Once-for-All Channel Mixers (HYPERTINYPW): Generative Compression for TinyML

Yassien Shaalan
Independent Researcher
yassien@gmail.com

ABSTRACT

Neural networks on microcontrollers are constrained by kilobytes of flash/SRAM, where 1×1 pointwise (PW) mixers often dominate memory even after `INT8` quantization. We present HYPERTINYPW, a *compression-as-generation* method that replaces most *stored* PW weights with *generated* weights: a shared micro-MLP synthesizes PW kernels once at load time from tiny per-layer codes, caches them, and executes them with standard integer operators. This preserves commodity MCU runtimes and incurs only a one-off synthesis cost; steady-state inference matches `INT8` separable CNNs. Sharing a latent basis across layers removes cross-layer redundancy, while keeping PW_1 in `INT8` stabilizes early, morphology-sensitive mixing. We also introduce TinyML-faithful *packed-byte* accounting (generator, heads/factorization, codes, kept PW_1, backbone) and a unified evaluation protocol with validation-tuned thresholds and bootstrap CIs. On three ECG benchmarks (Apnea-ECG, PTB-XL, MIT-BIH), HYPERTINYPW improves the macro-F_1–vs.–flash Pareto: at $\sim$225 kB it achieves near-iso performance to a $\sim$1.4 MB CNN while being $6.31\times$ smaller (84.15% fewer bytes), retaining $\geq$95% of large-model macro-F_1. Beyond ECG, HYPERTINYPW transfers to TinyML audio: on Speech Commands keyword spotting it reaches 96.2% test accuracy (98.2% best validation), supporting that generate-and-cache channel mixing applies broadly to embedded sensing workloads where repeated linear mixers dominate memory.

1 INTRODUCTION

Deep learning models for biosignal analytics, such as ECG, are increasingly expected to run *directly on microcontrollers (MCUs)*. On-device inference improves *privacy, reliability, and energy proportionality*, since data never leaves the sensor and decisions can be made in real time. Yet MCUs offer only tens of kilobytes of flash and SRAM and limited compute extensions (e.g., Arm M-series DSP). These constraints create a sharp tension between the promise of local analytics and the cost of deploying modern convolutional neural networks.

Among TinyML backbones, *separable 1D CNNs* are attractive: depthwise (DW) convolutions dominate multiply–accumulates, while *pointwise (PW, 1×1) convolutions* concentrate most parameters. Unfortunately, even after `INT8` quantization, multiple PW layers still dominate flash usage, often pushing total storage beyond 64 kB. Thus, the PW mixers-not the depthwise layers-become the limiting factor for MCU deployment.

Classical compression techniques-quantization, pruning, low-rank or tensor factorization-shrink parameters but still *store a full set of weights for every PW layer*. Dynamic weight generation (HyperNetworks, DFN, CondConv) can reduce redundancy but typically *generate kernels per input*, adding branching, SRAM pressure, and latency jitter incompatible with real-time MCU workloads. What is missing is a strategy that *directly targets the PW bottleneck* while respecting strict device constraints: no per-example branching, minimal SRAM, and unmodified integer kernels.

We propose to replace *stored PW weights* with *generated weights once per layer at load time*. Our method, HYPERTINYPW, uses a shared micro-MLP to synthesize most PW kernels from tiny per-layer codes, while deliberately keeping the first PW (PW_1) in `INT8` to anchor morphology-sensitive early mixing. Generation occurs once-at boot or lazily when first needed-then synthesized weights are cached and reused. Inference proceeds entirely with *standard integer operators*, ensuring compatibility with existing TinyML stacks. We also contribute a *TinyML-faithful packed-byte accounting* that includes the generator, its heads, the retained PW_1, codes, and the backbone.

Beyond storage savings, the design enforces a *shared latent basis across layers*. This reduces redundancy and acts as an *efficient-coding prior*: layers reuse common generative factors rather than relearning separate mixers. From

a representation-learning perspective, this behaves like *implicit multi-task regularization across layers*, helping preserve balanced detection even under tight flash budgets. An analogy is skill reuse in humans-experts deploy compact routines (rhythmic or grammatical motifs) across many tasks. Similarly, our generator emits shared transformations that multiple layers can repurpose.

Beyond compression, our goal is to make channel mixing *deployable* on real MCUs. We therefore (i) report *packed-byte* sizes that match what ships; (ii) keep inference on unmodified integer kernels (CMSIS-NN/TFLM compatible); and (iii) characterize *boot vs. lazy* synthesis, peak SRAM, and the latency/energy impact of one-shot generation. This positions cross-layer generative mixing as a *systems design* for TinyML, not just a model variant.

We validate the approach on three representative ECG tasks, Apnea-ECG (minute-level apnea detection), PTB-XL (diagnostic proxy), and MIT-BIH (AAMI arrhythmia grouping)—using *record/patient-wise splits* to avoid identity leakage. Experiments include comprehensive ablations (hybrid vs. all-synth, latent sizes (d_z, d_h, r), precision 4–8 bit, knowledge distillation, focal loss), structured alternatives under equal flash, and system-level boot-time and SRAM measurements. Finally, we analyze the *accuracy–memory Pareto frontier*, offering guidance for MCU-constrained deployment.

We complement the core ECG study with (i) a non-ECG TinyML benchmark (keyword spotting on Speech Commands) to assess modality transfer, (ii) an aggressive compression baseline (ternary weights) to contextualize the extreme size–accuracy regime, and (iii) clarifications on deployability constraints (e.g., why general-purpose entropy coding can be incompatible with strict SRAM/boot limits), while HYPERTINYPW preserves standard `INT8` kernels after one-shot synthesis.

Contributions. (1) **Compression-as-generation for PW layers.** A shared micro-MLP synthesizes most 1×1 mixers from tiny per-layer codes once at load time, while PW_1 remains `INT8` to stabilize early morphology; steady-state inference uses standard integer kernels (no custom ops). (2) **TinyML-faithful accounting and deployment.** Exact *packed-byte* sizes (generator, heads/factorization, codes, kept PW_1, backbone), plus a deployment analysis of *boot* vs. *lazy* synthesis, SRAM peaks, and compatibility with CMSIS-NN/TFLM. (3) **Latency/energy profiling.** A lightweight pipeline that reports per-inference latency and energy on MCU backends (virtual and on-device), isolating generation overhead from steady-state compute. (4) **Rigorous validation under MCU budgets.** Cross-dataset results (Apnea-ECG, PTB-XL, MIT-BIH), ablations over (d_z, d_h, r) and precision (4–8 bit), and Pareto curves of macro-F_1 vs. packed flash that expose the $\sim$225 kB elbow. At $\sim$225 kB, HYPERTINYPW compresses a 1.4 MB baseline by $6.31\times$ (84.15% fewer bytes) while retaining $\approx 95\%$ of its macro-F_1 on Apnea-ECG and achieving near-iso macro-F_1 on PTB-XL. This places HYPERTINYPW at the mid-budget elbow of the accuracy–flash Pareto while preserving a TinyML-faithful deployment path (packed-byte accounting, one-shot load-time synthesis, and integer-only inference).

2 RELATED WORK

Our work connects four areas: TinyML backbones, compression of channel mixing, dynamic weight generation, and ECG deep learning. Below we situate HYPERTINYPW in each thread and address likely counter-arguments.

Tiny models and MCU deployment. MobileNet-style backbones (Howard et al., 2017; Sandler et al., 2018; Howard et al., 2019) reduce MACs via depthwise (DW) layers but concentrate parameters in 1×1 pointwise (PW) mixers. MCU software stacks (CMSIS-NN, TFLM) enable `INT8` inference (Lai et al., 2018; David et al., 2021), MLPerf Tiny standardizes tasks (Banbury et al., 2021), and co-design (MCUNet, Once-for-All) explores NAS for fit/latency (Lin et al., 2020; Cai et al., 2020). One might argue that NAS can simply remove or shrink PW layers. Empirically, completely eliminating PW mixing collapses channel reuse and hurts accuracy; even after NAS, remaining PW layers dominate flash. HYPERTINYPW is orthogonal: it retains PW expressivity but amortizes storage across layers.

Compression methods. Quantization (Jacob et al., 2018) (including ternary/binary variants (Li et al., 2016; Zhu et al., 2017)), pruning/sparsity (Han et al., 2016; Frankle & Carbin, 2019), and low-rank/tensor factorization (Denton et al., 2014) reduce parameters, yet the model still *stores* a parameterization per PW layer. In the 32–64 KB regime, several PW matrices remain the bottleneck. A counter-claim is that aggressive low-rank or k-sparse PW will suffice. However, these approaches attack *within-layer* redundancy and do not capture *cross-layer* regularities; they also incur per-layer metadata that adds up at TinyML scale. HYPERTINYPW instead ties layers through a shared generator and tiny layer codes, and can *compose* with low-rank/sparse heads (we evaluate structured baselines at equal flash).

Structured transforms. Algebraic operators (circulant, Toeplitz, Kronecker, ACDC) (Sindhwani et al., 2015; Moczulski et al., 2016) shrink matrices and sometimes speed up inference. It may seem these remove the need for generation. In practice, they restrict each PW *independently*, leaving the per-layer storage pattern intact; they also require kernel support to realize speedups on MCUs. Our approach is complementary: structured heads can further compress

H_l (or A_l, B), but the novelty is *cross-layer* synthesis that leaves inference kernels unchanged.

Dynamic and generated weights. HyperNetworks (Ha et al., 2017), dynamic filter networks (Jia et al., 2016), CondConv (Yang et al., 2019), and dynamic convolution (Chen et al., 2020) generate or mix kernels *per input*, which suits GPUs/TPUs. A natural question is whether small dynamic modules could also fit MCUs. The barrier is not only parameter count but per-input control flow, SRAM peaks, and latency jitter, which violate real-time/energy budgets. HYPERTINYPW generates *once at load time*, caches weights, and then runs standard `INT8` inference with no runtime control changes.

ECG deep learning and TinyML. Large ECG models achieve strong accuracy (Rajpurkar et al., 2017; Hannun et al., 2019; Ribeiro et al., 2020) on datasets such as PTB-XL (Wagner et al., 2020) and MIT-BIH (Moody & Mark, 2001), but embedded deployments often omit *packed-byte* accounting or exceed MCU flash. One might claim that downsampling or binarization makes any model fit. In practice, these shortcuts degrade morphology-sensitive detection and still leave multiple PW layers too large. We explicitly report deployable packed bytes (generator, heads, codes, kept PW_1, backbone) and demonstrate record/patient-wise generalization within 32–64 KB.

Positioning. Relative to compression, HYPERTINYPW avoids per-layer storage by synthesizing most PW weights from a shared micro-MLP and tiny codes; relative to structured transforms, it introduces cross-layer parameter tying that can coexist with algebraic heads; relative to dynamic methods, it eliminates per-input cost by performing *layer-constant* generation at load time; relative to prior ECG TinyML, it provides TinyML-faithful accounting and a deployable path under strict MCU budgets. The combination—shared generator, keep-first-PW hybrid, and packed-byte evaluation—appears novel in MCU-scale biosignal inference.

3 METHOD

Our goal is to compress separable CNNs for microcontrollers by replacing most stored pointwise (PW) channel mixers with compact, generated representations. Instead of storing every PW kernel, we store only a tiny per-layer code and use a shared generator to expand these codes into full kernels once at load time. This preserves compatibility with standard `INT8` inference while tying layers together through a shared latent basis.

The section is organized as follows. We first introduce the setup and notation for separable CNNs. We then describe our generative channel mixing approach, including how codes, generator, and heads interact. Next, we detail deployable storage via packed-byte accounting, present the training objective, and explain the calibration pipeline. Finally, we outline MCU deployment options, show the load-time synthesis algorithm, and compare complexity to conventional PW stacks.

3.1 Setup and notation

We consider a compact separable 1D CNN for ECG: each block applies a depthwise temporal convolution (DW) followed by a 1×1 PW channel mixer. Let $x \in \mathbb{R}^{C_{\mathrm{in}}\times T}$ be the input (channels $\times$ time). A PW layer l multiplies the channel dimension by $W_l \in \mathbb{R}^{C_{\mathrm{out}}^{(l)}\times C_{\mathrm{in}}^{(l)}}$ (bias omitted). In conventional TinyML deployment, every W_l is stored (typically `INT8`), and the sum $\sum_l C_{\mathrm{out}}^{(l)} C_{\mathrm{in}}^{(l)}$ dominates flash.

3.2 Generative channel mixing (layer-constant synthesis)

Instead of storing each full PW matrix, we assign a small *code* z_l to each layer and let a shared generator produce its weights once at load time. Crucially, generation is *layer-constant*, never per input: the synthesized weights are cached and then reused by standard integer kernels.

Each code $z_l \in \mathbb{R}^{d_z}$ is mapped by the generator g_ϕ into an embedding:

$$h_l = g_\phi(z_l), \tag{1}$$

where $h_l \in \mathbb{R}^{d_h}$ is a compact hidden vector summarizing layer l.

A light per-layer head H_l then projects this embedding into the vectorized kernel and reshapes it into the full PW matrix:

$$\widehat{w}_l = H_l h_l \in \mathbb{R}^{C_{\mathrm{out}}^{(l)} C_{\mathrm{in}}^{(l)}}, \qquad \widehat{W}_l = \mathrm{reshape}\big(\widehat{w}_l,\ C_{\mathrm{out}}^{(l)}, C_{\mathrm{in}}^{(l)}\big). \tag{2}$$

To compress further, H_l can be factorized into a small per-layer adapter A_l and a shared matrix B:

$$H_l = A_l B, \quad A_l \in \mathbb{R}^{(C_{\mathrm{out}}^{(l)} C_{\mathrm{in}}^{(l)})\times r},\ B \in \mathbb{R}^{r\times d_h},\ r \ll d_h. \tag{3}$$

This means most of the capacity lives in B (shared), while each layer only stores its lightweight adapter A_l.

Because early mixing is morphology-sensitive, we deliberately keep PW_1 as a stored `INT8` layer and synthesize only $\mathrm{PW}_{2:L}$.

3.3 Deployable storage: packed-byte accounting

We carefully count the deployable flash footprint. For each tensor τ with N_τ elements stored at bitwidth b_τ, the footprint is

$$\left\lceil \frac{N_\tau\, b_\tau}{8} \right\rceil \ \text{bytes}. \tag{4}$$

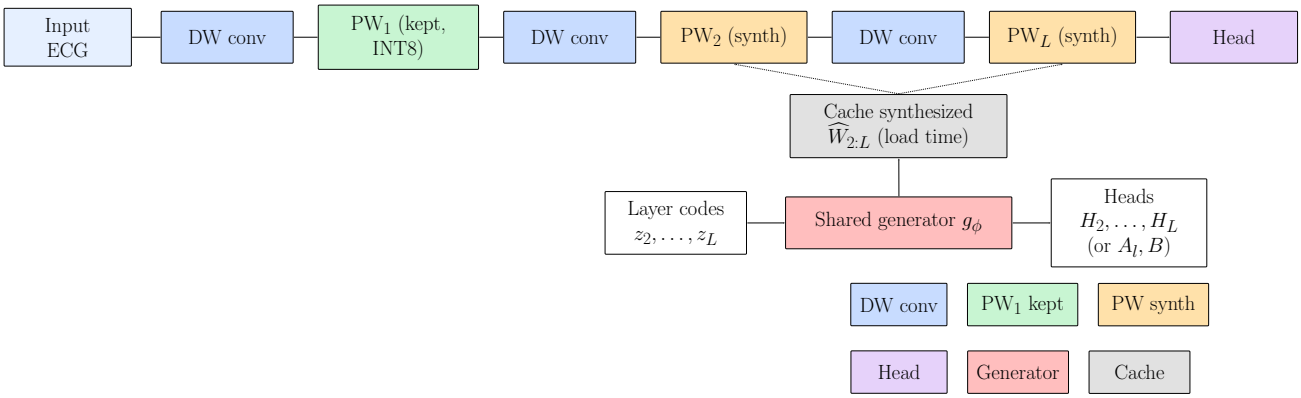


Figure 1: Architecture overview. Depthwise (blue), PW$_1$ kept in INT8 (green), synthesized PW layers (orange), classifier head (purple). The shared generator (red) produces PW$_{2:L}$ once at load time and caches them (gray); steady-state inference uses standard integer kernels.

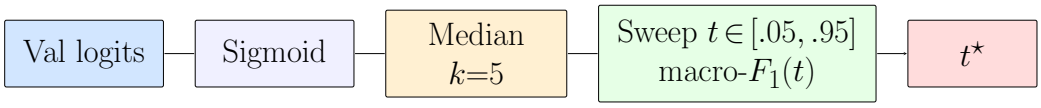


Figure 2: Validation thresholding pipeline.

The total flash is the sum across generator parameters, heads (or factorized A_l, B), codes, the kept PW$_1$, and backbone layers. We quantize $\{\phi, H_l, z_l\}$ to 4/6/8 bits, while keeping stem, DW, PW$_1$, and classifier at `INT8`. This ensures a fair, deployable accounting.

3.4 End-to-end training (stability, size, and imbalance-aware)

We train the generator and student jointly with AdamW, applying GN(1) instead of BN for small-batch stability, NaN-safe initialization, gradient clipping, and an EMA of weights. The composite loss is

$$\begin{aligned}\mathcal{L} = \; & \mathrm{CE}(y, \hat{y}) \\ & + \lambda_{\text{foc}}\, \mathrm{Focal}_\gamma(y, \hat{y}) \\ & + \lambda_{\text{KD}}\, \mathrm{KL}\big(\sigma(\hat{y}/T) \,\|\, \sigma(\hat{y}^{\text{teach}}/T)\big) \\ & + \lambda_{\text{feat}}\, \big\|\hat{f} - \hat{f}^{\text{teach}}\big\|_2^2 \\ & + \lambda_{\text{softF1}}\, \mathcal{L}_{\text{softF1}}(\hat{y}, y) \\ & + \lambda_{\text{spec}}\, \mathcal{R}_{\text{spec}}(\theta) \\ & + \lambda_{\text{size}}\, \big\|\theta_{\text{heads,codes}}\big\|_1 .\end{aligned} \tag{5}$$

where CE drives baseline accuracy, focal loss counteracts class imbalance by emphasizing hard or minority examples, KL distillation and feature matching transfer knowledge from a larger teacher, soft-F_1 optimizes directly for the evaluation metric, spectral regularization stabilizes dynamics by constraining layer smoothness, and the L_1 penalty enforces compact codes and heads to reduce flash usage; jointly, these terms promote stability, imbalance-awareness, and size efficiency, enabling HYPERTINYPW to converge reliably while remaining sensitive to rare events under microcontroller constraints. Unlike prior TinyML training objectives that rely on only CE or CE+KD, this formulation unifies metric-aware, imbalance-aware, and compression-aware terms into a single end-to-end objective, making the loss itself a vehicle for co-designing accuracy, robustness, and deployability.

3.5 MCU deployment: boot vs. lazy synthesis

Synthesis runs once per layer and cached weights are reused thereafter. Two schedules are possible: *boot synthesis*, generating all PW$_{2:L}$ at startup (faster inference, longer boot), or *lazy synthesis*, generating each PW$_l$ on first use (shorter boot, one-time stall). Steady-state inference always uses `INT8` kernels; g_ϕ is never called per input. Peak SRAM is bounded by the largest PW plus activations, and weights can be streamed to flash if needed. Regarding Kernel compatibility, synthesis emits `INT8`-quantized PW tensors laid out to match standard 1×1 conv GEMV paths in CMSIS-NN/TFLM. No graph rewrites or per-example control flow are introduced. As a result, deployment reduces to a one-time "weight install" followed by vanilla integer inference.

Algorithm 1 Load-time synthesis and caching (MCU side)

for $l = 2$ **to** L **do**
  $h_l \leftarrow g_\phi(z_l)$ *(Eq. 1)*
  $\widehat{w}_l \leftarrow H_l h_l$ *(or $A_l B h_l$, Eq. 2–3)*
  $\mathrm{PW}_l \leftarrow \mathrm{reshape}(\widehat{w}_l)$; cache PW_l
Inference: run `INT8` DW/PW with cached $\{\mathrm{PW}_l\}$; no calls to g_ϕ per input.

3.6 Complexity and storage

A standard PW stack stores $\sum_l C_{\mathrm{out}}^{(l)} C_{\mathrm{in}}^{(l)}$ `INT8` parameters. In HYPERTINYPW, the stored footprint becomes

$$\mathrm{flash\_total} = |\phi| + \Big(\sum_l |H_l| \text{ or } \sum_l |A_l| + |B| \Big) + \sum_l |z_l| + |PW1| + |\mathrm{stem, DW, head}|. \quad (6)$$

With $(d_z, d_h, r) \ll C_{\mathrm{out}}^{(l)} C_{\mathrm{in}}^{(l)}$, the packed bytes (Eq. 4) drop sharply while inference cost matches the baseline. Boot/lazy synthesis adds a one-off cost proportional to the total PW size.

3.7 VAE-head baseline (decoder-free at deploy)

As an additional baseline, we evaluate a lightweight 1D VAE encoder whose latent z is classified by a small MLP. The decoder is used only during training for latent regularization and reconstruction consistency, then discarded at deployment; the stored footprint therefore includes only the encoder and classification head, reported under the same packed-byte accounting. This baseline was originally considered to complement our generative synthesis approach by testing whether a learned latent manifold could implicitly encode meaningful structural priors without explicit parameter generation. However, while the VAE-head achieves competitive compactness, it lacks the ability to synthesize or adapt weights across layers, and its latent space regularization often trades off discriminative sharpness for reconstruction fidelity. As a result, it serves mainly as a contrastive probe showing that generative *weight synthesis*, rather than purely latent compression, is responsible for the gains observed in HYPERTINYPW.

4 SYSTEM PROFILING AND DEPLOYMENT

Why not entropy-code weights? General-purpose compressors (e.g., DEFLATE/LZMA or Huffman-style entropy coding) can shrink *stored* weights offline, but typical MCU deployment stacks assume statically allocated tensor arenas and do not tolerate a full-model decompression peak in SRAM at load time (Warden & Situnayake, 2019; David et al., 2021). In contrast, HYPERTINYPW synthesizes each PW layer independently and can be scheduled at boot or lazily per layer, bounding transient SRAM peaks and leaving steady-state inference unchanged.

Targets and kernels. We target Arm M-series MCUs running TFLM with CMSIS-NN integer kernels; all PW layers execute through stock 1×1 conv/GEMV paths. Our method requires no custom ops: synthesized PW tensors are cached as ordinary INT8 weights.

Profiling methodology (proxies). We separate one-shot load-time synthesis from steady state. Steady-state latency is obtained from an instruction/cycle model of the CMSIS-NN/TFLM kernels on a virtual MCU configuration (e.g., Arm FVP / Renode / QEMU). Energy per inference is derived from cycles via a calibrated board-level current model (normalized nJ/inference). These are *hardware-agnostic proxies* intended for model-to-model comparison rather than absolute device benchmarking.

Boot vs. lazy synthesis. *Boot* compiles $\mathrm{PW}_{2:L}$ at startup (higher startup time, no first-inference stall). *Lazy* compiles on first use (fast boot, one-time per-layer stall). Both schedules yield identical steady-state latency because inference runs entirely on cached INT8 weights.

SRAM peaks and streaming. Peak SRAM is the maximum of $\{$largest PW activation tensor, workspace$\}$. If writable flash is available, synthesized PW tensors can be streamed back to flash to cap SRAM peaks; otherwise we synthesize layer-wise with buffer reuse.

Portability. Because inference uses unmodified integer kernels and generation happens only at load time, the approach ports to any TFLM-compatible stack with INT8 convolutions. We release scripts to regenerate packed-byte counts and cycle/energy proxies.

5 EXPERIMENTAL SETUP

5.1 Datasets & Preprocessing

We choose three *single-lead* ECG corpora that jointly cover screening, clinical diagnostics, and arrhythmia detection while stressing MCU constraints (short windows, low-rate inputs, and class priors from balanced to highly skewed). To test whether the approach is specific to ECG, we also include a standard TinyML audio benchmark (keyword spotting) with lightweight front-end features.

Apnea-ECG (PhysioNet (Ichimaru & Moody, 1999; Goldberger et al., 2000)): Originally collected for sleep apnea screening, this dataset provides single-lead overnight ECGs with minute-wise apnea annotations. We segment each record into 18 s windows at 100 Hz, apply per-window z-score normalization with a variance guard, and split 80/10/10 by *record/patient* to avoid identity leakage. Its label distribution is notably skewed toward non-apnea seg-

ments, making it a good testbed for imbalance-aware training.

PTB-XL (Wagner et al., 2020): A large clinical 12-lead ECG dataset with diverse diagnostic annotations, designed to reflect real-world recording variability. It has 21,837 records from 18,885 patients and recordings at both 500 and 100 Hz. We downsample to 100 Hz, use lead II, binarize labels (NORM vs. any diagnostic superclass), extract 10 s windows, and re-materialize 8/1/1 folds for train/val/test. Its scale, label heterogeneity, and real-world noise make it a strong benchmark for generalization under compression.

MIT-BIH Arrhythmia (PhysioNet (Moody & Mark, 2001; Goldberger et al., 2000)): A long-standing reference corpus of arrhythmia recordings with beat-level annotations from 47 subjects. We adopt the AAMI binary setup (normal vs. arrhythmia) with single-lead inputs, following common tiny-ECG practice (Association for the Advancement of Medical Instrumentation (AAMI), 1998). This dataset is compact but challenging, with highly imbalanced arrhythmia types and patient-specific morphology differences.

Speech Commands (KWS) (Warden, 2018): A widely used TinyML audio benchmark for keyword spotting, consisting of 1 s utterances sampled at 16 kHz over a small fixed vocabulary. It contains 105,829 utterances over 35 words, recorded from 2,618 speakers. We follow the common 12-class setup (10 keywords + `unknown` + `silence`) and compute MFCC features (40 coefficients $\times$ 101 frames) per clip using a standard pipeline (Davis & Mermelstein, 1980). We train/evaluate on the official splits and report classification accuracy.

All datasets use a consistent window/clip-level prediction setup and lightweight normalization/feature extraction, enabling controlled comparisons under deployment-faithful constraints; dataset sizes and split compositions are summarized in Table 6 (and window-level counts for ECG in Table 7).

This mix provides complementary regimes, approximately balanced for Apnea-ECG (train) with a notable prior shift in val/test, mildly skewed for PTB-XL, and highly imbalanced for MIT-BIH, so we can study accuracy–flash tradeoffs and calibration under deployment-faithful conditions. The strong skew in MIT-BIH (Table 6) motivates reporting macro-F_1 and balanced accuracy in addition to AUC.

5.2 Model suite and baselines

We evaluate HYPERTINYPW alongside a diverse set of compact baselines spanning feature-engineered and deep 1D architectures. Unless stated otherwise, all models use the same training protocol in §5.3 and are evaluated with the same validation-tuned thresholding in §5.4. For ECG, we compare models across packed-flash budgets by post-training packing at 8 or 6 bits (no QAT).

HYPERTINYPW (ours). We implement *compression-as-generation*: most pointwise (PW) channel mixers are not stored explicitly, but synthesized once at load time from compact per-layer codes using a shared micro-MLP generator; the first mixer PW_1 is kept in `INT8` to anchor morphology-sensitive early mixing. We sweep generator/code sizes $(d_z, d_h) \in \{(4, 12), (6, 16)\}$, packing bit-widths $\{8, 6\}$ for $\{\phi, H, z\}$, and optionally enable knowledge distillation (KD) from a larger teacher.

TinyVAE-Head. A lightweight depthwise/pointwise (DW/PW) encoder produces features h, followed by a variational head that predicts a latent distribution $q_\psi(z \mid h)$; a reparameterized sample z is then fed to a small classifier. This baseline tests whether a learned low-dimensional bottleneck (via a VAE-style latent) can achieve favorable accuracy–flash trade-offs. We evaluate both with and without KD, and apply the same 8/6-bit post-training packing used for HYPERTINYPW.

TinySeparableCNN. A compact 1D depthwise-separable CNN using DW temporal filtering followed by PW mixing layers (Howard et al., 2017). This is a standard TinyML design point that typically performs well at very small flash budgets. We run it under both 8-bit and 6-bit post-training packing.

CNN3_Small. A small conventional 1D CNN backbone with a few temporal convolution blocks and a lightweight classifier head. This baseline represents non-separable compact CNNs where capacity comes from stored convolutional filters rather than generator sharing. We run it under 8/6-bit packing.

ResNet1D_Small. A small residual 1D CNN with short skip connections to stabilize optimization and improve representation at modest size (He et al., 2016). This provides a stronger compact baseline than plain CNNs at similar flash. We run it under 8/6-bit packing.

RegularCNN. A higher-capacity 1D CNN used as the accuracy-oriented reference/teacher (Hinton et al., 2015). It provides an upper bound for the achievable performance on each dataset and serves as the KD teacher where KD is enabled. For fair size accounting, we also report its packed-flash footprint under the same packing pipeline.

HRVFeatNet. A classical feature-engineered baseline: fixed 16-D HRV (and amplitude) statistics computed per window are fed to a linear classifier (Acharya et al., 2006). This establishes a low-flash lower bound that does not rely on learned deep representations.

Across datasets, this suite spans feature-engineered and deep models and covers a broad range of packed-flash budgets for Pareto analysis.

5.3 Training

Optimization uses **AdamW**; BatchNorm is replaced with **GroupNorm(1)** for small/variable batches. We use focal loss (macro-F_1 is the selection metric). Optional KD blends focal with a teacher KL term from `RegularCNN` (default α=0.7, T=2). We apply two light regularizers (soft-F_1 auxiliary and spectral leakage penalty), gradient clipping, and track an **EMA** of weights. Unless stated, we use post-training packing at 8 or 6 bits (no QAT).

5.4 Evaluation & Selection

We operate at window level. On validation we pass logits through a sigmoid, apply a 1D median filter (k=5) across windows, and sweep a uniform grid $t \in [0.05, 0.95]$ (19 points) to pick the threshold $t^\star$ that maximizes macro-F_1. For **test**, we evaluate the **RAW** (non-EMA) checkpoint at the validation-tuned $t^\star$ with the same smoothing. EMA is reported as an ablation with its own $t^\star_{\text{EMA}}$; main tables use RAW to avoid post-hoc selection.

5.5 Metrics & Accounting

We report accuracy, balanced accuracy, macro-F_1 (primary), and ROC–AUC with **95%** cluster bootstrap CIs over record/patient groups (1,000 resamples; stratified fallback otherwise). Selected runs include confusion matrices.

For efficiency we report (*i*) **deployable packed bytes** (kB) that include the generator core, per-layer heads (or A_ℓ, B), latent codes, the kept PW_1, and the backbone; (*ii*) parameter count and MACs; and (*iii*) **system *proxies***:

- **Latency (proxy):** steady-state cycles from instruction-count models of CMSIS-NN/TFLM integer conv/GEMV kernels (compiled with `-O3`) on an Arm M-class configuration in *virtual* MCU backends (e.g., Arm FVP / Renode / QEMU). $\text{PW}_{2:L}$ are synthesized once at load time and then cached; we report steady-state inference only.
- **Energy (proxy):** normalized nJ/inference derived from cycles via a board-level current model (datasheet-calibrated). These figures are model-comparable but not tied to a specific board SKU.

All Pareto plots compare *macro-F_1 vs. packed flash*. Scripts reproduce packed-byte accounting and cycle/energy proxies exactly.

Table 1: Best *test* results under ≤ 256 kB packed flash.

Dataset	Acc	Macro-F_1	BalAcc	AUC	Flash (kB)	Model
Apnea-ECG	0.7391	0.7172	0.7164	0.8324	225.46	HYPERTINYPW
PTB-XL	0.6310	0.6291	0.6327	0.8760	225.46	HYPERTINYPW
MIT-BIH	0.9016	0.5673	0.562	0.962	225.27	HYPERTINYPW

5.6 Reproducibility

We fix seeds, enforce record/patient disjointness, and use the same threshold grid and median filter for all runs. To support reproducibility, we release the full implementation of HYPERTINYPW, including model definitions, training and evaluation scripts, packed-byte accounting utilities, experiment runners, and scripts/instructions for obtaining the public datasets, at [1].

6 RESULTS

We evaluate HYPERTINYPW under TinyML constraints on three single-lead ECG tasks and contrast it with compact and large baselines. We report *packed-byte* flash sizes (what ships) and emphasize MCU-feasible deployments that preserve standard `INT8` inference kernels.

6.1 MCU-feasible budget ($\leq$256 kB)

Across datasets, the best configurations under a $\leq$256 kB flash envelope consistently use a hybrid design (keep PW_1, synthesize $\text{PW}_{2:L}$) and lie near an elbow around $\sim$225 kB (Table 1). Despite aggressive storage constraints, HYPERTINYPW preserves balanced detection on Apnea-ECG and PTB-XL while remaining competitive on the more skewed MIT-BIH setting.

6.2 Compression versus a large baseline

Relative to the $\sim$1.4 MB `regularcnn1d` baseline, HYPERTINYPW at $\sim$225 kB achieves a $6.31\times$ **flash reduction** (84.15% fewer bytes) while retaining $\geq$95% of the large model's macro-F_1 on Apnea-ECG and PTB-XL. Table 2 makes the trade-off explicit: although smaller baselines compress more aggressively in absolute terms, HYPERTINYPW delivers the strongest *accuracy retention per stored byte* at the mid-budget elbow. This is the key practical distinction from ultra-compact baselines: they win in the extreme low-flash corner, whereas HYPERTINYPW preserves much more of the large model's accuracy once a deployment can afford roughly 200–250 kB.

[1] `https://github.com/yassienshaalan/tinyml-gen`

Table 2: Compression vs. `regularcnn1d` (flash=1422.00 kB). Reported are packed flash, compression factor (×), flash reduction (%), and macro-F_1 retention (%) on Apnea-ECG and PTB-XL.

Model	Flash (kB)	Compress (×)	Flash ↓ (%)	Apnea F_1 retain (%)	PTB F_1 retain (%)
HYPERTINYPW	225.46	**6.31**	**84.15**	**95.40**	**99.97**
resnet1dsmall	62.49	22.76	95.61	87.52	98.92
tinyseparablecnn	14.49	98.14	98.98	88.59	98.11
tinyvaehead	10.16	139.96	99.29	85.59	94.36
hrvfeatnet	0.53	2683.02	99.96	66.56	84.87

Table 3: Best model under representative packed-flash budgets, using test macro-F_1 as the primary selection metric. The preferred architecture switches near the mid-budget regime.

Dataset	Budget (kB)	Winner	Flash (kB)	Macro-F_1
Apnea-ECG	≤32	tinyseparablecnn	14.49	0.6660
Apnea-ECG	≤64	tinyseparablecnn	14.49	0.6660
Apnea-ECG	≤128	tinyseparablecnn	14.49	0.6660
Apnea-ECG	≤256	HYPERTINYPW	225.46	**0.7172**
PTB-XL	≤32	tinyseparablecnn	14.49	0.6174
PTB-XL	≤64	resnet1dsmall	62.49	0.6225
PTB-XL	≤128	resnet1dsmall	62.49	0.6225
PTB-XL	≤256	HYPERTINYPW	225.46	**0.6291**
MIT-BIH	≤32	tinyvaehead	10.16	0.5218
MIT-BIH	≤64	resnet1dsmall	62.49	0.5450
MIT-BIH	≤128	resnet1dsmall	62.49	0.5450
MIT-BIH	≤256	HYPERTINYPW	225.27	**0.5673**

6.3 Budget-wise winner switch

The appendix-level budget tables reveal a consistent *winner switch* as the flash envelope grows. Below ≈64 kB, compact separable/residual baselines are preferred because the generator overhead of HYPERTINYPW cannot yet be amortized; at ≤256 kB, HYPERTINYPW becomes the best-performing compressed model on all three ECG datasets. Table 3 condenses this deployment rule into a single view.

This table complements the Pareto curves by turning the frontier into a simple model-selection rule: choose ultra-compact baselines for the extreme corner (≤32–64 kB), but switch to HYPERTINYPW once the platform can absorb a mid-budget footprint, where synthesized channel mixers yield the largest gain per additional stored byte.

6.4 Pareto efficiency (macro-F_1 vs. packed flash)

We visualize the full accuracy–flash trade-off in Fig. 3. The Pareto fronts show that HYPERTINYPW dominates a wide range of baselines in the mid-budget regime, with diminishing returns beyond the ∼225 kB knee.

6.5 Cross-domain evaluation (Keyword Spotting)

Audio domain (keyword spotting). To test transfer beyond biosignals, we evaluate HYPERTINYPW on keyword spotting using Google Speech Commands v0.02 (Warden, 2018). Each 1 s, 16 kHz clip is represented by standard MFCC features (40 coefficients × 101 frames) (Davis & Mermelstein, 1980). We train for 20 epochs with Adam (lr $= 10^{-3}$, batch size 64) and report test accuracy. Training converges rapidly (>96% validation accuracy by epoch 4), peaks at 98.2% validation accuracy (epoch 12), and shows mild late-stage overfitting thereafter. HYPERTINYPW achieves 96.2% test accuracy; under an all-`INT8` packed-parameter upper bound, the stored model parameters are ≈235 kB for this configuration, fitting within a ≤256 kB flash envelope (excluding MFCC feature-extraction code and runtime).

Table 4: Keyword spotting (Speech Commands) and ternary-weight baseline on PTB-XL. Sizes are packed flash (kB); ternary includes scale-factor overhead.

Task	Method	Flash (kB)	Metric	Value
Speech Commands (12-class)	HYPERTINYPW	≤235	Acc	0.962
PTB-XL (binary)	HYPERTINYPW	72.29	BalAcc	0.7936
PTB-XL (binary)	Ternary (2-bit)	6.70	BalAcc	0.5532

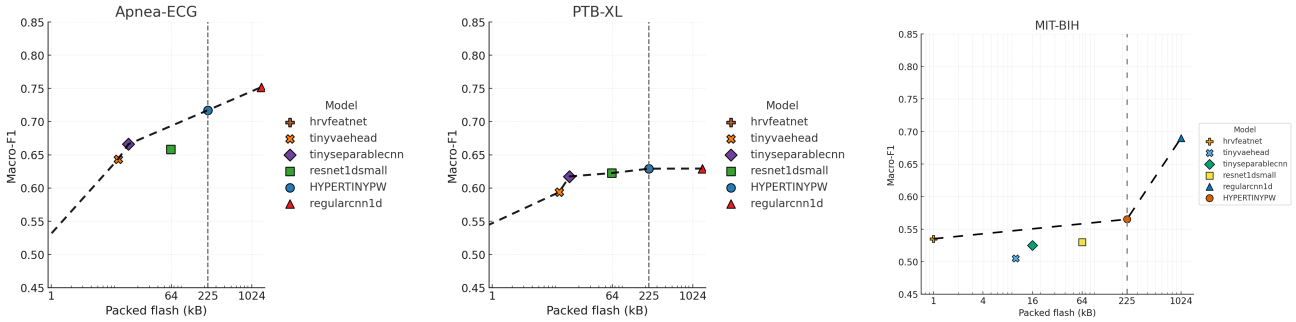
Table 5: Multi-scale validation on Apnea-ECG (test accuracy; 20 epochs each). "Flash" is the packed representation stored for deployment. Compression ratio is constant at 12.5× across scales.

Configuration	Params	Flash (kB)	Acc
Small (base=16, latent=16)	231,325	72.29	0.8206
Medium (base=20, latent=20)	347,453	108.58	0.8043
Large (base=24, latent=24)	489,015	152.82	0.8488

Extreme quantization baseline (ternary weights) To contextualize accuracy-size trade-offs under more aggressive compression than `INT8`, we implement a ternary-weight baseline following standard ternary quantization (Li et al., 2016; Zhu et al., 2017) and compare on PTB-XL using balanced accuracy. As summarized in Table 4, ternary packing yields a much smaller stored footprint, but incurs a large drop in balanced accuracy relative to HYPERTINYPW in our setting.

On PTB-XL, ternary quantization reduces flash further but collapses toward a majority-class predictor in our setting (balanced accuracy 55.3%; Table 4); HYPERTINYPW preserves balanced performance while remaining MCU-feasible. Per-class accuracies and confusion-matrix diagnostics for this comparison are reported in Appendix §B (Experiment 2).

Multi-scale validation (100K–500K parameter regime) To validate the claimed operating range, we sweep three model scales by varying base width and code dimension and train each for 20 epochs on Apnea-ECG. Table 5 shows that (i) packed flash scales smoothly with width (72 kB → 153 kB) and (ii) test accuracy remains stable (80–85%), supporting the scalability claim. Additional ablations at matched flash (bit-width/KD variants) are summarized in Table 14, and full per-model results under common flash budgets are reported in Tables 11–13.



(a) Apnea-ECG **(b)** PTB-XL **(c)** MIT-BIH

Figure 3: Pareto fronts: macro-F_1 vs. packed flash. Points indicate evaluated configurations; lines trace the non-dominated frontier. Lines trace the non-dominated frontier; the $\sim$225 kB elbow consistently yields the best accuracy–flash trade-off.

7 DISCUSSION

We interpret the results along four axes: (i) where accuracy gains originate under tight flash budgets, (ii) how compression and efficiency compare across model sizes, (iii) how the approach generalizes beyond ECG (supported by added cross-domain evidence), and (iv) what deployment behaviors matter in practice. The expanded camera-ready results sharpen two points that were previously deferred to the appendix: first, the *winner switch* across flash budgets is now explicit in Table 3; second, the compression table makes clear that HYPERTINYPW is not simply "smaller than a large CNN" but is the strongest *retention-oriented* compressed model in the mid-budget regime.

Practical deployment guidance In TinyML deployments, the binding constraint is often a *joint* flash/SRAM envelope rather than flash alone, because feature extraction, activation buffers, and kernel workspaces compete for SRAM. HYPERTINYPW keeps *steady-state inference* compatible with standard `INT8` operators after a one-shot materialization step: weights are synthesized once (boot or lazy-on-first-use) and then cached as ordinary tensors, so the runtime operator set and per-example control flow remain unchanged. When boot-time latency is critical, synthesis can be amortized (e.g., executed once at install time or cached across power cycles), trading a slightly larger on-device footprint for the same OTA/update size of the packed representation. This makes the flash–boot–SRAM trade-off explicit without changing the runtime operator set. This lens connects the Pareto elbows in Fig. 3 with the per-model tables and explains why HYPERTINYPW is most effective around the 200–250 kB region.

What the added budget table changes Table 3 turns the full appendix sweep into a concrete deployment rule. On Apnea-ECG, the preferred model stays ultra-compact up to $\leq$128 kB and then flips to HYPERTINYPW at $\leq$256 kB; PTB-XL shows the same pattern with a transitional residual baseline at 64–128 kB; MIT-BIH similarly moves from VAE-head/residual baselines to HYPERTINYPW only once the budget reaches the mid-budget range. This is important because it clarifies that our claim is *not* that HYPERTINYPW dominates every budget, but rather that it consistently becomes the best compressed choice once the generator overhead is amortized. That is a more precise and actionable characterization for practitioners selecting models by flash budget.

Runtime proxies after one-shot synthesis The hardware-oriented appendix tables also clarify a subtle but practically important point: steady-state latency is not strictly monotonic in flash size. After one-shot synthesis and caching, HYPERTINYPW reaches 2.38 ms proxy latency on Apnea-ECG versus 3.72 ms for `regularcnn1d`, despite using much less flash; on PTB-XL, however, operator mix and tensor shapes dominate, and the lowest-flash model is not automatically the fastest. This reinforces that deployment choices should not be made from packed bytes alone: flash, SRAM, latency, and energy must be co-reported, especially when comparing architectures with different pointwise/depthwise balance.

Mid-budget elbow and why it appears HYPERTINYPW replaces most stored pointwise mixers with weights synthesized once at load time from small layer codes and a shared generator. This ties layers through common factors, reduces redundancy across mixers, and concentrates capacity where it matters. As a result, the first 200-250 kB buys a disproportionate gain in channel-mixing expressivity while keeping integer-only kernels; beyond this elbow, additional bytes yield diminishing returns. A multi-scale sweep (Table 5) supports the stability claim raised in review: in the $\sim$100K-500K parameter regime, performance changes smoothly with scale while packed flash remains governed by the same generator+code structure, consistent with an elbow driven by where channel-mixing capacity turns on efficiently.

Compression and efficiency in context Relative to a $\sim$1.4 MB CNN, HYPERTINYPW at $\sim$225 kB achieves

about $6.3\times$ lower flash (84% fewer bytes) with near-iso performance on PTB-XL and about 95% macro-F_1 retention on Apnea-ECG. Compared with $\leq$64 kB compact CNNs, it delivers the largest accuracy jump per kB, forming the elbow seen in the Pareto curves. The best-under-budget tables (Tables 11–13) sharpen this picture: HYPERTINYPW becomes the best choice at $\leq$256 kB, whereas the $\leq$32–64 kB corner is better served by compact separable/residual baselines. In short, HYPERTINYPW currently targets the *mid-budget* regime: in its present design, the generator-code-head decomposition converts $\sim$200 kB of flash into channel-mixing capacity especially efficiently, while the extreme $\leq$32 kB corner is better served by ultra-compact hand-designed baselines. We expect that further shrinking and re-parameterizing the per-layer heads (e.g., shared codebooks, lower-rank/head tying, or more aggressive mixed precision) can push this operating point downward and expand coverage toward smaller flash budgets.

Near-RegularCNN accuracy without RegularCNN size At the mid-budget elbow ($\sim$200–250 kB), our method matches (PTB-XL) or retains $\geq$95% (Apnea-ECG) of RegularCNN macro-F_1 while using $\sim$16% of its flash ($\sim$225 kB vs. $\sim$1.4 MB). On PTB-XL, the gap to RegularCNN at this budget is within the bootstrap 95% CI (absolute difference $\leq$0.5 points macro-F_1), indicating practical non-inferiority at deployment-friendly size. *Future work* will push toward full RegularCNN parity at mid budgets (via improved codebooks/heads and calibration) and shift the elbow to lower budgets ($\leq$128 kB) through generator distillation and mixed-precision caching.

Baselines across sizes At very small budgets (10–60 kB), hand-tuned separable or small residual CNNs are the best anchors; VAE-head is competitive for its size and HRV features set a classical lower bound (Tables 11–13 and Tables 8–10). Moving to $\sim$225 kB lets the generator express richer families of mixers while PW_1 stabilizes early morphology, which closes most of the remaining gap to large models without changing inference kernels (Tables 8–9 and Fig. 3).

A stronger compression comparator clarifies the design point: an aggressive ternary-weight baseline can achieve extremely small packed size, but in our PTB-XL experiment it exhibits unstable training dynamics and majority-class collapse (very low negative-class accuracy with high positive-class accuracy), yielding substantially lower balanced accuracy (Table 4; per-class accuracies and the confusion matrix are reported in Appendix §B, Experiment 2). This addresses the concern that a stronger quantization baseline may dominate: extreme quantization can win on bytes yet lose robustness under imbalance and calibration, whereas retaining a reliable morphology-sensitive front-end (PW_1) while compressing redundant mixers yields a more stable mid-budget trade-off.

Effect of label skew on the Pareto On MIT-BIH, the 6-10% positive rate penalizes macro-F_1 relative to AUC; nevertheless, HYPERTINYPW at $\sim$225 kB remains on the accuracy–flash frontier. On Apnea-ECG, the train$\rightarrow$val/test prior shift (50%$\rightarrow$66/62% positives; Appendix Table 6) favors recall-oriented thresholds; HYPERTINYPW preserves balanced detection while small baselines trade recall for size. PTB-XL's near-balanced splits explain why EMA can surpass RAW in some runs. Across all three, HYPERTINYPW's cross-layer generator yields the biggest accuracy jump per kB at the mid-budget elbow ($\sim$200–250 kB), and the split-level prior tables help interpret apparent AUC-F_1 discrepancies.

Calibration behavior and a lightweight remedy We select thresholds per branch after median smoothing. RAW is consistently stronger on Apnea-ECG and in the current MIT-BIH snapshot; EMA sometimes helps PTB-XL. On MIT-BIH, several EMA runs adopted high $t^\star$ and collapsed positives, indicating threshold drift under imbalance rather than weak separability. AUC around 0.96 with macro-F_1 near 0.56 suggests good ranking but a brittle global threshold; lightweight remedies such as per-class or subject-aware calibration, or simple beat-wise post-processing, could raise F_1 without adding flash or changing kernels.

Generalization and extensibility beyond ECG While the majority of the analysis focused on the ECG benchmarks, the mechanism targets repeated *linear mixing operators*, so it should transfer to other settings where parameter mass is dominated by repeated linear/channel-mixing layers. We substantiate this with a cross-domain validation on keyword spotting (Speech Commands): the same generate-and-cache mixer strategy achieves strong accuracy on audio features without changing the integer-only inference path (Table 4). More broadly, *compression-as-generation* is naturally extensible to (i) mobile CNNs with heavy 1×1 bottlenecks/expansions, (ii) transformer blocks where Q/K/V and MLP projections dominate parameters, and (iii) channel-mixing MLP-style blocks, provided many layers share weight structure up to low-dimensional variation captured by per-layer codes.

Positioning among efficiency methods Quantization/pruning compress *stored* weights per layer, and factorization reduces but does not eliminate per-layer redundancy; input-conditioned methods (e.g., HyperNetworks/CondConv) increase runtime/SRAM cost. HYPERTINYPW instead generates PW mixers *once* and then runs a standard `INT8` inference path, targeting a distinct trade-off: shared-weight expressivity with static deployment cost. We also explored NAS-style selection under the generator+head parameterization; at very small

targets, head/generator scaling can dominate and even inflate candidates, making strict budget-constrained search ill-conditioned. We therefore focus on direct baseline comparisons and diagnostics at deployment-relevant budgets.

Deployment considerations and limitations Synthesis is performed once (boot or first-use) and cached; steady-state inference uses unchanged `INT8` kernels. Packed-byte sizes include the generator, heads/factorization, codes, retained PW_1, and backbone, while peak SRAM is driven by the largest pointwise activation/workspace. Two limitations: (i) the current design prioritizes *accuracy–compression balance* in the mid-budget regime; further head/coding refinements (e.g., tighter sharing/low-rank tying) are expected to push the same mechanism toward smaller flash budgets, and (ii) latency/energy are proxy-based and should be validated on-device. Finally, end-to-end deployment may be dominated by modality-specific front-ends (e.g., MFCC extraction), so future evaluations should report full pipeline flash/SRAM/energy, not just model parameters.

Environmental and practical implications Beyond accuracy and compression, HYPERTINYPW reduces the number of stored parameter bytes that must be shipped, updated, and accessed from non-volatile memory. In large-scale edge deployments, these reductions can lower OTA/update payloads and flash pressure, and can translate into meaningful cumulative resource savings. The latency/energy tables also indicate that steady-state efficiency is not strictly monotonic with flash (it depends on operator mix and tensor shapes), reinforcing the need for *co-reporting* flash, SRAM, latency, and energy when selecting a deployment point on the Pareto.

8 CONCLUSION

We introduced HYPERTINYPW, a *compression-as-generation* approach for TinyML in which a shared micro-MLP synthesizes most 1×1 channel mixers from compact per-layer codes once at load time, while PW_1 remains `INT8` to anchor morphology-sensitive early mixing. The resulting models run with standard integer kernels and are reported using deployable *packed-byte* flash accounting. Across three ECG benchmarks, HYPERTINYPW consistently forms the mid-budget accuracy–flash elbow: at $\sim$225 kB it delivers a $6.31\times$ flash reduction versus a $\sim$1.4 MB CNN (**84%** fewer bytes) with **near-iso** PTB-XL performance and $\geq$**95%** macro-F_1 retention on Apnea-ECG, without custom kernels or per-example control flow. A cross-domain keyword spotting result (96.2% on Speech Commands) further supports that the same generate-and-cache mechanism generalizes beyond ECG.

More broadly, the mechanism extends to model families whose parameter mass is dominated by repeated *linear channel mixing*, including (i) efficient CNN blocks with heavy 1×1/projection layers, (ii) transformer-style Q/K/V and MLP projections, and (iii) channel-mixing MLP blocks where layers differ by low-dimensional variation. In these cases, synthesizing large linear operators from compact codes can reduce flash while preserving a standard integer inference path.

Future work. We will refine the generator/heads/codes design to push the operating point toward smaller flash budgets (e.g., shared codebooks, lower-rank/head tying, and mixed precision) while keeping integer-only inference; complete cycle-accurate latency/energy measurement including cold-boot synthesis; and strengthen calibration under imbalance. We also plan to extend compression-as-generation to additional modalities and architectures (efficient vision backbones, transformer blocks for audio/time-series, and MLP-based graph models) and quantify end-to-end gains in representative edge deployments.

REFERENCES

Acharya, U. R., Joseph, K. P., Kannathal, N., Lim, C. M., and Suri, J. S. Heart rate variability: a review. *Medical & biological engineering & computing*, 44(12):1031–1051, 2006.

Association for the Advancement of Medical Instrumentation (AAMI). Testing and reporting performance results of cardiac rhythm and st segment measurement algorithms, 1998. ANSI/AAMI EC57:1998/(R)2008.

Banbury, C. et al. Mlperf tiny benchmark. In *NeurIPS Datasets and Benchmarks Track*, 2021.

Cai, H., Gan, C., and Han, S. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations (ICLR)*, 2020.

Chen, Y., Dai, X., Liu, M., Chen, D., Yuan, L., and Liu, Z. Dynamic convolution: Attention over convolution kernels. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

David, R., Duke, J., Jain, A., Janapa Reddi, V., Jeffries, N., Li, J., Kreeger, N., Nappier, I., Natraj, M., Regev, S., Rhodes, R., Wang, T., and Warden, P. Tensorflow lite micro: Embedded machine learning for tinyml systems. In *Proceedings of Machine Learning and Systems*, volume 3, pp. 800–811, 2021. URL `https://proceedings.mlsys.org/paper_files/paper/2021/hash/8a8e816b5a84a55b1a2289a7d31240d3-Abstract.html`.

Davis, S. B. and Mermelstein, P. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366, 1980. doi: 10.1109/TASSP.1980.1163420.

Denton, E. L., Zaremba, W., Bruna, J., LeCun, Y., and Fergus, R. Exploiting linear structure within convolutional networks for efficient evaluation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014.

Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. In *International Conference on Learning Representations (ICLR)*, 2019.

Goldberger, A. L., Amaral, L. A., Glass, L., Hausdorff, J. M., Ivanov, P. C., Mark, R. G., Mietus, J. E., Moody, G. B., Peng, C.-K., and Stanley, H. E. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. *Circulation*, 101(23): e215–e220, 2000.

Ha, D., Dai, A. M., and Le, Q. V. Hypernetworks. In *International Conference on Learning Representations (ICLR)*, 2017.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR)*, 2016.

Hannun, A. Y., Rajpurkar, P., Haghpanahi, M., Tison, G. H., Bourn, C., Turakhia, M., and Ng, A. Y. Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network. *Nature Medicine*, 2019.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q. V., and Adam, H. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2019.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

Ichimaru, Y. and Moody, G. B. Development of the polysomnographic database on cd-rom. *Psychiatry and Clinical Neurosciences*, 53(2):175–177, 1999.

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. 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.

Jia, X., De Brabandere, B., Tuytelaars, T., and Van Gool, L. Dynamic filter networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Lai, L., Suda, N., and Chandra, V. Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus. *arXiv:1801.06601*, 2018.

Li, F., Liu, B., Wang, X., Zhang, B., and Yan, J. Ternary weight networks. *arXiv preprint arXiv:1605.04711*, 2016. doi: 10.48550/arXiv.1605.04711.

Lin, J., Chen, W.-M., Lin, Y., Gan, C., and Han, S. Mcunet: Tiny deep learning on iot devices. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Moczulski, M., Denil, M., Appleyard, J., and de Freitas, N. Acdc: A structured efficient linear layer. In *International Conference on Learning Representations (ICLR)*, 2016.

Moody, G. B. and Mark, R. G. The impact of the mit-bih arrhythmia database. *IEEE Engineering in Medicine and Biology Magazine*, 20(3):45–50, 2001.

Rajpurkar, P. et al. Cardiologist-level arrhythmia detection with convolutional neural networks. *arXiv:1707.01836*, 2017.

Ribeiro, A. L. P. et al. Automatic diagnosis of the 12-lead ecg using a deep neural network. *Nature Communications*, 2020.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

Sindhwani, V., Sainath, T. N., and Kumar, S. Learning with structured transforms for small-footprint deep learning. In *NIPS Workshop on Efficient Methods for Deep Neural Networks*, 2015.

Wagner, P., Strodthoff, N., Bousseljot, R. D., Samek, W., Schaeffter, T., et al. Ptb-xl, a large publicly available electrocardiography dataset. *Scientific Data*, 7(1):154, 2020.

Warden, P. Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*, 2018. doi: 10.48550/arXiv.1804.03209.

Warden, P. and Situnayake, D. *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019. ISBN 9781492052043.

Yang, B., Bender, G., Le, Q., and Ngiam, J. Condconv: Conditionally parameterized convolutions for efficient inference. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Zhu, C., Han, S., Mao, H., and Dally, W. J. Trained ternary quantization. In *International Conference on Learning Representations (ICLR)*, 2017. URL `https://openreview.net/forum?id=S1_pAu9xl`.

A ADDITIONAL TABLES AND DIAGNOSTIC DETAILS

This appendix collects additional tables and diagnostic details that complement the main-paper results and support reproducibility. It includes split-level class priors and window counts, complete per-model test metrics across datasets, "best-under-budget" summaries for common flash envelopes, steady-state latency/energy estimates after one-shot materialization, and Pareto-front visualizations summarizing the accuracy–flash trade-off.

A.1 Data splits and class balance

Table 6 reports split-specific class priors for each dataset. These priors matter because our decision threshold $t^\star$ is tuned on validation and then applied to test: when the positive prior shifts between splits, the operating point that maximizes macro-F_1 can move even if ranking quality (AUC) remains high. The effect is most pronounced on Apnea-ECG, where the training split is near-balanced but validation/test are substantially positive-skewed; this favors recall-oriented thresholds and helps explain why models with similar AUC can separate more clearly in macro-F_1 and balanced accuracy. PTB-XL is comparatively stable across splits, so differences between methods tend to reflect representational capacity rather than prior shift. MIT-BIH is highly imbalanced in all splits, which naturally penalizes macro-F_1 relative to AUC and makes calibration more sensitive: small changes in $t^\star$ can have large effects on minority-class F_1. Table 7 provides the corresponding window-level counts used in the capped (2,000) per-split evaluation; reporting both priors and counts makes it easier to interpret apparent metric discrepancies and to reproduce the thresholding conditions.

Table 6: Class priors (%) per split (Class 0 = negative/normal, Class 1 = positive/event for ECG; for KWS, Class 0 = non-target (`unknown+silence`), Class 1 = target keyword classes aggregated). These priors contextualize threshold-tuned macro-F_1 and balanced accuracy under prior shift.

Dataset (Split)	Class 0 (%)	Class 1 (%)	Samples
Apnea-ECG (Train)	50.25	49.75	2000
Apnea-ECG (Val)	33.85	66.15	2000
Apnea-ECG (Test)	38.40	61.60	2000
PTB-XL (Train)	53.20	46.80	2000
PTB-XL (Val)	44.35	55.65	2000
PTB-XL (Test)	43.80	56.20	2000
MIT-BIH (Train)	92.95	7.05	2000
MIT-BIH (Val)	90.20	9.80	2000
MIT-BIH (Test)	93.50	6.50	2000
Speech Commands (KWS) (Train)	–	–	12,786
Speech Commands (KWS) (Val)	–	–	500
Speech Commands (KWS) (Test)	–	–	500

Table 7: Window-level class distributions per split (limit = 2,000 per split).

Dataset	Split	Total	Class 0	Class 1
MIT-BIH	Train	2000	1859 (92.95%)	141 (7.05%)
	Val	2000	1804 (90.20%)	196 (9.80%)
	Test	2000	1870 (93.50%)	130 (6.50%)
Apnea-ECG	Train	2000	1005 (50.25%)	995 (49.75%)
	Val	2000	677 (33.85%)	1323 (66.15%)
	Test	2000	768 (38.40%)	1232 (61.60%)
PTB-XL	Train	2000	1064 (53.20%)	936 (46.80%)
	Val	2000	887 (44.35%)	1113 (55.65%)
	Test	2000	876 (43.80%)	1124 (56.20%)

A.2 Full per-model test metrics

Tables 8–10 report the best test-set metrics for each baseline model, alongside packed flash size. Two high-level patterns are consistent across datasets: (i) the largest model (`regularcnn1d`) provides an upper bound on achievable macro-F_1 but at prohibitive flash, and (ii) HYPERTINYPW closes most of this gap at the mid-budget elbow ($\sim$225 kB), while the very small models provide strong byte-efficiency but saturate earlier. On PTB-XL, HYPERTINYPW is effectively indistinguishable from the large model in macro-F_1 at $\sim$16% of the flash, demonstrating that storing all pointwise mixers is not necessary for near-peak performance. On Apnea-ECG, HYPERTINYPW retains most of the large-model macro-F_1 while maintaining strong AUC, whereas ultra-compact models show a larger AUC drop, consistent with limited channel-mixing capacity. On MIT-BIH, AUC remains high while macro-F_1 is lower across all models due to extreme imbalance; this reinforces the role of calibration (threshold choice) and suggests that improving imbalance-aware post-processing can yield additional gains at fixed flash.

Table 8: Apnea-ECG: best per-model *test* metrics (all baselines).

Model	Flash (kB)	Macro-F_1	Acc	BalAcc	AUC
regularcnn1d	1422.00	**0.7518**	0.7639	0.7484	0.8415
HYPERTINYPW (HyperTinyPW)	225.46	0.7172	0.7391	0.7164	0.8324
tinyseparablecnn	14.49	0.6660	0.6924	0.6688	0.6504
resnet1dsmall	62.49	0.6580	0.6786	0.6590	0.6660
tinyvaehead	10.16	0.6435	0.6605	0.6438	0.7358
hrvfeatnet	0.53	0.5004	0.5156	0.5022	0.4899

Table 9: PTB-XL: best per-model *test* metrics (all baselines).

Model	Flash (kB)	Macro-F_1	Acc	BalAcc	AUC
regularcnn1d	1422.00	**0.6293**	0.6315	0.6325	0.8814
HYPERTINYPW (HyperTinyPW)	225.46	0.6291	0.6310	0.6327	0.8760
resnet1dsmall	62.49	0.6225	0.6274	0.6233	0.8770
tinyseparablecnn	14.49	0.6174	0.6219	0.6184	0.8684
tinyvaehead	10.16	0.5938	0.5965	0.5962	0.8071
hrvfeatnet	0.53	0.5341	0.5364	0.5367	0.7173

Table 10: MIT-BIH: best per-model *test* metrics (all baselines).

Model	Flash (kB)	Macro-F_1	Acc	BalAcc	AUC
regularcnn1d	1422.00	**0.6293**	0.930	0.927	0.972
HYPERTINYPW (HyperTinyPW)	225.27	0.5673	0.902	0.562	0.962
resnet1dsmall	62.49	0.5450	0.865	0.540	0.945
tinyseparablecnn	14.49	0.5332	0.851	0.528	0.939
tinyvaehead	10.16	0.5218	0.839	0.520	0.932
hrvfeatnet	0.53	0.5004	0.828	0.502	0.920

A.3 Best results under flash budgets

Tables 11, 12, and 13 summarize the best-performing models under representative packed-flash budgets. The key takeaway is the *budget threshold* at which the preferred architecture changes: below ≈64 kB, compact separable/residual models dominate because the generator overhead cannot yet be amortized; once budgets reach ≈256 kB, HYPERTINYPW becomes the consistent winner across datasets, marking the onset of the mid-budget elbow where synthesizing channel mixers yields the largest accuracy gain per stored byte. This "winner switch" aligns with the Pareto-front shape in Fig. 3 and makes the practical deployment implication explicit: if an MCU budget is in the ∼200–250 kB range, HYPERTINYPW is the most accurate choice among the evaluated compressed alternatives.

Table 11: Apnea-ECG: best *test* metrics under flash budgets (packed kB).

Budget	Model	Flash (kB)	Macro-F_1	BalAcc	AUC
≤32	tinyseparablecnn	14.49	0.6660	0.6688	0.6504
≤64	tinyseparablecnn	14.49	0.6660	0.6688	0.6504
≤128	tinyseparablecnn	14.49	0.6660	0.6688	0.6504
≤256	HYPERTINYPW	225.46	**0.7172**	**0.7164**	**0.8324**

Table 12: PTB-XL: best *test* metrics under flash budgets (packed kB).

Budget	Model	Flash (kB)	Macro-F_1	BalAcc	AUC
≤32	tinyseparablecnn	14.49	0.6174	0.6184	0.8684
≤64	resnet1dsmall	62.49	0.6225	0.6233	0.8770
≤128	resnet1dsmall	62.49	0.6225	0.6233	0.8770
≤256	HYPERTINYPW	225.46	**0.6291**	**0.6327**	**0.8760**

Table 13: MIT-BIH: best *test* metrics under flash budgets (packed kB).

Budget	Model	Flash (kB)	Macro-F_1	BalAcc	AUC
≤32	tinyvaehead	10.16	0.5218	0.520	0.932
≤64	resnet1dsmall	62.49	0.5450	0.540	0.945
≤128	resnet1dsmall	62.49	0.5450	0.540	0.945
≤256	HYPERTINYPW	225.27	**0.5673**	**0.562**	**0.962**

A.4 Ablations and hardware-oriented metrics

Table 14 summarizes key HYPERTINYPW ablations across datasets and highlights that the same generator-based parameterization can be tuned by bit-width and (optionally) knowledge distillation while staying within a similar packed-flash envelope. In the current snapshot, KD effects are dataset-dependent: on MIT-BIH, KD has a small impact at fixed flash, while on Apnea-ECG the KD variant underperforms, suggesting that calibration/imbalance effects can dominate over representation limits in some settings. Tables 15 and 16 report steady-state latency and energy after one-shot synthesis (cached). These results emphasize that flash and latency/energy are not perfectly coupled: operator mix and tensor shapes can dominate (e.g., the unusually high PTB-XL `resnet1dsmall` latency in this proxy model), motivating joint reporting of flash and runtime costs. Finally, Table 17 contextualizes accuracy retention under compression: HYPERTINYPW achieves the strongest macro-F_1 retention among compressed models at the elbow while delivering $6.31\times$ flash compression relative to the large CNN.

Table 14: Unified ablation summary for HYPERTINYPW (HYPERTINYPW). MIT-BIH includes the full variant sweep; Apnea-ECG and PTB-XL rows report current best (no sweep yet). RAW branch; validation-tuned $t^\star$; k=5 median smoothing.

Dataset	d_z, d_h	Bits	KD	Macro-F_1	Acc	BalAcc	AUC	Flash (kB)
MIT-BIH	4,12	8	off	0.5650	0.8947	0.5617	0.9618	225.27
MIT-BIH	4,12	6	off	0.5485	0.8356	0.5775	0.9430	225.27
MIT-BIH	4,12	8	on	0.5641	0.8874	0.5649	0.9570	225.27
Apnea-ECG	6,16	8	off	0.7172	0.7391	0.7164	0.8324	225.46
Apnea-ECG	4,12	6	off	0.7024	0.7110	0.7007	0.7546	225.27
Apnea-ECG	6,16	8	on	0.5447	0.6377	0.5933	0.8080	225.46
PTB-XL	4,12	8	off	0.6200	0.6219	0.6237	0.8680	225.27
PTB-XL	6,16	6	off	0.6291	0.6310	0.6327	0.8760	225.46
PTB-XL	4,12	8	on	0.5982	0.5983	0.6115	0.8676	225.27

Table 15: Apnea-ECG: steady-state latency and energy after one-shot synthesis (cached).

Model	Flash (kB)	Chosen Macro-F$_1$	Latency (ms)	Energy (mJ)
hrvfeatnet	0.53	0.5004	0.799	9.6e-08
tinyvaehead	10.16	0.6435	0.563	0.00857827
tinyseparablecnn	14.49	0.6660	0.831	0.0754276
resnet1dsmall	62.49	0.6580	1.963	0.0517667
HyperTinyPW	225.46	0.7172	2.383	0.0147649
regularcnn1d	1422.00	0.7518	3.717	7.12494

Table 16: PTB-XL: steady-state latency and energy after one-shot synthesis (cached).

Model	Flash (kB)	Chosen Macro-F$_1$	Latency (ms)	Energy (mJ)
hrvfeatnet	0.53	0.5341	0.814	9.6e-08
tinyvaehead	10.16	0.5938	0.465	0.00857827
tinyseparablecnn	14.49	0.6174	0.900	0.0754276
resnet1dsmall	62.49	0.6225	96.756	0.0517667
HyperTinyPW	225.46	0.6291	7.024	0.0147649
regularcnn1d	1422.00	0.6293	4.005	7.12494

Table 17: Compression vs. `regularcnn1d` (flash=1422.00 kB). Reported are packed flash, compression factor (×), flash reduction (%), and macro-F_1 retention (%) on Apnea-ECG and PTB-XL.

Model	Flash (kB)	Compress (×)	Flash ↓ (%)	Apnea F_1 retain (%)	PTB F_1 retain (%)
HYPERTINYPW	225.46	**6.31**	**84.15**	**95.40**	**99.97**
resnet1dsmall	62.49	22.76	95.61	87.52	98.92
tinyseparablecnn	14.49	98.14	98.98	88.59	98.11
tinyvaehead	10.16	139.96	99.29	85.59	94.36
hrvfeatnet	0.53	2683.02	99.96	66.56	84.87

B ADDITIONAL EXPERIMENTS AND ANALYSES

This appendix provides fuller protocols and diagnostics for the additional evaluations summarized in §6.5. The goal is to document training/splitting choices and to explain failure modes observed under extreme compression, without changing the main-paper narrative.

B.1 Keyword spotting on Speech Commands

Task. We evaluate HYPERTINYPW on Google Speech Commands v0.02 (Warden, 2018) in the standard 12-class setup (10 keywords + `unknown` + `silence`), widely used in TinyML benchmarking (Warden & Situnayake, 2019).

Features. Each 1 s, 16 kHz clip is converted to MFCC features (40 coefficients $\times$ 101 frames) using a standard MFCC pipeline (Davis & Mermelstein, 1980), yielding a 40×101 time–frequency representation.

Model and training. We train a 234,853-parameter HYPERTINYPW instance for 20 epochs (Adam, $\mathrm{lr} = 10^{-3}$, batch size 64, cross-entropy). We report best validation accuracy and final test accuracy.

Outcome. HYPERTINYPW reaches 96.2% test accuracy with best validation accuracy 98.2% (epoch 12), with mild late-stage overfitting. The FP32 parameter footprint is 917.39 kB; the packed parameter footprint upper-bounds at ≈235 kB under an all-`INT8` packing assumption (parameters only; feature extraction/runtime not included) (David et al., 2021).

Table 18: Speech Commands v0.02 keyword spotting results (12 classes).

Model	Params	FP32 size	Best Val Acc	Test Acc
HYPERTINYPW	234,853	917.39 kB	98.2% (ep12)	96.2%

B.2 Ternary quantization baseline on PTB-XL

Setup. To contextualize the extreme size regime, we implement a ternary-weight baseline following standard ternary quantization (Li et al., 2016; Zhu et al., 2017). We evaluate on PTB-XL with a fold split (folds 1–8 train, 9 validation, 10 test). Both HYPERTINYPW and ternary are trained for 20 epochs with Adam ($\mathrm{lr} = 10^{-3}$), batch size 32, weight decay 10^{-5}, and ReduceLROnPlateau (factor 0.5, patience 3). To mitigate imbalance we use weighted cross-entropy (class weights $[1.16,\ 0.88]$).

Size accounting. Ternary weights use 2 bits/weight plus learned per-output-channel scale factors (one FP32 scale per output channel), yielding 6.70 kB packed size in our implementation.

Diagnostics. Although ternary achieves very small flash, it collapses toward predicting the positive class: negative-class accuracy drops sharply while positive-class accuracy remains high, producing substantially lower balanced accuracy.

	Class 0 (normal)	Class 1 (abnormal)
HYPERTINYPW per-class acc.	83.93%	74.80%
Ternary per-class acc.	13.13%	97.51%

	HYPERTINYPW		Ternary	
	Pred 0	Pred 1	Pred 0	Pred 1
True 0	799	153	125	827
True 1	314	932	31	1215

B.3 Why we do not include architecture search

We prototyped a search variant where widths/kernel sizes/depth are varied while still using the generator-based pointwise parameterization. A practical issue is that the PW-head output tensor shape can dominate at very small target models, so some candidates *inflate* rather than compress because head/generator parameters become a large fraction of the network. This makes strict budget-constrained search ill-conditioned and confounds comparisons; we therefore focus on fixed, budget-aligned backbones and report scaling behavior directly in Table 5.

B.4 Synthesis profiling status

One-shot synthesis happens once at boot (or lazily on first use); steady-state inference uses standard CMSIS-NN/TFLM kernels. We began profiling synthesis time versus steady-state inference time; initial instrumentation detected mismatches in layer naming for synthesized pointwise layers. Completing cycle-accurate synthesis-time measurement on target MCUs is left as future work; the steady-state latency/energy results reported in Appendix A are unaffected.