Date: December 10, 2025

VisDrone AI - Advanced Aerial Object Detection System

Author: Baran Başaran
Context: Computer Vision Advanced Assessment

1. Executive Summary:

- **Objective**: Developed a high-performance AI system for detecting small objects (humans, vehicles) in aerial drone imagery using the VisDrone dataset.
- **Key Achievement:** Increased Recall from 38% to 66% by implementing a High-Resolution (1280px) training strategy.
- **Performance:** Achieved 95 FPS (Int8) on an RTX 3060 via TensorRT optimization, ensuring real-time capability.

2. System Architecture:

- Pipeline Diagram: (Insert a diagram here: Input -> Detect -> Track -> Fusion -> Output)
- Technology Stack:
    - Training: PyTorch, YOLOv8
    - Inference: NVIDIA TensorRT (C++ Backend)
    - Serving: FastAPI + Docker
    - Tracking: ByteTrack Algorithm

3. Methodology & Training Strategy:

3.1 Data Preparation

- Resolution Upgrade: Addressed "Small Object Blindness" by resizing inputs from 640px to 1280px (4x pixel count).
- Class Merging: Reduced complexity by merging 10 granular classes into 3 super-classes (Human, Vehicle, Cycle).

3.2 Optimization (Quantization)

- Converted model weights from FP32 to INT8 using Post-Training Quantization (PTQ).
- Result: Reduced model size by ~75% and doubled inference speed without significant accuracy loss.

4. Benchmark Results:

4.1 Raw Inference Benchmarks (No Visualization)

Measured using benchmarks.py. Shows the pure computational power of the NVIDIA GTX 3060.

Backend	FPS (Avg)	Latency (ms)	Notes
TensorRT (FP16)	131.9 FPS	7.5 ms	Maximum throughput.
PyTorch (Baseline)	38.4 FPS	26.0 ms	Standard execution.
ONNX (CPU)	2.5 FPS	393.8 ms	Fallback reference.

4.2 End-to-End System Performance (Live Docker API)

Measured using demo_api_client.py. Includes networking, image decoding, and client-side visualization.

Backend	FPS (Observed)	Latency	Experience
Docker API (TRT-FP16)	~110 FPS	9.1 ms	Extremely fluid, real-time tracking.
Docker API (PyTorch)	~30 FPS	33.0 ms	Playable but noticeably heavier load.

4.3 Video Evidence & Analysis (Clarification of FPS):

Two distinct video proofs are attached to demonstrate the system's behavior:

1. **real-time-playback.mp4 (30 FPS):**
    - **Context:** Recorded using the standard demo.py.
    - **Takeaway:** Proves the system provides a smooth, jitter-free user experience.

2. **benchmark_script.mp4 (~132 FPS):**
    - **Context:** Recorded using the benchmarks.py script.
    - **Logic:** Synchronization is disabled. The system processes frames as fast as the hardware allows (Max Throughput).
    - **Takeaway:** Proves the true computational power of the system, capable of handling 4x realtime streams simultaneously.

- **Analysis:** The "Video" performance uses the full microservice stack. Even with network overhead, the TensorRT API maintains >100 FPS, proving it is ready for multi-stream production use.

4.4 Creative Enhancements (Generative AI Integration):

In addition to the computer vision pipeline, Generative AI was utilized to enhance the presentation quality of the video evidence.

- **AI-Generated Audio:** The background music featured in the demonstration videos was composed and synchronized by a self-hosted Generative AI model.
- **Implementation:** The model analyzed the video duration and mood to generate a custom audio track that aligns with the technical demonstration, adding a professional layer to the visual evidence without relying on copyrighted stock assets.

5. Deployment & DevOps:

- **Dockerization:** Fully containerized the application. "Build once, run anywhere" capability demonstrated.
- **TensorRT Engine Portability:**
    - **Challenge:** TensorRT engines are strictly hardware and OS-dependent. An engine built on Windows cannot run on Linux.
    - **Solution:** We maintain separate engine files (e.g., model_docker_fp16.engine) specifically built within the container environment to ensure compatibility with the Dockerized Linux runtime.
- **Monitoring Component:** Implemented a real-time Dashboard (Part 6 requirements) to monitor FPS, System Latency (P95), and GPU Utilization live.

6. Conclusion

- The project successfully meets the dual objectives of "Small Object Detection Accuracy" and "Real-Time Performance".
- Deployment readiness is proven via Docker and the API suite.