and four of the highest consumer end graphics cards costs only a couple of dollars an hour. We ended up with a configuration of four RTX 5090s and the newest generation AMD EPYC. We chose to use Runpod, as they have a free tier that allowed us to train the initial models at no cost.

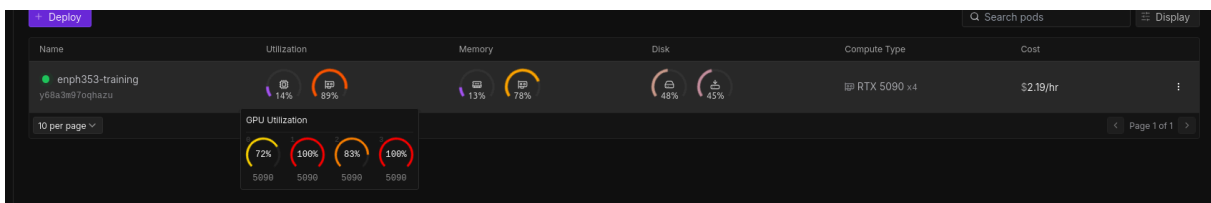


Figure 3: Runpod instance with 4 RTX 5090s for training.

The challenge with having this much computational power is no longer how long it takes, but how to distribute the computation well. We ran into many issues where batch sizes would not automatically scale and artifacts and models would not upload on multi-GPU setups. Since neither of us have multiple GPUs personally, it ended up with a lot of debugging a machine that we can't test before production.

Once we got it working though, our training was blazing fast and let us iterate through models *extremely* quickly. We could train a large YOLO model with roughly 100 GiB of data in only a few hours.



Figure 4: Example YOLO training dataset from Weights and Biases.

2.3. Classical image processing

As the YOLO model was surprisingly bad at recognizing the fizz signs relative to the characters on the signs, we had to employ some other methods to help our YOLO model.

2.3.1. Original strategy

Originally we were hoping to run YOLO on our frame, and then use our ‘fizz sign’ detection to cut in on just the sign and feed that smaller image back into the same model to get out our clues.

2.3.2. Simpler approach

We ended up exploiting the fact that the sign border is always an exact blue that doesn’t show up anywhere else on the map. With an HSV transform on the image, and some careful tuning of the thresholding we could easily pick out the fizz signs and feed the cut in image to our YOLO model, which then had much better performance.



Figure 5: Examples of different thresholds when selecting for the fizz sign blue

2.4. Results

Our final OCR model ended up being extremely reliable and we could consistently detect every sign as we drove past. It was also extremely slow and made driving very difficult with all the extra computational power we required



Figure 6: GUI for viewing the robot’s camera feed and the cut in image fed to our YOLO OCR model.

2.4.1. More cloud computing

Since our OCR could be completely decoupled from the simulation minus sending back guesses on the ROS network, and we had lots of free cloud compute credits, we set up a Modal endpoint to process all our images. This meant we could focus all our compute on sending as many images as we could take and having our endpoint auto-scale to process as many images as we could send, as fast as we could send them. This was very effective, and driving by signs that are only in frame for a second or two would return back twenty to thirty guesses for that sign, which we could then be sent to the `clue_collector_node`.

2.5. Histogram Voting System

```
Clue Collection Status
=== Clue Collection Status ===
Started: True, Enabled: True

SIZE: FAMILY(8), Y(1)
VICTIM: MUFF MCFLUOR(5), MUFF MCFLE(1), MUFF MCFLUTCDOR(1)
CRIME: SABOTAGE(15), EB C T A G E(2), E SABOTAGE(1)
TIME: BAKING FINAL(10), AK G FINAL(2), KNFINAL(2)
PLACE: BAKING EXPO(8), BAKING EXPR(1), BAKINGEXPO(1)
MOTIVE: No observations
WEAPON: No observations
BANDIT: No observations
```

Figure 7: Debug interface for histogram voting showing many votes for correct clues

The `clue_collector_node` maintains a histogram (frequency counter) for each of the 8 clue types, incrementing counts each time a particular value is detected. Every 2 seconds, it publishes the most frequently observed value for each clue type to the scoring system, using a simple majority-vote approach to handle OCR errors and partial reads. When any single BANDIT value accumulates 5 or more matching observations, the collector submits all clues and signals the end of the run.

3. Line following

3.1. Reinforcement Learning

We attempted a number of RL systems for training the robot to traverse the course, however none were successful enough to merit their use at competition. While we attempted to use a Deep Q-Network (DQN), Double Deep Q-Network (DDQN), and Soft Actor-Critic (SAC), the only implementation that showed merit was a discrete action Proximal Policy Optimization (PPO) model. There are a number of theories about why DQN, DDQN, and SAC did not learn; the leading of which is that there was some sort of bug in our implementation of the learning that either prevented learning or made the convergence times exponentially large. It was the possibility of an error in the learning system that prompted the use of Ray RLlib (RLlib) for the PPO implementation. While this model showed significant promise and was observed to have learned many of the fundamentals of driving, it was ultimately unsuitable because its convergence times were in the 48-72 hour range which did not allow enough time to iterate towards suitable hyper parameters.

3.1.1. Training and the reward function

We attempted a number of different loss functions in hopes of incentivizing learning faster, however all of them shared a common goal: make progress towards a waypoint. We developed both a script which would collect positions of the robot as the course was manually navigated and a system for placing movable waypoint markers around the play area (as seen in Figure 8).

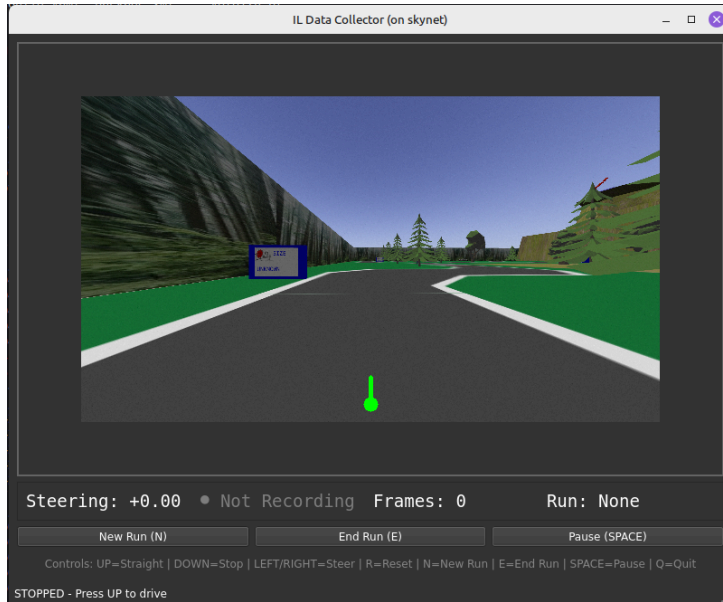


Figure 10: Imitation Learning Training Data Collection Program

This GUI let us keep training runs continuously, and allowed for easy capture of a few sections that we kept having difficulty with, namely the lake of despair and turning the corner into the brick tunnel.

3.2.2. IL Model Training

Our driving model was based on NVIDIA’s end-to-end learning paper for self driving cars [2]¹. We tried both both large and medium sized models for our driving, but with the large model being so much more reliable we were willing to pay the extra compute cost to hopefully get better results from our training, given we had fairly limited time to train.

Our training strategy was quite simple. We would drive the course at a constant speed and only provide input via the left and right arrow keys. We applied exponential smoothing on our input but largely we simply correlated the current picture with a direction that we as a human have picked. Our model would then try to replicate our movements. After a few hours of training we would test the model and drive any sections that our model was having trouble with more. We attempted training both locally and on GPU servers, but the GPU’s were not required since, compared to YOLO, the model was quite small.

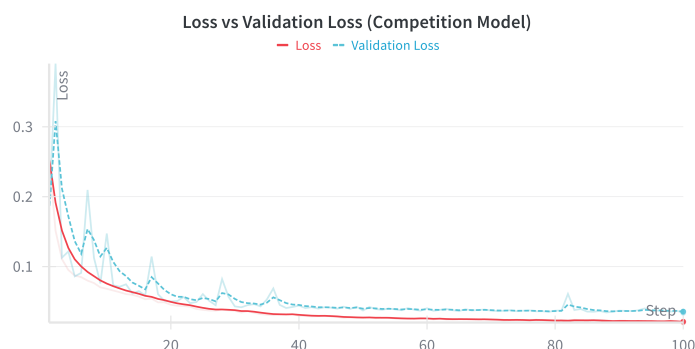


Figure 11: Loss and validation loss for IL steering model

During training it was observed that there was a small delta in the loss between training and validation system. We believe this to be a normal generalization gap caused by the inherent noise of the driving data. During training, it was noticed that some parts of the course, like the bridge, performed meaningfully worse. For those sections, we collected additional training data specifically for them. Note that the train-val split was done per frame with a 20% split to validation.

¹See Section 6.1 for the exact model architecture

Metric	Value
Runs	51
Frames indexed (total)	38 557
Frames / indexed run (mean)	756.0
Training / validation samples	30 845 / 7 712
Train/val split	80.0% / 20.0%
Run types	Full course: 12 · Partial Runs: ² 39

Table 1: IL training data overview

3.2.3. IL Model Use

After our model was trained we had a few things we could change, primarily our linear speed and angular turning rate. These parameters were originally mirrors of what the model had been trained on, but we found that turning up the angular turning rate made by 20% caused our model to be much better at staying on the path that we hoped it would take. We also assumed that the model would output a noisy signal that would need to be smoothed; however we found that the steering system was much more responsive and did not have negative impacts from directly passing the model output with scaling to the twist.

We exported the trained imitation-learning model to the Open Neural Network Exchange (ONNX) format primarily to decouple training from deployment. Training was done in a more modern Python/TensorFlow environment, while the python interpreter that comes with ROS does not support newer versions of TensorFlow models. By converting the network to ONNX and using a lightweight ONNX runtime-based inference node, we avoided these version conflicts entirely, and we could decouple the training from the deployment, that way the `inference_node` only needs to load a single `.onnx` file from W&B.

3.3. Obstacle Detection

Pedestrian and vehicle detection are both handled by the `pedestrian_detection_node` which runs a YOLO12n runs continuously at 10Hz (simulation time). The model is configured with a confidence threshold of 0.5 and an IoU threshold of 0.45. These values were manually chosen based detection efficacy, though they were modified from the Ultralytics documentation [3]. Both IoU and confidence likely could have been tuned more, but for at these values they were observed to work sufficiently well.

3.3.1. Pedestrian Detection

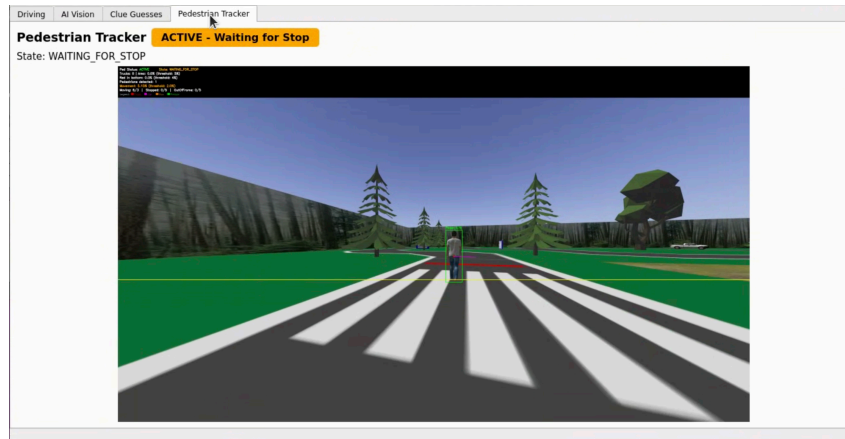


Figure 12: Example detection of pedestrian moving across the frame

The `inference_node` uses an HSV filter on the bottom 10% of the frame, if >50% of pixels match the filter, it triggers a message to the `/pedestrian_tracker/enable` and the stopping of motion in the `inference_node`. The `pedestrian_detection_node` then begins to track any pedestrian bounding boxes. Once it sees a bounding box move (by at least 2% of the image width in 0.1s sim-time), then stop, it publishes a message to `/pedestrian_tracker/safe_to_proceed` indicating to the `inference_node` that it is safe to begin moving again. This was chosen so that the robot always begins moving at the start of the pedestrians loiter time allowing us to move out of the way before it crosses back. In the event of a

²Runs on just the bridge or Yoda section to improve performance

re-spawn, the `inference_node` has internal logic to re-enable crosswalk detection. See Section 4.1.3 for explanation of why more complex logic may have been merited.

State	Description
IDLE	Tracking disabled. Waiting for enable signal from crosswalk detection.
WAITING_FOR_MOVEMENT	Enabled at crosswalk. Looking for a pedestrian that is moving.
WAITING_FOR_STOP	Pedestrian detected moving. Waiting for them to stop or exit frame.
SAFE	Pedestrian has stopped or left frame. Safe to proceed signal sent.

Table 2: Pedestrian tracker state machine states

3.3.2. Truck Detection

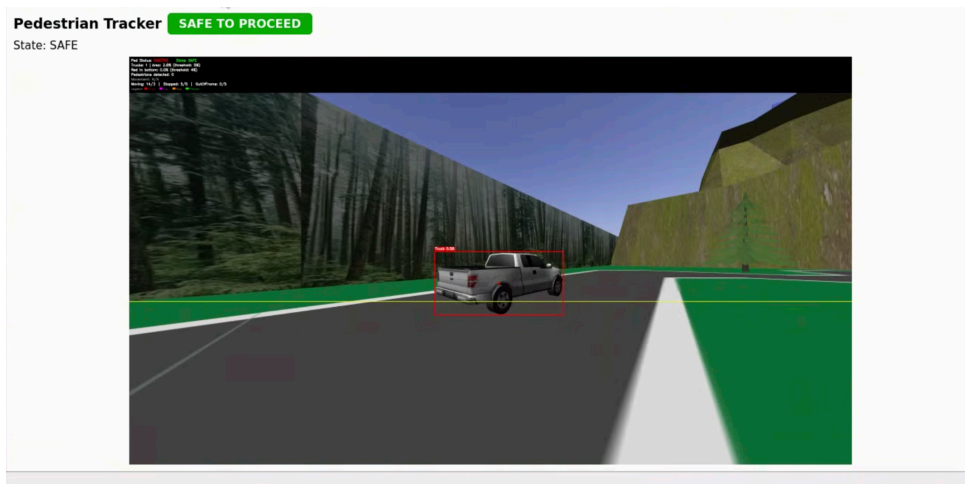


Figure 13: Example detection of truck in frame

If the model ever considers a bounding box for either the car or truck classes to be $>5\%$ of the image, it publishes to `/pedestrian_tracker/truck_blocking` which is a latched boolean. Whenever this topic is published, the inference node immediately stops moving forward and waits for the inference node to report that there is no truck present. Once this happens it begins a 1.5 second delay before moving forward. This slight delay is to debounce the truck signal so that small orientation changes to the truck which might effect inference do not cause the robot to move.

```
self._truck_blocking_pub = rospy.Publisher(
    "/pedestrian_tracker/truck_blocking",
    Bool,
    queue_size=1,
    latch=True)
```

Listing 3: Truck detection latch creation

3.3.3. Baby Yoda Avoidance

Baby Yoda is trivially avoided by choosing a route which drives over the hill instead of into the path of Baby Yoda as shown in Figure 1.

4. Conclusion

4.1. Issues at competition

4.1.1. OCR and numbers

Unfortunately, we did not realize until the day before competition that numbers would be part of the clue bank. Once we realized that our YOLO model was not trained on numbers we hastily added them to the

training set. We got a reasonable model for numbers that seemed to be performing well. Unfortunately we did not test with numbers before competition and we were using the old model still trained on only characters.

4.1.2. Driving

Our imitation learning based driving had a couple issues that we ran into during competition. The easier of the two to deal with was that our robot would frequently drive off the bridge into the lake of despair. We ran into this issue semi-frequently when running before competition, and our solution was to just continue to drive the bridge section over and over again. Unfortunately, our other issue of varying computational power changing the efficacy of our driving model exacerbated our issue of driving into the lake. We attempted to mitigate this by pausing the simulation in between each frame that we send to the driving model, but lower powered computers still had issues with driving properly. This motivated us to try containerizing our ROS simulation with docker and running it on the cloud for competition, but we eventually settled on just doing a fresh install of Ubuntu 20.04 onto a high powered cloud computer and streaming the video feed to our laptop. While crashes still happened, we further mitigated this by adding respawn logic if our robot's camera feed did not change sufficiently over a few frames, which had it's own issues.

4.1.3. Respawn logic

Immediately before competition, we changed our re-spawn logic to re-spawn at the parked truck if the robot had already read all signs up to that point as a mitigation against falling into the water. One of the nuances of our re-spawn logic is that on re-spawn, the crosswalk detection algorithm is reset to avoid hitting the pedestrian on subsequent runs. Unfortunately, the logic was not modified to prevent this reset when the model is spawned at the parked truck, which caused the robot to actively search for a cross walk. Due to our broad-HSV filtering for crosswalk, the tunnel was falsely classified as a crosswalk causing the robot to wait, without bound, for a pedestrian that did not exist.

4.2. Future improvements

With the relatively short turnaround from starting work to competition day arriving, there were a few areas where we would have liked to had some more testing and tried some different approaches. Of course the most obvious improvements would be to fix the issues that we faced in competition, but beyond that there were more interesting improvements that we think could be made.

4.2.1. Reinforcement Learning

While IL is an extremely viable and proven strategy for this course, with more time we would have liked to have an RL driving strategy. Our reasoning is that RL means that we can pack much more information and strategy into an RL model. Rather than having a separate node for object detection we could penalize the model for running into either of these objects. Instead of driving at constant speed we might see that our car would speed up on straight sections for faster driving. With our IL model we were limited by the average driving of our course, and it required lots of human labeled data. Although RL would likely require the same or more human intervention, we could hope to get a much better driving model out of it.

4.2.2. YOLO OCR training

One big misstep when we were training our YOLO model for OCR was not performing a perspective transform on any of the images. While we got great performance for most of the signs regardless, the sign right before the brick tunnel gave us a lot of trouble and it's why we set up the Modal endpoint to process an absurd number of pictures. Had we included a perspective transform in our data augmentation step we believe that we might have been able to rely more heavily on YOLO for fizz sign recognition.

4.2.3. Optimized code

The current implementation of our detective is effective yet computationally heavy. So much so that running it on our personal laptops was pretty much impossible. It would be interesting to see what could be done to improve performance and not rely so heavily on cloud computation.

Bibliography

- [1] M. Isbasescu, “ENPH 353 2025 Fizz Detective - competition notes.” [Online]. Available: https://docs.google.com/document/d/1HZPVtHqixIunRM0CZ0yX2lfPTpDN1_Subw2rpB5XDRg/edit
- [2] M. Bojarski *et al.*, “End to End Learning for Self-Driving Cars.” [Online]. Available: <https://arxiv.org/abs/1604.07316>
- [3] Ultralytics, “Model Prediction with Ultralytics YOLO - Inference Arguments.” Accessed: Nov. 11, 2025. [Online]. Available: <https://docs.ultralytics.com/modes/predict/#inference-arguments>

Glossary

CLI – Command-Line Interface 8

CNN – Convolutional Neural Network: Neural network architecture commonly used for image perception tasks 4

DDQN – Double Deep Q-Network: Variant of DQN that reduces overestimation bias by decoupling action selection and evaluation 7, 7

DQN – Deep Q-Network: Value-based RL algorithm that uses a neural network to approximate the Q-function 7, 7

GPU – Graphics Processing Unit 4, 5, 5, 8

GUI – Graphical User Interface 4, 6

Gazebo – Gazebo Simulator: Physics-based robotics simulation environment often used with ROS 4

HSV – Hue Saturation Value: Color-space representation that separates chromatic content (hue) from intensity and vividness, often used in CV pipelines for robust color thresholding 6, 10, 12

IL – Imitation Learning 2, 2, 2, 2, 4, 4, 8, 8, 8, 9, 9, 9, 10, 10, 12, 15

LLM – Large Language Model: Neural network trained on large text corpora for language understanding and generation 2, 21, 21

OCR – Optical Character Recognition: Techniques for detecting and reading text from images 2, 4, 4, 4, 4, 6, 6, 7, 12, 12

ONNX – Open Neural Network Exchange: Framework-agnostic format for exporting trained neural networks so they can be run efficiently in different runtimes (e.g., ONNX Runtime) without depending on the original training library 10, 10, 10

PPO – Proximal Policy Optimization: On-policy policy-gradient RL algorithm using clipped objectives for stable training 7, 7, 15, 15

RL – Reinforcement Learning 2, 2, 2, 4, 7, 7, 8, 8, 8, 12, 12, 12, 12, 12, 15, 15, 15

RLlib – Ray RLlib: Reinforcement learning library built on Ray for scalable training and experimentation 7, 15

ROS – Robot Operating System: Robotics middleware for message passing, tooling, and system integration 3, 3, 10

SAC – Soft Actor-Critic: Off-policy actor-critic RL algorithm that maximizes expected reward and entropy for robust policies 7, 7, 15, 15

W&B – Weights & Biases: Experiment tracking and ML workflow platform for logging metrics, artifacts, and sweeps 4, 4, 10

YOLO – You Only Look Once: A model family by Ultralytics for real-time object detection and segmentation 2, 2, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 11, 12, 12

Git – Git Version Control 4

uv – The uv Python Package Manager: A fast Python tool for managing dependencies, virtual environments, and project workflows by Astral 4, 4

5. Appendix A - Results of RL

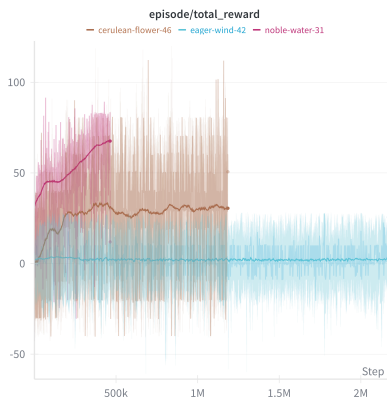


Figure 14: Average reward per episode with RLlib PPO implementation, 24h~1M steps

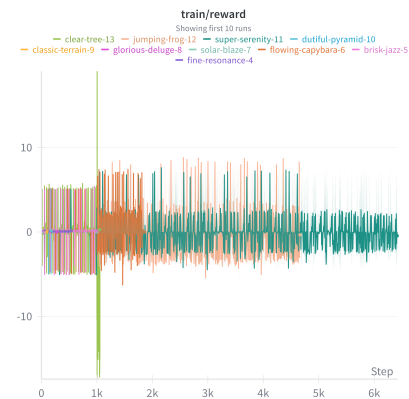


Figure 15: Average reward per episode with custom SAC implementation, 24h~4.5k steps

Figure 14 illustrates 3 trainings using our PPO system with varying success. The pink run was trained with the robot always re-spawning at the first waypoint. While this system was successful, it did not scale because it meant that the network was only learning on the last few frames of each episode. This was fixed in the brown run, where the spawn point was randomized. The plateau that it reaches indicates that there was some underlying problem preventing the model from performing better. In our estimation, this is likely due to the waypoints having too large of hit-boxes. The blue line was an identical training to that of the brown to demonstrate the potential that the model may be stuck in some local minimum.

Figure 15 demonstrates the lack of learning in models trained with our custom SAC implementation with a discrete action space. There is evidence that the data is not pure noise as at approximately 1k steps, the entropy decrease is reflected in the marginally increased training reward. Note that this training occurs with different reward scaling and step delineation, the plot corresponds to the vast majority of runs only reaching the first waypoint directly in front of the spawn point.

While Reinforcement Learning did not function for competition, we are confident that it is ultimately the most effective strategy to develop a high speed-integrated control system as a properly trained model would not require pedestrian, Yoda, truck, or crash detection modules, which could massively simplify development. Furthermore, an RL system with a continuous action space would likely reach much higher speeds than our hand-tuned IL model could. See Section 6 for descriptions of the models.

6. Appendix B - Steering model architectures

This appendix provides detailed architectural specifications for all neural network models implemented in this project. This data was pulled directly from the competition repo (commit 7dc20d0).

6.1. Imitation Learning Model (TensorFlow/Keras)

A convolutional neural network inspired by NVIDIA’s end-to-end learning paper [2] for autonomous driving, adapted for 84×84 input images. The model predicts continuous steering values from camera images.

Layer (type)	Output Shape	Parameters
image_input (InputLayer)	(None, 84, 84, 3)	0
conv1 (Conv2D) kernel: 5×5, stride: 2×2, ReLU	(None, 40, 40, 24)	1,824
batch_normalization	(None, 40, 40, 24)	96
conv2 (Conv2D) kernel: 5×5, stride: 2×2, ReLU	(None, 18, 18, 36)	21,636
batch_normalization_1	(None, 18, 18, 36)	144
conv3 (Conv2D) kernel: 5×5, stride: 2×2, ReLU	(None, 7, 7, 48)	43,248
batch_normalization_2	(None, 7, 7, 48)	192
conv4 (Conv2D) kernel: 3×3, stride: 1×1, ReLU	(None, 5, 5, 64)	27,712
batch_normalization_3	(None, 5, 5, 64)	256
conv5 (Conv2D) kernel: 3×3, stride: 1×1, ReLU	(None, 3, 3, 64)	36,928
batch_normalization_4	(None, 3, 3, 64)	256
flatten (Flatten)	(None, 576)	0
dropout (Dropout, rate=0.3)	(None, 576)	0
fc1 (Dense, ReLU)	(None, 100)	57,700
dropout_1 (Dropout, rate=0.3)	(None, 100)	0
fc2 (Dense, ReLU)	(None, 50)	5,050
fc3 (Dense, ReLU)	(None, 10)	510
steering_output (Dense, tanh)	(None, 1)	11
Total		195,563
Trainable		195,091
Non-trainable		472

Table 3: Imitation Learning Steering Model Architecture

Training Configuration:

- **Loss Function:** Mean Squared Error (MSE)
- **Optimizer:** Adam (learning rate = 0.001)
- **Input Normalization:** RGB values scaled to [0, 1]
- **Output Range:** [-1, 1] (steering via tanh activation)

6.2. Discrete Q-Network (PyTorch) – Double DQN

A CNN-based Q-network for discrete action selection, conditioned on waypoint target vectors. Uses Double DQN for stable training.

Layer (type)	Output Shape	Parameters
Encoder		
Conv2d in: 3, out: 32, kernel: 5×5, stride: 2, pad: 2	(B, 32, 64, 112)	2,432
GroupNorm (groups=8, channels=32)	(B, 32, 64, 112)	64
ReLU	(B, 32, 64, 112)	0
Conv2d in: 32, out: 64, kernel: 3×3, stride: 2, pad: 1	(B, 64, 32, 56)	18,496
GroupNorm (groups=8, channels=64)	(B, 64, 32, 56)	128
ReLU	(B, 64, 32, 56)	0
Conv2d in: 64, out: 128, kernel: 3×3, stride: 2, pad: 1	(B, 128, 16, 28)	73,856
GroupNorm (groups=8, channels=128)	(B, 128, 16, 28)	256
ReLU	(B, 128, 16, 28)	0
Flatten	(B, 57344)	0
Concatenate [features, target_vec]	(B, 57346)	0
Q-Value Head		
Linear (in: 57346, out: 512)	(B, 512)	29,361,664
ReLU	(B, 512)	0
Linear (in: 512, out: 256)	(B, 256)	131,328
ReLU	(B, 256)	0
Linear (in: 256, out: 5)	(B, 5)	1,285
Total Parameters		29,589,509

Table 4: Discrete Q-Network Architecture (input: 128×224×3 image + 2D target vector)

Action	Throttle	Steering
0	0.4	−0.7 (hard right)
1	0.4	−0.3 (slight right)
2	0.4	0.0 (straight)
3	0.4	+0.3 (slight left)
4	0.4	+0.7 (hard left)

Table 5: Discrete Action Space for DQN

Training Configuration:

- **Algorithm:** Double DQN with experience replay
- **Optimizer:** Adam (learning rate = 1×10^{-4})
- **Replay Buffer:** 50,000 transitions

- **Target Network Update:** Every 2,000 steps
- **Discount Factor (γ):** 0.99
- **Exploration:** ϵ -greedy (fixed at 0.05)

6.3. Continuous Steering Policy (PyTorch) – Actor-Critic

An actor-critic CNN policy with continuous action outputs, auxiliary waypoint prediction, and shared feature extraction.

Layer (type)	Output Shape	Parameters
Shared Encoder		
Conv2d in: 3, out: 32, kernel: 5×5, stride: 2, pad: 2	(B, 32, 64, 112)	2,432
GroupNorm (groups=8, channels=32)	(B, 32, 64, 112)	64
ReLU	(B, 32, 64, 112)	0
Conv2d in: 32, out: 64, kernel: 3×3, stride: 2, pad: 1	(B, 64, 32, 56)	18,496
GroupNorm (groups=8, channels=64)	(B, 64, 32, 56)	128
ReLU	(B, 64, 32, 56)	0
Conv2d in: 64, out: 128, kernel: 3×3, stride: 2, pad: 1	(B, 128, 16, 28)	73,856
GroupNorm (groups=8, channels=128)	(B, 128, 16, 28)	256
ReLU	(B, 128, 16, 28)	0
Flatten	(B, 57344)	0
Shared MLP Head		
Linear (in: 57344, out: 512)	(B, 512)	29,360,640
ReLU	(B, 512)	0
Linear (in: 512, out: 128)	(B, 128)	65,664
ReLU	(B, 128)	0
Output Heads		
delta_head (Linear) Auxiliary: predicts [dx, dy] to waypoint	(B, 2)	258
control_mean (Linear) Actor: mean of [throttle, steering]	(B, 2)	258
value_head (Linear) Critic: state value V(s)	(B, 1)	129
log_std (Parameter) Learned log-std per action dimension	(2,)	2
Total Parameters		29,522,183

Table 6: Continuous Actor-Critic Policy Architecture (input: 128×224×3 image)

Output Transformations:

$$\text{throttle} = \sigma(\text{action_raw}_0) \in [0, 1]$$

$$\text{steering} = \tanh(\text{action_raw}_1) \in [-1, 1]$$

Training Configuration:

- **Algorithm:** Actor-Critic (A2C-style)
- **Optimizer:** Adam (learning rate = 1×10^{-4})
- **Rollout Length:** 128 steps
- **Discount Factor (γ):** 0.99
- **Value Loss Coefficient:** 0.5
- **Entropy Coefficient:** 0.01
- **Auxiliary Loss Weight:** 0.1
- **Gradient Clipping:** max norm = 1.0

6.4. RLlib Distributed PPO (Default CNN)

Uses RLlib’s default vision network architecture for 84×84 image inputs with PPO training. Configured with PyTorch backend.

Layer (type)	Output Shape	Parameters
CNN Encoder		
Conv2d in: 3, out: 32, kernel: 8×8 , stride: 4	(B, 32, 21, 21)	6,176
ReLU	(B, 32, 21, 21)	0
Conv2d in: 32, out: 64, kernel: 4×4 , stride: 2	(B, 64, 10, 10)	32,832
ReLU	(B, 64, 10, 10)	0
Conv2d in: 64, out: 64, kernel: 3×3 , stride: 1	(B, 64, 8, 8)	36,928
ReLU	(B, 64, 8, 8)	0
Flatten	(B, 4096)	0
Linear (in: 4096, out: 256)	(B, 256)	1,048,832
ReLU	(B, 256)	0
Output Heads		
policy_head (Linear) Action logits	(B, 7)	1,799
value_head (Linear) State value	(B, 1)	257
Total Parameters		1,126,824

Table 7: RLlib Default Vision Network Architecture (input: $84 \times 84 \times 3$ image)

Action	Throttle	Steering	Description
0	0.3	-1.0	slow, hard left
1	0.6	-0.7	medium, moderate left
2	0.8	-0.3	fast, slight left
3	1.1	0.0	fastest, straight

4	0.8	+0.3	fast, slight right
5	0.6	+0.7	medium, moderate right
6	0.3	+1.0	slow, hard right

Table 8: Discrete Action Space for RLlib PPO

Training Configuration:

- **Algorithm:** Proximal Policy Optimization (PPO)
- **Framework:** RLlib with PyTorch backend
- **Input Normalization:** RGB values scaled to $[0, 1]$

6.5. Model Comparison Summary

Model	Framework	Input Size	Parameters	Algorithm
Imitation Learning	TensorFlow	$84 \times 84 \times 3$	196K	Supervised
Discrete Q-Network	PyTorch	$128 \times 224 \times 3 + 2$	29.6M	Double DQN
Continuous Policy	PyTorch	$128 \times 224 \times 3$	29.5M	Actor-Critic
RLlib PPO	PyTorch	$84 \times 84 \times 3$	1.1M	PPO

Table 9: Comparison of steering network architectures

7. Appendix C - Conversations with ChatGPT, Codex, and Claude

7.1. Large Language Models (LLMs) as knowledge gathering tools

Why used LLM tools in 2 primary ways. The first was as a knowledge agent, to understand what we were doing and answer questions about the syntax or algorithms we were using. See the below example where we used it to help clarify a permission issue where `rsync` was attempting to assign permissions to a user for whom file permissions could not be transferred.

User: Explain `rsync -avz`

LLM: `rsync -avz` is a very common combo.

Breaks down like this:

- `-a` = archive mode
 - This is the “do it properly” flag. It tells `rsync` to:
 - copy recursively (directories and their contents)
 - preserve permissions, timestamps, symbolic links
 - preserve owner, group, and device files (when possible)
 - Internally, `-a` is roughly `-rlptgoD`
- `-v` = verbose
 - Print what `rsync` is doing: which files it’s copying, some stats, etc.
- `-z` = compress data during transfer
 - Compresses file data while sending it over the network (like ssh), then decompresses on the other side.
 - Good for slow networks; not helpful (and just wastes CPU) for local disks or super-fast links.

Typical usage:

```
rsync -avz source/ user@host:/path/to/dest/
```

User: What if I dont want it to copy permissions

LLM: Nice, this is a subtle but important one.

`-a` (archive) includes preserving permissions (`-p`), so if you don’t want that, you have two main options:

1. Easiest: override `-a` with `--no-perms`:

```
rsync -avz --no-perms source/ user@host:/path/to/dest/
```

- `-a` = archive (includes `-p`)
- `--no-perms` = tell `rsync` not to preserve permissions, even though `-a` normally would

This keeps all the other nice archive behaviors (recursive, timestamps, symlinks, etc.), just not permissions.

7.2. LLMs as coding agents

To speed up development, we made extensive use of coding agent like `codex`, `claude code`, and `cursor`. These tools integrate directly into the development environment. This means that they are directly able to access the entire working directory and performs searches of the code base. Their project-specific knowledge makes tool like this excellent for both generating code to add functionality and in diagnosing bugs.

User: I have an intermittent issue where when I press the startbutton in the GUI, sometimes it will say started, but then immediately pause again. This is unacceptable. Find anywhere that might be causing this behavior/

LLM:

▣ I'll help you investigate this intermittent start/pause issue. Let me explore the codebase to find the GUI code and identify what might be causing the automatic pause.

▣ Explore(Explore GUI start/pause logic)
└ Done (24 tool uses · 70.8k tokens · 26m 6s)

▣ Based on the exploration, I found a critical race condition in the publisher initialization. Let me read the actual GUI code to confirm and identify the exact issue:

▣ Read(control_pkg/scripts/control_gui.py)
└ Read 10 lines

▣ Read(control_pkg/scripts/control_gui.py)
└ Read 40 lines

▣ Read(control_pkg/scripts/control_gui.py)
└ Read 10 lines

▣ Now let me check if the GUI itself subscribes to the /started topic, which could create a feedback loop:

▣ Search(pattern: "rospy\.Subscriber.*started", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└ Found 0 lines

▣ Search(pattern: "self\._started\s*=", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└ Found 3 lines (ctrl+o to expand)

▣ Search(pattern: "publish.*started|\/started.*publish", glob: "**/*.py", output_mode: "content")
└ Found 3 lines (ctrl+o to expand)

▣ Search(pattern: "startButton\.clicked\.connect", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└ Found 7 lines (ctrl+o to expand)

▣ Read(control_pkg/scripts/control_gui.py)
└ Read 30 lines

▣ Let me check if there's any code that might disable/enable the button or check for specific conditions when starting:

▣ Search(pattern: "startButton\.(setEnabled|isEnabled|setDisabled)", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└ Found 0 lines

▣ Search(pattern: "run_ended|_run_ended", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└ Found 0 lines

```
Search(pattern: "keyPressEvent|QShortcut|setShortcut", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└─ Found 0 lines
```

```
Search(pattern: "startButton.*|=|QPushButton.*start", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└─ Found 0 lines
```

```
Search(pattern: "**/*.ui")
└─ Found 2 files (ctrl+o to expand)
```

```
Bash(grep -A 10 -B 2 "startButton" /Users/joshuahimmens/code/ENPH353_HTTP_418/control_pkg/ui/Control_gui.ui)
└─
    </item>
    <item>
        <widget class="QPushButton" name="startButton">
... +10 lines (ctrl+o to expand)
```

```
Search(pattern: "_connect_signals\\(|connect_signals\\(", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└─ Found 15 lines (ctrl+o to expand)
```

```
Read(control_pkg/scripts/control_gui.py)
└─ Read 30 lines
```

Wait, I think I found a potential issue! Let me check if the button has `autoRepeat` enabled, which can cause multiple click events:

```
Bash(grep -i "autorepeat\\|autoRepeat" /Users/joshuahimmens/code/ENPH353_HTTP_418/control_pkg/ui/Control_gui.ui)
└─ No matches found
```

```
Search(pattern: "startButton\\. (click|animateClick|setDown|setChecked)", path: "control_pkg/scripts/control_gui.py", output_mode: "content")
└─ Found 1 line (ctrl+o to expand)
```

Based on my investigation, I found two issues that could be causing this behavior:

Issue 1: Missing Debounce Protection (MOST LIKELY CAUSE)

The button handler at `control_gui.py:1520-1525` is vulnerable to rapid/double clicks:

```
def _on_start_stop_toggle(self) -> None:
    """Toggle start/stop state."""
    if self._started:
        self._on_stop_clicked()
    else:
        self._on_start_clicked()
```

If the button signal fires twice quickly (double-click, signal bounce, or mouse hardware issue), the sequence would be:

1. First click: `_started=False` → calls `_on_start_clicked()` → sets `_started=True`
2. Second click (immediate): `_started=True` → calls `_on_stop_clicked()` → sets `_started=False`

This explains the intermittent behavior - it only happens when there's a double-click or signal bounce.

Issue 2: Latched Publisher Without Initial State

At `control_gui.py:1232`, the publisher is created with `latch=True` but never publishes an initial message:

```
self._started_pub = rospy.Publisher("/started", Bool, queue_size=1, latch=True)
```

While `self._started = False` is set at line 960, this value is never published to the topic, creating potential state inconsistencies with subscribers.

— Recommended Fixes

Fix 1: Add debouncing to prevent rapid toggles (`control_gui.py:1520`):

- Add a timestamp check to ignore clicks that happen within 500ms of the last click
- Or add a flag to prevent reentrant calls

Fix 2: Publish initial state (`control_gui.py:1232`):

- Immediately publish `Bool(data=False)` after creating the publisher to establish the initial state

Would you like me to implement these fixes?

8. Appendix D - YOLO OCR data

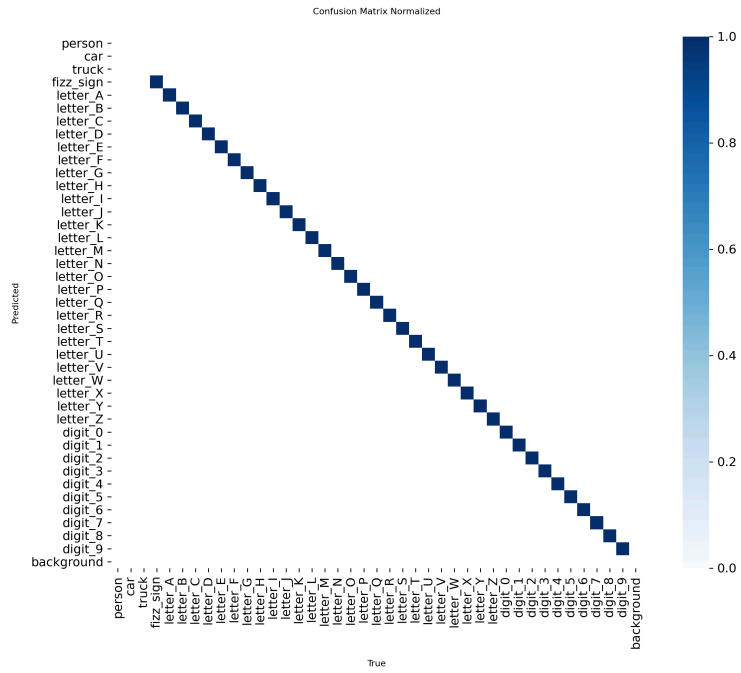


Figure 16: Confusion matrix for our best YOLO model trained on numbers and characters.

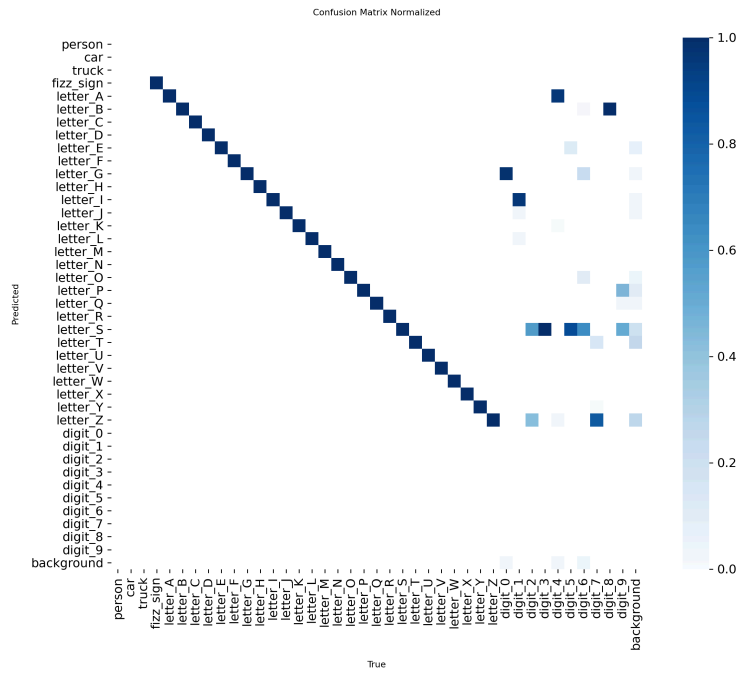


Figure 17: Confusion matrix for YOLO model accidentally used during competition, trained on characters only.

9. Appendix E - The music server

The music server was a gimmick for competition wherein upon the final message being sent to the scoring app. An HTTP request would be sent to start playing one of two songs:

- If the competition was completed, with all clues correct: God Is by Kanye West
- Otherwise: Happy Xmas (War Is Over) by John Lennon

The running instance was connected to a laptop running the music server through a Tailscale VPN overlay network. This is the same network which enabled us to have remote ssh access without making the competition instance public.