Per-Axis Weight Deltas for Frequent Model Updates

Anonymous Author(s)
Affiliation
Address
email

Abstract

Serving many task-specialized LLM variants is often limited by the large size of fine-tuned checkpoints and the resulting cold-start latency. Since fine-tuned weights differ from their base model by relatively small structured residuals, a natural approach is to represent them as compressed deltas. We propose a simple 1-bit *delta* scheme that stores only the sign of the weight difference together with lightweight per-axis (row/column) FP16 scaling factors, learned from a small calibration set. This design preserves the compactness of 1-bit deltas while more accurately capturing variation across weight dimensions, leading to improved reconstruction quality over scalar alternatives. From a systems perspective, a streamlined loader that transfers packed deltas in a single operation per module reduces cold-start latency and storage overhead, with artifacts several times smaller than a full FP16 checkpoint. The method is drop-in, requires minimal calibration data, and maintains inference efficiency by avoiding dense reconstruction. Our experimental setup and source code are available at `https://anonymous.4open.science/r/Per-Axis-Weight-Deltas-for-Frequent-Model-Updates-0F1C/`.

1 Introduction

Large foundation models continue to grow in size and computational demand, making both training and deployment increasingly resource-intensive [Kaplan et al., 2020]. Once pre-trained, these models are often adapted to downstream tasks through fine-tuning. Depending on the setting, fine-tuning may involve updating all parameters with a supervised objective (full fine-tuning), applying low-rank updates as in LoRA [Hu et al., 2021] or other parameter-efficient fine-tuning methods [Houlsby et al., 2019, Ben Zaken et al., 2022, Mahabadi et al., 2021, Dettmers et al., 2023, Zhang et al., 2023, Liu et al., 2024b, Kopiczko et al., 2024], or reinforcement learning post-training, which can target either entire weight matrices or restricted subsets of parameters [Han et al., 2024]. In cases where fine-tunes are represented as full weight updates, serving multiple variants remains a deployment challenge. Each fine-tuned checkpoint must be stored and loaded in its entirety, and switching between them requires keeping large weight tensors resident in GPU memory. This is particularly costly for inference providers that serve many users or domains simultaneously, and for continual adaptation settings where new model variants are introduced frequently [Sheng et al., 2024, Chen et al., 2023]. Yet weights of fine-tuned models are rarely far from their base counterparts. Across a variety of adaptation procedures, the resulting weight matrices tend to differ from the pre-trained model only by relatively small residuals, both in magnitude and in spectral structure [Liu et al., 2024a]. This suggests that storing a full checkpoint per fine-tune is wasteful: the information required to recover the specialized model lies in a compact delta relative to the shared base. Prior work has demonstrated that such deltas can be compressed aggressively while still enabling accurate reconstruction of the fine-tuned model at inference time [Liu et al., 2024a]. However, they rely on coarse parametrizations that ignore variation in residual scales across rows or columns of weight matrices, leading to reconstruction errors that could be avoided with more structured representations.

Submitted to NeurIPS 2025 CCFM Workshop. Do not distribute.

At the same time, introducing too much precision or auxiliary metadata risks negating the efficiency benefits.

This paper introduces a *1-bit delta* representation—storing only the binary sign mask of the weight difference $\mathbf{B} = \operatorname{sign}(\mathbf{W_f} - \mathbf{W_b})$ and learning lightweight per-row/column scales—designed to balance those trade-offs: maintaining the simplicity and low storage overhead of delta compression, while adding lightweight per-axis scaling to better capture the axis-specific patterns in model weights. We show that this approach improves approximation quality at negligible extra cost, enabling faster and more memory-efficient serving of many fine-tuned variants from a single shared base model.

2 Method

We propose a parameter-efficient method for storing a fine-tuned model by leveraging its shared architecture with a base model. The core idea is to represent the output of fine-tuned weights as a sum of the base weights and a compressed residual term.

Let a model be composed of L layers. For layer i we have base and fine-tuned weights $W_b^{(i)}, W_f^{(i)} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$. We define $\Delta \mathbf{W}^{(i)} = \mathbf{W_f}^{(i)} - \mathbf{W_b}^{(i)}$ and the 1-bit sign mask $\mathbf{B}^{(i)} = \operatorname{sign}(\Delta \mathbf{W}^{(i)}) \in \{-1, +1\}^{d_{\text{out}} \times d_{\text{in}}}$. After that we patch via a per-axis broadcasted scale

$$\widehat{\mathbf{W}}^{i} = \mathbf{v}^{(i)} \odot \mathbf{B}^{(i)} + \mathbf{W_b}^{(i)}, \quad \mathbf{v}^{(i)} \in \begin{cases} \mathbb{R}^{1 \times d_{\text{out}}} & \text{(row)}, \\ \mathbb{R}^{d_{\text{in}} \times 1} & \text{(col)}, \end{cases}$$

where $\odot$ replicates $\mathbf{v}^{(i)}$ by columns (row mode) or rows (col mode); see Fig. 1.

This approach achieves significant compression. The storage cost per layer is reduced from floating-point weights to a single bitmask and a single vector. This enables the efficient storage of multiple fine-tuned models specialized for different tasks, all of which share the same underlying base weights.

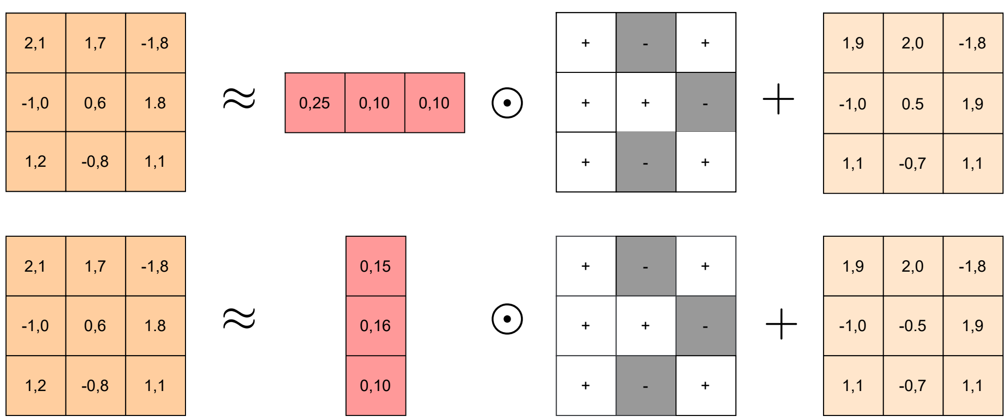

Figure 1: Approximating the fine-tuned weights $\mathbf{W_f}$ by $\mathbf{v} \odot \mathbf{B} + \mathbf{W_b}$: a compact 1-bit sign residual, where $\mathbf{v}$ is a vector, $\mathbf{B} \in \{-1, +1\}$ is the binary sign matrix, and $\mathbf{W_b}$ is the base weight matrix.

Prior evidence against weight reconstruction. The objective is not to recover the exact parameter values, but to preserve the function the network computes - i.e., to match outputs under realistic inputs. A line of works shows that minimizing weight-space error (e.g., round-to-nearest) is a weak surrogate for preserving model behavior: (i) Nagel et al. [2020] demonstrate that round-to-nearest is suboptimal and introduce *loss-aware* adaptive rounding that consistently outperforms weight-nearest at low bit widths; (ii) Frantar et al. [2023] explicitly minimize *layer-output* error (Hessian-aware) and report large gains over RTN on LLMs at 3–4 bits; (iii) Li et al. [2021] formulate *block reconstruction* of activations with a second-order analysis, enabling PTQ at 2 bits; (iv) Lin et al. [2024] argue that salient channels should be selected via *activation* statistics rather than weights; (v) Xiao et al. [2023] argue that while weights are relatively straightforward to quantize compared to activations, the difficulty can be mitigated by rescaling weights to absorb part of the activation complexity.

Calibration cache, training, and stacking. For each target layer i, the vector $\mathbf{v}^{(i)}$ is trainable while $\mathbf{W_b}^{(i)}$ and $\mathbf{B}^{(i)}$ are frozen at inference. We extract a small calibration set of 50 C4 [Raffel et al., 2023] samples and build a per-layer cache of $(\mathbf{X}, \mathbf{Y})$ pairs: $\mathbf{X}$ is the input that has to be passed to the i layer of the compressed model (i.e., the output of the already-compressed stack up to layer $i-1$, immediately before entering layer i), and $\mathbf{Y}$ is the fine-tuned outputs of the original none compressed finetuned layer, while $\hat{\mathbf{Y}}$ denotes the output produced by compressed layer. We attach forward hooks to the teacher to collect $\mathbf{Y}$ and to the student to collect $\mathbf{X}$, store both as BF16 tensors. For each target layer i we instantiate both axis variants and fit only their scale vectors on the cache with an MSE objective,

$$\mathcal{L}_{\text{layer}} = \frac{1}{n} \left\| \mathbf{Y} - \hat{\mathbf{Y}} \right\|_2^2,$$

using AdamW for 5 epochs under the same budget across variants. The axis is selected by validation MSE on the held-out shard, and the original layer is replaced with the better variant. We sweep all linear projections in attention and MLP blocks and install the selected module per layer, yielding a compressed student stacked on top of the shared base.

Implementation remarks. We run on Llama-3.1-8B, using Llama-3.1-8B-Instruct as the teacher and Llama-3.1-8B as the student. Due to limited VRAM, we used two RTX 4090 GPUs and split fine-tuned weights and compressed weights across devices. We cache teacher layer outputs (fine-tuned, `cuda:0`) and student inputs (compressed, `cuda:1`) via forward hooks as detached BF16 clones stored on `cuda:1`. Masks $\mathbf{B}^{(i)}$ stay packed end-to-end (1 bit along input axis), vectors $\mathbf{v}^{(i)}$ are FP16, and base weights are kept as (in, out) BF16. We use non-blocking transfers and a single `.to(device)` per module. The full algorithm can be seen in 4.

3 Experiments

3.1 Setup

We adopt a simple evaluation setting: Llama-3.1-8B as the base model and Llama-3.1-8B-Instruct as the fine-tuned target, evaluated zero-shot on ARC-Challenge, ARC-Easy [Clark et al., 2018], HellaSwag [Zellers et al., 2019], PIQA [Bisk et al., 2019], and Winogrande [Sakaguchi et al., 2019]. Unless noted otherwise, all methods use the same calibration budget of 150 samples drawn from C4 [Raffel et al., 2023].

For our vector scales we use AdamW, learning rate 1×10^{-5}, for five epochs; BitDelta (scalar) uses the same pipeline but with a single scalar per matrix and one epoch for training. Unless stated otherwise, we report zero-shot accuracy (%) on the public test splits using the same prompt formatting across methods.

For additional descriptive analysis of the selected delta-quantization axis, see Appendix A; per-sub-type counts and layer-wise trends are shown in Figure 2.

Models and baselines. Baseline denotes the fine-tuned model without any delta compression. BitDelta (scalar) is the 1-bit sign mask with a single learned scalar per matrix. Our method is with a 1-bit sign mask and a learned per-row or per-column vector of scales.

3.2 Main results

Table 1 summarizes *zero-shot accuracy* on ARC-Challenge, ARC-Easy, HellaSwag, PIQA, and Winogrande—using Llama-3.1-8B as the base and Llama-3.1-8B-Instruct as the fine-tuned target. Vector (row/col) improves the average score over the baseline by 0.97 points and over *BitDelta* (scalar) by 0.28 points. Gains are consistent on ARC-Challenge/Easy and Winogrande; HellaSwag is on par, while PIQA shows a small drop versus BitDelta. See Appendix A for a breakdown by module sub-type (Figure 2)

Storage and load-time. Our delta representation stores the fine-tuned model as a compact $\sim 3\,\text{GB}$ artifact on disk for the 8B setting (Table 2)—about $5.4\times$ smaller than a full FP16 checkpoint. Under identical allocator/seeds and cold-start conditions on LLAMA-3.1-8B, the average load time over 10 runs to apply the vector–delta on top of the base is $0.80\,\text{s}$, whereas loading the entire fine-tuned FP16

Table 1: Zero-shot accuracy (%) after calibrating on 150 samples from C4. Vector scales are trained for five epochs with learning rate $1\mathrm{e}{-5}$; BitDelta uses the same setup with a single scalar per matrix.

Model	ARC-C	ARC-E	HellaSwag	PIQA	Winogrande	Avg
Baseline	51.70	81.81	59.06	79.86	73.87	69.26
BitDelta (scalar)	52.55	82.32	59.73	**81.22**	73.95	69.95
Vector (row/col)	**53.58**	**82.99**	**59.78**	80.63	**74.19**	**70.23**

Table 2: Checkpoint sizes: A full 8B FP16 checkpoint is 14.9 GiB so both deltas are $5.4\times$ smaller.

Artifact	Size (MB)	Size (MiB)	vs. FP16 8B weights
Scalar	2974	2836	$\approx 5.25\times$ smaller
Vector	2980	2842	$\approx 5.24\times$ smaller

checkpoint takes 2.08 s. Thus the delta path uses less per-model load time for a much smaller on-disk and transfer footprint per-model. This is especially useful when maintaining or hot-swapping many fine-tuned versions of a given base model.

4 Limitations

When a scalar suffices Our gains rely on the anisotropy of the task-induced deltas $\mathbf{\Delta W}$ across rows/columns. If a layer's delta is nearly isotropic, a single global scale can match quality while avoiding the metadata and compute introduced by per-row/column vectors.

Fixed 1-bit signs (no sparsity) We fix $\mathbf{B} \in \{-1, +1\}^{d_{\text{in}} \times d_{\text{out}}}$ at 1 bit per entry. This forbids explicit zeros/sparsity and can propagate noise for very small-magnitude entries unless one adds debiasing or confidence filtering. Consequently, the patch is dense and incurs slight additional MACs (extra steps) and memory overhead compared to a pure binary (sign-only) matrix.

Calibration dependence Vector scales are learned with an activation-aware objective using a small calibration set to estimate C_x. Distribution shift between calibration and deployment may reduce effectiveness; larger or stratified calibration improves robustness but increases preparation time and memory.

Layer coverage We patch linear projections (attention and MLP). We do not modify normalizations, some biases, or tied embeddings; if task-specific changes concentrate there, our method may yield limited benefits.

No mask learning The sign mask $\mathbf{B}$ is fixed and we do not learn signs or structure. At aggressive bit budgets, learning $\mathbf{B}$ may be beneficial for downstream performance.

5 Conclusion

We introduced a 1-bit delta scheme with lightweight per-axis (row/column) FP16 scales learned via output matching. Empirically, across five zero-shot benchmarks, the method attains an average accuracy of 70.23 vs. 69.95 for a scalar 1-bit delta and 69.26 for the uncompressed baseline. Limitations include layers with near-isotropic deltas and reliance on small calibration sets. Future work includes blockwise per-group scaling, learning the sign structure, INT4/FP8 co-design, and broader multi-tenant evaluations.

Our method delivers higher average accuracy than both the baseline and the scalar BitDelta while preserving the same storage efficiency. From a systems perspective, our loader reduces cold-start latency. Overall, vector scales provide a better match to the anisotropy of task deltas at negligible extra storage cost.

References

Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. Bitfit: Simple parameter-efficient fine-tuning for transformers. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Association for Computational Linguistics, 2022. doi: 10.18653/v1/2022.acl-short.1. URL `https://aclanthology.org/2022.acl-short.1`.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. 2019. URL `https://arxiv.org/abs/1911.11641`.

Lequn Chen, Zihao Ye, Yongji Wu, Danyang Zhuo, Luis Ceze, and Arvind Krishnamurthy. Punica: Multi-tenant lora serving. 2023. URL `https://arxiv.org/abs/2310.18547`.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. 2018. URL `https://arxiv.org/abs/1803.05457`.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. 2023. URL `https://arxiv.org/abs/2305.14314`.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers, 2023. URL `https://arxiv.org/abs/2210.17323`.

Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey, 2024. URL `https://arxiv.org/abs/2403.14608`.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. *arXiv preprint arXiv:1902.00751*, 2019. URL `https://arxiv.org/abs/1902.00751`.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. 2021. URL `https://arxiv.org/abs/2106.09685`.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. 2020. URL `https://arxiv.org/abs/2001.08361`.

Dawid J. Kopiczko, Tijmen Blankevoort, and Yuki M. Asano. Vera: Vector-based random matrix adaptation. 2024. URL `https://arxiv.org/abs/2310.11454`.

Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. Brecq: Pushing the limit of post-training quantization by block reconstruction, 2021. URL `https://arxiv.org/abs/2102.05426`.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration, 2024. URL `https://arxiv.org/abs/2306.00978`.

James Liu, Guangxuan Xiao, Kai Li, Jason D Lee, Song Han, Tri Dao, and Tianle Cai. Bitdelta: Your fine-tune may only be worth one bit. *Advances in Neural Information Processing Systems*, 37: 13579–13600, 2024a.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation. 2024b. URL `https://arxiv.org/abs/2402.09353`.

Rabeeh Karimi Mahabadi, James Henderson, and Sebastian Ruder. Compacter: Efficient low-rank hypercomplex adapter layers. 2021. URL `https://arxiv.org/abs/2106.04647`.

Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? Adaptive rounding for post-training quantization. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7197–7206. PMLR, 13–18 Jul 2020. URL `https://proceedings.mlr.press/v119/nagel20a.html`.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. 2023. URL `https://arxiv.org/abs/1910.10683`.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. 2019. URL `https://arxiv.org/abs/1907.10641`.

Ying Sheng, Shiyi Cao, Dacheng Li, Coleman Hooper, Nicholas Lee, Shuo Yang, Christopher Chou, Banghua Zhu, Lianmin Zheng, Kurt Keutzer, Joseph E. Gonzalez, and Ion Stoica. S-lora: Serving thousands of concurrent lora adapters. 2024. URL `https://arxiv.org/abs/2311.03285`.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 38087–38099. PMLR, 23–29 Jul 2023. URL `https://proceedings.mlr.press/v202/xiao23c.html`.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? 2019. URL `https://arxiv.org/abs/1905.07830`.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. 2023. URL `https://arxiv.org/abs/2303.10512`.

A Additional analysis of delta-quantization axis

This appendix provides descriptive statistics for the learned choice of the delta-quantization axis (`row` vs. `column`) across module sub-types and depth.

Counts by sub-type. Figure 2 summarizes how often each sub-type selects a `row` or a `column` axis for delta quantization. Overall, attention projections (`q`, `v_proj`, `o_proj`) and the MLP `down_proj` tend to prefer `row`, while `gate_proj` and `up_proj` show a stronger `column` preference, with `k_proj` being more mixed. These tendencies are consistent with the differing input/output aspect ratios of the corresponding weight matrices.

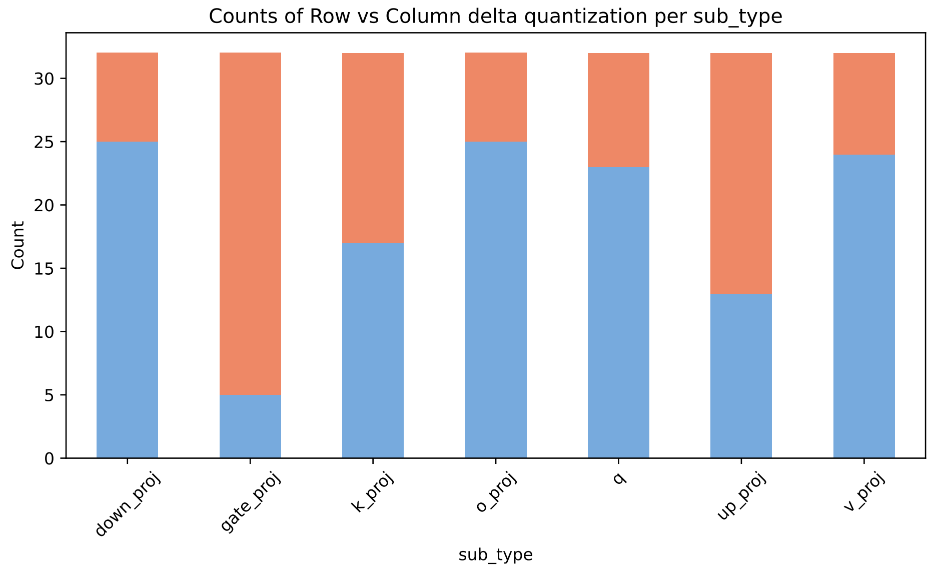

Figure 2: counts of `row` vs. `column` delta-quantization per sub_type (row in blue, column in red).

Algorithm 1 Register and use forward hooks to build calibration caches for a layer L

Require: Teacher model W_f on `cuda:0`, student $\widehat{W}$ on `cuda:1`, target layer name L, train loader $\mathcal{D}_{\text{tr}}$, val loader $\mathcal{D}_{\text{val}}$, train steps T, eval steps E
Ensure: Caches $(X_{\text{tr}}, Y_{\text{tr}})$ and $(X_{\text{val}}, Y_{\text{val}})$ on `cuda:1`

Initialize empty maps $\textsc{Inputs}[L]$, $\textsc{Outputs}[L]$ ▷ device=`cuda:1`, dtype=BF16
$h_{\text{out}} \leftarrow$ register forward hook on $W_f[L]$ that appends *detached BF16 output* to $\textsc{Outputs}[L]$ on `cuda:1`
$h_{\text{in}} \leftarrow$ register forward hook on $\widehat{W}[L]$ that appends *detached BF16 input* to $\textsc{Inputs}[L]$ on `cuda:1`
for $t = 1$ to T **do** ▷ build train cache
  Fetch batch $b \leftarrow \mathcal{D}_{\text{tr}}$
  Run W_f on b (moved to `cuda:0`, no grad) ▷ fills $\textsc{Outputs}[L]$
  Run $\widehat{W}$ on b (moved to `cuda:1`, no grad) ▷ fills $\textsc{Inputs}[L]$
end for
for $e = 1$ to E **do** ▷ build val cache
  Repeat the two forwards with $\mathcal{D}_{\text{val}}$
end for
Remove hooks h_{out}, h_{in}
$X_{\text{tr}}, Y_{\text{tr}} \leftarrow$ first T items of $\textsc{Inputs}[L], \textsc{Outputs}[L]$
$X_{\text{val}}, Y_{\text{val}} \leftarrow$ last E items of $\textsc{Inputs}[L], \textsc{Outputs}[L]$
return $(X_{\text{tr}}, Y_{\text{tr}}), (X_{\text{val}}, Y_{\text{val}})$

Algorithm 2 Train per-row/column scaling vectors via activation matching

---

Require: Compressed layer M (ROW or COL) with learnable α, train cache $(X_{\text{tr}}, Y_{\text{tr}})$, val cache $(X_{\text{val}}, Y_{\text{val}})$, epochs K, learning rate η
Ensure: Trained α and validation loss L_{val}

Initialize AdamW on α only with LR η; optional cosine scheduler over K epochs
for $k = 1$ to K **do**
  Train: For each minibatch $(x, y) \in (X_{\text{tr}}, Y_{\text{tr}})$:
    $y_{\text{pred}} \leftarrow M(\textsc{ReshapeForLayer}(x))$ ▷ reshape to layer input shape if needed
    $L \leftarrow \|y_{\text{pred}} - y\|_2^2$; backprop only through α; optimizer step; scheduler step
end for
Validate: $L_{\text{val}} \leftarrow$ mean of $\|M(\textsc{ReshapeForLayer}(x)) - y\|_2^2$ over $(X_{\text{val}}, Y_{\text{val}})$ (no grad)
return (α, L_{val})

Algorithm 3 End-to-end validation loss (student vs. cached teacher logits)

Require: Student model $\widehat{W}$ on `cuda:1`, val loader $\mathcal{D}_{\text{val}}$, cached teacher logits $\{\ell_t^*\}$ aligned by batch
Ensure: Scalar validation loss L_{end}

$L_{\text{end}} \leftarrow 0, n \leftarrow 0$
for first N batches b in $\mathcal{D}_{\text{val}}$ **do**
    Move b to `cuda:1`, run $\widehat{W}$ under AMP to get logits ℓ
    $L_{\text{end}} \leftarrow L_{\text{end}} + \|\ell - \ell_b^*\|_2^2; n \leftarrow n + 1$
end for
return L_{end}/n

Algorithm 4 Per-layer compression with Row/Col selection by end loss

Require: Base weight $W_b^{(L)}$, fine-tuned $W_f^{(L)}$, layer name L, loaders $\mathcal{D}_{\text{tr}}, \mathcal{D}_{\text{val}}$
Ensure: Replace layer L with the better of ROW/COL

$\Delta W \leftarrow W_f^{(L)} - W_b^{(L)}; \quad B \leftarrow \textsc{Pack}(\text{sign}(\Delta W)^\top)$
$(X_{\text{tr}}, Y_{\text{tr}}), (X_{\text{val}}, Y_{\text{val}}) \leftarrow$ Alg. 1 for L
Build **Col** module $M_{\text{col}}(B, \alpha_c)$ with $\alpha_c \leftarrow \text{mean}(|\Delta W|, \text{axis} = 1)$; train via Alg. 2 with LR 1×10^{-4}, epochs 5
$E_{\text{col}} \leftarrow$ Alg. 3 on $\widehat{W}$ after swapping in M_{col}
Build **Row** module $M_{\text{row}}(B, \alpha_r)$ with $\alpha_r \leftarrow \text{mean}(|\Delta W|, \text{axis} = 0)$; train via Alg. 2 with LR 1×10^{-5}, epochs 5
$E_{\text{row}} \leftarrow$ Alg. 3 on $\widehat{W}$ after swapping in M_{row}
if $E_{\text{row}} \leq E_{\text{col}}$ **then**
    $\textsc{ReplaceLayer}(L \leftarrow M_{\text{row}})$
else
    $\textsc{ReplaceLayer}(L \leftarrow M_{\text{col}})$
end if

Algorithm 5 Model-wide application from a saved delta file (row/col-aware)

Require: Student model $\widehat{W}$, delta dict diff (keys: `.mask_row`, `.coeff_row`, `.mask_col`, `.coeff_col`)

for all modules $(\text{name}, \text{mod})$ in $\widehat{W}$ where $\text{NameContains}(\text{name}, \{\text{mlp}, \text{self\_attn}\})$ and $\text{NameContains}(\text{subname}, \{\text{proj}\})$ **do**
    if diff has name+.mask_row **then**
        $\textsc{CompressLayerRow}(\text{name}, \text{diff})$
    else if diff has name+.mask_col **then**
        $\textsc{CompressLayerCol}(\text{name}, \text{diff})$
    end if
end for
Optionally compute L_{end} via Alg. 3