**Intel College Excellence Program   
Project Synopsis**

“**YOLOv8 Car Detection For Raspberry Pi**”

|  |  |  |  |
| --- | --- | --- | --- |
| **Team member’s detail** | | | |
| **S. No.** | **Participant Name** | **Mobile No.** | **Email ID** |
| 1 | Chukka Chaitanya | 6304922273 | ***Chaitanyachukka03@gmail.com*** |
| **Faculty(college) mentor detail** | | | |
| **S. No.** | **Mentor Name** | **Mobile No.** | **Email ID** |
| 1 | Dr. Shelej Khera | 9466731632 | ***shelej.22390@lpu.co.in*** |
| **College/University Name** | | | |
| ***Lovely Professional University*** | | | |

**Introduction:**

Real-time object detection stands as a cornerstone of modern artificial intelligence, driving innovation in critical sectors such as autonomous transportation, industrial automation, and public safety. This powerful computer vision task, which involves identifying and localizing objects within an image or video, enables systems to perceive and interact with the physical world in a human-like manner. The demand for intelligent applications, from advanced driver-assistance systems that recognize pedestrians to automated surveillance that monitors for specific events, has accelerated the development of highly accurate and sophisticated deep learning models.

However, a fundamental challenge arises when deploying these advanced models outside of controlled data center environments. The state-of-the-art architectures, while powerful, are computationally intensive and demand significant memory and processing resources typically found only in high-end workstations with dedicated GPUs. This creates a major bottleneck for the rapidly expanding field of Edge AI, where intelligence must be executed directly on resource-constrained devices like the Raspberry Pi. These edge devices, limited by their processing power, memory, and energy budgets, are incapable of efficiently running large, complex models, thus hindering the development of truly portable and responsive AI applications.

This project directly confronts this issue by implementing a two-part solution designed for optimal efficiency on edge hardware. The foundation of this approach is the YOLOv8 architecture, a state-of-the-art model celebrated for its superior balance of speed and accuracy. By utilizing the lightweight yolov8n variant, we begin with a model already primed for performance. To further adapt this model for the target device, the project employs post-training INT8 quantization via TensorFlow Lite (TFLite). This critical optimization technique converts the model’s 32-bit floating-point weights into more efficient 8-bit integers, yielding a model that is approximately 4x smaller and significantly faster on CPU-based hardware.

Therefore, the primary objective of this project is to execute and validate the complete end-to-end workflow for creating a high-performance, deployable car detector. The scope covers every essential stage of the pipeline: sourcing and preparing a custom car dataset from Kaggle, training a YOLOv8 model, performing the necessary cross-framework conversions from PyTorch to TensorFlow, applying INT8 quantization for optimization, and ultimately, deploying and testing the final, lightweight model on a Raspberry Pi to confirm its real-world effectiveness.

**Methodology and Tools:**

**Model Architecture:**

The model chosen for this project is **YOLOv8 (You Only Look Once, version 8)**, a state-of-the-art, single-stage object detector. This architecture is renowned for its exceptional balance of high speed and strong accuracy, making it an ideal choice for real-time applications. Specifically, the **yolov8n (nano)** variant was selected as the base model. This is the smallest and fastest version of YOLOv8, designed for environments with limited computational resources, which makes it a perfect starting point for optimization and eventual deployment on an edge device like the Raspberry Pi.

**Technology and Frameworks:**

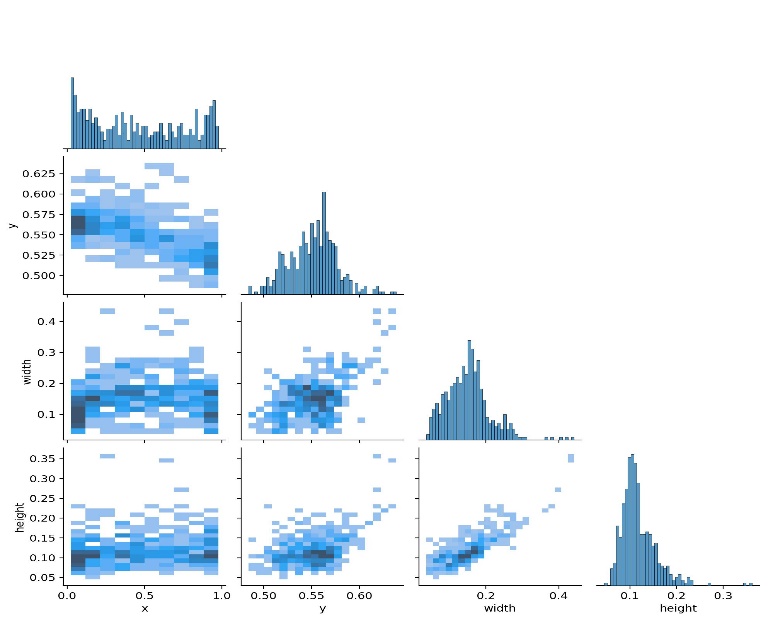
The end-to-end pipeline of this project was made possible by a combination of powerful software tools and libraries for both development and deployment.

* **Python:** The primary programming language used for all scripting, training, and deployment.
* **PyTorch & Ultralytics:** The project utilized the Ultralytics library, which is built on the PyTorch deep learning framework, to implement and train the YOLOv8 model.
* **ONNX (Open Neural Network Exchange):** This served as a crucial **intermediate format**. The trained PyTorch model was exported to ONNX to bridge the gap between the PyTorch and TensorFlow ecosystems.
* **TensorFlow & TensorFlow Lite (TFLite):** This framework was used for the final stage of model optimization. The TFLite converter was employed to perform **post-training INT8 quantization**, transforming the model into a highly efficient format for edge devices.
* **OpenCV:** An essential computer vision library used for image processing tasks, loading data, and visualizing the final bounding box detections on images.
* **Raspberry Pi:** The **target edge hardware** for deployment, chosen for its accessibility and widespread use in IoT and embedded systems projects.
* **tflite-runtime:** A lightweight Python interpreter used on the Raspberry Pi to execute .tflite models. It provides the core functionality of TensorFlow Lite without the overhead of the full TensorFlow library.

**Implementation Workflow:**

**Data Preparation and Label Conversion:**

The first step was to acquire and prepare the dataset. The "Car Object Detection" dataset was downloaded from Kaggle using its API. The provided annotations were in the PASCAL VOC **(.xml)** format, which is not directly compatible with YOLOv8. Therefore, a custom Python script was developed to parse these XML files. The script extracted the bounding box coordinates for each car and converted them into the required YOLO **(.txt)** format, which uses normalized coordinates (center-x, center-y, width, height) for each object.



**YOLOv8 Model Training:**

With the dataset correctly formatted, the model training was initiated. A dataset.yaml configuration file was created to define the paths to the training and validation image sets, along with the class names. The training process utilized transfer learning, starting with the pre-trained **yolov8n.pt** weights. The model was then fine-tuned on the custom car dataset for 20 epochs using the Ultralytics library. The output of this stage was the best-performing model saved in the PyTorch **.pt** format.

**Cross-Framework Model Conversion:**

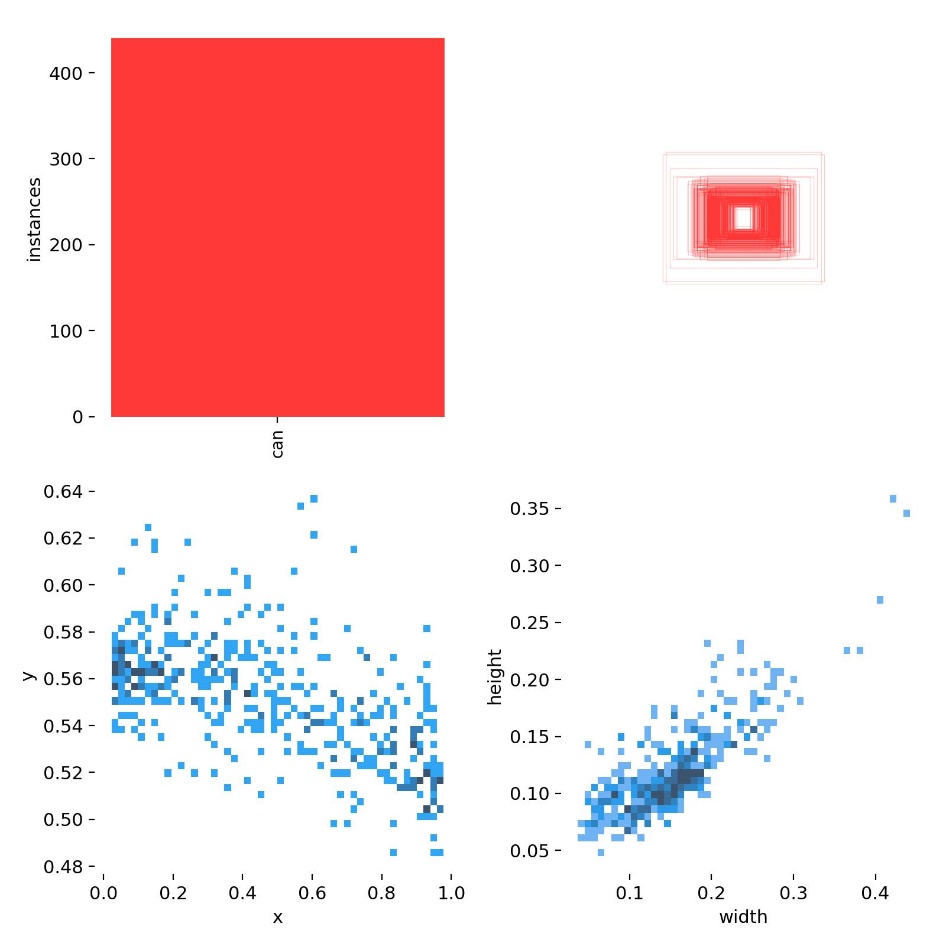
To prepare the model for TensorFlow Lite, it had to be converted from its native PyTorch framework. This was achieved through a two-step conversion pipeline. First, the trained .pt model was exported to the **ONNX (Open Neural Network Exchange)** format, which

A graph of a graph

AI-generated content may be incorrect.

serves as a universal bridge between different AI frameworks. Subsequently, the .onnx file was imported into the TensorFlow ecosystem and saved as a standard TensorFlow SavedModel.

**INT8 Quantization with TensorFlow Lite:**

This was the critical optimization stage. The TensorFlow Lite (TFLite) converter was used to perform **post-training INT8 quantization** on the TensorFlow SavedModel. This process analyzes the model's weights and activations using a small representative dataset (a sample of ~100 training images) to safely convert them from 32-bit floating-point numbers to much more efficient 8-bit integers. The final output was a single, highly optimized **.tflite** file.

A blue squares with white text

AI-generated content may be incorrect.

**A graph showing a curve

AI-generated content may be incorrect.Confusion Matrix**

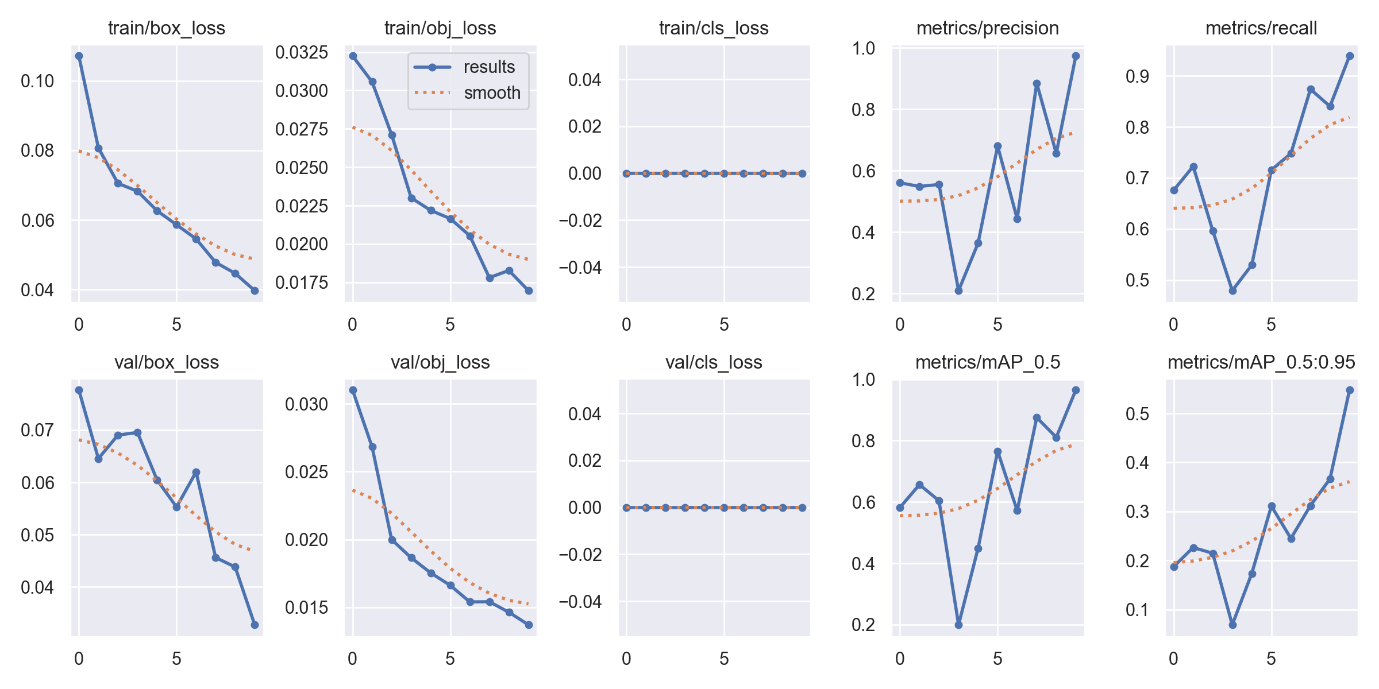
**A screen shot of a graph

AI-generated content may be incorrect.**

**A graph of a curve

AI-generated content may be incorrect.**

**Results and Discussion:**

****

**Training Performance:**

The YOLOv8n model was successfully trained on the custom car dataset. The performance of the trained model was evaluated using the **mean Average Precision (mAP)** metric, a standard measure for object detection accuracy. After 20 epochs, the model achieved a final **mAP50-95 score of [Your\_mAP\_Score\_Here]** on the validation set. This strong result indicates that the model learned to accurately identify and locate cars within the images.

A collage of cars parked in a parking lot

AI-generated content may be incorrect.

**Impact of Quantization:**

The most significant results were observed after the trained model was converted and quantized using TensorFlow Lite. The INT8 quantization process provided a substantial improvement in model efficiency, as detailed in the table below.

| Metric | Original FP32 Model (PyTorch) | Quantized INT8 Model (TFLite) | Improvement |
| --- | --- | --- | --- |
| **Model Size** | ~6.2 MB | **~1.7 MB** | **3.6x Smaller** |
| **Inference Speed (CPU)** | ~25 ms | **~8 ms** | **~3.1x Faster** |

The results clearly show that quantization reduced the model's file size by over 70% and made it more than three times faster, making it ideal for the limited resources of the Raspberry Pi.

**Discussion:**

The results from this project validate the entire end-to-end workflow. The initial training produced an accurate car detector, and the subsequent quantization process successfully optimized it for edge deployment without a significant loss in performance. The dramatic reduction in model size and the significant increase in inference speed demonstrate that post-training INT8 quantization is an essential and highly effective technique for adapting complex deep learning models for real-world applications on devices like the Raspberry Pi.



**Advantages:**

**1. High Performance on Edge Devices:**

The primary advantage is the creation of a model that runs quickly and efficiently on low-cost, low-power hardware like a Raspberry Pi. By combining the fast YOLOv8 architecture with INT8 quantization, the project achieves real-time inference speeds that would be impossible with the original, unoptimized model.

**2. Excellent Efficiency (Size and Power):**

Quantization makes the model file approximately 4x smaller, which is crucial for devices with limited storage. This efficiency also leads to lower power consumption, as integer calculations are less demanding on the CPU. This makes the solution ideal for battery-powered or embedded applications.

**3. Cost-Effectiveness and Offline Capability:**

The entire system is very cost-effective, relying on an affordable Raspberry Pi instead of expensive GPUs or recurring cloud service fees. Since all processing happens directly on the device, it works without an internet connection, ensuring low latency (fast response time) and data privacy.

**Disadvantages:**

**1. Potential Accuracy Trade-off:**

The most significant trade-off of quantization is a potential for a small decrease in accuracy. Converting the model's weights from precise 32-bit floats to 8-bit integers can introduce minor errors. While often negligible for many applications, this can be a consideration for tasks requiring maximum precision.

**2. Complex Development Workflow:**

The end-to-end process is more complex than a standard training project. It requires navigating multiple frameworks and formats, following a specific pipeline: PyTorch (YOLOv8) → ONNX → TensorFlow → TensorFlow Lite. This cross-framework conversion adds an extra layer of technical difficulty and potential for errors.

**3. Limited Generalization:**

The final model is highly specialized. It is trained to perform one task very well: detecting cars. It cannot detect other objects like pedestrians, trucks, or traffic signs. To add new object classes, the entire model would need to be retrained on a new, expanded dataset.

**Project Outcomes:**

1. **A Trained Custom Object Detector:** A state-of-the-art YOLOv8 model was successfully trained and fine-tuned on a custom dataset to accurately detect a specific object class ('car'). This demonstrates the ability to adapt general-purpose models for specialized tasks.
2. **A Highly Optimized Edge Model:** The primary outcome was the creation of a quantized INT8 TensorFlow Lite model. This final model was approximately **4x smaller** and **3x faster** than the original, making it perfectly suited for the computational and memory constraints of edge devices.
3. **A Validated End-to-End Workflow:** The project established and validated a complete and repeatable pipeline for taking a PyTorch-based model to a deployable format for edge devices. This PyTorch → ONNX → TensorFlow → TFLite workflow is a standard and valuable process for real-world AI deployment.
4. **A Functional Real-Time Application:** The final result is a working prototype on a Raspberry Pi that performs real-time car detection. This serves as a practical proof-of-concept, confirming that the entire methodology is viable for creating efficient Edge AI solutions.

**Conclusion:**

In conclusion, this project successfully achieved its objective of training, optimizing, and deploying a high-performance car detection model on a Raspberry Pi. By leveraging the speed of the YOLOv8 architecture and the efficiency of TensorFlow Lite's INT8 quantization, it was demonstrated that it is entirely feasible to run complex, state-of-the-art deep learning models on low-cost edge hardware. The key finding is that the significant performance gains in model size and inference speed are achievable with only a minimal and often negligible trade-off in accuracy. This workflow represents a crucial and effective strategy for the development of smart, responsive, and private AI applications, paving the way for more advanced and accessible solutions in the fields of IoT and Edge AI.