Federated On-Device Training on Arduino Nano 33 BLE

Haoran Li Jiuen Feng Yuhe Bian Zhehao Chen

December 2025

Outline

Project Context

- Hardware: **Arduino Nano 33 BLE** + TinyML Shield.
- Sensors: **IMU** – accelerometer, gyroscope, orientation.
- Target task: classify IMU segments into multiple gestures / movements in “Spell Game”.
- Federated learning on two boards, communication through BLE
- Constraint: training **directly on the board** with very limited RAM and compute.

Why Federated Learning on Tiny Devices?

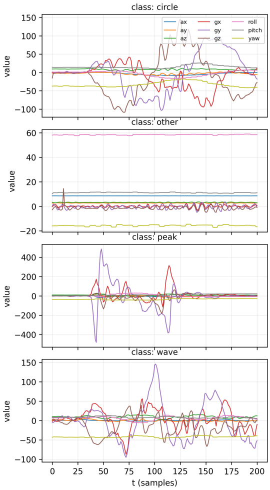
- In many IoT scenarios, there are **multiple devices** observing related data distributions.
- Sending raw sensor data to a server is:
  - Bandwidth expensive.
  - Potentially privacy sensitive.
- Federated learning idea:
  - Each device trains a **local model** on its own data.
  - Devices only exchange **model parameters**, not raw data.

Raw Data from Edge Impulse

- Data collected with Edge Impulse pipeline.
- Stored as `.cbor` messages:
  - Payload includes: `interval_ms`, `sensors`, `values`.
  - Each sample: around **2000 ms**, sampled at **100 Hz**.
- Shape of `values`:

$$(T, 9) \approx (201, 9),$$

where 9 channels are (accel, gyro, orientation).



Why Not Use Raw 201×9 Data on Board?

- Flattened vector length: $201 \times 9 = 1809$ features.
- With 32-bit floats, a single sample already takes:

$$1809 \times 4 \approx 7.1 \text{ kB}.$$

- On-device training uses:
  - Training data buffer.
  - Network weights and gradients.
  - Stacks, BLE buffers, etc.
- Conclusion: **too large** for robust on-board training.
- Solution: **compress each segment into 75 scalar features**.

75-D Feature Extraction (Python)

- For each segment $v \in \mathbb{R}^{T\times 9}$:
  - Global statistics per channel (9 dims):
    - mean, std, min, max $\Rightarrow 9 \times 4 = 36$.
  - Split time into 3 segments:
    - mean in each segment $\Rightarrow 3 \times 9 = 27$.
  - Per-channel energy:

$$\text{energy} = \frac{1}{T}\sum_t v_{t,c}^2 \Rightarrow 9 \text{ dims.}$$

  - Magnitude RMS for 3 groups (accel / gyro / orientation):

$$\text{RMS}_{\text{acc}},\ \text{RMS}_{\text{gyro}},\ \text{RMS}_{\text{ori}} \Rightarrow 3 \text{ dims.}$$

- Total: $36 + 27 + 9 + 3 = \mathbf{75}$ features per sample.

Train / Val / Test Split

- Python preprocessing:
  - Read all `training` and `testing` `.cbor` files.
  - Convert to $X \in \mathbb{R}^{N \times 75}$, label vector y.
  - Map label names (e.g., `circle`, `noise`) to integers.
  - Randomly shuffle and split into:
    - Train.
    - Validation.
    - Test.
- Store as `.npz` for PC training and also export as `data.h` for Arduino.

Network Architecture on Nano 33 BLE

- Simple fully connected network:

$$75 \to 64 \to \text{classes\_cnt}.$$

- Activation:
  - Hidden layer: ReLU.
  - Output layer: softmax.
- Loss:
  - Cross-entropy between predicted probabilities and one-hot labels.
- Implemented in C with:
  - Manually allocated layers and neurons.
  - Forward and backward propagation using SGD.

On-Device Data & Normalization

- `data.h` contains:
  - `train_data[train_cnt][75]`.
  - `validation_data[val_cnt][75]`.
  - `test_data[test_cnt][75]`.
  - `train_labels`, `validation_labels`, `test_labels`.
  - Feature-wise `feature_min[75]`, `feature_max[75]` from train set.
- Before feeding to the network (on board):

$$x'_j = \frac{x_j - \text{feature\_min}_j}{\text{feature\_max}_j - \text{feature\_min}_j}.$$

- Normalization is implemented as a small C function operating on the `input[]` buffer.

Baseline: Local On-Device Training

- Optimizer: **SGD** with fixed learning rate.
- Example parameters:
  - `#define LEARNING_RATE 0.0015`
  - `#define EPOCH 50`
- Training loop on board:
  - Shuffle training indices.
  - For each sample:
    1. Load feature vector into `input[]`.
    2. Normalize using `feature_min/max`.
    3. Forward propagation.
    4. Backward propagation, update weights.
- Accuracy monitored on train / val / test after each epoch.

From Single Board to Federated Setup

- We have two Nano 33 BLE boards:
  - Each board has its own local dataset from different users' gesture recordings.
- Idea: In each round
  - Each board trains locally for several epochs.
  - Boards use BLE to **exchange network weights**.
  - Then **average** the weights as a simple federated aggregation.
- No raw sensor data is transmitted.

Parameter Packing for BLE

- In the C code, the network is represented as:
  - Layers $L[i]$.
  - Neurons with weight vector `W[]` and bias `B`.
- Function `packUnpackVector(Type)`:
  - `PACK`: serialize all weights and biases into a flat `WeightBiasPtr[]` buffer.
  - `UNPACK`: read back from buffer to local network.
  - `AVERAGE`: average between received buffer and local network and update both.
- This buffer is then sent via BLE characteristic(s) between the two boards.

Simple Federated Averaging

- In standard FedAvg:

$$w^{(t+1)} = \sum_k \frac{n_k}{N} w_k^{(t+1)},$$

where w_k is client k's local weights.

- In our prototype with two boards:
  - Use equal weighting as an approximation:

$$w_{\text{new}} = \frac{w_A + w_B}{2}.$$

  - Implemented in `packUnpackVector(AVERAGE)`.
- Federated round:
  1. Each board trains locally for some epochs.
  2. Exchange weight buffers via BLE.
  3. Average and update local networks.

Federated Training Schedule

- Example schedule:
  - Local epochs per round: E (e.g., 1–5).
  - Number of federated rounds: R.
- On each board:
  1. Repeat local training for E epochs.
  2. Pack parameters and send to peer via BLE.
  3. Receive peer parameters into buffer.
  4. Call `packUnpackVector(AVERAGE)` to update network.
- Trade-off:
  - Larger E: fewer communications, more local drift.
  - Smaller E: more communication, better synchronization.

PC-Side Reference Training (PyTorch)

- Use the same 75-D features and labels in Python:
  - Network: 75–64–`classes_cnt`.
  - Loss: cross-entropy.
  - Optimizer: **Adam**.
- Observations:
  - Adam with hundreds of epochs can train the network to a good accuracy.
  - Simple SGD (without momentum, on the PC) is much harder to tune and converges slowly.
- PC model acts as an **upper bound** for the on-device performance.

% Placeholder for PC training curves (Adam vs SGD)

Comparing Local vs Federated On-Device Training

- Baseline: Each board trains independently on its own data.
  - May overfit to its own user / recording style.
- Federated version:
  - Periodic parameter exchange and averaging.
  - Helps share knowledge about different local distributions.
- Metrics to compare:
  - Training accuracy on each board.
  - Validation / test accuracy using a held-out set (from PC).

% Placeholder for accuracy curves: local vs federated

Limitations

- **Optimizer**: on-device training uses plain SGD; no Adam or momentum due to:
    - Code complexity.
    - Extra memory for moment estimates.
- **Communication**:
    - BLE bandwidth is limited.
    - Transmitting full parameter vector can be slow.
- **Scalability**:
    - Prototype only tested with two boards.
    - No central server; logic is peer-to-peer.

Possible Improvements

- On-device optimization:
  - Implement lightweight variants of Adam / momentum, or adaptive learning rate schedules.
- Model compression:
  - Quantize weights / send only deltas to reduce BLE traffic.
- Protocol:
  - Generalize from 2 boards to K devices, with either a central aggregator or a gossip protocol.
- Data:
  - Collect more user-specific IMU trajectories to better demonstrate personalization + federated averaging.

Summary

- Built a **full pipeline**:
  - Raw IMU segments $\Rightarrow$ 75-D features.
  - Python preprocessing and PC training.
  - Exported `data.h` for on-device training.
- Implemented a small **on-device neural network**:
  - 75–64–`classes_cnt`, ReLU + softmax, SGD training.
- Designed and tested a **federated learning prototype**:
  - Two Nano 33 BLE boards exchanging weights via BLE.
  - Simple parameter averaging after local training epochs.
- Demonstrated that even with tight resource constraints, federated ideas can be prototyped on microcontrollers.

Thank you for listening!