أكاديمية كاوست
KAUST ACADEMY

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Hunting Leopards with Half a Brain

Omar Alharbi, Abdullah Alhindi, Abdullah Alghanim, Rasheed Hamidaddin, Basil Alshareef, Basil Alshehri, Mojtaba Alshams

omar.alharbi@kaust.edu.sa, abdullah.alhindi@kaust.edu.sa, abdullah.alghanim@kaust.edu.sa, rasheed.hamidaddin@kaust.edu.sa, basilahmed.alshareef@gmail.com, basil.alshehri55@gmail.com, mojtaba.alshams@kaust.edu.sa

Abstract

The Arabian leopard (*Panthera pardus nimr*) is critically endangered, with only a small remnant population surviving across a vast range. Conservation depends on knowing where those animals still are, but the terrain is remote and off-grid, putting real-time detection beyond the reach of cloud infrastructure. We present a split inference architecture that distributes YOLO26L computation across two Raspberry Pi devices: a Pi Zero 2W runs the backbone in the field, while a Pi 3 runs the remaining layers with post-processing. To minimise network payload, we introduce a learned convolutional bottleneck that compresses the transmitted tensor by roughly two orders of magnitude. Measured on the deployed models over a real device-to-device link, the split runs end-to-end slightly faster than the full on-device model, with minimal loss in accuracy. We demonstrate that the learned bottleneck is what makes split inference beat plain edge deployment—on payload, on accuracy, and marginally on speed—for real-time wildlife monitoring.

Date: August 24, 2026

Code: https://github.com/EE-Omar/Hunting-Leopards-with-Half-Brain

Contents

1 Introduction **2**

1.1 Problem Statement . . . 2
1.2 Motivation for Split Inference . . . 2
1.3 Objectives . . . 2
1.4 Contribution Summary . . . 3

2 Related Work and Background **3**

2.1 Object Detection with YOLO . . . 3
2.2 Model Splitting and Edge Inference . . . 4
2.3 Quantization-Aware Training . . . 5
2.4 Learned Bottleneck and Feature Compression . . . 5

3 System Overview **7**

4 Methodology **9**

4.1 Data Collection and Dataset Construction . . . 9
4.2 Split Inference Architecture . . . 10
4.3 Bottleneck Design: Learned Compression . . . 12

5 Results and Discussion **14**

5.1 Accuracy: YOLO26L Primary Model . . . 14
5.2 On-Device Baseline: Full Model on the Pi Zero 2 W . . . 17
5.3 Raw Split-Point Evaluation . . . 18
5.4 Pipelined Execution: Experiment and Results . . . 19
5.5 Learned Bottleneck: Communication Reduction . . . 20
5.6 Limitations and Caveats . . . 21

6 Conclusion **22**

6.1 Summary of Contribution . . . 22
6.2 Practical Deployment . . . 22
6.3 Technical Insights . . . 22
6.4 Reproducibility . . . 22
6.5 Future Directions . . . 23

References **24**

1 Introduction

1.1 Problem Statement

The Arabian leopard is an endangered species, with approximately 200 individuals remaining in the wild. Protecting what is left of the population depends on knowing where the animals are and when they move, which in turn depends on continuous monitoring of the areas they inhabit. Those areas are remote and off-grid: there is no internet connectivity and no access to cloud computing infrastructure, so every stage of detection has to run on hardware that is physically present in the field.

The detection task itself is not reducible to motion sensing. The system has to separate the leopard from seven other classes that share the same habitat and the same camera frames (cheetah, hyena, nubian ibex, camel, cat, dog and person) and report a leopard present/absent decision per frame. This calls for a real object detector, and a single Raspberry Pi is not able to run one in real time on its own: the full YOLO26L model executed entirely on a single Raspberry Pi Zero takes approximately 2630 ms per frame, or 0.380 FPS. The detection workload therefore has to be organized differently, rather than simply being moved onto a smaller device.

1.2 Motivation for Split Inference

Our approach is to distribute the computation across two devices instead of one. The field node, a Raspberry Pi Zero with a camera, executes only layers 0–3 of the network, which account for 838,658 of the model's parameters, or roughly 3.4% of the total. This keeps the power draw of the device deployed in the wild as low as possible. The intermediate feature map is then sent over a TCP socket on a local wireless LAN to a Raspberry Pi 3, which acts as a local inference server and runs the remaining layers 4–23 together with post-processing. Both halves run on ONNX Runtime on CPU only, with no GPU, no accelerator, no internet and no cloud, and a single head device can serve several field nodes.

Compressing the model through quantization was not an available alternative, since INT8 export failed in our pipeline. What we compress instead is the transmitted feature map, using a small learned bottleneck trained to preserve the information the remaining layers actually need rather than to reconstruct the original activations faithfully. The design is governed by a three-way trade-off between detection accuracy, the size of the payload sent over the wire, and end-to-end latency.

1.3 Objectives

The goal of this work is a detection system that runs entirely on hardware present in the field, and five requirements follow from that goal.

First, the detection pipeline must be deployed across two devices rather than one: a Raspberry Pi Zero carries the backbone and sits with the camera in the monitored area, while a Raspberry Pi 3 carries the head and acts as a local inference server. Both halves run on CPU alone, with no accelerator and no cloud dependency of any kind.

Second, the end-to-end per-frame latency must come in below that of the full model running on the field device alone, so that the split is not paid for in speed.

Third, the payload placed on the wire between the two devices must be as small as possible. In an off-grid deployment the link is a local wireless connection shared by several field nodes, so every kilobyte transmitted per frame is paid for twice, once in latency and once in the energy budget of a battery-powered device.

Fourth, this saving must not be bought with detection quality. The system is required to hold

its accuracy close to the baseline obtained by running the complete model on the field device, and in particular to preserve performance on the leopard class, which is the only class the deployment actually acts on.

Fifth, the system must degrade gracefully rather than fail. If the link to the head device drops, the field node continues to detect on its own instead of going blind for the duration of the outage.

1.4 Contribution Summary

This report makes three contributions.

The first is a learned convolutional bottleneck with 16-channel reconstruction. A 4,112-parameter encoder on the field device compresses the intermediate feature map from 256 channels to 16 before transmission, and a 594,432-parameter decoder on the head device reconstructs the 256-channel tensor before the remaining layers consume it. The pair is fully convolutional, so the 40×40 spatial grid is preserved, the P3 skip connection stays aligned, and the network downstream of the split is left unchanged. The transmitted tensor drops from 1,638,400 bytes per frame to 25,600 bytes, a factor of 64×.

The second is the first configuration in which split inference beats the full on-device baseline. End to end the split runs at 0.386 FPS against 0.380 FPS for the full model on the Pi Zero, a marginal gain of +1.4%, while sending far less data. This matters because the obvious version of the idea does not work: a split that transmits the feature map uncompressed reaches only 0.346 FPS, 8.9% *slower* than simply running the whole model on the field device. The bottleneck is what turns split inference from a loss into a win.

The third is a measurement of what that costs. The accuracy loss is approximately 1.97 mAP, and it falls almost entirely on classes other than the target. For the leopard itself the AP50-95 falls only from 0.6267 to 0.6156, or −1.8% relative, which is negligible in deployment.

2 Related Work and Background

This section covers the four bodies of work our system is built on: the detector family we deploy, the split-inference literature that motivates cutting a network across two devices, the quantization machinery that lets us send an 8-bit tensor over the wire, and the learned-bottleneck literature that makes that tensor small enough to be worth sending at all.

2.1 Object Detection with YOLO

The YOLO family treats detection as a single regression pass over a dense grid of predictions rather than a propose-then-classify pipeline, which is what makes it the default choice for latency-sensitive deployments. Successive versions have progressively removed hand-designed components: YOLOv8 replaced anchor boxes with an anchor-free, decoupled head, and YOLOv10 removed non-maximum suppression (NMS) from the inference path entirely by training with consistent dual assignments—a one-to-many branch that supplies rich supervision during training, and a one-to-one branch that emits exactly one prediction per object at inference time [1]. YOLO26 continues this end-to-end design, and its one-to-one head is the version we deploy: it outputs a pre-decoded $(1, 300, 6)$ tensor of $[x_1, y_1, x_2, y_2,$ confidence, class_id] rows directly, with no NMS stage [2].

This property matters more for our system than it would for a single-device deployment. NMS is a data-dependent, poorly vectorized post-processing step whose cost varies with the number of candidate boxes in a frame, and it is awkward to represent inside an exported ONNX graph. Removing it means the second half of our split is a pure tensor graph with a deterministic

per-frame cost, which is what allows us to attribute measured latency cleanly to the backbone device, the link, and the head device rather than to a variable post-processing tail.

We use two variants of the same architecture for different purposes. YOLO26L is the primary model, chosen for accuracy (55.0 mAP50-95 on COCO); YOLO26n is the fallback, roughly 6× smaller at 2.4M parameters and 40.9 mAP50-95. The gap between them, 14.1 mAP, is the price of the graceful-degradation path described in Section 5.4: it is a large loss in absolute terms, but a better outcome than no detection at all when the head device is unreachable.

2.2 Model Splitting and Edge Inference

The idea of cutting a network between a weak device and a stronger one was established by Neurosurgeon [3], which partitions a DNN at layer granularity using regression models that predict the per-layer latency and energy cost on each side, then selects the cut point at runtime. Its central finding is that fine-grained partitioning consistently beats both device-only and cloud-only execution, and—importantly for us—that the best split point is not a fixed property of the architecture but shifts with layer type, network conditions, and server load. Edgent [4] extended this by jointly choosing the split point and the effective depth of the network via early exits, optimizing accuracy under a latency budget rather than treating the two as independent objectives. Matsubara et al. [5] survey this space and organize it into split computing and early exiting; they also note that the large majority of the literature targets image classification rather than detection, and assumes hardware considerably stronger than the class of device we deploy on.

Two structural facts about CNNs motivate cutting early. First, parameters are heavily concentrated in the deep layers: in YOLO26L, layers 0–3 account for 838,658 parameters, or roughly 3.4% of the model, so the field device holds a small fraction of the weights and a correspondingly small memory footprint. Second, and working against the first, intermediate feature maps in the early layers are frequently *larger* than the input image that produced them. Shao and Zhang [6] name this the data amplification effect and show that because of it, split-point selection alone—what they call the rudimentary approach—yields limited latency gains; only combining the split with communication-aware compression dominates the baselines on both computation and communication cost. Our own measurements reproduce this directly. The layer-3 output tensor, of shape $(1, 256, 40, 40)$ at a 320×320 input, occupies approximately 1600 KB in FP32, and a split that transmits it uncompressed runs at 0.346 FPS—8.9% *slower* than simply running the whole model on the Pi Zero. Splitting without compression is a net loss.

The detection-specific literature makes the same point more sharply. Matsubara and Levorato [7] performed the first layer-wise analysis of split computing for complex object detectors and found that naive splitting does not reduce inference time for R-CNN-family models, because detection backbones lack small natural bottlenecks in convenient early positions; injecting a trained bottleneck is what makes split computing viable for detection at all. Bakhtiarnia et al. [8] approach this from the opposite direction, showing that some modern architectures (EfficientNet variants) contain enough natural bottlenecks to switch among them dynamically with no retraining—but their method explicitly does not apply to architectures lacking early natural bottlenecks, which still require artificial injection. For YOLO specifically, Yuan et al. [9] report that intermediate tensor size varies by roughly 50× across candidate split points in YOLOv2, and that YOLO-family architectures benefit less from split-point optimization than a plain classifier such as VGG16. Taken together, these results are the justification for our design choice: for a YOLO backbone, selecting a good layer is necessary but not sufficient, and a bottleneck has to be engineered in deliberately.

Two recent systems split YOLO on real hardware and are the closest published precedents

to ours. Lee et al. [10] build a reconfigurable architecture supporting multiple split points within one set of shared weights, reaching 30 FPS with YOLOv5 on an NVIDIA Jetson TX2 over WiFi—2.6× faster than edge-only and 1.6× faster than cloud-only execution. Ma et al. [11] split YOLOv11 and compress the transmitted features with Q-DeltaMask, a quantized delta-bitmask encoding that transmits only the difference between consecutive frames' features, achieving up to 92% transmission reduction and 4.6× throughput for 1.9 percentage points of mAP. Both differ from our setting in ways that matter. Q-DeltaMask's entire advantage rests on frame-to-frame redundancy in a continuous video stream; a camera trap captures sparse, motion-triggered bursts in which consecutive frames may be minutes apart and contain a moving animal, so temporal delta encoding has little to exploit. And a Jetson TX2 is orders of magnitude more capable than a Raspberry Pi Zero 2W. The combination of a single-shot detector, a per-frame learned bottleneck, and genuinely ultra-constrained CPU-only hardware on both sides of the link remains unaddressed.

2.3 Quantization-Aware Training

Uniform affine quantization maps a floating-point activation x to an unsigned 8-bit integer through a scale s and a zero point z:

$$q = \mathrm{clip}\left(\left\lfloor \frac{x}{s} \right\rceil + z,\ 0,\ 255\right), \qquad \hat{x} = s\,(q - z), \tag{1}$$

a scheme formalized for integer-arithmetic-only inference by Jacob et al. [12]. Applied naively after training, it is a blind operation: the network never sees the error it introduces. Quantization-aware training (QAT) instead simulates the round trip during the forward pass so the weights can adapt to it. The obstacle is that the rounding operator has zero gradient almost everywhere, which would block backpropagation entirely. The straight-through estimator (STE) [13] resolves this by treating rounding as the identity function in the backward pass, passing the gradient through unchanged within the clipping range. This is what allows our encoder to learn weights that are robust to the 8-bit round trip rather than merely tolerating it.

The remaining choice is calibration: how to set s and z. Min/max calibration takes the observed extremes of the activation distribution, which makes it maximally sensitive to outliers—a single extreme value stretches the scale and wastes quantization levels across the region where the bulk of the data actually lives. Percentile and entropy-based calibration instead clip a small tail fraction in exchange for finer steps in the dense region, which is typically a net reduction in quantization error when the distribution is long-tailed [14]. Post-SiLU activations are precisely this case: bounded below near -0.278, unbounded above, with a heavy right tail. We therefore calibrate at the 99.9th percentile, clipping approximately 0.166% of activation values and yielding a scale of 0.051089 and a zero point of 133.

It is worth distinguishing this from whole-model INT8 quantization, which we attempted and which failed to export in our pipeline. What we quantize is not the model but the transmitted tensor—a much narrower intervention, applied at a single point in the graph, whose cost we can isolate and measure.

2.4 Learned Bottleneck and Feature Compression

If the transmitted tensor has to shrink, the question is what should do the shrinking. The answer developed across a decade of work is a small trainable encoder-decoder pair straddling the split point. BottleNet [15] introduced the idea, inserting an autoencoder-style module at the cut together with a training procedure to recover the accuracy lost to lossy compression, and established that a purpose-built compression architecture beats sending raw or generically

compressed features. BottleNet++ [16] refined it into a lightweight convolutional encoder-decoder trained end-to-end alongside a simulated channel layer, reporting up to 64× bandwidth reduction for less than 2% accuracy loss—and, critically for split-point reasoning, showing that a stronger bottleneck permits splitting *earlier* in the network without paying the usual communication penalty. BottleFit [17] attacked the cost of the bottleneck itself, reducing the on-device head model by up to 83× and cutting power and latency by 40% and 62% respectively on a Raspberry Pi; it is the closest real-hardware precedent for the kind of numbers we report. Head Network Distillation [18] offers the main alternative training strategy, distilling the on-device head against the unmodified network as a teacher rather than training the bottleneck end-to-end.

Our own bottleneck sits within this lineage: a $256 \rightarrow 16$ channel encoder (Conv2d, $k = 1$, 4,112 parameters) on the Pi Zero and a 594,432-parameter decoder on the Pi 3, fully convolutional so that the 40×40 spatial grid is preserved and the P3 skip connection stays aligned. The asymmetry is deliberate and follows BottleFit's principle: the field device pays almost nothing, and the reconstruction cost is pushed onto the device that can afford it.

Two results explain why this works better than its compression ratio would suggest. The SC2 Benchmark [19] establishes that supervised feature compression generally beats naive input compression at equivalent bitrates, and that split point, reference model, and encoder-decoder architecture jointly determine the achievable rate-distortion tradeoff—none of these choices is independent of the others. Shao et al. [20] supply the theoretical account, deriving feature compression from information bottleneck theory: the encoder should discard information relevant only to reconstructing the input while preserving information relevant to the task. This is the framework in which our own result becomes interpretable. A 16× channel reduction costs only 0.0197 mAP50-95, and the target class loses just 0.011 AP—reconstruction fidelity and detection quality are simply not the same quantity.

It is worth being precise about the mechanism here, because it is easy to overstate. Our Stage-1 training minimizes an MSE reconstruction objective over cached layer-3 activations; we did not fine-tune against detection loss. The bottleneck is therefore not *trained* to be task-oriented in the sense of [20]. What the result shows instead is that a $256 \rightarrow 16$ projection learned purely for reconstruction already retains enough task-relevant structure that the layers downstream of the split barely register the loss—a weaker claim than task-oriented training would support, but a useful one, and an indication that an explicitly detection-driven objective still has headroom in it. We return to this in Section 6.5.

Finally, Matsubara and Levorato [21] combine bottleneck compression with an on-device filter that discards frames containing no object of interest before transmission. For a wildlife camera trap, where the overwhelming majority of frames are empty, this is the natural complement to what we have built—compressing a frame that never needed to be sent is wasted work—and we return to it in Section 6.5.

3 System Overview

The proposed system is a distributed architecture designed for detecting Arabian leopards using multiple camera nodes deployed across a monitored wildlife area. The main objective of the system is to distribute the computational workload of the object detection process between resource-constrained field devices and a central processing device. Each field node consists of a Raspberry Pi Zero 2 W connected to a camera, while a Raspberry Pi 3 operates as the central processing device. The system uses a split-inference approach, in which the YOLO26 Large model is divided between the Raspberry Pi Zero 2 W devices and the central Raspberry Pi 3.

The use of a distributed architecture allows cameras to be positioned at different locations throughout the monitored area while sharing a central device for the later stages of inference. This is particularly suitable for wildlife monitoring, where cameras may need to cover a relatively large area and operate using devices with limited computational resources. Rather than requiring every field node to execute the complete YOLO26 Large model, the computational workload is divided between the field nodes and the central Raspberry Pi 3.

When a camera captures a frame, the Raspberry Pi Zero 2 W begins the first stage of the inference process. The captured image is provided as input to the backbone of the YOLO26 Large model. The backbone extracts visual features from the image and transforms the original pixel-level information into an intermediate feature representation. These extracted features contain information required by the subsequent stages of the object detection model.

The intermediate features generated by the backbone are then passed through an encoder located on the Raspberry Pi Zero 2 W. The purpose of the encoder is to compress the feature representation before network transmission. This reduces the amount of intermediate data that needs to be communicated between the field device and the central processing device. Consequently, the Raspberry Pi Zero 2 W is responsible for image acquisition, initial feature extraction, and feature compression.

After the encoding stage, the resulting compressed feature tensor is transmitted from the Raspberry Pi Zero 2 W to the Raspberry Pi 3 through the local wireless network. Instead of transmitting the original camera frame to the central device, the system communicates the compressed intermediate representation produced by the first part of the inference pipeline. This creates a clear separation between the processing performed at the field node and the processing performed centrally.

The Raspberry Pi 3 acts as the central processing device for the distributed system. Once a compressed feature tensor is received from a field node, it is passed through the decoder. The decoder reconstructs the feature representation required by the remaining part of the YOLO26 Large model. The reconstructed features are subsequently provided to the detection head, which performs the final stages of the object detection process and produces the detection output.

The final output is used to determine whether an Arabian leopard is present or absent in the captured frame. In this way, the complete inference process is performed collaboratively between the Raspberry Pi Zero 2 W and Raspberry Pi 3 rather than being executed entirely on a single device. The field node handles the early processing stages, while the central device handles feature reconstruction and the remaining detection operations.

Another important characteristic of the proposed architecture is its ability to support multiple camera nodes. Several Raspberry Pi Zero 2 W devices can be deployed at different locations

within the monitored wildlife area, with each device connected to its own camera. Every field node independently captures frames, executes its local portion of the YOLO26 Large model, compresses the resulting feature representation, and sends the compressed tensor through the local wireless network.

The central Raspberry Pi 3 therefore serves as a shared processing point for the deployed camera nodes. Feature tensors received from different field nodes follow the same decoding and detection pipeline. This architecture avoids requiring a separate central processing device for every camera node and provides a common inference point for the monitored area. It also separates the system into clearly defined field-processing and central-processing components.

From an architectural perspective, the system can therefore be divided into three main stages. The first is the **field processing stage**, consisting of the camera, Raspberry Pi Zero 2 W, YOLO26 Large backbone, and encoder. The second is the **communication stage**, in which the compressed feature tensor is transferred through the local wireless network. The third is the **central processing stage**, consisting of the Raspberry Pi 3, decoder, and detection head. Together, these stages form the complete distributed inference pipeline.

The overall processing pipeline can be summarized as:

Camera → Raspberry Pi Zero 2 W → Backbone → Encoder → Compressed Feature Tensor → Local Wireless Network → Raspberry Pi 3 → Decoder → Detection Head → Leopard Present / Absent

Figure 1 illustrates the complete system architecture and the flow of data from the distributed camera nodes to the central Raspberry Pi 3. It shows how multiple field nodes perform the initial stages of YOLO26 Large inference before transmitting compressed intermediate features, while the central Raspberry Pi 3 performs feature reconstruction and the remaining detection stages.

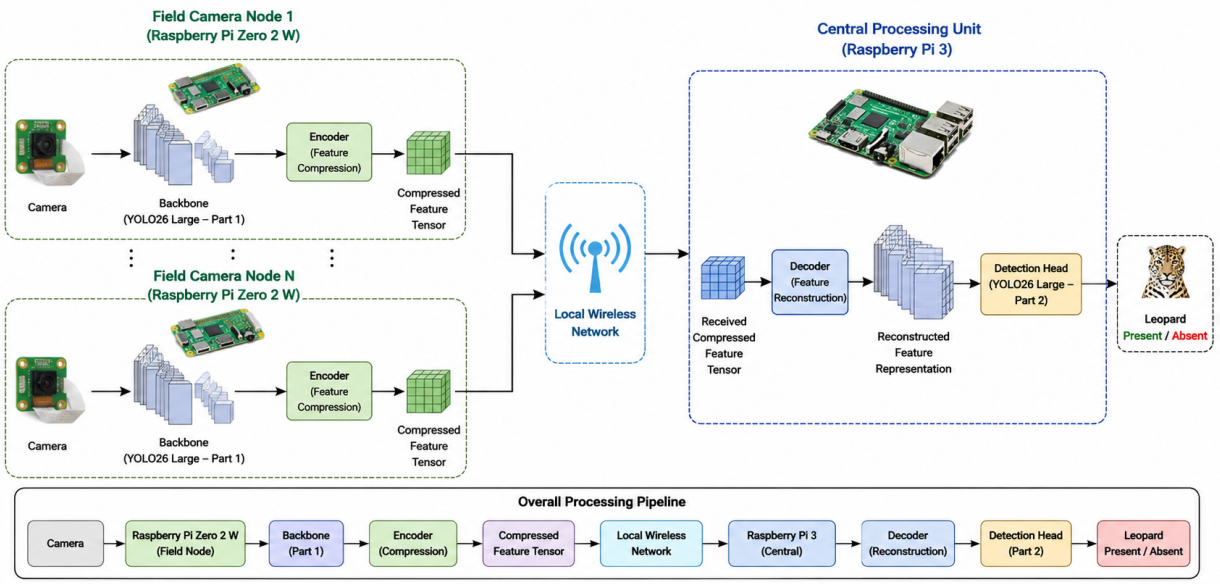


Figure 1 End-to-end system architecture for Arabian leopard detection using distributed split inference. Multiple Raspberry Pi Zero 2 W camera nodes perform the backbone and encoder stages and transmit compressed feature tensors over the local wireless network to a central Raspberry Pi 3, which performs decoding and the remaining detection stages.

4 Methodology

4.1 Data Collection and Dataset Construction

4.1.1 Rationale

The detector has one deployment target: the Arabian leopard. The dataset is weighted heavily towards that class for exactly that reason, it is the only object the system is required to detect and report.

The remaining classes are not detection targets in their own right. A camera deployed in leopard habitat sees mostly *other* things: ibex, camels, feral cats and dogs, herders, and empty desert. These classes were included and labelled so that they function as **hard negatives** — examples that teach the detector what a leopard is *not*. Cheetah and hyena matter most here, since both share the spotted or mottled coat pattern that dominates the leopard class; a single-class leopard detector has no mechanism for learning that a cheetah is not a leopard, because it has never been shown one.

Alongside these, background images containing no annotated object at all were included, so the model learns to stay silent — which is what it will be doing for the overwhelming majority of deployed frames.

The objective of this composition was to **reduce false positives**, and it is targeted at that rather than at balanced multi-class performance. The class distribution is consequently skewed, and the non-leopard classes are represented only well enough to serve their role as negatives. Collecting a larger and more evenly balanced set across all classes is left as future work.

4.1.2 Sources

No single public dataset covers this composition. The dataset was assembled from three kinds of source:

1. **LILA BC** (Labeled Information Library of Alexandria: Biology and Conservation) [22] — the primary leopard source, used in both dataset versions. It was distributed in COCO format and converted to Ultralytics YOLO format with `ultralytics.data.converter.convert_coco`, becoming the `v1` single-class dataset. It was originally split into train and validation only; the validation split was halved with a fixed seed (42) to produce a held-out test split.
2. **Seven datasets from Roboflow Universe**, all CC BY 4.0, listed in Table 1. These supply every non-leopard class, plus additional leopard imagery layered on top of the LILA BC leopards.
3. **Negative sample images** — desert, vegetation, rock and empty scene photographs with no annotated object — collected separately to act as background frames.

4.1.3 Merging

Every source numbers its classes from zero in its own local namespace, so the class id `0` means `leopard` in one folder and `cheetah` in another. Merging them naively would silently mislabel everything. The merge script therefore reads each source's `data.yaml`, maps local class names onto a single global list, and rewrites the first field of every label line accordingly:

```
0=leopard, 1=cheetah, 2=hyena, 3=nubian_ibex,
      4=camel, 5=cat, 6=dog, 7=person
```

Negative samples were carried through with empty `.txt` label files, which is how Ultralytics represents a background image.

Table 1 Source datasets. All Roboflow Universe entries are CC BY 4.0.

Source	Classes taken	Provides
LILA BC [22]	leopard	leopard
Leopard [23]	leopard	leopard (added on top of LILA BC)
Cheetah [24]	Cheetah	cheetah
Hyena [25]	hyena	hyena
Nubian Ibex [26]	ibex	nubian_ibex
Camel [27]	camel	camel
Dogs & Cats [28]	cats, dogs	cat, dog
People [29]	person	person
Negative Samples	—	background

4.1.4 Model Selection and Training

The deployed detector is **YOLO26-Large**, fine-tuned from the Ultralytics pretrained checkpoint on the merged dataset. A large model was chosen deliberately: the premise of this work is that the detector is too heavy for the field device to run alone, and a model that comfortably fits on a Pi Zero would make the split architecture decorative.

Training used 640×640 input, batch size 16, and mixed precision on a single GPU, with early stopping (`patience` = 10) monitored against validation mAP. The checkpoint selected for deployment is the best-scoring epoch by that criterion.

The trained model is exported to ONNX at the 320×320 deployment resolution. All accuracy figures reported in Section **??** are measured on that exported artefact rather than on the training checkpoint, because it is what actually runs on the devices.

4.2 Split Inference Architecture

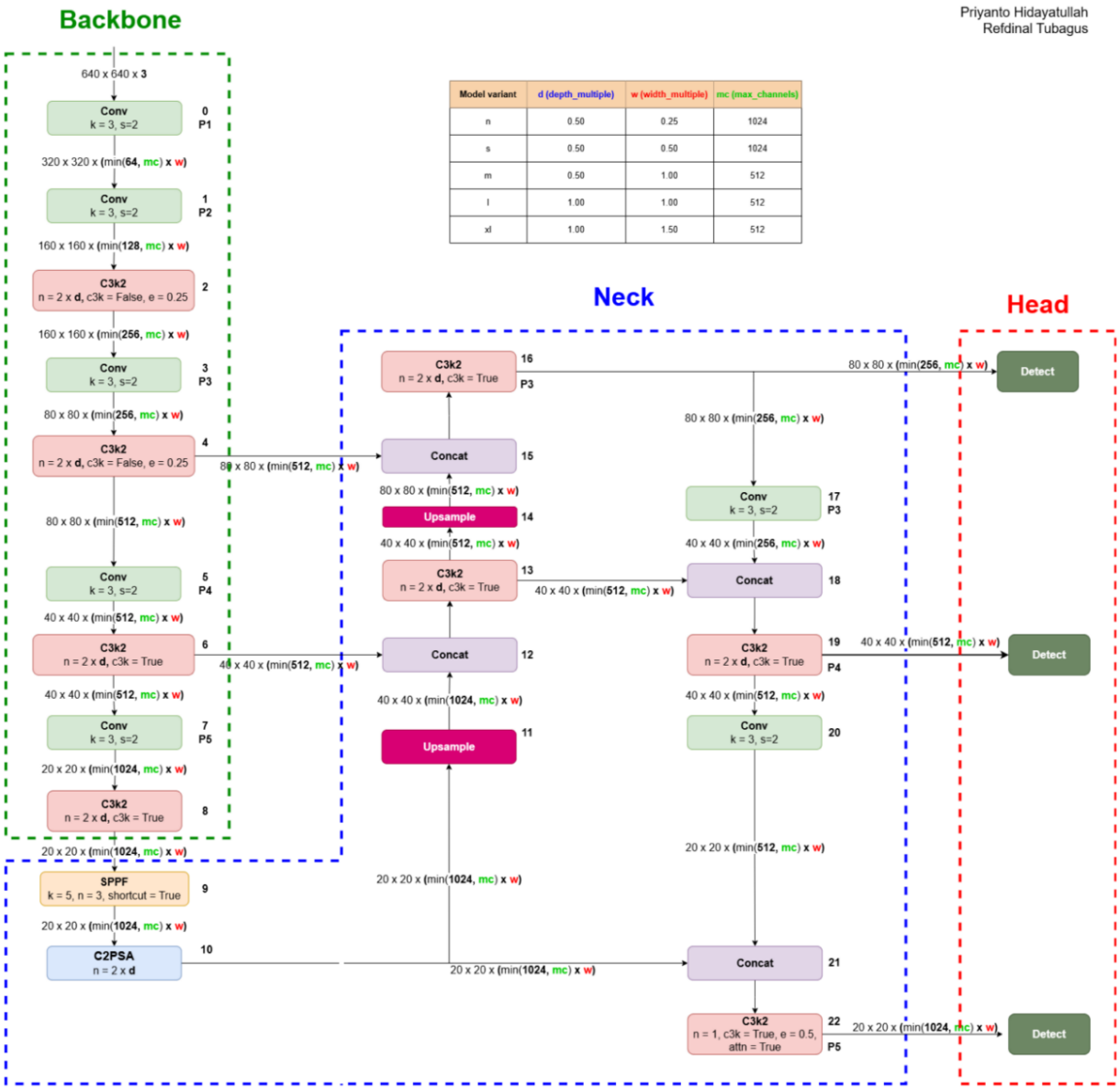
The split point determines both what crosses the network each frame and how the model's compute divides between the two devices. It was therefore selected empirically rather than assumed.

Constraint: single-tensor cuts.. A cut is only usable if exactly one tensor crosses the head/tail boundary. YOLO26's neck consumes backbone features through FPN skip connections: the P3, P4 and P5 paths originate at backbone layers 4, 6 and 10 and are consumed by concatenation nodes further down the graph hidayatullah2026yolo26comprehensivearchitectureoverview. Cutting downstream of a layer whose output is reused later would force both the current activation *and* the retained skip tensor across the link, multiplying the payload and complicating the transport protocol. Candidates were therefore restricted to boundaries with a single crossing tensor, which confines the admissible region to the early backbone—up to the output of layer 4, before the P3 skip is retained—plus the trivial cuts at the detection head. Figure 2 marks this region and the selected cut on the YOLO26 architecture.

Candidate enumeration and export.. Every tensor in the exported ONNX graph was tested for the single-crossing property, yielding 18 admissible candidates spanning `/model.0/conv` to `/model.23/Transpose`. For each, a head/tail model pair was extracted with `onnx.utils.extract_model`. Every pair was verified numerically equivalent to the unsplit model before profiling, so that any accuracy difference observed later could be attributed to the bottleneck rather than to the split itself.

Profiling protocol.. Each candidate was deployed on the target pair—YOLO26-Large at 320×320, head on the Raspberry Pi Zero 2W, tail on the Raspberry Pi 3, communicating over TCP on a

Priyanto Hidayatullah
Refdinal Tubagus



Model variant	d (depth_multiple)	w (width_multiple)	mc (max_channels)
n	0.50	0.25	1024
s	0.50	0.50	1024
m	0.50	1.00	512
l	1.00	1.00	512
xl	1.00	1.50	512

Figure 1: YOLO26 Architecture Diagram

Figure 2 YOLO26 architecture, adapted from [30]. The admissible split region is confined to the early backbone: layers 4, 6 and 10 are consumed by the neck's concatenation nodes, so any cut beyond the output of layer 4 would force more than one tensor across the split boundary. The selected cut is marked at the output of layer 3.

local wireless link. For each candidate we recorded the payload transmitted per frame, the peak resident set size (RSS) of each half, and per-stage latency (head compute, transfer, tail compute), discarding warm-up frames so that only steady-state behaviour was measured.

Selection criteria.. Two criteria were fixed in advance:

1. **Payload per frame**—the wireless link is the shared resource, so the smallest transmitted tensor is preferred.
2. **Peak head RSS**—the Pi Zero 2W has 512 MB of RAM, so the head must fit with margin alongside the ONNX runtime and frame buffers.

Latency was recorded for context but not used as a primary criterion, since a raw-tensor split is dominated by transfer cost—precisely the cost the learned bottleneck is designed to remove.

The selected cut, `/model.3/act/Mul_output_0`, is the tensor into which the learned bottleneck (Section 4.3) was inserted. Measured outcomes are reported in Section **??**.

4.3 Bottleneck Design: Learned Compression

4.3.1 Problem

Splitting the network at layer 3 places the tensor `/model.3/act/Mul_output_0` on the link. At 320×320 input its shape is $(1, 256, 40, 40)$: a dense activation map that is large relative to the capacity of a wireless link between two single-board computers. Transmitted raw in either fp32 or fp16, it makes the transfer stage the dominant cost of the pipeline, to the point that splitting can be slower than running the whole model on the field device. The split point is not the problem; the payload is.

The approach taken instead is a small *learned* autoencoder, trained specifically on this layer's activations, which reduces the channel dimension and transmits the result as uint8.

4.3.2 Architecture

The bottleneck is fully convolutional and preserves the 40×40 spatial grid—no pooling, no flattening, no fully connected layer. This is a hard constraint rather than a stylistic choice: the reconstructed tensor is fed back into the unmodified remainder of the network, whose skip connections require the spatial layout to line up exactly.

Table 2 Bottleneck components. Only the encoder runs on the field device.

Part	Definition	Device
Encoder	`Conv2d(256, 16, k=1)`	Pi Zero
FakeQuant	uint8 affine, percentile-calibrated, STE (training only)	—
Decoder	`Conv2d(16, 256, k=1)` → `SiLU` → `Conv2d(256, 256, k=3)`, residual add	Pi 3

4.3.3 Quantization

Reducing the channel count alone still leaves an fp32 tensor to transmit. Sending the latent as uint8 divides that by four again, but only if the network can tolerate 8-bit values. Rather than quantizing a trained network after the fact, quantization is simulated *during* training: the `FakeQuant` module performs the exact round-trip the pipeline performs, and a straight-through estimator passes gradients through the non-differentiable rounding. The encoder therefore learns to produce a latent that survives 8 bits.

4.3.4 Qunatize Format

Quantization is performed in NumPy on the field device, outside the ONNX graph. The encoder and decoder ONNX models both remain fp32 and exchange a uint8 buffer. This keeps quantization a *tensor* format rather than a *model* format, which avoids depending on INT8 model export and keeps the fp32 path available as a fallback.

The existing frame header carries a dtype byte, already reserved for this: `dtype = 1` is legacy fp32, and `dtype = 2` is uint8, followed by the scale and zero-point.

4.3.5 Training Procedure

Training is **self-supervised reconstruction only**. The YOLO model is frozen throughout; no labels and no detection loss are involved. The objective is the mean squared error between the original 256-channel layer-3 tensor and its reconstruction:

$$\mathcal{L} = \left\| x - \mathrm{dec}\big(Q(\mathrm{enc}(x))\big) \right\|_2^2 .$$

Because the backbone is frozen, its activations never change between epochs. Layer-3 activations for the training images were therefore cached once to an fp16 memory-mapped file, and the training loop reads directly from that cache. This is what made CPU-only training practical.

Preprocessing for the cache deliberately mirrors the deployment path (`cv2.resize` → BGR→RGB → /255) rather than the Ultralytics letterbox pipeline, so the bottleneck is trained on the same distribution of activations it will encounter in the field.

Table 3 Bottleneck training configuration.

Cached images	3,000 (train split, seed 42)
Epochs	40
Optimiser	AdamW, cosine schedule
Learning rate	2×10^{-3}
Batch size	32
Validation	5% hold-out
Model selection	best epoch by validation relative L_2
Hardware	CPU-only laptop

4.3.6 Export and Verification

The encoder and decoder are exported to ONNX on their own and then *composed* onto the already-verified split halves with `onnx.compose.merge_models`, rather than re-exporting the whole network.

Accuracy verification was run **end-to-end** through a real uint8 round-trip on the validation split, so the figures reported in Section **??** are a property of the deployed artefacts rather than of a PyTorch approximation of them.

5 Results and Discussion

5.1 Accuracy: YOLO26L Primary Model

5.1.1 Baseline Detection Performance

The baseline YOLO26L model was first evaluated on the validation set to establish its detection performance before evaluating split inference and feature compression. Figure 3 shows the precision–recall curves for the eight classes. The target class, leopard, achieves a strong AP@0.5 of 0.988. However, performance differs substantially between classes, partly because the dataset is imbalanced. Therefore, results for classes with only a small number of validation instances should be interpreted cautiously.

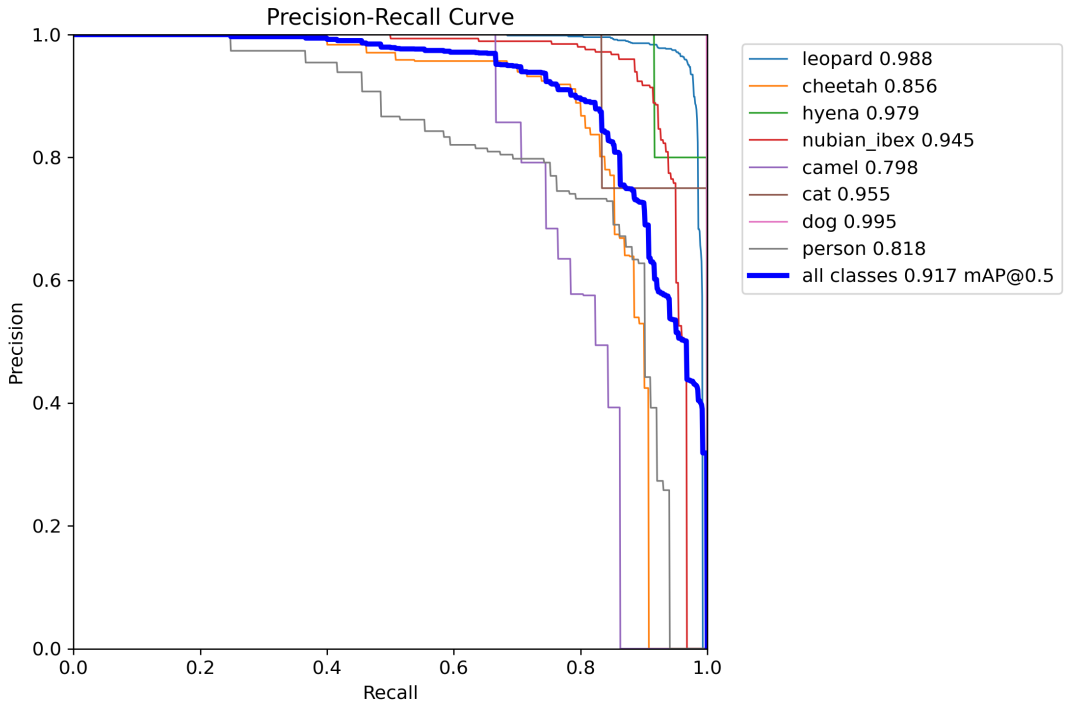


Figure 3 Precision–recall curves of YOLO26L for all classes on the validation set.

5.1.2 Qualitative Detection Performance

Figure 4 presents representative detections on infrared night images, which are particularly relevant to the intended camera-trap deployment. The model detects leopards under partial occlusion, near the edge of the image, and under low-contrast infrared conditions. These examples demonstrate that the detector remains effective in challenging visual conditions that are expected during deployment.

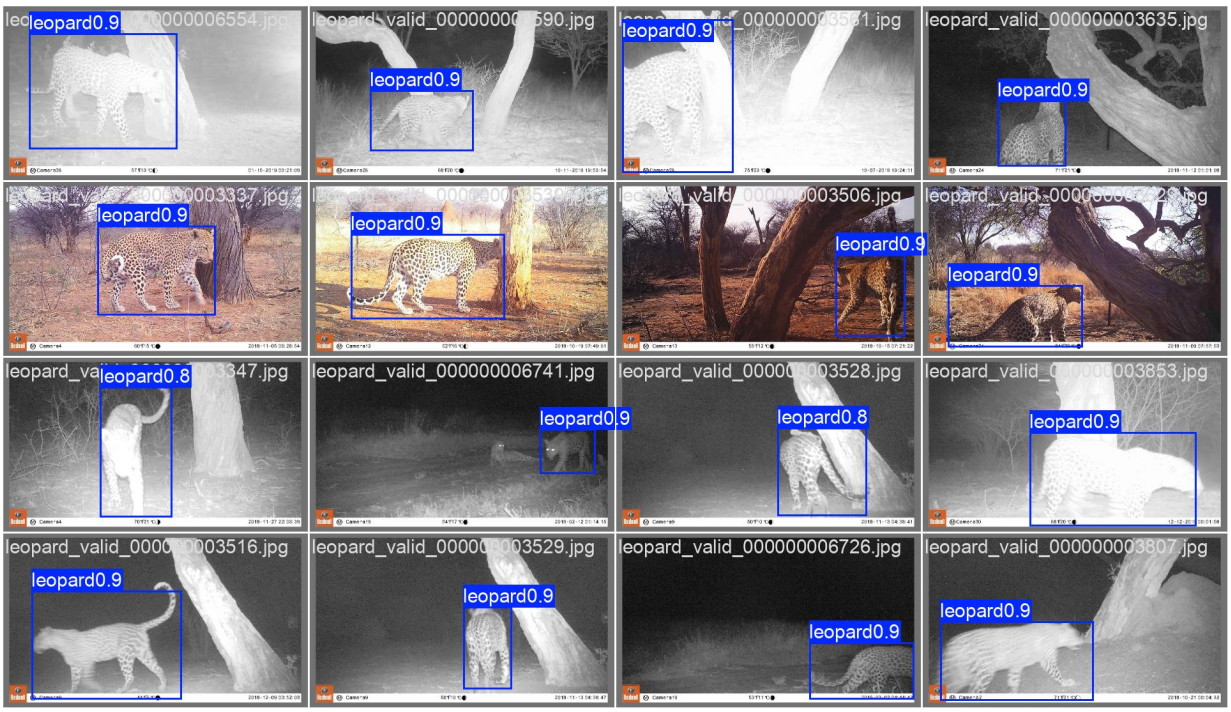


Figure 4 Examples of YOLO26L detections on infrared night images.

5.1.3 Error Analysis

Figure 5 shows the normalised confusion matrix. Class-to-class confusion is generally limited, indicating that most errors arise from the object–background boundary rather than confusion between visually similar animal species. This distinction is important for deployment because missed detections and background-related false positives are the primary detection failure modes.

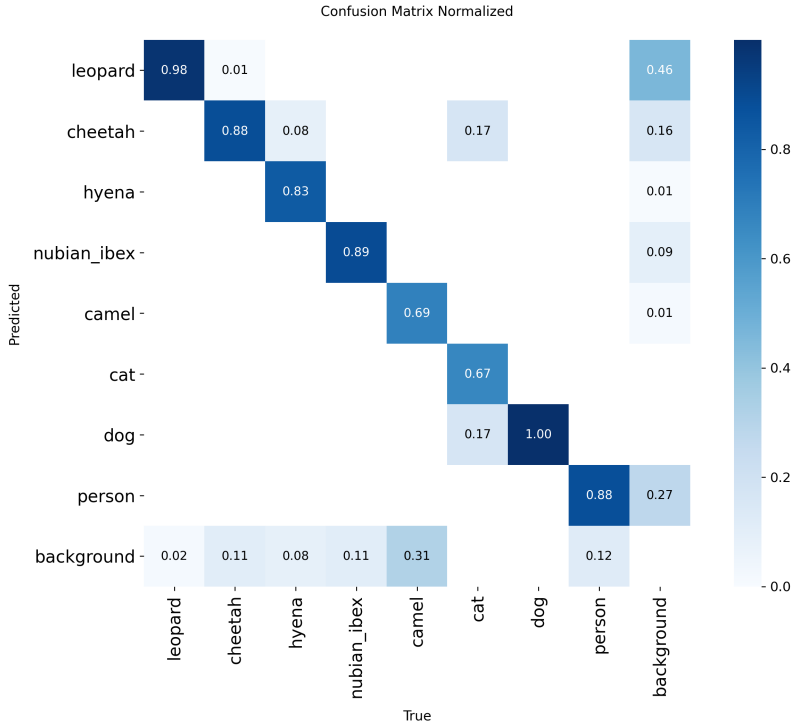


Figure 5 Normalised confusion matrix. Class-to-class confusion is minimal; the dominant error mode is background, which accounts for 0.46 of leopard and 0.27 of person predictions.

5.1.4 Accuracy After Bottleneck Compression

The accuracy impact of the learned bottleneck was evaluated on 1,212 validation images using the deployed ONNX models and a real UINT8 communication round-trip. The baseline and raw split produce identical predictions, confirming that raw splitting itself does not change the detector output. With the learned bottleneck, mAP50–95 decreases from 0.6535 to 0.6338, corresponding to an absolute reduction of 0.0197. For the target leopard class, AP50–95 decreases from 0.6267 to 0.6156, an absolute reduction of only 0.011. The bottleneck therefore trades a small reduction in detection accuracy for a substantial reduction in communication cost. Because the raw split is bit-exact, the observed accuracy difference is attributable to feature compression and quantization rather than to the split operation itself.

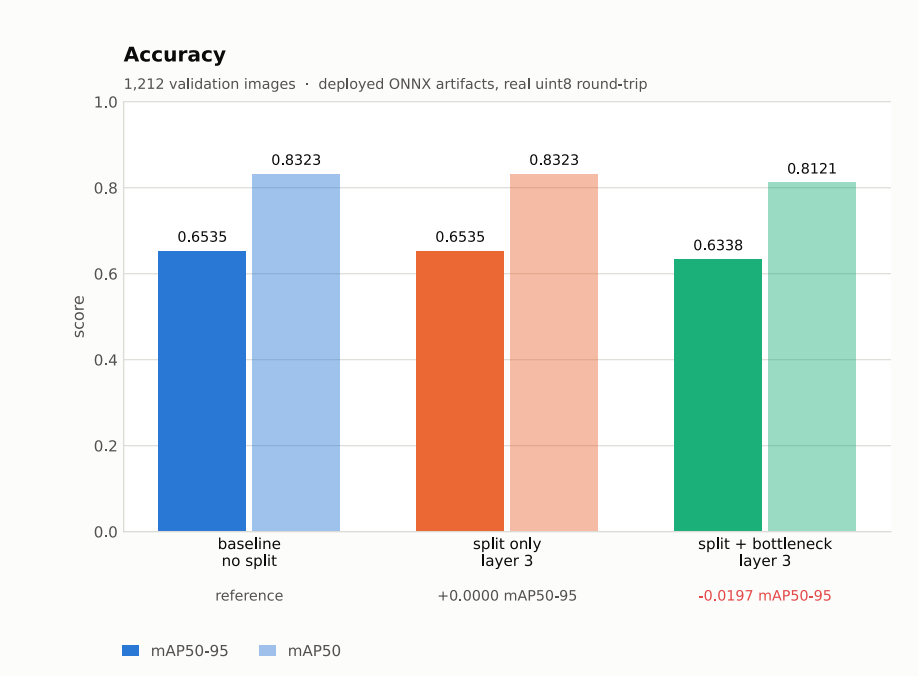


Figure 6 Normalised confusion matrix for YOLO26L on the validation set.

5.2 On-Device Baseline: Full Model on the Pi Zero 2 W

The entire YOLO26L model was executed on a single Pi Zero 2W without network communication or assistance from the Pi 3. This configuration, referred to as B1, provides the sustained on-device reference for evaluating the pipelined split system. Five independent runs of 100 frames were performed. The first 20 frames were discarded as warm-up and the final 10 frames as drain frames, leaving 70 steady-state frames per run. A USB camera workload was active during the experiment to represent the intended deployment environment.

Table 4 On-device baseline (B1): YOLO26L running entirely on the Pi Zero 2W. Five independent runs of 70 measured frames each. Latency percentiles are computed over the per-frame records; throughput is measured from completion timestamps across the steady-state window.

Run	n	Mean (ms)	SD (ms)	p50 (ms)	p95 (ms)	FPS	Peak T (°C)
1	70	2705.1	225.4	2640.3	2920.1	0.3374	82.2
2	70	3053.5	210.1	3116.7	3391.1	0.3031	83.3
3	70	3006.9	318.0	2995.6	3189.8	0.3069	82.7
4	70	2965.1	154.0	3004.7	3135.9	0.3189	82.7
5	70	2984.3	173.6	3033.8	3201.9	0.3167	82.7
All	**350**	**2943.0**	**253.9**	**2975.4**	**3210.2**	**0.3166**	**83.3**

The sustained baseline achieves **0.3166 FPS**, with a mean latency of **2943 ms per frame**. The peak resident memory usage is approximately 314 MB out of the 463 MB available on the device. Model loading and the first inference were excluded from the steady-state measurements. All 350 measured frames reproduced the reference detections exactly. However, the Pi Zero 2W reached approximately 82–83 °C during sustained execution, above the 80 °C soft-throttle threshold. Consequently, these measurements represent sustained operation under the actual thermal load of the deployment rather than best-case cold-start performance.

5.3 Raw Split-Point Evaluation

Before applying the learned bottleneck, the model was evaluated across the measured split positions to characterise how computation and communication costs vary throughout the network. Each configuration divides inference between the Pi Zero 2W and the Pi 3, with the intermediate activation transferred over the wireless LAN. The sweep shown in Figures 7 and 8 includes the evaluated layer positions without removing positions that were subsequently classified as non-valid for a practical single-tensor split. This provides a complete view of the cost landscape, while the validity constraint remains in effect when identifying deployable split points.

5.3.1 Communication Cost Across Split Points

Figure 7 shows the total amount of data crossing the split boundary for each evaluated layer. The early single-tensor candidates have substantially different payload sizes: Layers 0, 1, 2, 3, and 4 require approximately 6400, 3200, 6400, 1600, and 3200 KB per frame, respectively. The result highlights the importance of considering communication cost when selecting a split point. Although Layer 23 has an exceptionally small transmitted tensor, it is retained only as a control because it places essentially the complete model on the Pi Zero 2W and therefore does not constitute a useful computational split. Among the practical early candidates, Layer 3 provides the smallest transmitted activation while still leaving a substantial portion of the network for execution on the Pi 3.

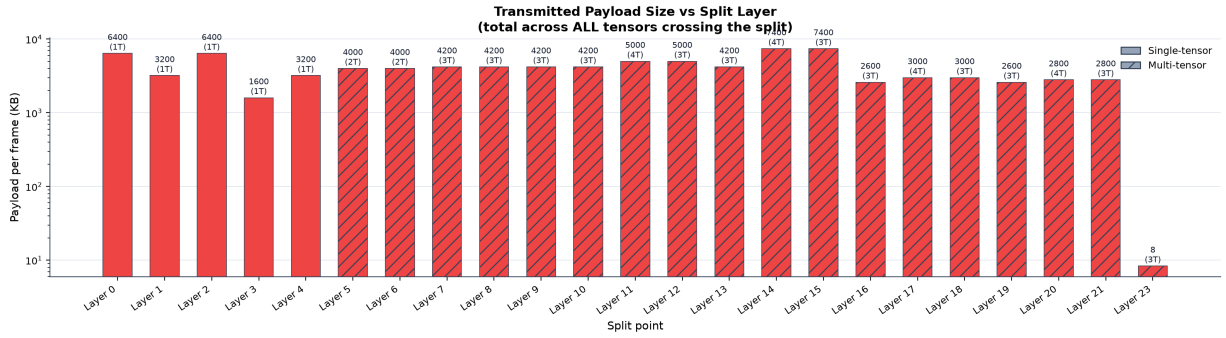


Figure 7 Transmitted payload size across the evaluated split layers. Single-tensor and multi-tensor positions are shown together to characterise the full communication cost landscape; non-valid positions are retained for analysis but are not treated as deployable split points.

The multi-tensor positions further illustrate why the split cannot be selected from layer depth alone. Once multiple tensors cross the boundary, the transmitted payload generally increases to several megabytes per frame, despite the deeper split. Therefore, moving the split deeper does not necessarily reduce communication; the tensor dependencies created by the network architecture can instead increase the amount of data that must be transmitted.

5.3.2 Latency Breakdown and Split Selection

The corresponding latency breakdown is shown in Figure 8. The results demonstrate that the cost of a split is determined by the combined head computation, network transfer, and tail computation rather than by edge computation alone. At Layer 3, the raw split has a total measured latency of approximately 4653 ms per frame, with both the communication and tail-computation stages contributing substantially to the end-to-end cost. Several multi-tensor positions incur much larger transfer costs, producing total latencies of several seconds and, in the most expensive case, approximately 30 s per frame.

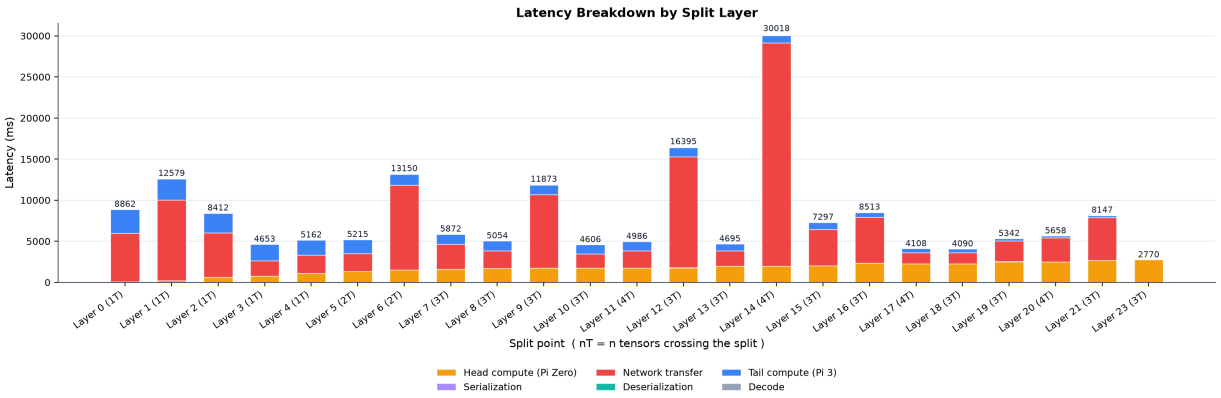


Figure 8 Latency breakdown across the evaluated split layers. Total latency combines head computation on the Pi Zero 2W, network transfer, and tail computation on the Pi 3, with additional communication overheads included where measured.

These measurements show that reducing the computation assigned to the Pi Zero does not by itself guarantee a faster inference pipeline. The wireless transfer stage can offset the computational savings introduced by splitting, particularly when large or multiple tensors cross the boundary. This observation motivates the use of Layer 3 as the selected split point and, more importantly, motivates the learned bottleneck introduced at that point: the objective is not simply to move computation to the Pi 3, but to reduce the amount of intermediate information that must cross the wireless link.

5.4 Pipelined Execution: Experiment and Results

The raw split system processes one frame at a time when $K = 1$. After the Pi Zero finishes processing a frame, the system waits for transmission and Pi 3 processing before the next frame can fully progress through the pipeline. This serial behaviour limits sustained throughput. Pipelining allows multiple frames to occupy different stages simultaneously. The parameter K denotes the maximum number of frames allowed to be in-flight in the distributed pipeline at the same time. For example, at $K = 2$, one frame can be processed by the Pi 3 while the Pi Zero begins processing the next frame. The model architecture and split point remain unchanged; only the execution scheduling changes.

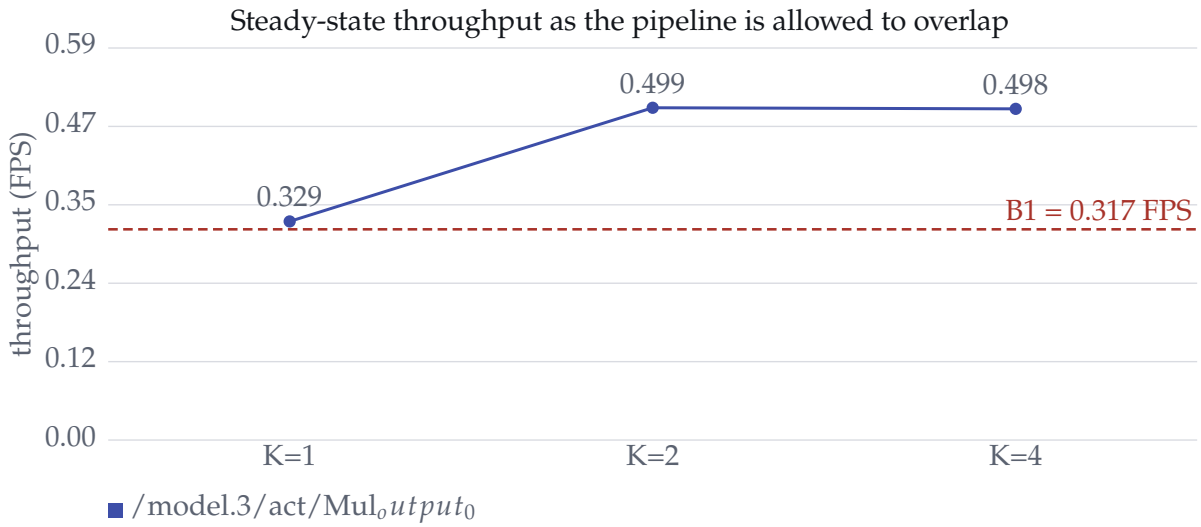


Figure 9 Steady-state throughput versus the maximum number of frames in flight (K). The dashed line indicates the sustained on-device B1 baseline.

The results show that pipelining is necessary to obtain a meaningful throughput advantage from distributed execution. At $K = 1$, the measured throughput is 0.329 FPS, only slightly above the sustained B1 baseline of 0.3166 FPS. Increasing the number of in-flight frames to $K = 2$ raises throughput to 0.499 FPS, corresponding to approximately a 58% improvement over the on-device baseline. Increasing K further to 4 produces 0.498 FPS, showing that additional concurrency provides essentially no throughput benefit under the measured conditions. The improvement therefore comes primarily from overlapping the stages of the pipeline rather than from increasing the computational capacity of either device.

5.4.1 Throughput–Latency Trade-off and Choice of K

Pipelining improves sustained throughput but does not reduce the end-to-end latency of an individual frame. As K increases, more frames can overlap, but frames may also wait in the pipeline before reaching the Pi 3. For this reason, the value of K must be selected as a system-level trade-off rather than simply maximising FPS. In the final deployment configuration, $K = 2$ provides the preferred compromise between increased sustained throughput and queueing delay. Increasing the number of frames in flight further provides limited additional throughput while substantially increasing end-to-end latency. Thus, pipelining changes the optimisation objective from minimising the sum of all stage latencies for one frame to maximising the completion rate of a stream of frames. This distinction is important for the camera-trap use case, where sustained frame processing is more relevant than minimum latency for a single isolated image.

5.5 Learned Bottleneck: Communication Reduction

The raw split evaluation showed that transmitting intermediate activations can become a significant communication cost. The learned bottleneck addresses this problem by compressing the intermediate representation before transmission and reconstructing a task-relevant representation before the remaining model is executed on the Pi 3.

5.5.1 Payload and Communication Cost

The learned bottleneck reduces the transmitted intermediate feature from approximately 1600 KB in FP32 form to approximately 25 KB in UINT8 form. This corresponds to approximately a

64× reduction in payload size. The associated transmission time decreases from approximately 522 ms to 73 ms per frame. The bottleneck therefore changes the role of the network in the system. Rather than transferring a large raw activation tensor, the Pi Zero transmits a compact task-relevant representation. This reduction is the main mechanism that makes the selected distributed inference configuration practical on the available wireless link.

5.5.2 Overall Bottleneck Performance

The learned bottleneck preserves most of the original detection performance while substantially reducing communication cost. Compared with the raw representation, the compressed UINT8 representation reduces mAP50–95 by 0.0197 overall and by only 0.011 for the target leopard class. The final result is therefore a system-level trade-off: a small loss in detection accuracy is accepted in exchange for a 64× reduction in communication payload. The bottleneck does not attempt to preserve the raw activation exactly; instead, it learns a compact representation that preserves the information required by the downstream detection task.

5.6 Limitations and Caveats

5.6.1 Experimental Scope

- The experiments used disk-streamed images rather than a continuous live camera feed.
- Full long-term field-deployment measurements were outside the scope of this study.
- The experiments were conducted on the available hardware and wireless network configuration, so absolute timings may change on different devices or links.

5.6.2 Thermal and Throughput Constraints

The sustained B1 measurements show that the Pi Zero 2W reaches approximately 82–83 °C during continuous full-model inference. The reported baseline throughput therefore reflects sustained thermally loaded operation rather than cold-start performance. For the split system, pipelining improves sustained throughput by overlapping stages, but it increases the end-to-end latency experienced by individual frames. The Pi 3 tail remains an important throughput constraint because the pipeline cannot sustain a completion rate substantially greater than its slowest stage.

5.6.3 Dataset Limitations

- The dataset has significant class imbalance, with the leopard class containing substantially more examples than several rare classes.
- Performance estimates for classes with few validation samples are less statistically stable.
- More balanced data collection and stratified evaluation would improve confidence in rare-class performance.

6 Conclusion

6.1 Summary of Contribution

This work demonstrates a practical split-inference architecture for running YOLO26L on resource-constrained edge hardware. The evaluation shows that simply dividing the model between the Pi Zero 2W and Pi 3 is not sufficient: intermediate feature maps can introduce a substantial communication cost that offsets the computational savings of model partitioning. The selected Layer 3 split provides a practical early boundary, while the learned bottleneck reduces the transmitted representation from approximately 1600 KB to 25 KB, corresponding to a 64× reduction in payload. This reduction is achieved with a decrease of only 0.0197 in overall mAP50–95 and 0.011 in AP50–95 for the target leopard class. Together, these results show that communication-aware split inference can make a large detector usable on hardware that cannot efficiently execute the complete model by itself.

6.2 Practical Deployment

The resulting architecture assigns the early computation and feature encoding to the Raspberry Pi Zero 2W at the field node, while the Raspberry Pi 3 performs feature reconstruction and the remaining detection stages. This division reduces the amount of model computation that must be sustained by the field device while avoiding dependence on cloud infrastructure or an Internet connection. The pipelined evaluation further shows that allowing multiple frames to remain in flight can substantially improve sustained throughput, with $K = 2$ providing the preferred measured operating point. The architecture therefore provides a feasible local edge-processing configuration for camera-trap deployments in which connectivity to remote cloud services cannot be assumed.

6.3 Technical Insights

Several technical observations emerge from the evaluation. First, split-point selection and communication cost must be considered jointly: a deeper split is not necessarily better when multiple feature tensors cross the boundary. Second, the experiments show that memory capacity is not the only constraint on the Pi Zero; the communication cost of intermediate representations can become the dominant system-level limitation. Third, the learned bottleneck does not need to reproduce the transmitted activation exactly in order to preserve useful detection performance. A compact representation can discard information that is not required by the downstream detector while retaining task-relevant structure. Finally, pipelining changes the optimisation objective from the latency of a single frame to the sustained completion rate of a stream, making throughput and queueing behaviour important deployment considerations.

6.4 Reproducibility

The implementation, benchmark scripts, ONNX model artefacts, and bottleneck training procedure are documented in the project repository. The experiments were performed on the deployed ONNX artefacts and the target Raspberry Pi hardware, allowing the reported communication, throughput, and accuracy measurements to be reproduced under comparable hardware and network conditions. The separation between the model, split inference pipeline, and bottleneck components also allows the same evaluation methodology to be extended to other YOLO26 model variants.

6.5 Future Directions

The next step is to validate the complete system under a continuous live-camera workload and over longer deployment periods. This would allow the effects of sustained thermal behaviour, wireless variability, queueing, and camera-triggered frame bursts to be measured directly. The bottleneck could also be improved by incorporating a detection-aware training objective rather than relying only on reconstruction loss, potentially reducing the remaining accuracy gap. Further work should investigate quantisation of the downstream model, more balanced and stratified evaluation data, and alternative YOLO26 model sizes to explore the accuracy–latency–communication trade-off. Finally, integrating automatic startup and recovery on the Raspberry Pi devices would move the prototype closer to a self-contained field deployment.

References

[1] Ao Wang et al. "YOLOv10: Real-Time End-to-End Object Detection". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2024. DOI: 10.48550/arXiv.2405.14458.

[2] Ultralytics. *YOLO26: Ultralytics Documentation*. https://docs.ultralytics.com/models/yolo26/. Accessed: 2026-08-18. 2025.

[3] Yiping Kang et al. "Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge". In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 615–629. DOI: 10.1145/3037697.3037698.

[4] En Li et al. "Edge AI: On-Demand Accelerating Deep Neural Network Inference via Edge Computing". In: *IEEE Transactions on Wireless Communications* 19.1 (2020), pp. 447–457. DOI: 10.1109/TWC.2019.2946140.

[5] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. "Split Computing and Early Exiting for Deep Learning Applications: Survey and Research Challenges". In: *ACM Computing Surveys* 55.5 (2022), pp. 1–30. DOI: 10.1145/3527155.

[6] Jiawei Shao and Jun Zhang. "Communication-Computation Trade-Off in Resource-Constrained Edge Inference". In: *IEEE Communications Magazine* 58.12 (2020), pp. 20–26. DOI: 10.1109/MCOM.001.2000373.

[7] Yoshitomo Matsubara and Marco Levorato. "Split Computing for Complex Object Detectors: Challenges and Preliminary Results". In: *Proceedings of the 4th International Workshop on Embedded and Mobile Deep Learning (EMDL '20)*. ACM, 2020, pp. 7–12. DOI: 10.1145/3410338.3412338.

[8] Arian Bakhtiarnia et al. "Dynamic Split Computing for Efficient Deep Edge Intelligence". In: *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2023. DOI: 10.48550/arXiv.2205.11269.

[9] Xin Yuan et al. *A QoE-Aware Split Inference Accelerating Algorithm for NOMA-based Edge Intelligence*. arXiv preprint. 2024. DOI: 10.48550/arXiv.2409.16537.

[10] Ji Chan Lee et al. "A Reconfigurable Neural Architecture for Edge–Cloud Collaborative Real-Time Object Detection". In: *IEEE Internet of Things Journal* 9.23 (2022), pp. 23390–23404. DOI: 10.1109/JIOT.2022.3205682.

[11] Duy V. Ma et al. "Enabling Lightweight Split Inference for Real-Time Detection in Embedded Traffic Cameras". In: *Proceedings of the 20th Asian Internet Engineering Conference (AINTEC '25)*. ACM, 2025. DOI: 10.1145/3763400.3763445.

[12] Benoit Jacob et al. "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286.

[13] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. arXiv preprint. 2013. DOI: 10.48550/arXiv.1308.3432.

[14] Hao Wu et al. *Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation*. arXiv preprint. 2020. DOI: 10.48550/arXiv.2004.09602.

[15] Amir Erfan Eshratifar, Amirhossein Esmaili, and Massoud Pedram. "BottleNet: A Deep Learning Architecture for Intelligent Mobile Cloud Computing Services". In: *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. 2019, pp. 1–6. DOI: 10.48550/arXiv.1902.01000.

[16] Jiawei Shao and Jun Zhang. "BottleNet++: An End-to-End Approach for Feature Compression in Device-Edge Co-Inference Systems". In: *IEEE International Conference on Communications Workshops (ICC Workshops)*. 2020, pp. 1–6. DOI: 10.48550/arXiv.1910.14315.

[17] Yoshitomo Matsubara et al. "BottleFit: Learning Compressed Representations in Deep Neural Networks for Effective and Efficient Split Computing". In: *IEEE 23rd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. 2022, pp. 337–346. DOI: 10.1109/WoWMoM54355.2022.00043.

[18] Yoshitomo Matsubara et al. "Head Network Distillation: Splitting Distilled Deep Neural Networks for Resource-Constrained Edge Computing Systems". In: *IEEE Access* 8 (2020), pp. 212177–212193. DOI: 10.1109/ACCESS.2020.3039714.

[19] Yoshitomo Matsubara et al. "SC2 Benchmark: Supervised Compression for Split Computing". In: *Transactions on Machine Learning Research (TMLR)* (2023). URL: https://openreview.net/forum?id=p28wv4G65d.

[20] Jiawei Shao, Yuyi Mao, and Jun Zhang. "Learning Task-Oriented Communication for Edge Inference: An Information Bottleneck Approach". In: *IEEE Journal on Selected Areas in Communications* 40.1 (2022), pp. 197–211. DOI: 10.48550/arXiv.2102.04170.

[21] Yoshitomo Matsubara and Marco Levorato. "Neural Compression and Filtering for Edge-Assisted Real-Time Object Detection in Challenged Networks". In: *25th International Conference on Pattern Recognition (ICPR)*. IEEE, 2021, pp. 2272–2279. DOI: 10.48550/arXiv.2007.15818.

[22] *LILA BC: Labeled Information Library of Alexandria — Biology and Conservation*. https://lila.science/datasets. Repository of open camera-trap and conservation datasets. URL: https://lila.science/datasets (visited on 08/17/2026).

[23] Namratha Jujya. *Leopard Dataset*. https://universe.roboflow.com/namratha-jujya/leopard-nbxvh. Version 2, CC BY 4.0. URL: https://universe.roboflow.com/namratha-jujya/leopard-nbxvh.

[24] Animal Detection. *Cheetah Dataset*. https://universe.roboflow.com/animal-detection-uausv/cheetah-7mzvi. Version 1, CC BY 4.0. URL: https://universe.roboflow.com/animal-detection-uausv/cheetah-7mzvi.

[25] Sakshi. *Hyena Dataset*. https://universe.roboflow.com/sakshi-0ll7y/hyena. Version 2, CC BY 4.0. URL: https://universe.roboflow.com/sakshi-0ll7y/hyena.

[26] Shlomi Peretz. *Nubian Ibex Finder Dataset*. https://universe.roboflow.com/shlomi-peretz-1evif/nubian-ibex-finder. Version 3, CC BY 4.0. URL: https://universe.roboflow.com/shlomi-peretz-1evif/nubian-ibex-finder.

[27] Project 5last. *Camel Dataset*. https://universe.roboflow.com/project-5last/camel-hyh5i. Version 1, CC BY 4.0. URL: https://universe.roboflow.com/project-5last/camel-hyh5i.

[28] Birds. *Dogs and Cats Dataset*. https://universe.roboflow.com/birds-gbqbp/dogs-and-cats-pcdpa. Version 1, CC BY 4.0. URL: https://universe.roboflow.com/birds-gbqbp/dogs-and-cats-pcdpa.

[29] Yara Hashesh. *People Dataset*. https://universe.roboflow.com/yara-hashesh-9u8ie/people-ojb8y. Version 3, CC BY 4.0. URL: https://universe.roboflow.com/yara-hashesh-9u8ie/people-ojb8y.

[30] Priyanto Hidayatullah and Refdinal Tubagus. *YOLO26: A Comprehensive Architecture Overview and Key Improvements*. 2026. arXiv: 2602.14582 [cs.CV]. URL: https://arxiv.org/abs/2602.14582.