



YOLOkémon: Capturing the Hidden Cost of Quantization in Mobile Object Detection

Jiseung Jeong (Student ID 20211301)

Ulsan National Institute of Science and Technology, Ulsan, South Korea

wjdwltmd1151@unist.ac.kr

Abstract

Quantization is usually justified with frame-aggregated metrics such as mAP or mean confidence. Our iOS app runs FP32, FP16, and INT8 variants of YOLO26n on an iPhone 11 (A13 Bionic), selects targets by voice, and logs every frame. Under INT8, mean confidence drops by only 4.4 pp (74.4% $\rightarrow$ 70.0%), yet the per-frame detection rate of the *cell phone* class falls from 93.7% to 45.6%, and on the CPU (XNNPACK) path we use, INT8 runs $1.38\times$ slower than FP32. We also use *frame-to-frame IoU*, a simple temporal-stability measure, as the core of a voice-driven capture mini-game, YOLOkémon. On this device, FP16 gave the best storage–latency trade-off in our runs.

1 Introduction

Real-time object detection on a phone must balance accuracy against speed and energy, and prior work has paired smartphone object recognition with audio feedback to help users find objects [1]. Quantizing an FP32 model to FP16 or INT8 shrinks its memory and storage footprint and, on hardware with low-precision accelerators, also speeds up inference [3]. The *cost* of quantization, however, is usually reported with static, frame-aggregated metrics such as mAP, top-1 accuracy, or mean confidence. These metrics miss what a user actually sees in a live camera preview: whether some classes become unstable, whether boxes jitter from frame to frame, and whether the “faster” model is in fact slower on the execution path the device actually uses.

We measure this gap by running FP32, FP16, and INT8 variants of YOLO26n [2], an NMS-free member of the YOLO family [4], on an iPhone 11 with the same scene and the same voice command. Two results stand out:

- **The average hides a class-level loss.** Mean confidence drops by only 4.4 pp (74.4% $\rightarrow$ 70.0%), but the per-frame detection rate of *cell phone* falls from 93.7% to 45.6% (-48.0 pp).
- **Quantization is not always faster.** On the CPU (XNNPACK) path, the fastest one available for this model, INT8 runs $1.38\times$ slower than FP32.

Contributions.

1. **Frame-to-frame IoU** as a temporal-stability proxy: the IoU between the target’s boxes in consecutive frames. It targets the box jitter that frame-aggregated metrics average away.
2. **YOLOkémon**, a voice-driven capture mini-game built on this metric. The user names a target, and capture progress grows only while its box stays stable, so the cost of quantization is both felt by the user and measured by the app (§6).
3. **Transparent reporting of platform limits** (battery, thermal, delegates), stating which assignment metrics had to be replaced by proxies (§7).

2 Voice Interaction and Target Selection

Push-to-talk. We use push-to-talk rather than an always-on listener. Tapping the mic button on the Live tab starts SFSpeechRecognizer, which stops after 1.2 s of silence or an 8 s hard timeout. The transcript is mapped to one of six intents (Table 1). The tokenizer removes stopwords and first matches verb-only phrases such as “give up”. It then scans n -grams from $n = 3$ down to 1 against the 80 COCO labels, so “cell phone” wins over “phone”. The opening verb decides whether the match becomes a capture or a highlight.

Robustness. The en-US recognizer makes a few recurring mistakes, which we correct in two layers. (1) A single-token *alias dictionary* rewrites homophones (*cop*, *cap* $\rightarrow$ *cup*; *phone* $\rightarrow$ *cell phone*; *chose*, *chosen* $\rightarrow$ *choose*). The parser and the on-screen transcript share this dictionary, so they always agree. (2) A *phrase-level mishear list* catches multi-word errors: “give up” often comes back as “k-pop”, “keep up”, or “hip-hop”, and “capture the cup”

Table 1: Voice intents produced by the parser.

Intent	Example utterance	Effect
.highlight(class)	“find the cup”	Draw the target’s box in green
.capture(class)	“I choose the cup”	Start a YOLOkémon capture (§6)
.stopCapture	“give up”, “stop”	Abort the current capture
.openPokedex	“open pokedex”	Switch to the Pokédex tab
.unsupported(noun)	“find the dragon”	Reply “unsupported class: dragon”
.unparseable	(silence or noise)	No action

as “capture.com”. Matching runs on a normalized copy (lowercase, non-alphanumerics replaced by spaces), so hyphenated and dotted variants fold onto the same pattern.

UI feedback. A shared AppHeader subtitle shows lifecycle and target state; the Live Summary card shows the target, last utterance, average latency, and FPS; a semi-transparent overlay covers the screen while the mic is listening.

3 Model Quantization

Post-training quantization (PTQ). PTQ re-encodes a trained FP32 network at a lower bit-width *without retraining* [3]. FP16 is a plain IEEE-754 half-precision cast of the weights and needs no calibration; on the CPU path, TFLite by default dequantizes these weights to FP32 at load time, so FP16 mainly saves storage, not compute. INT8 maps each real value w to an 8-bit integer q and back:

$$q = \text{clamp}(\text{round}(w/s) + z, q_{\min}, q_{\max}), \quad \hat{w} = s(q - z), \quad (1)$$

with scale s and zero-point z . Following the TFLite spec, weights are quantized symmetrically per channel ($z = 0$, range $[-127, 127]$), activations asymmetrically per tensor ($[-128, 127]$), and biases stay in INT32.

Conversion pipeline. All three models come from the same Ultralytics YOLO26n checkpoint, exported with ultralytics to fp32.tflite. fp16.tflite uses TFLite’s float16 optimization. int8.tflite uses full-integer PTQ, with 128 COCO128 images streamed through a representative_dataset generator to calibrate activation ranges; weight scales come directly from the trained weights. The domain gap between COCO128 and our desk scene is discussed in §7.

Model size. The files are 9.88 / 5.21 / 3.11 MB, i.e., $0.53\times$ (FP16) and $0.31\times$ (INT8) of FP32. FP16 lands near the ideal $0.5\times$. INT8 misses the ideal $0.25\times$ because some content does not shrink by $4\times$ (INT32 biases, per-channel scales, graph metadata); we did not measure each component. A smaller file does not mean faster inference: in our setup, INT8 is the smallest model but the slowest one (§5.2).

4 System Design and Implementation

4.1 App screens

We implement the three required screens plus a Pokédex tab for the extension (Figure 1).

- **Live** (Fig. 1a) covers all six required items in one view: camera preview, box overlay, inference pause, precision toggle, voice trigger, and latency/FPS display.
- **Dashboard** (Fig. 1b) shows average latency, FPS, mean confidence, model size, thermal state, battery level and drop, memory, delegate, input size, and last update time.
- **Export** writes the session log as CSV/JSON and opens the iOS share sheet (§4.3).
- **Pokédex** (Fig. 1c) records which classes were captured, and with which precision (§6).

4.2 Inference pipeline

Each camera frame passes through the following stages:

```
AVCaptureSession (720×1280) → frameQueue → Preprocessor.letterbox (640×640) → InferenceCore actor
→ Yolo26PostProcessor.decode (NMS-free, top- $K$ ) → [Detection] in source pixels → LiveOverlayView
```

Concurrency. Work is split across three layers: a serial frameQueue, the InferenceCore actor, and the MainActor. Moving inference off the main actor removed a ~ 110 ms per-frame stall on the main thread, so the preview and SwiftUI rendering no longer wait for the model. DetectionEngine.switchPrecision(to:) swaps the TFLite Interpreter inside the same actor, so an in-flight inference can never race a model swap.

Fast preprocessing. Replacing the original vImage RGB conversion and letterbox (~ 352 ms/frame) with vImageScaleNoCrop_PlanarF and in-place pixel-format conversion cut preprocessing to 8 ms, a $44\times$ speedup. This single change saves more time than any quantization setting we measured (§5.5).

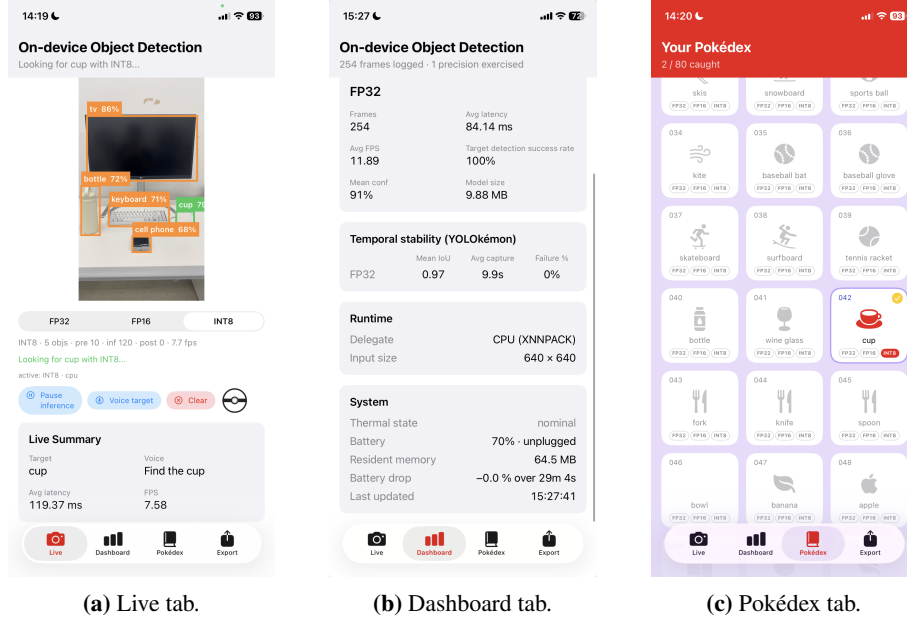


Figure 1: Three of the app’s four tabs (the Export tab, a share sheet, is not shown). (a) INT8 active after “find the cup”: the cup is highlighted in green, with the control buttons, tab bar, and Live Summary card visible. (b) All dashboard metrics on one screen (values from an earlier test session). (c) Cell #042 (cup) caught with INT8: the icon is coloured and the INT8 badge is lit.

Table 2: Performance summary (warm-up frames trimmed). *Mean conf*: average detection confidence per frame. *Success rate*: fraction of frames in which the voice target (cup) was detected. *Lat. ratio*: average latency relative to FP32.

Precision	Latency avg / p95 (ms)	FPS	Size (MB)	Mean conf	Success rate	Lat. ratio
FP32 (baseline)	81.0 / 86.1	11.2	9.88	74.4%	100%	1.00×
FP16	83.3 / 88.1	10.8	5.21	74.8%	100%	1.03×
INT8	111.4 / 124.7	8.4	3.11	70.0%	100%	1.38×

4.3 Logging and export

SessionLogStore keeps the in-memory frame log and per-precision summaries. LogExporter writes Documents/pa2_logs/session_*. {csv,json}: a 22-column CSV (timestamp_ms ... capture_iou) and a JSON file that adds a captures array. PokedexStore saves the Pokédex to Documents/pokedex.json. For offline analysis we AirDrop the logs to a Mac and group rows by precision (Tables 2 and 3), expand predictions_json (Figure 4), and filter on capture_progress (Figure 5).

5 Performance Evaluation

5.1 Experimental setup

All runs use one iPhone 11 (A13 Bionic) with iOS 26.4, built with Xcode 26.4, Swift 5, and TensorFlow Lite 2.x. Inference runs on the CPU with the XNNPACK delegate; the Metal and CoreML delegates were slower on this model (§7). Each precision was streamed for 1–2 minutes at the same desk scene (tv, bottle, keyboard, cup, cell phone), with *cup* as the voice target, in the order FP32 → FP16 → INT8. Between sessions we waited at least 30 s, and longer if needed, until thermalState returned to nominal. After trimming warm-up frames, each precision contributes 423–583 frames (about 50 s at the measured FPS).

5.2 Latency, FPS, and model size

Table 2 summarizes the main metrics. FP16 has 2.8% higher latency and 3.6% lower FPS than FP32 while halving the file size. INT8 is the smallest model but adds 38% latency and drops throughput from 11.2 to 8.4 FPS. Figure 2 shows that this is a systematic shift, not a long tail: INT8 forms its own wider cluster. Over time (Figure 3) the same spread appears as deeper and more frequent FPS dips, and the average run of consecutive frames that miss the target is about 3× longer under INT8 than under FP32.

5.3 Accuracy proxies and per-class detection rate

Without ground-truth labels for the camera stream, mAP cannot be computed. Because the scene is fixed, we compare precisions with relative proxies: target success rate and mean confidence (Table 2), and the per-class

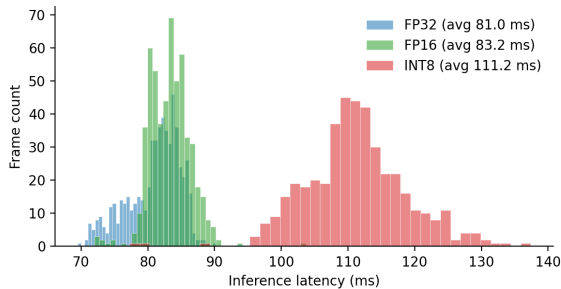


Figure 2: Inference-latency histogram per precision. FP32 and FP16 overlap; INT8 forms a separate, wider cluster.

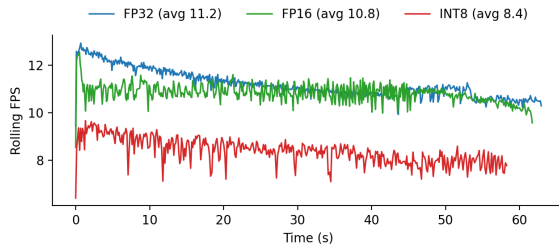


Figure 3: Rolling FPS over 60 s. INT8 has more frequent and deeper dips: its wider latency distribution shows up as jitter over time.

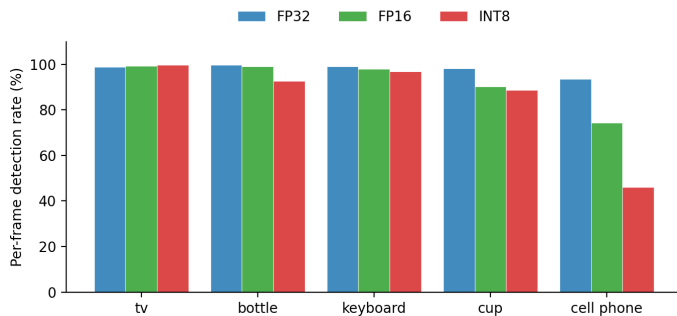


Figure 4: Per-class *per-frame detection rate* (%): the fraction of frames in which the class is detected at least once. Normalizing per frame removes the bias from different frame counts (FP32: 583, FP16: 541, INT8: 423). Under INT8, tv is unchanged and bottle, keyboard, and cup lose at most 10 pp, while cell phone drops from 93.7% to 45.6%.

Class	FP32	FP16	INT8
tv	99.0	100.0	99.8
bottle	99.8	99.8	93.1
keyboard	99.3	98.7	96.9
cup	98.5	90.9	88.7
cell phone	93.7	74.9	45.6

detection rate recommended by the assignment, parsed from `predictions_json`. Raw hit counts would favour FP32, which processes more frames (583 vs. 423 for INT8), so we report the *per-frame detection rate* (Figure 4). It shows three patterns:

- (i) **tv** is essentially unaffected, likely because a large, flat object tolerates the added noise.
- (ii) **bottle, keyboard, and cup** lose a small but consistent 2–10 pp.
- (iii) **cell phone** is the clear outlier: INT8 loses 48.0 pp, 5–20 $\times$ more than any other class. We hypothesize that its glossy, reflective surface changes appearance from frame to frame, which makes it especially sensitive to INT8 rounding noise.

5.4 System metrics: memory, thermal, and energy

Table 3 lists memory, thermal, and battery readings. INT8 uses the least memory; FP16 uses the most and was the only session to end in the serious thermal state. iOS does not expose raw battery current to apps, so we record `UIDevice.batteryLevel` at session start and report the change. With 1% resolution and one short session per precision, these readings (0 / 5 / 0 pp) cannot attribute energy use to precision; the FP16 drop needs longer, repeated runs to confirm.

Table 3: System metrics at the end of each session.

Metric	FP32	FP16	INT8
Peak resident memory (MB)	156.5	181.9	145.5
Thermal state at end	fair	serious	fair
Battery drop over the session (%)	0	5	0
Drain rate (%/min, proxy)	< 1	~ 3	< 1

5.5 Preprocessing: the larger hidden cost

The preprocessing fix in §4 (352 ms $\rightarrow$ 8 ms per frame) saves more time than the gap between the fastest and slowest precision (30 ms). On mobile, the data-conversion path can dominate end-to-end latency, so quantization should not be evaluated in isolation from preprocessing.

6 YOLOkémon: A Voice-Driven Capture Mini-Game

Motivation. Accuracy and latency are aggregated over frames, but users watch objects *across* frames. Frame-to-frame IoU measures this stability directly; a small game makes its cost felt, not only computed.

Game mechanics. Saying “I choose the cup” (or tapping the Pokéball) starts a capture. Each new frame updates a progress value $p \in [0, 1]$ using the IoU s between the target’s previous and current boxes:

$$p \leftarrow \text{clamp}\left(p + \frac{\Delta(s)}{75}, 0, 1\right), \quad \Delta(s) = \begin{cases} +1 & s > 0.85, \\ +0.3 & 0.60 < s \leq 0.85, \\ -0.5 & \text{otherwise.} \end{cases} \quad (2)$$

A capture needs at least 75 stable frames: about 2.5 s at 30 fps, or 7–9 s at the 8–11 FPS measured here. Reaching $p = 1$ fires `onCaptured`; falling from $p \geq 0.7$ back to 0 fires `onLateFailure`. Box jitter lowers s and slows or reverses progress.

Why voice? During a capture the user holds the phone steady, and small hand motions (we observed shifts of ~ 6 px at 640 px input) already lower IoU. Tapping the screen would add motion and disturb the very metric being measured; voice starts and stops a capture without touching the phone.

Pokédex. For each captured class, `pokedex.json` stores the set of precisions it was caught with, so one class can show several badges (Figure 1c).

Results. We ran one capture per precision (~ 76 `capture_iou` frames each). Mean IoU is 0.970 / 0.987 / 0.976 for FP32 / FP16 / INT8, a range of only 0.017 (Figure 5). Because every sample exceeds 0.85, no capture entered the slow or penalty branch: each needed essentially the minimum number of frames, so the capture times (9.6 / 7.5 / 10.0 s) reflect throughput during capture rather than box jitter. With one capture each, we cannot say whether the IoU differences persist across runs.

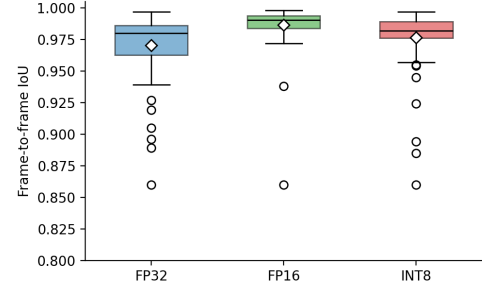


Figure 5: Frame-to-frame IoU during capture ($n = 76$ frames per precision; diamonds mark means). All samples exceeded 0.85; FP16’s distribution is the highest and narrowest.

7 Discussion and Conclusion

Best configuration. On the iPhone 11 CPU path, **FP16** offered the best storage–latency trade-off: 2.8% higher latency than FP32 at half the file size (5.21 MB). It is not free: cup and cell phone lose 7.6 and 18.8 pp, and its session had the highest memory and a serious thermal state, which one session per precision cannot separate from run-to-run variation. FP32 remains the safer choice when detection reliability matters more than storage. INT8 is the smallest (3.11 MB) but $1.38\times$ slower and halves cell-phone detection on this path; it becomes attractive only with an INT8-optimized execution path, e.g., an NPU delegate that can run the whole graph.

Where the cost shows up. (i) *Temporal stability* during capture differs little: mean IoU stays within 0.970–0.987. (ii) *Capture time* (7.5–10.0 s) mainly tracks throughput, since no capture triggered a penalty. (iii) *Per-frame detection rate* reveals a -48.0 pp loss on cell phone, while the other classes stay within 10 pp. (iv) *Energy* cannot be resolved with 1% battery readings. The aggregate metrics in Table 2 hide what (iii) exposes: a small drop in mean confidence can coexist with a large class-level loss.

Platform limits. iOS constrains three required or recommended metrics. (a) *Battery*: no API for raw current, so we use the `batteryLevel` change and accept its 1% resolution. (b) *Thermal*: `ProcessInfo.thermalState` only reports four levels, which we display with a colour code. (c) *Delegates*: YOLO26’s NMS-free decode head contains ops such as `WHILE`, `RANGE`, and `TOPK_V2` that the Metal and CoreML delegates do not support; they accepted only 28 and 48 of the 685 ops, respectively. In our tests both were slower than plain XNNPACK, plausibly because of graph partitioning and CPU–accelerator synchronization, so we fix the delegate to CPU/XNNPACK and show it on the dashboard. Finally, INT8 is calibrated on COCO128 rather than on our scene, which may contribute to its per-class losses.

Failure handling and future work. The app handles denied mic permission, misrecognized speech (§2), rapid precision toggling (serialized by the actor), returning to the camera, and losing the target mid-capture. Future work includes repeated sessions per precision, automatic throttling when the thermal state reaches *critical*, and YOLOkémon runs with stricter thresholds (IoU 0.92, a 1.0 s window).

References

- [1] F. A. Rahman, F. A. Pusparani, W. L. Yeoh, and O. Fukuda, “Smartphone-based seamless scene and object recognition for visually impaired persons,” *Information*, vol. 16, no. 9, p. 808, 2025. <https://doi.org/10.3390/info16090808>
- [2] Ultralytics, “YOLO26 model documentation,” accessed 2026. <https://docs.ultralytics.com/models/yolo26/>
- [3] TensorFlow, “Post-training quantization,” TensorFlow Lite documentation, accessed 2026. https://www.tensorflow.org/lite/performance/post_training_quantization
- [4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” in *Proc. IEEE CVPR*, 2016, pp. 779–788.