MODEL CASTING & LOW-PARAMETER GATING: TOWARDS MORE SPARSELY ACTIVATED FFNS

Anonymous authors
Paper under double-blind review

ABSTRACT

This paper introduces *model casting*, a mid-training recipe that drastically sparsifies the activations within the Feed-Forward Network (FFN) layer. With this strategy, at inference time, we first compute the output of the gating matrix and, thanks to its high sparsity, we avoid computations with the two other matrices, reducing the FLOP count by up to $3\times$. While this theoretical speedup is an upper bound, model casting translates into significant speedups both on CPU and GPU.

We then introduce LoPA Gating, a new FFN design that increases the maximum theoretical speedup. It is a low-FLOPs parameterization of the gating matrix that overcomes the $3\times$ cap by allocating fewer FLOPs and parameters to the gating matrix, compared to the two other FFN matrices that are sparsely activated.

We consider two cases: (i) we cast a pre-trained model with a sparsity-inducing activation; (ii) we train with LoPA from scratch. In all settings, we significantly outperform existing pruning solutions and regular `RELU`-fication. For instance, at matched quality, we achieve a $3.2\times$ FLOP speedup with LoPA Casting, against $1.6\times$ at best for competing methods top-p and TEAL. Using dedicated kernels, we achieve an actual $3.31\times$ speed-up on GPU at 90% sparsity, past the $3\times$ ceiling of standard gating; `RELU`-fication, meanwhile, plateaus below 80% sparsity.

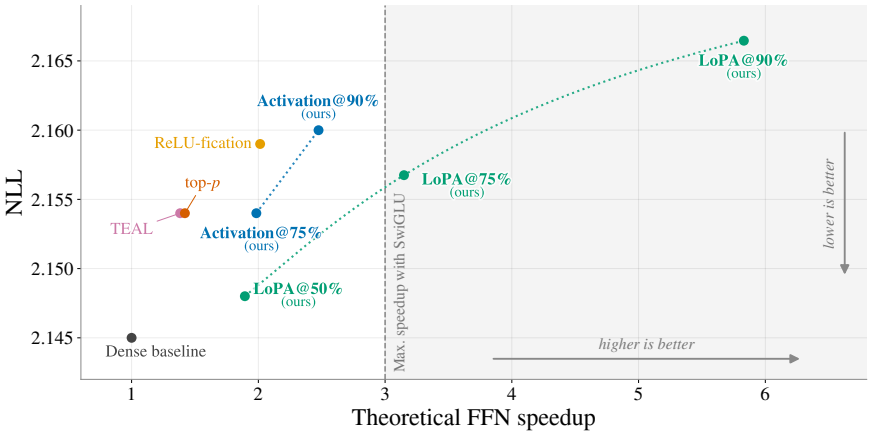


Figure 1: **Model casting speedup at comparable quality,** *vs* the baseline dense model (reference $\times 1$ and NLL = 2.145) and existing sparsification techniques: `RELU`-fication and the training-free top-p and TEAL. At a 90% sparsity target, LoPA Casting provides a $5.8\times$ speed-up for a 0.93% NLL increase over the dense baseline. See detailed results in Tables 1 and 2, and practical speed-up in Section 5.3.

1 INTRODUCTION

In current Large Language Model (LLM) architectures (Vaswani et al., 2017), there exist two distinct memorization mechanisms. One is the short-term memory, determined by the sequence at hand and instantiated as a sequence of keys and values produced in the attention mechanism. The other one is the fixed parametric memory, which is populated at training time. This memory is primarily contained in the part of the model that has the most parameters: the feed-forward (FFN) layers.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Both these memories involve some significant computational complexity. In particular, reducing the quadratic complexity of the attention mechanism is a major line of research in LLM optimization. Although the FFN's inference cost is independent of sequence length, for large models it dominates the complexity in most settings, unless the input context is extremely long.

In this paper, we focus on optimizing FFN inference speed. We show how to fine-tune a pre-trained LLM so that its FFN post-activations become sparse enough to yield actual inference speedups, against the popular belief that unstructured activation sparsity is unexploitable. Our approach, which combines two main ingredients, trades large speedups for a much lower decrease in quality than existing methods, see Figure 1. Specifically, our contributions are as follows:

- A mid-training strategy combining an activation swap with a sparsity-inducing loss, which effectively sparsifies FFN blocks without degrading model quality. This method is straightforward to implement: we replace the `SILU` activation with `R-S+`, a `RELU`-derived variant, and add an adaptive ℓ_1 penalty. Unlike prior methods, the sparsity level is a target we set: we reach 90% sparsity at a small NLL cost.
- A new FFN design based on an efficient parameterization of W_1 that is combined with the activation swap strategy. It lowers the FFN FLOPs floor, enabling further efficiency gains.
- Custom CPU and GPU kernels showing that the induced sparsity is practically exploitable: on a 5.8B-parameter model at 90% sparsity, we obtain $1.53\times$ end-to-end CPU and $2.06\times$ FFN GPU speedups with standard gating, increasing to $1.88\times$ and $3.31\times$ when casting is combined with LoPA Gating, and to $3.90\times$ on GPU at 95% sparsity.
- Experiments showing that our approach is not tied to specific pre-trained models: model casting translates to Mixture-of-Experts, casting either the expert FFN blocks or the router that selects active experts. It also transfers to an off-the-shelf base model Qwen3-1.7B.

2 Related Work

Emergent and trained-in activation sparsity. When a sparse activation such as `RELU` is used, trained transformers produce sparse intermediate activations without this being an explicit design choice (Li et al., 2023). Luo et al. (2025) show that the effect follows a power law in the amount of training data and that it is largely insensitive to model scale. The switch to `SILU`-gated designs (Dauphin et al., 2017; Shazeer, 2020) traded that property away: the `SILU`-gated FFNs that are now standard (Jiang et al., 2023; Riviere et al., 2024; Touvron et al., 2023a) have better performance but produce no exact zeros to exploit. Conversely, pre-training from scratch with `RELU` instead risks permanently dead neurons. Recovering the sparsity afterwards is often treated during a mid-training stage: Mirzadeh et al. (2024) swap the activation to `RELU` when fine-tuning `GELU` or `SILU` models (Zhang et al., 2022; Touvron et al., 2023b), and report much lower sparsity for gated FFNs than for non-gated ones. Lomeli et al. (2025) prepare the network during pre-training by stochastically switching between `RELU` and `SILU` for negative values. In both cases, the attainable sparsity is determined by the activation swap rather than directly controlled. In contrast, with model casting, we explicitly set the desired sparsity.

Post-hoc sparsification. A complementary family exploits sparsity at inference time without retraining the base model: predictor-based *contextual sparsity*, which selects an input-dependent subset of heads and neurons and underlies serving systems such as PowerInfer (Liu et al., 2023b; Song et al., 2023); magnitude thresholding across all attention and MLP matrices, as in TEAL (Liu et al., 2025a) and top-p (Szatkowski et al., 2026); calibrated per-layer thresholding of the `SILU` gate itself (Lee et al., 2024); and transform-then-threshold variants that first make the activations more compressible (Zhang et al., 2025; Liu et al., 2025b). Such methods are bounded by the sparsity a pre-trained model happens to admit. For instance, Szatkowski et al. (2026) report a *critical sparsity*, the level reachable at 1% quality degradation, of at most 40% for an 8B model. None of the existing methods can prune the always-on gate, which is the cost that our LoPA Gating proposal removes. The predictor-based schemes additionally target `RELU` models, where the MLP sparsity is already present, rather than the non-sparse gated FFNs we consider. We compare against TEAL, top-p and `RELU`-fication in Section 5.2; their implementation details are provided in Appendix B.

Sparsity and efficiency by architecture. A third line of work changes the parameterization rather than the activation. Since LoRA (Hu et al., 2021), low-rank factorizations have been used well

beyond adaptation: to pre-train the FFN block with structured matrices (Wei et al., 2024b) or on the low-rank manifold of the product (Mo et al., 2025), as a sum of low-rank and sparse matrices that is not restricted to a low-rank structure (Han et al., 2024), and as the diagonal-plus-low-rank structure of Chen et al. (2026), which restores the universal approximation property. LoPA Gating (Section 3.3) belongs to this family, but is used to make the *gate* cheap rather than the whole block small. Mixture-of-Experts models (Yang et al., 2024; Wei et al., 2024a; DeepSeek-AI et al., 2024; Jiang et al., 2024) instead place the sparsity in the routing, activating a subset of experts per token, conventionally through a softmax followed by top-k selection (DeepSeek-AI et al., 2024); Wang et al. (2025) make that selection differentiable with `RELU` and an ℓ_1 loss. The two views meet: Lee et al. (2024) observe that the `SILU` gate of a dense FFN can be read as a MoE-style router over hidden units, which is why the same casting mechanism applies to the FFN gate and to the MoE router alike, see Appendix C for further details.

3 METHOD

Model casting refers to changing part of a pre-trained model's architecture midway through training and continuing to train so that the model adapts to the change. We consider three distinct setups:

- **Activation Casting**: the model is pre-trained with a full matrix W_1 (Standard Gating); at casting time, we replace the non-sparse activation by the new `R-S+` activation (Section 3.1), and mid-train the new model to magnify sparsity (Section 3.2).
- **LoPA Casting**: the model is pre-trained with the new LoPA Gating parameterization, see Section 3.3. Then we apply activation casting as in the first setup. The resulting model has the same number of parameters but is faster at inference. However, this setting requires the model to be trained from scratch with the LoPA Gating architecture.
- **Architecture Casting**: we start from a pre-trained off-the-shelf model with Standard Gating, see Section 3.4. At casting time, the full W_1 matrix is converted to LoPA Gating and Activation Casting is applied at the same time.

3.1 PRELIMINARIES

Gated FFN. Transformers (Vaswani et al., 2017) have a number of blocks consisting of attention and feedforward components. We focus on FFN components with gated design, where, on an input $x \in \mathbb{R}^D$, a gated FFN of hidden dimension $H > D$ computes as follows:

$$y = W_2 \times (A(W_1 \times x) \odot (W_3 \times x)) \text{ with } W_1, W_3 \in \mathbb{R}^{H \times D} \text{ and } W_2 \in \mathbb{R}^{D \times H}, \tag{1}$$

where we assume column vectors. The symbol $\odot$ denotes element-wise multiplication. The activation A is either non-sparse, such as `SILU` or `GELU`, or sparse, such as `RELU`. We consider single input vectors x, i.e. we do not batch inputs.

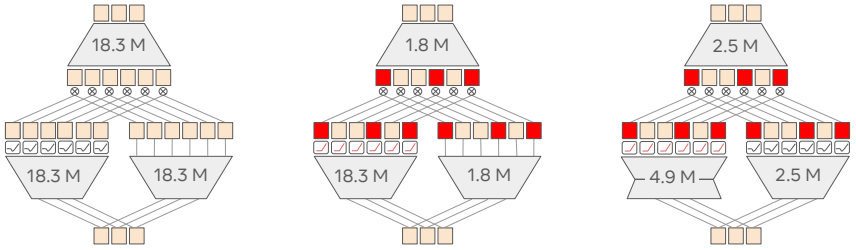
`R-S+` activation. We consider the activation `R-S+` defined in the adjacent table, where $\sigma(x) = 1/(1 + \exp(-x))$ is the sigmoid function. It is a hybrid of `RELU` and `SILU`, equal to `SILU` on the positive side, thereby reducing the discrepancy introduced when casting the activation from `SILU`, while providing the sparsification behavior of `RELU` for $x < 0$.

	$x < 0$	$x \geq 0$
`RELU`	0	x
`SILU`	$x \cdot \sigma(x)$	$x \cdot \sigma(x)$
`R-S+`	0	$x \cdot \sigma(x)$

FFN FLOP-ratio. Each of the three matrices costs HD multiply-adds per token, so a dense FFN costs $3HD$. A zero entry is exploitable only if it is known before the matmul it is fed to: a zero in the gate $A(W_1x)$ is known after the W_1x computation and before W_3, so it lets us skip one row of W_3 and one column of W_2, while W_1 is always computed in full. Writing $\text{cost}(W)$ for the multiply-adds actually executed, we measure the reduction factor of FLOPs as

$$\text{FLOP-ratio} = \frac{3HD}{\text{cost}(W_1) + \text{cost}(W_2) + \text{cost}(W_3)} \tag{2}$$

with standard gating and a sparsity fraction $s \in [0, 1]$, $\text{cost}(W_1) = HD$ and $\text{cost}(W_2) = \text{cost}(W_3) = (1 - s)HD$, giving FLOP-ratio $= 3/(3 - 2s) \in [1, 3]$.



(a) Dense SwiGLU **(b)** Sparse SwiGLU **(c)** Sparse LoPA Gating

Figure 2: **Fewer FLOPs with sparsity and LoPA Gating.** FLOPs per linear layer at $D = 2560$, $H = 7168$ and rank $r = 400$, with 90% sparse R-S+ gates. In (c), the hidden dimension grows from H to $H' = 9787$ to keep a constant parameter count. The FLOPs reductions are due to sparsity in (b,c) and to LoPA Gating in (c).

3.2 Activation casting

Activation casting is our simplest proposal: We start from a checkpoint taken at a stage where the learning rate is stable, and replace the SILU activation of every FFN by R-S+. We then resume the training with a short warm-up and a stable phase so that the model adapts to the new activation, then decay the learning rate as usual. During this mid-training phase, we add the ℓ_1 loss term to magnify sparsity. Since R-S+ coincides with SILU for $x \geq 0$, model casting only perturbs the negative half of the activation. The alternative RELU-fication (Mirzadeh et al., 2024), by contrast, demands an adaptation on the whole data range, which has a larger impact on performance.

Increasing sparsity with a regularizer. We measure the FFN sparsity (after mid-training with model casting) as the proportion of zeros in the post-activation gate output. Lomeli et al. (2025) report more than 90% sparsity with their *stochastic activations* trained from scratch. However, they observed much lower sparsities when this stochastic activation is applied mid-training.

Hence, we further increase the sparsity by adding an ℓ_1 sparsity loss for the hidden activations:

$$\mathcal{L} = \mathcal{L}_{\text{CE}} + \frac{\lambda}{H} \sum_{\ell=1}^{L} \left| \mathbf{h}^{(\ell)} \right|_1$$

where $\mathcal{L}_{\text{CE}}$ is the standard per-token cross-entropy loss, $\mathbf{h}^{(\ell)} \in \mathbb{R}^H$ is the FFN activation at layer ℓ, i.e. $A(W_1 x)$, and $|\cdot|_1$ is the ℓ_1 norm (sum of absolute values). Thus, there is a trade-off: when λ increases, the activation is sparser, but $\mathcal{L}_{\text{CE}}$ increases, degrading the model's generation quality.

Adaptive ℓ_1 loss. The relationship between the weight λ and the sparsity s depends on the model scale; see Appendix D. To set s, we adopt an *adaptive* ℓ_1 regularization (Wang et al., 2025): given a target sparsity $s^\star$, λ is initialized at λ_0 and at training step i it is updated as $\lambda_{i+1} = \lambda_i \cdot \beta^{\text{sign}(s^\star - s_i)}$, where s_i denotes the average sparsity of all activations at step i and $\beta > 1$ is a hyperparameter. Hence, λ increases if the sparsity is below the target and vice versa. We set $\lambda_0 = 10^{-3}$ and $\beta = 1.03$.

3.3 LoPA Gating

The maximum theoretical FLOP-ratio is 3 (Equation 2 with $s = 1$), as we need to compute $W_1 \times x$ to know where the zeros are. We propose an architecture change to the FFN blocks to reduce this cost: we make cost(W_1) cheaper by decomposing it and reducing its number of parameters. To keep the model's capacity fixed, we redistribute the saved parameters to W_3 and W_2, see Figure 2(c).

LoPA Gating parameterization. We represent W_1 as:

$$W_1 = G + UV \text{ where } U \in \mathbb{R}^{H \times r} \text{ and } V \in \mathbb{R}^{r \times D}, \tag{3}$$

where $G \in \mathbb{R}^{H \times D}$ is a non-trainable matrix with a single 1 on each row, and 0 elsewhere, see Figure 3. We call the full-rank W_1 version *Standard Gating* to distinguish it from LoPA Gating.

Adding the matrix G prevents W_1 from becoming rank-deficient, similar to the diagonal-plus-low-rank parameterization of Chen et al. (2026), for which a full-rank additive term restores the universal

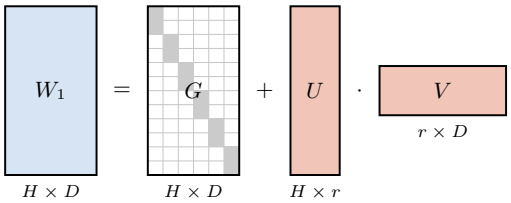


Figure 3: **LoPA Gating parameterization**. The gating matrix is decomposed as $W_1 = G + UV$, U and V are low-rank trainable matrices and G is a non-trainable sparse matrix. The resulting W_1 is full-rank, yielding much better results than $W_1 = UV$ only, with the same number of FLOPs and parameters.

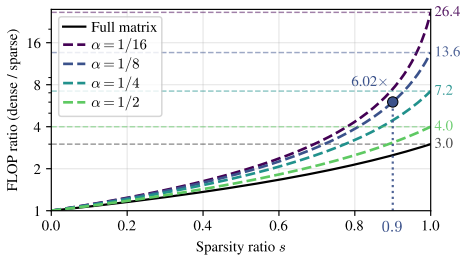


Figure 4: **FLOP-ratio for LoPA Gating**, as a function of the rank ratio $\alpha = r/D$ and of the sparsity ratio s. At $\alpha = 1/8$ and 90% sparsity, the FFN FLOPs are reduced by $6.02\times$.

approximation property of the FFN. Our G generalizes that diagonal to any matrix with a single non-zero per row. The design is justified empirically in Appendix A: LoPA Gating outperforms a low-rank gate ($W_1 = UV$), and any fixed matrix with a single ± 1 per row performs equally well. Architecture Casting, described in the following, exploits this degree of freedom.

LoPA Gating in practice. The rank r controls the distribution of compute between the three matrices. Lowering the rank makes inference cheaper but can degrade the validation NLL. Noting $r = \alpha D$, where $\alpha \in (0, 1)$, we set $\alpha = 1/8$ unless specified. Replacing W_1 by the product of two matrices increases the depth of the network, making training at large scale unstable. To overcome this, we add a Root Mean Square Normalization (RMSNorm, Zhang & Sennrich, 2019) between the two low-rank terms: $W_1(x) = Gx + U \cdot \mathrm{RMSNorm}(Vx)$.

Adaptive hidden dimension. To keep the FFN's number of parameters constant with LoPA Gating, we adapt the hidden dimension H to H'. Therefore, we solve for H' so that $r(H' + D) + 2H'D$ is equal to the standard parameter count $3HD$: $H' = D(3H - r)/(r + 2D)$.

Double activation. At this stage, LoPA Gating usually underperforms relative to Standard Gating (see Table 4). Intuitively, reducing the capacity before the activation weakens the FFN's ability to use its only non-linearity. The *double activation* avoids this: an activation function is applied to the outputs of *both* W_3 and W_1, see Figure 2 (we cast only W_1's activation). This results in a slight performance increase compared to the Standard Gating, see Appendix A for detailed ablations.

FLOP-ratio for LoPA Gating. Omitting the negligible cost of the fixed G, the gate $W_1 = UV$ costs $r(D + H')$, while W_3 and W_2 cost $(1 - s)DH'$ each:

$$\text{FLOP-ratio}(s) = \frac{r(D + H') + 2DH'}{r(D + H') + 2(1 - s)\, DH'}. \tag{4}$$

In our scaling ladder, H/D is kept constant at $8/3$ across model sizes. When $s \to 1$ the FLOP-ratio floor depends solely on $\alpha = r/D$, see Figure 4. For example, with $\alpha = 1/8$ LoPA Gating's FLOP-ratio becomes 13.5, to be compared to just 3 for Standard Gating.

3.4 Architecture Casting

Model casting can be applied on top of a pre-trained model: we refer to this as *Architecture Casting*, in which the full gating matrix W_1 is replaced with LoPA Gating: at casting time, we replace the W_1 matrix with the low-rank expression from Equation 3, where the matrices are chosen to approximate the current W_1. We then resume the training of the model, which adapts to both the `R-S+` activation and the new architecture. For Architecture Casting, we do not introduce an RMSNorm between U and V, nor an activation after W_3, in order to keep the cast model as close as possible to the original checkpoint. Note that replacing the full W_1 matrix by this parameterization reduces the number of FFN parameters (since we cannot adapt the hidden dimension H), which degrades the model's performance as expected.

LoPA Gating approximation. To approximate the matrix $W_1 \in \mathbb{R}^{H \times D}$ with a LoPA Gating FFN at a fixed rank r, we solve:

$$\min_{M \in \mathcal{C}, U \in \mathbb{R}^{H \times r}, V \in \mathbb{R}^{r \times D}} \|W - M - UV\|_F^2 \tag{5}$$

where we choose $\mathcal{C} \subset \{0, -1, +1\}^{H \times D}$ to have exactly one nonzero entry per row (see Appendix A for alternative designs). Solving this optimization is intractable, so we resort to an alternating minimization strategy, which is guaranteed to decrease the objective: in turn we (*1*) fix U, V: the optimal position and sign of the ± 1 in each row of M are given by the largest-magnitude entry of the row in $W - UV$; (*2*) fix M: the optimal UV is the truncated Singular Value Decomposition (SVD) of $W - M$. Since $\mathcal{C}$ is discrete, it converges in a finite number of iterations (a few steps in practice). We start the optimization by setting $U = V = 0$ and updating M first.

4 Experimental Setup

Model architecture We train dense decoder-only models. The model architecture is derived from Llama3 (Dubey et al., 2024). We use the Llama3 tokenizer, a fast Byte-Pair Encoding tokenizer implemented with TikToken, with a vocabulary of 128,000 regular tokens as well as 256 reserved tokens. The transformer blocks use grouped-query attention (Ainslie et al., 2023), RMSNorm (Zhang & Sennrich, 2019) prenormalization, rotary positional encoding (RoPE) (Su et al., 2021) with $\theta = 5 \cdot 10^5$ and document causal masking.

Training schedule The Chinchilla scaling law (Hoffmann et al., 2022) states that the minimal training cost to obtain a given loss is reached when the number T of training tokens is roughly $20\times$ the number of model parameters (20 TPP). We use this T as the unit for the training duration, because it is independent of training hours (hardware dependent) or optimization steps (depends on batch size), and can be compared over model sizes. To improve the inference-time accuracy, we consider training regimes longer than 1T, following recent practice (Dubey et al., 2024; Riviere et al., 2024). We distinguish the pre-training (PT) from the mid-training (MT) budget, i.e. the number of tokens spent on model casting after the activation swap, excluding the final decay.

The PT phase lasts 4T. Since the model needs to adapt to the activation change, we reset the optimizer state when starting MT. The MT is divided into a three-phase WSD learning rate schedule:

- **W**arm-up: The learning rate linearly increases from 0 to γ over $T/20$ tokens. This base learning rate γ and the batch size are determined by scaling laws (Appendix E);
- **S**table: the learning rate is maintained at γ until reaching T, 2T, 4T or 7T training tokens. The loss decreases slowly during this phase.
- **D**ecay: the learning rate is decreased with a Cosine schedule over T tokens. The loss gain during this phase (the "accuracy boost") is roughly constant.

The WSD schedule allows us to quantify how many extra tokens are needed to reach the same training loss as the baseline. Unless specified, we perform 4T pre-training, 4T W+S and 1T decay.

Dense baseline The baseline is a vanilla `SILU`-based model that continues from the same pre-trained checkpoint, using the same MT budget and WSD schedule, but without an activation swap or ℓ_1 loss. The comparison is therefore matched on both pre-training and mid-training compute.

Metrics. We measure the sparsity and Negative Log-Likelihood (NLL), both computed on a held-out validation set. The NLL on a validation set is less noisy than the training loss because it is evaluated on the same set of sequences for all models. We average the sparsity s over all FFN hidden activations. This is the sparsity of the gate post-activation $A(W_1 x)$.

Scaling ladder The scaling ladder starts with a set of twelve pre-trained models spanning nearly two orders of magnitude in size, ranging from 30 million to 5.79 billion non-embedding parameters (Szilvasy et al., 2026). We use the scaling ladder models for mid-training with model casting plus ℓ_1 and for LoPA Gating. The AdamW optimizer (Loshchilov & Hutter, 2017) learning rate is tuned for the WSD schedule. Each model in the ladder is trained for a total duration of 8T, see Appendix E for details.

Table 1: **Model casting enables significant FLOPs speedups at low NLL impact.** Results for 0.81B-parameter models (level 8 in Table 12). ΔNLL is computed relative to the first row in **bold**. We indicate the Base or LoPA architecture with the activation [in square brackets]. Table 15 in Appendix H reports the same setups across casting durations.

Setup (pre-training → mid-training)	NLL	ΔNLL	Sparsity	Speedup
Baselines: mid-training and RELU-fication				
Base[SILU] → Base[SILU] (dense baseline)	**2.145**	—	—	1.00×
Base[SILU] → Base[RELU] (RELU-fication)	2.159	+0.65%	76%	2.00×
(a) Activation Casting				
Base[SILU] → Base[R-S+]	2.153	+0.36%	69%	1.84×
Base[SILU] → Base[R-S+, ℓ_1@90%]	2.160	+0.68%	89%	2.48×
(b) LoPA Casting				
LoPA[SILU] → LoPA[SILU]	2.150	+0.24%	—	1.00×
LoPA[SILU] → LoPA[R-S+]	2.148	+0.13%	51%	1.89×
LoPA[SILU] → LoPA[RELU]	2.149	+0.19%	57%	2.11×
LoPA[SILU] → LoPA[R-S+, ℓ_1@90%]	2.165	+0.93%	89%	5.83×
(c) Architecture Casting (−21% parameters and max FLOPs)				
Base[SILU] → LoPA[SILU]	2.180	+1.63%	—	1.39×
Base[SILU] → LoPA[R-S+]	2.186	+1.90%	73%	4.21×
Base[SILU] → LoPA[RELU]	2.194	+2.27%	78%	4.87×
Base[SILU] → LoPA[R-S+, ℓ_1@90%]	2.193	+2.20%	90%	7.94×

5 RESULTS

5.1 MODEL CASTING

Impact on NLL. Table 1 shows model casting results during mid-training. We compare the three setups (Activation, LoPA, and Architecture casting); for each setup, we train models both with and without the sparsifying ℓ_1 loss, along with a reference run that uses the SILU activation throughout the entire training duration. Each model is pre-trained for $4T$, cast for $4T$, and annealed for $1T$ with a decaying learning rate. Activation and LoPA casting coupled with the ℓ_1 loss achieve sparsity levels close to 90%, while keeping the NLL degradation below 1%. Adding an ℓ_1 loss slightly degrades the performance, but increases the sparsity level to the 90% target, enabling FLOPs speedups up to $5.83\times$ in LoPA Casting. As Architecture Casting reduces the parameter count, it leads to a larger degradation of the NLL; however, Appendix G (Table 14) shows that Architecture Casting's NLL is on par with models with equivalent parameter counts. We show that these findings are consistent at larger model scales in Appendix I, where we observe smaller relative NLL degradations for 5.79B models. We also evaluate the mid-trained models' downstream performance on a suite of 12 short-context tasks and compare them against a control given the same mid-training budget without model casting. Activation Casting costs 0.4 points of the Core-12 average at a 90% sparsity target, and 1.6 points on Qwen3-1.7B; see Appendix J.

Catch-up duration. Another way of looking at the training process is to measure how many additional training tokens are needed to recover the model's pre-casting NLL. We report this catch-up duration across model scales in Appendix K (Table 19). Recovering the pre-casting NLL takes at most $1.25T$ without the ℓ_1 loss (0.5–$1.0T$ for LoPA Casting); driving the model to 90% sparsity with the ℓ_1 loss raises this to 1.5–$2.75T$ (2.75T for the largest LoPA Casting model). The catch-up duration is roughly constant across model scales.

Mixture-of-Experts. Model casting also applies to MoE models, which is another way of performing sparse inference. We evaluate on the expert FFN blocks of an eight-expert, top-2 model during mid-training for 4T, against a SILU top-2 model given the same mid-training budget (NLL 1.981). With a light ℓ_1 penalty ($\lambda = 10^{-3}$), the model reaches 59.0% activation sparsity for $+0.033$ NLL, and targeting 90% sparsity reaches 88.6% for $+0.083$ NLL. By comparison, switching to top-1 routing — the standard architectural route to sparsity — ends up at a *worse* NLL (2.073 vs 2.064) after the same mid-training budget, while yielding no activation sparsity at all. Thus, MoE and Activation Casting can be combined. Appendix C provides the full experimental setup, an alternative router that selects active experts through casting, and additional ablations; see Tables 8, 9 and 10.

Table 2: **Trained-in sparsity gives a larger FLOP speedup than post-hoc pruning at matched quality.** FFN theoretical FLOP speedup ($1/\text{FLOP-ratio}$; dense $= 1\times$, higher is better), at batch size 1, with all methods matched at the reference NLL. See Appendix B for other training budgets.

		Theoretical FLOP speedup ↑		
FFN Architecture	Ref. NLL↓	Model casting @75% (ours)	top-p	TEAL
Standard Gating	2.154	**2.0**	1.4	1.4
LoPA Gating	2.153	**3.2**	1.3	1.3

Compatibility with quantization and post-training. In Appendix L we show that quantization-aware training can be applied on top of models sparsified with model casting. Combining both, we obtain a 1.57% degradation of the NLL in `int8`. Our low-precision kernels, designed to exploit the sparsity, achieve an $8.13\times$ speedup at 90% sparsity. Model casting also survives post-training: LCFT and SFT cost model casting variants about as much as the dense control (Appendix M).

Additional ablations. We validate some of our design choices in the appendix. An ℓ_1 loss *without* model casting cannot induce significant sparsity and degrades performance, and pre-training with `SILU` and ℓ_1 before casting worsens the final sparsity (see Appendices N and O). Among alternative casting activations (Appendix P), `SILU` and `GELU` give the best NLL; Shifted `RELU` (Mirzadeh et al., 2024; Qiu et al., 2017) gives the highest sparsity at a high NLL cost. `R-S+` keeps the NLL close to `SILU` at moderate sparsity; the squared `RELU` advocated by Zhang et al. (2024) when training from scratch gives the *lowest* exact-zero sparsity in our model casting setting.

5.2 Comparison to prior FFN sparsification methods

We compare model casting with i) two training-free FFN sparsification approaches, top-p (Szatkowski et al., 2026) and TEAL (Liu et al., 2025a) and ii) another fine-tuning approach, `RELU`-fication (Mirzadeh et al., 2024). See Appendix B for the implementation details. Every FLOP ratio we report is normalized to an equal-parameter dense standard-gating `SILU` model. For the training-free approaches, we further mid-train the pre-trained 4T model for the same total duration.

Training-free approaches. In Table 2, we match all methods with a common reference NLL (the validation NLL of model casting at 75% sparsity) and compare the corresponding FLOP speedup. Model casting results in higher sparsity levels and FLOPs speedups than post-hoc approaches: 2–$3.2\times$ against 1.3–$1.6\times$ for top-p and TEAL. Moreover, the advantage grows for longer schedules, see Table 6 in Appendix B.

Trained approaches. Table 1 compares our method versus `RELU`-fication. Model casting reaches higher sparsity at a larger FLOP speedup. The full comparison across mid-training budgets is in Table 7 of Appendix B. For Activation Casting, this comes with an NLL difference below 0.01 nats; for LoPA Casting the 90% target costs a small NLL increase (0.01–0.04) but the model is substantially cheaper. Moreover sparsity is a target we set, so casting reaches around 90% in every setting, whereas `RELU`-fication offers no comparable control mechanism.

Casting an off-the-shelf checkpoint. Table 3 applies Activation Casting and Architecture Casting to Qwen3-1.7B-Base: both reach more than 89% sparsity, and Architecture Casting incurs 0.062 higher NLL than Activation Casting but removes 294M parameters. Mid-training details, the learning-rate sweep and downstream evaluations are in Appendix Q.

5.3 CPU/GPU optimization for FFNs with sparse activations

Exploiting sparsity is mainly useful at LLM generation time, during single-token autoregressive decoding. With batched computation, the sparsity patterns do not coincide between batch elements. This reduces the sparsity as the rows/columns of W_3 and W_2 must be loaded for the union of the inverse sparsity masks.

CPU and GPU kernels. On CPU, we use a sparsity-aware AVX-512 kernel. On GPU, we implement a family of Triton kernels that fuse the W_2 and W_3 projections and skip the inactive neurons.

Table 3: Qwen3-1.7B base model mid-trained with model casting, $\gamma = 7.4 \times 10^{-5}$. The 0T row is the pre-trained checkpoint that we try to recover with sparse models. "Decay" indicates if cosine annealing is applied.

Method	Budget	Decay	Sparsity (%)	NLL	Speedup
Pre-trained (no casting)	0T	—	—	**1.843**	1.00×
Activation Casting PT → Base[R−S+, ℓ_1@90%]	1T	✗	89.6	1.912	2.48×
	2T	✗	89.6	1.908	2.48×
	4T	✗	89.6	1.907	2.48×
	5T	✓	89.5	1.875	2.48×
Architecture Casting PT → LoPA[R−S+, ℓ_1@90%]	1T	✗	89.6	2.020	8.01×
	2T	✗	89.7	1.991	8.05×
	4T	✗	89.7	1.971	8.05×
	5T	✓	89.6	1.937	8.01×

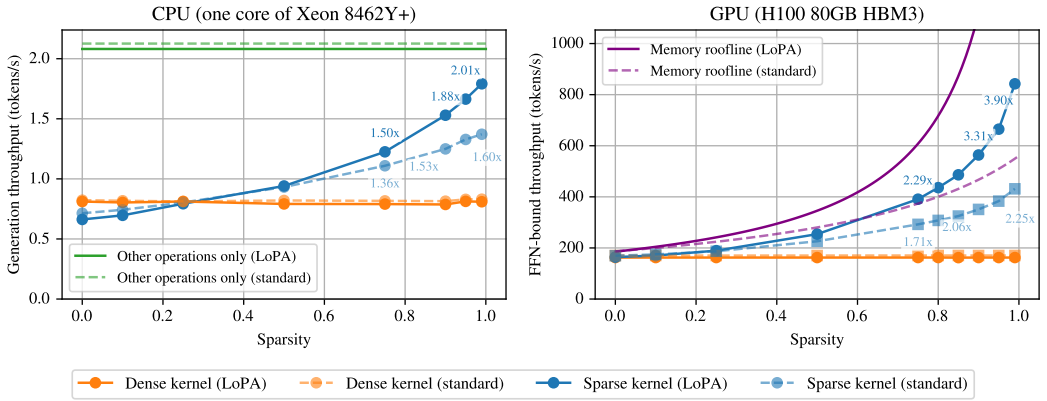


Figure 5: **Sparsity translates to wall time improvements on CPU and GPU.** Throughput as a function of sparsity, for a 5.79B non-embedding parameters model (level 13 in the scaling ladder from Table 12) in float32. CPU times are end-to-end; GPU times cover the FFN only, with the memory-bandwidth roofline shown as an upper bound. At 90% sparsity, the speedup on GPU is 2.06× for Standard Gating and 3.31× for LoPA Gating.

The exact blocking and tiling scheme is selected according to the sparsity. All results are for float32 computations, see GPU results for bfloat16 in Appendix R. On the GPU, overheads such as KV-cache management, Python interpretation, attention masks, and RoPE account for a larger fraction of runtime. These overheads are less significant on larger models or carefully optimized software stacks. We plot rooflines corresponding to incompressible costs in order to assess the tightness of our implementations. On CPU, we plot the other operations, which account for the time spent outside of the FFN and are independent of sparsity. On GPU, we plot the memory roofline, i.e. the speedup that would be attained by just loading the weights from High Bandwidth Memory.

Wall-clock speedups. Figure 5 reports wall-clock timings: end-to-end on the CPU, and FFN-only on the GPU, in contrast to the theoretical FLOP speedups reported in Section 5.2. On the CPU, 90% sparsity directly translates into a practical 1.53× wall-clock speedup for Standard Gating, and 1.88× for LoPA Gating. For the GPU, the wall-clock speedup of the FFN inference is around 2.06× at 90% sparsity for Standard Gating, and 3.31× for LoPA Gating. At 95% sparsity on the GPU, LoPA Gating achieves 3.90× (while Activation Casting is limited to 3×).

6 Conclusion

Szatkowski et al. (2026) define the "critical sparsity" to be the sparsity attainable at a 1% quality drop, and estimate it as below 40% for an 8B LLM. We show that for the model casting introduced in this work, this number is closer to 90%, and can be set as a target at training time. Building on this, we introduced LoPA Gating, a novel FFN block that allocates the compute between the three FFN matrices to substantially improve the frontier of sparsity acceleration. We proposed Architecture Casting, to convert a pre-trained model (for example Qwen) to LoPA Gating, and model casting applies to the experts of MoE models as well. Our models achieve up to 3.3× wall-clock FFN speedups compared to standard dense designs, while keeping the NLL degradation below 1%.

AI USE STATEMENT

In this work, we used generative AI tools to implement methods and to provide feedback on, but not design, our research methodology and experiments. We have not used generative AI tools for building theoretical models or conceptual frameworks, for cleaning and re-formatting datasets, for proposing or refining hypotheses, for supporting qualitative and thematic data analysis, nor for interpreting results. Designing synthetic datasets, formulating mathematical claims, providing critical ingredients for proofs, writing proofs and assisting with translations are not applicable to this work. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 4895–4901, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.298. URL https://aclanthology.org/2023.emnlp-main.298/.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *ArXiv*, abs/2108.07732, 2021. URL https://api.semanticscholar.org/CorpusID:237142385.

Yonatan Bisk, Rowan Zellers, Ronan Le bras, Jianfeng Gao, and Yejin Choi. PIQA: Reasoning about physical commonsense in natural language. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):7432–7439, Apr. 2020. doi: 10.1609/aaai.v34i05.6239. URL https://ojs.aaai.org/index.php/AAAI/article/view/6239.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, and OpenAI team. Evaluating large language models trained on code. 2021.

Ying Chen, Aoxi Li, Jihun Kim, and Javad Lavaei. Structural correspondence and universal approximation in diagonal plus low-rank neural networks. *arXiv preprint arXiv:2605.05659*, 2026.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. BoolQ: Exploring the surprising difficulty of natural yes/no questions. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2924–2936, 2019. URL https://aclanthology.org/N19-1300/.

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. *ArXiv*, abs/1803.05457, 2018. URL https://api.semanticscholar.org/CorpusID:3922816.

Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 933–941. JMLR.org, 2017.

DeepSeek-AI, Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. DeepSeek-V2: A strong, economical, and efficient mixture-of-experts language model, 2024. URL https://arxiv.org/abs/2405.04434.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Andi Han, Jiaxiang Li, Wei Huang, Mingyi Hong, Akiko Takeda, Pratik Jawanpuria, and Bamdev Mishra. SLTrain: a sparse plus low rank approach for parameter and memory efficient pretraining. *Advances in Neural Information Processing Systems*, 37:118267–118295, 2024.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=d7KBjmI3GmQ.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

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.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B, 2023. URL https://arxiv.org/abs/2310.06825.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de Las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts. *ArXiv*, abs/2401.04088, 2024. URL https://api.semanticscholar.org/CorpusID:266844877.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. 05 2017. doi: 10.48550/arXiv.1705.03551.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard H. Hovy. Race: Large-scale reading comprehension dataset from examinations. In *Conference on Empirical Methods in Natural Language Processing*, 2017. URL https://api.semanticscholar.org/CorpusID:6826032.

Donghyun Lee, Jaeyong Lee, Genghan Zhang, Mo Tiwari, and Azalia Mirhoseini. CATS: Context-aware thresholding for sparsity in large language models. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=v3w2a7EInO.

Zonglin Li, Chong You, Srinadh Bhojanapalli, Daliang Li, Ankit Singh Rawat, Sashank J. Reddi, Ke Ye, Felix Chern, Felix Yu, Ruiqi Guo, and Sanjiv Kumar. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=TJ2nxciYCk-.

James Liu, Pragaash Ponnusamy, Tianle Cai, Yoon Kim, and Ben Athiwaratkun. Training-free activation sparsity in large language models. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 98302–98322, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/f3f2ff9579ba6deeb89caa2fe1f0b99c-Paper-Conference.pdf.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023a. URL https://openreview.net/forum?id=1qvx610Cu7.

Kai Liu, Bowen Xu, Shaoyu Wu, Xin Chen, Hao Zhou, Yongliang Tao, and lulu hu. La roSA: Enhancing LLM efficiency via layerwise rotated sparse activation. In *Forty-second International Conference on Machine Learning*, 2025b. URL https://openreview.net/forum?id=1b6NNpFYI4.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient LLMs at inference time. In *International Conference on Machine Learning*, pp. 22137–22176. PMLR, 2023b.

Maria Lomeli, Matthijs Douze, Gergely Szilvasy, Loic Cabannes, Jade Copet, Sainbayar Sukhbaatar, Jason Weston, Gabriel Synnaeve, Pierre-Emmanuel Mazaré, and Hervé Jégou. Stochastic activations, 2025. URL https://arxiv.org/abs/2509.22358.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Yuqi Luo, Chenyang Song, Xu Han, Yingfa Chen, Chaojun Xiao, Xiaojun Meng, Liqun Deng, Jiansheng Wei, Zhiyuan Liu, and Maosong Sun. Sparsing law: Towards large language models with greater activation sparsity. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=SBUc5wirM8.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2381–2391, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1260. URL https://aclanthology.org/D18-1260/.

Seyed Iman Mirzadeh, Keivan Alizadeh-Vahid, Sachin Mehta, Carlo C del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. ReLU strikes back: Exploiting activation sparsity in large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=osoWxY8q2E.

Zhanfeng Mo, Long-Kai Huang, and Sinno Jialin Pan. Parameter and memory efficient pretraining via low-rank Riemannian optimization. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu (eds.), *International Conference on Learning Representations*, volume 2025, pp. 7907–7936, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/16fafdc5e8ab22780873d607ce4387a8-Paper-Conference.pdf.

Siyuan Qiu, Xiayang Xu, and Bolin Cai. Frelu: Flexible rectified linear units for improving convolutional neural networks. In *2017 IEEE International Conference on Image Processing (ICIP)*, pp. 355–359. IEEE, 2017.

Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, et al. Gemma 2: Improving open language models at a practical size. *ArXiv*, abs/2408.00118, 2024. URL https://api.semanticscholar.org/CorpusID:270843326.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. WinoGrande: An adversarial winograd schema challenge at scale. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):8732–8740, Apr. 2020. doi: 10.1609/aaai.v34i05.6399. URL https://ojs.aaai.org/index.php/AAAI/article/view/6399.

Noam M. Shazeer. GLU variants improve transformer. *ArXiv*, abs/2002.05202, 2020. URL https://api.semanticscholar.org/CorpusID:211096588.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU, 2023.

Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *ArXiv*, abs/2104.09864, 2021. URL https://api.semanticscholar.org/CorpusID:233307138.

Filip Szatkowski, Patryk Będkowski, Alessio Devoto, Jan Dubiński, Pasquale Minervini, Mikołaj Piórczyński, Simone Scardapane, and Bartosz Wójcik. Universal properties of activation sparsity in modern large language models. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=I9M5OoOBnI.

Gergely Szilvasy, Manuel Faysse, Maria Lomeli, Matthijs Douze, Pierre-Emmanuel Mazaré, Loïc Cabannes, Wen-tau Yih, and Hervé Jégou. Self-pruned key-value attention: Learning when to write by predicting future utility, 05 2026.

Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. CommonsenseQA: A question answering challenge targeting commonsense knowledge. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 4149–4158, 2019. URL https://aclanthology.org/N19-1421/.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023a. URL https://api.semanticscholar.org/CorpusID:257219404.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems*, 2017. URL https://api.semanticscholar.org/CorpusID:13756489.

Ziteng Wang, Jun Zhu, and Jianfei Chen. ReMoE: Fully differentiable mixture-of-experts with reLU routing. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=4D0f16Vwc3.

Tianwen Wei, Bo Zhu, Liang Zhao, Cheng Cheng, Biye Li, Weiwei Lü, Peng Cheng, Jianhao Zhang, Xiaoyu Zhang, Liang Zeng, Xiaokun Wang, Yutuan Ma, Rui Hu, Shuicheng Yan, Han Fang, and Yahui Zhou. Skywork-MoE: A deep dive into training techniques for mixture-of-experts language models, 2024a. URL https://arxiv.org/abs/2406.06563.

Xiuying Wei, Skander Moalla, Razvan Pascanu, and Caglar Gulcehre. Building on efficient foundations: Effective training of LLMs with structured feedforward layers. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 4689–4717. Curran Associates, Inc., 2024b. doi: 10.52202/079017-0153. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/0877af85978e9e630b77f6221db47876-Paper-Conference.pdf.

Qwen An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxin Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yi-Chao Zhang, Yunyang Wan, Yuqi Liu, Zeyu Cui, Zhenru Zhang, Zihan Qiu, Shanghaoran Quan, and Zekun Wang. Qwen2.5 technical report. *ArXiv*, abs/2412.15115, 2024. URL https://api.semanticscholar.org/CorpusID:274859421.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? In *Annual Meeting of the Association for Computational Linguistics*, 2019. URL https://api.semanticscholar.org/CorpusID:159041722.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *ArXiv*, abs/1910.07467, 2019. URL https://api.semanticscholar.org/CorpusID:113405151.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Zhengyan Zhang, Yixin Song, Guanghui Yu, Xu Han, Yankai Lin, Chaojun Xiao, Chenyang Song, Zhiyuan Liu, Zeyu Mi, and Maosong Sun. ReLU2 wins: Discovering efficient activation functions for sparse LLMs. *ArXiv*, abs/2402.03804, 2024. URL https://api.semanticscholar.org/CorpusID:267499856.

Zhenyu (Allen) Zhang, Zechun Liu, Yuandong Tian, Harshit Khaitan, Zhangyang Wang, and Steven Li. R-Sparse: Rank-aware activation sparsity for efficient LLM inference. *ArXiv*, abs/2504.19449, 2025. URL https://api.semanticscholar.org/CorpusID:278165071.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *ArXiv*, abs/2311.07911, 2023. URL https://arxiv.org/abs/2311.07911.

APPENDIX CONTENTS

A Ablation on the LoPA Gating design **15**

B Baselines: additional details **16**

C Case study: Mixture-of-Experts **18**

D NLL vs sparsity for different fixed lambda values **21**

E Scaling ladder **21**

F Evaluation benchmarks **21**

G Architecture Casting vs Standard Gating at reduced parameter count **23**

H Full results for different casting durations **24**

I Validation NLL for a model with 5.8B weights **24**

J Short-context evaluations **25**

K Catch up times at mid-training **25**

L Quantization-aware training **27**

M Post-training **28**

N Enforcing sparsity with ℓ_1 loss only **29**

O Ablating the use of ℓ_1 loss during pre-training **29**

P Casting with other activation functions **30**

Q Qwen3 mid-training details **31**

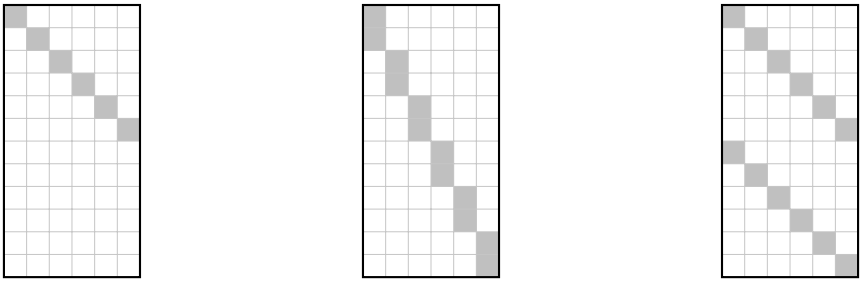
R GPU speedup in bfloat16 **33**

A ABLATION ON THE LOPA GATING DESIGN

In order to demonstrate the effectiveness of the LoPA Gating architecture as an under-parameterized matrix design, we compare models with alternative architectural variations. Unless stated otherwise, W_1 is parameterized using LoPA Gating: the LoPA variant consisting of low-rank and fixed diagonal-like components, with double activation (activations on the outputs of both W_1 and W_3), and the hidden dimension of the FFN blocks is increased in order to match the number of parameters of the baseline (as described in Subsection 3.3). All models are trained with `SILU` for a duration of $8T$, without any sparsity loss. Their specific characteristics are as follows:

- **Full matrix (baseline)**: standard SwiGLU block, with 3 full matrices and a single activation function on the output of W_1;
- **Full matrix, double activation**: 3 full matrices, with activations on the outputs of both W_1 and W_3;
- **Low-rank**: parameterization $W_1 = UV$ (no fixed component);
- **Low-rank + Identity**: parameterization $W_1 = I_{\text{pad}} + UV$, where I_{pad} is the identity matrix I_D padded at the bottom with zeros;
- **Low-rank + Diagonal (LoPA)**: parameterization $W_1 = G + UV$, with $[G]_{i,j} = 1$ if $j = \lfloor i/k \rfloor$, and 0 otherwise, where $k = H/D$;
- **Low-rank - Diagonal (LoPA)**: parameterization $W_1 = -G + UV$ (similar to Low-rank + Diagonal, but with -1 on the diagonal instead of $+1$)
- **Low-rank + Tiled Identity (LoPA)**: parameterization $W_1 = T + UV$, where T is the concatenation of identity matrices I_D;
- **Single activation**: same as Low-rank + Diagonal, but no activation is added to the output of W_3.

The non-learnable components of the LoPA variants are visualized in Figure 6. Note that Low-rank + Diag and Low-rank + Tiled Identity are identical up to a permutation. We included their comparison as a sanity check, but their minor differences in NLL are to be attributed to training noise.



(a) Low-rank + Identity: I_{pad} (b) Low-rank + Diagonal: G (c) Low-rank + Tiled Identity: T

Figure 6: **Illustration of the non-learnable component.** We depict, for $H = 12$ and $D = 6$, the fixed component $M \in \mathbb{R}^{H \times D}$ of three low-rank variants. (a) Low-rank + Identity uses the padded identity I_{pad}. (b) Low-rank + Diagonal uses the repeated diagonal G. (c) Low-rank + Tiled Identity uses the tiled identity T, that is $(Tx)_i = x_{\,i \bmod D}$. All non-zeros are fixed 1s (gray). The last two variants, or any permutation of those, correspond to what we call LoPA. The simple identity I_{pad} differs from LoPA as some rows only contain zeros.

Table 4: **Ablations of different design choices for the W_1 matrix.** Unless stated otherwise, W_1 is parameterized using LoPA Gating, consisting of Low-rank + Diagonal matrices with double activation (activations on the outputs of both W_1 and W_3). All models are trained with `SILU` for a duration of $8T$, without any sparsity loss, and have the same parameter count. We report the standard deviation of the NLL of the baseline, computed over 4 seeds. †Low-rank + Diag and Low-rank + Tiled Identity are identical up to a permutation. We include their comparison as a sanity check. Their minor differences in NLL are to be attributed to training noise.

	Average validation NLL							
	Level 2				Level 8			
Parameterization	r=64	r=128	r=256	r=512	r=128	r=256	r=512	r=1024
Full matrix (baseline)		2.800 ± 0.005				2.152 ± 0.002		
Full matrix, double activation		2.795				2.153		
Low-rank	2.829	2.813	2.804	2.806	2.183	2.167	2.154	2.153
Low-rank + Identity	2.810	2.799	2.795	2.805	2.159	2.156	2.152	2.154
Low-rank + Diagonal†	**2.787**	**2.778**	2.775	**2.792**	2.154	**2.151**	**2.148**	2.156
Low-rank - Diagonal	**2.787**	2.780	**2.774**	2.817	2.157	2.152	2.150	**2.152**
Low-rank + Tiled Identity†	**2.787**	2.779	**2.774**	2.793	2.155	**2.151**	**2.148**	2.158
LoPA Gating w. single activation	2.804	2.810	2.805	2.811	2.179	2.165	2.155	2.156

In Table 4 we show that LoPA Gating (Low-rank + Diagonal) and Low-rank + Tiled Identity outperform the other variants we tested. Interestingly, those parameterizations with a 1 on each row of the fixed matrix are significantly better than Low-rank + Identity, despite representing a space of matrices with the same maximum rank. Furthermore, while double activation did not significantly improve the NLL in the standard gating setup, it substantially reduced it for the LoPA Gating parameterization.

In general, these results justify our choice to use LoPA Gating as an efficient parameterization of W_1. Furthermore, this experimentally confirms that the position of the 1s in the row of the fixed matrix M is not meaningful. This leads us to choose the set of matrices with a single ± 1 in each row as the set $\mathcal{C}$ when approximating a full matrix with a low-rank parameterization.

B BASELINES: ADDITIONAL DETAILS

In this section we describe the implementation details of the baselines we report in Tables 6 and 7. Sparsity can in principle be induced at several sites of the FFN, each skipping different matmuls: the two inputs (s_g, s_u), the gate output $A(W_1 x)$ (s_{pre}, the s of Section 3.1), and the gated intermediate

$\text{out}2 \cdot x_3$ (s_{gated}, which skips further columns of W_2). Our method targets the gate, and, with double activation, additionally the gated intermediate; the input sites are what training-free methods such as top-p and TEAL exploit. Table 5 catalogs each site and the matmuls it reduces.

Table 5: **Summary of sparsification approaches.** The two FFN sparsity mechanisms and the matmuls each reduces. Input sparsity (s_g, s_u; dim D) skips input coordinates; hidden-activation sparsity ($s_{\text{pre}}, s_{\text{gated}}$; dim H') skips hidden units.

Symbol	Tensor	Dim	Produced by	Skips
s_g	input $x \to W_1$	D	TEAL/top-p	W_1 input read (V)
s_u	input $x \to W_3$	D	TEAL/top-p	W_3 input read
s_{pre}	$A(W_1 x)$	H'	ours (trained), top-p (gate)	W_3 rows + W_2 cols
s_{gated}	$\text{out}2 \cdot x_3$	H'	ours, TEAL/top-p	W_2 cols

- **`RELU`-fication** (Mirzadeh et al., 2024): consists of swapping the FFN activation to `RELU` during mid-training. No additional ℓ_1 loss is used so the sparsity cannot be controlled.
- **Top-p** (Szatkowski et al., 2026): a training-free method applied at inference. For an activation vector v (per token, over the feature dimension), it keeps the smallest set of largest-magnitude entries whose cumulative mass $\sum |v|^L$ reaches a fraction greater than $p \in [0, 1]$ of the total, and zeros the rest; $p = 1$ keeps everything (dense) and smaller p gives more sparsity. We use $L = 1$ (cumulative absolute value) and choose p to match the reference NLL exactly. The same p is applied uniformly across all layers. We can target any FFN site independently: the input x, the gate $A(W_1 x)$, the up-projection $W_3 x$, the gated intermediate, or the two input sites jointly. The intermediate site is used for the per-site comparison and all input sites are used for the all-FFN comparison.
- **TEAL** (Liu et al., 2025a): a training-free method that induces *input* activation sparsity by static magnitude thresholding. For each FFN matrix it zeros every input entry with $|x| \leq t$ before the matrix-vector multiplication, where the threshold $t = t_p$ is a fixed scalar per (layer, matrix), calibrated offline as the p-quantile of $|x|$. Per-matrix sparsities are allocated by the block-wise greedy search (their Algorithm 1) that minimizes the block output ℓ_2 error at a fixed block budget. We calibrate the thresholds on a held-out slice of the same evaluation mixture. For comparability with our FFN-site sparsity, we apply TEAL to the FFN matrices only—W_1, W_3 on the input x and W_2 on the gated intermediate—leaving attention dense; the reported sparsity is therefore FFN-only.

Table 6: **Full version of Table 2: trained-in sparsity gives a larger theoretical FLOP speedup than post-hoc pruning at matched quality.** FFN theoretical FLOP speedup, at a shared reference NLL equal to our method at 75% sparsity. Models are pre-trained $4T$ then cast for the listed mid-training (MT) budget; ✓ adds a $1T$ decay. Table 2 reports the decayed $4T$ row only; here we give every mid-training budget, with and without decay. Casting stays near $2.0\times$ for Activation Casting and $3.2\times$ for LoPA Casting, while top-p and TEAL fall toward $1.3\times$ as the model is mid-trained longer.

				Theoretical FLOP speedup ↑		
FFN Architecture	MT	Decay	Ref. NLL ↓	Ours @75%	top-p	TEAL
Activation Casting	1T	✗	2.326	**2.0**	1.6	1.6
	2T	✗	2.307	**2.0**	1.6	1.5
	4T	✗	2.294	**2.0**	1.5	1.4
	4T	✓	2.154	**2.0**	1.4	1.4
LoPA Casting	1T	✗	2.331	**3.2**	1.5	1.6
	2T	✗	2.309	**3.2**	1.4	1.5
	4T	✗	2.294	**3.2**	1.4	1.4
	4T	✓	2.153	**3.2**	1.3	1.3

Table 7: **Full version of the RELU-fication comparison of Table 1: model casting reaches higher sparsity at a larger theoretical FLOP speedup.** Both are trained-activation methods reported at their own operating points. Table 1 reports the decayed $4T$ operating point only; here we give every mid-training budget, with and without decay. At comparable NLL, model casting attains 89% sparsity for both Standard Gating and LoPA Gating, whereas RELU-fication plateaus at 75–79% and 57–62% respectively.

			Model casting @90% (ours)			RELU-fication		
FFN Architecture	MT	Decay	NLL↓	FLOP sp.↑	Spars.↑	NLL↓	FLOP sp.↑	Spars.↑
Standard Gating	1T	✗	2.340	**2.5**	89%	2.330	2.0	75%
	2T	✗	2.315	**2.5**	89%	2.312	2.1	77%
	4T	✗	2.298	**2.5**	89%	2.299	2.1	79%
	4T	✓	2.160	**2.5**	89%	2.159	2.0	76%
LoPA Gating	1T	✗	2.346	**5.8**	89%	2.326	2.3	62%
	2T	✗	2.323	**5.8**	89%	2.304	2.3	62%
	4T	✗	2.307	**5.9**	90%	2.291	2.3	62%
	4T	✓	2.165	**5.8**	89%	2.149	2.1	57%

C Case study: Mixture-of-Experts

One advantage of model casting is that it can be used for mid-training a variety of models with different architectures, for example, Mixture-of-Experts (MoE) models. We train from scratch and successively mid-train with model casting i) the FFN blocks of a MoE model and ii) an alternative MoE router that natively selects active experts leveraging model casting as a case study.

Setup. The MoE models are initialized from a Llama 3.2 1B dense model, which we convert to a MoE by replicating the original FFN blocks for each expert with some noise added to break symmetries. Each model has eight experts, of which two are active, with no shared expert; pre-training uses a baseline learning rate of 10^{-4}, a sequence length of 4096, and a duration of 4T. We train two variants: one with a cosine schedule (used only for the additional results below) and one with the WSD schedule. After pre-training, we mid-train with model casting and an ℓ_1 loss for MT durations of 1T, 2T and 4T, comparing against mid-training with SILU as a baseline. MoE is a favorable setting for casting: it reduces the FFN/attention ratio, especially for long sequences, and shrinks the effective batch each expert sees. Experts use a mixed-precision recipe (bf16 non-expert computation, fp32 router, and fp8 expert GEMMs with DeepSeek-V3-style fine-grained scaling); see Table 11 for details.

MoE Router with Sigmoid and Top-k. Given a token representation $x_t \in \mathbb{R}^d$ at position t, the router computes per-expert logits via a linear projection $\ell_t = W_g x_t \in \mathbb{R}^E$, where E is the number of routed experts and $W_g \in \mathbb{R}^{E\times d}$ is the gating matrix. Expert affinities are obtained by an element-wise sigmoid, $s_{t,e} = \sigma(\ell_{t,e}) \in (0,1)$, which treats each expert independently and does not force the scores to compete on the simplex (in contrast to the softmax). An additive, non-differentiable load-balancing bias $b \in \mathbb{R}^E$ (updated online from per-expert token counts) is added *only for selection*, and the top-k experts are chosen as $\mathcal{T}_t = \text{top-}k\big(s_t + b\big)$, while the routing weights are read from the *unbiased* scores $w_{t,e} = s_{t,e}$ for $e \in \mathcal{T}_t$. The weights are optionally L_p-normalized across the selected experts and rescaled by a global constant ρ, giving the layer output

$$y_t \;=\; \rho \sum_{e\in\mathcal{T}_t} w_{t,e}\, f_e(x_t), \tag{6}$$

where $f_e(\cdot)$ denotes the e-th expert FFN. Because $\sigma(\cdot)$ is applied element-wise and the resulting gate directly multiplies the expert output, the router weight modulates the contribution of each active expert rather than acting purely as a selection signal.

MoE Router with Activation Casting. In addition to applying model casting on MoE FFN blocks, we can replace the typical expert selection mechanism of the MoE router by first using a SILU router together with a sparsifying loss when training from scratch. Subsequently, we apply model casting with the sparsifying loss during mid-training. This mechanism enables us to select which experts are active natively without relying on the top-k mechanism.

In Table 8, we present the results of replacing the standard router mechanism of selecting active experts with activation casting. We experimented with setting fixed penalty weights $\lambda \in \{0.1, 0.3\}$ or using the adaptive scheme from ReMoE (Wang et al., 2025), setting a target sparsity to match the standard sparsity of the router (two out of eight experts are active) and $\alpha = 1.05$. We observe that using activation casting with $\lambda = 0$ is the most competitive approach in terms of validation NLL. However, even when we set the target sparsity at 75% during training, the router sparsity at validation time is about five percentage points below the target.

Table 8: **MoE's Average NLL and router sparsity**. We group runs by MT budget (4T, 2T, 1T), and rank them by NLL within each group. As expected, `R-S+` without sparsity loss on the router results in the best NLL but very little sparsity (the model using all the experts most of the time); best sparsity is obtained using the usual sigmoid and top-k routing, but model casting can find intermediate operating points.

Router	λ (PT)	λ (MT)	Train	MT	Avg NLL	Avg sparsity
`R-S+`	0	0	1T	4T	**1.981**	5%
`R-S+`, fixed λ	0.1	0.1	1T	4T	1.999	49%
Sigmoid, top-k	—	—	1T	4T	2.007	**75%**
`R-S+`, fixed λ	0.3	0.1	1T	4T	2.011	65%
`R-S+`, adaptive λ	0.3	@70%	1T	4T	2.023	70%
`R-S+`, adaptive λ	0.1	@70%	1T	4T	2.027	71%
`R-S+`	0	0	1T	2T	**1.997**	6%
`R-S+`, fixed λ	0.1	0.1	1T	2T	2.016	52%
Sigmoid, top-k	—	—	1T	2T	2.020	**75%**
`R-S+`, fixed λ	0.3	0.1	1T	2T	2.030	68%
`R-S+`, adaptive λ	0.3	@70%	1T	2T	2.038	70%
`R-S+`, adaptive λ	0.1	@70%	1T	2T	2.039	72%
`R-S+`	0	0	1T	1T	**2.026**	8%
Sigmoid, top-k	—	—	1T	1T	2.043	**75%**
`R-S+`, fixed λ	0.1	0.1	1T	1T	2.044	53%
`R-S+`, fixed λ	0.3	0.1	1T	1T	2.062	70%
`R-S+`, adaptive λ	0.1	@70%	1T	1T	2.062	72%
`R-S+`, adaptive λ	0.3	@70%	1T	1T	2.067	69%

Activation Casting the MoE FFN blocks. In contrast to the dense models trained with the WSD learning rate schedule, in Table 9, we observe that using model casting for mid-training a MoE model does not catch up with the pre-trained loss when we mid-train the MoE for 4T when using Activation Casting in the FFN blocks. In the MoE case, mid-training with `SILU` catches up with the pre-training final loss when we mid-train for 4T.

In Table 10, we report the performance of a MoE with eight experts that selects the top two versus a MoE with eight experts that selects the top one. The latter model is naturally sparser since it has only one active expert. However, this results in a degradation of performance. Hence, generally it is better to have more active experts and use model sparsity to have higher sparsity levels.

We also explore increasing penalty weight λ to further encourage sparsity in MoE FFN blocks, in Figure 7: the sparsity of MoE can increase by up to 20%, at the cost of NLL degradation.

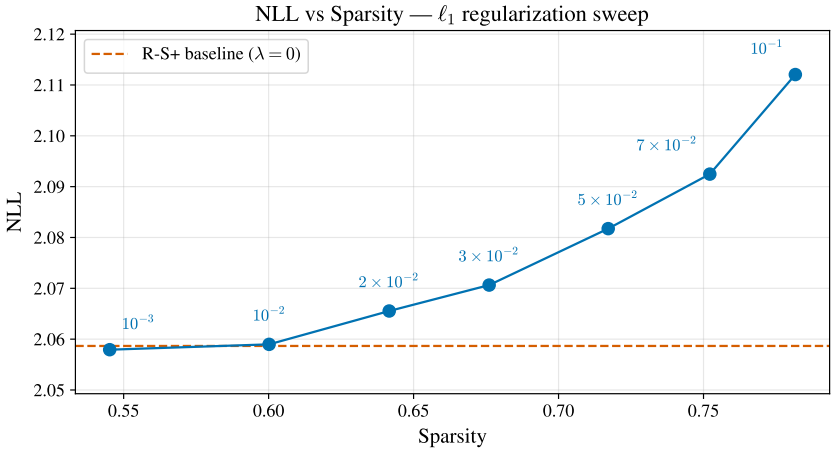


Figure 7: **MoE FFNs sparsity vs. NLL** for different values for the ℓ_1 penalty weight, using cosine schedule. The baseline corresponds to model casting of the FFN to `R-S+` with $\lambda = 0$.

Table 9: **NLL degradation for MoE FFN Activation Casting.** We report the average NLL and sparsity for different penalty weights for ℓ_1 loss with fixed or adaptive λ, in the FFN Activation Casting setup for MoEs. Rows with @$x\%$ use an adaptive ℓ_1 loss for a given sparsity target of $x\%$. We mid-train the MoE with 8 experts top-2 with model casting on the experts FFN blocks using a WSD learning rate schedule. All models start from the same pre-trained checkpoint, trained for 4T.

Activation	MT	λ	NLL	ΔNLL	Sparsity (FFN)
`SILU` (pre-trained)	0T	0	2.008	+1.36%	–
`SILU`	4T	0	1.981	+0.00%	–
`R-S+`	4T	0	2.020	+1.97%	56.4%
`R-S+`	4T	10^{-3}	2.014	+1.65%	59.0%
`R-S+`	4T	10^{-2}	2.017	+1.82%	66.2%
`R-S+`	4T	@75%	2.022	+2.04%	72.8%
`R-S+`	4T	10^{-1}	2.044	+3.15%	82.4%
`R-S+`	4T	@90%	2.064	+4.19%	88.6%

Table 10: **FFN post-activation sparsity vs. lower number of experts.** Performance of MoE top-1 vs MoE top-2, out of 8 experts. `R-S+` @90% with 2 active experts results in lower NLL and lower FLOPs than `SILU` with 1 active expert.

Activation	top-k	MT	NLL	Sparsity (FFN)
`SILU` (pre-trained)	2	0T	2.008	–
`R-S+` @90%	2	4T	2.064	88.6%
`SILU` (pre-trained)	1	0T	2.091	–
`SILU`	1	4T	2.073	–

Table 11: **Mixed-precision training setup.** All fp8 tensors use the E4M3 format (no E5M2).

Component	**Precision**	**FP8 format**
Attention / norms / embeddings / residual	bf16	—
Router GEMM + gating (`SILU`) + routing losses	fp32	—
Expert FFN — forward	fp8 (1×128 act, 128×128 wt)	E4M3
Expert FFN — dgrad (incl. grad. quant.)	fp8 (tile-wise 1×128)	E4M3
Expert FFN — wgrad (`SILU`)	fp8	E4M3
Expert FFN — wgrad (`R-S+`)	bf16	—
Weight-gradient accumulation	fp32	—

D NLL VS SPARSITY FOR DIFFERENT FIXED LAMBDA VALUES

Penalizing the gate activation $A(W_1x)$ with an ℓ_1 loss during mid-training increases sparsity further without large NLL degradation across a wide range of penalty weights λ. Figure 8 shows this sparsity/NLL tradeoff for models of levels ranging from 2 to 10: the NLL barely degrades even beyond 90% sparsity, though the sparsity reached at a given λ varies with model scale.

Interestingly, we observe that larger models are easier to sparsify. Without sparsity-inducing loss, the "natural" sparsity of the model gradually increases, gaining 6 percentage points when scaling up from level 2 to level 10. At any fixed value of λ, the coefficient of the ℓ_1 loss, the sparsity also increases with respect to the model size. For instance, at $\lambda = 0.03$, the sparsity jumps of 10 percentage points between level 2 and 10, suggesting that larger models are easier to sparsify.

E SCALING LADDER

We provide in Table 12 the details of the scaling ladder used for all our experiments.

Table 12: **Scaling ladder.** Non-embedding parameters, architecture (FFN input dimension D, FFN hidden dimension H, number of layers L), peak learning rate γ and training budget per compute level. The cosine schedule decays to $\gamma/100$.

		Architecture				Training budget	
Scale	Non-Embed Params (B)	D	H	L	Peak LR (γ)	Train FLOPs	Total Tokens (B)
0	0.03	640	2048	7	2.06×10^{-3}	7.3×10^{18}	7.65
1	0.05	768	2048	8	1.87×10^{-3}	1.3×10^{19}	10.73
2	0.07	768	2048	11	1.70×10^{-3}	2.2×10^{19}	14.86
3	0.12	1024	3072	10	1.50×10^{-3}	5.1×10^{19}	24.33
4	0.17	1024	3072	14	1.35×10^{-3}	8.9×10^{19}	33.97
5	0.28	1280	4096	14	1.20×10^{-3}	1.9×10^{20}	52.85
6	0.37	1536	4096	15	1.07×10^{-3}	3.8×10^{20}	81.03
7	0.51	1536	4096	21	9.67×10^{-4}	6.9×10^{20}	113.62
8	0.81	1920	5120	21	8.66×10^{-4}	1.5×10^{21}	168.44
9	1.26	2304	6144	23	7.71×10^{-4}	3.1×10^{21}	253.76
10	1.84	2560	7168	26	6.98×10^{-4}	6.1×10^{21}	359.87
13	5.79	3840	10 240	38	5.01×10^{-4}	5.3×10^{22}	1141.52

F EVALUATION BENCHMARKS

We employ two types of benchmarks for zero or few-shot evaluation, which we describe herein, see Table 13 for a summary. The first type is code generation tasks: HumanEval+ (Liu et al., 2023a) and MBPP (Austin et al., 2021). The second type consists of commonsense and general reasoning: HellaSwag (Zellers et al., 2019), ARC (Clark et al., 2018), PIQA (Bisk et al., 2020), OBQA (Mihaylov et al., 2018), WinoGrande (Sakaguchi et al., 2020), NQ (Kwiatkowski et al., 2019), RACE (Lai et al., 2017) and TQA (Joshi et al., 2017). For the Qwen3 experiments of Appendix Q we additionally report three knowledge and commonsense benchmarks: MMLU (Hendrycks et al., 2021), CSQA (Talmor et al., 2019) and BoolQ (Clark et al., 2019). For the post-trained models of Appendix M we also report instruction-following with IFEval (Zhou et al., 2023).

Code generation We use two benchmarks that evaluate the code generation capabilities of AI models: HumanEval+ and MBPP.

- The HumanEval+ (Liu et al., 2023a) benchmark is an extension of HumanEval (Chen et al., 2021), which is designed to evaluate the functional correctness of code generated by AI models.
- MBPP (Austin et al., 2021) is designed to evaluate the code generation abilities of AI models, particularly for Python programming tasks.

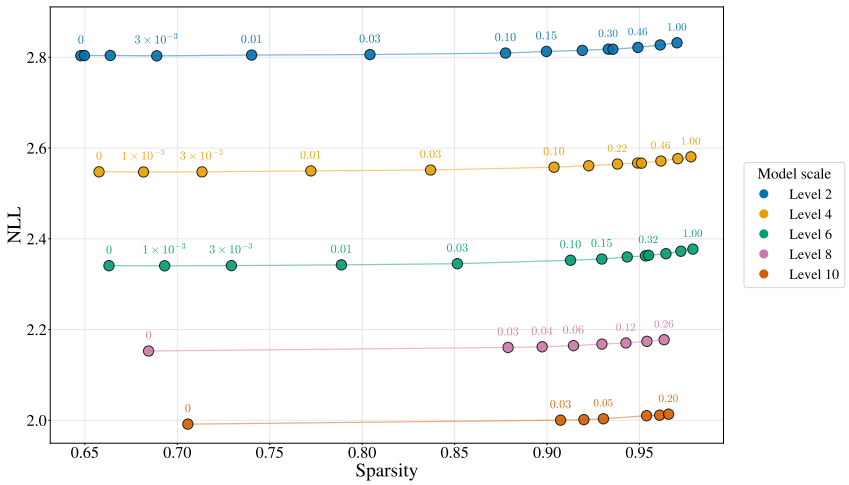


Figure 8: **Impact of a fixed ℓ_1 loss.** NLL as a function of sparsity for different fixed λ values, for level 2, 4, 6, 8 and 10 dense models mid-trained with Activation Casting. The numbers on the markers indicate λ, the weight of the ℓ_1 loss.

Table 13: **Summary of benchmarks used for evaluation.** We prompt level 8 models in a few-shot setting, and report a metric that depends on the benchmark, as well as the average performance. `acc_char` is the Character-Length Normalized Accuracy.

Benchmark	Metric	# Shots	Type
HellaSwag	`acc_char`	0	Choice
WinoGrande	`acc_char`	0	Choice
ARC-e	`acc_char`	0	Choice
ARC-c	`acc_char`	0	Choice
PIQA	`acc_char`	0	Choice
OBQA	`acc_char`	0	Choice
RACE-m	`acc_char`	0	Choice
RACE-h	`acc_char`	0	Choice
HumanEval+	pass@1	0	Generation
MBPP	pass@1	3	Generation
TriviaQA	f1	5	Generation
NaturalQ	f1	5	Generation

Common sense and general reasoning We use benchmarks consisting of question-answer or multiple-choice questions designed to evaluate the commonsense reasoning abilities of AI models, particularly in the context of natural language understanding: HellaSwag, ARC, PIQA, OBQA, WinoGrande, NaturalQuestions, RACE and TQA.

- HellaSwag (Zellers et al., 2019) consists of multiple-choice questions where each question contains a short context (a sentence or paragraph) followed by four possible continuations. Only one continuation is correct and makes sense given the context.
- The AI2 Reasoning Challenge (ARC) (Clark et al., 2018) benchmark consists of multiple-choice science questions typically found in elementary and middle school exams. The ARC questions require a mix of factual knowledge, commonsense reasoning, and multi-step inference.
- The Physical Interaction Question-Answering (PIQA) (Bisk et al., 2020) benchmark consists of multiple-choice questions about how to accomplish simple physical tasks. Each question presents a short scenario and two possible solutions, between which only one is physically plausible.
- The OpenBook QA (OBQA) (Mihaylov et al., 2018) benchmark consists of 6,000 multiple-choice questions based on elementary science facts. Each question is designed to require

combining a provided "open book" science fact with additional commonsense or general knowledge.

- WinoGrande (Sakaguchi et al., 2020) benchmark consists of multiple-choice questions. Each question presents a sentence with a pronoun and two possible antecedents; the task is to choose the correct referent for the pronoun.
- Natural Questions (NQ) (Kwiatkowski et al., 2019) benchmark is designed to evaluate the ability of an AI model to answer real user questions using information from Wikipedia.
- The Reading Comprehension from Examinations (RACE) (Lai et al., 2017) benchmark consists of passages and multiple-choice questions to assess how well AI models can comprehend and reason about written passages.
- The Trivia QA benchmark (TQA) (Joshi et al., 2017) is a reading comprehension dataset that pairs trivia questions with evidence documents from which answers can be derived.

Instruction following IFEval (Zhou et al., 2023) measures whether a model obeys explicit, programmatically checkable instructions: for instance, constraints such as a required word count, a forbidden word, or a specified output format. It consists of over 541 prompts. Such compliance examples are verified using a script rather than judged. This benchmark is only meaningful once a model has been trained to follow instructions, so we report it only for the post-trained models. We report `inst_acc`, the fraction of individual constraints satisfied, and `prompt_acc`, the fraction of prompts for which every constraint is satisfied.

Knowledge and commonsense The three benchmarks below are reported only for the Qwen3-1.7B experiments, where we probe the knowledge retained by the base model after mid-training. These are not part of the Core-12 average in Table 13.

- The Massive Multitask Language Understanding (MMLU) (Hendrycks et al., 2021) benchmark consists of multiple-choice questions spanning 57 subjects, from elementary mathematics to law and medicine. It is designed to measure the breadth of world knowledge and problem-solving ability acquired during pre-training. We report the macro-average over the 57 subtasks.
- The CommonsenseQA (CSQA) (Talmor et al., 2019) benchmark consists of multiple-choice questions derived from ConceptNet, where each question has one correct answer and four distractors that share a semantic relation with the same source concept, so that answering requires commonsense knowledge rather than surface cues.
- The BoolQ (Clark et al., 2019) benchmark consists of naturally occurring yes/no questions paired with a Wikipedia passage. Despite the binary answer format, the questions are unprompted and often require entailment-like inference over the passage.

G Architecture Casting vs Standard Gating at reduced parameter count

We compare the Architecture Casting models to SwiGLU models trained from scratch with a reduced number of parameters. This is achieved by reducing the hidden dimension of the model to $H = 3712$, so that the number of parameters per FFN is equal to the parameter count of the FFNs after Architecture Casting.

The results of this comparison are shown in Table 14. The gap between the dense baseline with reduced parameters and the Architecture casting models is initially large, because casting is highly disruptive: the model must simultaneously absorb a new activation and a reduced-rank gate. However, the gap narrows as the casting time increases, and Architecture casting with `SILU` eventually outperforms the baseline at iso-parameters.

This shows that Architecture Casting produces models with competitive NLL for their parameter count, and that the NLL degradation in Table 1 is to be attributed to parameter and sparsity differences, not to the architecture change during training.

Table 14: **Architecture Casting produces models with competitive NLL for their parameter count.** Impact of casting time ("MT" column) on the NLL of models. We compare level 8 models trained with Architecture Casting (three first NLL columns) against a dense SILU model trained from scratch with hidden dimension reduced to $H = 3712$ so that its FFN parameter count matches that of the cast models.

Training duration			Score: NLL				Δ NLL vs reduced params		
			no ℓ_1		ℓ_1	Reduced params	no ℓ_1		ℓ_1
PT	MT	Total	SILU	R-S+	R-S+	SILU	SILU	R-S+	R-S+
4T	0T	5T	2.240	2.255	2.276	**2.217**	+1.05%	+1.73%	+2.66%
4T	0.5T	5.5T	2.216	2.226	2.240	**2.207**	+0.42%	+0.87%	+1.53%
4T	1T	6T	2.203	2.211	2.222	**2.200**	+0.16%	+0.50%	+1.01%
4T	2T	7T	**2.197**	2.204	2.212	2.199	-0.09%	+0.22%	+0.61%
4T	4T	9T	**2.180**	2.186	2.193	2.184	-0.17%	+0.10%	+0.40%

H FULL RESULTS FOR DIFFERENT CASTING DURATIONS

In Table 15, we present the full results in a setup identical to Table 1, across different casting durations. All models are pre-trained for 4T, but have a casting phase ranging from 0T to 4T; we always finish the casting phase with 1T tokens of annealing, in which the learning rate is decayed. The results show that the gap between cast models and the SILU reference narrows as the casting period increases. In particular, Architecture Casting presents a wide initial gap due to the large architectural change in the model at casting time, but this gap quickly reduces as well.

Table 15: **Evolution of NLL during casting.** Detailed results when training a 0.81B dense model (level 8 in the scaling ladder from Table 12), performing the three model casting recipes. All models are pre-trained for 4T; the "Casting" column gives the mid-training budget and "Total" counts every token spent, including the $1T$ of decay. For LoPA and Architecture Casting, the rank ratio α is set to $1/8$. The ℓ_1 loss targets 90% sparsity at training time; the sparsity measured on the validation set is about one point lower, which is why the ℓ_1 columns read 89%. In addition to R-S+, we simulate a model casting with SILU as a control experiment.

Training duration		Score: NLL			R-S+ sparsity	
Casting	Total	SILU	R-S+	R-S+, ℓ_1	no ℓ_1	ℓ_1
(a) Activation Casting						
0T	5T	2.178	2.190	2.214	61%	89%
0.5T	5.5T	2.169	2.179	2.197	63%	89%
1T	6T	2.162	2.171	2.185	65%	89%
2T	7T	2.161	2.170	2.179	66%	89%
4T	9T	2.145	2.153	2.160	69%	89%
(b) LoPA Casting						
0T	5T	2.183	2.195	2.224	48%	89%
0.5T	5.5T	2.176	2.181	2.205	49%	89%
1T	6T	2.172	2.172	2.192	50%	89%
2T	7T	2.162	2.167	2.185	51%	89%
4T	9T	2.150	2.148	2.165	51%	89%
(c) Architecture Casting						
0T	5T	2.240	2.255	2.276	69%	89%
0.5T	5.5T	2.216	2.226	2.240	70%	89%
1T	6T	2.203	2.211	2.222	71%	89%
2T	7T	2.197	2.204	2.212	72%	89%
4T	9T	2.180	2.186	2.193	73%	90%

I VALIDATION NLL FOR A MODEL WITH 5.8B WEIGHTS

We validate the applicability of model casting to models of larger scale by applying it to 5.79B-parameter models (level 13 in Table 12). We use the same setup as for the level 8 models presented in Table 1 of the main text and in Table 15 of Appendix H.

In Table 16 we observe that LoPA Casting is able to achieve a 0.65% relative degradation in NLL with respect to the dense baseline, a smaller relative gap than for the 0.81B models. This is consistent with the findings of Figure 8: larger models are easier to sparsify, where a fixed λ penalty resulted in higher sparsity levels for larger models.

Table 16: **Scaling up model casting.** Results for 5.79B-parameter models (level 13 in Table 12). ΔNLL is computed relative to the first row in **bold**, corresponding to a control experiment conducted with `SILU` throughout training. All models are pre-trained for 4T tokens, and mid-trained for 4T tokens of stable learning rate, followed by 1T tokens of learning rate decay.

Setup (pre-training $\rightarrow$ mid-training)	NLL	ΔNLL	Sparsity	Speedup
Base[`SILU`] $\rightarrow$ Base[`SILU`] (dense baseline)	**1.781**	—	—	1.00×
LoPA[`SILU`] $\rightarrow$ LoPA[`R-S+`, ℓ_1@90%]	1.792	+0.65%	89%	5.86×

J SHORT-CONTEXT EVALUATIONS

In this section, we provide short-context evaluations to complement the validation NLL results presented in Table 15. The benchmarks and their configurations are described in Appendix F.

Models trained from scratch. In Table 18 we report the Core-12 benchmarks for the level 8 models of the scaling ladder, pre-trained for $4T$ and then mid-trained for a further $4T$ followed by $1T$ of decay. We use a different data mixture during mid-training and we compute the deltas for model casting with respect to the `SILU` control consisting of exactly the same mid-training budget without model casting, so that we can measure the cost of model casting alone. That cost is small: averaged over the twelve benchmarks, Activation Casting loses 0.1 points at a 75% sparsity target and 0.4 at 90%, and LoPA Casting loses 0.2 at 75% and gains 0.1 at 90%. No individual benchmark moves by more than 2.6 points in either direction. Note that the two model casting families disagree on the individual benchmarks that move, this suggests that what is left is largely run-to-run noise rather than a systematic loss. For reference, mid-training itself is worth considerably more than model casting costs: it lifts the average from 38.2 to 39.6 for Activation Casting and from 38.0 to 39.0 for LoPA Casting.

Extra knowledge benchmarks for Qwen3. Since Qwen3-1.7B-Base is pre-trained far beyond our scaling-ladder budgets, it retains knowledge that the ladder models never acquire: level 8 scores 38.2 on the Core-12 average, close to chance on the harder multiple-choice tasks, whereas Qwen3-1.7B-Base reaches 65.0 on MMLU, 74.9 on CSQA and 75.6 on BoolQ. We therefore add these three knowledge and commonsense benchmarks to the Core-12 suite, to check whether mid-training erodes what the base model already knows.

Qwen3-1.7B. In Table 17 we report the downstream performance of an off-the-shelf checkpoint at the decayed $5T$ mark (mid-training details in Appendix Q), where the control is the *`SILU, no swap`* arm: the same mixture and budget, but keeping the native `SILU` activation. We again observe that the mid-training mixture contributes to a performance degradation more significantly than model casting. Concretely, it degrades the Core-12 mean from 51.9 to 49.4 and the knowledge-3 mean from 71.8 to 63.3, a loss of 2.5 and 8.5 points, whereas applying model casting on top costs only 1.6 and 0.7 points respectively; the activation swap without the ℓ_1 loss costs 0.4 on Core-12 while gaining 1.0 on knowledge-3. The cost concentrates in code generation, where Activation Casting gives up 4.3 points on HumanEval+ and 3.8 on MBPP; on the knowledge benchmarks it loses at most 3.0 points and improves on BoolQ. Qualitatively this matches the from-scratch models, e.g. the mixture dominates and casting is second order.

K CATCH UP TIMES AT MID-TRAINING

In Table 19, we report the catch-up duration across model scales for a fixed 4T pre-training budget, for both Activation Casting and LoPA Casting.

Table 17: **Effect of casting mid-trained `Qwen3-1.7B` downstream performance.** Change in downstream performance (percentage points) at the decayed $5T$ checkpoint (100 TPP), at peak LR $\gamma = 7.40 \times 10^{-5}$. Deltas are taken against the *`SILU`, no swap* control (the same model mid-trained on the same mixture without the activation swap) so they isolate the cost of casting from the cost of the mid-training mixture.

			Δ vs. `SILU` control (points)	
Task	Qwen3-1.7B-Base (pre-trained)	`SILU`, no swap (control)	`R-S+` (no ℓ_1)	Activation Casting ℓ_1@90%
HellaSwag	66.5	69.7	−0.7	−1.2
ARC-e	68.1	72.7	−1.6	−1.6
ARC-c	45.1	45.0	−0.2	**−2.5**
PIQA	76.1	76.8	+0.3	−0.9
WinoGrande	64.5	65.3	−0.6	+0.0
OBQA	38.6	39.6	+0.2	**−2.2**
RACE-m	63.9	58.4	+0.3	−1.0
RACE-h	46.7	42.0	+0.2	+0.1
HumanEval+	39.6	23.8	−1.2	**−4.3**
MBPP	55.2	40.0	−0.6	**−3.8**
TriviaQA	38.7	40.6	−0.7	−1.9
NaturalQ	19.5	18.6	−0.1	−0.5
Core-12 mean	**51.9**	**49.4**	**−0.4**	**−1.6**
Knowledge / commonsense				
MMLU	65.0	57.2	−0.8	**−3.0**
CSQA	74.9	65.6	+0.2	**−2.0**
BoolQ	75.6	67.2	**+3.5**	**+2.8**
Knowledge-3 mean	**71.8**	**63.3**	**+1.0**	**−0.7**

Table 18: **Effect of casting mid-trained level-8 models on downstream performance.** Within each family the deltas are taken against that family's *4T* column, that is the same model mid-trained for the same budget without casting. The *0T* and *4T* columns are absolute %: *0T* is the pre-trained checkpoint (4T WS + 1T D) before mid-training, so comparing the two gives the gain from mid-training alone. All rows are decayed. Bold marks changes of at least 2 points. Per-benchmark metrics are given in Table 13.

	Activation Casting				LoPA Casting			
Task	0T	4T	75%	90%	0T	4T	75%	90%
HS	57.0	59.1	−0.4	−1.1	56.7	59.1	−1.0	−1.6
WG	59.1	59.4	−0.2	−0.8	57.1	58.2	−0.3	+1.5
ARC-e	60.3	61.5	−0.8	−1.4	57.3	57.8	**+2.1**	**+2.6**
ARC-c	31.2	34.2	−1.0	−1.5	31.0	31.9	+0.6	+0.3
PIQA	72.6	73.0	+0.5	+0.5	73.0	74.0	−0.5	−0.5
OBQA	34.2	35.8	+0.2	+1.0	36.6	36.0	+0.6	−0.8
RACE-m	52.2	53.5	−0.2	−0.8	50.6	52.3	−0.3	+0.1
RACE-h	38.3	39.5	−0.5	−0.6	38.4	39.2	+0.4	−0.2
HE+	6.7	6.7	**+2.4**	+1.8	7.9	7.3	−1.2	+1.8
MBPP	10.8	13.0	−0.2	−0.4	11.6	12.2	−1.4	+0.4
TQA	24.1	26.4	+0.0	−0.6	24.3	27.0	−0.6	−1.3
NQ	11.6	12.6	−0.7	−0.6	11.5	13.1	−0.7	−1.3
Average	**38.2**	**39.6**	−0.1	−0.4	**38.0**	**39.0**	−0.2	+0.1

Table 19: **Mid-training duration required to catch up** with the SILU model's performance, without the decay phase, for different model sizes with pre-trained models of 4T duration. Models are evaluated every 0.25T, and we report the first time at which the finetuned model catches up to the original NLL, rounded up by at most 0.25T. When the ℓ_1 loss is applied (rightmost columns), fixing the target sparsity at 90% during training leads to a measured sparsity around 89%. For LoPA Casting, the rank ratio α is set to $1/8$.

		Casting without ℓ_1			Casting with ℓ_1		
Model scale	NLL	Duration	NLL	Sparsity	Duration	NLL	Sparsity
(a) Activation Casting							
2	3.013	1.00T	3.011	66%	1.50T	3.004	89%
4	2.738	0.75T	2.725	67%	2.00T	2.735	89%
6	2.521	1.00T	2.518	67%	1.50T	2.516	89%
8	2.321	1.25T	2.317	70%	1.75T	2.313	89%
10	2.146	1.25T	2.144	71%	2.00T	2.141	89%
(b) LoPA Casting							
2	3.009	0.75T	3.007	48%	1.50T	3.001	89%
4	2.743	0.50T	2.742	48%	2.00T	2.739	89%
6	2.523	0.75T	2.520	52%	1.75T	2.515	89%
8	2.328	0.75T	2.323	56%	1.75T	2.317	89%
10	2.145	1.00T	2.141	57%	2.75T	2.137	89%

L QUANTIZATION-AWARE TRAINING

We demonstrate that post-activation sparsity is compatible with quantization by applying Quantization-Aware Training (QAT) on top of models trained with our three model casting recipes.

Method. We consider 0.81B models (level 8 in Table 12) pre-trained with SILU for 4T, fine-tuned with model casting for 5T (4T of stable and 1T of decaying learning rate). We then apply a simple weight quantization approach where weights are converted either to int8 or int4 at inference. During QAT, we quantize and de-quantize the weights to prepare the model for inference. We apply QAT for 0.5T; optimizer state is reset, a warmup is done for 10% of QAT, and the learning rate then follows a cosine decay. During QAT, the ℓ_1 loss is kept to maintain the target sparsity level. We validate the practical gains of combining quantization and post-activation sparsity by showcasing the performances of low-precision sparse kernels.

Results. Table 20 reports the performance of the different models with and without QAT, at different precisions. For each casting family, we report both the activation swap (R-S+ + ℓ_1@90%), as well as a reference experiment where we use SILU throughout training. Note that Architecture casting underperforms due to its reduced number of parameters. Interestingly, quantizing sparse models is slightly less costly than dense models. For instance, quantizing to int8 the SILU reference of Activation casting costs 0.015 nats, while quantizing the sparse equivalent (R-S+ + ℓ_1@90%) only costs 0.013 nats; the same numbers hold for LoPA casting. Therefore, combining both methods results in sub-additive NLL degradations, suggesting that both methods are independent and compatible.

Figure 9 demonstrates major speedups when simultaneously exploiting post-activation sparsity and low-precision inference. Unlike the NLL results above, these kernel measurements are taken on a larger 5.79B model (level 13 in Table 12). Without sparsity, quantizing the weights to int8 and int4 provides $2.81\times$ and $3.35\times$ FFN speedups, respectively, while LoPA in FP32 results in a $3.31\times$ speedup at 90% sparsity. Combined, we obtain speedups of $8.13\times$ (int8) and $8.51\times$ (int4) at 90% sparsity compared to the full-precision dense baseline.

Table 20: **Impact on NLL of quantization and casting.** After Activation, LoPA, and Architecture casting, we apply QAT and quantize models in a weight-only `int4` and `int8` setting. Degradation is relative to the activation-casting `SILU` model at full precision in **bold**.

	FP32		`int8`		`int4`	
Setup (pre-training → mid-training)	NLL	ΔNLL (%)	NLL	ΔNLL (%)	NLL	ΔNLL (%)
Base[`SILU`] → Base[`SILU`]	**2.145**	—	2.160	+0.70%	2.178	+1.54%
Base[`SILU`] → Base[`R-S+`, ℓ_1@90%]	2.160	+0.68%	2.173	+1.29%	2.188	+2.01%
LoPA[`SILU`] → LoPA[`SILU`]	2.147	+0.09%	2.162	+0.76%	2.180	+1.62%
LoPA[`SILU`] → LoPA[`R-S+`, ℓ_1@90%]	2.166	+0.99%	2.179	+1.57%	2.194	+2.29%
Base[`SILU`] → LoPA[`SILU`]	2.180	+1.63%	2.193	+2.21%	2.213	+3.16%
Base[`SILU`] → LoPA[`R-S+`, ℓ_1@90%]	2.193	+2.21%	2.205	+2.76%	2.222	+3.59%

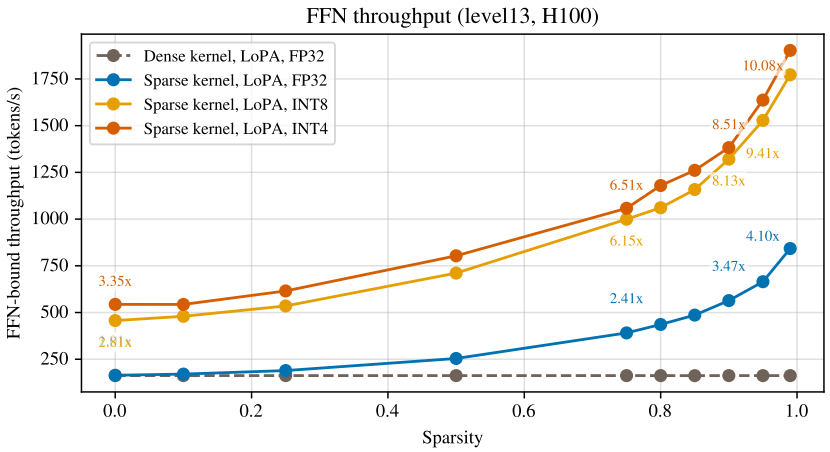


Figure 9: **Measured speedups for quantized sparse models.** FFN-bound throughput as a function of the activation sparsity, for a 5.79B non-embedding parameters model (level 13 in the scaling ladder from Table 12) at different quantization levels (`int8` and `int4`).

M Post-training

LCFT and SFT. To check that model casting survives post-training, we use our own mid-trained level 8 models (starting from the corresponding pre-trained models from the scaling ladder of Table 12), with and without model casting, and apply a standard post-training pipeline: long-context fine-tuning (LCFT) followed by supervised fine-tuning (SFT). Six variants are compared throughout: Activation Casting and LoPA Casting at 75% and 90% sparsity targets versus a dense control with `SILU`.

Table 21: Average downstream performance across post-training, relative to the `SILU` dense baseline. Δ_{pre} is the gap before post-training, Δ_{SFT} the gap after 8000 SFT steps, and $\Delta_{\text{corr}} = \Delta_{\text{SFT}} - \Delta_{\text{pre}}$ isolates what post-training itself costs each variant. All values are rounded to one decimal, so the columns reconcile exactly.

	Core-12 average			relative to the dense baseline		
Setup (pre-training → mid-training)	pre-LCFT	LCFT	SFT	Δ_{pre}	Δ_{SFT}	Δ_{corr}
Base[`SILU`] → Base[`SILU`]	39.6	39.7	36.6	—	—	—
Base[`SILU`] → Base[`R-S+`, ℓ_1@75%]	39.5	39.0	36.8	−0.1	+0.2	+0.3
Base[`SILU`] → Base[`R-S+`, ℓ_1@90%]	39.2	38.7	36.0	−0.4	−0.6	−0.2
LoPA[`SILU`] → LoPA[`R-S+`, ℓ_1@75%]	38.8	38.6	36.1	−0.8	−0.5	+0.3
LoPA[`SILU`] → LoPA[`R-S+`, ℓ_1@90%]	39.1	38.1	36.0	−0.5	−0.6	−0.1

Table 21 gives the Core-12 evaluation tasks average relative to the dense baseline, before post-training and after 8000 SFT steps. After SFT, the model casting variants that target 90% sparsity have a small degradation of 0.1–0.2 with respect to the dense baseline. However, both Activation Casting and LoPA Casting @75% improve by 0.3 after SFT with respect to the dense baseline. In contrast, the LCFT stage is close to neutral, moving the average by at most 1.0 point in either direction, so SFT accounts for almost all of the change. SFT is expensive for every model alike, costing 2.4 to 3.2 points by step 8000. Hence, the degradation is due to the SFT datamix rather than model casting.

We also run a suite of chat-format evaluations, of which three carry positive instruction-following signal at this scale: `cruxeval_output`, `gsm8k_chat` and `mbpp_chat`. Across the six variants, these agree to within 1.6, 4.0 and 3.2 points at step 8000, so model casting does not measurably change chat behavior. In particular, on `gsm8k_chat` the chat model beats the base model by a wide margin (7.8 against 4.9), confirming that SFT trains the chat template. We do not report the multiple-choice chat evaluations, which sit at or below chance, nor a format-compliance check that saturates on every variant. Finally, in Table 22, we report instruction-following abilities on IFEval (Zhou et al., 2023) after 8000 SFT steps. Note that we can only conduct this evaluation after SFT has trained the chat template. Both pre-training and LCFT use the plain `tiktoken` tokenizer. Every Δ is measured against the single dense control in the first row, so the LoPA Casting entries also carry the architecture difference. We can conclude that all model casting variants retain measurable instruction-following ability. In particular, the LoPA Casting at 75% does better than the dense baseline. There is some degradation for the 90% model casting variants which could be attributed to using such an extreme sparsity operating point.

Table 22: IFEval after 8000 SFT steps. Scores are *loose*; the strict variant penalizes formatting alone and is largely common-mode across variants. Δ is measured against the Activation Casting dense control in the first row, a single reference for the whole table, so the LoPA Casting rows also carry the architecture difference.

	`inst_acc`		`prompt_acc`	
Setup (pre-training $\rightarrow$ mid-training)	score	Δ	score	Δ
Base[`SILU`] $\rightarrow$ Base[`SILU`]	33.9	—	22.9	—
Base[`SILU`] $\rightarrow$ Base[`R-S+`, ℓ_1@75%]	33.9	+0.0%	22.0	−3.9%
Base[`SILU`] $\rightarrow$ Base[`R-S+`, ℓ_1@90%]	32.0	−5.6%	21.1	−7.9%
LoPA[`SILU`] $\rightarrow$ LoPA[`R-S+`, ℓ_1@75%]	35.1	+3.5%	23.3	+1.7%
LoPA[`SILU`] $\rightarrow$ LoPA[`R-S+`, ℓ_1@90%]	32.5	−4.1%	19.2	−16.2%

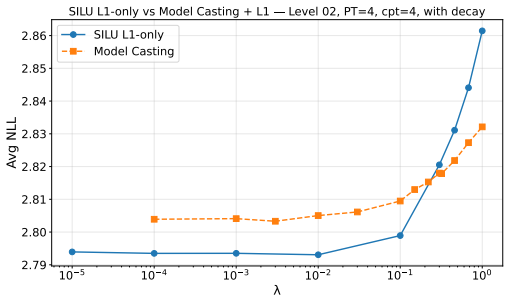
N Enforcing sparsity with ℓ_1 loss only

Figure 10 shows that enforcing sparsity by only adding an ℓ_1 loss to the cross-entropy loss, a) incurs a higher performance degradation at a given sparsity penalty, and b) does not result in significant sparsity levels.

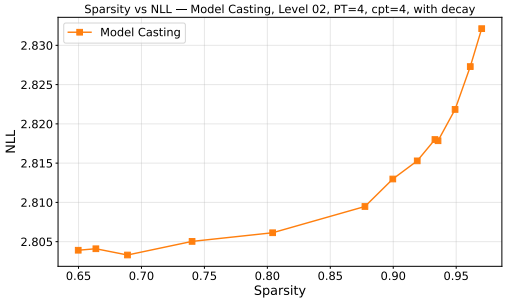
O Ablating the use of ℓ_1 loss during pre-training

In this section, we ask whether applying an ℓ_1 loss during pre-training with `SILU`, before the model casting phase, could be beneficial. To do so, we train models from scratch for 4T using an ℓ_1 penalty loss on the output of `SILU`; we then apply activation casting for 5T tokens (4T of stable learning rate schedule, and an annealing phase of 1T), using `R-S+` and ℓ_1 loss with the same weight as during pre-training.

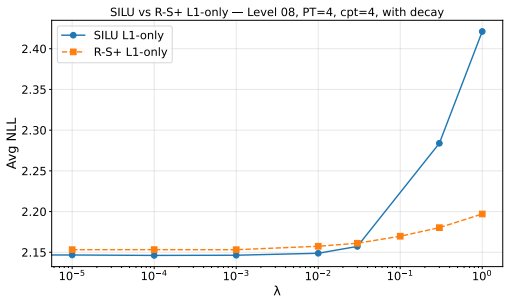
Table 23 shows that adding the ℓ_1 loss during pre-training results in similar or worse NLL (up to training noise), and reduces the sparsity level after casting. Furthermore, using ℓ_1 loss during pre-training prevents the use of an off-the-shelf model, as it requires a different training recipe from scratch. Therefore, these results indicate that applying model casting as a mid-training step only is sufficient.



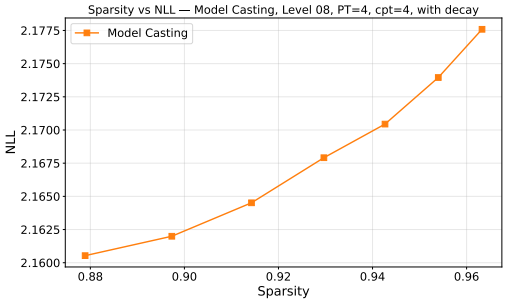
(a) NLL vs sparsity penalty for model casting +ℓ_1 vs ℓ_1 only.



(b) Sparsity vs NLL with model casting + ℓ_1



(c) NLL vs sparsity penalty for model casting +ℓ_1 vs ℓ_1 only.



(d) Sparsity vs NLL with model casting + ℓ_1

Figure 10: ℓ_1 loss without model casting induces higher performance degradation for the same sparsity penalty λ with level 2 (top) and level 8 (bottom) models.

Table 23: **Using ℓ_1 loss during pre-training degrades final sparsity.** Ablations of using the ℓ_1 loss during pre-training. Models are pre-trained and cast with the same ℓ_1 weight λ. Adding the ℓ_1 loss on the output of `SILU` during pre-training results in similar or worse NLL (up to training noise), and reduces the sparsity level after casting.

		ℓ_1 loss weight (λ)					
Metric	ℓ_1 during PT	10^{-4}	3×10^{-4}	10^{-3}	3×10^{-3}	10^{-2}	3×10^{-2}
Level 2							
NLL	✓	**2.749**	**2.748**	**2.747**	2.750	2.751	2.754
NLL	✗	2.750	2.750	2.750	**2.749**	**2.750**	**2.753**
Sparsity	✓	**65.3%**	65.6%	66.7%	68.1%	71.0%	73.6%
Sparsity	✗	65.2%	**65.7%**	**66.8%**	**69.3%**	**74.5%**	**80.9%**
Level 8							
NLL	✓	**2.159**	**2.160**	**2.159**	**2.160**	2.164	2.173
NLL	✗	2.160	2.161	2.162	2.162	**2.163**	**2.166**
Sparsity	✓	**73.6%**	**74.1%**	74.5%	78.5%	**83.8%**	88.3%
Sparsity	✗	**73.6%**	**74.1%**	**76.6%**	**80.0%**	**83.8%**	**88.7%**

P CASTING WITH OTHER ACTIVATION FUNCTIONS

Model casting only requires that the replacement activation produce exact zeros; `R-S+` is one choice among many. Here we cast the same pre-trained models with a range of alternative activations and report the resulting NLL and FFN activation sparsity in Table 24, for a level 2 and a level 8 model, across mid-training budgets of $1T$, $2T$ and $4T$, with and without the final decay. All runs share the pre-trained checkpoint and differ only in the activation used at casting time.

The comparison separates two properties that are easily conflated. SILU, G-S+ and GELU give the best NLL, but they are not sparsifying activations at all, so they bound the attainable quality rather than competing as candidates. Among the sparsifying activations, Shifted RELU reaches the highest sparsity (75–79%) but pays for it in NLL, and the p-RELU family trades sparsity away as p grows, from 66% at $p = 1.189$ down to 51% at $p = 1.681$. R-S+ sits at the useful point of that trade-off: its NLL tracks SILU to within a few thousandths at every budget, while still delivering 65–69% sparsity before any ℓ_1 penalty is applied. Notably squaredRELU, which Zhang et al. (2024) advocate for from-scratch training, gives the *lowest* exact-zero sparsity here (40%), which is why we do not adopt it for casting.

Table 24: **SILU and GELU result in the best performances for casting.** NLL and FFN activation sparsity when casting with different activation functions. Both models are pre-trained for $4T$; each NLL column is a mid-training budget, and "+D" adds the final $1T$ decay. Sparsity varies little with the budget, so we report it at the decayed checkpoint only. SILU, G-S+ and GELU are not sparsifying activations and have no exact zeros to report; they bound the attainable NLL rather than competing on sparsity. Bold marks the best NLL per column.

	Level 2					Level 8				
	NLL↓				Spars.	NLL↓				Spars.
Activation	$1T$	$2T$	$4T$	+D	+D	$1T$	$2T$	$4T$	+D	+D
R-S+	3.011	2.984	2.966	2.804	65%	2.326	2.307	2.295	2.153	69%
SILU	**2.999**	**2.970**	2.945	**2.794**	—	2.326	2.307	2.295	2.153	—
G-S+	**2.999**	2.974	**2.944**	2.808	—	**2.314**	2.298	**2.287**	2.148	—
GELU	**2.999**	2.973	2.955	2.808	—	**2.314**	**2.297**	**2.287**	**2.147**	—
pRELU (p=1.189)	3.022	2.999	2.956	2.811	66%	2.327	2.309	2.295	2.154	70%
pRELU (p=1.414)	3.012	2.983	2.954	2.804	59%	2.325	2.306	2.293	2.152	62%
pRELU (p=1.681)	3.013	2.984	2.966	2.803	51%	2.326	2.306	2.292	2.151	52%
squaredRELU	3.014	2.989	2.966	2.803	40%	2.327	2.306	2.291	2.151	40%
0.5 RELU	3.022	2.999	2.960	2.811	71%	2.330	2.311	2.298	2.158	75%
RELU	3.022	2.990	2.961	2.811	72%	2.330	2.312	2.299	2.159	76%
Shifted RELU (δ=1.2)	3.020	2.991	2.960	2.811	75%	2.334	2.313	2.301	2.160	79%

Q QWEN3 MID-TRAINING DETAILS

We apply our existing mid-training recipe, in terms of the scaling ladder Chinchilla units ($T = 10\,400$, for the model that is closest in size, level 8, which has 0.81B non-embedding parameters, this corresponds to 20 TPP) as follows: we mid-train for 4T of warmup-stable (from which T/5 is for warmup) and then with decay for 1T (this corresponds to mid-training for 100 TPP).

Short sweep for determining the learning rate for mid-training. We conduct a sweep to determine which learning rate least degrades the validation NLL; the results of this sweep are presented in Tables 25 and 26, respectively for short context evals and validation NLL.

Qwen3 mid-training with $\gamma = 7.4 \times 10^{-5}$. Validation NLL is reported in Table 27; the corresponding short-context evaluations are in Table 17 of Appendix J.

Validation NLL. In Tables 26 and 27 we report NLL with respect to different data mixtures: *In-mixture* covers five datasets included in the training data mixture; *All 9* averages nine evaluation datasets; *GovReport* is an out-of-mixture probe, reported separately but included in *All 9*. The nine therefore comprise the five in-mixture datasets and four out-of-mixture probes, of which *GovReport* is the one we single out.

Table 25: **Short context evaluations for different peak-lr for mid-training Qwen3-1.7B.** The evaluation is computed post-cooldown at step 1,250. All arms branch from `Qwen3-1.7B-Base` with the `R-S+AO` activation swap. Bold marks the best arm per task; * marks the chosen LR.

	Peak LR γ					
Task	1.00×10^{-5}	7.40×10^{-5} *	1.05×10^{-4}	1.48×10^{-4}	2.50×10^{-4}	7.40×10^{-4}
Core-12 short-context benchmarks						
HSwag	.6279	.6451	**.6458**	.6439	.6359	.5969
ARC-e	.7268	.7281	**.7328**	.7230	.7087	.6283
ARC-c	.4489	.4575	**.4592**	.4403	.4206	.3519
PIQA	.7361	**.7503**	.7470	.7448	**.7503**	.7388
WinoG	.6219	**.6306**	.6267	**.6306**	.6235	.6140
OBQA	.3820	.3980	.4000	**.4020**	.3780	.3680
RACE-m	**.6121**	.6100	.6038	.5968	.5815	.5306
RACE-h	**.4437**	.4342	.4234	.4185	.4177	.3848
HE+	**.2988**	.2744	.2927	.2866	.2805	.1646
MBPP	.4020	.4200	**.4320**	.4160	.3920	.2420
TQA	.3207	.3401	**.3402**	.3352	.3175	.2544
NQ	.1527	.1571	**.1580**	.1572	.1542	.1229
Mean	.4811	.4871	**.4885**	.4829	.4717	.4164
Knowledge / commonsense						
MMLU	.5749	**.5920**	.5904	.5841	.5619	.4245
CSQA	.6830	**.6880**	.6749	.6806	.6585	.5389
BoolQ	**.6911**	.6749	.6719	.6725	.6685	.6443

Table 26: **Validation NLLs using different peak-lr for mid-training Qwen3-1.7B.** Held-out NLL per token and FFN activation sparsity for Qwen3-1.7B. Every arm runs $0.093T$ (1,250 steps, 2.62 B tokens) without ℓ_1 loss. Bold marks the best arm per NLL column; * marks the chosen learning rate.

Peak LR γ	In-mixture	Δ vs. base	All 9	GovReport	Sparsity (%)
Pre-trained BASE	1.711	—	1.843	2.152	—
Swap, 0 steps	10.256	+8.544	9.996	9.788	76.77
1.00×10^{-5}	1.761	+0.050	1.927	2.212	79.79
7.40×10^{-5} *	**1.724**	**+0.013**	**1.895**	**2.188**	78.59
1.05×10^{-4}	1.727	+0.015	1.898	2.190	78.60
1.48×10^{-4}	1.734	+0.022	1.906	2.195	78.76
2.50×10^{-4}	1.757	+0.046	1.932	2.213	79.40
7.40×10^{-4}	1.871	+0.159	2.047	2.298	82.96

Table 27: **Mid-training Qwen3-1.7B with model casting.** Held-out NLL per token and FFN activation sparsity for Qwen3-1.7B mid-training at the swept-optimum peak $\gamma = 7.40 \times 10^{-5}$. Runs differ only in FFN activation and sparsity penalty: `R-S+` is the swap with no penalty, "Activation Casting @90%" adds an adaptive-ℓ_1 penalty targeting 90% sparsity, and *SILU, no swap* is plain mid-training. Bold marks the best trained checkpoint per NLL column.

Setup (pre-training → mid-training)	Budget	In-mixture	Δ vs. base	All 9	GovReport	Sparsity (%) (All 9)
Base[SILU] *(pre-trained)*	—	1.711	—	1.843	2.152	—
Base[SILU] → Base[R-S+]	$0T$	10.256	+8.544	9.996	9.788	76.77
	$1T$	1.702	−0.010	1.881	2.163	76.84
	$2T$	1.698	−0.013	1.881	2.155	76.43
	$3T$	1.699	−0.012	1.884	2.151	76.25
	$4T$	1.701	−0.011	1.887	2.151	76.11
	$5T^{\dagger}$	1.665	−0.046	1.854	2.116	75.18
Base[SILU] → Base[R-S+, ℓ_1@90%]	$1T$	1.730	+0.019	1.912	2.190	89.57
	$2T$	1.722	+0.010	1.908	2.179	89.62
	$3T$	1.719	+0.008	1.906	2.172	89.57
	$4T$	1.719	+0.007	1.907	2.168	89.60
	$5T^{\dagger}$	1.685	−0.027	1.875	2.136	89.54
Base[SILU] → Base[SILU]	$1T$	1.689	−0.023	1.868	2.155	—
	$2T$	1.690	−0.022	1.873	2.150	—
	$3T$	1.692	−0.019	1.878	2.147	—
	$4T$	1.695	−0.017	1.881	2.146	—
	$5T^{\dagger}$	**1.660**	**−0.051**	**1.849**	**2.113**	—

R GPU SPEEDUP IN BFLOAT16

As inference is frequently done in bfloat16, we also provide in Figure 11 the run times observed for this format.

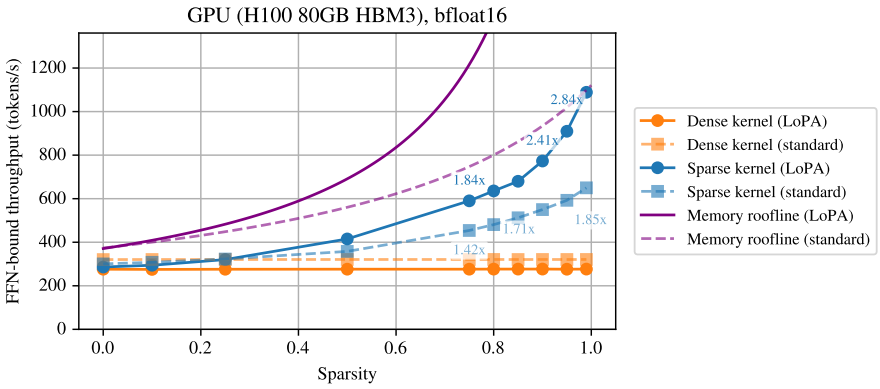


Figure 11: **Measured speedups in bfloat16.** Throughput as a function of the activation sparsity, for a 5.79B non-embedding parameters model (level 13 in the scaling ladder from Table 12) in bfloat16. At 90% sparsity the speedup on GPU is 1.71× for Standard Gating and 2.41× for LoPA Gating compared to the standard dense case.