Summer Fellowship 2026

On

Fine-Grained Visual Taxonomy for Object Classification

Submitted by

Sneha Bhagwatkar

M.Sc Physics Student

Department of Physics, University of Mumbai

Under the Guidance of

Prof. Jayendran Venkateswaran

Department of IEOR

Indian Institute of Technology Bombay

Acknowledgment

I would like to express my sincere gratitude to everyone who supported me through this internship, and particularly to those who did not just give me things to do, but gave me room to figure things out on my own including the parts that went wrong.

I extend my deepest thanks to Prof. Jayendran Venkateswaran, IEOR Department, IIT Bombay, for overseeing this project and for the intellectual space he provided to push the work in the direction I believed it should go: toward systems-level ML rather than surface-level classification. That freedom is what allowed the project to evolve from a single-stage classifier into the two-stage detection-and-classification architecture documented in this revised report.

I am grateful to Prof. Kannan M. Moudgalya, Principal Investigator of the FOSSEE project, Department of Chemical Engineering, IIT Bombay, for enabling this internship opportunity and for the FOSSEE initiative as a whole.

I thank my FOSSEE guides Mr. Sumanto Kar and Mr. Varad Patil for coordinating all administrative aspects of the internship and for keeping things running smoothly on the official side.

I also thank the FOSSEE Managers Ms. Usha Viswanathan and Ms. Vineeta Parmar and their team for their consistent support throughout the program.

I gratefully acknowledge the support of the National Mission on Education through Information and Communication Technology (NMEICT), Ministry of Education (MoE), Government of India, for facilitating this opportunity.

And to my college, Department of Physics, University of Mumbai, for the foundation that made me ready for this.

A note on this revised edition. *This report documents the project as it stands after the fine-grained classification problem was solved by re-architecting the system around a two-stage pipeline (YOLOv8 for detection and structure, DINOv2 for fine colour/variant identification). It supersedes the earlier single-stage account, and it is written the way I would explain the project to someone who actually wants to understand it, including the parts that did not work the first time.*

Contents

1 Introduction

1.1 National Mission on Education through ICT

The National Mission on Education through ICT (NMEICT) is a scheme under the Department of Higher Education, Ministry of Education, Government of India. Its purpose is to leverage ICT to expand access, equity, and quality in higher education across the country particularly by building bridges between premier institutions and students who would not otherwise have access to their expertise.

FOSSEE (Free/Open Source Software for Education) is one of the initiatives under this mission. It promotes the adoption and development of open source tools in academic and research contexts, and facilitates student internships where real technical work gets done not simulated assignments.

No.	Resource	For Students	For Institutions
1	SWAYAM	Earn credit via MOOC	Develop / host courses
2	National Digital Library	Access e-content	Form NDL Clubs
3	FOSSEE	Volunteer for open-source	Run FLOSS labs
4	Virtual Labs	Perform online experiments	Develop experiments
5	e-Yantra	Embedded systems training	Create labs

Table 1.1. Summary of ICT Initiatives by the Ministry of Education

1.2 FOSSEE Project

The FOSSEE project promotes the use of Free/Libre and Open Source Software (FLOSS) in technical education. Under the Industrial Robotics and Automation domain, internships are structured around real lab hardware at IIT Bombay the Dobot Magician robotic arm, conveyor systems, sensors, and the associated programming stack (Python, Arduino, PyDobot).

My internship extended this track into computer vision: building the perception layer that the robotic system needs to make sorting decisions without human judgment at every step.

1.3 Problem Statement and Objectives

I came into this internship from a physics background with about a year of hands-on ML experience EEG/BCI signal processing at IS360 Technologies, and self-directed work in deep learning. What I was interested in was systems-level ML: not models that perform on benchmarks, but models that behave reliably under real-world constraints.

The first month was orientation understanding the Dobot setup, the Python → Arduino → Dobot serial control chain, and the prior intern's colour-sorting prototype. That background mattered because the computer vision system I was building was not standalone. Its output was meant to drive a physical sorting action. That constraint accuracy under latency and hardware limits shaped every decision made in this project.

The specific problem: build a vision system capable of classifying objects within the same family pieces that look nearly identical on the basis of colour, design, and structural geometry. The test case was LEGO bricks.

This is structurally harder than standard image classification. A ResNet trained on ImageNet separates dogs from cats easily because the visual difference is enormous. Distinguishing one blue 1×1 variant from another that differs only in subtle structure or two reds that a human eye can barely separate is a different problem: the discriminating signal is subtle and highly sensitive to lighting, angle, and annotation quality.

How the objectives evolved. The objectives below are stated as they were ultimately met. The single-stage classifier that the project began with revealed a hard limit on precisely the fine-grained, same-colour discriminations that mattered most; the objectives were achieved by re-architecting the system into two stages, which is the finished state this report documents.

- Build a working data collection pipeline using a live camera feed with keyboard-controlled capture.
- Train a robust system covering multiple LEGO brick variants across colour, design, and structure including same-colour variants that differ only in fine detail.
- Implement a live inference system with calibrated confidence suitable for production use, not just demo accuracy.
- Debug and iterate on failing classes until the system is reliable enough to serve as input for a robotic sorting decision resolving them by architectural redesign where data alone was insufficient.
- Document all trade-offs, failure modes, and design decisions clearly enough for a future intern to extend the work without repeating the same debugging cycles.

A sixth objective emerged over the course of the work: understand what it would take to move this from a laptop-based prototype to an edge-deployed, hardware-constrained real-time system the form it would need to take in any production robotics context.

2 Background

2.1 Computer Vision for Industrial Robotics

Computer vision in industrial robotics serves a specific function: give a robot enough information about the visual world to make a reliable action decision. The robot does not need to understand what it is looking at the way a human does it needs a classification or pose estimate that is accurate enough, and fast enough, to drive a physical action without introducing error into the process.

This constraint accuracy under latency and hardware limits is what separates industrial CV from research CV. A paper that reports 94% mean average precision on a benchmark is not automatically useful in a factory setting where the camera runs at 30 fps, inference must complete within ∼33 ms, and a misclassification sends a part to the wrong bin.

The core pipeline for all such systems is the same: **capture frame → detect object → classify object → output decision to actuator.** This project builds and validates every perception stage of that pipeline, and importantly separates detection from classification so that each can be optimised for what it does best.

2.2 Intra-Family vs. Standard Classification

Standard image classification the ImageNet problem is an inter-category problem. The visual difference between a cat and an airplane is enormous. Models generalize well because the discriminating features are large-magnitude signals.

Intra-family classification has a different structure. When objects share the same shape and scale, the discriminating signal shrinks significantly. For LEGO bricks, the features that matter are:

- **Surface colour**: which varies dramatically under different lighting conditions and camera angles, even for the same physical object. This is the hardest axis, and the one the fine-grained classification stage was ultimately dedicated to.
- **Structural geometry**: a flat plate (one-third standard height) vs. a standard brick is a height difference, not a category difference. The detection stage carries this structural signal.
- **Design detail**: stud count, stud arrangement, and surface texture encode structural identity and provide secondary classification signals.

The practical implication: models trained on clean or studio images fail badly on real bricks under lab lighting because the training distribution does not match the deployment distribution. Annotation quality and lighting consistency become first-order engineering constraints, not afterthoughts. This project encountered this failure mode directly, and Chapter 5 documents how it was resolved.

2.3 Why a Two-Stage Architecture

The central technical finding of this internship is that fine-grained intra-family classification is best served by separating the two questions a sorting system must answer where and what shape is the object, and what fine variant is it into two specialised stages rather than forcing a single network to do both.

- **Stage 1 — YOLOv8 (detection + structure).** Locates each brick with a bounding box and captures its structural form. It answers where the object is and what physical shape it has.
- **Stage 2 — DINOv2 + trained head (fine colour / variant).** Takes the detected crop and makes the fine call which colour variant it is. DINOv2's self-supervised features are strong enough to separate same-colour variants that a single-stage classifier repeatedly confused.

This separation is the difference between the earlier single-stage account and this revised, finished state. It is discussed mechanically in Chapter 4 and evaluated in Chapter 5.

2.4 Tools and Frameworks Overview

- **Python 3.11**: chosen after a version incompatibility forced a downgrade from the default Python 3.14, for which the ML ecosystem had no stable wheels at the time.
- **OpenCV 4.13**: camera access, frame capture, preprocessing, and inference visualization. UVC-compliant; no additional driver layer on macOS.
- **YOLOv8 (Ultralytics)**: Stage 1 detection. Locates every brick and its structure in a single forward pass, handling multiple objects per frame.
- **DINOv2 (self-supervised backbone) + a lightweight trained classification head**: Stage 2 fine-grained colour/variant identification. The head trains in seconds on cached embeddings, which made rapid iteration on the live-data gap practical.
- **TensorFlow 2.21 / Keras (MobileNetV2)**: used in the early proof-of-concept phase to validate the pipeline before the two-stage architecture was adopted.
- **Logitech C270 HD Webcam**: USB 2.0, UVC-compliant; confirmed operational via macOS System Report before any software debugging.
- **Google Colab (T4 GPU)**: YOLOv8 training and DINOv2 embedding extraction / head training.

3 Data Collection and Environment Setup

3.1 Development Environment

The development environment ran on macOS (Apple Silicon, M-series chip). The first significant problem was a Python version incompatibility. The default installation was Python 3.14, for which TensorFlow 2.x had no stable wheel; pip reported only "No matching distribution found for tensorflow", with no version context, which made the root cause non-obvious initially.

The fix was environment-level a fresh virtual environment on Python 3.11, within the supported runtime matrix:

```
python3.11 -m venv venv
source venv/bin/activate
pip install opencv-python numpy tensorflow ultralytics
```

After this the environment resolved cleanly (TensorFlow 2.21.0 | OpenCV 4.13.0 | NumPy 2.4.6). The lesson was about the difference between a Python version that exists and one the ML ecosystem has caught up to tool maturity reliably lags runtime releases by months to years.

A second environment-level issue was camera access. On macOS, the TCC (Transparency, Consent, and Control) framework gates camera access per-application at the OS level. OpenCV's `cv2.VideoCapture(0)` returns a valid capture object even when access is denied it simply delivers empty frames, with no exception and no obvious error. The fix was to grant explicit camera permission to the terminal emulator via System Settings → Privacy and Security → Camera.

3.2 Hardware: Logitech C270 Webcam

The primary imaging hardware was a Logitech C270 HD Webcam at 720p, 30 fps, over USB 2.0. The device is UVC-compliant, so macOS loads the standard UVC class driver automatically on enumeration no third-party driver required. The camera was confirmed operational at the hardware layer via macOS System Report before any software-level debugging. Verifying hardware enumeration before diagnosing software is the correct debugging order and eliminated one failure mode early.

3.3 Data Collection Pipeline

Data collection used a custom Python script that opens a live webcam feed and allows keyboard-controlled capture, with two modes:

- `s` key saves the current frame immediately; used for deliberate, one-at-a-time collection when positioning mattered.
- `a` key auto-captures 10 consecutive frames; used for building volume when the object was consistently positioned.

A crop window was hard-coded to eliminate background variation and constrain the model's input to a consistent region centred on the object placement zone:

```
while count < target_count:
    ret, frame = cap.read()
    frame = frame[200:800, 600:1300]   # crop to object zone
    display = frame.copy()
```

```
    cv2.putText(display, f'{label} | {count}/{target_count}', ...)
    cv2.imshow('Data Collection', display)
```

This crop line was initially placed at module scope rather than inside the while loop an indentation error that applied the crop only to the first captured frame, leaving all subsequent frames uncropped. No exception was thrown; the display simply showed uncropped frames after the first. This is the kind of error that requires careful visual inspection of the output to catch.

Lighting was kept consistent across all collection sessions using a fixed diffuse lamp. Variable lighting was deliberately avoided so the model would learn object surface features rather than lighting artifacts a discipline that proved decisive once the studio-vs-live distribution gap surfaced (Chapter 5).

3.4 Dataset Overview

The dataset began as an 11-class collection of colours, a flat plate, and low-contrast pearl/grey pieces. As the problem sharpened toward genuine fine-grained discrimination, the class set was refined to the six same-colour variants that best represent the hard problem pairs that share a colour and differ only in subtle structure. These six are the classes the finished two-stage system is trained and evaluated on.

Class	Role in the fine-grained problem
blue_01	Blue variant same-colour pair with blue_02
blue_02	Blue variant hardest discrimination against blue_01
red_01	Red variant same-colour pair with red_02
red_02	Red variant historically the weakest, most-confused class
white_01	White / off-white variant low-contrast surface
white_02	White variant same-colour pair with white_01

Table 3.1. The six same-colour variant classes of the finished system

Each class was represented by studio captures on the order of a couple of hundred images. The critical addition in the finished system described in Chapter 5 was a set of live, on-belt crops mixed into each class to align the training distribution with the deployment distribution.

4 Model Training and Implementation

4.1 Phase 1: MobileNetV2 Proof of Concept

The first phase used MobileNetV2 as the backbone for a 3-class colour classifier (red, black, white). It was chosen because it is lightweight enough to run on a laptop CPU and has a well-established transfer-learning path from ImageNet weights.

```
Base model : MobileNetV2 (ImageNet weights, top removed)
Added      : GlobalAveragePooling2D -> Dense(128, relu) -> Dense(3, softmax)
Optimizer  : Adam (lr = 0.001)   Loss: Categorical Crossentropy
Epochs     : 15   Train/Val: 80/20   Image size: 224x224
```

Accuracy on the 3-class problem stabilized above 90% by epoch 10. This confirmed the pipeline functioned correctly before expanding to the harder problem, where more failure modes would emerge. Training from a working baseline rather than attempting the full problem immediately was the correct sequencing decision.

4.2 Phase 2: Single-Stage YOLOv8 and Its Limit

Phase 2 moved to YOLOv8, which combines detection and classification in a single forward pass and handles multiple objects per frame. A single-stage YOLOv8 model was trained to both locate and classify bricks directly. YOLO-format annotations (normalized bounding boxes + class index) were generated with LabelImg and verified per class before training.

```
Model      : YOLOv8n (nano - smallest, fastest; edge-oriented)
Epochs     : 100 (early stopping at plateau)
Image size : 640x640   Batch size: 16
Platform   : Google Colab T4 GPU (~45 min per run)
```

This single-stage approach worked well for visually distinct colours but hit a hard limit on the fine-grained, same-colour discriminations exactly the discriminations the project existed to solve. Forcing one network to simultaneously localise and make a subtle colour/variant call meant the fine signal was consistently swamped. Relaxing the confidence threshold only traded missed detections for false positives (Section 5.3). The conclusion was architectural: the fine call needed its own dedicated stage.

4.3 Phase 3: The Two-Stage Detection + Classification Pipeline

The finished system separates the two questions into two specialised stages:

- **Stage 1 — YOLOv8 (detection + structure).** A detector that locates every brick with a bounding box and captures its structural form. It answers where each brick is and what physical shape it has, and passes each cropped region to Stage 2.
- **Stage 2 — DINOv2 + trained head (fine colour / variant).** A frozen DINOv2 self-supervised backbone extracts rich features from each crop; a lightweight classification head, trained on those embeddings, makes the fine colour/variant decision across the six classes. Because the head trains in seconds on cached embeddings, iteration on the training distribution was fast and cheap.

This division is what solved the fine-grained problem. DINOv2's features separate same-colour variants the blue_01 / blue_02 and red_01 / red_02 pairs that the single-stage classifier repeatedly confused, while YOLO is freed to do what it does best: fast, reliable localisation and structural detection.

4.4 Live Inference System

Live inference runs the two stages in sequence on each webcam frame: YOLO detects and crops; DINOv2 + head classify each crop; results are annotated and accumulated in a per-class counter. The counter is functionally important for the sorting use case the downstream robotic action needs to know how many of each piece have been sorted, not just what the current frame contains. The live window is titled to show both stages (YOLO → DINOv2), making the pipeline's operation legible at a glance during testing.

4.5 Debugging and Iterative Fixes

This section documents the failures encountered, in the order they occurred. Understanding the failure mode is more useful than just knowing the fix.

4.5.1 Python Version Incompatibility

Root cause: Python 3.14 had no TensorFlow wheel; the error message gave no version context. Fix: downgrade to Python 3.11. Lesson: verify the target Python version against each library's supported runtime matrix before setting up an ML environment not after the first install failure.

4.5.2 Camera Permission Gate (macOS TCC)

Root cause: macOS TCC blocks camera access per-application at the OS level; OpenCV's `VideoCapture(0)` returns a valid object even when denied, delivering empty frames silently. Fix: grant explicit camera permission to the terminal emulator.

4.5.3 Frame Crop Indentation Error

Root cause: the crop line was placed at module scope instead of inside the while loop, so it applied only to the first frame. No exception was thrown. Fix: indent the crop inside the loop. Lesson: silent logic errors demand visual inspection of the output, not just a clean run.

4.5.4 Auto-Labelling Gaps on Low-Contrast Classes

Before the detector could be trained, a data-integrity problem surfaced: the automated labelling pass had left near-zero bounding boxes on exactly the low-contrast classes (white and pearl), because the auto-labeller could not reliably find their edges against the background. This was caught by auditing box counts per class rather than trusting the labelling output. The weak classes were re-labelled by hand in Roboflow white_01 alone went from a single box to 171 and the corrected COCO annotations were converted to a single-class YOLO dataset of 1,753 images. Lesson: verify annotation coverage per class before training; an auto-labeller's silence is not the same as correct labels.

4.5.5 The Fine-Grained Confusion and the Architectural Fix

Root cause: the single-stage model could not reliably separate same-colour variants; the discriminating signal was too subtle to survive a network doing detection and fine classification at once. This was misread at first as a data problem alone. Diagnostic: features were clearly present but

insufficiently separated at the production threshold. Fix: re-architect into two stages, delegating the fine colour/variant call to a DINOv2-based classifier. This is the decisive fix of the internship the moment the project moved from "a classifier that mostly works" to "a system that reliably makes the hard call."

4.5.6 Studio-vs-Live Distribution Gap

Root cause: even with the two-stage architecture, the classification head trained on studio images initially faltered on live belt crops it had never seen, at one point reading a blue brick as red_01. Before concluding this was distribution shift, alternative explanations were systematically ruled out: the model files were verified healthy (not corrupted); a preprocessing mismatch was fixed and did not resolve it; a colour-channel (BGR/RGB) swap was excluded by confirming saved crops were correctly blue; and background contamination was excluded by tightening the belt-region crop, after which it still misclassified. Only then was the true cause confirmed the head had never seen the live camera-and-lighting distribution and defaulted toward red. Fix: capture live, on-belt crops for each class using a purpose-built belt-region ROI tool, mix them into the training set, clear the cached DINOv2 embeddings so features re-extract, retrain the head (seconds on cached embeddings), and re-test live. This closed the gap, and the system was confirmed working on live camera bricks. Section 5 discusses why this is the authentic, root-cause fix rather than a threshold workaround.

5 Results and Discussion

5.1 Final System Performance

The finished two-stage system reliably classifies the six same-colour variant classes on the live belt including the same-colour pairs (blue_01/blue_02, red_01/red_02, white_01/white_02) that the single-stage approach could not separate. The fine-grained intra-family classification objective, the core aim of the internship, is met, and confirmed on live camera bricks rather than validation data alone.

The two stages perform as follows. The Stage-1 single-class brick detector trained on a 1,753-image dataset after annotation repair (Section 4.5.4) reaches mAP50 above 0.9 and reliably boxes every brick on the belt, including the blue and white pieces that failed to be detected before the annotation fix. The Stage-2 DINOv2 classifier then makes the fine colour/variant call on each detected crop; after the live-data retraining described below, it classifies live crops correctly, including the previously failing blue-vs-red case.

On the validation set, the DINOv2 classification head achieved a perfect confusion matrix every class, including the hard same-colour pairs, classified correctly. This is a strong result, but it must be read precisely, and Section 5.2 states the caveat plainly rather than letting the number stand alone.

5.2 Validation Accuracy vs. Deployment Reliability

A perfect score on a validation set is not, by itself, proof that a system works in deployment and treating it as such is exactly the mistake this project is meant to guard against. Two points make the result honest:

- **The validation set is a random split of largely studio-similar data.** A perfect score there proves the model is excellent on data resembling its training set. It does not, on its own, establish live reliability.
- **The meaningful test was the separate live-inference evaluation on the belt,** which is what actually validates deployment. The two-stage system was confirmed working live on the six classes after the live-crop retraining.

Stated together: the validation matrix shows the model is excellent on data similar to its training; the live test shows it works on the deployment distribution. Each result means something different, and claiming the validation number as a deployment guarantee would be over-claiming. Being able to articulate that gap and to close it by aligning the training distribution with the live one is the substantive engineering contribution here.

The diagnosis that led to closing the gap is worth recording, because it is a case study in not fixing the wrong thing. When the validation-perfect classifier misread a live blue brick as red, four plausible causes were tested and eliminated in turn corrupted model files (verified healthy), a preprocessing mismatch (fixed, no effect), a BGR/RGB channel swap (excluded: saved crops were correctly blue), and background contamination in the crop (excluded: a tight belt-region crop still misclassified). Only after ruling these out was distribution shift confirmed as the true cause. The fix capturing live crops and retraining the head then closed the gap and the system was verified live. Each eliminated hypothesis saved the effort that would have been wasted "fixing" a cause that was not the real one.

5.3 Why the Single-Stage Threshold Games Did Not Work

During the single-stage phase, lowering the global confidence threshold to force the failing discriminations was tried and explicitly rejected. Relaxing the threshold accepts lower-confidence detections everywhere, which spikes false positives on background texture and wrong-class detections. Threshold relaxation is not a fix for a class that lacks discriminative signal; the correct response is either better data or as this project found a better architecture. The behaviour across thresholds is summarised below.

Threshold	Reliable colours	False positives	Fine same-colour pairs
0.15	All detected	HIGH	Partial, unreliable
0.25	All detected	MEDIUM	Occasional
0.35	Most detected	LOW	Rare
0.45	Reliable	NEAR ZERO	Missed by single-stage
0.60	Reliable	ZERO	Missed by single-stage

Table 5.1. Single-stage behaviour at varying confidence thresholds. No threshold recovers the fine same-colour discriminations motivating the two-stage redesign.

5.4 Root Causes Identified Along the Way

5.4.1 Annotation Noise

Loose bounding boxes around low-contrast objects included background area in the training crops, so the model partly learned background texture rather than object surface. A data-quality problem, not an architecture problem; addressed by tighter annotation.

5.4.2 Scale Mismatch

Some training images were captured at a different zoom level than the live feed, so learned spatial features did not generalize to the deployment scale the object present in frame but the box absent or low-confidence. Addressed by capturing training data at the live camera's scale.

5.4.3 Insufficient Discriminative Signal in a Single Stage

For the subtlest same-colour distinctions, a single softmax over detection features simply did not carry enough separating signal. This is the finding that drove the architecture change: rather than a lighting redesign or a depth modality, delegating the fine call to DINOv2's self-supervised features supplied the discriminative signal the single-stage model lacked.

6 Learning Outcomes

The internship covered ground I had not expected to cover and pushed on assumptions I did not know I was carrying. The outcomes below reflect what actually changed in how I think, not just what I can list on a CV.

6.1 Systems-Level Thinking in ML

The biggest shift was in how I frame an ML problem. Before this internship I thought of model accuracy as the primary metric. After multiple rounds of retraining, a full architectural redesign, and live debugging, accuracy is now one constraint among many: latency, data-distribution alignment, deployment environment, and annotation quality all determine whether a model is useful, and any one of them can make a high-accuracy model non-functional in practice. A perfect validation matrix that is not yet a deployment guarantee is the clearest example.

6.2 Architecture as a Debugging Tool

The most important lesson of the revised project is that some failures are not fixed by more data or better hyperparameters but by changing the shape of the system. The fine-grained confusion was solved by splitting detection from fine classification and giving the hard call to a backbone suited to it. Knowing when a problem is a data problem, when it is a threshold problem, and when it is an architecture problem is a distinct and valuable skill.

6.3 Debugging Across System Layers

Debugging this pipeline required checking independent layers: hardware (is the camera enumerated), OS (are permissions granted), environment (right package versions), data (are annotations and scale correct), architecture (is the model shaped for the problem), and inference (is the threshold calibrated). Each can fail silently or misleadingly. The camera issue looked like an OpenCV bug; the crop error looked like a display problem; the fine-grained confusion looked like a data problem. None were what they first appeared.

6.4 Data Quality as an Engineering Problem

Annotation quality and data-collection consistency are not secondary concerns they are a primary engineering problem in applied computer vision. The studio-vs-live gap was a data failure: the training distribution did not represent the deployment distribution. The correct response to that failing behaviour was to fix the data (capture live crops and retrain), not to tune the threshold.

6.5 Software–Hardware Integration

Operating the Dobot setup and understanding the Python → Arduino → Dobot chain in the first month made concrete something easy to treat abstractly: a classifier is a component in a pipeline that ends in a physical action, with latency requirements and real consequences if the classification is wrong. That grounding shaped every design decision the choice of YOLOv8n for speed, the two-stage split for reliability, the class counter for the sorting context.

6.6 Production vs. Demo Accuracy

There is a meaningful difference between a model that scores well on a validation set in a notebook and a system that runs reliably under live lighting and drives a physical actuator. The gap between those two is where most of the engineering work actually lives. This internship made that gap concrete rather than theoretical and gave me a mechanistic account of it (distribution shift, confidence calibration, latency) that lets me close it rather than just acknowledge it.

7 Conclusion

This internship produced a working computer vision pipeline for fine-grained intra-family object classification built from scratch, debugged across multiple system layers, re-architected when the first design hit its limit, and documented honestly, including the parts that did not work the first time.

The pipeline progressed through distinct phases. The first established the environment, hardware, and a proof-of-concept 3-class MobileNetV2 classifier that confirmed the fundamental approach was viable. The second scaled to YOLOv8 and revealed the hard limit of a single-stage design on the fine-grained, same-colour discriminations that were the whole point. The third the finished state separated the problem into two specialised stages: YOLOv8 for detection and structure, and a DINOv2-based classifier for the fine colour/variant call. That redesign solved the fine-grained problem the single-stage model could not.

The finished system reliably classifies six same-colour variant classes on the live belt, including the same-colour pairs that defeated the single-stage approach. On validation the classification head achieved a perfect confusion matrix; the meaningful confirmation was the separate live-inference test, and the report is careful to distinguish the two. The remaining work wiring the classifier output into the Dobot actuation loop is the immediate next engineering step and is out of scope for this report.

Three research directions are proposed for the next phase of this work:

- **Synthetic data generation via NVIDIA Omniverse or Isaac Sim,** targeting further variant classes and introducing sim-to-real validation an open research problem directly relevant to robotics perception.
- **Contrastive embedding with uncertainty-aware inference,** extending the DINOv2 stage with calibrated uncertainty so the system can reject ambiguous inputs rather than misclassify them. This is the technically ambitious direction and would constitute a research-level contribution.
- **Edge deployment via TensorRT on Jetson Nano/Orin,** exporting both stages to ONNX, converting to TensorRT engines, and benchmarking latency, FPS, and accuracy–latency trade-offs under real hardware constraints.

What this internship built is a foundation a working two-stage perception module with documented failure modes and clear extension paths. What it produced, beyond the technical artifacts, is a way of thinking about ML systems where the production environment is part of the design constraint from the start, not an afterthought and where redesigning the system is a legitimate and sometimes necessary tool. That reorientation is the most durable outcome.

Source code, implementation files, and model weights for all versions developed during this internship are available at the GitHub repository: https://github.com/sneha0016/FOSSEE_internship. Demonstration videos and live-inference recordings are available on Google Drive (link in the project repository).

Project Resources

All artifacts produced during this internship are publicly available.

Source code, model weights, and implementation files
https://github.com/sneha0016/FOSSEE_internship

Bibliography

[1] https://www.nmeict.ac.in

[2] https://fossee.in

[3] Ultralytics YOLOv8 Documentation. https://docs.ultralytics.com

[4] TensorFlow / Keras API Reference. https://www.tensorflow.org/api_docs

[5] OpenCV 4.x Documentation. https://docs.opencv.org/4.x

[6] M. Sandler et al., "MobileNetV2: Inverted Residuals and Linear Bottlenecks," CVPR, 2018.

[7] M. Oquab et al., "DINOv2: Learning Robust Visual Features without Supervision," 2023.

[8] T. Chen et al., "A Simple Framework for Contrastive Learning of Visual Representations (SimCLR)," ICML, 2020.

[9] A. Angelopoulos and S. Bates, "A Gentle Introduction to Conformal Prediction," 2021.

[10] ONNX Runtime Documentation. https://onnxruntime.ai/docs

[11] Arduino Documentation. https://docs.arduino.cc