TwIST: Rigging the Lottery in Transformers via Zero-Shot Subnetworks

Anonymous authors
Paper under double-blind review

Abstract

We introduce TwIST, a novel distributed system for efficient training and deployment of Transformer-based models. TwIST unifies two important paradigms: distributed training and pruning, to induce an inference time property that allows us to randomly sample competitive subnetworks on the fly. TwIST trains randomly generated subnetworks in parallel, periodically aggregating and resampling, yielding high-performance subnets that require no fine-tuning or search heuristics. This enables robust, zero-cost pruning at deployment, achieving perplexity scores close to state-of-the-art post-training methods while bypassing their post-training overhead (e.g., calibration, Hessian inversion). Crucially, even when post-training baselines operate on a *stronger* dense checkpoint than TwIST, they still lose at aggressive sparsity, often by orders of magnitude, evidence that designing training around subnet structure produces better sparse models than searching for them after the fact. We validate this across three architectures (GPT-2 and GPT-2 XL, with multi-head attention and Qwen3-0.6B with grouped-query attention) over a wide range of text-generation tasks. TwIST's advantage emerges under aggressive pruning (e.g., 50%+ sparsity), where it significantly outperforms baselines. For example, on GPT-2/WikiText-103 it achieves 23.14 PPL while the closest baseline follows at 31.64. At the most aggressive Qwen3-0.6B pruning setting, TwIST reduces perplexity by approximately 2.5–$4.6\times$ relative to the strongest evaluated baseline, as those baselines collapse entirely. Furthermore, as a *structured pruning* method, TwIST produces smaller, dense matrices, translating to tangible inference speedups and memory savings on inference server deployments (e.g., CPUs) that lack sparse computation support. We provide the complete implementation here.

1 Introduction

Transformer-based models (OpenAI et al., 2024; Brown et al., 2020) have reshaped AI through their performance across an extraordinary range of tasks, with the GPT family in particular showing strong generalization on diverse and complex language benchmarks (II & Katz, 2022; Chen et al., 2021; Wei et al., 2022). But this performance comes at a steep cost. Training and deploying these models is computationally enormous. Large-scale LLM training almost exclusively relies on distributed training. Meanwhile, the gap between the hardware that produces these models and the hardware most users own continues to widen. Closing that gap, and democratizing access to capable models, has driven substantial research in both distributed training and model compression. On the compression side, the two dominant levers are *quantization* (Dettmers et al., 2022; Frantar et al., 2023; Ahmadian et al., 2023; Yao et al., 2022), which lowers the numerical precision of stored weights, and *pruning*, which removes parameters outright. The two are complementary, but here we focus on pruning, where the question of *which* parameters can be removed without harm is sharper and less settled.

The most evocative framing of pruning (LeCun et al., 1989; Hassibi et al., 1993) comes from the Lottery Ticket Hypothesis (LTH) (Frankle & Carbin, 2019): dense networks contain sparse "winning tickets" that can match full-model accuracy when trained from their original initialization. A decade of follow-up has largely treated pruning as a *search problem*, with methods differing in when the search runs and what it costs. At LLM scale, post-training pruning (PTP) (Frantar & Alistarh, 2023; Sun et al., 2024; Bhuiyan et al., 2025) compresses a finished checkpoint without retraining, but still requires calibration and, for some methods, per-layer reconstruction. Iterative magnitude pruning with rewinding (Renda et al., 2020) alternates pruning and retraining, while sparsity-inducing training

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.

(Evci et al., 2021; Sanh et al., 2020; Liu et al., 2022a) bakes the search into the optimizer through regularization or masking. At the other extreme, init-time methods such as SNIP (Lee et al., 2019) and SynFlow (Tanaka et al., 2020) select the subnet before training begins. These methods differ in cost structure but share a common premise: training produces one model, and sparsity is something you either extract from it, sculpt it toward, or commit to beforehand. Recent results suggest this premise is worth questioning. Gale et al. (2019) found that simple magnitude pruning can match or outperform more complex sparsification methods at scale, while Prasanna et al. (2020) found that randomly extracted subnetworks of pretrained BERT models (Devlin et al., 2019) remain highly trainable and competitive with carefully extracted subnetworks. Random pruning at initialization has likewise been studied (Su et al., 2020; Liu et al., 2022b; Gadhikar et al., 2023).

These threads point to a question that the pruning literature has mostly side-stepped: **Is the search always necessary, and is retraining always required?** Rather than train a dense model and hunt for a subnet that survives compression, can we train a model whose randomly sampled structured subnets perform competitively at the evaluated sparsity levels, without retraining, calibration, or search? This would build compressibility directly into training while keeping the parent model intact and allowing deployment sizes to be chosen afterward. A single checkpoint could then supply an on-the-fly spectrum of "zero-shot subnetworks" tailored to different compute budgets, from datacenter GPUs to workstations and smartphones.

Related Work. The idea of inducing structural robustness during training to support random subnetwork extraction has been studied through dropout and shared-weight training (Srivastava et al., 2014; Gomez et al., 2019; Yu et al., 2018; Fan et al., 2019). LayerDrop (Fan et al., 2019) randomly removes Transformer layers during training, enabling inference-time depth reduction without additional fine-tuning. Slimmable Networks (Yu et al., 2018) preserve depth and train overlapping, shared-weight CNN subnetworks at different widths, enabling on-the-fly width adjustment. A closer and more recent method, LLaMaFlex (Cai et al., 2025), showed that zero-shot pruning can also be enabled in large-scale Transformers based models using a learned router to select configurations for a parameter budget and modulation networks to adjust activations for the selected configuration. These methods operate under different setups, but are motivated by the broader goal of enabling zero-shot subnetwork extraction directly within the parent model, sidestepping conventional search-and-retrain compression pipelines.

Our approach and contributions. We introduce *Transformers with Independent Subnetwork Training (*`TwIST`*)*, a lightweight distributed training algorithm based on the IST family (Yuan et al., 2019; Wolfe et al., 2022; Dun et al., 2023; Hu et al., 2023; Dun et al., 2022). `TwIST` enables efficient distributed training of language models, the setting in which large-scale model training almost exclusively occurs, while inducing zero-shot subnetwork extraction at inference time. It partitions the attention and FFN blocks of the Transformer across workers, producing subnetworks that retain the parent's depth while reducing width, without parameter sharing besides a small set of designated components. By randomly sampling different subnets across compute nodes and periodically aggregating them into the shared parent, `TwIST` trains complementary subnetworks on different data shards, collectively updating the entire parent each round. This differs from ordinary dropout, which randomly masks units without guaranteeing full-model coverage, though dropout can remain complementary and be applied within each worker's subnetwork. To the best of our knowledge, `TwIST` is the first distributed training algorithm designed to enable zero-shot subnetworks in the transformer parent model, providing a route to flexible deployment without importance ranking, learned configuration selection, or additional trainable modulation parameters. Our contributions are:

- We introduce `TwIST`, a distributed training algorithm extending independent subnetwork training (IST) to Transformers with multi-head and grouped-query attention. `TwIST` reduces training time by 33% relative to distributed data parallel (DDP) methods (Appendix A.1), alongside reductions in memory usage and communication. Its structured random extraction produces subnetworks with smaller dense matrices, enabling inference savings with standard hardware kernels and a $1.49\times$ increase in small-batch CPU throughput (Appendix A.2).
- We empirically validate *Zero-Shot Subnetworks (ZSS)* on GPT-2, GPT-2 XL, and Qwen3-0.6B across WikiText-103, TinyStories, and C4. Randomly extracted `TwIST` subnetworks are competitive in several moderate-sparsity configurations, with their strongest advantages emerging under aggressive pruning, without calibration data, search, or fine-tuning at extraction. For example, on WikiText-103, retaining $4/12$ of the targeted attention and FFN blocks in GPT-2 yields a perplexity

of 23.14 versus 31.64 for the closest baseline, while retaining $2/8$ in Qwen3-0.6B yields 22.73 versus 30.13 (Tables 1 and 3).

- We show that TwIST subnetworks exhibit stability across random samples and architectural robustness under training–deployment size mismatch, characterizing the reliability and limits of flexible, on-demand deployment (Figures 4 and 5).

2 BACKGROUND

Traditional distributed training follows either *data parallelism* (Farber & Asanovic, 1997; Raina et al., 2009; Li et al., 2020), where each node holds the full model and processes a local data shard, or *model parallelism* (Dean et al., 2012; Huang et al., 2019), where the model's computation and parameters are partitioned across nodes. For data shards $\{\mathcal{D}_s\}_{s=1}^{S}$, both paradigms optimize the full-model objective

$$\boldsymbol{W}^{\star}_{\text{dense}} = \arg\min_{\boldsymbol{W}} \frac{1}{S} \sum_{s=1}^{S} \mathbb{E}_{(x,y)\sim\mathcal{D}_s} \left[\ell\big(f_{\boldsymbol{W}}(x), y\big) \right],$$

where $f_{\boldsymbol{W}}$ denotes the complete model. The distinction lies in how this objective is evaluated: data-parallel workers evaluate full replicas, whereas model-parallel workers jointly evaluate the model through cross-node activation and gradient exchange. Neither objective explicitly optimizes extracted subnetworks; compression is typically handled separately through post-training pruning, calibration, or fine-tuning (Han et al., 2016; Frantar & Alistarh, 2023; Yang et al., 2025b).

Problem Formulation. We consider a distributed setup over S compute nodes, jointly training a Transformer-based *super-network* $\boldsymbol{W}$ with L layers, each containing G attention blocks and R feedforward blocks. Rather than optimizing only the full model, we optimize deployable subnetworks selected by structured operators Π_s, resampled at each communication round:

$$\boldsymbol{W}^{\star} = \arg\min_{\boldsymbol{W}} \frac{1}{S} \sum_{s=1}^{S} \mathbb{E}_{\substack{(x,y)\sim\mathcal{D}_s \\ \Pi_s\sim p_s(\Pi)}} \left[\ell\big(f_{\Pi_s(\boldsymbol{W})}(x), y\big) \right].$$

Each Π_s selects, at each layer $\ell \in [L]$, subsets $\mathcal{G}_\ell^s \subseteq [G]$ of attention blocks and $\mathcal{R}_\ell^s \subseteq [R]$ of feedforward blocks, while retaining designated shared components, to form a complete subnetwork $\boldsymbol{w}_s = \Pi_s(\boldsymbol{W})$. The distribution $p_s(\Pi)$ may reflect each node's memory, latency, and energy budgets. Unlike model parallelism, each selected subnetwork is independently executable, so its loss can be evaluated locally without cross-node activation exchange. Unlike conventional data parallelism, workers train resampled subnetworks rather than full-model replicas. This incorporates subnetwork structure into the training objective, coupling distributed training with subsequent subnetwork extraction.

3 OVERVIEW OF TwIST

Figure 1 presents the TwIST system architecture, which enables flexible model compression while reducing pretraining costs. TwIST has three main components: ***(i)*** a **subnet generator** that creates model blueprints based on device resources, ***(ii)*** a **dispatcher** that materializes and sends subnets to workers, and ***(iii)*** an **aggregator** that merges the trained subnets back into the central model. In each communication round, subnets are generated, trained in parallel on separate data shards, and aggregated at the server until convergence. At deployment, the generator uses the target resource constraints to construct a subnet of the desired size, which is then dispatched to the client.

Subnet Generator. During training, the *subnet generator* samples subnets uniformly at random while ensuring that every parameter is trained. We model each layer as N_{full} structurally disjoint *blocks*, with each subnet retaining $N_{\text{sub}} \leq N_{\text{full}}$ blocks; the ratio $\kappa = N_{\text{sub}}/N_{\text{full}}$ controls subnet size, with smaller κ corresponding to higher sparsity. The model consists of shared parameters (token embeddings, final projections, and layer norms, denoted $\boldsymbol{W}^{\text{shared}}$) and partitioned Transformer layers. Each layer l contains an attention module and a feedforward module, which we decompose into discrete blocks:

Attention Blocks: To natively support Grouped Query Attention (GQA), we partition the attention parameters into G distinct structural blocks, grouping the query heads with their corresponding shared

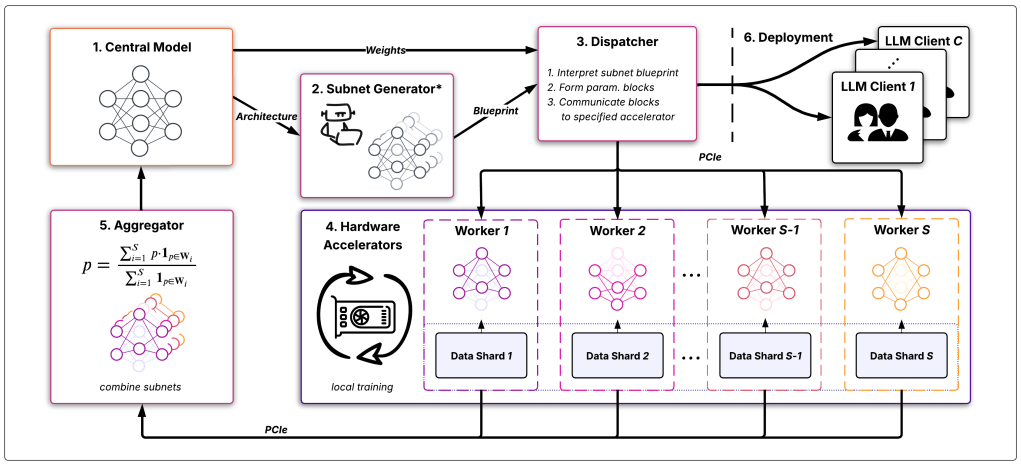

Figure 1: TwIST system overview. **(1)** From a central model, **(2)** a subnet generator* creates diverse subnets. **(3)** A dispatcher sends these subnets via Peripheral Component Interconnect express (PCIe) to **(4)** multiple workers for parallel training on distinct data shards. **(5)** An aggregator updates the central model by averaging the parameters from the trained subnets using the shown formula. **(6)** The final model is then deployed to LLM (Large Language Model) clients for inference. (*The generator supports different heuristics for training vs. deployment.)

key and value heads. The g-th attention block is defined as: $\boldsymbol{W}_g^{\text{AttnBlock}} = (\boldsymbol{W}_g^Q, \boldsymbol{W}_g^K, \boldsymbol{W}_g^V, \boldsymbol{C}_g^{\text{attn}})$ where $\boldsymbol{W}_g^K, \boldsymbol{W}_g^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$ are the single key and value heads for the group, $\boldsymbol{W}_g^Q \in \mathbb{R}^{d_{\text{model}} \times \frac{H}{G} d_{\text{head}}}$ represents the associated subset of query heads, and $\boldsymbol{C}_g^{\text{attn}} \in \mathbb{R}^{\frac{H}{G} d_{\text{head}} \times d_{\text{model}}}$ is the corresponding output projection slice. Here, H is the total number of query heads, G is the number of key/value groups, and d_{head} is the head dimension. For GPT-2 and GPT-2 XL, multi-head attention is the special case $G = H$, so each block contains one query head, one key head, one value head, and the corresponding output-projection slice.

Feedforward Blocks: We partition the FFN hidden dimension d_{inner} into R equal-sized blocks, where R is a user-defined granularity parameter and $q = d_{\text{inner}}/R$. Using column-vector notation, the r-th block in the two-projection GELU FFN used by GPT-2 and GPT-2 XL is $\boldsymbol{W}_r^{\text{FFNBlock}} = \left(\boldsymbol{W}_r^{\text{ffn}}, \boldsymbol{C}_r^{\text{ffn}}\right)$, where $\boldsymbol{W}_r^{\text{ffn}} \in \mathbb{R}^{q \times d_{\text{model}}}$, $\boldsymbol{C}_r^{\text{ffn}} \in \mathbb{R}^{d_{\text{model}} \times q}$. The two slices correspond to the same q hidden neurons. Hidden-layer biases are sliced with these neurons, while the output bias is shared. For the SwiGLU FFN used by Qwen3, each block instead contains the corresponding gate, up, and down projections: $\boldsymbol{W}_r^{\text{FFNBlock}} = \left(\boldsymbol{W}_r^{\text{gate}}, \boldsymbol{W}_r^{\text{up}}, \boldsymbol{W}_r^{\text{down}}\right)$, where $\boldsymbol{W}_r^{\text{gate}}, \boldsymbol{W}_r^{\text{up}}, (\boldsymbol{W}_r^{\text{down}})^\top \in \mathbb{R}^{q \times d_{\text{model}}}$. All projections within a block use the same hidden-neuron indices, allowing the selected blocks to form a complete FFN of reduced hidden width.

By interpreting our weights as a concatenation of these blocks, creating a subnetwork reduces to choosing which blocks from the central model are included in each subnetwork, i.e., finding $\mathcal{A}_s \subseteq [N_{\text{full}}]$ for each s. Since some blocks may be more critical than others for subnet performance (Michel et al., 2019; Zheng et al., 2025), we also allow a set of blocks $\mathcal{C}$ to be shared across all subnetworks. In our experiments this is generally unnecessary, and we use $\mathcal{C} = \emptyset$ except for sharing the first and last few layers to avoid severe performance drops (Kim et al., 2024). Algorithm 1 provides an overview of the subnet generation algorithm used during training based on the random heuristic. Every generated subnetwork must i) contain the set of common blocks, ii) every block is assigned to at least one subnetwork, iii) and all subnetworks must satisfy the size constraint N_{sub}. These constraints can be formally defined as two inequalities $N_{\text{sub}} \leq N_{\text{full}}$ and $N_{\text{full}} - |\mathcal{C}| \leq S(N_{\text{sub}} - |\mathcal{C}|)$. Taken together we get a bound on the size of our subnets during training $\frac{N_{\text{full}} + (S-1)|\mathcal{C}|}{S} \leq N_{\text{sub}} \leq N_{\text{full}}$.

Dispatcher. The *dispatcher* materializes the generator's blueprint on a server-client topology with the full model held on the server. Shared parameters ($\boldsymbol{W}^{\text{shared}}$) are broadcast; partitioned matrices ($\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V, \boldsymbol{C}^{\text{attn}}, \boldsymbol{W}^{\text{ffn}}, \boldsymbol{C}^{\text{ffn}}$) are chunked into blocks, concatenated per the blueprint into submatrices, and scattered. Concatenating before scattering collapses inter-worker communication rounds and drives the latency gains in Figure 6.

Algorithm 1 Generate Training Subnet Blueprint

Require: N_{full}, total blocks per layer; S, number of subnets; N_{sub}, blocks per subnet (with $N_{\text{full}} + (S-1)|\mathcal{C}| \leq S \cdot N_{\text{sub}}$); $\mathcal{C} \subseteq \{1, \ldots, N_{\text{full}}\}$, indices of blocks shared by every subnet.
Ensure: Subnets $\mathcal{A}_1, \ldots, \mathcal{A}_S$ with $\mathcal{C} \subseteq \mathcal{A}_s$, $|\mathcal{A}_s| = N_{\text{sub}}$, and $\bigcup_s \mathcal{A}_s = \{1, \ldots, N_{\text{full}}\}$.

```
# Step 1: seed every subnet with the shared anchor blocks.
for s = 1, ..., S do
    A_s ← C
end for
# Step 2: distribute the remaining blocks round-robin so every block appears in ≥ 1 subnet.
R ← uniform random shuffle of {1, ..., N_full} \ C
for i = 1, ..., |R| do
    s ← ((i − 1) mod S) + 1
    A_s ← A_s ∪ {R_i}
end for
# Step 3: top up any subnet that is still smaller than N_sub.
for s = 1, ..., S do
    while |A_s| < N_sub do
        b ← uniform random element from {1, ..., N_full} \ A_s
        A_s ← A_s ∪ {b}
    end while
end for
return A_1, ..., A_S
```

Aggregator. The *aggregator* inverts the dispatcher: it folds the trained subnets back into the central model. Since the S subnets are mostly disjoint, partitioned parameters are copied back without collision. Shared parameters are averaged across the subnets that hold them using FedAvg (McMahan et al., 2023).

Algorithmic Properties

Asymptotic Memory Ratio. A `TwIST` subnet with $N_{\text{sub}}/N_{\text{full}}$ blocks per layer occupies more memory than the naive ratio $N_{\text{sub}}/N_{\text{full}}$ suggests, because shared parameters $\boldsymbol{W}^{\text{shared}}$ are not partitioned. This penalty shrinks at scale: as attention and FFN parameters dominate over embeddings and layer norms, the gap closes. Figure 2 shows that on GPT-2 variants, a half-block subnet approaches $\approx 50\%$ of the full model's footprint as model size grows. Parameter-counting derivations are in Appendix E.

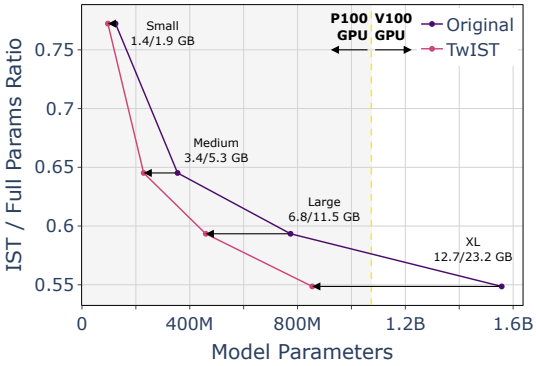

Figure 2: `TwIST`'s asymptotic impact on memory for GPT-2 model variants. Subnets have half the blocks of the full model.

Activation Rescaling. Under the stated initialization assumptions, retaining N_{sub} of N_{full} blocks scales the expected squared output norm of a layer by $N_{\text{sub}}/N_{\text{full}}$. `TwIST` therefore rescales subnet activations by $\sqrt{N_{\text{full}}/N_{\text{sub}}}$ prior to each residual connection to keep subnet activations on the same scale as the parent's, preserving this second moment at initialization. We defer the details and derivation to Appendix D.

Exploration-Exploitation Trade-off. `TwIST` can be viewed as an aggressive form of structured distributed dropout built on the IST family and RaPTr (Panigrahi et al., 2024). Repeated subnet resampling creates an exploration–exploitation trade-off controlled in part by the number of shared anchor blocks, $|\mathcal{C}|$. When $|\mathcal{C}|$ is small, training is distributed more broadly across the partitioned model, improving the *population* of possible subnetworks. As $|\mathcal{C}|$ increases, optimization becomes concentrated on the shared anchor blocks, improving subnetworks that retain this core while reducing coverage of the remaining blocks. We measure these effects using two inference-time metrics: *random-extraction perplexity* (RAND SE), which evaluates uniformly sampled subnetworks, and *anchor-aware perplexity* (AA SE), which retains the $|\mathcal{C}|$ shared blocks and samples the remaining slots. Thus, smaller $|\mathcal{C}|$ favors exploration and lower RAND SE, while larger $|\mathcal{C}|$ favors exploitation and lower AA SE. Figure 3 confirms this behavior for $|\mathcal{C}| \in \{0, 1, 2, 3\}$ on GPT-2 and Qwen3-0.6B; on Qwen3, a 5.7-PPL improvement in AA SE corresponds to a 138-PPL degradation in RAND SE.

Because our deployment setting assumes metadata-free random extraction, we use $|\mathcal{C}| = 0$ throughout Section 4. Full results and discussions are provided in Appendix B.3.

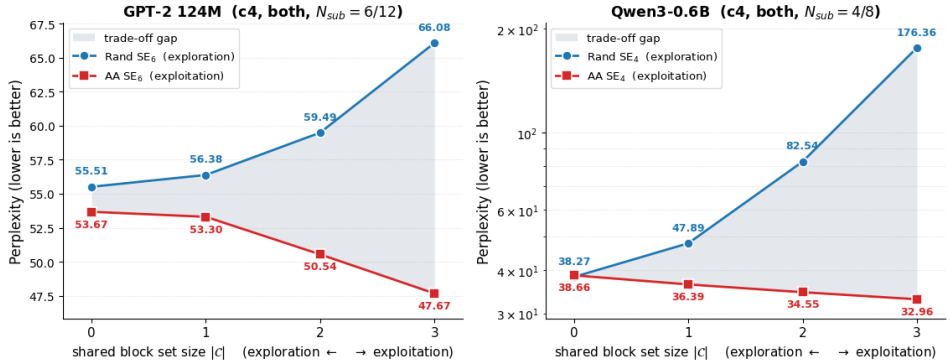

Figure 3: Exploration-exploitation trade-off as $|\mathcal{C}|$ varies on c4 (`both` setting) for GPT-2 124M and Qwen3-0.6B with fixed anchors. Note the log y-axis on Qwen3.

4 Experiments

Models. Our primary experiments use the decoder-only GPT-2 124M (Radford et al., 2019), with 12 Transformer layers, 12 multi-head attention heads, and hidden size 768. We also evaluate Qwen3-0.6B (Yang et al., 2025a), a larger decoder-only Transformer with 28 layers, hidden size 1024, and grouped-query attention (GQA) with 16 query heads sharing 8 key/value groups. For Qwen3, each `TwIST` attention block corresponds to one KV group: one K head, one V head, the associated two query heads, and the matching output-projection slice. To keep partition granularity symmetric across modules, we divide each FFN into 8 blocks of $d_{\text{inner}}/8 = 384$ neurons, so a single $N_{\text{sub}} \in 1, 2, 4$ controls both modules under the `both` setting, with $\kappa = N_{\text{sub}}/8$. Full Qwen3 configuration details are in Appendix C.3. We additionally evaluate GPT-2 XL (1.5B) on WikiText-103 (Appendix B.1).

Datasets. We evaluate on three benchmarks: WikiText-103-raw-v1 (Merity et al., 2016), TinyStories (Eldan & Li, 2023), and a single shard of the C4 corpus (Raffel et al., 2019). For each (model, dataset) pair we both train on the dataset's training split and evaluate on its held-out split. Sequences are packed to 1024 tokens.

Optimization and partitioning. Our primary subnet-quality experiments use Masked `TwIST`, which simulates physical partitioning via activation masking and is equivalent to True `TwIST` at the level of gradient flow; all three variants (Masked, True, Hybrid) are described in Appendix C.1. We finetune each pretrained checkpoint for 3 epochs using Adam with a constant learning rate of 1×10^{-4}: GPT-2 on four Tesla P100 GPUs (batch size 2 per GPU) and Qwen3-0.6B on eight H200 GPUs (batch size 8 per GPU). For `TwIST`, we set $|\mathcal{C}| = 0$, share the first two and last two layers ($N_{\text{sub}} = N_{\text{full}}$), and uniformly partition the remaining layers. We vary N_{sub} and evaluate attention-only (`attn`), feedforward-only (`ffn`), and combined (`both`) settings. Each worker trains its subnet for 15 batches before random repartition; interval repartition ablations are provided in Appendix B.4.

Baseline and Evaluation Metrics. Our experiments evaluate both subnet quality and training efficiency. Appendix A.1 reports training efficiency, including a $1.49\times$ speedup over DDP at $\kappa = 4/12$ (`both`). We apply `TwIST` in a **two-stage pipeline**: first, we train the model using `TwIST` (Figure 1); then, we extract a random subnet from the trained dense model. Implementation details for subnet extraction are provided in Appendix C.2. Appendix A.3 separates measured training costs from unmeasured post-training pruning overhead, while Appendix B.2 compares the dense parent models from `TwIST` and DDP on GPT-2 to quantify the dense-quality trade-off. For our two-stage pipeline baselines, we first train the model using standard data parallelism (Li et al., 2020), then obtain a subnet through either random extraction or one of the following one-shot post-training pruning methods:

- **SparseGPT** zeros a chosen subset of weights, then updates the kept weights by solving a local quadratic (Hessian-weighted) reconstruction problem so the layer's responses on the calibration data match the original responses as closely as possible (Frantar & Alistarh, 2023).

- **Wanda** ranks weights by combining their magnitudes with input activation statistics. As before, pruning decisions are made using a small calibration set (Sun et al., 2024).
- **Z-Pruner** uses Z-score-normalized weight importance and calibration activations to select a pruning mask without retraining (Bhuiyan et al., 2025).
- **Block Prune** partitions attention and FFN projections into blocks and masks those with the lowest empirical Fisher importance. Related head-pruning work uses a different first-order score (Michel et al., 2019).

Table 1: GPT-2 (124M) subnet performance in terms of perplexity (PPL) on WikiText, C4, and TinyStories; bold = best.

Setting	Dataset	κ	Data Parallelism					TwIST (Ours)	
			SparseGPT	Wanda	Z-Pruner	Block Prune	Rand-SE	Rand-SE	Rand-SE$_{6/12}$
attn	WikiText	6/12	**16.81**	19.33	17.75	30.20	32.56	17.74	17.74
		4/12	20.06	86.09	27.42	78.26	61.12	**18.25**	18.88
		3/12	25.76	57.72	44.12	133.41	273.46	**18.49**	19.90
	C4	6/12	**31.74**	33.81	85.42	75.65	4662.16	33.35	33.35
		4/12	44.48	67.36	79.38	79.56	1998.25	**36.14**	40.14
		3/12	64.41	118.62	89.97	78.43	963.49	**37.83**	43.85
	TinyStories	6/12	**3.99**	4.47	11.95	11.65	1811.54	4.00	4.00
		4/12	4.90	7.62	13.89	13.58	65.67	**4.11**	4.26
		3/12	6.11	12.82	16.91	15.02	688.21	**4.21**	4.70
both	WikiText	6/12	**19.11**	36.19	24.85	247.01	176.71	21.32	21.32
		4/12	31.64	173.66	109.11	617.98	974.31	**23.14**	25.53
		3/12	58.16	284.59	238.09	1156.67	2587.42	24.86	**24.32**
	C4	6/12	**36.55**	46.30	507.89	686.93	4495.12	56.03	56.03
		4/12	77.16	176.27	620.77	1103.23	103151.81	**67.49**	85.02
		3/12	141.30	329.62	1482.52	2710.66	5001.93	**80.90**	119.84
	TinyStories	6/12	**4.49**	6.78	77.26	182.26	4800.54	5.16	5.16
		4/12	7.45	31.90	152.60	514.11	1405.25	**5.62**	6.35
		3/12	12.39	85.35	1179.44	2891.97	8709.66	**5.89**	8.20
ffn	WikiText	6/12	**17.99**	24.57	20.65	62.93	63.23	18.90	18.90
		4/12	24.00	63.14	41.57	162.74	174.95	**19.69**	21.39
		3/12	32.63	146.65	88.64	298.84	361.33	**20.51**	23.51
	C4	6/12	**31.67**	35.79	138.71	215.01	177.81	42.63	42.63
		4/12	**41.58**	68.88	359.58	523.15	479.11	47.91	56.10
		3/12	56.42	137.81	690.51	780.20	608.56	**51.36**	74.28
	TinyStories	6/12	**4.02**	4.98	22.51	53.83	47.18	4.38	4.38
		4/12	5.20	11.09	102.04	121.42	97.47	**4.55**	5.01
		3/12	6.85	29.92	263.54	204.54	287.75	**4.68**	6.09

Pruned Subnet Quality. Tables 1 and 3 summarize our results. *Random Subnet Extraction* (Rand-SE) refers to TwIST's deployment-time procedure: given a TwIST-trained parent, we sample a random subnet of the target size using the generator from Section 3. We define the subnet ratio as $\kappa = N_{\text{sub}}/N_{\text{full}}$ (smaller κ = higher sparsity), with $N_{\text{full}} = 12$ for GPT-2 and 8 for Qwen3-0.6B (matching GQA's native 8 KV groups; see Appendix C.3). All pipelines prune the same parameter types (attn, ffn, or both) at identical sparsity. By default the extraction ratio matches the training ratio; for mismatched cases we write Rand-SE$_{N_{\text{sub}}/N_{\text{full}}}$, where the subscript denotes the *training* ratio. For example, Rand-SE$_{6/12}$ is a single GPT-2 parent trained at $\kappa = 6/12$ but evaluated at every row's κ, demonstrating that one parent can deploy subnets at multiple sparsity levels.

TwIST matches or outperforms SOTA one-shot methods at zero post-training cost and without calibration data (Table 2). The advantage emerges sharply under aggressive pruning ($\kappa \leq 4/12$), particularly in the both setting where attention and FFN are simultaneously thinned. On GPT-2/WikiText-103 at $\kappa = 4/12$, TwIST reaches 23.14 PPL versus SparseGPT's 31.64. The gap widens on Qwen3-0.6B, where tight GQA coupling makes random unstructured pruning catastrophic: at $\kappa = 1/8$ (both) on C4, TwIST stays bounded while Z-Pruner and Block Prune diverge by orders of magnitude. At batch size 1 on a Xeon CPU, GPT-2 throughput rises from 3.24 to 4.82 tokens/s at $\kappa = 3/12$ (both) ($1.49\times$; Appendix A.2).

Table 2: Asymptotic cost of producing one pruned subnet from a trained model. P denotes the full-model parameter count, L the number of Transformer layers, d_{model} the model hidden width used to approximate the per-layer second-order matrix cost, N_{calib} the number of calibration tokens, K the number of blocks removed, and $\kappa = P_{\text{sub}}/P$ is the kept-block ratio.

Method	**FLOPs**	**Memory traffic**	**Requires data?**
SparseGPT	$O(L \cdot d_{\text{model}}^3 + N_{\text{calib}} P)$	$O(P)$	Yes (calibration)
Wanda	$O(N_{\text{calib}} P)$	$O(P)$	Yes (calibration)
Block Prune	$O(N_{\text{calib}} P)$	$O(\kappa P)$	Yes (calibration)
Z-Pruner	$O(N_{\text{calib}} P)$	$O(P)$	Yes (calibration)
TwIST (ours)	$O(1)$	$O(\kappa P)$	No

Table 3: Qwen3-0.6B subnet performance in terms of perplexity (PPL) on WikiText, C4, and TinyStories; bold = best. Large baseline values are rounded to integers.

Setting	**Dataset**	κ	**Data Parallelism**					**TwIST** (Ours)	
			SparseGPT	Wanda	Z-Pruner	Block Prune	Rand-SE	Rand-SE	Rand-SE$_{4/8}$
attn	WikiText	4/8	12.00	**11.75**	26.08	29.03	27.89	12.78	12.78
		2/8	**14.63**	15.37	125.93	127.00	123.11	15.54	15.38
		1/8	29.17	34.08	478.23	474.18	2286	24.28	**22.94**
	C4	4/8	26.78	**26.26**	60.28	67.43	61.29	28.92	28.92
		2/8	**33.67**	35.36	301.78	283.55	233.89	37.45	35.92
		1/8	71.63	81.23	721.67	712.70	2035	79.47	**54.34**
	TinyStories	4/8	2.97	**2.96**	5.22	5.19	5.35	3.09	3.09
		2/8	**3.36**	3.64	16.44	15.46	15.77	3.94	3.39
		1/8	5.40	7.79	54.55	51.08	105.94	8.23	**4.04**
both	WikiText	4/8	**13.14**	13.26	75604	81849	324.04	16.04	16.04
		2/8	30.13	37.80	710690	1140000	29321	**22.73**	33.63
		1/8	171.07	188.62	1980000	2140000	2030000	**40.96**	745.98
	C4	4/8	**29.32**	29.47	116665	258380	427.72	38.27	38.27
		2/8	78.63	104.51	1950000	8320000	20893	**53.06**	73.40
		1/8	422.65	421.05	3140000	2630000	2910000	**90.94**	21122
	TinyStories	4/8	**3.07**	3.11	2112	4480	20.96	3.34	3.34
		2/8	4.58	5.75	121886	575383	1865	**4.20**	4.84
		1/8	19.11	32.13	7660000	797190	5420000	**7.61**	22.09
ffn	WikiText	4/8	**12.46**	12.92	34865	34865	45.29	13.89	13.89
		2/8	20.81	27.19	134701	155491	404.34	**17.88**	18.86
		1/8	61.80	125.29	167616	201431	21749	**26.36**	32.20
	C4	4/8	**27.74**	28.67	49014	49014	92.61	32.68	32.68
		2/8	47.90	63.00	229518	222977	612.83	**42.90**	45.33
		1/8	143.87	269.63	460166	353147	27819	**61.95**	75.92
	TinyStories	4/8	**3.02**	3.08	3162	3162	6.12	3.19	3.19
		2/8	4.00	5.13	8297	9009	42.38	4.31	**3.47**
		1/8	10.00	26.56	13908	10963	20456	6.08	**4.48**

System Stability. Figure 4 plots the eval loss distributions for subnets identified by Twist ($SE_{6/12}$) and the DDP baseline. These results, shown for the attn setting, demonstrate that the TwIST distributions are sharply concentrated, indicating low variance across different random subnet generations. In contrast, DDP exhibits significantly higher variance. This variance in DDP-generated subnets becomes more pronounced as the level of sparsity increases.

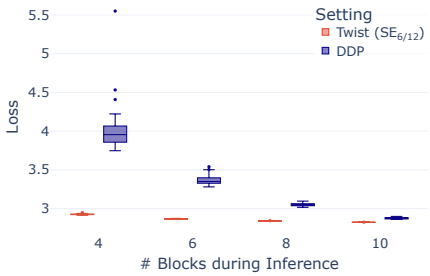

Figure 4: Distribution of eval loss for randomly generated subnets in the attn configuration.

Architectural Robustness. We next ask whether a TwIST-trained model generalizes to subnet sizes it was never trained on. Figure 5 sweeps the train-time block count X against the eval-time count Y (out of 12) for the attn setting; the diagonal $Y = X$ is the matched baseline. The both and ffn settings are in Appendix B.6 and tell the same story.

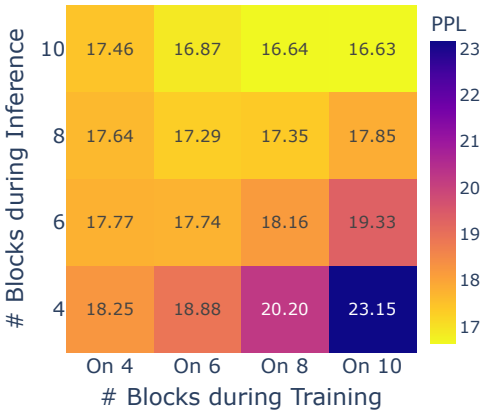

Figure 5: Heatmap of subnet robustness for the `attn` setting. Brighter colors (yellow) signify lower PPL, and darker colors (blue) signify higher PPL.

The observed loss concentration supports consistent random extraction in these evaluated settings; it does not establish that every subnet performs well. The `ffn` and `both` settings show the same trend (Appendix B.5).

Two asymmetries emerge. *Inferencing downward* ($X > Y$), training wide and deploying narrow, degrades sharply: PPL jumps ≈ 42 points (from 18 to 60) at the $X{=}10, Y{=}4$ corner. *Inferencing upward* ($X < Y$) often matches or beats the diagonal. When the deployment target is unknown, err small at training time. Most strikingly, training at a mid-level $X = 6$ yields uniformly low PPL across all Y, a sweet spot that minimizes worst-case degradation in either direction. This suggests a possible route to fault-tolerant deployment, but node-failure recovery is not evaluated here. Size flexibility is also bounded: on Qwen3/C4 (`both`), training at $4/8$ and extracting at $1/8$ gives 21,122 PPL (Table 3).

5 Limitations and Broader Impact

Limitations. Our evaluation reaches GPT-2 XL (1.5B), but not 7B–70B scales, where optimization, communication, and memory behavior may differ. Our systems measurements are also limited to the available training hardware and a single server CPU for inference. Still, the measured reductions in training time, communication, memory, and CPU latency show that `TwIST`'s structured subnetworks provide tangible systems benefits, supporting the practicality of the approach beyond perplexity gains alone. Finally, our quality evaluation is primarily based on perplexity across three language-modeling datasets. These experiments establish robust subnetwork behavior across models and sparsity levels, but do not yet test downstream reasoning, instruction-following, or generation tasks. We leave these explorations to future work.

Broader Impact. If `TwIST` scales to larger foundation models, a single checkpoint could serve a range of compute budgets without maintaining separately pruned or fine-tuned variants, reducing the cost of multi-scale deployment. Its structured subnetworks may also broaden access to capable models on commodity hardware, while carrying the same reduced oversight and misuse risks associated with increasingly accessible open-weight models.

6 Conclusion and Future Work

We introduced `TwIST`, a distributed training algorithm for extracting *Zero-Shot Subnetworks (ZSS)* from the evaluated autoregressive Transformers without post-extraction fine-tuning. By training subnets in parallel and periodically aggregating, `TwIST` folds the search for sparse models into training itself, performing competitively against SOTA post-training methods at moderate sparsity and outperforming them, often by orders of magnitude, under aggressive structured pruning. Because `TwIST` removes whole blocks rather than scattered weights, its subnets translate directly to speedups on inference server hardware lacking efficient sparse compute.

The most exciting direction this opens is a new *deployment paradigm*: a single, openly released `TwIST`-trained foundation model from which any user, on any device, from a datacenter GPU to a phone, can extract a subnet matching their compute budget by selecting a single integer N_{sub}, with no fine-tuning, calibration data, or search. A future goal is for the same checkpoint to serve a phone at $\kappa = 1/8$ and a workstation at $\kappa = 4/8$, collapsing the proliferation of model variants (7B, 13B, 70B and various distilled or quantized derivatives) that currently fragments open-weight ecosystems. Realizing this vision requires scaling `TwIST` to multibillion-parameter regimes, where the interplay between subnet sampling and modern training dynamics, including mixture-of-experts routing, long-context attention, and curriculum schedules, remains an open question. We see `TwIST` as a first step toward treating compressibility not as an afterthought, but as a first-class objective of pretraining.

AI USE STATEMENT

In this work, we used generative AI tools for retrieval and discovery of relevant literature, particularly related work on pruning, dropout, and LLaMaFlex. Generative AI tools were also used to aid and polish the writing of the paper. Specifically, they were used to improve the writing of the introduction and related work, polish the presentation of the problem formulation in the background section, reduce redundancies in the algorithmic properties and experimental sections, and improve the organization and presentation of the experimental results to form a more cohesive narrative. Generative AI tools were additionally used throughout the manuscript to identify and correct grammatical and spelling errors.

All AI-assisted literature retrieval and writing were reviewed by the authors. References identified with AI assistance were manually verified against the original sources, and all AI-assisted edits were checked to ensure that the technical content, claims, numerical results, and intended meaning of the manuscript were preserved. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Arash Ahmadian, Saurabh Dash, Hongyu Chen, Bharat Venkitesh, Stephen Gou, Phil Blunsom, Ahmet Üstün, and Sara Hooker. Intriguing properties of quantization at scale, 2023. URL https://arxiv.org/abs/2305.19268.

Amir Beck and Luba Tetruashvili. On the convergence of block coordinate descent type methods. *SIAM Journal on Optimization*, 23(4):2037–2060, 2013. doi: 10.1137/120887679. URL https://doi.org/10.1137/120887679.

Samiul Basir Bhuiyan, Md. Sazzad Hossain Adib, Mohammed Aman Bhuiyan, Muhammad Rafsan Kabir, Moshiur Farazi, Shafin Rahman, and Nabeel Mohammed. Z-pruner: Post-training pruning of large language models for efficiency without retraining, 2025. URL https://arxiv.org/abs/2508.15828.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Ruisi Cai, Saurav Muralidharan, Hongxu Yin, Zhangyang Wang, Jan Kautz, and Pavlo Molchanov. Llamaflex: Many-in-one llms via generalized pruning and weight sharing. In *International Conference on Learning Representations*, volume 2025, pp. 42250–42263, 2025.

Rich Caruana. Multitask learning. In *Machine Learning*, volume 28, pp. 41–75, 1997. doi: 10.1023/A:1007379606734.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C.J. Burges, L. Bottou, and K. Weinberger (eds.), *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/6aca97005c68f1206823815f66102863-Paper.pdf.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3.int8(): 8-bit matrix multiplication for transformers at scale. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 30318–30332. Curran Associates, Inc., 2022. doi: 10.52202/068431-2198. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/c3ba4962c05c49636d4c6206a97e9c8a-Paper-Conference.pdf.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pp. 4171–4186, 2019.

Chen Dun, Cameron R. Wolfe, Christopher M. Jermaine, and Anastasios Kyrillidis. Resist: Layer-wise decomposition of resnets for distributed training, 2022. URL https://arxiv.org/abs/2107.00961.

Chen Dun, Mirian Hipolito, Chris Jermaine, Dimitrios Dimitriadis, and Anastasios Kyrillidis. Efficient and light-weight federated learning via asynchronous distributed dropout. In *International Conference on Artificial Intelligence and Statistics*, pp. 6630–6660. PMLR, 2023.

Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english?, 2023. URL https://arxiv.org/abs/2305.07759.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. Rigging the lottery: Making all tickets winners, 2021. URL https://arxiv.org/abs/1911.11134.

Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout, 2019. URL https://arxiv.org/abs/1909.11556.

P. Farber and K. Asanovic. Parallel neural network training on multi-spert. In *Proceedings of 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pp. 659–666, 1997. doi: 10.1109/ICAPP.1997.651531.

Jonathan Frankle and Michael Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019. URL https://arxiv.org/abs/1803.03635.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023. URL https://arxiv.org/abs/2301.00774.

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

Advait Gadhikar, Sohom Mukherjee, and Rebekka Burkholz. Why random pruning is all we need to start sparse, 2023. URL https://arxiv.org/abs/2210.02412.

Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks, 2019. URL https://arxiv.org/abs/1902.09574.

Aidan N. Gomez, Ivan Zhang, Siddhartha Rao Kamalakara, Divyam Madaan, Kevin Swersky, Yarin Gal, and Geoffrey E. Hinton. Learning sparse networks using targeted dropout, 2019. URL https://arxiv.org/abs/1905.13678.

Anil K. Gupta, Ken G. Smith, and Christina E. Shalley. The interplay between exploration and exploitation. *Academy of Management Journal*, 49(4):693–706, 2006. doi: 10.5465/amj.2006.22083026. URL https://doi.org/10.5465/amj.2006.22083026.

Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, 2016. URL https://arxiv.org/abs/1510.00149.

B. Hassibi, D.G. Stork, and G.J. Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, pp. 293–299 vol.1, 1993. doi: 10.1109/ICNN.1993.298572.

Erdong Hu, Yuxin Tang, Anastasios Kyrillidis, and Chris Jermaine. Federated learning over images: vertical decompositions and pre-trained backbones are difficult to beat. In *2023 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 19328–19339. IEEE, 2023.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019. URL https://arxiv.org/abs/1811.06965.

HuggingFace. Model memory estimator. https://huggingface.co/docs/accelerate/en/usage_guides/model_size_estimator, 2025.

Michael Bommarito II and Daniel Martin Katz. Gpt takes the bar exam, 2022. URL https://arxiv.org/abs/2212.14402.

Bo-Kyeong Kim, Geonmin Kim, Tae-Ho Kim, Thibault Castells, Shinkook Choi, Junho Shin, and Hyoung-Kyu Song. Shortened llama: Depth pruning for large language models with comparison of retraining methods, 2024. URL https://arxiv.org/abs/2402.02834.

Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. In D. Touretzky (ed.), *Advances in Neural Information Processing Systems*, volume 2. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.

Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. Snip: Single-shot network pruning based on connection sensitivity, 2019. URL https://arxiv.org/abs/1810.02340.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. Pytorch distributed: Experiences on accelerating data parallel training, 2020. URL https://arxiv.org/abs/2006.15704.

Shiwei Liu, Tianlong Chen, Xiaohan Chen, Zahra Atashgahi, Lu Yin, Huanyu Kou, Li Shen, Mykola Pechenizkiy, Zhangyang Wang, and Decebal Constantin Mocanu. Sparse training via boosting pruning plasticity with neuroregeneration, 2022a. URL https://arxiv.org/abs/2106.10404.

Shiwei Liu, Tianlong Chen, Xiaohan Chen, Li Shen, Decebal Constantin Mocanu, Zhangyang Wang, and Mykola Pechenizkiy. The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training, 2022b. URL https://arxiv.org/abs/2202.02643.

H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2023. URL https://arxiv.org/abs/1602.05629.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one?, 2019. URL https://arxiv.org/abs/1905.10650.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe,

Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Abhishek Panigrahi, Nikunj Saunshi, Kaifeng Lyu, Sobhan Miryoosefi, Sashank Reddi, Satyen Kale, and Sanjiv Kumar. Efficient stagewise pretraining via progressive subnetworks, 2024. URL https://arxiv.org/abs/2402.05913.

Sai Prasanna, Anna Rogers, and Anna Rumshisky. When bert plays the lottery, all tickets are winning, 2020. URL https://arxiv.org/abs/2005.00561.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019.

Rajat Raina, Anand Madhavan, and Andrew Y. Ng. Large-scale deep unsupervised learning using graphics processors, 2009. URL https://doi.org/10.1145/1553374.1553486.

Alex Renda, Jonathan Frankle, and Michael Carbin. Comparing rewinding and fine-tuning in neural network pruning, 2020. URL https://arxiv.org/abs/2003.02389.

Victor Sanh, Thomas Wolf, and Alexander M. Rush. Movement pruning: Adaptive sparsity by fine-tuning, 2020. URL https://arxiv.org/abs/2005.07683.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

Jingtong Su, Yihang Chen, Tianle Cai, Tianhao Wu, Ruiqi Gao, Liwei Wang, and Jason D. Lee. Sanity-checking pruning methods: Random tickets can win the jackpot, 2020. URL https://arxiv.org/abs/2009.11094.

Mingjie Sun, Zhuang Liu, Anna Bair, and J. Zico Kolter. A simple and effective pruning approach for large language models, 2024. URL https://arxiv.org/abs/2306.11695.

Hidenori Tanaka, Daniel Kunin, Daniel L. K. Yamins, and Surya Ganguli. Pruning neural networks without any data by iteratively conserving synaptic flow, 2020. URL https://arxiv.org/abs/2006.05467.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Chaoqi Wang, Guodong Zhang, and Roger Grosse. Picking winning tickets before training by preserving gradient flow, 2020. URL https://arxiv.org/abs/2002.07376.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models, 2022. URL https://arxiv.org/abs/2206.07682.

Cameron R. Wolfe, Jingkang Yang, Arindam Chowdhury, Chen Dun, Artun Bayer, Santiago Segarra, and Anastasios Kyrillidis. Gist: Distributed training for large-scale graph convolutional networks, 2022. URL https://arxiv.org/abs/2102.10424.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Yifan Yang, Kai Zhen, Bhavana Ganesh, Aram Galstyan, Goeric Huybrechts, Markus Müller, Jonas M. Kübler, Rupak Vignesh Swaminathan, Athanasios Mouchtaris, Sravan Babu Bodapati, Nathan Susanj, Zheng Zhang, Jack FitzGerald, and Abhishek Kumar. Wanda++: Pruning large language models via regional gradients, 2025b. URL https://arxiv.org/abs/2503.04992.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers, 2022. URL https://arxiv.org/abs/2206.01861.

Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks, 2018. URL https://arxiv.org/abs/1812.08928.

Binhang Yuan, Cameron R Wolfe, Chen Dun, Yuxin Tang, Anastasios Kyrillidis, and Christopher M Jermaine. Distributed learning of deep neural networks using independent subnet training. *arXiv preprint arXiv:1910.02120*, 2019.

Youwei Zheng, Yuxi Ren, Xin Xia, Xuefeng Xiao, and Xiaohua Xie. Dense2moe: Restructuring diffusion transformer to moe for efficient text-to-image generation, 2025. URL https://arxiv.org/abs/2510.09094.

A SYSTEMS EVALUATION

A.1 TRAINING EFFICIENCY

Theoretically, TwIST reduces the number of model parameters, which is expected to decrease memory usage and communication costs (see Appendix E for a detailed derivation). We empirically validate the resulting performance gain by measuring training latency. Figure 6 plots the latency, measured in milliseconds per batch (ms / batch), where lower values indicate faster performance. We compare TwIST to the Distributed Data Parallel (DDP) method, representing full-model fine-tuning and serving as our baseline. DDP exhibits the slowest speed at approximately 385 ms/batch.

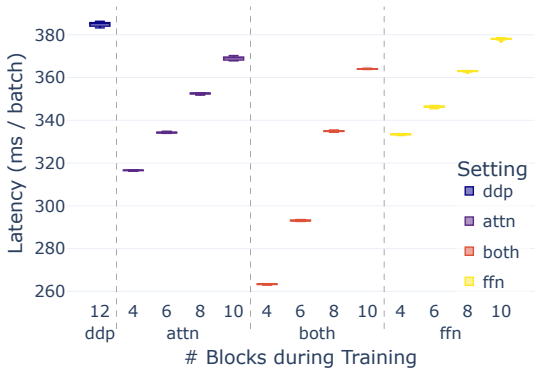

Figure 6: Training latency ablation across pruning settings. Lower is faster.

As expected, we observe throughput increases as the number of blocks in a subnet decreases from ten to four: splitting only attention blocks (attn), splitting only feedforward network blocks (ffn), and splitting (both). Compared to the DDP baseline, four block subnet strategies offer significant efficiency gains. The ffn method (≈334 ms/batch) achieves a 1.15 speedup (a 13.2% time reduction), while the attn method (≈317 ms/batch) reaches a 1.21 speedup (a 17.7% time reduction). Our proposed both approach (≈263 ms/batch) is markedly superior, delivering a 1.46 speedup and cutting training time by 31.7% relative to the full-model baseline, marking a computational advantage.

Table 4: Memory (GB), communication volume (TB), and wall clock time (HH:MM:SS) across pruning settings on four compute nodes.

Setting	κ	Memory (GB)	Comm (TB)	Wall clock
attn	10/12	1.87	44.35	4:19:06
	8/12	1.82	43.20	4:08:30
	6/12	1.79	42.05	3:52:27
	4/12	1.72	40.90	3:41:12
both	10/12	1.77	42.05	4:14:03
	8/12	1.64	38.60	3:49:50
	6/12	1.50	35.14	3:25:14
	4/12	1.34	31.69	3:04:39
ffn	10/12	1.82	43.20	4:30:31
	8/12	1.73	40.90	4:14:19
	6/12	1.63	38.60	4:06:53
	4/12	1.53	36.29	3:54:19
DDP	12/12	1.88	45.50	4:35:42

Table 4 reports memory (GB), communication volume (TB), and wall clock time across TwIST with subnet ratio κ and the DDP baseline on four compute nodes. Memory is the amount of GPU memory allocated on each compute node. Communication is the total inter-node training traffic, defined in Appendix E. Wall clock time is the training duration, including evaluation, for three epochs.

Similar to the trend we observe in throughput, all three metrics improve monotonically as κ decreases. The both setting at $\kappa = 4/12$ yields the largest gains relative to DDP, with memory saving of 0.54 GB

(28.7%); communication decrease of 13.81 TB (30.4%); and wall clock time reduction of about 1.5 hours (33.0%), corresponding to a $1.49\times$ speedup. Interestingly, although `attn` communicates more than `ffn` at the same κ (for $\kappa = 4/12$: 40.90 TB versus 36.29 TB), it still finishes faster with a 13 minutes advantage. This discrepancy likely arises because attention blocks are computationally heavier than feedforward blocks, so removing them reduces a greater portion of the overall computation per subnetwork.

A.2 CPU Inference

To address single-sample deployment dynamics, we benchmarked GPT-2 (124M) at batch size 1 on an inference server CPU (Intel Xeon Silver 4214 CPU @ 2.20GHz, FP32 precision, PyTorch native GEMM execution backend).

Table 5: GPT-2 (124M) single-sample deployment performance at batch size 1 on CPU.

Setting	κ	TTFT (ms)	Throughput (TPS)	Speedup
`baseline`	12/12	211.04	3.24	1.00x
`attn`	6/12	196.31	3.49	1.08x
	3/12	190.57	3.60	1.11x
`ffn`	6/12	173.42	3.87	1.20x
	3/12	160.07	4.20	1.30x
`both`	6/12	160.69	4.21	1.30x
	3/12	141.85	4.82	1.49x

Key Takeaway: Even at batch size 1 on CPU without specialized sparse kernels, intra-layer structured pruning consistently yields higher throughput and reduces Time-To-First-Token (TTFT).

A.3 Training and Deployment Cost Comparison

A.3.1 Theoretical FLOP Comparison

For an idealized compute comparison, assume physical subnet execution, compute proportional to the active block ratio, and negligible shared-parameter and repartitioning costs. This approximation does not describe the Masked implementation. Assume M deployment targets. Let F_{dense} be the cost of standard dense training (DDP), F_{PTP} be the calibration/reconstruction FLOPs per sparsity tier for post-training pruning, and $\bar{\kappa} \in (0, 1]$ be the average active subnetwork ratio during TwIST training:

- **Dedicated Retraining (From Scratch):** Costs $M \cdot (\bar{\kappa} F_{\text{dense}})$, scaling linearly with the number of deployment targets.
- **Train-then-Prune (DDP + PTP):** Costs $F_{\text{dense}} + M \cdot F_{\text{PTP}}$. Under the above approximation, its base training cost exceeds `TwIST`'s when $\bar{\kappa} < 1$.
- **TwIST (Ours):** Costs $\approx \bar{\kappa} F_{\text{dense}}$, remaining constant regardless of M because subnetwork extraction at deployment time requires zero matrix-update FLOPs ($\mathcal{O}(\boldsymbol{\kappa} P)$ memory slicing).

A.3.2 Measured Wall-Clock Training Costs

Beyond theoretical FLOPs, `TwIST` yields immediate hardware efficiency gains during training steps. Because workers process physically smaller subnetwork blocks rather than full dense matrices, `TwIST` reduces per-node GPU memory, decreases inter-node communication volume, and accelerates per-batch execution. We demonstrate this on GPT-2 over 3 epochs (Table 4, Appendix A.1):

Here $T_{\text{PTP}}(M)$ is the unmeasured pruning time for M targets. As shown in Table 4 and Figure 6, `TwIST` ($\boldsymbol{\kappa = 4/12}$, both) achieves a $\mathbf{1.49}\times$ speedup (a 33.0% reduction in wall-clock time) and cuts inter-node communication traffic by 30.4% compared to standard DDP.

Table 6: Measured GPT-2 training costs and symbolic post-training pruning time. Pruning runtime was not measured.

Pipeline / Method	Kept ratio (κ)	GPU Mem / Node	Comm Volume	Wall-Clock Time
Standard DDP	12/12 (Dense)	1.88 GB	45.50 TB	4h 35m 42s
DDP + Post-Training Pruning (SparseGPT)	4/12	1.88 GB	45.50 TB	4h 35m 42s + $T_{\text{PTP}}(M)$
TwIST (Ours)	**4/12 (both)**	**1.34 GB**	**31.69 TB**	**3h 04m 39s**

B Additional Experiments and Ablations

B.1 GPT-2 XL Validation

To address the scaling properties of `TwIST`, we completed an additional scale-up evaluation on GPT-2 XL (1.5B parameters) using WikiText-103-raw-v1.

Experimental setup: We train for 3 epochs using a learning rate of 1×10^{-4}, batch size 2 per rank on 4 GPUs. For each model, we select the best checkpoint among 3 epochs before applying subnetwork extraction or pruning methods.

Here, κ denotes the retained block ratio. Thus, $\kappa \in \{12/25, 8/25, 7/25\}$ corresponds to retaining 48%, 32%, and 28% of targeted structural blocks (52%, 68%, and 72% structural sparsity).

All numerical entries below are testing perplexity (PPL). Bold indicates the best result in each row.

Table 7: GPT-2 XL (1.5B) subnet performance in terms of perplexity (PPL) on WikiText; bold = best.

Setting	κ	Data Parallelism					**TwIST** (Ours)	
		SparseGPT	Wanda	Z-Pruner	Block Prune	Rand-SE	Rand-SE	Rand-SE$_{12/25}$
`attn`	12/25	**12.25**	12.40	16.90	14.10	15.42	12.52	12.52
	8/25	13.73	16.61	23.13	18.78	22.08	**13.35**	13.46
	7/25	14.86	22.08	24.54	20.26	25.69	**13.57**	13.94
`both`	12/25	**12.94**	13.30	176.07	31.61	35.51	15.51	15.51
	8/25	**16.84**	26.91	3096	123.09	314.56	18.01	20.30
	7/25	19.87	52.24	3610	143.84	710.32	**18.78**	24.58
`ffn`	12/25	**12.62**	12.86	19.76	25.91	21.53	13.58	13.58
	8/25	15.11	16.75	38.17	76.71	60.82	**14.45**	15.27
	7/25	16.74	19.74	53.85	108.75	101.97	**14.75**	16.32

B.2 Dense-Parent Quality

While the primary goal of `TwIST` is to enable zero-shot subnetwork extractability rather than optimizing solely for dense pretraining, comparing dense model performance against standard Distributed Data Parallel (DDP) provides valuable context.

To address this, we evaluated validation perplexity on WikiText-103 using GPT-2 (124M). Results are summarized below:

Table 8: Dense-parent validation perplexity on WikiText-103 (GPT-2 124M).

Training Method	Dense Perplexity (↓)
Standard DDP Dense	15.96
TwIST $\mathbf{SE}_{6/12}$ Both	19.06

Key Insights & Context

- **Architectural Robustness ("Inferencing Upward"):** Evaluating the full dense model after `TwIST` training ($\mathbf{SE}_{6/12}$) illustrates our architectural robustness findings. Because training predominantly optimizes smaller subnets (6/12 blocks per targeted layer), evaluating all 12

blocks per layer requires the network to perform inference at a larger capacity than it was strictly optimized for.

- **Competitive Performance Under Disadvantage:** Standard DDP optimizes the dense 124M model exclusively throughout training. Despite this dedicated advantage, the dense parent model from `TwIST` preserves strong capabilities, with a modest $\sim$3 PPL gap.
- **Practical Applications:** `TwIST` is designed for scenarios requiring dynamic model elasticity. If maximum dense performance is the sole objective, `TwIST` can be configured to train on full subnets, naturally reducing to standard DDP.

B.3 $|\mathcal{C}|$ Ablation: Numerical Details

Exploration-Exploitation Trade-off. `TwIST` builds on established methods like IST-family (Yuan et al., 2019; Wolfe et al., 2022; Dun et al., 2023; Hu et al., 2023; Dun et al., 2022) or RaPTr (Panigrahi et al., 2024) and acts as a form of aggressive, structured dropout, providing strong regularization properties. By repeatedly forming new subnets throughout training using a random heuristic, `TwIST` unlocks a tradeoff between the full chaos of random pruning without further finetuning and the stability of a prune-at-initialization (PaI) approach. Conceptually, the dynamic resembles block coordinate descent, as the system iteratively optimizes distinct groups of parameters (i.e., subnets) (Beck & Tetruashvili, 2013), and the entire system dynamics can be understood through the lens of an exploration-exploitation trade-off (Gupta et al., 2006). Concretely, we measure the trade-off through two complementary inference-time metrics on a trained checkpoint: the *random-extraction* perplexity (RAND SE), where a deployer samples a uniformly random subnet of size N_{sub}, and the *anchor-aware* perplexity (AA SE), where a deployer retains the $|\mathcal{C}|$ trained anchor blocks and samples only the remaining $N_{\text{sub}} - |\mathcal{C}|$ slots from the partitioned remainder. RAND SE measures how well-trained the *population* of possible subnets is; AA SE measures how well-trained the *concentrated core* is.

Exploration ($|\mathcal{C}|$ small). Small $|\mathcal{C}|$, $\frac{N_{\text{sub}}}{N_{\text{full}}}$, or repartition interval each promote exploration: the system continuously trains samples from a diverse population of random subnets, and gradient density is spread roughly uniformly across all N_{full} blocks. This broadens the sampled subnet population; whether it improves optimization geometry or functional diversity remains untested. The deployment-side signature of this regime is low RAND SE: a randomly chosen N_{sub}-subnet inherits an evenly trained population.

Exploitation ($|\mathcal{C}|$ large). Conversely, a large $|\mathcal{C}|$, $\frac{N_{\text{sub}}}{N_{\text{full}}}$, or repartition interval each promote exploitation as the pool of available subnets becomes increasingly static, and the objective narrows from exploring a variety of subnets to primarily training a fixed set of shared weights as in a PaI method (Lee et al., 2019; Wang et al., 2020). This strategy yields interconnected benefits. It fosters *greater network alignment*; by forcing all subnets to co-train the same set of shared core parameters, it encourages them to find common, functionally similar solutions. This is particularly effective as $\mathcal{C}$ is intended to capture the components most critical to performance (Michel et al., 2019) and is analogous to hard parameter sharing in multi-task learning (Caruana, 1997). The deployment-side signature is low AA SE: a subnet that retains the anchors reaps their concentrated training.

Empirical validation. Figure 3 sweeps $|\mathcal{C}| \in \{0, 1, 2, 3\}$ on GPT-2 (Radford et al., 2019) and Qwen3-0.6B (Yang et al., 2025a) on C4 (Raffel et al., 2019). We create subnets from dense checkpoints by sampling both the attention-head and FFN blocks, the `both` setting, using $N_{\text{sub}}/N_{\text{full}} = 6/12$ and $4/8$, respectively. The two curves diverge from $|\mathcal{C}| = 0$ as the proposed trade-off would suggest: AA SE drops from $53.67 \rightarrow 47.67$ on GPT-2 and $38.66 \rightarrow 32.96$ on Qwen3, while RAND SE rises from $55.51 \rightarrow 66.08$ and $38.27 \rightarrow 176.36$ respectively. The Qwen3 trade-off is sharply asymmetric: a 5.7-PPL exploitation gain costs 138 PPL of exploration, showing that anchor-aware gains need not improve random-extraction quality.

The setup for success. The optimal $|\mathcal{C}|$ depends on which deployment mode is anticipated. For random subnet sampling at inference, the unrestricted, metadata-free deployment that is the focus of this work, $|\mathcal{C}| = 0$ is the right default: it minimizes RAND SE and avoids the noise tax of an undertrained partitioned remainder. For anchor-aware deployment, increasing $|\mathcal{C}|$ reduces AA SE over the evaluated range. We adopt $|\mathcal{C}| = 0$ for the experiments in Section 4. This choice is supported by the measured random-extraction results.

B.4 Aggregation Interval Ablation

To analyze sensitivity to the repartition interval k (steps per synchronization), we perform an ablation sweep across $k \in \{15, 30, 60, 120\}$ using GPT-2 $\kappa = 6/12$ both:

Table 9: Aggregation-interval sensitivity on GPT-2/WikiText-103 at $\kappa = 6/12$ (both).

Repartition Interval	**PPL**
15 (Default)	21.32
30	21.26
60	21.32
120	21.35

The results show that TwIST remains stable across aggregation intervals ($k = 15$ to $k = 120$). While $k = 30$ achieves the best perplexity (21.26), we use $k = 15$ as a conservative default to ensure low model drift.

B.5 Additional System Stability Experiments

Figure 7 presents the evaluation loss distributions for TwIST ($SE_{6/12}$) compared against the DDP baseline in the both configuration. The analysis varies the number of active blocks during inference (4, 6, 8, 10). TwIST maintains a consistent low loss (3.1–3.2) with negligible variance across all configurations. The DDP baseline exhibits substantially higher loss and high variance, particularly with fewer active blocks: median loss is highest at 4 blocks (≈6.8) and progressively improves toward TwIST's level only at 10 blocks (≈3.2).

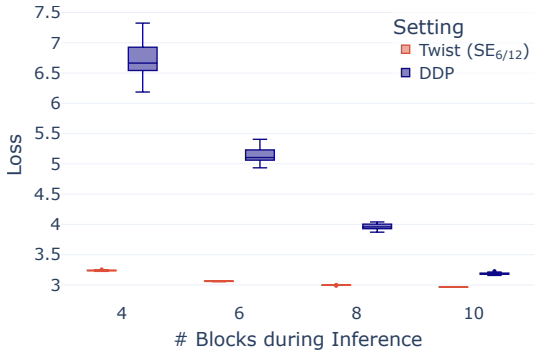

Figure 7: Distribution of eval loss for randomly generated subnets in the both configuration. TwIST ($SE_{6/12}$) versus DDP baseline across various subnet ratios.

The ffn configuration in Figure 8 confirms the same pattern. TwIST ($SE_{6/12}$) achieves a consistently low loss (≈3.0) with near-zero variance regardless of active block count. The DDP baseline, while exhibiting somewhat lower overall losses than in the both setting (median ≈5.1 at 4 blocks vs. ≈6.8 in Figure 7), shows the same dependence on network width: loss and variance peak at the leanest width and only converge toward TwIST at full 10-block inference. Together, Figures 7 and 8 support consistent subnet performance over the evaluated widths.

B.6 Additional Architectural Robustness Experiments

Figures 9 and 10 examine the mismatch between training and inference widths under the both and ffn settings. Both heatmaps confirm the asymmetric pattern observed in the main paper.

In the both setting, models trained on wide subnets degrade catastrophically at narrow inference: the model trained "On 10" blocks reaches 17.97 PPL at 10-block inference but jumps to 60.05 PPL at 4-block inference. The model trained "On 4" blocks remains stable across all evaluated inference widths (PPL 21.76–23.14).

The ffn setting reproduces the trend: the model trained "On 10" blocks degrades from 17.16 PPL (10-block inference) to 36.86 PPL (4-block inference), while the "On 4" model stays in a tight

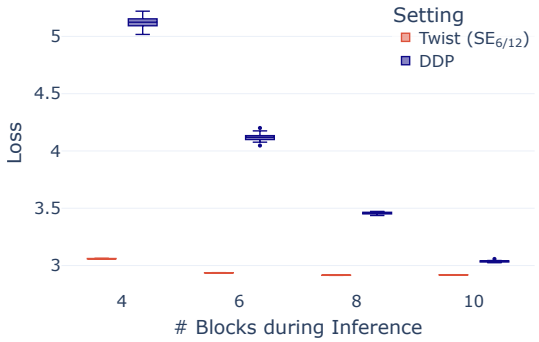

Figure 8: Distribution of eval loss for randomly generated subnets in the `ffn` configuration. `TwIST` ($SE_{6/12}$) versus `DDP` baseline across various subnet ratios.

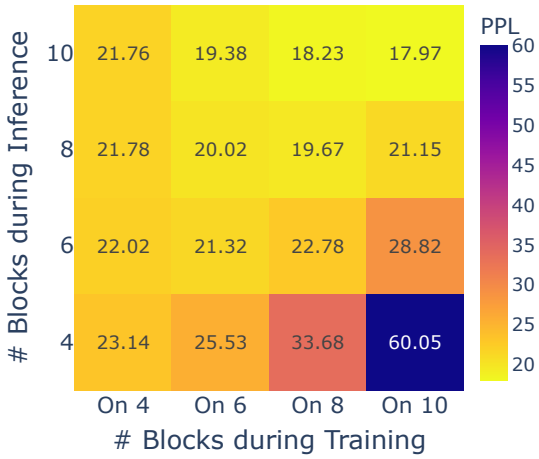

Figure 9: Heatmap of subnet robustness for the `both` setting. Brighter (yellow) is lower PPL.

19.69–22.04 range regardless of inference width. Both heatmaps reinforce the conclusion that target sparsity must be considered from the start of training; traditional dense training collapses on smaller subnets.

C Implementation Details

This section presents `TwIST`'s implementation. We begin with a careful treatment of the three `TwIST` variants that differ in their fidelity to physical partitioning. We then break down the pruning techniques in the `TwIST` framework, and provide the hardware configurations used in our experiments.

C.1 `TwIST` Variants

Our `TwIST` variants include Masked `TwIST`, True `TwIST`, and Hybrid `TwIST`. Each variant reflects a different level of faithfulness to physically partitioning parameters and requires a different degree of modification to the source model. Masked `TwIST` is the simplest to implement, while both True `TwIST` and Hybrid `TwIST` require direct modification of the Hugging Face source code. We describe the three variants in order of increasing implementation complexity.

Masked `TwIST`. Masked `TwIST` is a simulated version of `TwIST`. Instead of physically scattering the parameters across workers, we emulate subnet training by masking out activations that correspond to inactive blocks. For the feedforward layers, we implement this by defining a `MaskingHook`, which zeros out activations specified by a mask matrix. For the attention layers, we make use of the existing `head_mask` argument in Hugging Face's implementation. In both cases, the module output is multiplied element-wise by the mask $\boldsymbol{M}$, producing $\boldsymbol{y} = \boldsymbol{M} \odot f(\boldsymbol{x})$, so that only the active subnet contributes to the output. During evaluation, we disable masking by setting every element of $\boldsymbol{M}$ to 1.

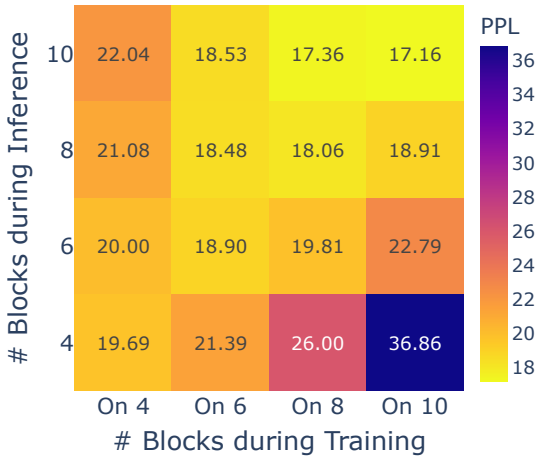

Figure 10: Heatmap of subnet robustness for the `ffn` setting. Brighter (yellow) is lower PPL.

This method does not require any source code modification, making it the fastest and most portable way to prototype TwIST's behavior while maintaining functional equivalence to True TwIST at the level of gradient flow and scaling.

True TwIST. True TwIST introduces real physical partitioning of the model parameters and often requires changes to the transformer source code. In Hugging Face's original GPT-2 implementation, each attention layer defines square projection matrices with $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$. To support physically smaller subnets, we must untie the width of an attention weight matrix from its height. For example, $\boldsymbol{W}^Q$ is the concatenation of weights for several heads and thus the second dimension of $\boldsymbol{W}^Q$ varies with the number of heads while the first remains fixed as it represents the token embedding dimension. Additionally, we define each layer's width independently of other layers to add capability for sharing some layers while splitting others. At runtime, a *central hardware accelerator* maintains the complete model parameters, while $S-1$ workers each hold a smaller physical subset of those parameters. The central accelerator handles parameter partitioning, scattering, and aggregation, whereas each worker locally trains its assigned subnet. This design yields memory and communication savings because each worker stores, updates, and sends only the subset of parameters allocated to its subnet.

Hybrid TwIST. Hybrid TwIST combines the physical subnets of True TwIST with the masking strategy of Masked TwIST. The central accelerator participates in training as a regular worker while maintaining the complete parameter set. To prevent duplication of model copies on the central device, we apply masking to the central model so that it behaves like a smaller subnet during training. The remaining $S-1$ workers train physically smaller subnets, constructed and synchronized in the same way as in True TwIST.

Figure 11 summarizes the three TwIST variants.

C.2 Pruning Implementation

The pruning pipeline in TwIST focuses on the same transformer modules that are partitioned during subnetwork training. To maintain flexibility, pruning is implemented through a modular architecture composed of several interoperable components. We describe the key components below and summarize how they are integrated into full multi-stage pruning workflows in the experimental section.

Deployment. To generate the pruned model for deployment, we mirror the subnet generation procedure used during TwIST training. These randomly sampled subnets allow cheap and efficient deployment of pruned models without requiring any complex search procedures or retraining, forming the foundation of TwIST by allowing the deployment of models of varying sizes from a single checkpoint.

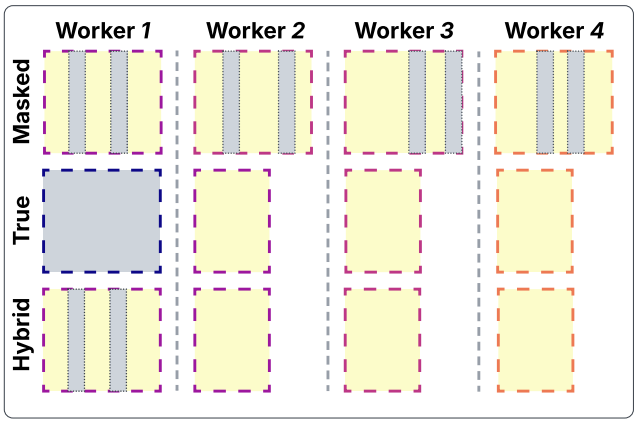

Figure 11: Visualization of the three TwIST variants and their training dynamics in the case of $S = 4$, where Worker 1 is the central accelerator. In Masked TwIST, subnets are simulated by masking activations within a single shared model. In True TwIST, each worker trains a physically smaller subnet that is scattered from and later synchronized with the central model. In Hybrid TwIST, the central model participates in training as a masked subnet while the remaining workers train physical subnets. Within each repartition interval, the yellow regions indicate active parameters being updated during training, while the grey regions denote inactive parameters that are frozen or masked out.

Training Backends. We provide two distinct backends for training models. The DDP module trains the given model under data parallelism, where all workers share identical parameters. The twist module trains the model with the Masked TwIST methodology in Section C.1.

Evaluation Mode. Our evaluation implementation varies depending on whether we want to evaluate the subnetwork or the full model. To evaluate the subnetwork, we mask parameters as needed and apply the appropriate scaling hooks before performing evaluation on each worker's subnet. Each subnet is evaluated independently, and the resulting test losses are averaged across workers. To evaluate the full model, we assess the model directly without additional scaling or masking.

Hardware Setup. We set deterministic seeds to ensure reproducibility across our experiments. For GPT-2 experiments, we used four Tesla P100 GPUs (16 GB each) connected via PCIe. For Qwen3-0.6B experiments, we used eight H200 GPUs (DDP). All training was performed with sequence length 1024, three epochs, and Adam ($\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$, weight decay 0) at a constant learning rate 1×10^{-4}.

C.3 Qwen3-0.6B Setup Details

This section gives the full configuration used to extend Table 3 from GPT-2 to Qwen3-0.6B (Yang et al., 2025a).

Model. Qwen3-0.6B is a decoder-only transformer with grouped-query attention (GQA): 16 query heads share 8 key/value heads (2 queries per KV group). Hidden size is 1024, intermediate (FFN) size 3072, and depth 28 layers, in contrast to GPT-2 small's 12 layers, 12 multi-head attention heads, and hidden 768 (FFN 3072).

Table 10: Architectural comparison between GPT-2 small (the original TwIST target) and Qwen3-0.6B.

	GPT-2 small	**Qwen3-0.6B**
Attention type	MHA	GQA (16 Q / 8 KV)
# layers	12	28
Hidden / FFN size	768 / 3072	1024 / 3072
# attention "blocks"/layer	12 heads	8 KV groups
# FFN blocks/layer	12	8
Edge layers (kept full, shared)	first 2 + last 2	first 2 + last 2
Middle layers (partitioned)	8	24

Symmetric 8-block partition. For Qwen3, one `TwIST` attention block corresponds to one *KV group* (1 K head + 1 V head + the 2 query heads bound to it + the matching output-projection slice). To keep partition granularity symmetric across modules, an FFN block corresponds to a contiguous slice of $d_{\text{inner}}/8 = 384$ neurons in each layer's MLP, matching the 8 KV-group count of the attention partition (this departs from the GPT-2 convention of 12 FFN blocks).

Single ratio for both modules. Because attention and FFN are now both partitioned into 8 native blocks, a single integer N_{sub} controls both modules under the `both` setting. We sweep $N_{\text{sub}} \in \{1, 2, 4\}$ over $N_{\text{full}} = 8$ and report ratios as $\kappa = N_{\text{sub}}/8$. Edge layers always retain all 8 KV groups and 8 FFN blocks. As an example, for the $4/8$ "both" row the per-layer block counts written into the generated configs are

```
n_ist_attn_blocks_by_layer = [8]*2 + [4]*24 + [8]*2
n_ist_ffn_blocks_by_layer = [8]*2 + [4]*24 + [8]*2
```

Training recipe. All Qwen3 `TwIST` and dense-baseline runs use an identical recipe across C4, TinyStories, and WikiText-103: sequence length 1024, per-GPU batch size 8 across 8 H200 GPUs (effective batch 64), 3 epochs, Adam ($\beta_1 = 0.9, \beta_2 = 0.999, \varepsilon = 10^{-8}$, weight decay 0), constant learning rate 1×10^{-4}, gradient checkpointing enabled. `TwIST` uses the masked trainer with `repartition_interval = 15` and uniform random partitioning (no smart-assign). At extraction, $\sqrt{N_{\text{full}}/N_{\text{sub}}}$ activation rescaling is applied per section. Pruning baselines (SparseGPT, Wanda, Z-Pruner, Block Prune) use 128 calibration samples drawn from the dataset's training split, executed on a single GPU.

D Correcting Activation Shift

We analyze how subsampling attention heads or feedforward neurons changes the second moment of a module's output at initialization. For full and subsampled modules evaluated at the same input, let $\boldsymbol{y}$ and $\boldsymbol{y}'$ denote their outputs before the residual connection. Under the assumptions below, retaining a fraction $\kappa = N_{\text{sub}}/N_{\text{full}}$ of equal-sized blocks gives

$$\mathbb{E}[\|\boldsymbol{y}'\|^2] = \kappa\, \mathbb{E}[\|\boldsymbol{y}\|^2].$$

Equivalently, their root-mean-square (RMS) norms satisfy

$$\sqrt{\mathbb{E}[\|\boldsymbol{y}'\|^2]} = \sqrt{\kappa}\, \sqrt{\mathbb{E}[\|\boldsymbol{y}\|^2]}.$$

Thus, multiplying subnet outputs by $1/\sqrt{\kappa} = \sqrt{N_{\text{full}}/N_{\text{sub}}}$ preserves the expected squared output norm at initialization. Theorems D.4 and D.6 establish this result for the idealized FFN and attention modules defined below, using the notation of (Panigrahi et al., 2024).

D.1 Key Components

Layer Normalization. As defined in (Panigrahi et al., 2024), a layer normalization function $f_{\text{ln}} : \mathbb{R}^{d_{\text{model}}} \to \mathbb{R}^{d_{\text{model}}}$ with learnable parameters $\boldsymbol{\gamma}, \boldsymbol{b} \in \mathbb{R}^{d_{\text{model}}}$ operates on an input vector $\boldsymbol{x} \in \mathbb{R}^{d_{\text{model}}}$. The process first normalizes $\boldsymbol{x}$ to $\boldsymbol{z} = (\boldsymbol{x} - \mu)/\sigma$, where μ and σ are the mean and standard deviation of the elements in $\boldsymbol{x}$. The final output $\boldsymbol{y}_{\text{ln}}$ is then computed as $\boldsymbol{y}_{\text{ln}} = \boldsymbol{\gamma} \odot \boldsymbol{z} + \boldsymbol{b}$.

Multi-Head Attention. From (Vaswani et al., 2023), the scaled dot-product attention function for a single head is defined as

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_{\text{head}}}}\right)\boldsymbol{V},$$

where $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times d_{\text{head}}}$. The output of a single head is a matrix of weighted sums of the value vectors, where the weights are determined by the dot-product similarity between the queries and keys. Multi-head attention allows a model to jointly attend to information from various representation subspaces at different positions. A multi-head attention layer, denoted as f_{attn}, processes a sequence of input vectors $\{\boldsymbol{x}_n\}_{n=1}^N$ and outputs a sequence $\{\boldsymbol{y}_n\}_{n=1}^N$. The layer uses H attention heads, where each head h has its own parameter matrices $\{\boldsymbol{W}_h^Q, \boldsymbol{W}_h^K, \boldsymbol{W}_h^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}\}$. The output is the concatenation of the outputs from each head, projected by a final matrix $\boldsymbol{C}^{\text{attn}} \in \mathbb{R}^{H d_{\text{head}} \times d_{\text{model}}}$:

$$\boldsymbol{Y}_{\text{attn}} = \text{Concat}(\text{head}_1, \ldots, \text{head}_H)\boldsymbol{C}^{\text{attn}},$$

where $\text{head}_h = \text{Attention}(\boldsymbol{Q}_h, \boldsymbol{K}_h, \boldsymbol{V}_h)$. The query, key, and value matrices for each head are derived from the input as $\boldsymbol{Q}_h = \boldsymbol{X}\boldsymbol{W}_h^Q$, $\boldsymbol{K}_h = \boldsymbol{X}\boldsymbol{W}_h^K$, and $\boldsymbol{V}_h = \boldsymbol{X}\boldsymbol{W}_h^V$, with $\boldsymbol{X} \in \mathbb{R}^{N \times d_{\text{model}}}$ being the matrix of input vectors.

Feedforward. A feedforward network (FFN) layer, $f_{\text{ffn}} : \mathbb{R}^{d_{\text{model}}} \to \mathbb{R}^{d_{\text{model}}}$, is defined with parameters $\{\boldsymbol{W}^{\text{ffn}} \in \mathbb{R}^{d_{\text{inner}} \times d_{\text{model}}}, \boldsymbol{C}^{\text{ffn}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{inner}}}\}$ and uses the σ_{relu} activation function. For an input vector $\boldsymbol{x} \in \mathbb{R}^{d_{\text{model}}}$, the output $\boldsymbol{y} \in \mathbb{R}^{d_{\text{model}}}$ is given by $\boldsymbol{y}_{\text{ffn}} = \boldsymbol{C}^{\text{ffn}}\sigma_{\text{relu}}(\boldsymbol{W}^{\text{ffn}}\boldsymbol{x})$.

D.2 Transformer Layer Architecture

A pre-layernorm transformer layer integrates the above components in a sequential manner as described in (Panigrahi et al., 2024). Given an input sequence matrix $\boldsymbol{X} \in \mathbb{R}^{N \times d_{\text{model}}}$, the layer produces an output matrix $\boldsymbol{Y} \in \mathbb{R}^{N \times d_{\text{model}}}$ through the following steps:

1. **Attention Layer Normalization**: $\boldsymbol{X}_{\text{attnln}} = f_{\text{ln}}(\boldsymbol{X}; \boldsymbol{\gamma}_{\text{attn}}, \boldsymbol{b}_{\text{attn}})$.
2. **Multi-Head Attention**: $\boldsymbol{Y}_{\text{attn}} = f_{\text{attn}}(\boldsymbol{X}_{\text{attnln}})$.
3. **Residual Connection**: $\boldsymbol{Y}_{\text{attnlayer}} = \boldsymbol{X} + \boldsymbol{Y}_{\text{attn}}$.
4. **FFN Layer Normalization**: $\boldsymbol{X}_{\text{ffnln}} = f_{\text{ln}}(\boldsymbol{Y}_{\text{attnlayer}}; \boldsymbol{\gamma}_{\text{ffn}}, \boldsymbol{b}_{\text{ffn}})$.
5. **FFN Function**: $\boldsymbol{Y}_{\text{ffn}} = f_{\text{ffn}}(\boldsymbol{X}_{\text{ffnln}})$ (applied row-wise).
6. **Final Output**: $\boldsymbol{Y} = \boldsymbol{Y}_{\text{attnlayer}} + \boldsymbol{Y}_{\text{ffn}}$.

D.3 Initialization

For this analysis, all weight entries are mutually independent and initialized as follows:

$$
\begin{aligned}
[\boldsymbol{W}_h^Q]_{ab},\, [\boldsymbol{W}_h^K]_{ab},\, [\boldsymbol{W}_h^V]_{ab} &\sim \mathcal{N}\left(0, \frac{1}{d_{\text{model}}}\right), \\
[\boldsymbol{C}^{\text{attn}}]_{ab} &\sim \mathcal{N}\left(0, \frac{1}{H d_{\text{head}}}\right), \\
[\boldsymbol{W}^{\text{ffn}}]_{ab} &\sim \mathcal{N}\left(0, \frac{2}{d_{\text{model}}}\right), \\
[\boldsymbol{C}^{\text{ffn}}]_{ab} &\sim \mathcal{N}\left(0, \frac{1}{d_{\text{inner}}}\right).
\end{aligned}
$$

Layer-normalization parameters are initialized to $\boldsymbol{\gamma} = \mathbf{1}$ and $\boldsymbol{b} = \mathbf{0}$. As in the component definitions above, projection biases are omitted.

Each module's input is independent of that module's weights and has finite second moments; its components need not be independent. The full and subsampled modules are evaluated at the same input. Subsets are fixed in advance or sampled independently of the inputs and weights. Retained parameters are copied from the parent without reinitialization, so their initialization variances remain those of the full module. Expectations are over initialization, any random input, and any random subset selection.

Lemma D.1 (Linear projection second moment)**.** *Let $\boldsymbol{x} \in \mathbb{R}^{d_{in}}$ satisfy $\mathbb{E}[\|\boldsymbol{x}\|^2] < \infty$. Let $\boldsymbol{W} \in \mathbb{R}^{d_{out} \times d_{in}}$ have independent entries $W_{ji} \sim \mathcal{N}(0, \sigma_W^2)$ and be independent of $\boldsymbol{x}$. Then*

$$\mathbb{E}[\|\boldsymbol{W}\boldsymbol{x}\|^2] = d_{out}\sigma_W^2\, \mathbb{E}[\|\boldsymbol{x}\|^2].$$

Proof. Conditioning on $\boldsymbol{x}$, independence and zero means of the weight entries eliminate all cross-terms:

$$
\begin{aligned}
\mathbb{E}\big[\|\boldsymbol{W}\boldsymbol{x}\|^2 \mid \boldsymbol{x}\big] &= \sum_{j=1}^{d_{\text{out}}} \sum_{i,k=1}^{d_{\text{in}}} x_i x_k\, \mathbb{E}[W_{ji}W_{jk}] \\
&= d_{\text{out}}\sigma_W^2 \|\boldsymbol{x}\|^2.
\end{aligned}
$$

Taking expectation over $\boldsymbol{x}$ proves the claim. □

Lemma D.2 (ReLU second moment)**.** *Under the assumptions of Lemma D.1,*

$$\mathbb{E}\big[\|\sigma_{relu}(\boldsymbol{W}\boldsymbol{x})\|^2\big] = \frac{1}{2} d_{out}\sigma_W^2\, \mathbb{E}[\|\boldsymbol{x}\|^2].$$

Proof. Conditional on $\boldsymbol{x}$, each coordinate $a_j = [\boldsymbol{W}\boldsymbol{x}]_j$ is symmetric about zero. Therefore,

$$\mathbb{E}[\max(0, a_j)^2 \mid \boldsymbol{x}] = \frac{1}{2}\mathbb{E}[a_j^2 \mid \boldsymbol{x}].$$

Summing over j, taking expectations, and applying Lemma D.1 gives the result. □

Theorem D.3 (FFN squared-norm scaling)**.** *Under the initialization assumptions above, let*

$$\boldsymbol{y}_{ffn} = \boldsymbol{C}^{ffn}\sigma_{relu}(\boldsymbol{W}^{ffn}\boldsymbol{x}).$$

For a subset $\mathcal{J} \subseteq \{1, \dots, d_{inner}\}$ of size $1 \le d'_{inner} \le d_{inner}$, define

$$\boldsymbol{W}' = \boldsymbol{W}^{ffn}_{\mathcal{J},:}, \qquad \boldsymbol{C}' = \boldsymbol{C}^{ffn}_{:,\mathcal{J}}, \qquad \boldsymbol{y}'_{ffn} = \boldsymbol{C}'\sigma_{relu}(\boldsymbol{W}'\boldsymbol{x}).$$

Then

$$\mathbb{E}[\|\boldsymbol{y}'_{ffn}\|^2] = \frac{d'_{inner}}{d_{inner}}\mathbb{E}[\|\boldsymbol{y}_{ffn}\|^2].$$

Proof. First fix $\mathcal{J}$. Since the selected weights retain their original initialization variances, Lemmas D.1 and D.2 give

$$\begin{aligned}\mathbb{E}[\|\boldsymbol{y}'_{\text{ffn}}\|^2] &= \frac{d_{\text{model}}}{d_{\text{inner}}}\mathbb{E}\big[\|\sigma_{\text{relu}}(\boldsymbol{W}'\boldsymbol{x})\|^2\big] \\ &= \frac{d_{\text{model}}}{d_{\text{inner}}} \cdot \frac{1}{2} d'_{\text{inner}} \cdot \frac{2}{d_{\text{model}}}\mathbb{E}[\|\boldsymbol{x}\|^2] \\ &= \frac{d'_{\text{inner}}}{d_{\text{inner}}}\mathbb{E}[\|\boldsymbol{x}\|^2].\end{aligned}$$

The first equality uses independence of $\boldsymbol{C}'$ from $\sigma_{\text{relu}}(\boldsymbol{W}'\boldsymbol{x})$. Setting $d'_{\text{inner}} = d_{\text{inner}}$ gives $\mathbb{E}[\|\boldsymbol{y}_{\text{ffn}}\|^2] = \mathbb{E}[\|\boldsymbol{x}\|^2]$, proving the identity. For an independently sampled subset of the same size, averaging over $\mathcal{J}$ preserves the identity. □

Theorem D.4 (FFN RMS Scale Factor)**.** *Under the conditions of Theorem D.3, let $\kappa_{ffn} = d'_{inner}/d_{inner}$. Then*

$$\sqrt{\mathbb{E}[\|\boldsymbol{y}'_{ffn}\|^2]} = \sqrt{\kappa_{ffn}}\,\sqrt{\mathbb{E}[\|\boldsymbol{y}_{ffn}\|^2]}.$$

Consequently, the rescaled output $\widetilde{\boldsymbol{y}}'_{ffn} = \boldsymbol{y}'_{ffn}/\sqrt{\kappa_{ffn}}$ satisfies

$$\mathbb{E}[\|\widetilde{\boldsymbol{y}}'_{ffn}\|^2] = \mathbb{E}[\|\boldsymbol{y}_{ffn}\|^2].$$

Proof. The first identity follows by taking square roots of the nonnegative quantities in Theorem D.3. The second follows by dividing its squared-norm identity by $\kappa_{\text{ffn}} > 0$. □

Theorem D.5 (Attention squared-norm scaling)**.** *Consider multi-head attention under the initialization assumptions above, with H heads of equal dimension d_{head}, the same attention-mask rule for each head, and output*

$$\boldsymbol{Y}_{attn} = \mathrm{Concat}(head_1, \dots, head_H)\boldsymbol{C}^{attn}.$$

Retain a subset $\mathcal{H} \subseteq \{1, \dots, H\}$ of size $1 \le H' \le H$, together with the corresponding rows of $\boldsymbol{C}^{attn}$, to obtain $\boldsymbol{Y}'_{attn}$. For every token position i,

$$\mathbb{E}[\|[\boldsymbol{Y}'_{attn}]_{i,:}\|^2] = \frac{H'}{H}\mathbb{E}[\|[\boldsymbol{Y}_{attn}]_{i,:}\|^2].$$

Proof. First fix $\mathcal{H}$. Let $\boldsymbol{a}_{i,h} = [\text{head}_h]_{i,:}$ and let $\boldsymbol{C}_h$ denote the d_{head} rows of $\boldsymbol{C}^{\text{attn}}$ corresponding to head h. Then

$$[\boldsymbol{Y}_{\text{attn}}]_{i,:} = \sum_{h=1}^{H} \boldsymbol{a}_{i,h}\boldsymbol{C}_h, \qquad [\boldsymbol{Y}'_{\text{attn}}]_{i,:} = \sum_{h\in\mathcal{H}} \boldsymbol{a}_{i,h}\boldsymbol{C}_h.$$

Set $\sigma_C^2 = 1/(Hd_{\text{head}})$. The output projection is independent of all head outputs. Conditioning on these outputs, its independent zero-mean entries eliminate the cross-terms, yielding

$$\mathbb{E}\big[\|[\boldsymbol{Y}_{\text{attn}}]_{i,:}\|^2 \mid \{\boldsymbol{a}_{i,h}\}_{h=1}^{H}\big] = d_{\text{model}}\sigma_C^2 \sum_{h=1}^{H}\|\boldsymbol{a}_{i,h}\|^2,$$
$$\mathbb{E}\big[\|[\boldsymbol{Y}'_{\text{attn}}]_{i,:}\|^2 \mid \{\boldsymbol{a}_{i,h}\}_{h=1}^{H}\big] = d_{\text{model}}\sigma_C^2 \sum_{h\in\mathcal{H}}\|\boldsymbol{a}_{i,h}\|^2.$$

The heads are identically distributed at initialization, so, for a fixed position i, define $s_i = \mathbb{E}[\|\boldsymbol{a}_{i,h}\|^2]$, which does not depend on h. Taking expectations gives

$$\mathbb{E}[\|[\boldsymbol{Y}_{\text{attn}}]_{i,:}\|^2] = d_{\text{model}}\sigma_C^2 H s_i,$$
$$\mathbb{E}[\|[\boldsymbol{Y}'_{\text{attn}}]_{i,:}\|^2] = d_{\text{model}}\sigma_C^2 H' s_i.$$

Comparing these expressions proves the claim. For an independently sampled subset of the same size, averaging over $\mathcal{H}$ preserves the identity. No independence between head outputs or uniform-attention approximation is required. □

Theorem D.6 (Attention RMS Scale Factor)**.** *Under the conditions of Theorem D.5, let $\kappa_{attn} = H'/H$. Then, for each position i,*

$$\sqrt{\mathbb{E}[\|[\boldsymbol{Y}'_{attn}]_{i,:}\|^2]} = \sqrt{\kappa_{attn}}\,\sqrt{\mathbb{E}[\|[\boldsymbol{Y}_{attn}]_{i,:}\|^2]}.$$

Consequently, the rescaled output $\widetilde{\boldsymbol{Y}}'_{attn} = \boldsymbol{Y}'_{attn}/\sqrt{\kappa_{attn}}$ satisfies

$$\mathbb{E}[\|[\widetilde{\boldsymbol{Y}}'_{attn}]_{i,:}\|^2] = \mathbb{E}[\|[\boldsymbol{Y}_{attn}]_{i,:}\|^2].$$

Proof. Take square roots of the nonnegative quantities in Theorem D.5, then divide its squared-norm identity by $\kappa_{\text{attn}} > 0$. □

Implication for block subsampling. For equal-sized FFN blocks, retaining a fraction $\kappa = N_{\text{sub}}/N_{\text{full}}$ of the blocks retains the same fraction of hidden neurons. The same relation holds for attention blocks containing equal numbers of heads. Thus both modules use the rescaling factor $\sqrt{N_{\text{full}}/N_{\text{sub}}}$ before their residual connection. These are exact second-moment statements at initialization for the same module input. They do not assert equality of mean norms or preservation of the full activation distribution.

E Computing Memory and Communication

We present formulas for the parameter count of a GPT-2 style transformer. Let $N_{\text{embd}} = N_{\text{vocab}} \cdot d_{\text{model}}$ be the embedding parameter count, where N_{vocab} is the vocabulary size. The attention layer parameter count is $N_{\text{attn}} = 4d_{\text{attn}}d_{\text{model}} + 3d_{\text{attn}} + d_{\text{model}}$, where d_{attn} is the number of heads in that layer multiplied by the head dimension. The first term covers the $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V, \boldsymbol{C}^{\text{attn}}$ weights; the next two cover biases for $\{\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V\}$ and $\boldsymbol{C}^{\text{attn}}$ respectively. The feedforward layer parameter count is $N_{\text{ffn}} = 2d_{\text{ffn}}d_{\text{model}} + d_{\text{ffn}} + d_{\text{model}}$, and the layer-norm parameter count is $N_{\text{ln}} = 4d_{\text{model}}$. The transformer layer count is $N_{\text{layer}}^l = N_{\text{ln}} + \alpha_l N_{\text{attn}} + \beta_l N_{\text{ffn}}$, where α and β encode attention and FFN sparsity. Total parameters are

$$N_{\text{model}} = N_{\text{embd}} + \sum_l N_{\text{layer}}^l + N_{\text{proj}},$$

where N_{proj} is the final projection: 0 for text generation with tied weights, $d_{\text{model}} \cdot N_{\text{labels}}$ for classification.

The parameter count is roughly proportional to the memory footprint at load time (where dtype determines bytes per param) and at train time (about four times the model size when using Adam) (HuggingFace, 2025). Since every parameter is sent and received per communication round, total communication cost is proportional to communication rounds times parameter count.