doYOLOngo: An On-Device Voice-Driven YOLO Detector

Jeonghoon Park
UNIST
hoonably@unist.ac.kr

1 Introduction

This report presents *doYOLOngo*, an iOS-based voice-driven object detection system using YOLO26n TensorFlow Lite models. The system performs real-time camera inference, allows users to select a target object through speech commands such as "find bottle" or "detect chair," and highlights the selected target separately from other detections. This design combines on-device object detection with user-driven interaction, which is important for practical mobile vision systems under limited computational resources.

To evaluate mobile deployment trade-offs, the application supports FP32, FP16, and INT8 model formats with runtime precision switching. The system is organized into three interfaces: a live detection screen for camera inference and target-aware visualization, a benchmark dashboard for latency, FPS, model size, and detection statistics, and an export interface for saving CSV or JSON logs. The following sections describe the voice command pipeline, model quantization, detection implementation, performance evaluation, and the observed trade-offs among the three precision formats. The source code is available at: `https://github.com/hoonably/doYOLOngo`

2 Voice Interaction and Target Selection

2.1 Command Parsing and Class Mapping

The voice interface uses a tree-constrained fuzzy command parser rather than simple keyword matching. Speech input is first converted into text using Apple's Speech Framework, and the resulting utterance is parsed along a predefined command tree. At each parsing stage, Levenshtein-distance-based fuzzy matching is applied only to the candidates that are valid children of the current tree node. This design prevents the parser from searching over all possible commands at once and reduces accidental activations from casual speech.

The root node first accepts only valid command triggers, such as `find`, `search`, `show`, `detect`, and `target`. If the first recognized word is not sufficiently close to one of these triggers, the utterance is rejected without changing the application state. This trigger-level fuzzy matching allows the system to recover small recognition errors in command-like utterances, while still rejecting unrelated speech.

After a valid object trigger is selected, the parser moves to the object-class branch and performs fuzzy matching only over the 80 COCO classes supported by the YOLO model. The edit distance is computed as the minimum number of insertions, deletions, and substitutions required to transform the recognized word into a candidate class name [1; 2]. A length-dependent tolerance is used so that short class names are matched strictly while longer class names can tolerate more recognition errors. For example, a misrecognized target phrase such as "border" can be mapped to the closest valid class, "bottle," if it satisfies the distance threshold.

When a valid class is matched, it is stored as the current target object and used during YOLO post-processing to mark each detection as either target or non-target. Because fuzzy matching is constrained by the command tree, the system can support flexible speech input while avoiding the main weakness of naive keyword detection: false activation from arbitrary words that happen to resemble object names.

2.2 Target Feedback and Voice Extension

The live detection screen provides immediate feedback for voice interaction. While the microphone is active, a listening banner shows the recognized text. After a target is selected, a target banner displays the current class and the number of matching detections in the current frame. In the camera overlay, target detections are highlighted with orange bounding boxes, while non-target detections are shown in white. This makes the selected object easy to identify in real time.

The system also supports voice-based system commands beyond ordinary target selection. For example, "Activate Overdrive" switches the active model to INT8 for faster inference, "Disable Limiters" switches back to FP32 for more reliable detection, and "Flashbang" turns on the camera torch. Since these commands are handled by the same tree-constrained parser, they are matched only within the system-command branch and are not confused with COCO object classes. Thus, voice input controls not only the selected target but also the runtime behavior of the detector.

3 Model Quantization

The detection pipeline uses three YOLO26n TensorFlow Lite models: FP32, FP16, and INT8. The FP32 model is used as a baseline without quantization. The FP16 model was obtained through the official Ultralytics export pipeline with half-precision enabled, and the INT8 model was generated using TensorFlow Lite post-training quantization with `tf.lite.TFLiteConverter`. All models share the same interface: an input tensor of shape $[1, 640, 640, 3]$ and an output tensor of shape $[1, 300, 6]$, where each output entry contains bounding-box coordinates, confidence score, and class index.

Table 1: Model precision formats and file sizes.

Model	File Size	Format
FP32	10.3 MB	TFLite baseline
FP16	5.4 MB	FP16 TFLite
INT8	3.3 MB	Dynamic-range quantized TFLite

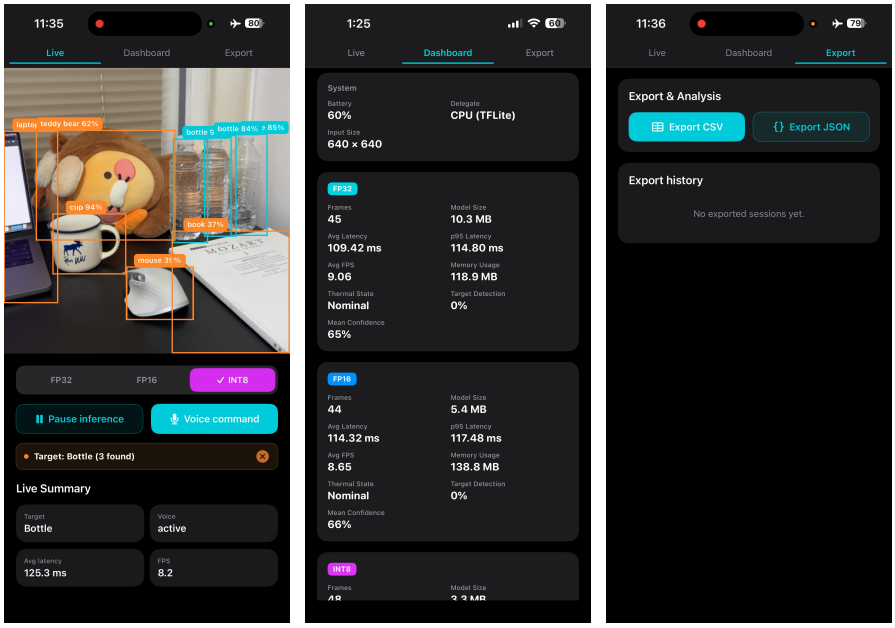
For INT8 conversion, full integer quantization with a representative dataset was first tested, but it produced unstable detection outputs because bounding-box coordinates and confidence scores were quantized together in the same output tensor. The resulting output scale was too coarse, causing incorrect bounding boxes. Therefore, the final INT8 model uses dynamic range quantization: weights are quantized to INT8, while activations and input/output tensors remain in floating-point format. This choice reduces the model size from 10.3 MB to 3.3 MB while preserving compatibility with the iOS inference pipeline and avoiding calibration-induced output instability. From a deployment perspective, this reduction is meaningful because the model file directly affects app bundle size and loading overhead; the INT8 model uses only about 32% of the FP32 file size, while FP16 keeps roughly half of the baseline size.

4 System Design and Implementation

The application is implemented as an iOS app with a SwiftUI interface and a service-oriented MVVM structure. As shown in Figure 1, it consists of three screens: live detection, benchmark dashboard, and export/analysis. The `LiveViewModel` coordinates `CameraService`, `YOLOInferenceService`, and `VoiceService`, while benchmark entries are shared with the dashboard and export views.

The real-time pipeline starts from an `AVCaptureSession`. Each `CMSampleBuffer` frame is processed on a background inference queue, center-cropped, resized to 640×640, converted from BGRA to RGB, and normalized to $[0, 1]$. The TensorFlow Lite interpreter then runs with four threads, and the output tensor $[1, 300, 6]$ is decoded into bounding boxes, confidence scores, and class indices. Detections below the confidence threshold of 0.35 are filtered out.

Detection results are rendered as a SwiftUI overlay on the camera preview. Target detections selected by voice are highlighted in orange, while non-target detections are shown in white. For runtime precision switching, the inference service loads the selected FP32, FP16, or INT8 `.tflite` file from



(a) Live detection (b) Benchmark dashboard (c) Export/analysis

Figure 1: Implemented application screens: live detection, benchmark dashboard, and export/analysis.

the app bundle and reallocates tensors. Since all models share the same input and output format, the same preprocessing and post-processing code is reused.

During inference, the app records timestamped entries including frame index, precision, latency, FPS, target class, predicted classes, confidence scores, and system measurements. The dashboard summarizes average latency, FPS, model size, thermal state, and target detection success rate, and the export screen saves the logs as CSV or JSON.

The benchmark pipeline was extended with system instrumentation for mobile resource analysis. The app records resident memory usage using the Mach task_info API and collects thermal state through the public iOS ProcessInfo API. Since raw battery-current measurements are not exposed by the public iOS APIs used in this implementation, the app records the publicly available battery level from UIDevice as a coarse energy-related indicator instead of a fine-grained current measurement. These values are logged together with latency, FPS, precision, target class, predictions, and confidence scores. The dashboard computes sustained statistics over a 5-second sliding window to reduce transient noise from model loading and runtime precision switching.

5 Performance Evaluation and Results

5.1 Experimental Setup

The evaluation was conducted on an iPhone 15 Pro Max using the rear camera under a fixed camera angle. The test scene contained eight tabletop objects: three bottles, a laptop, a cup, a mouse, a teddy bear, and a book. The same scene was tested under two illumination settings, bright and dim. For each condition, the model was switched in the order of FP32, FP16, and INT8, and each precision was executed for approximately 10 seconds. The exported CSV logs were analyzed using only the stable interval of each segment to reduce noise from button interaction and model switching. To avoid thermal-throttling bias, the quantitative benchmark logs were collected separately from the longer demo recording after allowing the device to cool down.

5.2 Precision-Level Runtime Performance

Table 2 summarizes the average FPS, average latency, p95 latency, detected objects per frame, and memory usage for each precision and illumination condition.

Table 2: Inference performance by precision and illumination condition.

Condition	Precision	FPS	Avg Lat. (ms)	p95 Lat. (ms)	Obj./Frame	Mem. (MB)
Bright	FP32	9.06	109.42	114.80	8.18	118.9
	FP16	8.65	114.32	117.48	8.23	138.8
	INT8	9.13	103.81	107.01	8.14	109.8
Dim	FP32	8.54	116.34	119.52	7.45	119.1
	FP16	8.19	120.41	124.09	6.88	138.8
	INT8	8.86	109.06	112.76	6.71	109.9

In both illumination conditions, INT8 provided the best runtime performance, but the magnitude of the improvement differed by condition. In the bright condition, INT8 increased FPS from 9.06 to 9.13 compared with FP32 and reduced average latency from 109.42 ms to 103.81 ms. This corresponds to a modest 0.8% FPS improvement and a 5.1% latency reduction. In the dim condition, INT8 increased FPS from 8.54 to 8.86 and reduced latency from 116.34 ms to 109.06 ms, corresponding to a 3.7% FPS improvement and a 6.3% latency reduction.

FP16 did not provide a runtime advantage in this implementation. Although FP16 reduced the model file size, it was slower than FP32 in both lighting conditions. This suggests that lower numerical precision does not always guarantee faster mobile inference, especially when runtime conversion, tensor layout, or backend-specific overhead is involved.

The p95 latency results show that the inference pipeline was stable during the benchmark windows. For all precision formats, p95 latency was only slightly higher than average latency, indicating that the measurements were not dominated by occasional slow frames. INT8 also showed the lowest p95 latency in both bright and dim conditions, which supports its advantage for sustained real-time inference.

Memory usage did not strictly follow model file size. INT8 showed the lowest runtime memory footprint, while FP16 used more memory than FP32 despite its smaller file size. This indicates that TFLite interpreter buffers and precision-conversion overhead can affect runtime memory usage, so model file size alone does not fully determine memory efficiency.

5.3 Illumination and Failure Case Analysis

Illumination affected both runtime and detection behavior. In the updated measurement, the bright condition achieved higher FPS than the dim condition for all precision formats. For example, in the FP32 case, FPS decreased from 9.06 in the bright condition to 8.54 in the dim condition, while latency increased from 109.42 ms to 116.34 ms. A similar trend appeared for FP16 and INT8. Since the YOLO inference graph has a mostly fixed computational cost for a fixed 640×640 input, the lower FPS under dim lighting is more likely due to end-to-end camera and system factors, such as auto-exposure, frame-delivery variability, OS scheduling, or measurement noise, rather than recognition difficulty alone.

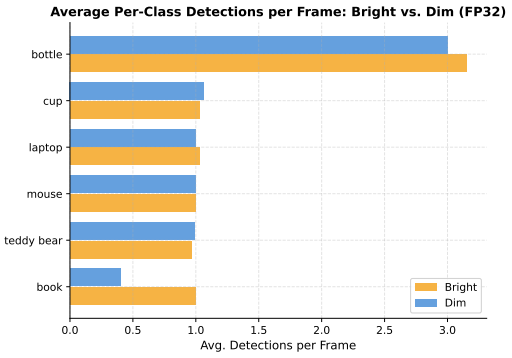


Figure 2: Average per-class detections per frame under bright and dim conditions for FP32.

Figure 2 shows the average number of detections per frame for each object class under bright and dim conditions. Large and distinctive objects such as the laptop and cup were relatively stable, while smaller or low-contrast objects were more sensitive to illumination. For example, the book detection rate decreased in the dim condition, indicating that weak visual details can reduce class-level detection reliability.

The clearest failure case was the mouse under dim illumination. As shown in Figure 3, the detection rate decreased sharply as precision was reduced: FP32 maintained almost perfect detection, FP16

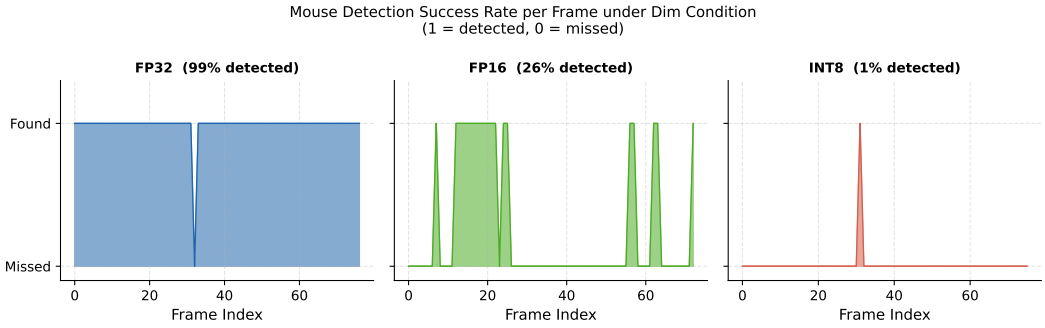


Figure 3: Mouse detection success rate under dim illumination for FP32, FP16, and INT8.

dropped substantially, and INT8 detected the mouse in only about 1–2% of frames. This shows that INT8 is efficient in normal scenes but can be fragile when quantization is combined with weak visual input, especially for small objects.

Overall, INT8 was still the best choice for speed-oriented deployment because it achieved the lowest average and p95 latency in both lighting conditions. However, FP32 remained more reliable for difficult cases such as low illumination and small target objects. These results support the need for runtime precision switching rather than relying on a single fixed model format.

Because public iOS APIs do not expose raw battery current, battery level was logged only as a coarse energy-related indicator; it changed by less than 0.1 percentage points, so precision-wise energy comparison was inconclusive.

6 Discussion and Conclusion

The experiments show that INT8 is the most suitable precision format for speed-oriented mobile deployment. It achieved the lowest average latency and p95 latency in both bright and dim conditions, although its FPS gain over FP32 was modest in the bright scene. FP16 reduced the model file size, but it did not improve runtime performance and showed the highest runtime memory usage. This indicates that file-size reduction alone does not guarantee faster or lighter execution, because TensorFlow Lite interpreter buffers and precision-conversion overhead can also affect runtime behavior.

However, INT8 was not always the most reliable choice. In the dim-light mouse experiment, INT8 detected the small target in only a small fraction of frames, while FP32 remained much more stable. Therefore, INT8 is useful for ordinary scenes where speed is more important, whereas FP32 is preferable for difficult cases such as low illumination and small or low-contrast objects. This trade-off also justifies runtime precision switching and the voice-based commands: “Activate Overdrive” can select INT8 for faster inference, while “Disable Limiters” can return to FP32 when robustness is more important.

The main limitation is that the INT8 model uses dynamic range quantization rather than full integer quantization, so activations remain in floating-point format. In addition, the detector is still sensitive to illumination changes and small objects. Future improvements could include better calibration for full INT8 quantization, illumination-aware preprocessing, temporal smoothing across frames, and more robust voice matching. Overall, the results show that no single precision format is optimal for all mobile detection conditions, and that runtime precision switching is useful for balancing speed and reliability.

References

[1] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[2] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. doi: 10.1145/321796.321811.