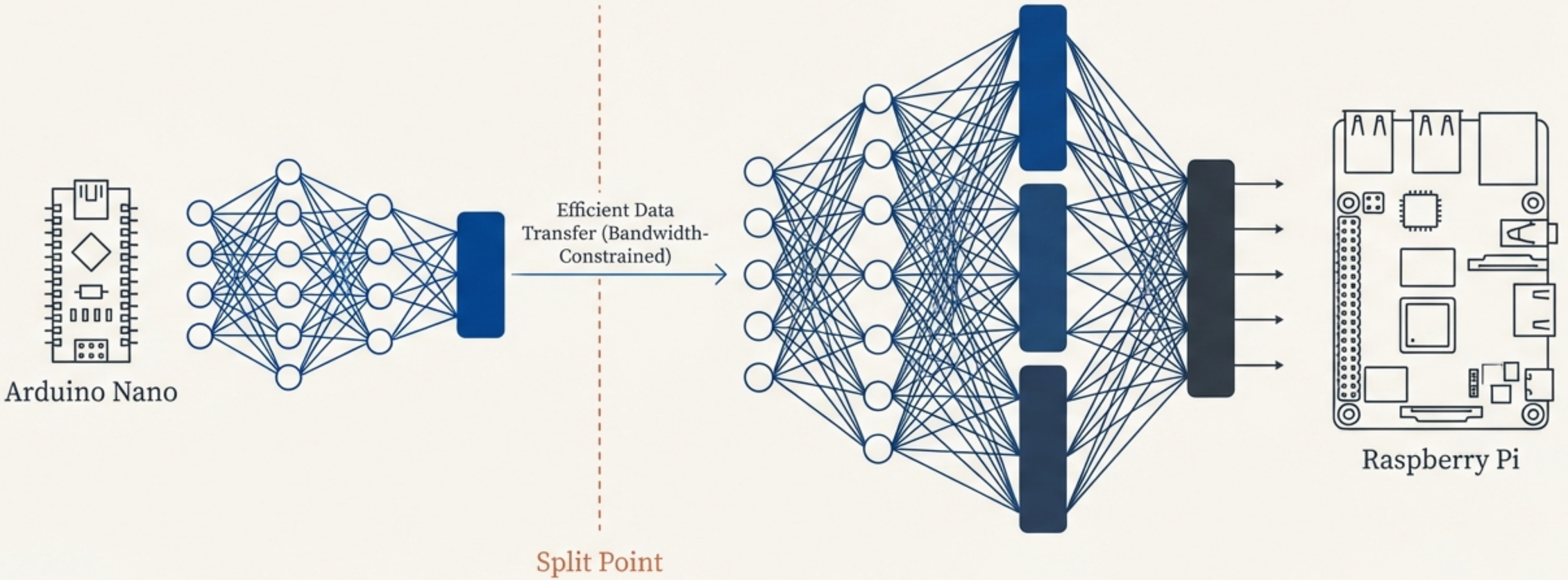
Hardware-Aware Hybrid Intelligence

A Case Study in Bandwidth-Constrained Split Learning Across an Arduino Nano and Raspberry Pi



The AI Edge Dilemma: Powerful Models vs. Tiny Devices

The Goal

Deploy sophisticated deep learning models on resource-constrained microcontrollers like the Arduino Nano 33 BLE Sense.

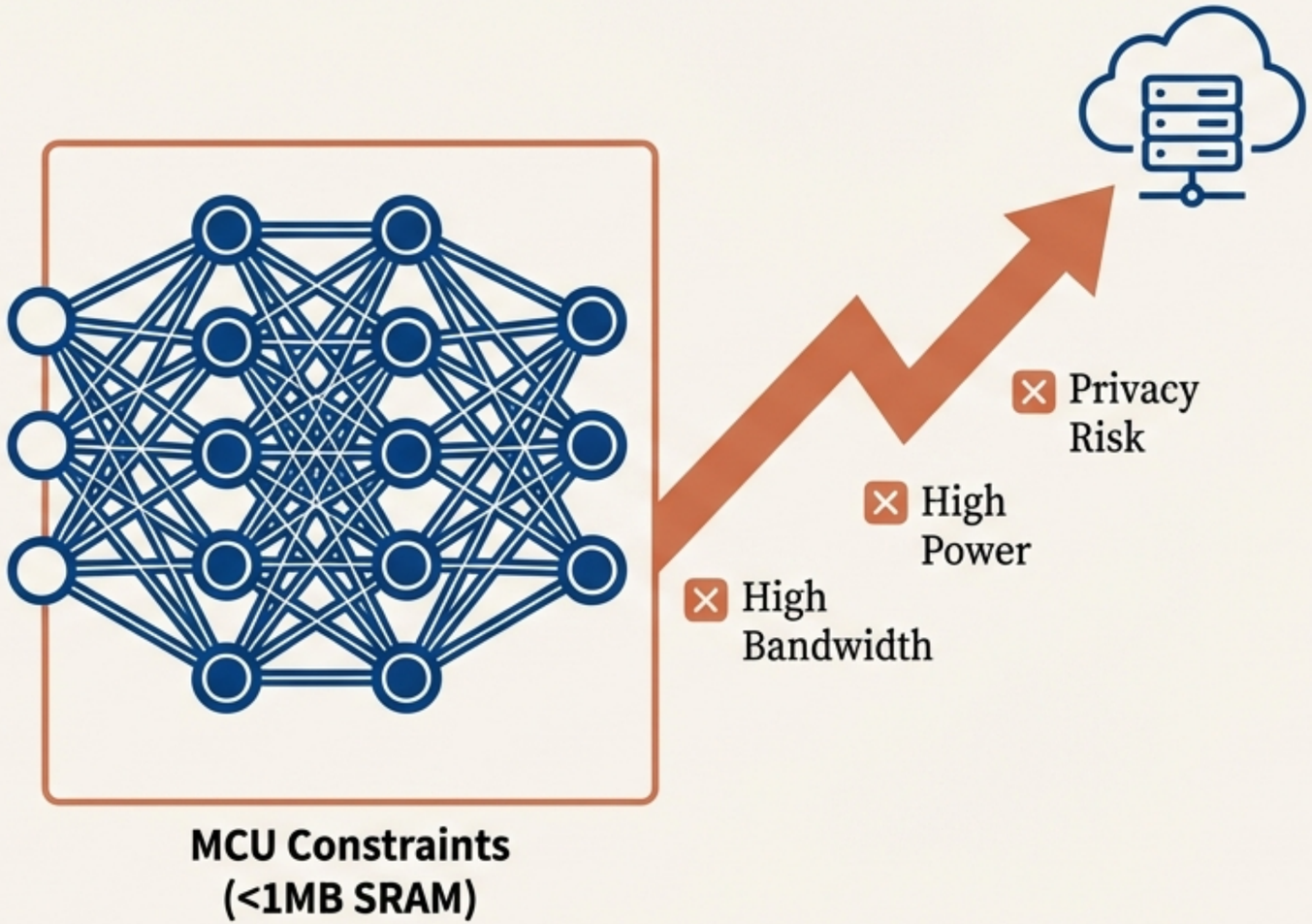
The Wall

These devices suffer from a "utility gap"—their strict memory (<1MB SRAM/Flash) and processing limits make running complex models locally infeasible.

The Flawed Alternative

Offloading raw sensor data (like images) to a remote server is not a solution. This approach introduces prohibitive costs:

- Excessive Bandwidth & Power Consumption
- Significant Data Privacy Risks

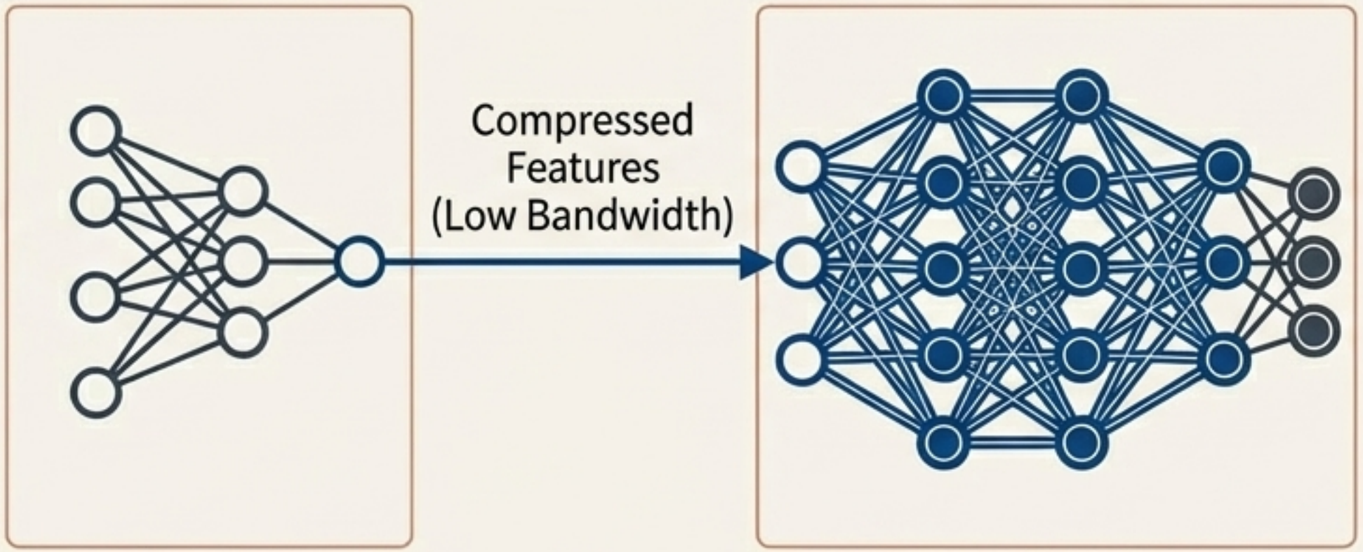


The Solution: A Hybrid Approach with Split Learning

Core Concept

Instead of an all-or-nothing choice, we partition the neural network across devices with different capabilities.

Arduino (ClientNet)
The Scout



Raspberry Pi (ServerNet)
The Command Center

The Critical Advantage

Only a small, compressed feature tensor ('smashed data') is transmitted. Raw data never leaves the edge device, preserving privacy and dramatically reducing bandwidth.

ClientNet

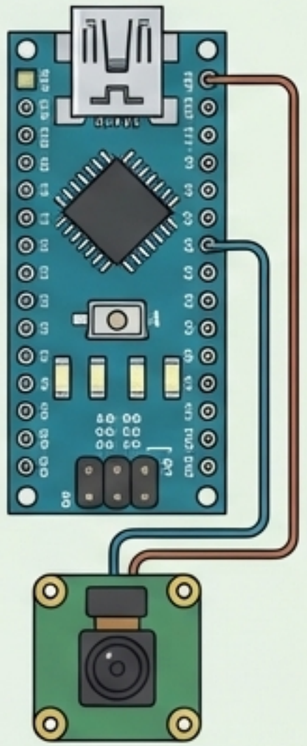
The first few lightweight layers run on the Arduino. It performs initial feature extraction on-device, identifying basic patterns.

ServerNet

The remaining, computationally heavy layers run on a more capable local Raspberry Pi, performing the final classification.

System Architecture: A Tale of Two Chips

The Scout
(Client Side - Arduino Nano)



Captures local images.

Runs the `ClientNet` model to produce intermediate activations.

Raw images NEVER leave the device.

128-byte int8 Feature Tensor

~~>3KB Raw Image~~

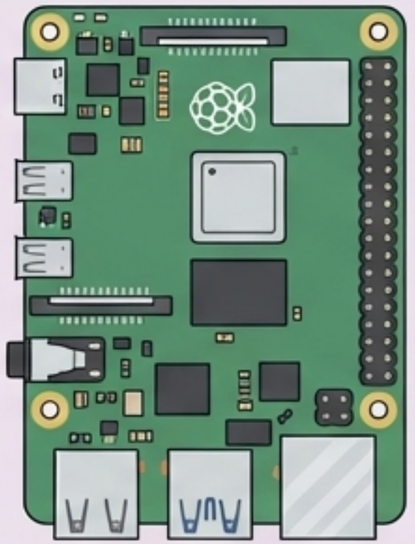
1-byte Decision (0/1)

The Command Center
(Server Side - Raspberry Pi)

Receives the feature tensor from the Arduino.

Executes the remaining `ServerNet` layers to get a final prediction.

Sends a simple binary decision (0/1) back to the Arduino.



No raw image is ever transmitted.

The End-to-End Data Pipeline

Arduino

1. Capture (Arduino)
Captures 160x120 frame, downsamples to 32x32x3.

2. Extract (Arduino)
Runs `ClientNet` (TFLite Micro) to produce a 128-byte feature tensor (shape 8x4x4).

3. Transmit (BLE)
Sends the 128 bytes in chunks over Bluetooth Low Energy.

128-byte Feature Tensor

1-byte Decision (0/1)

8. Actuate (Arduino)
Receives the response and prints the final result: "DOG DETECTED".

Raspberry Pi

4. Reassemble (Raspberry Pi)
Receives and reassembles chunked packets.

5. Dequantize (Raspberry Pi)
Uses `calibration_scale` from `meta.json` to convert features for inference.

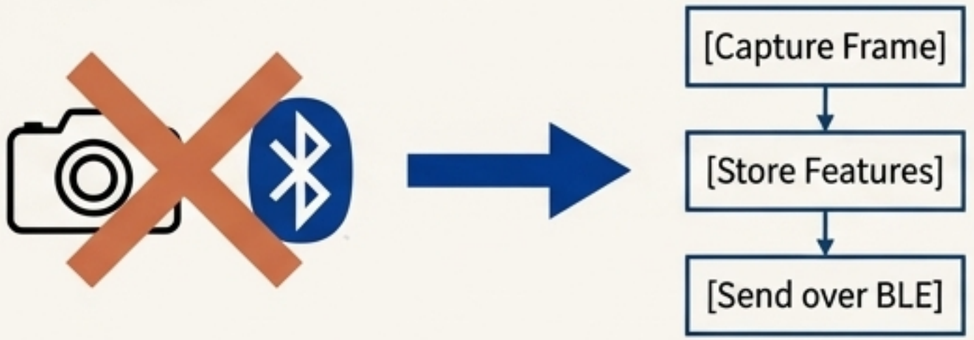
6. Infer (Raspberry Pi)
Runs `ServerNet` for binary classification (Dog vs. Not-Dog).

7. Reply (BLE)
Sends a 1-byte response (0 or 1) back to the Arduino.

The Client: Engineering for Extreme Constraints on Arduino

The Hardware Challenge

- **Memory/Compute**: Limited SRAM prevents running convolutions on data exceeding 20KB.
- **The Hardware Conflict**: The camera and Bluetooth radio cannot operate simultaneously. This discovery forced a sequential, buffered workflow.



Implementation Details (`ClientNet`)

- **Model:** A `client_model.h` TFLite Micro model header contains the client-side network.
- **Workflow:**
  1. Capture a frame and populate the input tensor.
  2. Invoke the TFLM interpreter, generating a 128-byte `int8_t` feature array.
  3. Store these features locally.
  4. *Only after the camera is inactive*, the device begins advertising and sending the stored features over BLE.

`Workingsketch_Picture_BLE_Reply (1).txt`

```
// Run client model and store intermediate features
bool capture_and_compute_features_when_idle() {
  if (BLE.connected()) return false;

  Serial.println("📸 Capturing + client infer (NOT
connected) ...");

  if (!fill_input_from_camera_frame()) return

  if (interpreter->Invoke() != kTfLiteOk) {
    Serial.println("❌ Invoke failed");
    return false;
  }
  // ...
  memcpy(feats_int8, output->data.int8, FEAT_LEN);
  feats_ready = true;
  Serial.println("✅ Features ready (stored)");
  return true;
}
```

This line is the critical gate. It explicitly prevents camera operation while a BLE connection is active, enforcing the sequential workflow.

The Server: Inference and Decision-Making on Raspberry Pi

Primary Role

To receive fragmented data, reconstruct the feature tensor, and perform the final, heavier part of the inference.

Implementation Details (`ServerNet`)

- Model: A custom PyTorch `ServerConvBinary` model (`server_conv_dogbin.pth`) specifically trained to operate on the 8x4x4 feature tensor from the client.
- Workflow (`recv_features_convbin_and_reply.py`):
  1. Receive & Reassemble: The `bleak` library listens for chunked BLE notifications and reassembles the 128-byte payload.
  2. Dequantize: Converts the `int8` data to `float32` using the crucial `calibration_scale` from `meta.json`.
  3. Infer: Feeds the tensor into the `ServerNet` model.
  4. Decide & Reply: Classifies based on the logit output (`logit >= 0` → dog=1) and writes a single byte response back to the Arduino via the `RESP_UUID`.

recv_features_convbin_reply_pi (1).txt

```
class ServerConvBinary(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(8, 16, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(16, 16, 3, padding=1),
            nn.ReLU(inplace=True),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(16, 1),
        )

    def forward(self, x):
        return self.net(x).squeeze(1)
```

The first convolutional layer is explicitly designed with `in_channels=8`, proving this model is purpose-built to accept the 8x4x4 feature tensor from the Arduino, not a raw image.

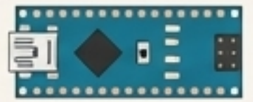
The Training Strategy: Engineering a Cohesive Split Model

The entire split network was trained *end-to-end* on a PC using PyTorch and the CIFAR-10 dataset. This ensures the client's feature represrtion is optimally designed for the server's classification task.

PC / Training Environment

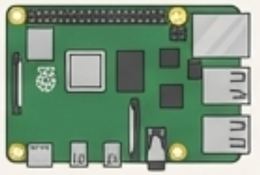
`client_model.h`

A TFLite Micro model containing the quantized `ClientNet` weights for on-device inference.



`server_conv_dogbin.pth`

The specialized `ServerNet` binary classifier, trained on the cached features from the `ClientNet`.



`meta.json`

Contains the essential `calibration_scale` needed to dequantize the features on the server before inference.

Performance Validated: Key System Metrics

Bandwidth Reduction

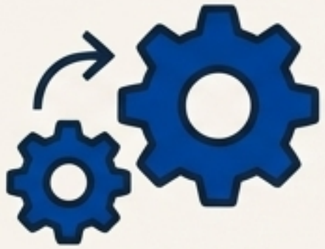
>99%

A **128-byte** quantized feature tensor was sent instead of the **38,400-byte** raw camera frame, resulting in a dramatic reduction in data transmission per inference.

Data Privacy

100%

Raw visual data never leaves the Arduino. Only an abstract, non-human-readable feature representation is transmitted, ensuring on-device privacy.



Compute Offloading

Successful

The most computationally expensive CNN layers were successfully offloaded from the memory-constrained Arduino to the more capable Raspberry Pi.

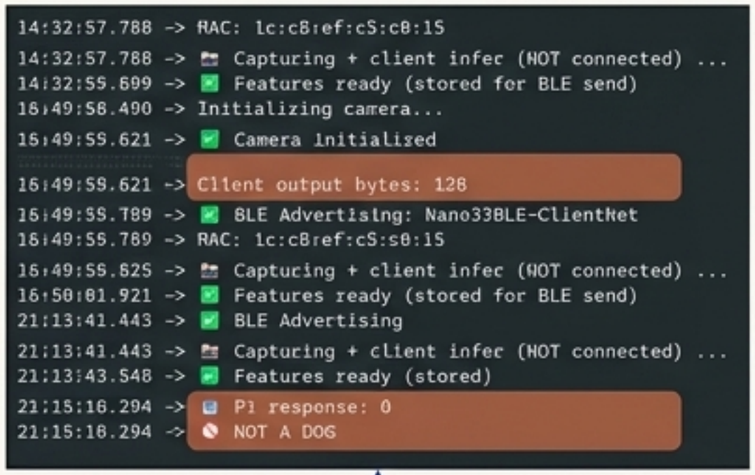
System Accuracy

83.40%

The final Pi-side binary classifier achieved a test accuracy of 83.40% on the dog vs. not-dog classification task.

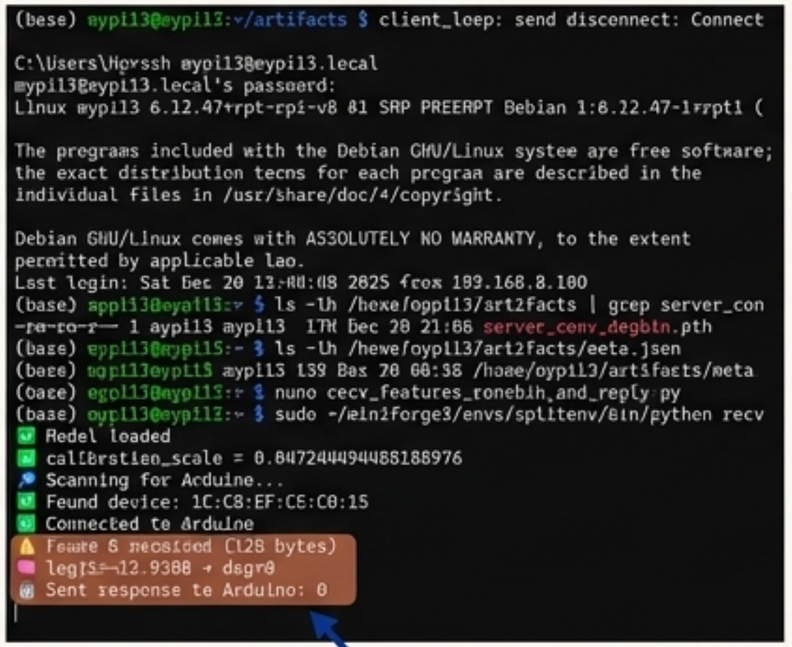
Evidence of a Successful Round Trip

1. Arduino: Capture & Actuate



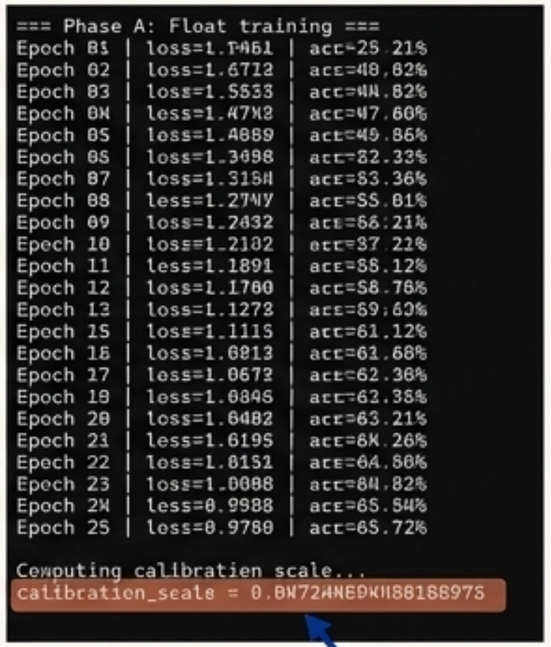
Arduino captures the image, computes 128 bytes of features, and prints the final result received from the Pi.

2. Raspberry Pi: Receive & Infer



Pi connects, receives the 128-byte frame, runs inference to get a logit value, decides the class, and sends the 1-byte response.

3. PC: Training & Calibration

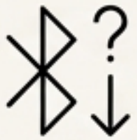


PC-side training generates not only the models but also the critical `calibration_scale` value, which is essential for the Pi to correctly interpret the Arduino's data.

Navigating Real-World Implementation Challenges

Challenge 1: Unstable Bluetooth Low Energy

Problem



Frequent, unexplained disconnections during continuous data transmission.

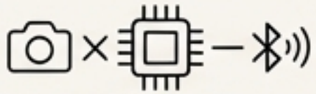
Solution

Required extensive debugging of UUIDs, tuning transmission delays (`CHUNK_DELAY_MS`), and implementing robust packet reassembly logic on the Pi.

Challenge 2: Camera vs. BLE Hardware Conflict

Problem

The Arduino Nano 33 BLE Sense hardware cannot operate the camera and radio simultaneously.



Solution

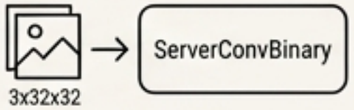
Architected a sequential “capture-then-send” workflow, buffering features locally before initiating any BLE communication.

Challenge 3: Model Input Incompatibility

Problem

Early Pi-side models failed because they expected a raw image (e.g., 3x32x32) instead of the 8x4x4 feature tensor from the Arduino.

Solution

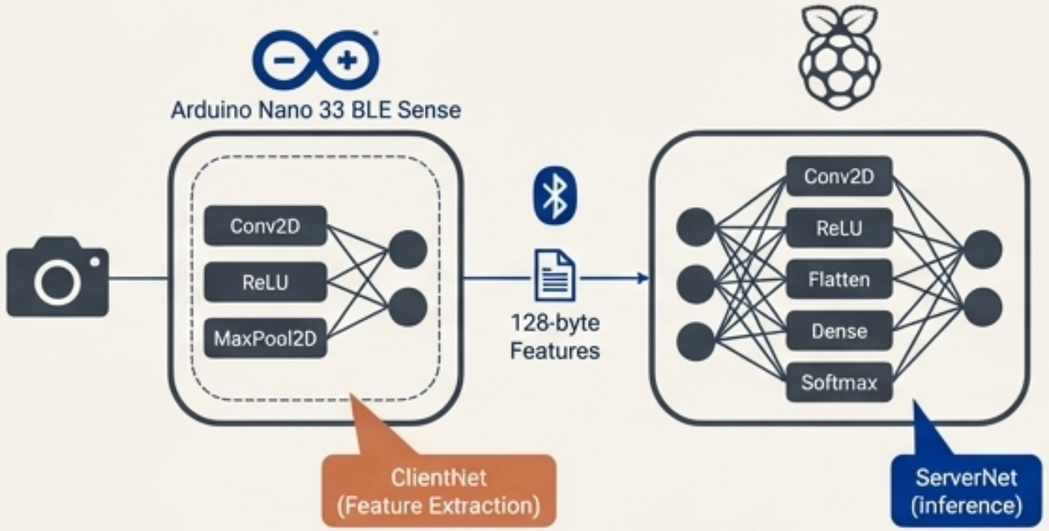


Trained a new, custom “conv head” (`ServerConvBinary`) for the Pi specifically designed to match the client’s output contract.

A Successful Scaffold for Hybrid Edge Intelligence

Key Accomplishment

This project successfully demonstrates a practical, low-bandwidth, and privacy-preserving solution for edge inference by intelligently splitting a neural network across constrained, heterogeneous hardware.



Future Directions & Improvements

- **Improve Client Accuracy:** Port the fully trained `ClientNet` weights (not just an approximation) to the Arduino's C++ environment. This is a complex but valuable task to maximize system accuracy.
- **Latency Optimization:** Profile and reduce the end-to-end time delay introduced by BLE transmission and Pi-side inference to improve performance for real-time applications.
- **Expand Functionality:** Move beyond a binary classifier to the full 10-class CIFAR-10 model, requiring modifications to the `ServerNet` and the Arduino's response handling.