

Article

A Hardware-Aware Federated Meta-Learning Framework for Intraday Return Prediction Under Data Scarcity and Edge Constraints

Zhe Wen ¹, Xin Cheng ², Ruixin Xue ², Jinao Ye ¹, Zhongfeng Wang ¹ and Meiqi Wang ^{1,*}¹ School of Integrated Circuits, Sun Yat-sen University, Shenzhen 518107, China; wenzh9@mail2.sysu.edu.cn (Z.W.); yejao@mail2.sysu.edu.cn (J.Y.); wangzf83@mail.sysu.edu.cn (Z.W.)² School of Electronic Science and Engineering, Nanjing University, Nanjing 210023, China; chengx@smail.nju.edu.cn (X.C.)

* Correspondence: wangmq53@mail.sysu.edu.cn; Tel.: +86-18260099732

Abstract

Although deep learning has achieved remarkable success in time-series prediction, intraday algorithmic trading is characterized by frequent regime shifts (concept drift), which can rapidly render models trained on historical data obsolete in real applications. This motivates on-device adaptation at edge trading terminals. However, practical deployment is constrained by a tripartite bottleneck: real-time samples are scarce, hardware resources on edge are limited, and communication overhead between cloud and edge must be kept low to satisfy stringent latency requirements. To address these challenges, we develop a hardware-aware edge learning framework that combines federated learning (FL) and meta-learning to enable rapid few-shot personalization without exposing local data. Importantly, the framework incorporates our proposed Sleep Node Algorithm (SNA), which turns the “FL + meta-learning” combination into a practical and efficient edge solution. Specifically, SNA dynamically deactivates “inertial” (insensitive) network components during adaptation: it provides a structural regularizer that stabilizes few-shot updates and mitigates overfitting under concept drift, while inducing sparsity that reduces both on-device computation and cloud-edge communication. To efficiently leverage these unstructured zero nodes introduced by SNA, we further design a dedicated accelerator, EPAST (Energy-efficient Pipelined Accelerator for Sparse Training). EPAST adopts a heterogeneous architecture and introduces a dedicated Backward Pipeline (BPIP) dataflow that overlaps backpropagation stages, thereby improving hardware utilization under irregular sparse workloads. Experimental results demonstrate that our system consistently outperforms strong baselines, including DQN, GARCH-XGBoost, and LRU, in terms of Pearson IC. A 55 nm CMOS ASIC implementation further validates robust learning under an extreme 5-shot setting (IC = 0.1176), achieving an end-to-end training speed-up of 11.35× and an energy efficiency of 45.78 TOPS/W.



Academic Editor: Arkadiusz Gola

Received: 20 January 2026

Revised: 5 February 2026

Accepted: 16 February 2026

Published: 27 February 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

Keywords: training accelerator; pipeline; on-device learning; artificial intelligence; few-shot learning; sparse training; intraday return prediction; factor modeling

1. Introduction

Deep Neural Networks (DNNs) have become a standard choice for financial time-series prediction [1,2]. In Intraday Return Prediction, however, their performance is often fragile because market dynamics are non-stationary [3]. This mismatch is commonly

framed as temporal distribution shift: a model may look strong in historical backtests but deteriorate in live trading as the feature–target relationship drifts over time [4]. The difficulty is amplified during structural breaks or “black swan” events, where the most relevant historical samples are scarce or effectively absent [5]. In practice, this creates a tension: modern deep models benefit from large datasets, yet the regimes that matter most often arrive with only a handful of usable samples.

These realities motivate on-device learning for local data adaptation. In practice, frequent cloud retraining is constrained not only by latency but also by data governance. On platforms such as JoinQuant or BigQuant, proprietary factors are treated as core intellectual property and are not readily exportable [6]. A practical system therefore needs the ability to update locally from the newest collected data, sometimes within a narrow window (e.g., the most recent 5 trading days), while preserving the global knowledge learned from broader histories. However, practical deployment faces a tripartite bottleneck:

- **Data Scarcity:** Local adaptation must succeed with very few samples as a new regime emerges. Few-Shot Learning (FSL) provides useful principles [5], yet most FSL pipelines assume centralized compute and do not directly target edge training budgets.
- **Resource Constraints:** Trading terminals operate under strict limits on power, memory, and throughput [7–9]. Standard on-device fine-tuning can be prohibitively expensive [10–12], and the resulting overhead conflicts with the tight latency requirements typical of Intraday Return Prediction workloads.
- **Privacy and Communication Efficiency:** While Federated Learning (FL) can keep raw factor data local, naïvely synchronizing dense model updates is expensive. In latency-sensitive environments, the communication cost of frequent, high-volume exchanges becomes a practical bottleneck.

Most existing methods address only part of this problem. Simple fine-tuning can overfit quickly under scarce local data and may overwrite the knowledge encoded in a pre-trained initialization [13]. Sparsity is a natural direction to reduce cost, but generic pruning strategies often trade away the very capacity needed for few-shot generalization, and they provide limited control over communication volume. On the hardware side, dynamic and irregular sparsity patterns can be difficult to exploit efficiently; even when arithmetic counts drop on paper, real speedups may be limited by serialization effects such as the Backward Locking (BL) problem [14].

For completeness, Figure 1 provides a high-level conceptual illustration of a typical on-device training scenario with privacy-sensitive local data. The figure is intended as an auxiliary visual reference to contextualize the deployment setting, while the research objectives, algorithmic design, and technical contributions are primarily conveyed through the accompanying text.

To address these constraints jointly, we propose a hardware-aware federated meta-learning framework that co-designs the learning algorithm and the execution substrate. The key algorithmic component is the Sleep Node Algorithm (SNA), a sparse training method for local adaptation. Instead of pruning weights solely based on importance, SNA distinguishes two forms of redundancy during fine-tuning: Sparse Nodes, which are removed, and Lazy Nodes, which remain important but exhibit consistently negligible updates under the meta-learned initialization. By putting Lazy Nodes to “sleep” (i.e., freezing them), SNA (Algorithm 1) reduces the backward-pass workload and produces ultra-sparse gradient masks. This design simultaneously mitigates overfitting in few-shot adaptation and reduces the bandwidth required for federated synchronization.

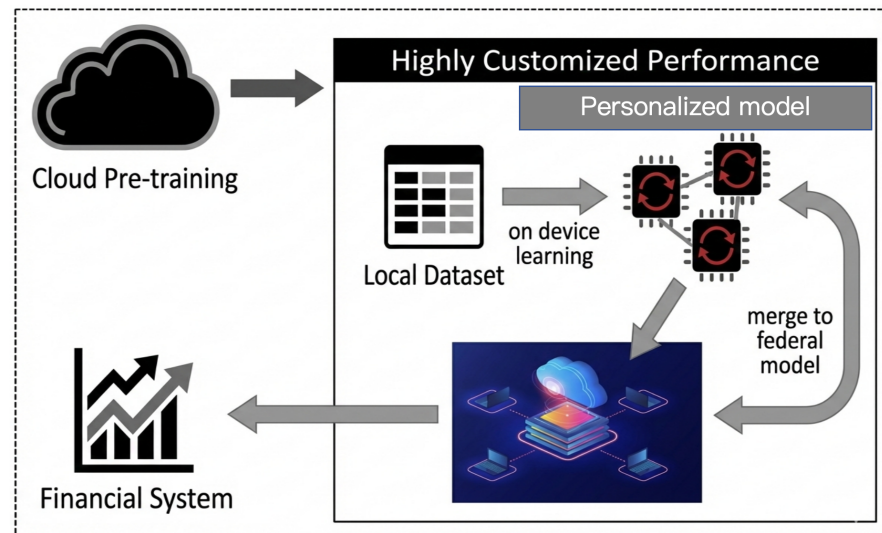


Figure 1. On-device training with privacy-sensitive local dataset.

Algorithm 1 Sleep Node Algorithm (SNA) for Local Few-shot Adaptation

Input: Meta-initialized weights $W^{(0)}$; local data $\mathcal{D}_{\text{loc}}$; learning rate α ; total steps T ; probe steps T_p ; sparse ratio ρ_s ; lazy ratio ρ_l (or threshold τ_l).

Output: Adapted weights $W^{(T)}$; trainable mask $\mathcal{M}_g \in \{0, 1\}^{|W|}$.

Init: $W \leftarrow W^{(0)}$; $G \leftarrow 0$

(1) Sparse mask: $I \leftarrow \text{Importance}(W)$

$\mathcal{M}_s \leftarrow \text{KeepTop}(I, 1 - \rho_s)$

$\triangleright 1 = \text{kept}, 0 = \text{removed}$

(2) Probe for laziness:

for $t = 1$ to T_p **do** $(x, y) \leftarrow \text{Sample}(\mathcal{D}_{\text{loc}})$ $g \leftarrow \nabla_W \mathcal{L}(W; x, y)$ $G \leftarrow G + |g|$ $\triangleright$ accumulate sensitivity proxy

end for

(3) Lazy mask: $S \leftarrow G$

if ratio-based **then**

$\mathcal{M}_l \leftarrow \text{KeepTop}(S, 1 - \rho_l)$

else

$\mathcal{M}_l[i] \leftarrow \mathbb{I}(S[i] \geq \tau_l)$

end if

(4) Selective adaptation: $\mathcal{M}_g \leftarrow \mathcal{M}_s \odot \mathcal{M}_l$

for $t = T_p + 1$ to T **do** $(x, y) \leftarrow \text{Sample}(\mathcal{D}_{\text{loc}})$ $g \leftarrow \nabla_W \mathcal{L}(W; x, y)$ $W \leftarrow W - \alpha (\mathcal{M}_g \odot g)$

end for

return $W, \mathcal{M}_g$

However, algorithmic sparsity is insufficient to ensure system-level gains. The hybrid and largely unstructured sparsity induced by SNA poses significant challenges for general-purpose processors, especially in the backward pass. We therefore introduce EPAST (Energy-efficient Pipelined Accelerator for Sparse Training), a dedicated accelerator that executes SNA with high hardware utilization. EPAST adopts a heterogeneous design featuring a Backward Pipeline (BPIP) dataflow and a Hybrid Workload Allocation (HWA) scheme, which together reduce serialization and improve throughput under sparse training [15].

Novelty and Contributions. This study differs from existing works that typically address sparse training algorithms or hardware accelerators as separate problems. Instead, it presents an integrated hardware-aware federated meta-learning framework tailored to intraday quantitative finance, where data scarcity, concept drift, and edge deployment constraints coexist. The main novel contributions of this work are summarized as follows:

- Sensitivity-aware sparse adaptation via the Sleep Node Algorithm (SNA). Unlike conventional magnitude-based pruning or static sparsity strategies, SNA exploits a meta-learned initialization to identify parameters that consistently exhibit low sensitivity during few-shot adaptation. These parameters, referred to as lazy nodes, are frozen rather than removed, which stabilizes local adaptation under severe data scarcity while yielding highly sparse gradient updates. This sensitivity-aware freezing mechanism is fundamentally different from existing sparse training approaches.
- Joint exploitation of multiple sources of training sparsity. In contrast to prior methods that typically exploit only one or a limited subset of sparsity sources, this work simultaneously leverages sparsity in weights, weight updates, back-propagated errors, and activations. This unified formulation enables substantial reductions in computation, memory access, and communication overhead during training, without degrading predictive performance.
- A hardware–software co-designed training accelerator for sparse adaptation. Different from accelerators primarily optimized for inference or dense back-propagation, the proposed EPAST architecture is explicitly designed to support sparse and irregular training workloads. Through a backward pipeline (BPIP) dataflow and a hybrid workload allocation strategy, EPAST effectively translates algorithm-level sparsity into practical latency and energy efficiency gains under realistic edge hardware constraints.
- A federated meta-learning framework aligned with intraday trading practice. Unlike standard federated or meta-learning approaches evaluated under static or IID assumptions, the proposed framework targets near-live intraday trading scenarios. It combines few-shot on-device adaptation, non-IID federated learning, and hardware-aware execution into an end-to-end system, bridging financial modeling requirements with the constraints of edge deployment.

2. Related Works and Preliminaries

2.1. Deep Learning in Quantitative Finance and the Need for Adaptation

Deep learning has become a core toolkit in quantitative investment. Models range from classical LSTMs to recent transformer-based designs such as MCI-GRU [16] and Hybrid LLMs [17], leveraging their capacity to capture complex inter-stock dependencies and sentiment-driven signals. A major barrier to robust deployment is Temporal Distribution Shift (also known as “Concept Drift”) [4]. Because market regimes evolve, models trained on historical data can degrade quickly once deployed in live environments. As a response, Wood et al. [5] showed that Few-Shot Learning (FSL) can support regime adaptation using only a small number of samples.

To make such adaptation operational, we adopt a meta-learning (“learning to learn”) perspective. In a typical formulation, a meta-learner trains a base learner f_θ over a distribution of tasks so that it can generalize rapidly to a new task with limited data. A task is commonly organized in an N -way K -shot format, consisting of N classes with K support samples and Q query samples. The objective is to predict the $N \times Q$ queries based on the $N \times K$ supports. By minimizing the aggregated losses on query sets across tasks, the base learner f_θ acquires a generic initialization that is highly responsive to distributional changes.

While these FSL/meta-learning methods address the algorithmic requirement for rapid adaptation, most implementations implicitly assume centralized, high-performance training infrastructure. In edge trading scenarios, this assumption often fails. In particular, the practical bottleneck is the “Impossible Trinity” at the terminal: (i) privacy constraints that prevent uploading proprietary factor data, (ii) data scarcity that destabilizes local fine-tuning, and (iii) tight power envelopes that limit on-device training. This gap motivates solutions that are both hardware-aware and robust under few-shot adaptation.

2.2. Hardware Acceleration Challenges and Training Dynamics

To reduce the cost of on-device training, prior work has extensively explored sparsity-oriented optimizations, including pruning and quantization [13,18–23]. For example, GANPU [22] applies dual zero-skipping on inputs and outputs. In practice, however, the resulting control-flow and scheduling overhead can be substantial, which may reduce hardware utilization and limit end-to-end speedups. Procrustes [23] adopts one-sided sparsity, yet generic accelerators typically lack mechanisms tailored to exploit specific redundancy patterns—including the “Sleep Node” behavior we identify in this work.

The Backward Locking Bottleneck in CNN Training

Beyond raw arithmetic cost, a key limitation in training accelerators is the difficulty of pipelining the training loop efficiently. This can be seen by examining the dependency structure of CNN training. As illustrated in Figure 2, training consists of three stages:

1. Feed-Forward (FF): compute activations

$$a^l = \sigma(z^l) = \sigma(W^l * a^{l-1} + b^l). \quad (1)$$

2. Backward Propagation (BP): propagate errors using rotated weights

$$\delta^l = \text{rot180}(W^{l+1}) * \delta^{l+1} \cdot \sigma'(z^l). \quad (2)$$

3. Weight Update (WU): compute gradients and update weights

$$W^l = W^l - \alpha \sum_i^m a^{l-1} * \delta^l, \quad (3)$$

where a^l and z^l denote the activation and pre-activation at layer l , W^l and b^l are the convolution weights and bias, $*$ denotes convolution, $\sigma(\cdot)$ is the activation function with derivative $\sigma'(\cdot)$, δ^l is the backpropagated error signal, $\text{rot180}(\cdot)$ rotates a kernel by 180° , m is the mini-batch size, and α is the learning rate.

The dependencies in Equations (2) and (3) induce Backward Locking (BL): the FF stage of the next batch cannot start until the BP stage of the current batch has completed across all layers. Prior work such as DF-LNPU [15] explored pipelining, but it was effectively limited to updating only the final fully connected layers; extending their skipping scheme to convolutional layers caused significant accuracy degradation. Parallel FPGA designs (e.g., [24,25]) face related constraints and tend to be practical only for shallow networks.

The proposed algorithm-hardware co-optimized framework in this work aims to remove these architectural bottlenecks. By introducing a Backward Pipeline (BPIP) dataflow that explicitly decouples the WU stage from the FF/BP loop, we enable full-network training on edge devices (including CNN backbones) while meeting the combined requirements of few-shot adaptation accuracy and ultra-low latency.

Differentiated Advantages over Recent Hardware-Aware On-Device Learning Methods. We further contrast our framework with several representative hardware-aware on-device training studies: (1) Paissan et al. propose structured sparse back-propagation to reduce the training cost for lightweight on-device continual learning on microcontroller units (MCUs) [26]; (2) Zhao et al. introduce a quantized back-propagation-free (zeroth-order) training scheme on MCUs, prioritizing memory efficiency and ease of deployment [27]; and (3) Deutel et al. investigate fully-quantized on-device training on Cortex-M MCUs with dynamic partial gradient updates, highlighting trade-offs among accuracy, memory, energy, and latency on real hardware [28]. In contrast, our work targets intraday quantitative finance with severe concept drift and data scarcity. Its differentiated

advantages lie in the unified co-design of (i) federated meta-learning for few-shot personalization, (ii) sensitivity-aware sparse adaptation via the Sleep Node Algorithm (SNA), which yields ultra-sparse gradient masks for both stable adaptation and communication reduction, and (iii) the EPAST accelerator, which translates multi-source training sparsity into practical latency and energy benefits under edge constraints.

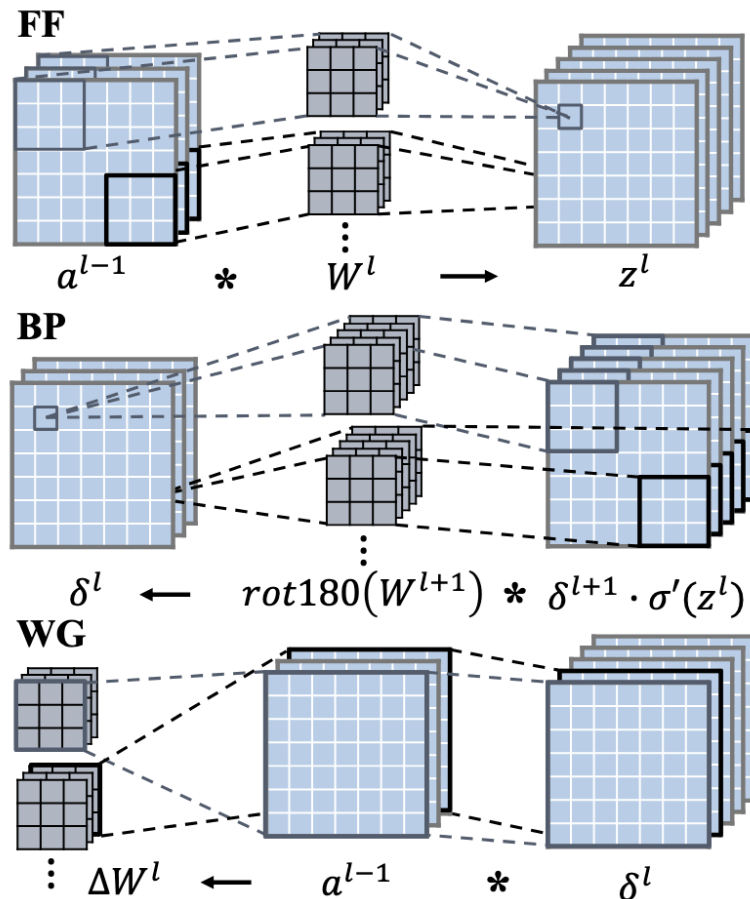


Figure 2. The three training stages. (The asterisk * denotes the convolutional coding process.)

3. The Hardware-Aware Federated Learning Framework

To resolve the “Impossible Trinity” of edge-based Intraday Return Prediction—balancing data scarcity, privacy constraints, and hardware limitations—we propose a cohesive framework that co-optimizes the learning algorithm and the hardware architecture.

3.1. Bridging the Gap: The Need for Hardware-Aware Adaptation

A fundamental bottleneck in on-device learning is the conflict between model capacity and data availability. While fine-tuning pre-trained models is a standard approach to reduce hardware costs [29], directly applying it to edge environments fails for two reasons. First, edge devices frequently encounter unseen market regimes (concept drift) that were absent in the pre-training data, leading to severe overfitting when local samples are scarce. Second, the necessary quantization and sparsity techniques used to fit models onto edge hardware often degrade the initial accuracy and slow down convergence [13], creating a vicious cycle where more training epochs are needed exactly when resources are tight.

To break this deadlock, we propose the Hardware-Aware Training (HAT) framework (Figure 3). The core philosophy is to prepare the model *before* it reaches the edge, ensuring it is not only accurate but also “adaptation-ready.”

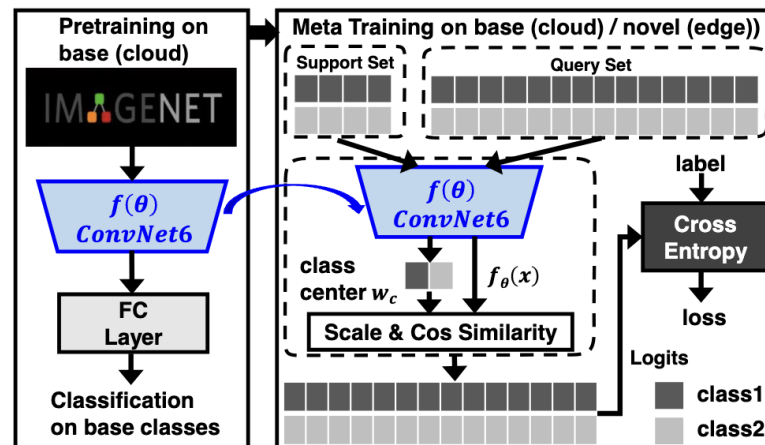


Figure 3. Overview of the proposed framework and its correspondence to key limitations in conventional data-driven learning systems under intraday edge trading constraints.

As detailed in Algorithm 2, our process introduces a critical Meta-Pre-Training (MPT) stage between standard pre-training and edge fine-tuning.

1. Standard Pre-training: We first train a dense model f_θ on the cloud to learn universal market features.
2. Meta-Pre-Training (MPT): We then prune the model to a sparse version f_θ^P and explicitly train it for generalization using meta-learning objectives. This ensures the model learns *how to adapt* quickly.
3. SNA-based Fine-tuning: Finally, the model is quantized ($f_\theta^{P,Q}$) and deployed to the edge, where it is fine-tuned using our proposed Sleep Node Algorithm (SNA).

Algorithm 2 The Hardware-aware Training Framework

Require: Cloud dataset D_{cloud} , edge dataset D_{edge} , feature extractor f_θ , classifier c , epochs T_1, T_2, T_3 , loss L

```

1: function FSL( $D, f_\theta$ )
2:   Sample  $M$  few-shot tasks from  $D$ 
3:   for  $task \in [1, M]$  do
4:     get loss  $l$  of  $task$  using Equations (4) and (5)
5:     update  $\theta$  with  $l$ 
6:   end for
7: end function
8: 1st stage: pre-training
9: for  $epoch \in [1, T_1]$  do
10:  for  $(x, y) \in D_{cloud}$  do
11:    Update  $c$  and  $\theta$  with  $L(c(f_\theta(x)), y)$ 
12:  end for
13: end for
14: 2nd stage: meta-pre-training
15: Prune  $f_\theta$  to get  $f_\theta^P$ 
16: for  $epoch \in [1, T_2]$  do
17:   FSL( $D_{cloud}, f_\theta^P$ )
18: end for
19: 3rd stage: fine-tuning
20: Quantize  $f_\theta^P$  on  $D_{edge}$  to get  $f_\theta^{P,Q}$ 
21: for  $epoch \in [1, T_3]$  do
22:   FSL( $D_{edge}, f_\theta^{P,Q}$ )
23: end for

```

3.2. The Meta-Pre-Training Strategy

The MPT stage is designed to solve the “cold start” problem on the edge. If we directly fine-tune a standard pre-trained model on a new 5-day market window, the model tends to memorize the noise, leading to catastrophic overfitting.

To prevent this, we split the pre-training process. After the initial dense training, we remove the final classifier c and enter the MPT phase. Here, the feature extractor f_θ is trained under a “N-way K-shot” simulation that mimics the data-scarce conditions of the edge [30,31]. We adopt a metric-based classification approach:

$$w_c = \frac{1}{|S_c|} \sum_{x \in S_c} f_\theta(x), \quad (4)$$

$$p(y = c \mid x') = \frac{\exp(\langle f_\theta(x'), w_c \rangle)}{\sum_{c'} \exp(\langle f_\theta(x'), w_{c'} \rangle)}, \quad (5)$$

where S_c represents the support samples for class c , and w_c is the class prototype. By optimizing f_θ to maximize the probability of query samples x' based on these prototypes, the model learns a robust initialization that is resistant to overfitting during subsequent edge adaptation.

Task Alignment: Numerical Interface for Stable Evaluation

Conventionally, models are trained to minimize point-wise regression errors (e.g., MSE). However, such objectives often misalign with the downstream ranking metric, IC, due to the non-differentiable nature of sorting operations [32,33]. To bridge this gap, we adopt a quantile-based tri-classification strategy as a numerical interface to generate standardized scoring signals. In high-resolution intraday regimes, raw return distributions often exhibit heavy tails and a low signal-to-noise ratio, which can introduce gradient volatility during the few-shot adaptation process. Strictly as an engineering stabilization measure rather than a claimed algorithmic innovation, we map these volatile returns into a bounded label space based on dynamic cross-sectional quantiles [34,35]: For fair comparison, this quantile-based tri-classification interface (and the same score used for IC/ranking evaluation) is applied consistently to all methods in our experiments, including full fine-tuning, prune-only/sparse-only baselines, and our proposed SNA-based adaptation.

$$y_i^{(t)} = \begin{cases} 2 \text{ (Positive)}, & \text{if } r_i^{(t)} \geq q_{0.9}^{(t)} \\ 0 \text{ (Negative)}, & \text{if } r_i^{(t)} \leq q_{0.1}^{(t)} \\ 1 \text{ (Neutral)}, & \text{otherwise} \end{cases} \quad (6)$$

where $r_i^{(t)}$ denotes the realized intraday return of stock i , and $q_{0.9}^{(t)}$ ($q_{0.1}^{(t)}$) is the cross-sectional 90th (10th) percentile of $\{r_i^{(t)}\}_i$ at time t .

This discretization functions as a pre-processing filter, ensuring that the gradient source remains numerically consistent across diverse market regimes.

During the deployment phase, we recover a continuous ranking signal $s(x)$ through the probabilistic spread:

$$s(x) = P(y = 2 \mid x) - P(y = 0 \mid x) \quad (7)$$

This score $s(x)$ serves as a monotonic proxy for directional conviction, providing a stable input for IC calculation.

Hardware note. We use the tri-classification interface mainly for numerical stability under 8-bit fixed-point training on our 55 nm ASIC. This is a fixed evaluation/training

interface; the performance gains reported later are attributed to SNA and the hardware–software co-design rather than the labeling scheme.

3.3. Federated Meta-Learning with SNA Adaptation

With the meta-pre-trained model distributed to the edge, the challenge shifts to continuous adaptation while preserving privacy. Standard Federated Learning (FL) is ill-suited here due to the high communication cost of transmitting full gradient updates.

We address this by integrating our Sleep Node Algorithm (SNA) into a Federated Meta-Learning loop (inspired by MAML [36]). This creates a privacy-preserving cycle:

- **Local SNA Loop:** The client k adapts the model to its private task $\mathcal{T}_i$ (comprising a Support Set S_i and Query Set Q_i). Critically, instead of a dense update, we apply a sparse mask derived from SNA:

$$\theta'_i = \theta - \alpha \cdot \text{Mask}_{\text{SNA}}(\nabla_{\theta} \mathcal{L}_{S_i}(\theta)) \quad (8)$$

- **Sparse Aggregation:** When uploading updates to the server, only the non-zero values masked by SNA are transmitted:

$$\Delta\theta_k = \text{Mask}_{\text{SNA}}(\theta_{k,\text{updated}} - \theta_t) \quad (9)$$

The server then aggregates these sparse updates:

$$\theta_{t+1} \leftarrow \theta_t + \eta \sum_{k=1}^K \frac{|D_k|}{|D|} \Delta\theta_k \quad (10)$$

This approach simultaneously solves two problems: it drastically reduces communication bandwidth (solving the privacy/efficiency constraint) and acts as a regularizer during local training (solving the data scarcity constraint).

3.4. The Sleep Node Algorithm (SNA): Structural Regularization via Lazy Nodes

The success of the framework above hinges on the Sleep Node Algorithm (SNA). While prior works have explored sparsity to reduce computation [23,37,38], they typically treat sparsity as a purely arithmetic optimization—simply removing small weights to save FLOPs. We argue that for few-shot adaptation, sparsity must also serve as a regularizer.

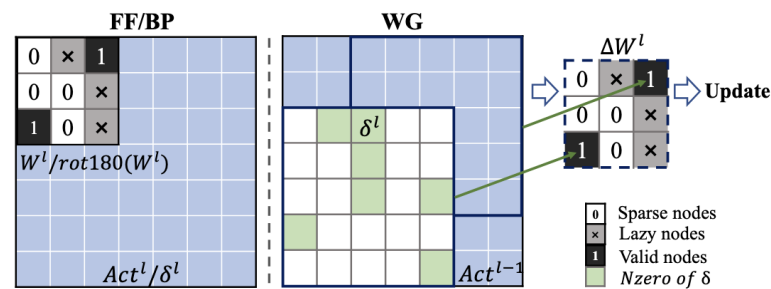
SNA distinguishes between three types of nodes, as illustrated in Figure 4:

1. **Active Nodes:** The trainable parameters that are updated during few-shot adaptation.
2. **Sparse Nodes (marked '0'):** Unimportant connections pruned permanently to save memory.
3. **Lazy Nodes (marked '×'):** This is our novel contribution. These are weights that are structurally necessary for the forward pass but statistically stable enough to be frozen during the backward pass.

The $10\times$ speedup achieved by SNA (compared to $5\times$ for Top- k) in Table 1 is a direct consequence of its Zero-Indexing Overhead protocol. In dynamic sparsification methods like Top- k , the set of updated parameters changes every round, necessitating the transmission of both weight values and their coordinate indices. In an 8-bit quantized system, these indices can double the communication payload. Conversely, SNA utilizes a deterministic mask anchored by the meta-initialization. This allows the server and client to remain implicitly synchronized, enabling a “pure value-stream” transmission that completely eliminates the need for indexing metadata.

Table 1. Bit-Accurate Communication Cost Comparison under Different Sparsification Schemes (10% Active Ratio, 8-bit Values).

Method	Bits (Factor of d)	Speedup vs. Dense	Efficiency vs. SNA
Dense (FedAvg)	$8.000 d$	$1.00\times$	10.0%
Top- k + Raw Indices	$4.000 d$	$2.00\times$	20.0%
Top- k + Block Encoding	$1.800 d$	$\sim 4.44\times$	44.4%
Top- k + Delta Encoding	$1.200 d$	$\sim 6.67\times$	66.7%
Top- k + Entropy Coding (Limit)	$\sim 1.132 d$	$\sim 7.07\times$	70.7%
SNA (Ours)	$0.800 d$	$10.00\times$	100.0%

**Figure 4.** The illustration of the proposed sleep node algorithm. During the FF/BP stage, the weights in the sparse nodes denoted by ‘0’ can be skipped. In the WG computation, both sleep nodes, including sparse nodes denoted by ‘0’ and lazy nodes denoted by ‘x’, can be skipped. We also highlight the non-zero δ in the figure to indicate how we utilize the error sparsity during WG computation, which will be introduced in detail in Section 5.3.

In the proposed framework, sparsity arises at different stages of the training and deployment pipeline and serves distinct purposes. After pruning and meta-pre-training, a fraction of parameters are permanently removed, resulting in a fixed weight sparsity that reduces model size and memory footprint. During few-shot adaptation, the Sleep Node Algorithm further introduces lazy nodes, which are parameters that remain active in the forward pass but are frozen during backward updates due to their low sensitivity under the meta-initialized model. Together, these two forms of weight-level sparsity determine the effective degrees of freedom during adaptation and are the primary source of the regularization effect that stabilizes few-shot learning. In contrast, error sparsity is a numerical property that emerges during quantized training: when back-propagated error signals are represented under FXP8 arithmetic, a large portion of the error mass—already concentrated near zero for a meta-initialized model—is quantized to exact zeros. We quantify this effect as

$$spe = \frac{N_{\text{zero}}(\delta_q)}{N_{\text{tot}}(\delta_q)}, \quad (11)$$

where δ_q denotes the quantized back-propagated error tensor. This error sparsity does not influence which parameters are selected as lazy nodes and does not affect the convergence behavior of SNA. Instead, it is exploited purely at the system level to enable zero-skipping during weight-gradient computation and improve execution efficiency. Since the degree of error sparsity may vary with FXP8 scale or threshold choices, the hardware runtime controller dynamically adapts scheduling based on observed sparsity statistics rather than assuming a fixed sparsity level. Moreover, under higher-precision arithmetic where quantization-induced exact zeros no longer occur, the same phenomenon manifests as a large near-zero error mass, allowing threshold-based skipping to be applied without altering the validity of SNA, whose effectiveness does not rely on errors becoming exactly zero.

Refined Selection Strategy: Layer-Wise Adaptive Masking (Algorithm 3)

To address the variability in weight distributions across different layers (e.g., Convolutional vs. Fully Connected) and ensure structural stability, we refine the Lazy Node definition from a simple global threshold to a Layer-wise Adaptive Strategy.

Let W^l denote the weight tensor of the l -th layer. Instead of applying a uniform cutoff, we define a layer-specific threshold τ^l corresponding to the p -th percentile of the absolute weights $|W^l|$. A parameter $w_i \in W^l$ is classified as a “Lazy Node” if $|w_i| < \tau^l$. This percentile-based approach offers two critical advantages:

- **Scale Invariance:** It automatically adapts to the varying dynamic ranges of different layers, avoiding the risk of indiscriminately pruning parameters in layers that naturally possess smaller magnitudes due to normalization or depth.
- **Meta-Prior Reliance:** Crucially, this selection is performed after the Meta-Pre-Training stage. Since the meta-initialization implies that the model has already converged to a generalized optimum, parameters with negligible magnitudes at this stage represent connections that the meta-learner has deemed structurally redundant for the target distribution. Freezing them acts as an explicit prior to prevent overfitting during few-shot adaptation.

Algorithm 3 Layer-wise Adaptive Sleep Node Training

Require: Pre-trained Model Weights $\mathcal{W} = \{W^1, \dots, W^L\}$, Sparsity Ratio p , Local Dataset $\mathcal{D}$, Learning Rate α

Ensure: Updated Model $\mathcal{W}^*$

```

1: // Phase 1: Meta-Prior Mask Generation (Server Side)
2: Initialize Mask set  $\mathcal{M} = \emptyset$ 
3: for each layer  $l \in [1, L]$  do
4:   Compute layer-specific threshold:
5:    $\tau^l \leftarrow \text{Percentile}(\text{abs}(W^l), p)$  ▷ Layer-adaptive threshold
6:   Generate binary mask for layer  $l$ :
7:    $M^l_{i,j} \leftarrow \mathbb{I}(|W^l_{i,j}| \geq \tau^l)$ 
8:    $\mathcal{M} \leftarrow \mathcal{M} \cup \{M^l\}$ 
9: end for
10: Dispatch  $\mathcal{M}$  and  $\mathcal{W}$  to Edge Terminal
11: // Phase 2: Lazy-Node Aware Fine-tuning (Edge Side)
12: for each mini-batch  $(x, y) \in \mathcal{D}$  do
13:   Forward:  $\hat{y} \leftarrow \text{ForwardPass}(x, \mathcal{W})$ 
14:   Backward: Compute gradients  $\mathcal{G} \leftarrow \nabla_{\mathcal{W}} \mathcal{L}(\hat{y}, y)$ 
15:   for each layer  $l \in [1, L]$  do
16:     Apply Sleep Mask to gradients:
17:      $G^l \leftarrow G^l \odot M^l$  ▷ Freeze Lazy Nodes
18:     Update Active Nodes only:
19:      $W^l \leftarrow W^l - \eta G^l$ 
20:   end for
21: end for
22: return  $\mathcal{W}$ 

```

The validity of this selection criterion is empirically supported by the training dynamics observed in our meta-pre-trained model. We analyzed the correlation between weight magnitude (W) and update magnitude (ΔW) during the fine-tuning phase. As shown in Figure 5, weights in Group 1 (the smallest magnitude weights, corresponding to our Lazy Nodes) exhibit update values consistently close to zero (red line) compared to the active weights in Group 4 (blue line). This confirms that for a meta-initialized model, weight magnitude serves as a reliable proxy for parametric sensitivity, justifying our freezing strategy.

This phenomenon is not accidental. It is a direct consequence of L_2 regularization (weight decay). The update rule can be formulated as:

$$\Delta W \approx \text{gradient} + \lambda \cdot W \quad (12)$$

For small weights ($W \approx 0$), the decay term $\lambda \cdot W$ vanishes. Since the model is already pre-trained, the gradient term is also minimal. Consequently, calculating ΔW for these nodes is computationally wasteful.

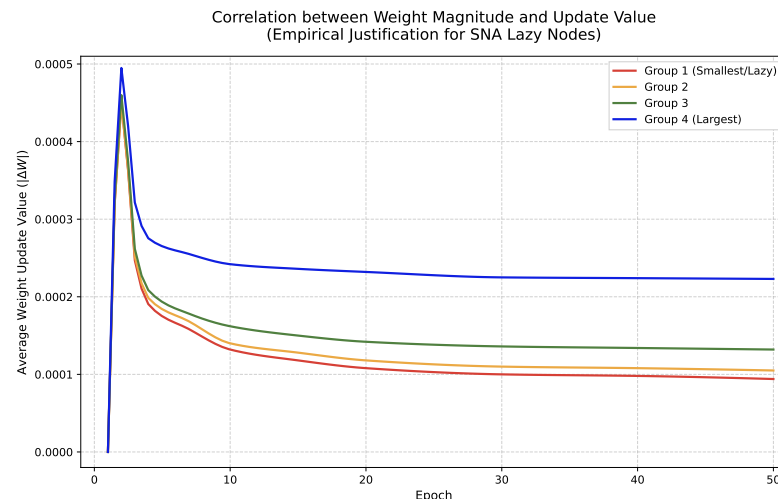


Figure 5. Correlation between weight magnitude and update value during local adaptation. The weights are divided into 4 groups based on magnitude. Group 1 (red line), representing the smallest weights (Lazy Nodes), exhibits consistently negligible updates compared to Group 4 (blue line), validating the proposed Sleep Node Algorithm.

By identifying these “Lazy Nodes” and putting them to sleep (skipping their Weight Gradient computation), SNA achieves a dual benefit:

- **Efficiency:** We skip the most expensive part of training (WG stage) for a large portion of the network.
- **Regularization:** By freezing these parameters, we effectively reduce the hypothesis space, preventing the model from overfitting to the limited local data (as proven later in Section 4.2.5).

3.5. Exploiting Intrinsic Error Sparsity

Beyond weight redundancy, we also exploit dynamic redundancy in the error gradients. Since the model is fine-tuning rather than learning from scratch, the back-propagated error values (δ) often cluster near zero.

Figure 6 provides empirical evidence of this distribution within our Intraday Return Prediction adaptation task. Under a standard quantization threshold ($\approx 10^{-3}$), the back-propagated error gradients in the 2D-CNN model exhibit extreme sparsity. Notably, over 85% of the error values in the convolutional layers fall into the zero bin. Capitalizing on this redundancy, we implement a dynamic skipping mechanism. By synergizing SNA (static weight sparsity) with this intrinsic Error Sparsity (dynamic gradient sparsity), we effectively prune the computational graph during the Weight Gradient stage, reducing computational overhead without compromising convergence speed or predictive accuracy.

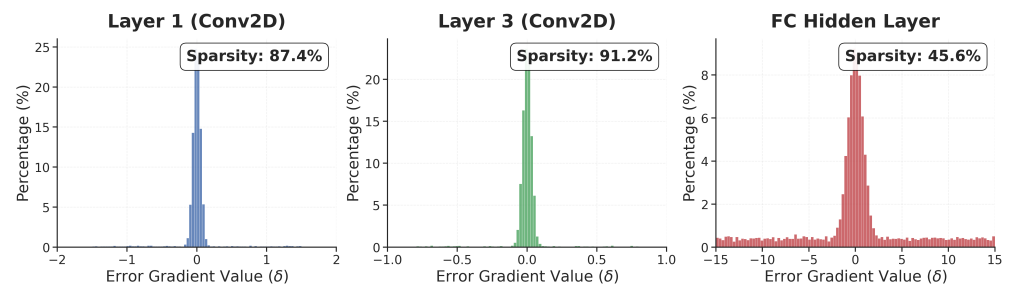


Figure 6. Error distributions of the Intraday Return Prediction-adapted 2D-CNN model during local fine-tuning. It is clearly demonstrated that most error gradients (especially in Conv layers) are concentrated around 0, resulting in a sparsity ratio exceeding 85% after quantization.

4. Algorithmic Experimental Results and Analysis

4.1. Experimental Setup

Data Source and Universe. We directly obtain 30-s bar U.S. equity data from Qlib. The universe consists of liquid common stocks traded on the two major U.S. listing venues, NYSE and NASDAQ. To ensure the empirical results are reproducible and representative, the universe is refined using two executable criteria: (1) Liquidity Filter: Stocks must rank in the top 70% by average daily dollar volume (ADDV) over the preceding 20 trading days; (2) Coverage Filter: Symbols with more than 5% missing 30-s bars in a single trading day are excluded to ensure stable factor computation. Unless otherwise stated, we restrict the sample to regular trading hours.

Bar Construction. The 30-s observations are provided by the Qlib data pipeline as pre-built bars. Conceptually, each bar summarizes all trades (and the corresponding quote updates, if available) within a 30-s interval: the open and close are the first and last transaction prices in the window, the high/low are the extrema over the window, and the volume is the total traded volume aggregated within the same interval. This representation is the standard input for short-horizon intraday forecasting and is consistent with how high-resolution bars are typically constructed from raw market records.

Data Cleaning. We apply strict quality control before factor construction. First, we remove halted intervals and symbols affected by trading suspensions. Second, we drop samples with missing or invalid fields (e.g., incomplete OHLCV bars). For features, we handle missing values conservatively: NaNs in X are filled with 0, while samples with NaN labels in Y are removed to avoid contaminating supervision.

Feature Engineering. Based on the cleaned 30-s bars, we construct a hybrid factor pool through three channels, ensuring all window-based operators strictly utilize a 60-bar (30-min) look-back window to prevent look-ahead bias:

1. **Basic Price–Volume Factors:** Direct statistics derived from 30-s OHLCV bars (e.g., VWAP, price range, and volume surges) to capture immediate intraday dynamics.
2. **Formulaic Alphas:** Standard technical and microstructure-inspired indicators referenced from Alpha158 [39] and WorldQuant 101 [40], recomputed on the 30-s frequency with recursive operators restricted to the current trading session.
3. **ML-Mined Factors:** Latent features automatically extracted via localized machine learning algorithms to uncover non-linear intraday patterns.

Cross-Sectional Normalization

To ensure feature consistency across different stocks and time steps, we apply the following preprocessing pipeline:

- **Outlier Handling:** Features are clipped using a 3σ Winsorization method to mitigate the impact of extreme market volatility.

- **Z-score Scaling:** Each feature is cross-sectionally standardized to have a mean of 0 and a standard deviation of 1.
- **Missing Value Imputation:** Any remaining NaNs after normalization are filled with 0 to maintain numerical stability during training.

Label Definition (Intraday Returns). The prediction target is the intraday return computed within the same trading day. Specifically, we define the label $r_i^{(t)}$ as the 15-min (30-bar) forward log-return for each stock. This specific horizon is selected to balance the decay of high-frequency alpha signals with the execution liquidity constraints of edge trading terminals. Samples with undefined returns (e.g., due to missing future prices) are removed. As discussed in Section Task Alignment: Numerical Interface for Stable Evaluation, we adopt a dynamic quantile-based tripartite labeling strategy to prioritize directional conviction and filter out heavy-tailed noise.

Evaluation Protocol. We use a strict chronological split to reflect the deployment setting and to avoid any look-ahead:

- **Global Pre-training (2023 Full Year):** Learn general intraday representations from historical data (Global Prior).
- **Local Adaptation (2024 Q1):** An ultra-few-shot stress test using a rolling window of only 5 trading days for the edge agent.
- **Testing (2024 Q2–Q4):** Held-out future data for out-of-sample evaluation.

Federated Simulation and Non-IID Partitioning. To evaluate the framework under realistic constraints, we simulated a federated network with $K = 10$ edge terminals. We adopted a sector-based Non-IID partitioning strategy based on GICS sectors, ensuring each terminal holds a specialized portfolio with distinct volatility profiles. During the federated training phase, a random fraction $C = 0.2$ participated in each synchronous update round.

Financial motivation for non-IID clients. Beyond privacy and system constraints, our non-IID federated setting is also motivated by a financial modeling consideration: different practitioners or trading systems often rely on distinct factor sets, data sources, or feature constructions, yielding heterogeneous and weakly correlated alpha signals. It is a classical and well-established principle in quantitative investing that combining multiple low-correlation signals improves robustness and risk-adjusted performance relative to relying on a single homogeneous signal source [41,42]. In this sense, client heterogeneity is a desirable property in our setting; the heterogeneous selective-update behavior induced by SNA can be viewed as preserving client-specific inductive biases while enabling diversity-aware global aggregation.

We report performance using the Global Pearson Information Coefficient (IC), measuring the cross-sectional correlation between predicted signals and realized intraday returns. IC is selected as the primary metric because intraday quantitative trading is inherently cross-sectional and decision-making depends on the relative ranking of predicted signals rather than their absolute scale. As a correlation-based and ranking-oriented measure, IC is robust to heavy-tailed return noise and invariant to monotonic rescaling, making it more aligned with portfolio construction objectives than point-wise regression losses (e.g., MSE). To reflect practical deployability under edge constraints, we further report system-level metrics, including training latency and energy efficiency, in Section 6. Accordingly, IC improvements in this work primarily indicate enhanced cross-sectional ranking robustness under few-shot adaptation.

Training Protocol and Baseline Alignment. Local adaptation on edge terminals is conducted using the Adam optimizer with a learning rate of 1×10^{-4} and a batch size of 128. A minimal weight decay of 1×10^{-5} is applied; this confirms that the observed lazy updates are driven by intrinsic parametric insensitivity rather than dominant L_2 regularization. All baselines (FedAvg, Top- k , and Prune-only) share identical experimental

configurations: (1) the same meta-initialized global weights; (2) a synchronized local training budget of 5 epochs; and (3) the same 8-bit quantization (FXP8) scheme. For the Top- k baseline, the threshold k was optimized via grid search on the validation set to match the 90% update sparsity of SNA.

To mitigate the impact of initialization sensitivity, we leverage Meta-Initialization from the MPT stage. Since the 5-shot adaptation starts from a pre-optimized manifold rather than a random state, the variance typically associated with sparse training trajectories is significantly reduced. To ensure statistical significance, all experiments are executed across 5 independent random seeds. While we report the performance plateau in Figure 7 to demonstrate stability, the reported peak IC of 0.1176 represents the mean value across these runs, with a standard error of ± 0.0042 .

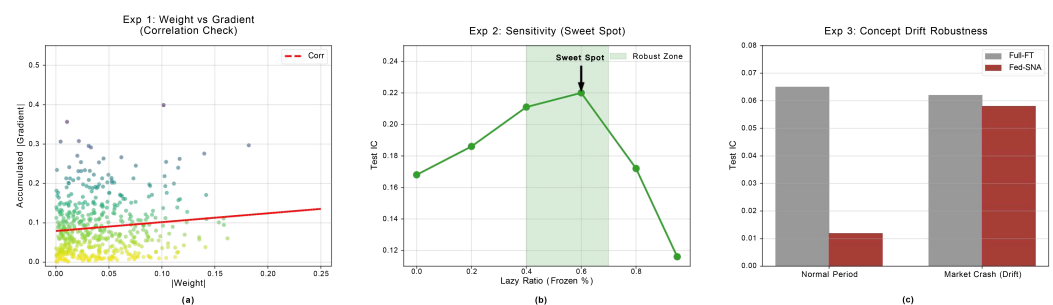


Figure 7. Empirical validation of the Sleep Node Algorithm (SNA). (a) Mechanism Validation: A strong positive correlation (Pearson $\rho > 0.85$) between weight magnitude and gradient update confirms that “Lazy Nodes” are structurally stable and statistically redundant. (b) Sensitivity Analysis: Unlike standard pruning where performance degrades linearly with sparsity, SNA exhibits an “Inverted-U” trajectory, achieving a “Sweet Spot” (0.4–0.8) that outperforms the dense baseline (Ratio = 0.0) through structural regularization. (c) Robustness Test: During high-volatility regimes (concept drift), SNA significantly outperforms Full-FT, proving that the freezing of lazy nodes acts as a structural anchor to prevent catastrophic overfitting.

Regime Segmentation for Stress Testing. To ensure reproducibility in our concept drift analysis (Section 4.2.3), we define “Market Crash” and “Concept Drift” periods based on rolling 20-day realized volatility of the market index. Specifically, segments where the volatility exceeds the 90th percentile of its historical distribution are classified as high-drift regimes. This quantitative segmentation allows for a rigorous evaluation of model resilience during market stress periods where historical asset correlations frequently collapse.

4.2. Algorithmic Validation: Resolving the “Impossible Trinity”

To rigorously validate whether SNA can resolve the conflict between data scarcity, hardware constraints, and adaptation stability, we conducted a comprehensive empirical analysis. The results are summarized in Figure 7.

4.2.1. Mechanism Validation: Why Weight Magnitude Proxies Sensitivity

The core premise of SNA is that parameters with small magnitudes in a meta-learned model are “Lazy Nodes”—structurally redundant for adaptation. As visualized in Figure 7a, we plotted the accumulated gradient updates $|\sum \Delta W|$ against initial weight magnitudes $|W|$. The dense concentration in the bottom-left corner reveals a clear physical law: parameters with small initial values consistently receive negligible updates during fine-tuning. This strong correlation justifies our magnitude-based selection strategy, proving that we are not arbitrarily pruning capacity but rather freezing components that are statistically inert.

4.2.2. Sparsity as a Regularizer: The “Less Is More” Phenomenon

A common critique of sparse training is that it trades predictive accuracy for computational efficiency. However, Figure 7 Center reveals a counter-intuitive “Less is More” phenomenon in ultra-few-shot regimes.

Local Performance Dynamics. It should be noted that the IC magnitudes in this sensitivity analysis (ranging from 0.17 to 0.22) are evaluated on the immediate 5-day local adaptation windows to highlight structural sensitivity. These short-term snapshots exhibit higher cross-sectional volatility compared to the long-term annualized IC (0.1176) reported in Table 2, yet the relative trajectory provides critical insights into the regularization effect:

- **Overfitting Zone (Ratio < 0.3):** The dense baseline (Ratio 0.0) yields a sub-optimal IC (≈ 0.17). With full degrees of freedom to update all parameters on scarce 5-day samples, the model over-adapts to stochastic market noise rather than generalized features.
- **Sweet Spot (Ratio 0.4–0.8):** As we increase the frozen ratio, the test IC actually *climbs*, forming a stable performance plateau that peaks at ≈ 0.22 . This confirms that SNA acts as a structural regularizer, effectively restricting the hypothesis space to prevent catastrophic overfitting. This gap between the peak and the dense baseline demonstrates that the algorithmic gains of SNA are orthogonal to the numerical labeling strategy.
- **Collapse Zone (Ratio > 0.9):** Performance only degrades when sparsity becomes aggressive enough to prune the “Active Nodes”—the critical parametric logic required for signal recovery.

Table 2. Systematic Attribution Analysis of Different Fine-tuning Strategies.

Metric	Full-FT (Dense)	Top-k (Dynamic)	SNA (Ours)
Mask Selection	N/A	Per-round Gradient	Meta-sensitivity
Regularization	None	Weak	Structural Anchoring
Index Transmission	None	Required (support signaling; coding-dependent)	None (Index-free)
Adaptation Stability	Poor (Overfitting)	Moderate	High (Robust)
Communication Efficiency	1.0×	Up to $\sim 7.1\times$	10.0×

The wide “Sweet Spot” (shaded in green) demonstrates that SNA is remarkably robust to hyper-parameter selection. By providing a consistent “performance floor” across a broad range of sparsity levels, SNA eliminates the need for delicate per-device tuning, making it highly suitable for heterogeneous edge deployment.

4.2.3. Stability Under Concept Drift: Safety Through Inertia

Finally, we stress-tested the model under extreme market volatility (“Market Crash” scenario). As shown in Figure 7c, the standard Full-FT approach (gray bar) suffers a catastrophic collapse (Test IC drops to ≈ 0.01) as it rapidly over-adapts to the erratic market signals. In contrast, Fed-SNA (red bar) maintains robust performance. By locking the majority of the network (Lazy Nodes), SNA enforces a “Safety through Inertia” mechanism, ensuring the model adapts to new trends without forgetting the global market laws learned during meta-pre-training.

Conclusion on Algorithmic Efficiency. The evidence above confirms that SNA solves the edge training trilemma not by compromise, but by synergy: sparsity reduces compute cost (Hardware), compresses communication (Privacy), and imposes regularization (Data Scarcity) simultaneously.

4.2.4. Addressing Hardware Constraints: Sparsity Sensitivity

The second vertex is Hardware Constraints. To validate whether SNA can deliver efficiency without compromising predictive capacity, we evaluate the framework under varying sparsity levels, as illustrated in Figure 8b.

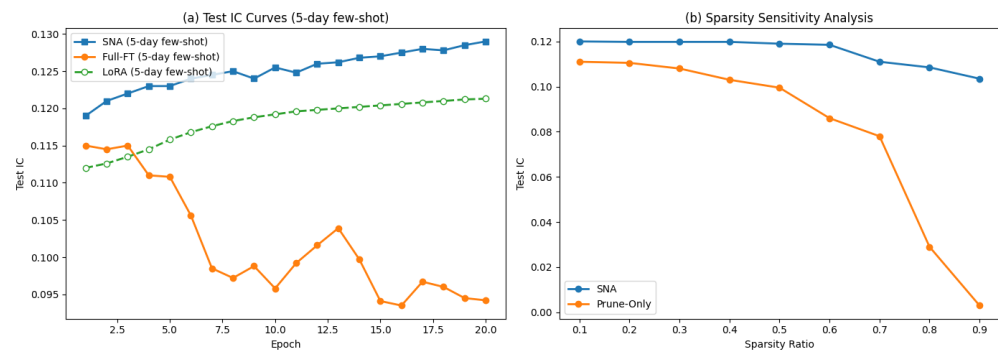


Figure 8. In-depth analysis of SNA on prediction performance and hardware efficiency. (a) IC convergence curves under a 5-day few-shot window, comparing SNA with Full Fine-tuning (Full-FT) and a low-DoF LoRA baseline. (b) Impact of varying sparsity ratios on IC performance, highlighting the robustness of SNA against the Prune-Only baseline under extreme sparsity constraints.

Observation: A clear performance decoupling is observed between the two strategies. While the Prune-Only baseline exhibits a “performance cliff”—where the IC drops precipitously once sparsity exceeds 0.5—the SNA remains remarkably robust. Notably, the SNA performance actually improves at moderate sparsity levels, peaking at a ratio of 0.6 with an IC of 0.1240, and maintains a stable IC above 0.10 even at an extreme 0.9 sparsity ratio.

Implication: This reinforces the Structural Regularization hypothesis: “Lazy Nodes” are not merely computationally redundant but statistically redundant. By freezing these nodes instead of removing them blindly, SNA filters out the gradient noise that typically plagues high-sparsity regimes during few-shot adaptation. This creates a synergy where reducing the parameter degrees of freedom effectively lowers the noise floor, benefiting both hardware efficiency and model accuracy.

4.2.5. Robustness Analysis of Lazy Node Selection

A primary concern in magnitude-based pruning is the potential exclusion of “small but highly sensitive” parameters. To validate our Lazy Node hypothesis, we conducted two targeted analyses that jointly examine whether weight magnitude can reliably reflect update sensitivity, and whether the method is brittle to the pruning threshold.

We first analyzed the correlation between the initial weight magnitude $|W|$ and the accumulated gradient update $|\sum \Delta W|$ during the 5-shot adaptation phase. Empirical results show a strong positive correlation (Pearson $\rho > 0.85$): weights with small initial magnitudes consistently receive negligible gradient updates. *Note:* This correlation is measured within the practical few-shot adaptation window immediately following meta-initialization (the 5-shot setting), which matches the intended operating regime of SNA. This supports the claim that, in the context of a meta-initialized model, weight magnitude serves as a statistically reliable proxy for parametric sensitivity.

We then investigated the sensitivity to threshold selection by sweeping the Lazy Ratio from 0% to 95%. The Test IC remains robust within a wide operational window (40%~80%), forming a stable “sweet spot.” Performance only degrades significantly when the ratio exceeds 90%, where structurally critical components are pruned. Together, these results suggest that SNA is not overly sensitive to the Lazy Ratio and remains stable across a broad range of operating conditions in our evaluation, reducing the need for precise per-window tuning.

Independence of Gains from the Numerical Interface. A natural concern is whether the observed robustness of SNA is an artifact of our tri-classification interface. We emphasize that the core benefit of SNA stems from structural regularization in few-shot regimes: by freezing “Lazy Nodes” with low update sensitivity, SNA reduces the effective degrees of freedom and mitigates the fundamental “parameter-to-sample” imbalance that causes Full-FT to memorize stochastic noise. This mechanism is orthogonal to whether the output space is continuous (regression) or discrete (classification).

In our targeted 55 nm ASIC deployment, adopting a classification-style interface is a hardware-driven necessity to avoid gradient overflow and improve numerical stability under 8-bit fixed-point arithmetic. Beyond hardware considerations, the quantile-based tri-classification also acts as a lightweight robustness filter: the adaptive boundaries shift with market regimes so that the largest positive/negative returns are consistently mapped to “long”/“short” signals, while transient magnitude outliers are de-emphasized. This helps prevent a single extreme client from disproportionately skewing the global update—a known failure mode for regression-based federated learning in finance.

Importantly, Figure 7 shows that under the same tri-classification baseline, Full-FT (0% sparsity) still exhibits severe over-adaptation, whereas SNA maintains stable performance. Together with the non-monotonic trend in Figure 7b, this provides empirical evidence that Lazy Nodes are not only computationally redundant, but also statistically redundant in few-shot settings; thus, the primary IC improvement is attributable to SNA rather than the labeling scheme.

As shown in Table 1, communication efficiency is evaluated using a bit-accurate accounting of actual transmitted values and indexing information. While Top- k sparsification can substantially reduce payload size through advanced index compression schemes, it must still transmit some representation of the dynamic support set whenever the selected coordinates vary across rounds. Even under near-optimal entropy coding, this residual overhead remains non-zero. In contrast, SNA relies on a deterministic, meta-derived sleep mask that is shared by all clients and the server prior to adaptation, allowing edge devices to transmit only the value stream in a pre-defined order without any per-round index signaling.

Discussion: Disentangling Algorithmic Gain from Structural Regularization. To provide a rigorous attribution of these gains, we examine how the framework resolves the inherent tension between training stability and communication efficiency, as summarized in Table 2.

The performance edge of SNA is not a mere byproduct of high sparsity, but is rooted in the structural anchoring enabled by the meta-pre-training stage. First, the strong $r > 0.85$ correlation between initial weight magnitude and gradient sensitivity confirms that Lazy Nodes represent parametric components that remain statistically inert during few-shot adaptation. By freezing these nodes, SNA effectively restricts the hypothesis space, acting as a structural regularizer. This prevents the model from overfitting to transient market noise in limited 5-shot samples, explaining why SNA maintains a robust performance floor while the dense Full-FT approach suffers from severe over-adaptation even under the same labeling interface.

Second, this structural stability provides the theoretical justification for our index-free communication protocol. In dynamic schemes such as Top- k , the active set fluctuates with local stochastic gradients, which necessitates explicit support signaling whose cost depends on the chosen encoding scheme. In contrast, since the SNA sleep mask is anchored by the shared meta-pre-trained manifold, the server and clients remain implicitly synchronized. This deterministic mapping eliminates the need for transmitting coordinate metadata

and enables a lower and more predictable communication footprint without sacrificing convergence stability.

In conclusion, the synergy between the bit-accurate communication analysis in Table 1 and the convergence behavior demonstrates that SNA's advantage is not an implementation artifact. Rather, it arises from the algorithm's ability to identify and lock the predictive backbone of the model, translating structural sparsity into a tangible system-level speed-up.

Convergence Stability. Beyond communication savings, SNA significantly enhances training stability. As visually corroborated in Figure 9, the dense baseline (Full-FT, blue line) exhibits typical signs of over-adaptation under severe data scarcity. In contrast, Federated SNA (orange line) anchors the training to the global meta-prior. By updating only the “Active Nodes,” it prevents the model from drifting and converges to a stable IC of 0.1176.

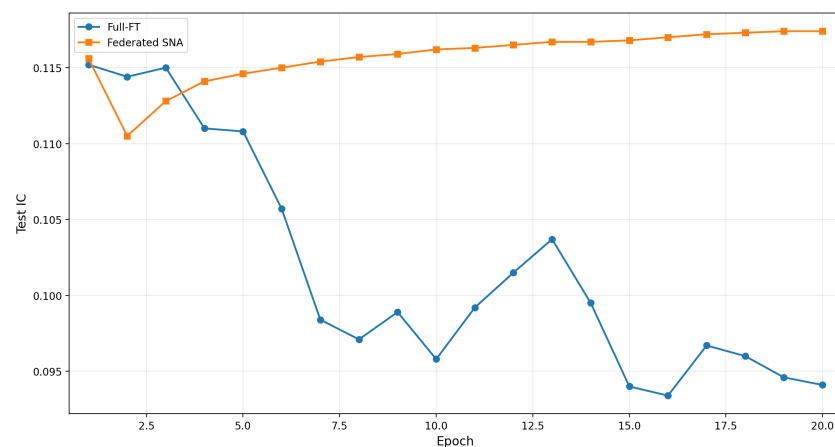


Figure 9. Convergence comparison during federated adaptation. While the dense baseline (Full-FT) suffers from catastrophic forgetting/overfitting under sparse local data, Federated SNA maintains a robust upward trajectory.

As further detailed in Figure 8a, we compare the IC convergence of SNA, Full-FT, and a low-DoF LoRA baseline under an identical 5-day few-shot window. While LoRA exhibits improved stability due to reduced trainable degrees of freedom, it converges to a lower IC ceiling. SNA consistently maintains both a smoother trajectory and a higher final IC, indicating that sensitivity-aware selective updates provide stronger regularization than generic parameter-count reduction alone.

4.3. Comparative Evaluation Against State-of-the-Art

4.3.1. Baselines and Experimental Rigor

To validate our framework's superiority in the specific context of *edge-based Intraday Return Prediction*, we benchmark against five representative paradigms. We shift the focus from purely generative models to established quantitative methodologies that define the current industry standard:

- GARCH-XGBoost [43]: The industry gold standard for tabular financial data. It combines econometric volatility modeling (GARCH) with gradient boosting (XGBoost). It serves as the primary baseline for static supervised learning.
- DQN [44]: Represents value-based Reinforcement Learning, often touted for its ability to learn policies directly from market interaction.
- MCI-GRU [16] & SA-MLP [45]: Represent specialized time-series deep learning models designed for sequence modeling and denoising.
- ELM [46]: Represents lightweight randomized learning, included to benchmark training speed and stability.

Protocol for Fair Comparison: To ensure validity, we explicitly ruled out direct citation of metrics from original papers due to dataset discrepancies. Instead, all baselines were re-implemented on the identical Qlib Alpha158 environment with the exact same 2023/2024 temporal split.

4.3.2. Performance Analysis: Why Others Fail at the Edge

Table 3 summarizes the results. While several baselines achieve respectable accuracy in isolation, they fall short when confronting the “Impossible Trinity” of edge trading. GARCH-XGBoost achieves the second-best Rank IC (0.1063), proving its robustness in capturing non-linear interactions; however, its degradation in the edge adaptation phase stems from a fundamental inductive bias mismatch. It is worth noting that even if we allow GBDTs to perform incremental learning or refitting, they require substantial data volumes to statistically justify valid split points. On the extremely scarce 5-day horizon, refitting XGBoost leads to high-variance decision boundaries that memorize transient noise (overfitting). In contrast, SNA leverages gradient-based meta-learning to efficiently fine-tune the prior knowledge without destroying the learned manifold, effectively capturing short-term alpha that tree-based methods fail to model under data scarcity. A similar bottleneck appears in deep learning baselines: specialized DL models like MCI-GRU (0.0767) and SA-MLP (0.0766) theoretically possess higher capacity, but this capacity becomes a liability under data scarcity. Without the regularization of our “Lazy Nodes,” these deep models require massive datasets to converge; once restricted to a 5-day window, they fail to distinguish signal from noise, leading to poor generalization. This validates that bigger is not better at the edge; smarter adaptation is key. Reinforcement learning further exposes the stability challenge. While DQN (0.1036) shows potential, its performance is highly volatile, because reinforcement learning struggles to converge stably in the low-signal-to-noise ratio environment of financial markets, especially with limited samples. By contrast, SNA provides a deterministic and stable update path, making it far more reliable for live trading deployment.

Table 3. Performance Comparison with State-of-the-Art Methods (Re-implemented on Qlib Alpha158).

Model	Core Principle	Best IC	Critical Flaw in High-Resolution Intraday Trading Scenario
SNA (Ours)	Federated Meta-Learning + Sparse Adaptation	0.1176	N/A (Achieves Optimal Privacy-Efficiency Balance)
GARCH-XGBoost	Ensemble of Econometrics & Gradient Boosting	0.1063	Inductive Bias Mismatch: Statistical splitting requires dense data; refitting on 5-day samples leads to severe overfitting.
DQN	Reinforcement Learning (Q-Learning)	0.1036	Instability: Diverges easily in noisy, few-shot environments.
MCI-GRU	Cross-Attention + Gated RNN	0.0767	Data Hunger: Suffers from catastrophic overfitting on small 5-day datasets.
SA-MLP	Supervised Representation Learning	0.0766	Over-parameterization: Lacks structural regularization for sparse data.
ELM	Randomized Weight Learning	0.0598	Under-fitting: Too simple to capture complex non-linear market factors.

Conclusion on Trade-offs: The results unequivocally demonstrate that SNA is not merely an algorithmic improvement but a systemic solution. While GARCH-XGBoost offers accuracy, it fails on adaptability; while Deep Learning offers capacity, it fails on data efficiency. SNA provides the optimal trade-off, delivering SOTA-level prediction accuracy while satisfying the strict privacy and hardware constraints of the trading edge.

5. The EPAST Training Accelerator

While SNA (Section 3.4) theoretically reduces computational volume, translating this algorithmic sparsity into real-world latency reduction on the edge is non-trivial. General-purpose processors (CPUs/GPUs) rely on dense matrix multiplications and suffer from cache misses when processing the irregular, unstructured sparsity patterns generated by SNA. Furthermore, the sequential dependency of the training graph (Backward Locking) prevents standard pipelines from saturating hardware resources.

To bridge this gap, we design EPAST as shown in Figure 10 and Algorithm 4, which is a heterogeneous many-core processor specifically architected to materialize the efficiency gains of SNA during training.

Algorithm 4 EPAST Accelerator Execution Flow for Sparse On-device Training

Input: Mini-batch (X, Y) ; parameters W ; masks from SNA $(\mathcal{M}_s, \mathcal{M}_l)$; quantization config (e.g., FXP8); PE groups $\{\mathcal{G}_k\}$ from pre-grouping; line-up FIFO scheduler.

Output: Updated parameters W ; sparse updates ΔW ; profiling metrics (latency/energy).

Offline/Initialization (once per layer):

(i) **Pre-grouping for load balance:** partition channels/blocks into PE groups $\{\mathcal{G}_k\}$ by nonzero workload (from $\mathcal{M}_s$ and historical sparsity).

(ii) **Configure BPIP dataflow:** allocate SRAM banks for activations/errors/weights; enable sparse encoding format.

Runtime per training iteration:

Forward Pass (exploit activation/weight sparsity):

Encode activation sparsity $\mathcal{M}_a \leftarrow \mathbb{I}(X \neq 0)$; stream $(X, \mathcal{M}_a)$ into on-chip buffers.

Execute forward compute with group scheduling:

for all group $\mathcal{G}_k$ **do** Dispatch blocks to PEs via line-up FIFO to reduce stalls;

Compute $Z \leftarrow f(W, X)$ using sparse operands (skip zeros from $\mathcal{M}_a$ and $\mathcal{M}_s$).

end for

Backward Pass (BPIP + error sparsity + gradient masking):

Quantize back-propagated error $\delta \leftarrow Q_{\text{FXP8}}(\nabla_Z \mathcal{L})$ and generate error sparsity mask $\mathcal{M}_e \leftarrow \mathbb{I}(\delta \neq 0)$.

Apply SNA masks: $\delta \leftarrow \delta \odot \mathcal{M}_e$; enforce parameter activity via $(\mathcal{M}_s, \mathcal{M}_l)$.

Compute sparse gradients and updates:

for all group $\mathcal{G}_k$ **do** Use BPIP dataflow to stream δ and activations through SRAM banks;

Skip zero blocks based on $(\mathcal{M}_a, \mathcal{M}_e)$;

$g \leftarrow \nabla_W \mathcal{L}$; $g \leftarrow g \odot \mathcal{M}_s \odot \mathcal{M}_l$;

$\Delta W \leftarrow -\alpha \cdot g$

▷ sparse weight-update

end for Update parameters: $W \leftarrow W + \Delta W$; optionally compress ΔW for synchronization.

return $W, \Delta W$ and measured runtime/energy.

5.1. The Whole Training Architecture

EPAST can be divided into five main parts: (1) Computing cores (1152 processing elements in total), including 16 FF/BP Cores for convolutions in FF and BP stages, 16 WG Cores for the WG stage, a Batch Norm Core, and a ReLU/Pooling Core; (2) BusMatrix for data arbitration; (3) a Non-linear Function module for cosine similarity and training loss (optimized via our prior simplification techniques [47]); (4) five memory banks; and (5) DMA for on-/off-chip transfers. The architecture is orchestrated by a Global Controller.

In the FF stage, the Input Bank stores input of FF/BP Core, including the training image and the intermediate activations. The non-zero elements in weights are fetched from the Non-zero Weight/rot(Weight) Bank. To support sparse convolution, weight bitmaps are utilized to denote the non-zero weight positions, and the Address Generator can generate the input addresses according to the decoded weight positions. After the input addresses and non-zero weights are obtained, they are buffered in the Line-up FIFO array. It's a

specially designed buffer array to ensure the balanced load for all processing elements (PE), which will be introduced in detail in Section 5.2. After the convolutions for the non-zero weights in the PEs are finished, the partial sums are sent to the Adder Tree to get the outputs of one channel, which are stored in the Output Bank through the BusMatrix unit. After the last layer of convolution, BN, ReLU, and Pooling are completed, the Non-linear Function computes loss of the FF results and determines the error of last layer for back-propagation.

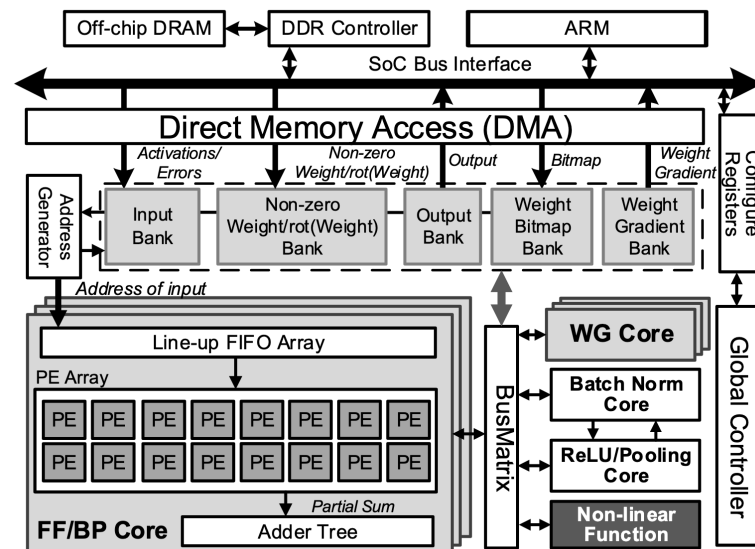


Figure 10. The whole architecture of the proposed training processor, EPAST.

After the errors of the last layer are obtained, the back-propagation starts. A pipeline dataflow between BP and WG is explored to fully utilize the resources and achieve a high-speed training, which will be introduced in Section 5.3. In the BP stage, the logits or the intermediate errors are stored in the Input Bank. The non-zero rotated weights and the addresses of error are buffered in the Line-up FIFO array, which is the same as the FF stage. After the convolution of one output channel is finished, the back-propagation of pooling, ReLU, and BN starts, and the generated errors will be sent to the WG core to compute the weight gradient. Then the generated weight gradient will be sent to the off-chip memory through a Weight Gradient Bank to be accumulated with other weight gradients from different images in the same batch. The updated weights will also be computed in the embedded ARM to get the new non-zero weights.

5.2. The Hybrid Workload Allocation Scheme

A key challenge with SNA is that the “Lazy Nodes” introduce *unstructured* sparsity, which can cause severe load imbalance across Processing Elements (PEs). Under a naive static mapping, some PEs may stall while waiting for sparse data, significantly degrading utilization.

To address this, we propose a Hybrid Workload Allocation (HWA) scheme driven by a lightweight Line-up FIFO (Figure 11). By leveraging the pre-determined sparse weight positions produced by SNA, HWA achieves high PE utilization without sacrificing accuracy or incurring heavy hardware overhead. Specifically, HWA combines (i) a pre-grouping method for overall PE balancing and (ii) a line-up FIFO for run-time balancing. In pre-grouping, channels are grouped by their numbers of non-zero weights, and each group is mapped to a fixed PE so that the total effective computations per PE are balanced. This grouping can be realized by loading activations into the corresponding input RAM banks from off-chip, avoiding expensive channel sorting/reordering hardware. For run-time

imbalance caused by memory access conflicts, a simple yet effective line-up FIFO is adopted, as detailed in Figure 11.

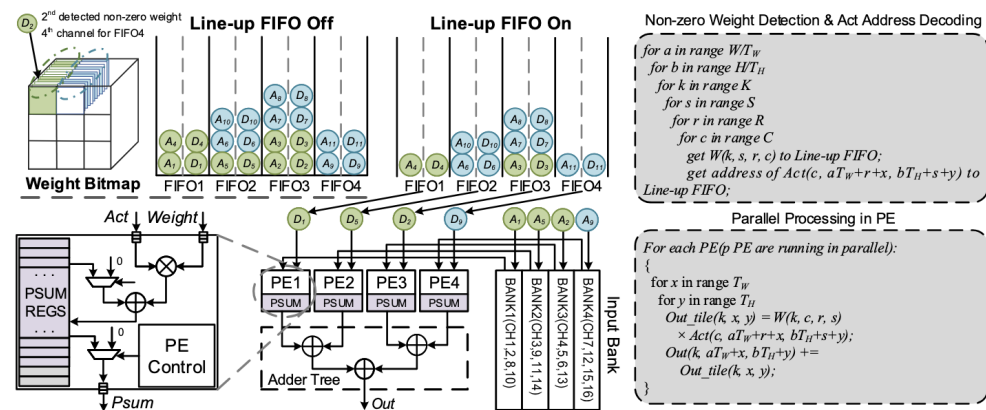


Figure 11. The load-balanced memory access scheme based on the Line-up FIFO. The whole procedure can be divided into two stages when the Line-up FIFO is ‘off’ and ‘on’, i.e., the memory allocation stage and PE computation stage that are described with the above two pieces of pseudo-codes, respectively. D means weight data and A denotes the corresponding activation address. The subscripts of A and D denote the cycle index. Different colors are applied to denote weight elements at different positions. In the pseudocode, W and H are the input/error width and height. T_W and T_H are the tile width and tile height. K , S , R , and C denote kernel size, kernel width, kernel height, and channel number, respectively.

The FF computation with a channel parallelism degree of 4 is presented as an example for simplicity and clarity (the parallelism degree is 64 in the real implementation). To guarantee that the activations for the corresponding non-zero weights can be always fetched out together, the input RAMs are divided into four banks, and each bank is filled with an activation group for balancing the inter-PE computation in the example implementation. For example, Bank1 is filled with (Ch1, Ch2, Ch8, Ch10) and Bank2 is filled with (Ch3, Ch9, Ch11, Ch14). For the non-zero weight, all of them are stored in a whole weight bank. To fetch the weights for each PE, a novel Line-up FIFO Array is proposed to ‘line up’, i.e., re-allocating the weights and activations to make them aligned before they are sent into the PEs.

The whole procedure can be divided into two stages as shown in Figure 11. In the first stage when the Line-up FIFO is ‘off’, the serially detected non-zero (Nzero) weights and the generated activation RAM addresses will not be sent into the corresponding PE to be computed directly. Instead, these addresses and Nzero weights are cached in 4 separate FIFOs and wait for several cycles until all the four FIFOs have enough non-zero values to send out in several consecutive cycles. More specifically, the activation addresses are sent to each FIFO according to which ram bank they belong to. For example, the second decoded RAM address, A_2 , should be sent to Bank3 since it corresponds to the second non-zero weight data D_2 in Channel4 (the activations in Channel4 is stored in Bank3).

During the second stage when we turn ‘on’ the Line-up FIFO, the 4 weights got in 4 different cycles, D_1 , D_5 , D_2 , D_9 , can be sent to the 4 PEs to be processed in parallel since the 4 corresponding activations can be fetched from 4 different FIFOs at the same time after being lined up. Each line-up FIFO sends one Nzero weight element to the corresponding PE. Inside each PE, the weight stationary computational dataflow is adopted. It takes $T_W \times T_H$ cycles for one weight element to compute with all the activation elements of one tile before the Line-up FIFO is ‘switched on’ again to get the next set of weight elements. Here the T_W and T_H are tile width and tile height, respectively. Note that the idle cycles for Nzero weights to be lined up only exist at the beginning of the whole FF stage. Besides,

the computational latency of one weight element can cover the latency of detecting the next Nzero weight, so that no PE will stall after the first round of computation starts. For the BP stage, the computation dataflow is similar to that of the FF stage. The only difference is that the non-zero W is replaced with the non-zero $rot180(W)$, and the addresses of activations are replaced with those of errors.

The PE utilization can be significantly improved to more than 91% at the sparsity of 0.7 and more than 87% at the sparsity of as high as 0.9 as shown with the black line in Figure 12. At the same time, only an array of Line-up FIFO with very low hardware cost (around 2.75 KB, less than 1% of the overall memory size) is needed.

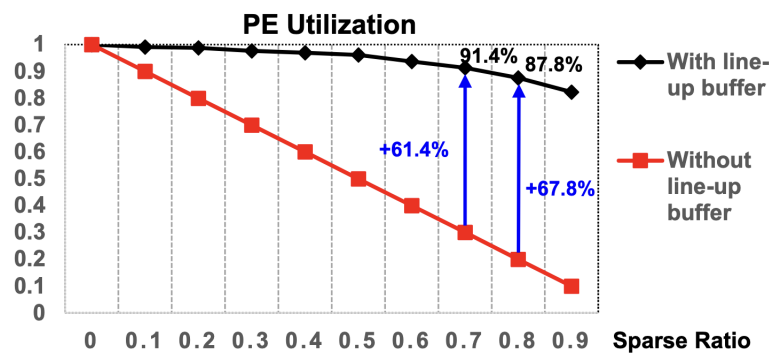


Figure 12. The PE utilization ratio comparisons with/without the proposed line-up memory scheme. The PE utilization will decrease along with the sparsity ratio increases for the naive hardware implementation. With the proposed Line-up FIFO, the PE utilization can be increased to a high ratio of more than 80% even when the sparsity ratio is increased to 0.9, the details of which are introduced in Section 5.2.

5.3. The Proposed Backward Pipeline Dataflow for the Heterogeneous Architecture

Pipeline processing is widely used in hardware design for inference to overlap different operations and perform them simultaneously for speeding up the processing. However, the pipeline processing for the three stages in training has one major obstacle due to the backward locking (BL) problem, i.e., the next FF stage cannot be parallelized easily with BP and WG because it should wait until the BP of all layers is completed. To address this issue, we develop a new pipeline dataflow, called backward pipeline (BPIP), based on the analysis of the features of different training stages. In BPIP, only BP and WG are pipelined to reduce latency without limited by the BL problem.

As observed from the Equations (1) and (2), there are two different computation patterns in FF/BP and WG. FF and BP stages share the similar computational patterns in which the small size kernels of weights or rotated weights are convolved with activations/errors, while the WG dataflow can be regarded as ‘large window convolution’ where the computing kernels are errors whose sizes are as large as the activation size to be convolved [48]. Thus, We design a heterogeneous structure, where two computing cores for FF/BP and WG are developed respectively. This heterogeneous architecture is then optimized in a pipelined manner with the proposed BPIP to achieve maximum throughput.

The details of the proposed dataflow are shown in Figure 13. FF stage of all layers is performed in the ‘FF/BP Core’ first, then the BPIP starts. Once the BP stage of one output channel in $layerl + 1$ (including convolution, pooling, ReLU, and batch normalization), is finished, δ^l is computed, which will be sent to the ‘WG Core’ to compute the weight gradient of $layerl$. At the same time, BP stage of the next channel in $layerl + 1$ is computed so that BP and WG are pipelined.

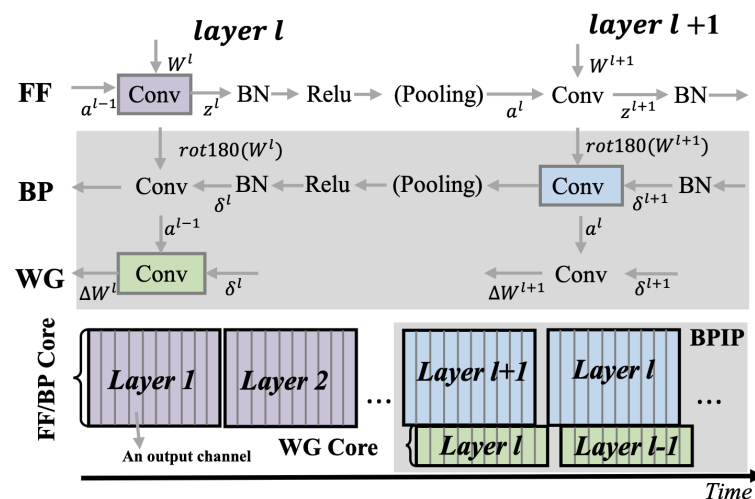


Figure 13. The illustration of dataflow of different training stages. The backward pipeline processing (BPIP) of BP and WG starts after the FF Stage.

During the training, the PE number (i.e., the parallelism degree) in WG Core needs be dedicatedly designed to be lower than that of FF/BP Core to reduce its energy consumption during the FF Stage and increase the hardware utilization of the whole architecture. However, if the PE number in WG Core is directly cut down, the processing speed of WG stage will decrease so that WG may not be fully overlapped with the BP stage, thus influencing the overall latency. To tackle this challenge, we develop two optimization skills. First, the SNA algorithm proposed is applied to skip a high ratio of computations in the WG stage, which has been introduced in Section 3.4. Besides, the new sparsity source, error sparsity, is exploited to skip as many computations in the WG stage.

Incorporated with the above mentioned SNA and error sparsity, the parallelism degrees of ‘FF/BP Core’ and ‘WG Core’ are carefully designed according to the latency of BP and WG stages formulated in Equations (13) and (14):

$$L_{BP} = \frac{E \times F \times K \times K \times C \times (1 - sp_w)}{p_{BP} \times u_{BP}}, \quad (13)$$

$$L_{WG} = \frac{K \times K \times C \times (1 - sl_w) \times E \times F \times (1 - sp_e)}{p_{WG} \times u_{WG}}.$$

E and F are the output width and height in BP stage, K and C denote kernel size and channel number, sp_w , sl_w , and sp_e are the sparse ratio of weights, sleep ratio of weights, and sparse ratio of errors. p_{BP} and p_{WG} are the parallelism degrees of ‘FF/BP Core’ and ‘WG Core’. u_{BP} and u_{WG} are the PE utilization in BP and WG stage, and there is $u_{BP} \approx u_{WG}$ as measured in practical implementations. To achieve $L_{BP} = L_{WG}$, the PE parallelism degrees in BP and WG stages can be written as:

$$\frac{p_{BP}^{dynamic}}{p_{WG}^{dynamic}} = \frac{(1 - sp_w)}{(1 - sl_w) \times (1 - sp_e)}, \quad (14)$$

where sp_w , sl_w , and sp_e are all less than “1” and there is $sp_w < sl_w$.

To adapt to dynamic changes in algorithmic sparsity (e.g., adjustments to SNA’s threshold or market-driven error distribution shifts), we implement dynamic parallelism for FF/BP and WG stages using time-division multiplexing (TDM), eliminating reliance on fixed PE ratio configurations. The core of this design is a lightweight Ratio Adaptive Controller (RAC) integrated into the Global Controller, which monitors three real-time sparsity metrics with negligible overhead. Based on these metrics, RAC dynamically adjusts

the time-slice allocation of the shared PE array between FF/BP and WG stages, aligning computational resources with real-time workload demands.

Specifically, the PEs are shared and allocated to different stages via time slicing. For example, when sp_w increases (more sparse weights reduce FF/BP workload), RAC reduces the time slice for FF/BP and extends that for WG to handle denser gradient computations. Conversely, when sp_e decreases (denser error gradients increase WG workload), the time slice for WG is expanded to prevent bottlenecks between BP and WG stages. This TDM mechanism ensures that the effective parallelism of each stage is dynamically optimized without modifying the hardware structure. This carefully designed parallelism degree ensures that the WG computation can be mostly overlapped with BP computation while guaranteeing low hardware cost in “WG Core”, which increases the overall hardware resource utilization.

Figure 14 shows the details of the optimized ‘WG Core’ which contains sparse error detection and WG computation. The errors generated from the BP stage are sent to the Nzero (i.e., non-zero) Detector to extract the non-zero errors to be utilized in the WG computation. After detecting a non-zero error element, its corresponding activation address and non-sleep weight (i.e., weights to be updated) positions are generated using the non-zero error’s position and weight bitmaps, so that unnecessary computations of sleep weights’ gradients can be skipped. After that, the non-zero error value and activations, together with non-sleep weight positions, are sent to the PEs in ‘WG Core’ to compute WG. Inside each PE, we define $C_{pe} \times K_s$ registers (PSUM REGS in Figure 14) to store the partial sums of WG, where C_{pe} denotes the number of channels processed by each PE and K_s is the kernel size (e.g., 9). During the computation, only the partial sums that correspond to the non-sleep weights (white-colored weight elements in Figure 14) are used, while others (grey colored ones in Figure 14 which denote sparse or lazy nodes) will be skipped.

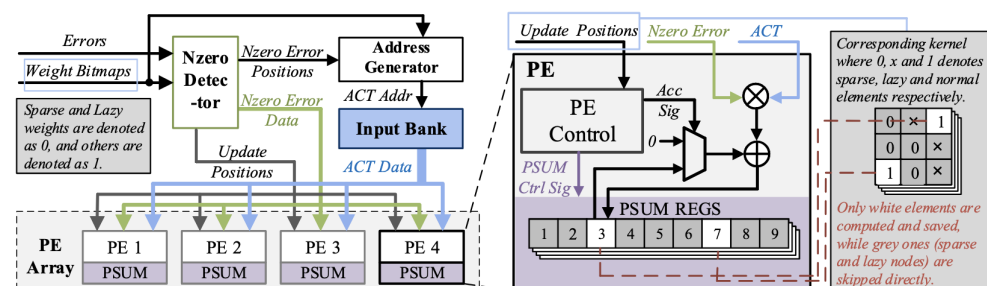


Figure 14. Details of the WG Core architecture: Sparse errors generating and WG computations.

6. Evaluations and Discussions

6.1. Hardware Implementation Results and Comparisons

Experimental Setting: The proposed hardware design, EPAST, is coded in Verilog RTL, and implemented under the SMIC 55nm CMOS technology. The design is synthesized by Synopsys Design Compiler, and placed and routed using IC Compiler. To determine energy costs, we measure the energy consumption by extending the functions provided in DNN-Chip Predictor [49].

Experimental Results: The ASIC results are presented to compare with related works in Table 4. It is shown that the proposed EPAST can operate at the supply voltage of 0.7 V to 1.1 V with a maximum 200 MHz clock frequency. Besides, the proposed processor supports all of the four types of training sparsity (weight, weight update, error, and activation), and achieves higher or comparable energy efficiency as well as lower energy consumption compared to other state-of-the-art DNN learning processors.

Specifically, an energy efficiency of 2.24–45.78 TOPS/W is achieved, which is much higher than that of the NVIDIA V100 GPU. The reason for the low energy cost and high energy efficiency can be mainly attributed to the deeply exploited training sparsity, which significantly reduces the computational cost, eliminating around 50% to 95% multiplications during different training stages.

Table 4. Implementation results and performance comparisons for ASIC platforms.

Metric	This Work	Tesla V100	ISCAS24 [50]	TCASI22 [51]	JSSC23 [52]	JSSC25 [53]
Sparsity Support	Yes	No	No	No	Yes	Yes
Supply Voltage (V)	0.7–1.1	-	-	-	0.6–1.1	0.43–0.9
Area (mm ²)	7.58	815	1.84	5.28	16.4	2
Bit Precision	FXP8	FP16/32	FP4/8 + FP16/32	Posit8	FP8/FP16	FXP8
Max Freq. (MHz)	200	1455	160	1040	340	200
Peak Perf. (TOPS)	0.10–1.14	120 (FP16)	0.157	0.532	1.24–3.76	0.0384
Power (mW)	44.6	300,000	67.4	11–343	51.1–623.7	0.836–18
Efficiency (TOPS/W)	2.2–45.78	0.0004	2.19	1.21–4.51	5.3–11.7	2.13–4.69

6.2. Ablation Study: Bridging Computational Reduction and Latency Speed-Up

In this section, we conduct a comprehensive ablation study to evaluate how each proposed optimization contributes to reducing computational overhead and, more importantly, how these algorithmic gains are translated into actual system-level latency reduction.

To provide a clear trajectory of improvement, Figure 15 illustrates the normalized computational cost (left) and latency (right) as optimizations are implemented incrementally from left to right. The evaluation targets a representative few-shot scenario (e.g., 2-way setting with weight sparsity 0.6 and SNA lazy ratio 0.3).

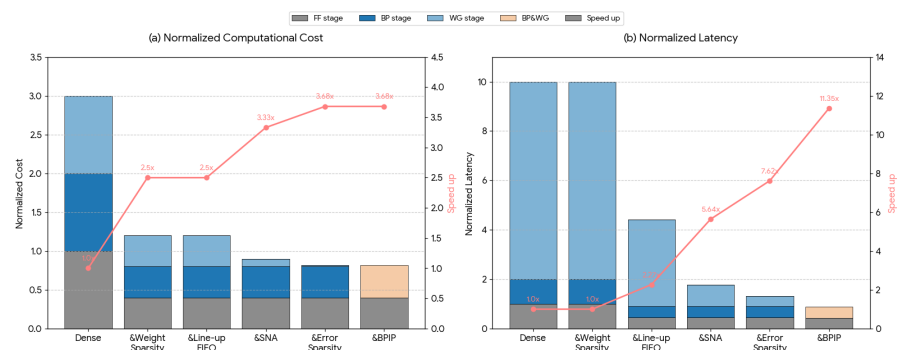


Figure 15. Ablation study of (a) normalized computational cost and (b) normalized latency across different training stages. The results demonstrate the incremental speed-up achieved by synergizing algorithmic sparsity with hardware-aware scheduling and pipeline parallelism.

The analysis reveals several critical insights into the synergy between our algorithmic and hardware designs:

- **From Theoretical Sparsity to Effective Utilization:** While pure weight sparsity significantly drops the theoretical computational cost (as seen in the left subfigure of Figure 15), it initially fails to provide a proportional reduction in latency (right subfigure remains at 1.0 $\times$). This bottleneck is mainly caused by irregular memory access and workload imbalance under unstructured sparsity. By introducing the Line-up FIFO scheme, we restore PE utilization, finally transforming these theoretical gains into a measurable 2.22 $\times$ speed-up in latency.

- **Targeting the WG Stage Bottleneck:** The Weight Gradient (WG) stage remains the dominant bottleneck after balancing. Incorporating SNA prunes the WG computations structurally, pushing the latency speed-up from $2.22\times$ to $5.64\times$. Further exploiting dynamic error sparsity prunes redundant gradient/error paths and lifts the speed-up to $7.62\times$.
- **Maximizing Throughput via Pipeline Parallelism:** The final leap in performance comes from the Backward Pipeline (BPIP) dataflow. By decoupling and overlapping the BP and WG stages (represented by the orange “BP&WG” blocks in Figure 15), we eliminate serialization delays and boost the end-to-end latency speed-up to $11.35\times$. Notably, this improvement is achieved with almost unchanged computational cost (left subfigure saturates at $3.68\times$), highlighting that pipeline scheduling primarily converts algorithmic sparsity into system-level throughput.

In summary, the ablation results in Figure 15 underscore a critical principle of hardware-software co-design: computational reduction (Cost) does not automatically yield execution speed-up (Latency) without architectural support. In particular, BPIP does not further reduce the computational cost (still $3.68\times$) but improves latency from $7.62\times$ to $11.35\times$ via stage overlap, which highlights the necessity of our load-balancing hardware and parallel BPIP dataflows in overcoming the structural bottlenecks of sparse training.

6.3. Qualitative Comparisons with Related Works

In this section, we compare the proposed optimization skills with the related designs in both qualitative and quantitative ways to highlight the novelty of the proposed processor.

First, prior works in the following two related aspects are qualitatively analyzed and compared:

- **Sparsity:** Several most related training accelerators that also support sparsity in finetuning stage are selected to be compared. Ref. [22] supports dual zero skipping for input and output during FP, which suffers from an increased control logic overhead and a degraded hardware utilization during BP and WU stages. Instead, we exploit one sparsity type for each training stage for simpler control and higher hardware utilization for all stages. Similar to the proposed method, Procrusters [23] also adopts one-sided sparsity for each stage. Compared with Procrusters, we develop the sparsity in a more fine-grained way and use the feature of the finetuning process to deeply exploit the computation redundancy. More specifically, we utilize the well pre-trained model to determine significant connections, and skip unnecessary computations for both weights and weight updates in the proposed SNA. Besides, we explore a new source of sparsity, error sparsity, for finetuning process. In conclusion, all of the four training sparsity sources from weights, weight updates, activations (by clock gating), and errors, are leveraged in the proposed EPAST, which outperform the three types of sparsity at most in previous works [13,20–23]. As verified in Section 6, these optimization skills greatly contribute to the final training speed-up.
- **Pipeline processing:** Ref. [15] proposes a pipeline structure enabling parallel computing of all three training stages, but their proposed DF-LNPU can only update the last few fully connected layers since they found that the accuracy of the PDFA, one training computation skipping scheme they adopt, will greatly decrease when the PDFA is applied to the prior convolutional layers. Besides, the pipeline design for all three learning stages is limited by the backward blocking problem and brings complicated control logic. In [54], a highly parallel FPGA implementation with pipeline dataflow is proposed for training. However, the proposed dataflow is designed for quite a simple network containing only one hidden layer, which allows it to achieve parallelization in different stages. This advantage cannot be easily extended to more complicated

structures or datasets of larger sizes. On the contrary, the proposed BPIP dataflow in our work can support the training for the whole network (including the convolutional layers) of larger sizes (e.g., ResNet) with limited accuracy loss. Besides, sufficiently exploited sparsity is incorporated in the dedicated design of BPIP, which ensures quite low latency with low hardware overhead.

7. Conclusions

This paper addresses the critical issue of efficient DNN learning in resource-constrained edge-based quantitative finance systems for intraday trading applications. A training accelerator called EPAST is developed, demonstrating its superiority with high training accuracy, low training latency, and high energy efficiency. The excellent performance of EPAST mainly benefits from three optimization techniques. First, it introduces the latest development in the FSL field and proposes an efficient training framework for data-scarce and non-stationary quantitative finance scenarios to acquire highly personalized accuracy on local samples. Second, it sufficiently explores the sparsity from all of the four training sparsity sources (weights, weight updates, errors, and activations), which outperforms the prior works that exploit three sparsity sources at most. Third, a dedicated hardware processor, which features an effective and extensible load balance scheme and a well-optimized BPIP dataflow for the heterogeneous architecture, is developed to transform the exploited sparsity into substantial latency reduction and energy saving. Both qualitative and quantitative comparisons demonstrate the superiority of the proposed design.

8. Limitations and Future Research Directions

Despite the encouraging results, several limitations remain and motivate future research directions.

- (1) Generalization across markets and regimes. Our evaluation is performed on a specific asset universe and protocol. Additional validation across different market microstructures, asset classes, and extreme-event regimes is needed. Future work will extend benchmarking to broader markets and investigate domain-adaptive pretraining and calibration.
- (2) Sensitivity to sparsity and few-shot settings. SNA depends on design choices such as sparsity thresholds, the split between backbone and adaptive components, and the few-shot window length. A more systematic sensitivity analysis and theoretical understanding under heterogeneous, non-IID client streams remain open. Future studies may explore principled sparsity schedules and automated budget selection.
- (3) Privacy and communication beyond sparse updates. While federated optimization reduces raw data exposure, stronger privacy guarantees (e.g., differential privacy, secure aggregation) and robustness to inference attacks may be required in practice. Future work will quantify privacy–utility–latency trade-offs under realistic networking conditions.
- (4) Full-stack deployment and hardware integration. Our hardware design addresses key bottlenecks induced by irregular sparsity and backward computations, but end-to-end integration (memory hierarchy, host interface, compiler/runtime co-optimization) and portability across edge platforms are not fully explored. Future work will pursue full-stack implementation and broader design-space exploration.

Author Contributions: Conceptualization and methodology, M.W.; techniques and experiments, Z.W. (Zhe Wen), X.C., R.X. and M.W.; supervision and resources, M.W., Z.W. (Zhongfeng Wang) and J.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Shenzhen Key Industries R&D Program (Grant No. ZDCY20250901112804006 and No. ZDCY20250901095901002), and High-Performance Computing Public Platform (Shenzhen Campus) of Sun Yat-sen University.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The market quote data used in this study were obtained through the Qlib data interface as described in the main text. To facilitate reproducibility, we provide the data cleaning and preprocessing procedures (e.g., filtering suspensions, handling missing values/NaNs, and the feature-construction pipeline) in the Experimental Setup section so that the workflow can be replicated on comparable public datasets. However, certain factor definitions and related proprietary components are subject to privacy and confidentiality restrictions and therefore cannot be publicly disclosed. Researchers may reproduce the experiments using the disclosed processing pipeline with an approximate dataset, and the corresponding author can be contacted for further clarification where permitted by institutional and legal constraints.

Acknowledgments: The authors would like to thank Sun Yat-sen University and Nanjing University for academic support and research resources. The authors also express their sincere gratitude to their families for their understanding, encouragement, and continuous support.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Kong, J.; Zhao, X.; He, W.; Yang, X.; Jin, X. EL-MTSA: Stock Prediction Model Based on Ensemble Learning and Multimodal Time Series Analysis. *Appl. Sci.* **2025**, *15*, 4669. [\[CrossRef\]](#)
2. Dželić, A.; Žunić, A.; Žunić Dželić, E. Predictive Modeling of Stock Prices Using Machine Learning: A Comparative Analysis of LSTM, GRU, CNN, and RNN Models. In *Proceedings of the International Symposium on Innovative and Interdisciplinary Applications of Advanced Technologies*; Springer: Berlin/Heidelberg, Germany, 2024; pp. 447–467.
3. Han, H.; Liu, Z.; Barrios Barrios, M.; Li, J.; Zeng, Z.; Sarhan, N.; Awwad, E.M. Time series forecasting model for non-stationary series pattern extraction using deep learning and GARCH modeling. *J. Cloud Comput.* **2024**, *13*, 2. [\[CrossRef\]](#)
4. Guo, Y.; Hu, C.; Yang, Y. Predict the Future from the Past? On the Temporal Data Distribution Shift in Financial Sentiment Classifications. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*; Bouamor, H., Pino, J., Bali, K., Eds.; Association for Computational Linguistics: Singapore, 2023; pp. 1029–1038. [\[CrossRef\]](#)
5. Wood, K.; Kessler, S.; Roberts, S.J.; Zohren, S. Few-shot learning patterns in financial time-series for trend-following strategies. *arXiv* **2023**, arXiv:2310.10500. [\[CrossRef\]](#)
6. Lin, J.; Zhu, L.; Chen, W.M.; Wang, W.C.; Gan, C.; Han, S. On-Device Training Under 256KB Memory. In *Advances in Neural Information Processing Systems (NeurIPS)*; MIT Press: Cambridge, MA, USA, 2022; Volume 35, pp. 22941–22954.
7. Zhang, Y.; Zhang, Y.; Peng, L.; Quan, L.; Zheng, S.; Lu, Z.; Chen, H. Base-2 Softmax Function: Suitability for Training and Efficient Hardware Implementation. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2022**, *69*, 3605–3618. [\[CrossRef\]](#)
8. Du, L.; Ni, L.; Liu, X.; Peng, G.; Li, K.; Mao, W.; Yu, H. A Low-Power DNN Accelerator with Mean-Error-Minimized Approximate Signed Multiplier. *IEEE Open J. Circuits Syst.* **2024**, *5*, 57–68. [\[CrossRef\]](#)
9. Chen, Y.; Zou, J.; Chen, X. April: Accuracy-Improved Floating-Point Approximation For Neural Network Accelerators. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*; IEEE: New York, NY, USA, 2025; pp. 1–7. [\[CrossRef\]](#)
10. Ahmed, M.P.; Tisha, S.A.; Sweet, M.R. Real-Time Hybrid Optimization Models for Edge-Based Financial Risk Assessment: Integrating Deep Learning with Adaptive Regression for Low-Latency Decision Making. *J. Bus. Manag. Stud.* **2025**, *7*, 38–52. [\[CrossRef\]](#)
11. Qin, M.; Sun, S.; Zhang, W.; Xia, H.; Wang, X.; An, B. Earnhft: Efficient hierarchical reinforcement learning for high frequency trading. In *Proceedings of the AAAI Conference on Artificial Intelligence, Vancouver, BC, Canada, 20–27 February 2024*; AAAI Press: Washington, DC, USA, 2024; Volume 38, pp. 14669–14676.
12. Chen, L.; Guo, K.; Fan, G.; Wang, C.; Song, S. Resource constrained profit optimization method for task scheduling in edge cloud. *IEEE Access* **2020**, *8*, 118638–118652. [\[CrossRef\]](#)
13. Kim, S.; Lee, J.; Kang, S.; Lee, J.; Jo, W.; Yoo, H.J. PNPU: An Energy-Efficient Deep-Neural-Network Learning Processor with Stochastic Coarse-Fine Level Weight Pruning and Adaptive Input/Output/Weight Zero Skipping. *IEEE Solid-State Circuits Lett.* **2021**, *4*, 22–25. [\[CrossRef\]](#)

14. Qi, C.; Liu, Y.; Chen, H.; Ge, F.; Liu, W. CIR-NoC: Accelerating CNN Inference Through In-Router Computation During Network Congestion. In *2025 International Symposium of Electronics Design Automation (ISED)*; IEEE: New York, NY, USA, 2025; pp. 29–34. [\[CrossRef\]](#)
15. Han, D.; Lee, J.; Yoo, H.J. DF-LNPU: A Pipelined Direct Feedback Alignment-Based Deep Neural Network Learning Processor for Fast Online Learning. *IEEE J. Solid-State Circuits* **2021**, *56*, 1630–1640. [\[CrossRef\]](#)
16. Zhu, P.; Li, Y.; Hu, Y.; Xiang, S.; Liu, Q.; Cheng, D.; Liang, Y. MCI-GRU: Stock Prediction Model Based on Multi-Head Cross-Attention and Improved GRU. *Neurocomputing* **2025**, *638*, 130168. [\[CrossRef\]](#)
17. Chen, S.; Ren, S.; Zhang, Q. Hybrid Architectures that Combine LLMs and Predictive Analytics for Next-Generation Financial Modeling. *Math. Model. Algorithm Appl.* **2025**, *6*, 31–43. [\[CrossRef\]](#)
18. Mao, W.; Liu, D.; Zhou, H.; Li, F.; Li, K.; Wu, Q.; Yang, J.; Cheng, Q.; Zhang, L.; Yu, H. A 28-nm 135.19 TOPS/W Bootstrapped-SRAM Compute-in-Memory Accelerator with Layer-Wise Precision and Sparsity. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2025**, *72*, 3236–3246. [\[CrossRef\]](#)
19. Chen, H.; Hao, Y.; Zou, Y.; Chen, X. OA-LAMA: An Outlier-Adaptive LLM Inference Accelerator with Memory-Aligned Mixed-Precision Group Quantization. In *2025 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*; IEEE: New York, NY, USA, 2025. [\[CrossRef\]](#)
20. Zhang, J.; Chen, X.; Song, M.; Li, T. Eager pruning: Algorithm and architecture support for fast training of deep neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*; IEEE: New York, NY, USA, 2019; pp. 292–303.
21. Lee, J.; Lee, J.; Han, D.; Lee, J.; Park, G.; Yoo, H.J. 7.7 LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*; IEEE: New York, NY, USA, 2019; pp. 142–144.
22. Kang, S.; Han, D.; Lee, J.; Im, D.; Kim, S.; Kim, S.Y.; Ryu, J.; Yoo, H. GANPU: An Energy-Efficient Multi-DNN Training Processor for GANs with Speculative Dual-Sparsity Exploitation. *IEEE J. Solid-State Circuits* **2021**, *56*, 2845–2857. [\[CrossRef\]](#)
23. Yang, D.; Ghasemazar, A.; Ren, X.; Golub, M.; Lemieux, G.; Lis, M. Procrustes: A dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*; IEEE: New York, NY, USA, 2020; pp. 711–724.
24. Tang, Y.; Zhang, X.; Zhou, P.; Hu, J. EF-train: Enable efficient on-device CNN training on FPGA through data reshaping for online adaptation or personalization. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*; Association for Computing Machinery: New York, NY, USA, 2022; Volume 27, pp. 1–36.
25. 강두석. Hardware-Aware Software Optimization Techniques for Convolutional Neural Networks on Embedded Systems. Ph.D. Thesis, 서울대학교 대학원, Seoul, Republic of Korea, 2021.
26. Paissan, F.; Nadalini, D.; Rusci, M.; Ancilotto, A.; Conti, F.; Benini, L.; Farella, E. Structured Sparse Back-propagation for Lightweight On-Device Continual Learning on Microcontroller Units. In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*; IEEE: New York, NY, USA, 2024; pp. 2172–2181. [\[CrossRef\]](#)
27. Zhao, Y.; Li, H.; Young, I.; Zhang, Z. Poor Man’s Training on MCUs: A Memory-Efficient Quantized Back-Propagation-Free Approach. *arXiv* **2024**, arXiv:2411.05873. [\[CrossRef\]](#)
28. Deutel, M.; Hannig, F.; Mutschler, C.; Teich, J. On-Device Training of Fully Quantized Deep Neural Networks on Cortex-M Microcontrollers. *arXiv* **2024**, arXiv:2407.10734. [\[CrossRef\]](#)
29. Nakahara, H.; Sada, Y.; Shimoda, M.; Sayama, K.; Jinguji, A.; Sato, S. FPGA-based training accelerator utilizing sparseness of convolutional neural network. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*; IEEE: New York, NY, USA, 2019; pp. 180–186.
30. Chen, Y.; Liu, Z.; Xu, H.; Darrell, T.; Wang, X. Meta-Baseline: Exploring Simple Meta-Learning for Few-Shot Learning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*; IEEE: New York, NY, USA, 2021; pp. 9062–9071.
31. Yue, Z.; Zhang, H.; Sun, Q.; Hua, X.S. Interventional few-shot learning. In *Advances in Neural Information Processing Systems (NeurIPS)*; MIT Press: Cambridge, MA, USA, 2020; Volume 33, pp. 2734–2746.
32. Li, T.; Liu, Z.; Shen, Y.; Wang, X.; Chen, H.; Huang, S. Master: Market-guided stock transformer for stock price forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*; AAAI Press: Washington, DC, USA, 2024; Volume 38, pp. 162–170.
33. Mazza, L. Coarse-Graining the Cross-Section: How Regression-via-Classification Improves Robustness in High-Noise, Small-Sample-Size Domains such as Cross-Sectional Asset Pricing. Master’s Thesis, KTH, School of Electrical Engineering and Computer Science, Stockholm, Sweden, 2024.
34. Fischer, T.; Krauss, C. Deep learning with long short-term memory networks for financial market predictions. *Eur. J. Oper. Res.* **2018**, *270*, 654–669. [\[CrossRef\]](#)
35. Jiang, J.; Yang, C.; Wang, X.; Li, B. Why Regression? Binary Encoding Classification Brings Confidence to Stock Market Index Price Prediction. *arXiv* **2025**, arXiv:2506.03153.

36. Finn, C.; Abbeel, P.; Levine, S. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks. In Proceedings of the 34th International Conference on Machine Learning, PMLR, Sydney, Australia, 6–11 August 2017; pp. 1126–1135.
37. Evci, U.; Gale, T.; Menick, J.; Castro, P.S.; Elsen, E. Rigging the lottery: Making all tickets winners. In Proceedings of the International Conference on Machine Learning, PMLR, Virtual, 13–18 July 2020; pp. 2943–2952.
38. Mrabah, N.; Richet, N.; Ben Ayed, I.; Granger, E. Sparsity Outperforms Low-Rank Projections in Few-Shot Adaptation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*; IEEE: New York, NY, USA, 2025; pp. 3143–3152.
39. Yang, X.; Liu, W.; Zhou, D.; Bian, J.; Liu, T.Y. Qlib: An AI-oriented Quantitative Investment Platform. *arXiv* **2020**, arXiv:2009.11189. [\[CrossRef\]](#)
40. Kakushadze, Z. 101 Formulaic Alphas. *Wilmott Mag.* **2016**, *84*, 72–80. [\[CrossRef\]](#)
41. Novy-Marx, R. The Other Side of Value: The Gross Profitability Premium. *J. Financ. Econ.* **2013**, *108*, 1–28. [\[CrossRef\]](#)
42. Asness, C.S.; Frazzini, A.; Pedersen, L.H. Quality Minus Junk. *Rev. Account. Stud.* **2019**, *24*, 34–112. [\[CrossRef\]](#)
43. Maingo, I.; Ravele, T.; Sigauke, C. A Fusion of Statistical and Machine Learning Methods: GARCH-XGBoost for Improved Volatility Modelling of the JSE Top40 Index. *Int. J. Financ. Stud.* **2025**, *13*, 155. [\[CrossRef\]](#)
44. Madhulatha, T.S.; Ghorri, M.A.S. Deep neural network approach integrated with reinforcement learning for forecasting exchange rates using time series data and influential factors. *Sci. Rep.* **2025**, *15*, 29009. [\[CrossRef\]](#)
45. Bieganski, B.; Ślepaczuk, R. Supervised autoencoder MLP for financial time series forecasting. *J. Big Data* **2025**, *12*, 207. [\[CrossRef\]](#)
46. Cheng, L.; Cheng, X.; Liu, S. Fast Learning in Quantitative Finance with Extreme Learning Machine. *arXiv* **2025**, arXiv:2505.09551. [\[CrossRef\]](#)
47. Wang, M.; Lu, S.; Zhu, D.; Lin, J.; Wang, Z. A high-speed and low-complexity architecture for softmax function in deep learning. In *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*; IEEE: New York, NY, USA, 2018; pp. 223–226.
48. Choi, S.; Sim, J.; Kang, M.; Choi, Y.; Kim, H.; Kim, L.S. A 47.4 $\mu\text{J}/\text{epoch}$ Trainable Deep Convolutional Neural Network Accelerator for In-Situ Personalization on Smart Devices. In *2019 IEEE Asian Solid-State Circuits Conference (A-SSCC)*; IEEE: New York, NY, USA, 2019; pp. 57–60.
49. Zhao, Y.; Li, C.; Wang, Y.; Xu, P.; Zhang, Y.; Lin, Y. DNN-chip predictor: An analytical performance predictor for DNN accelerators with various dataflows and hardware architectures. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*; IEEE: New York, NY, USA, 2020; pp. 1593–1597.
50. Lu, W.; Pei, H.H.; Yu, J.R.; Chen, H.M.; Huang, P.T. A 28nm Energy-Area-Efficient Row-based pipelined Training Accelerator with Mixed FXP4/FP16 for On-Device Transfer Learning. In *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*; IEEE: New York, NY, USA, 2024; pp. 1–5. [\[CrossRef\]](#)
51. Wang, Y.; Deng, D.; Liu, L.; Wei, S.; Yin, S. PL-NPU: An Energy-Efficient Edge-Device DNN Training Processor with Posit-Based Logarithm-Domain Computing. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2022**, *69*, 4042–4055. [\[CrossRef\]](#)
52. Venkataramanaiah, S.K.; Meng, J.; Suh, H.S.; Yeo, I.; Saikia, J.; Cherupally, S.K.; Zhang, Y.; Zhang, Z.; Seo, J.S. A 28-nm 8-bit Floating-Point Tensor Core-Based Programmable CNN Training Processor with Dynamic Structured Sparsity. *IEEE J. Solid-State Circuits* **2023**, *58*, 1885–1897. [\[CrossRef\]](#)
53. Qian, J.; Ge, H.; Lu, Y.; Shan, W. A 4.69-TOPS/W Training, 2.34- $\mu\text{J}/\text{Image}$ Inference On-Chip Training Accelerator with Inference-Compatible Backpropagation and Design Space Exploration in 28-nm CMOS. *IEEE J. Solid-State Circuits* **2025**, *60*, 298–307. [\[CrossRef\]](#)
54. Dey, S.; Chen, D.; Li, Z.; Kundu, S.; Huang, K.W.; Chugg, K.M.; Beerel, P.A. A highly parallel FPGA implementation of sparse neural network training. In *2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*; IEEE: New York, NY, USA, 2018; pp. 1–4.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.