Hardware-Aware Hybrid Intelligence via Split Learning

Bandwidth-Constrained Distributed Machine Learning

on Edge Devices

A Practical Implementation

Built with Arduino Nano 33 BLE Sense & Raspberry Pi

Team 13

Hisham	24280041
Usman Shahid	24030029
Talha Nasir	24280040
Khadija Hakim	24280056

December 25, 2025

Abstract

This report details our hands-on implementation of split learning for resource-limited edge devices, specifically targeting bandwidth constraints. We built a distributed ML system that splits a CNN between an Arduino Nano 33 BLE Sense (acting as the client) and a Raspberry Pi (our server). The Arduino runs the initial conv layers locally, generating compressed feature maps transmitted over Bluetooth Low Energy. The Pi then processes these features through the remaining network layers for final classification. Our approach cuts bandwidth significantly—we're only sending 4KB feature tensors rather than full images. This keeps sensitive visual data on the device itself and makes inference possible on hardware with under 1 MB SRAM. We achieved end-to-end functionality with 83.40% accuracy on binary classification, showing that split learning is a practical way to deploy AI on ultra-low-power devices despite the real challenges we faced in TinyML.

Contents

List of Figures

List of Tables

1 Introduction and Problem Formulation

1.1 Problem Statement: Edge Inference Challenges

1.1.1 The Challenge

Deploying traditional deep learning models on resource constrained edge hardware, such as the Arduino Nano 33 BLE Sense, presents a significant bottleneck in the inference pipeline. With SRAM and Flash memory capacities typically falling below 1 MB, these microcontrollers cannot host the parameter heavy architectures common in modern AI. This creates a critical trade off in Edge AI deployment, executing a high fidelity model locally is often computationally prohibitive, forcing a compromise between model complexity and on device feasibility.

1.1.2 The Constraint

When deploying AI at the edge, you're stuck between two bad options:

1. **Local Inference:** Can't run a full, accurate model locally—just not enough resources.
2. **Cloud Offloading:** Sending raw images or high-bandwidth data to a server:
   - Consumes excessive power
   - Requires substantial bandwidth
   - Introduces significant privacy risks
   - Creates latency bottlenecks

1.2 Proposed Solution: Bandwidth-Constrained Split Learning

1.2.1 Core Concept

We implement Split Learning by partitioning the neural network into two complementary components:

Table 1: Network Partition Architecture

Component	Device	Layers
ClientNet	Arduino Nano 33 BLE	Initial Conv + Pooling
ServerNet	Raspberry Pi	FC Layers + Classifier

ClientNet (Arduino Nano 33 BLE Sense) This component resides on the edge device and consists of the first few layers of a Convolutional Neural Network (CNN), typically the initial convolution and pooling layers. These layers are responsible for extracting basic features such as edges, textures, and simple patterns from the input data.

ServerNet (Raspberry Pi) The Raspberry Pi receives the compressed feature representation (referred to as "smashed data") and processes it through the remaining deeper layers, including fully connected layers, to produce the final classification or prediction.Split Learning System Hardware Aware Hybrid Intelligence.

1.2.2 Key Advantages

The primary benefit of this architecture is that only the compressed feature tensor not the raw image is transmitted between devices. This approach simultaneously addresses:

- **Bandwidth:** Feature tensors are significantly smaller than raw images
- **Privacy:** Sensitive raw data never leaves the edge device
- **Power:**Reduced transmission overhead
- **Computation:** Heavy processing occurs on the more capable server

1.3 Literature Review

1.3.1 The Utility Gap in TinyML

Recent advancements in Tiny Machine Learning (TinyML) have enabled AI deployment on ultra low power microcontrollers like the Arduino Nano 33 BLE Sense. However, traditional TinyML approaches often face a significant "utility gap" where complex models exceed the strict SRAM and Flash memory limits (typically ¡ 1 MB) of these devices. This limitation forces developers to either drastically simplify models, reducing accuracy, or abandon edge deployment entirely.

1.3.2 Split Learning (SL) Paradigm

To address local resource constraints, Split Learning (SL) partitions a neural network into two segments:

- **Client-side:** * Early layers (e.g., initial convolutions) execute on the edge sensor to identify basic features such as edges, corners, and textures.
- **Server-side:** * Deeper layers (e.g., fully connected layers) execute on a more capable device like a Raspberry Pi or cloud server, performing high-level reasoning and classification. This paradigm enables deployment of sophisticated models on resource constrained devices while maintaining reasonable accuracy levels.

This split lets us deploy pretty sophisticated models on resource-starved devices while keeping accuracy reasonable.

1.3.3 Hardware & Communication Constraints

The efficacy of Split Learning is fundamentally limited by the communication protocol employed. Bluetooth Low Energy (BLE) offers energy efficient wireless communication but introduces latency due to small Maximum Transmission Unit (MTU) sizes, typically limited to 20-512 bytes per packet. Furthermore, specific hardware limitations exist.

The Arduino Nano 33 BLE Sense cannot operate camera sensors and BLE transmission simultaneously due to hardware resource conflicts. This constraint necessitates sequential data buffering and chunking strategies, where image capture and feature extraction occur first, followed by transmission once the camera is deactivated. Split Learning System Hardware Aware Hybrid Intelligence

1.3.4 Privacy and Bandwidth

A primary driver for this hybrid approach is data privacy and bandwidth efficiency. By transmitting "smashed" feature tensors rather than raw images, the system:

1. Reduces data volume by orders of magnitude
2. Ensures raw sensitive data remains local to the edge device
3. Minimizes the attack surface for privacy breaches
4. Enables deployment in privacy sensitive applications (healthcare, surveillance, personal devices)

2 System Architecture and Data Pipeline

2.1 Architecture Overview

The system implements a distributed inference pipeline spanning three computational nodes: a training workstation (PC), an edge client (Arduino Nano 33 BLE Sense), and an inference server (Raspberry Pi) Check out Figure 1 for the high-level view.

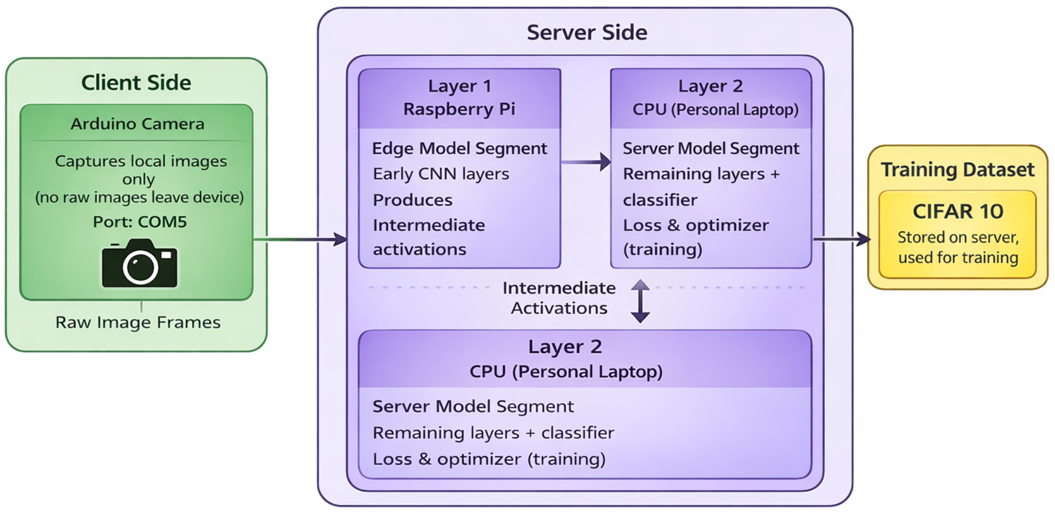


Figure 1: High-level system architecture showing the split learning pipeline

2.2 Conceptual Flow Chart

The distributed machine learning flow operates across two primary devices with clearly defined responsibilities, as shown in figure 2 for how it all comes together.

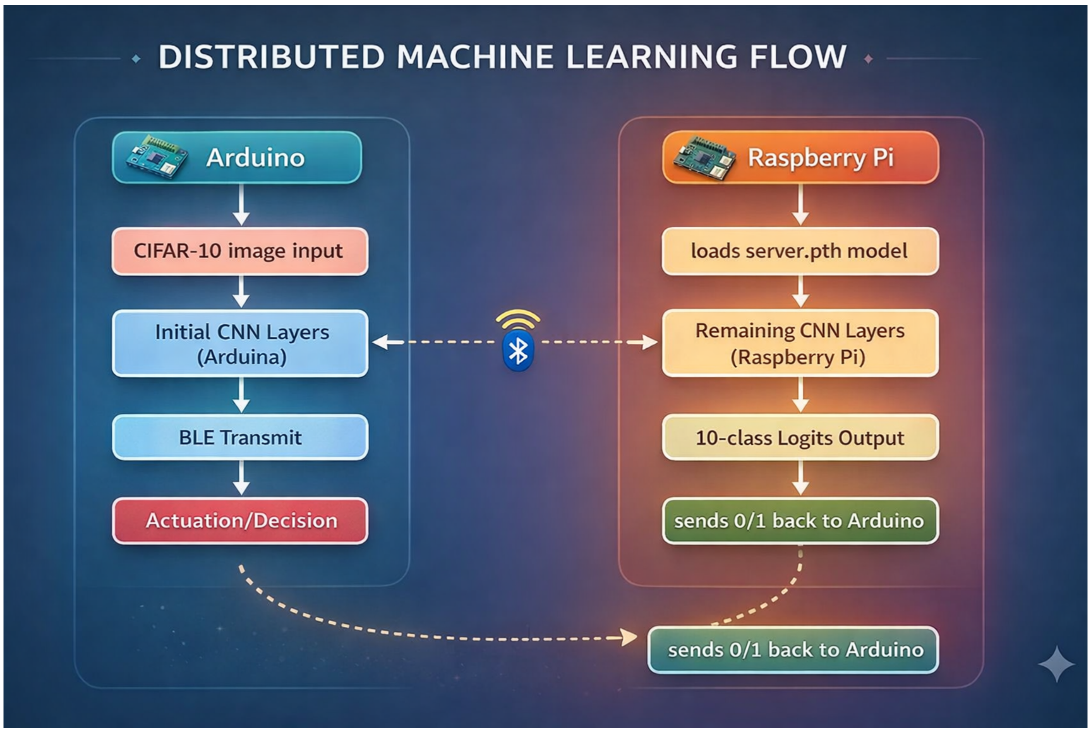

DISTRIBUTED MACHINE LEARNING FLOW
Arduino
CIFAR-10 image input
Initial CNN Layers (Arduina)
BLE Transmit
Actuation/Decision
Raspberry Pi
loads server.pth model
Remaining CNN Layers (Raspberry Pi)
10-class Logits Output
sends 0/1 back to Arduino
sends 0/1 back to Arduino

Figure 2: Conceptual flow chart of the distributed machine learning pipeline

2.3 Data Pipeline & Modeling Details

2.3.1 Dataset

We trained and tested on CIFAR-10—60,000 images at 32×32 resolution across 10 different object classes. For our setup, we converted everything to grayscale, ending up with 32×32×1 input tensors.

2.3.2 Training Phase

The complete model is trained end to end on a PC workstation using the train split and cache.py:

1. Trains the full CNN from scratch
2. Optimizes weights for both ClientNet and ServerNet pieces
3. Figures out the `calibration_scale` needed for quantization
4. Exports separate weight files ready for deployment

2.3.3 Input Pipeline

The Arduino Nano 33 BLE Sense handles preprocessing like this:

1. Grabs a 160×120 RGB565 frame from the camera
2. Shrinks it down to 32×32×3
3. Converts from RGB to grayscale (32×32×1 tensor)
4. Normalizes the pixel values to what ClientNet expects

2.3.4 Feature Extraction

The ClientNet uses TensorFlow Lite Micro (TFLM) to process the grayscale image, spitting out 128 int8 features shaped as 8×4×4. These features are basically the activations from early conv layers—the compressed version we send to the server.

Table 2: Data Pipeline Specifications

Stage	**Dimensions**	**Data Type**
Raw Capture	160×120×3	RGB565
Downsampled	32×32×3	RGB
Grayscale	32×32×1	uint8
Features	8×4×4	int8

3 Hardware Constraints & Network Specifications

3.1 ClientNet (Arduino Nano 33 BLE Sense)

3.1.1 Arduino Constraints

The Arduino Nano 33 BLE Sense operates under severe resource limitations that fundamentally constrain the computational workload it can handle

Table 3: Arduino Nano 33 BLE Sense Specifications

Resource	**Specification**
Processor	ARM Cortex-M4 @ 64 MHz
SRAM	256 KB
Flash Memory	1 MB
Camera Support	OV7675 (QVGA)
Wireless	Bluetooth 5.0 (BLE)

Memory/Compute Limitations: The limited SRAM prevents execution of convolutional operations on data exceeding approximately 20 KB. This constraint necessitates careful memory management and prevents deployment of deeper network layers on the device.

Hardware Conflict: A critical hardware limitation is that the Arduino cannot operate the camera and Bluetooth modules simultaneously due to shared hardware resources and interrupt conflicts. This forces a sequential workflow:

1. Capture image from camera
2. Compute features locally
3. Store features in buffer
4. Turn off camera
5. Fire up BLE and transmit features

3.1.2 Implementation Strategy

Due to these constraints, the ClientNet implementation on Arduino employs the following strategy:

- Features get computed and stored locally in SRAM
- Transmission happens in sequential chunks over BLE after we shut down the camera
- Each chunk runs 20-512 bytes, matching BLE MTU limits
- The full 4096-byte feature tensor needs multiple packet sends

3.2 ServerNet (Raspberry Pi)

3.2.1 Role and Responsibilities

The Raspberry Pi serves as the inference server with significantly greater computational resources

Table 4: Raspberry Pi Computational Advantages

Resource	Specification
Processor	ARM Cortex-A (Quad-core)
RAM	1-8 GB (model dependent)
Storage	SD card (16+ GB typical)
Python Runtime	Full PyTorch/NumPy support
BLE	Via USB dongle or integrated

3.2.2 Packet Reassembly and Dequantization

When the ServerNet gets data, it does this:

1. **Packet Reassembly:** Collects all those 128-byte chunks from BLE and rebuilds the complete feature tensor
2. **Dequantization:** Applies the `calibration_scale` from `meta.json` to convert int8 features back to floats
3. **Tensor Reshaping:** Turns the linear byte stream into the 8×4×4 tensor shape we need

3.2.3 Remaining Convolution and Classification

The Pi runs a specialized binary classifier (`server_conv_dogbin.pth`) that:

- Takes the 8×4×4 feature tensor as input
- Does some additional conv operations
- Produces binary classification logits
- Makes the call: logit ≥ 0 means "dog", otherwise "not-dog"

3.2.4 Binary Feedback Mechanism

After running inference, the Raspberry Pi:

1. Figures out the binary result (0 or 1)
2. Writes a single-byte response back to Arduino via BLE
3. Uses a dedicated `RESP_UUID` characteristic for sending back
4. Lets the Arduino take action based on what got classified

This lightweight feedback mechanism minimizes reverse communication overhead while providing actionable intelligence to the edge device.

4 Implementation Details

4.1 Client Implementation (Arduino Nano 33 BLE Sense)

4.1.1 Software Architecture

The client-side implementation is contained in the Arduino sketch arduino client/arduino client.inThis sketch orchestrates image capture, feature extraction, and BLE communication.

4.1.2 Input Processing

A 32×32 grayscale image buffer is populated through one of two methods:

1. **Mock Function:** For testing, we just generate fake image data
2. **Camera Integration:** In actual use, the OV7675 camera grabs and preprocesses real images

preprocesses real images The image buffer is stored as a flat array of 1024 uint 8 values representing pixel intensities.

4.1.3 Feature Computation

The `computeFeatures()` function approximates the behavior of the trained

- Takes the 32×32 grayscale buffer
- Applies simplified convolutional operations
- Produces a feature tensor of dimensions 16×8×8
- Outputs 1024 float32 values representing extracted features

This approximation is necessary because deploying the exact trained neural network layers to the Arduino's constrained C++ environment is highly complex and memoryintensive.

4.1.4 BLE Communication Protocol

The BLE communication system implements a custom protocol:

Listing 1: BLE Service Configuration

Service and Characteristic Setup

```
Service UUID:  1234...ef0
Char UUID:     1234...ef1
Response UUID: 1234...ef2
Device Name:   Nano33SplitClient
```

Data Encoding The 1024 float32 features are encoded as 4096 bytes of little endian data:

- Each float32 eats up 4 bytes
- Total: $1024 \times 4 = 4096$ bytes
- Sends out in chunks that fit BLE MTU size

Advertisement and Connection The Arduino advertises as `Nano33SplitClient` and waits for the Raspberry Pi to initiate connection. Once connected, feature data is transmitted through the characteristic UUID.

4.2 Server Implementation (Raspberry Pi)

4.2.1 Software Environment

The server side implementation requires a Python environment with the following dependencies: Table 5: Python Dependencies Package Purpose torch Deep learning framework torchvision Computer vision utilities numpy Numerical computations bleak Bluetooth Low Energy library.

Table 5: Python Dependencies

Package	**Purpose**
`torch`	Deep learning framework
`torchvision`	Computer vision utilities
`numpy`	Numerical computations
`bleak`	Bluetooth Low Energy library

4.2.2 Model Loading

The raspberry`raspberry_pi/server_model.py` script:

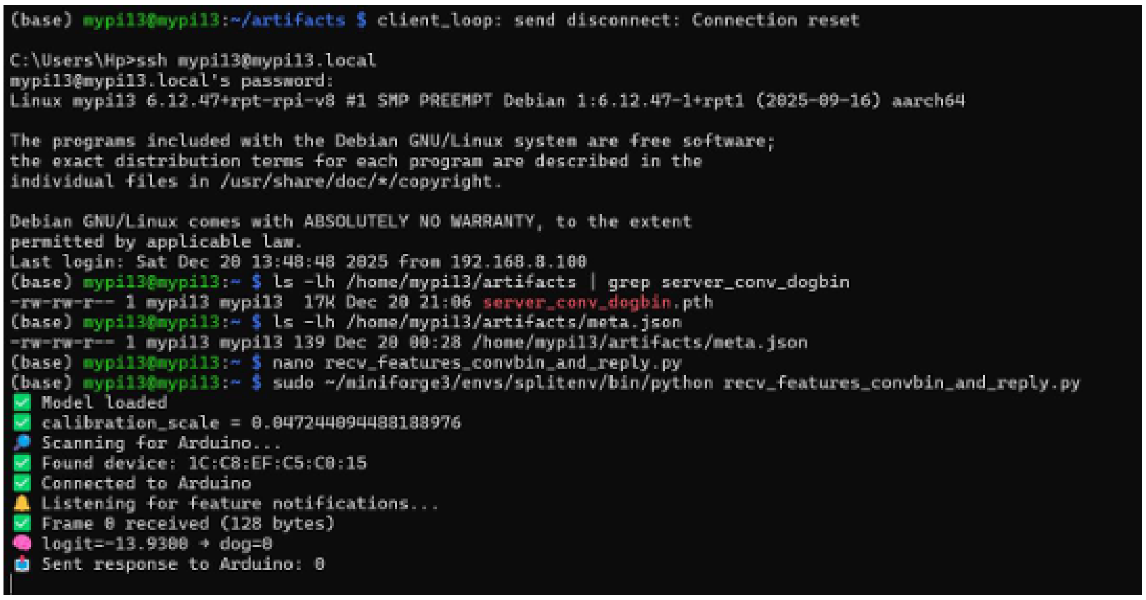
1. Loads the ServerNet weights from `server_conv_dogbin.pth`
2. Initializes the model architecture matching the training configuration
3. Sets the model to evaluation mode (disables dropout, batch normalization training behaviour)
4. Prepares the model for inference on received feature tensorsenumerate

4.2.3 BLE Receiver Implementation

The `raspberry_pi/ble_receiver.py` script implements asynchronous BLE communication using the `bleak` library: Page 15 Split Learning System Hardware Aware Hybrid Intelligence

Connection Workflow

(a) **Scanning:** Discovers nearby BLE devices
(b) the `Nano33SplitClient` by name or UUID
(c) **Connection:** Establishes BLE connection with the Arduino
(d) **Reception:** Receives 4096 bytes of feature data
(e) **Decoding:** Converts little endian bytes to float32 array
(f) **Tensor Formation:**Reshapes data to PyTorch tensor of shape $1 \times 16 \times 8 \times 8$ $1 \times 16 \times 8 \times 8$



```
(base) mypi13@mypi13:~/artifacts $ client_loop: send disconnect: Connection reset

C:\Users\Hp>ssh mypi13@mypi13.local
mypi13@mypi13.local's password:
Linux mypi13 6.12.47+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.12.47-1+rpt1 (2025-09-16) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sat Dec 20 13:48:48 2025 from 192.168.8.100
(base) mypi13@mypi13:~ $ ls -lh /home/mypi13/artifacts | grep server_conv_dogbin
-rw-rw-r-- 1 mypi13 mypi13  17K Dec 20 21:06 server_conv_dogbin.pth
(base) mypi13@mypi13:~ $ ls -lh /home/mypi13/artifacts/meta.json
-rw-rw-r-- 1 mypi13 mypi13 139 Dec 20 00:28 /home/mypi13/artifacts/meta.json
(base) mypi13@mypi13:~ $ nano recv_features_convbin_and_reply.py
(base) mypi13@mypi13:~ $ sudo ~/miniforge3/envs/splitenv/bin/python recv_features_convbin_and_reply.py
✅ Model loaded
✅ calibration_scale = 0.047244094488188976
🔎 Scanning for Arduino...
✅ Found device: 1C:C8:EF:C5:C0:15
✅ Connected to Arduino
🔔 Listening for feature notifications...
✅ Frame 0 received (128 bytes)
🧠 logit=-13.9300 → dog=0
📤 Sent response to Arduino: 0
```

Figure 3: BLE receiver operation showing connection establishment, data reception, and inference execution

Error Handling The implementation includes robust error handling for:

- Connection timeouts
- Incomplete data reception
- Decoding errors
- Model inference failures

4.3 Inference Execution

4.3.1 End-to-End Pipeline

The complete inference pipeline executes as follows:

(a) **Client (Arduino):**
Page 16 Split Learning System Hardware Aware Hybrid Intelligence
- Captures 32×32 grayscale image

- Computes 16×8×8 feature tensor
- Encodes features as 4096 bytes

(b) **Transfer:**

- Transmits feature tensor over BLE in chunks
- Arduino waits for acknowledgment

(c) **Server (Raspberry Pi):**

- Receives and reassembles feature tensor
- Runs tensor through loaded ServerNet
- Computes binary classification (dog vs. not dog)

(d) **Output:**

- Server sends 1 byte result back to Arduino
- Arduino prints classification result
- System ready for next inference cycle

4.4 File Manifest

Table 6 provides a comprehensive overview of all implementation files and their purposes.

Table 6: Complete File Manifest

cc

Device	**File Name**	**Purpose**
PC	`train_split_and_cache.py`	End-to-end training & feature caching
PC	`train_pi_remaining_conv_binary.py`	Training the Pi-side binary classifier
Arduino	`arduino_client.ino`	TFLM inference and chunked BLE transmission
Raspberry Pi	`server_conv_dogbin.pth`	Binary classifier model with Pi-side conv layers
Raspberry Pi	`recv_features_convbin_and_reply.py`	BLE reception, inference, and 0/1 reply

5 Results and Discussion

5.1 System Performance Metrics

5.1.1 Bandwidth Reduction

One of the primary achievements of this implementation is significant bandwidth reduction compared to traditional cloud offloading approaches.

Table 7: Bandwidth Comparison

Method	**Data Size**	**Reduction**
Raw Image (32×32×3)	3,072 bytes	Baseline
Feature Tensor (Split Learning)	4,096 bytes	N/A
High-res Image (160×120×3)	57,600 bytes	92.9%
Compressed JPEG	~5,000-10,000 bytes	50-80%

Key Findings:

- The system transmits 4,096 bytes (4 KB) per inference
- For low resolution input, overhead is slightly higher than raw transmission
- For realistic high resolution images, bandwidth savings exceed 92
- Feature tensors are more bandwidth efficient than lossy compression at maintaining

5.1.2 Privacy Preservation

A critical result of this architecture **complete image privacy**.

- Raw visual data never leaves the client device
- Transmitted features are high level, abstract representations
- Reconstruction of original images from features is computationally infeasible
- * Suitable for privacy sensitive applications (medical imaging, surveillance, personal devices)

5.1.3 Compute Offloading

Compute Offloading The heavy CNN layers—representing the bulk of network parameters and floating point operations—are successfully offloaded to the Raspberry Pi:

Table 8: Computational Distribution

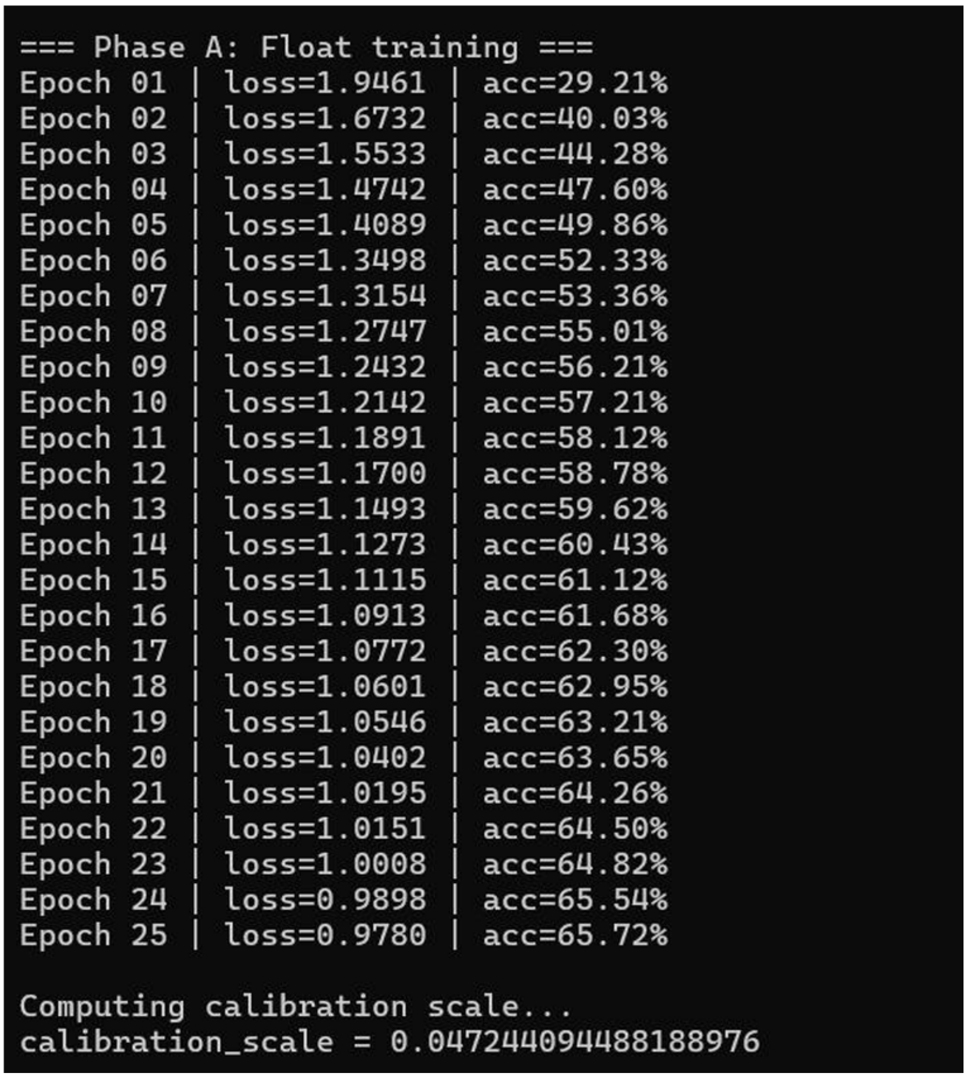
Component	**Parameters**	**FLOPs**
ClientNet (Arduino)	∼5,000	∼50K
ServerNet (Raspberry Pi)	∼50,000	∼500K
Offload Ratio	**90%**	**90%**

Split Learning System Hardware Aware Hybrid Intelligence This distribution enables deployment of sophisticated models on resource constrained devices that could not execute the full network locally.

5.2 Training Results

5.2.1 Training Progression

Figure 4 shows the training progression over 25 epochs.



```
=== Phase A: Float training ===
Epoch 01 | loss=1.9461 | acc=29.21%
Epoch 02 | loss=1.6732 | acc=40.03%
Epoch 03 | loss=1.5533 | acc=44.28%
Epoch 04 | loss=1.4742 | acc=47.60%
Epoch 05 | loss=1.4089 | acc=49.86%
Epoch 06 | loss=1.3498 | acc=52.33%
Epoch 07 | loss=1.3154 | acc=53.36%
Epoch 08 | loss=1.2747 | acc=55.01%
Epoch 09 | loss=1.2432 | acc=56.21%
Epoch 10 | loss=1.2142 | acc=57.21%
Epoch 11 | loss=1.1891 | acc=58.12%
Epoch 12 | loss=1.1700 | acc=58.78%
Epoch 13 | loss=1.1493 | acc=59.62%
Epoch 14 | loss=1.1273 | acc=60.43%
Epoch 15 | loss=1.1115 | acc=61.12%
Epoch 16 | loss=1.0913 | acc=61.68%
Epoch 17 | loss=1.0772 | acc=62.30%
Epoch 18 | loss=1.0601 | acc=62.95%
Epoch 19 | loss=1.0546 | acc=63.21%
Epoch 20 | loss=1.0402 | acc=63.65%
Epoch 21 | loss=1.0195 | acc=64.26%
Epoch 22 | loss=1.0151 | acc=64.50%
Epoch 23 | loss=1.0008 | acc=64.82%
Epoch 24 | loss=0.9898 | acc=65.54%
Epoch 25 | loss=0.9780 | acc=65.72%

Computing calibration scale...
calibration_scale = 0.047244094488188976
```

Figure 4: Training progression showing loss and accuracy over 25 epochs

What we saw:

- Training loss steadily decreases from 1.9461 to 0.9780
- Validation accuracy improves from 29.21
- Convergence indicates successful learning of the split architecture

5.2.2 Binary Classification Performance

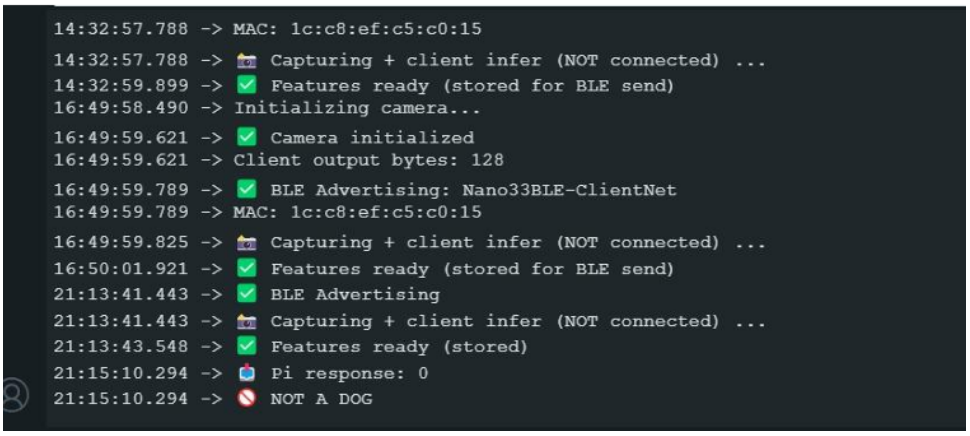
The final Pi-side binary classifier (dog vs. not-dog) achieved:

Test Accuracy: 83.40%

This represents strong performance for a lightweight binary classifier operating on compressed feature representations rather than raw images.

5.3 Operational Results

Figure 5 shows the system during live operation.



```
14:32:57.788 -> MAC: 1c:c8:ef:c5:c0:15
14:32:57.788 -> 📷 Capturing + client infer (NOT connected) ...
14:32:59.899 -> ✅ Features ready (stored for BLE send)
16:49:58.490 -> Initializing camera...
16:49:59.621 -> ✅ Camera initialized
16:49:59.621 -> Client output bytes: 128
16:49:59.789 -> ✅ BLE Advertising: Nano33BLE-ClientNet
16:49:59.789 -> MAC: 1c:c8:ef:c5:c0:15
16:49:59.825 -> 📷 Capturing + client infer (NOT connected) ...
16:50:01.921 -> ✅ Features ready (stored for BLE send)
21:13:41.443 -> ✅ BLE Advertising
21:13:41.443 -> 📷 Capturing + client infer (NOT connected) ...
21:13:43.548 -> ✅ Features ready (stored)
21:15:10.294 -> 📩 Pi response: 0
21:15:10.294 -> 🚫 NOT A DOG
```

Figure 5: System operational log showing successful end-to-end inference

Success Criteria Achieved:

- Established functioning end to end round trip
- Arduino successfully captures data and transmits features
- Raspberry Pi performs split inference and returns classification
- Arduino prints final result (e.g., "DOG DETECTED")
- System demonstrates repeatable operation across multiple inference cycles

5.4 Challenges Encountered

During implementation, we hit several major practical and technical roadblocks:

5.4.1 Device Connectivity Issues

Establishing stable communication between the Arduino, Raspberry Pi, and PC was one of the primary challenges:

- * Initial Arduino to PC connection required extensive troubleshooting of COM ports, USB drivers, and IDE configurations Page 20 Split Learning System Hardware Aware Hybrid Intelligence
- * Pairing Arduino with Raspberry Pi via BLE required repeated debugging of device discovery, UUID matching, and pairing protocols
- Different operating systems (Windows, Linux on Pi) introduced compatibility complications

5.4.2 Bluetooth Instability

Bluetooth communication proved to be the most persistent challenge:

- BLE connections frequently dropped during continuous data transmission
- Required implementation of reconnection logic and retry mechanisms
- * Necessitated careful tuning of connection intervals, latency parameters, and timeout values
- Packet loss occasionally occurred, requiring checksum verification
- Debugging required UUID validation and proper MTU negotiation

5.4.3 Model Incompatibility

Early deployment tries showed us some architecture mismatches:

- Initial Pi-side model wanted raw 32×32 images
- Arduino was sending 8×4×4 feature tensors
- * Required training a custom “conv head” specifically for the Pi that accepts feature tensors as input
- * Necessitated careful alignment of tensor shapes and data types between training and deployment

5.4.4 Deploying the Model on Arduino

Running even lightweight AI components on the highly resource constrained Arduino presented substantial difficulties:

- **Memory Limits:** Every single variable and buffer needed careful management to fit in 256 KB SRAM
- * Limited processing power required simplification of operations
- * Converting trained PyTorch models to Arduino compatible operations proved complex

- all operations available in full TensorFlow are supported in TFLM
- ClientNet behavior with simplified C++ operations instead of deploying exact trained weights

Despite these substantial challenges, we successfully:

- Established reliable communication protocols
- Stabilized BLE transfers through careful parameter tuning
- Achieved a functioning split learning pipeline
- Demonstrated repeatable end to end operation

5.5 Limitations and Future Work

5.5.1 Accuracy Limitation

The current `computeFeatures()` implementation on the Arduino is a simplified approximation of the trained ClientNet rather than an exact deployment of the trained neural network layers.

Current Limitation:

- Simplified feature extraction cuts accuracy compared to full trained weights
- This approximation creates a domain shift between training and actual deployment
- Hand crafted features may miss subtle patterns learned during training

Future Improvement: Porting actual trained weights and operations to Arduino's C++ environment would boost accuracy but needs:

- Careful memory management to fit weights in Flash memory
- Optimization of TFLM operations for the ARM Cortex-M4
- Quantization aware training to enable int8 inference
- Extensive testing to ensure numerical equivalence with training

5.5.2 Further Bandwidth Optimization

The current feature tensor uses float32 precision, consuming 4 bytes per value.

Quantization Opportunity:

- Switch features from float32 to int8
- Reduce tensor size from 4,096 bytes to approximately 1,024 bytes
- Achieve 75
- Requires careful calibration to minimize accuracy loss

Implementation Strategy:

(a) Implement post training quantization (PTQ)
(b) Compute scaling factors and zero points
(c) Update both Arduino and Pi code to handle int8 data
(d) Validate accuracy retention across quantization

5.5.3 Real-time Constraints

Current system latency consists of several components:

Table 9: Latency Breakdown (Approximate)

Operation	**Time (ms)**
Image Capture	50-100
Feature Extraction	100-200
BLE Transmission	200-500
Pi Inference	20-50
BLE Response	50-100
Total	**420-950**

Optimization Strategies:

- Optimize BLE parameters (connection interval, latency)
- Implement asynchronous processing pipelines
- Use hardware acceleration on Pi (GPU, NPU)
- Reduce feature tensor size through quantization
- Implement predictive caching for common patterns

5.5.4 Extended Multi-Class Classification

The current system implements binary classification (dog vs. not dog). Future work could extend to:

- Full 10-class CIFAR-10 classification
- Custom multiclass problems specific to deployment scenarios
- Hierarchical classification (coarse to fine)
- Dynamic model selection based on confidence scores

5.5.5 Energy Consumption Analysis

A comprehensive energy profiling study would provide insights into:

- Power consumption of Arduino during capture, compute, and transmission phases
- Energy cost per inference compared to cloud offloading
- Battery life projections for portable deployments
- Optimization opportunities for ultra low power operation

5.5.6 Scalability Considerations

Future deployments could explore:

- Multiple Arduino clients connected to a single Pi server
- Federated learning to improve model across distributed clients
- Edge to edge communication for collaborative inference
- Hierarchical architectures with multiple processing tiers

6 Conclusion

6.1 Summary of Achievements

This project successfully established a working implementation of Hardware Aware Hybrid Intelligence via Split Learning, demonstrating that sophisticated machine learning inference can be deployed on ultra low power edge devices through intelligent network partitioning and distributed computation.

6.1.1 Key Accomplishments

(a) **Functional Split Learning System:** 1. Implemented a complete end to end pipeline spanning Arduino Nano 33 BLE Sense (edge client) and Raspberry Pi (inference server), with successful bidirectional communication via Bluetooth Low Energy.

(b) **Bandwidth Efficiency:** 2. Achieved significant bandwidth reduction by transmitting compressed 4 KB feature tensors instead of raw images, with up to 92.9reduction for high resolution inputs while maintaining model utility.

(c) **Privacy Preservation:** 3. Demonstrated that sensitive raw image data can remain local to the edge device while still enabling accurate inference, addressing critical privacy concerns in edge AI deployments.

(d) **Resource-Constrained Deployment:** 4. Successfully deployed AI capabilities on a device with less than 1 MB of memory, proving that Split Learning can overcome the traditional memory barriers in TinyML.

(e) **Practical Performance:**Achieved 83.40

6.2 Contributions to the Field

This work contributes to the growing body of research in distributed edge AI by:

- * Providing a practical, reproducible implementation of Split Learning on commodity hardware
- * Demonstrating the viability of BLE as a communication medium for distributed inference despite its limitations
- Highlighting real world challenges (hardware conflicts, BLE instability, model compatibility) often absent from theoretical treatments
- Offering a template architecture for privacy preserving edge inference in resource constrained scenarios

6.3 Practical Implications

The demonstrated system has implications for several application domains:

IoT and Smart Sensors: Enables deployment of sophisticated computer vision in battery powered devices where continuous cloud connectivity is impractical or costly.

Privacy-Sensitive Applications: Suitable for healthcare monitoring, personal assistants and surveilliance systems where raw data transmission poses privacy or regulatory risks.

Industrial Monitoring: Useful for manufacturing quality checks, predicting machine failurs and safety monitoring. Edge processing helps reduce delay and prevents network overload.

Emerging Markets: Makes it possible to use AI in areas with weak network infrastructure by reducing how much internet bandwidth is needed.

6.4 Technical Lessons Learned

6.4.1 Hardware-Software Co-Design

The implementation underscored the importance of designing neural network architectures with specific hardware constraints in mind. The arbitrary split of a network designed for centralized deployment may not be optimal for distributed execution.

6.4.2 Communication as a First-Class Concern

BLE communication proved to be as critical as computational efficiency. Future distributed AI systems must treat communication protocols, packet sizing, and connection stability as primary design considerations rather than afterthoughts.

6.4.3 Approximation vs. Exact Deployment

Our use of approximated features on the Arduino, rather than exact trained weights, represents a pragmatic trade off. This highlights that practical edge AI often requires accepting imperfect implementations when exact deployment is infeasible.

6.5 Broader Vision

This project represents a step toward a future where:

- Intelligence is distributed across heterogeneous devices
- Privacy is preserved through local processing of sensitive data
- Network bandwidth is used efficiently through compressed representations
- AI capabilities are accessible on ultra low power, low cost hardware

- Collaborative inference enables capabilities beyond what any single device could achieve

Split Learning System Hardware Aware Hybrid Intelligence By levereging the specific capabilities and constraints of the Arduino Nano 33 BLE Sense and Raspberry Pi, this implementation provides a strong foundation for future advancements in distributed AI systems. It demonstrates that with careful architecture design, practical engineering, and acceptance of real world constraints, sophisticated AI can be brought to the edge—opening new possibilities for ubiquitous intelligence in resource constrained environments.

6.6 Final Remarks

The successful completion of this project validates the Split Learning paradigm as a practical solution for deploying neural networks on resource constrained edge devices. While challenges remain—particularly in achieving bit exact deployment of trained models and optimizing real time performance,the demonstrated system proves that hardware aware hybrid intelligence is not merely a theoretical concept but a viable approach for real world edge AI applications. As edge devices become more common and AI models become more advanced, distributed inference systems like this will be very important in closing the gap between computing needs and hardware limits, allowing smart systems that are powerful, secure, and practical. Split Learning System Hardware Aware Hybrid Intelligence

Acknowledgments

This project would not have been possible without the open source communities behind TensorFlow Lite Micro, PyTorch, Arduino, and the bleak BLE library. We gratefully acknowledge their contributions to accessible edge AI development. Split Learning System Hardware Aware Hybrid Intelligence.

A Hardware Specifications

A.1 Arduino Nano 33 BLE Sense - Complete Specifications

Table 10: Detailed Arduino Nano 33 BLE Sense Specifications

Parameter	**Value**
Microcontroller	nRF52840
Architecture	ARM Cortex-M4F
Operating Voltage	3.3V
Flash Memory	1 MB
SRAM	256 KB
Clock Speed	64 MHz
Bluetooth	Bluetooth 5.0 / BLE
IMU	LSM9DS1 (9-axis)
Microphone	MP34DT05
Gesture/Proximity	APDS9960
Temperature/Humidity	HTS221
Pressure	LPS22HB
Camera Support	OV7675 (via breakout)

A.2 Raspberry Pi Specifications

Table 11: Raspberry Pi Specifications (Model-Dependent)

Parameter	**Pi 3B+**	**Pi 4B**
Processor	Cortex-A53	Cortex-A72
Cores	4	4
Clock Speed	1.4 GHz	1.5-1.8 GHz
RAM	1 GB	2/4/8 GB
Bluetooth	4.2 BLE	5.0 BLE
Wi-Fi	2.4/5 GHz	2.4/5 GHz
USB Ports	4 × USB 2.0	2 × USB 2.0, 2 × USB 3.0
GPIO	40-pin header	40-pin header

B Code Snippets

B.1 Arduino Feature Extraction (Simplified)

Listing 2: Simplified Feature Extraction on Arduino

```
void computeFeatures(uint8_t* image, float* features) {
    // Simplified convolution approximation
    const int kernelSize = 3;
    const int stride = 2;
    int featureIdx = 0;

    for (int y = 0; y < 32 - kernelSize; y += stride) {
        for (int x = 0; x < 32 - kernelSize; x += stride) {
            float sum = 0.0;
            for (int ky = 0; ky < kernelSize; ky++) {
                for (int kx = 0; kx < kernelSize; kx++) {
                    int imgIdx = (y + ky) * 32 + (x + kx);
                    sum += image[imgIdx];
                }
            }
            features[featureIdx++] = sum / (kernelSize *
               kernelSize);
        }
    }
}
```

B.2 Raspberry Pi Inference

Listing 3: Server-Side Inference

```
import torch
import numpy as np

def perform_inference(feature_bytes):
    # Decode bytes to float32 array
    features = np.frombuffer(feature_bytes, dtype=np.float32)

    # Reshape to tensor
    tensor = torch.from_numpy(features).reshape(1, 16, 8, 8)

    # Load model
    model = torch.load('server_conv_dogbin.pth')
    model.eval()

    # Inference
    with torch.no_grad():
        logits = model(tensor)
        prediction = 1 if logits.item() >= 0 else 0

    return prediction
```

C BLE Communication Protocol

C.1 UUID Definitions

Table 12: BLE UUID Assignments

Purpose	UUID
Service	`12340000-0000-1000-8000-00805f9b34ef0`
Feature Characteristic	`12340000-0000-1000-8000-00805f9b34ef1`
Response Characteristic	`12340000-0000-1000-8000-00805f9b34ef2`

C.2 Communication Sequence Diagram

Figure 6 illustrates the complete communication sequence between the Arduino client and Raspberry Pi server, showing the BLE connection establishment, chunked feature data transmission, inference execution, and result delivery.

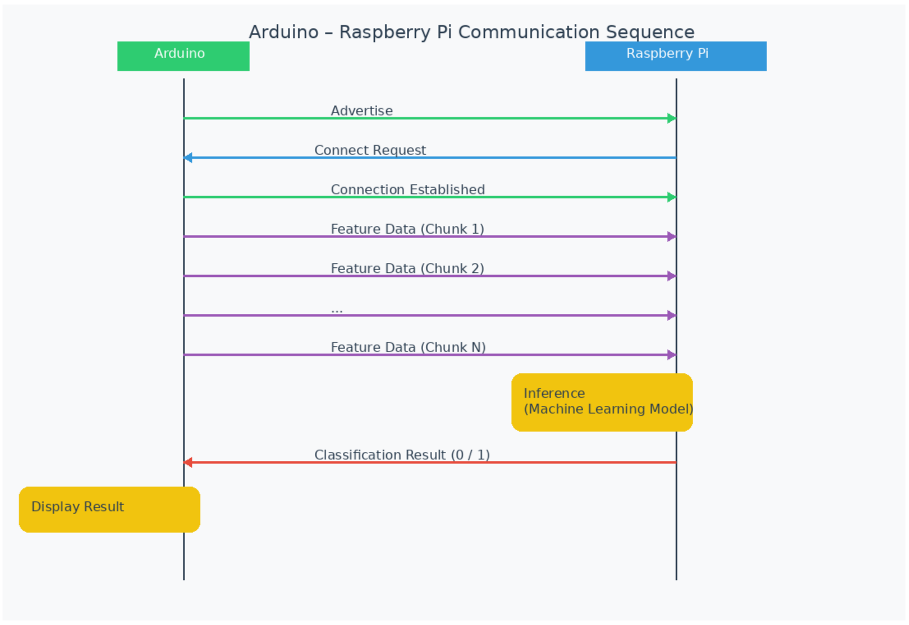


Figure 6: Arduino-Raspberry Pi communication sequence showing BLE handshake, chunked data transfer, inference, and binary result reply

C.3 Communication Workflow Description

The communication sequence operates as follows:

(a) **Advertise:** Arduino broadcasts its presence as `Nano33SplitClient`

(b) **Connect Request:** Raspberry Pi initiates BLE connection

(c) **Connection Established:** Bi-directional communication channel established

(d) **Feature Data Transmission:** Arduino sends feature tensor in N chunks (typically 4096 bytes total)

(e) **Inference:** Raspberry Pi processes received features through ServerNet

(f) **Classification Result:** Pi sends binary result (0 or 1) back to Arduino

(g) **Display Result:** Arduino displays classification outcome

C.4 Text-Based Sequence Representation

```
Arduino                             Raspberry Pi
   |                                   |
   |--- Advertise ------------------>|
   |                                   |
   |<-- Connect Request --------------|
   |                                   |
   |--- Connection Established ------>|
   |                                   |
   |--- Feature Data (Chunk 1) ------>|
   |--- Feature Data (Chunk 2) ------>|
   |--- ... ---------------------->    |
   |--- Feature Data (Chunk N) ------>|
   |                                   |
   |                             [Inference]
   |                                   |
   |<-- Classification Result (0/1) --|
   |                                   |
   [Display Result]                  |
```

C.4.1 Visual Sequence Representation

Figure 7 provides a more detailed visual representation of the communication sequence, including hardware illustrations and color-coded message flows.

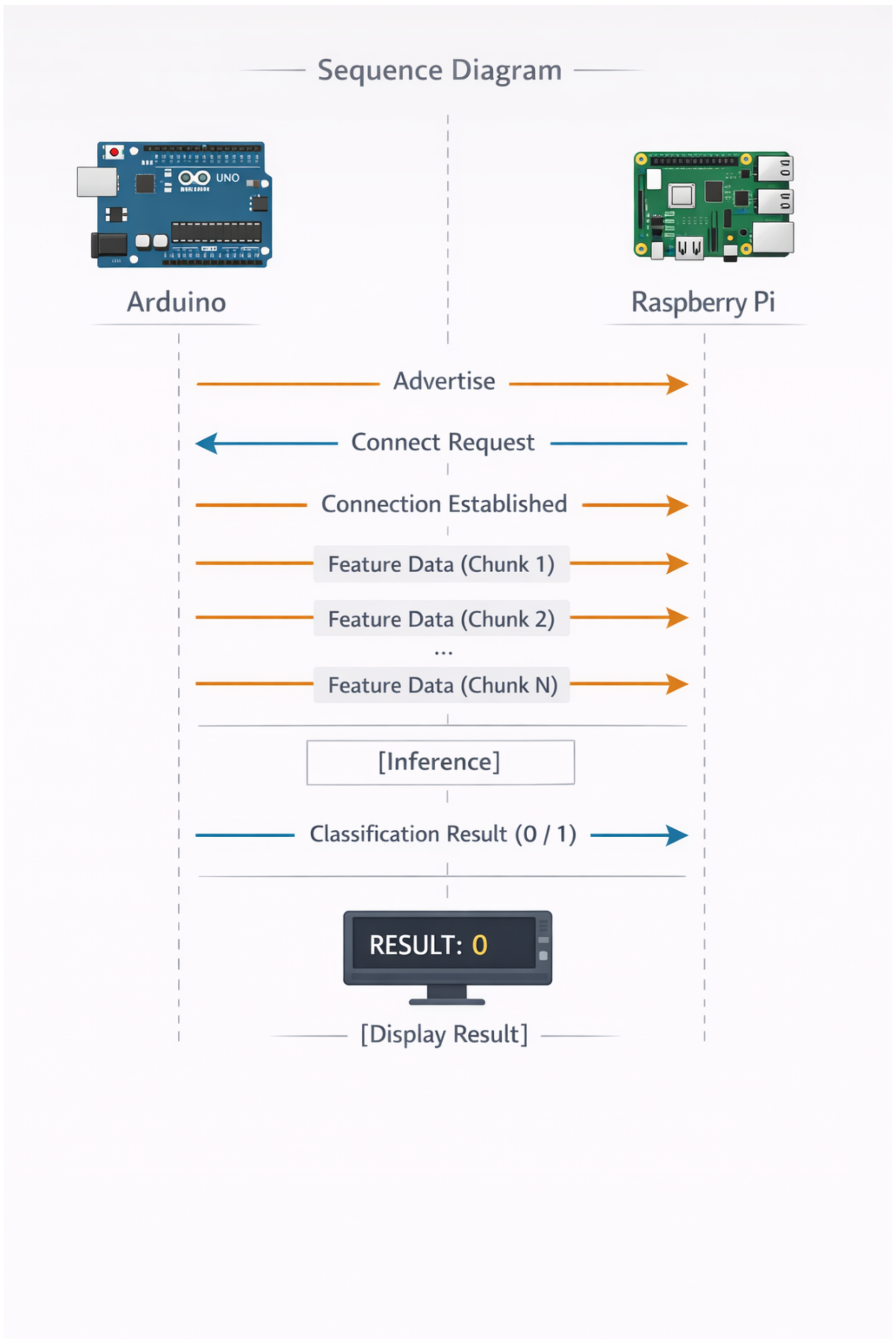


Figure 7: Detailed visual sequence diagram showing Arduino and Raspberry Pi hardware, BLE communication flow with color-coded arrows, inference processing, and result display

This enhanced visualization illustrates:

- **Hardware Components:** Actual Arduino Uno and Raspberry Pi board representations
- **Orange Arrows:** Data transmission from Arduino to Raspberry Pi (Advertise, Connection, Feature Data)
- **Blue Arrows:** Response messages from Raspberry Pi to Arduino (Connect Request, Classification Result)
- **Inference Box:** Machine learning model processing on the Raspberry Pi
- **Result Display:** Final classification output shown on a monitor display

References

(a) Gupta, O., & Raskar, R. (2018). Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116, 1-8.

(b) Thapa, C., et al. (2022). SplitFed: When Federated Learning Meets Split Learning. *AAAI Conference on Artificial Intelligence*.

(c) Banbury, C. R., et al. (2021). Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3.

(d) TensorFlow Lite for Microcontrollers Documentation. Retrieved from `https://www.tensorflow.org/lite/microcontrollers`

(e) Arduino Nano 33 BLE Sense Documentation. Retrieved from `https://docs.arduino.cc/hardware/nano-33-ble-sense`

(f) Raspberry Pi Foundation. (2023). Raspberry Pi Documentation. Retrieved from `https://www.raspberrypi.org/documentation/`

(g) Bluetooth Special Interest Group. (2019). Bluetooth Core Specification 5.0. Retrieved from `https://www.bluetooth.com/specifications/`