

TWIST: RIGGING THE LOTTERY IN TRANSFORMERS WITH INDEPENDENT SUBNETWORK TRAINING

Michael Menezes¹ Barbara Su¹ Xinze Feng¹ Yehya Farhat¹ Hamza Shili¹ Anastasios Kyrillidis¹

ABSTRACT

We introduce **TwIST**, a novel distributed system for efficient Large Language Model (LLM) training. Motivated by our “golden lottery ticket hypothesis,” **TwIST** trains subnetworks in parallel, periodically aggregating and resampling, yielding high-performance subnets (“golden tickets”) that require no fine-tuning. 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). **TwIST**’s advantage emerges under aggressive pruning (e.g., 50%+ sparsity), where it significantly outperforms baselines; for example, achieving 23.14 PPL while the closest baseline follows at 31.64. As a *structured pruning* method, **TwIST** produces smaller, dense matrices, translating to tangible inference speedups and memory savings on commodity hardware deployments (e.g., CPUs) that lack sparse computation support. We provide the complete implementation [here](#).

1 INTRODUCTION

Large Language Models (LLMs) (Achiam et al., 2023; Brown et al., 2020) have reshaped the field of AI with their performance across a wide range of tasks. The Generative Pretrained Transformer (GPT) family has proven to be a powerful architecture, demonstrating generalization on diverse and complex language benchmarks (Bommarito II & Katz, 2022; Chen et al., 2021; Wei et al., 2022). However, training and deploying these models come with massive computational costs. For instance, DeepSeek-V3 has approximately 671 billion parameters. Storing such a model with half-precision floating-point numbers (FP16) would require around 1.34 TB of memory (2 bytes per parameter \times 671B parameters). To put this into perspective, even for holding the parameters alone, this would necessitate about 17-20 NVIDIA A100 GPUs, each equipped with 80 GB of memory; real inference typically would need 25-100 A100s depending on context length and KV precision. Thus, to democratize the use of LLMs, the research community has explored powerful techniques to mitigate these resource bottlenecks, primarily through model compression. Two of the most prominent approaches are quantization and pruning.

Quantization reduces the memory footprint of a model by representing its parameters with lower-precision numerical formats (Dettmers et al., 2022; Frantar et al., 2022; Ahmadian et al., 2023; Yao et al., 2022). This technique is a

cornerstone of model compression, with two main strategies: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). PTQ offers a straightforward way to compress a pre-trained model but could sometimes lead to a non-negligible decline in accuracy. On the other hand, QAT simulates the quantization process during training, often preserving higher accuracy at the cost of increased training complexity and computational overhead. A commonality in these approaches is that the expensive full-precision full-model training rounds could still often be prerequisite.

A complementary approach to quantization is *pruning* (Le-Cun et al., 1989; Hassibi et al., 1993), which removes individual weights from the model. This approach is supported by concepts like the Lottery Ticket Hypothesis (LTH) (Frankle & Carbin, 2018), which claims that dense networks contain sparse “winning tickets” that can match full model accuracy. Despite LTH, the wide adoption of pruning for LLMs remains an open challenge, largely because finding these subnetworks is difficult. Some of the more successful methods require extensive retraining (Liu et al., 2018; Blalock et al., 2020) or costly iterative and fine-tuning procedures (Frankle & Carbin, 2018; Renda et al., 2020). In general, although other sparsity inducing (Evci et al., 2020; Sanh et al., 2020) or pruning-aware (Han et al., 2015b; Liu et al., 2021) training regimes have shown some moderate success, they still require multiple training passes and extensive amounts of memory. The more practical pruning techniques for LLMs are post-training pruning (PTP) methods, which compress an already trained model without any re-training (Bhuiyan et al., 2025a; Frantar & Alistarh, 2023; Sun et al., 2023). While PTP is computationally less de-

¹Department of Computer Science, Rice University, Texas, USA. Correspondence to: Anastasios Kyrillidis <anastasios@rice.edu>.

manding, it often still involves solving complex, and usually expensive subproblems. A common thread among these pruning strategies is that they require full-model training iterations, attempting to find important weights after this step.

It is unclear if the performance of current pruning algorithms represents an upper bound on the quality of sparse models. [Gale et al. \(2019\)](#) found that three different post-training methods all achieve about the same sparsity / accuracy trade-off. Thus, it is an open question whether better performance trade-offs are possible. [Prasanna et al. \(2020\)](#) study the LTH from the perspective of a pretrained BERT model ([Devlin et al., 2019](#)) and empirically confirm the existence of winning tickets. But their results also reveal other interesting insights. The same non-retrained winning lottery tickets of these models are actually not that far behind the retrained tickets. Furthermore, their work hints at the surprising viability of random pruning. While a randomly pruned and then retrained subnet often lags behind a winning ticket, its performance is not negligible. Such random pruning at initialization ([Su et al., 2020; Liu et al., 2022; Gadhikar et al., 2023](#)) is almost always favored for its simple, computationally cheap, and data independent nature.

Inspired by ([Prasanna et al., 2020](#)) and focusing on Transformer-based neural network training, if the main bottleneck is the expensive, iterative search for a specific “winning ticket,” and not the retraining, we ask whether we can redesign the training process itself to eliminate the search. I.e., what if, instead of creating a dense model with a few high-performing subnets, we could train a model where high performance is the default for most subnets?

This leads us to propose an extension of the LTH, which we term the *golden lottery ticket hypothesis*: ***A dense neural network can be trained such that the vast majority of its randomly sampled subnets achieve high performance without any subsequent training or further fine-tuning.*** Such a model would be inherently compressible and efficient, as sparsity could be achieved by simple random selection rather than a costly search algorithm. From here we ask: how can we regularize a model during training such that nearly any randomly sampled subnet is a “golden ticket?”

The practical benefits of such an approach could be significant. Consider a scenario with a diverse ecosystem of end-user devices, from high-end servers to everyday smartphones. With an inherently compressible parent model, we could deploy a spectrum of smaller “tickets” tailored to the computational capabilities of each device. This would enable a consistent user experience across different hardware, with each device running a version of the model that is not just smaller, but also comparably proficient. This paradigm shifts away from a one-size-fits-all deployment strategy to a more flexible and efficient distribution of AI.

Our approach and contributions. We sidestep the expensive search-and-retrain paradigm of pruning and instead introduce *Transformers with Independent Subnetwork Training (TwIST)*, a novel distributed training algorithm that is inspired by [Yuan et al. \(2019\); Wolfe et al. \(2024\); Dun et al. \(2023\); Hu et al. \(2023\); Dun et al. \(2022\)](#). The algorithm is efficient compared to other standard distributed algorithms and designed to imbue the model with an inherent structural robustness. By training independent subnets across different compute nodes, where each subnet spans the same number of layers as the original mode, TwIST encourages the full model to develop a weight structure where multiple pathways are inherently performant. This approach tackles training and deployment efficiency simultaneously, aiming to produce a model that is compressible by default. The key contributions of our work can be summarized as follows:

- We introduce the *golden lottery ticket hypothesis*: that a dense network can be trained so that randomly sampled subnets achieve high performance *without fine-tuning*.
- We empirically validate our hypothesis on text generation, using TwIST. Demonstrating the feasibility of zero-cost pruning at deployment. As shown in Table 1, TwIST is highly competitive with SOTA methods while incurring low post-training overhead (e.g., calibration, Hessian inversion).
- We show that TwIST’s subnets induce both system stability and architectural robustness, making it suitable for fault-tolerant model-parallel deployments (See Figures 4 and 5).
- We highlight that in aggressive structured pruning scenarios TwIST excels (e.g. at a 4/12 ratio, TwIST achieves 23.14 PPL while the closest PTP baseline follows at 31.64) and unlike unstructured methods we see tangible speedups on commodity hardware.

2 BACKGROUND

Notation. Vectors and matrices are represented with bold font (e.g., \mathbf{x}), while scalars by plain font (e.g., x or S). Capital letters distinguish matrices from vectors (e.g., \mathbf{W} vs w). Calligraphic uppercase letters denote sets (e.g., \mathcal{D}); the cardinality of \mathcal{D} is represented as $|\mathcal{D}|$.

Problem formulation. We consider a distributed training setup over S compute nodes, where node s holds local data $\mathcal{D}_s = \{(\mathbf{X}_i, \mathbf{Y}_i)\}_{i=1}^{|\mathcal{D}_s|}$. We assume each local dataset \mathcal{D}_s is drawn independently and identically distributed (i.i.d.) from a global data distribution. Our goal is to train a transformer-based model, whose parameters are collectively denoted

$$\mathbf{W} = \{\mathbf{W}^{\text{embd}}, \mathbf{W}^{\text{proj}}\} \cup \{\mathbf{W}_l^{\text{layer}}\}_{l=1}^L$$

where \mathbf{W}^{embd} is the token embedding, \mathbf{W}^{proj} is the task specific final projection, and each layer

$$\mathbf{W}^{\text{layer}} = \{\mathbf{W}^{\text{ln}}, \mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V, \mathbf{C}^{\text{attn}}, \mathbf{W}^{\text{ffn}}, \mathbf{C}^{\text{ffn}}\}$$

contains the layer norm parameters ($\mathbf{W}^{\text{ln}} \in \mathbb{R}^{d_{\text{model}} \times 2}$); the query, key, value ($\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times H d_{\text{head}}}$), and output ($\mathbf{C}^{\text{attn}} \in \mathbb{R}^{H d_{\text{head}} \times d_{\text{model}}}$) projection matrices for multi-head attention; and the feedforward network weights ($\mathbf{W}^{\text{ffn}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{inner}}}, \mathbf{C}^{\text{ffn}} \in \mathbb{R}^{d_{\text{inner}} \times d_{\text{model}}}$). Moreover, we define d_{model} as the model embedding dimension, d_{head} as the dimension of a single attention head, H as the number of attention heads, and d_{inner} as the feedforward hidden dimension. The goal is to find values for \mathbf{W} that achieve good accuracy on all data $\mathcal{D} = \cup_s \mathcal{D}_s$, by minimizing the following optimization objective:

$$\mathbf{W}^* \in \arg \min_{\mathbf{W}} \left\{ \mathcal{L}(\mathbf{W}) := \frac{1}{S} \sum_{s=1}^S \ell(\mathbf{W}_s, \mathcal{D}_s) \right\},$$

where $\ell(\mathbf{W}_s, \mathcal{D}_s) = \frac{1}{|\mathcal{D}_s|} \sum_{(\mathbf{X}_i, \mathbf{Y}_i) \in \mathcal{D}_s} \ell(\mathbf{W}_s, (\mathbf{X}_i, \mathbf{Y}_i))$. Here, $\ell(\mathbf{W}_s, \mathcal{D}_s)$ denotes the *local* loss function for user s , associated with a local model \mathbf{W}_s , that gets aggregated with the models of other users. \mathbf{W}_s is either a full copy of the global model at the current training round or a selected submodel of the global one.

Traditional distributed training follows either *data parallelism* (Farber & Asanovic, 1997; Raina et al., 2009; Li et al., 2020b), where each node trains the full model on local data, or *model parallelism* (Dean et al., 2012; Huang et al., 2019), where model layers or partitions are split across nodes. While both optimize the same global objective, they suffer from communication bottlenecks: data parallelism requires synchronizing large dense models, and model parallelism involves fine-grained layer-level exchanges that are costly in practice. Furthermore, while achieving a global model $\widehat{\mathbf{W}} \approx \mathbf{W}^*$ is the theoretical goal, a significant practical consideration remains: the deployability of this model. In many real-world scenarios, the compute nodes are resource-constrained edge devices where a large, dense model $\widehat{\mathbf{W}}$ would incur unacceptable latency and energy costs during inference.

The conventional solution here is to treat model compression as a separate, post-training step. This involves taking the fully trained $\widehat{\mathbf{W}}$ and applying pruning techniques to obtain a pruned model $\widehat{\mathbf{W}}'$, where, with a slight abuse of notation $|\widehat{\mathbf{W}}| \gg |\widehat{\mathbf{W}}'|$. Such a process is notoriously expensive as it necessitates an iterative cycle of removing parameters which might require solving complex subproblems and/or extensive fine-tuning to regain the initial accuracy (Han et al., 2015a; Frantar & Alistarh, 2023; Yang et al., 2025). Moreover, it decouples the primary training objective from the ultimate goal of obtaining an efficient final model.

Algorithm 1 Generate Training Subnet Blueprint

Require: N_{full} (# of full model blocks), S (# of subnets to generate), N_{sub} (# of blocks per subnet), \mathcal{C} (set of block indices common to all subnets)

Ensure: $\mathbf{A} \in \mathbb{N}^{S \times N_{\text{sub}}}$

```

1: # Assign common blocks
2:  $\mathbf{A}[:, :|\mathcal{C}|] = \mathcal{C}$ 
3: # Ensure every block is assigned
4: for  $i, b \in \text{enumerate}(\mathcal{C}')$  do
5:    $\mathbf{A}[i \bmod S, |\mathcal{C}| + [i/S]] = b$ 
6: end for
7: # Ensure every subnet has  $N_{\text{sub}}$  blocks
8: for  $s \in \text{range}(S)$  do
9:    $\mathcal{B}_{\text{filled}} \leftarrow \text{set}(\mathbf{A}[s, :])$ 
10:   $N_{\text{empty}} \leftarrow N_{\text{sub}} - |\mathcal{B}_{\text{filled}}|$ 
11:   $\mathbf{A}[s, -N_{\text{empty}} :] \leftarrow \text{unique\_choice}(\mathcal{B}'_{\text{filled}}, N_{\text{empty}})$ 
12: end for
13: return [sorted( $\mathbf{A}_s$ ) for  $\mathbf{A}_s \in \mathbf{A}$ ]

```

3 OVERVIEW OF TwIST

Figure 1 presents an overview of the TwIST system architecture. The design enables flexible and efficient model compression while also reducing cumulative training costs. TwIST is comprised of three main components, which abstractly relates to previous work (Yuan et al., 2019; Wolfe et al., 2024; Dun et al., 2023; Hu et al., 2023; Dun et al., 2022): (i) a **subnet generator** to create model blueprints based on the available edge devices, (ii) a **dispatcher** to materialize them on each edge device, and (iii) an **aggregator** to update the central model on the server. For each communication round, we first generate a set of subnetwork blueprints, create and send a subnetwork to each respective device, and finally send back the updated subnetworks to the server for aggregation. This process is repeated until convergence. For deployment, the subnetwork generator is supplied with the target constraints to construct a blueprint of the desired model size before the dispatcher sends the pruned model to the target client (edge device).

3.1 Subnet Generator

For efficient subnetwork creation during training, the *subnet generator* generates subnets uniformly at random while ensuring that every parameter is included in training. We focus only on the case of transformers that was missed by literature. Since subnets are formed by subsampling attention heads from the attention layers and neuron blocks from the feedforward layers, we henceforth use the term “blocks” to refer agnostically to either attention heads or feedforward layer neuron chunks. Put simply, a block is a chunk of parameters in memory. Formally, we represent the l -th attention layer with

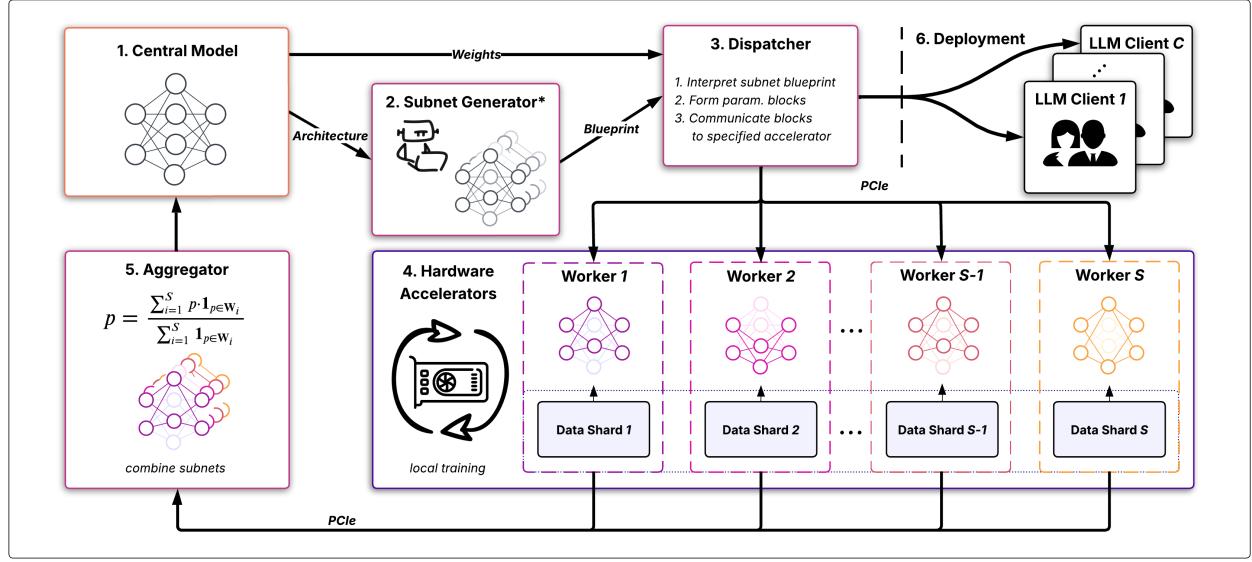


Figure 1. TwIST system overview. (1) From a central model, (2) a subnet generator* creates diverse subnets. (3) A dispatcher sends these subnets via Peripheral Component Interconnect express (PCIe) to (4) multiple workers for parallel training on distinct data shards. (5) An aggregator updates the central 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.)

H attention heads as $\mathbf{W}^{\text{attn}} = \{\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V, \mathbf{C}^{\text{attn}}\}$. For the l -th feedforward layer, we define it as $\mathbf{W}^{\text{ffn}} = \{\mathbf{W}^{\text{ffn}}, \mathbf{C}^{\text{ffn}}\}$. We partition the attention layer into H distinct blocks, where the h -th block of the attention layer is defined as $\mathbf{W}_h^{\text{AttnBlock}} = \{\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V, \mathbf{C}_h^{\text{attn}}\}$ where $\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V, (\mathbf{C}_h^{\text{attn}})^\top \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$. For the feedforward layer, we partition it into R distinct blocks, where R is a parameter that is defined by the user and influenced by the capacity of the edge device. The r -th feedforward block is defined as $\mathbf{W}_r^{\text{FfnBlock}} = \{\mathbf{W}_r^{\text{ffn}}, \mathbf{C}_r^{\text{ffn}}\}$ where $\mathbf{W}_r^{\text{ffn}}, (\mathbf{C}_r^{\text{ffn}})^\top \in \mathbb{R}^{d_{\text{model}} \times \frac{d_{\text{inner}}}{R}}$. All other parameters (e.g. layernorm, token embedding, and final projection) are shared across subnetworks.

By interpreting our weights as a concatenation of these blocks, the problem of creating a subnetwork simplifies to choosing which blocks from the central model should be included in each respective subnetwork. Our work focuses on the case of workers with homogeneous compute (e.g., identical GPUs). Thus, we represent the block assignment blueprint for each layer of the network at every communication round as a matrix $\mathbf{A} \in \mathbb{N}^{S \times N_{\text{sub}}}$ where S represents the number of workers and N_{sub} represents the number of blocks in a subnetwork. One core idea in pruning is that some blocks are more critical than others for overall subnetwork performance (Michel et al., 2019; Zheng et al., 2025). For this reason one can also define a set of blocks \mathcal{C} that is common across all subnetworks. We note that in all our

experiments this is not necessary, as TwIST exhibits competitive results without the need for fixing parameters across subnetworks more than necessary (i.e., $\mathcal{C} = \emptyset$). We make an exception and share the entire 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 subnetwork 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

$$\frac{N_{\text{full}} + (S - 1)|\mathcal{C}|}{S} \leq N_{\text{sub}} \leq N_{\text{full}}.$$

The above constraint only applies to subnetwork during training. When deploying, we relax our constraints and shift our focus to creating a single strong subnetwork.

3.2 Dispatcher

The *dispatcher* is responsible for materializing the subnet generator's blueprint. For this work, we employ a standard server-client network topology where the full central model is hosted on the server. After a user configures the number of blocks, the dispatcher iterates through the transformer's layers and determines the specific weights and biases to

chunk and the dimension along which to do so.

The shared weight matrices (e.g., word embedding and layer norms) are simply broadcasted. The remaining weight matrices (e.g., $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V, \mathbf{C}^{\text{attn}}, \mathbf{W}^{\text{ffn}}, \mathbf{C}^{\text{ffn}}$) are broken up into blocks, and based on the subnet generator’s blueprint are concatenated to form a submatrices before being scattered. This concatenation before scattering reduces the number of inter-worker communication rounds and is key to reaping the benefits in training latency demonstrated in Figure 6.

3.3 Aggregator

The *aggregator* is the counterpart of the dispatcher. It combines the partially trained subnets and updates the central model parameters. The S subnets produced by TwIST are mostly disjoint, meaning that most model parameters are not simultaneously partitioned to multiple subnetworks. Given that most parameters of the subnets are disjoint the aggregator copies the parameters back into the full central model, where no collisions occur. For the shared parameters we borrow the updated procedure of FedAvg (McMahan et al., 2017) and update the shared parameters by the average value across the subnetworks. Formally the updated value of a central model parameter p is given by

$$p = \frac{\sum_{i=1}^S p \cdot \mathbf{1}_{p \in \mathbf{W}_i}}{\sum_{i=1}^S \mathbf{1}_{p \in \mathbf{W}_i}}$$

where \mathbf{W}_i represents the i -th subnet’s trainable parameters.

3.4 Algorithmic Properties

The pursuit of desirable algorithmic properties shapes the design of TwIST. We explore these choices here.

Correcting Activation Shift. We have theoretically shown and empirically verified that subsampling blocks (i.e., attention heads or feedforward neurons) shifts the distribution of activations under common assumptions. In particular, when sampling N_{sub} out of N_{full} blocks from a layer, we observe a general relationship for the layer’s output activation \mathbf{y} :

$$\mathbb{E}[\|\mathbf{y}'\|] = \sqrt{\frac{N_{\text{sub}}}{N_{\text{full}}} \mathbb{E}[\|\mathbf{y}\|]},$$

where \mathbf{y} and \mathbf{y}' are the output activations of the full and subsampled layers, respectively. We scale subnet activations prior to the residual connection in the attention and feedforward layers by $\sqrt{\frac{N_{\text{full}}}{N_{\text{sub}}}}$ to counteract this effect. Derivations of the above relations are given in Appendix Theorem A.6 and Theorem A.10.

Asymptotic Memory Ratio. The TwIST method only partitions a subset of the parameters when forming models. Specifically, when constructing a subnet with N_{sub} blocks

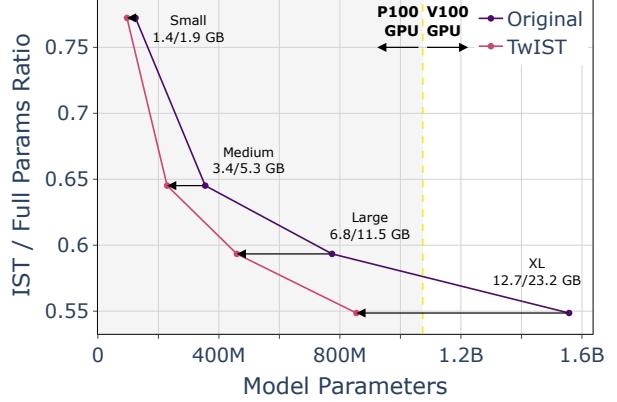


Figure 2. TwIST’s asymptotic impact on memory for GPT-2 model variants. Subnets have half the blocks of the full model.

out of a total of N_{full} blocks per layer, the physical memory consumption on the hardware accelerator exceeds the ideal proportional ratio of $\frac{N_{\text{sub}}}{N_{\text{full}}}$ due to the inclusion of unpartitioned shared parameters (e.g., embeddings and normalization layers). However, an important architectural trend in large-scale transformers is the relative increase in the proportion of attention and feedforward network parameters compared to static parameters, such as token embeddings and layer normalizations. As demonstrated by our analysis on variants of the GPT-2 architecture (Figure 2), this trend leads to an asymptotic memory convergence: for increasingly large full models, subnets comprising half the total blocks ($\frac{N_{\text{full}}}{2}$) approach $\approx 50\%$ of the full model’s total physical memory footprint. This pronounced asymptotic effect corroborates the model size reduction efficacy of memory-efficient techniques targeting sparsity in attention and feedforward layers.

Exploration-Exploitation Trade-off. TwIST builds on established methods like IST-family (Yuan et al., 2019; Wolfe et al., 2024; 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).

Exploration. A small C , $\frac{N_{\text{sub}}}{N_{\text{full}}}$, or repartition interval, each promote exploration as the system continuously trains samples from a diverse population of random subnets. This injection of stochasticity, compared to a static PaI method,

helps prevent the optimization from getting caught in sharp local minima (Evcı et al., 2019; Frankle et al., 2020; Kumar et al., 2024). Moreover, high physical subnet diversity facilitates the creation of more independent, functionally diverse subnets, giving all subnets an equal opportunity to train and effectively creating a population of “lottery tickets” (Evcı et al., 2020). As seen with ensemble learning, functional diversity is a well-established method for improving robustness to perturbations and adversarial attacks, as the ensemble members are less likely to share common failure modes (Pang et al., 2019; Fort et al., 2019).

Exploitation. Conversely, a large 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 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 two primary, interconnected benefits. First, 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 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). Second, this alignment, in turn, *stabilizes central model performance*, as the optimization process converges more consistently by exploiting a known set of functionalities.

The setup for success. This tradeoff reveals TwIST’s hyperparameters can be tuned to prioritize competing objectives. Leaning into exploration (via small C , low $\frac{N_{\text{sub}}}{N_{\text{full}}}$, or frequent repartitioning) promotes functional diversity and robustness. In contrast, leaning into exploitation (via large C , high $\frac{N_{\text{sub}}}{N_{\text{full}}}$, or infrequent repartitioning) fosters network alignment for a more stable and rapidly converging central model. As our work focuses on training for pruning, we choose hyperparameters (See Section 5) that lean into exploration. This choice provides the necessary stochasticity to avoid poor local minima and form a population of robust subnets, while still exploiting the shared structure C to maintain training stability and central model performance.

4 IMPLEMENTATION

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.

4.1 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 M , producing:

$$\mathbf{y} = M \odot f(\mathbf{x}),$$

so that only the active subnet contributes to the output. During evaluation, we disable masking by setting every element of M to 1.

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: $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$. To support physically smaller subnets, we must untie the width of an attention weight matrix from its height. For example, \mathbf{W}^Q is the concatenation of weights for several heads and thus the second dimension of \mathbf{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 sub-

nets 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 3 summarizes the three TwIST variants.

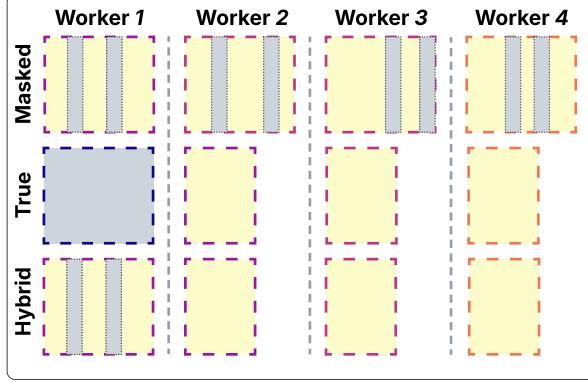


Figure 3. Visualization of the three TwIST variants and their training dynamics in the case of $S = 4$, where Worker 1 is the central accelerator. In Masked TwIST, subnets are simulated by masking activations within a single shared model. In True TwIST, each worker trains a physically smaller subnet 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.

4.2 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.

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 4.1.

Evaluation Mode. Our evaluation implementation varies depending on whether we want to evaluate the subnetwork or the full model. To evaluate the subnetwork, on one hand, 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, on the other hand, we assess the model directly without additional scaling or masking.

Hardware Setup. We set deterministic seeds to ensure reproducibility across our experiments. Our experiments used four Tesla P100 GPUs, each with 16 GB of memory, connected via Peripheral Component Interconnect express (PCIe) for synchronized parallel execution.

5 EXPERIMENTS

We construct our experiments to verify: a) TwIST allows us to identify effective subnets at varying sparsity levels without any post-training pruning (Section 5.1); b) TwIST shows less communication cost and speeds up the training process compared to the standard data-parallel method (Section 5.2).

Setup. Our main experiments use the decoder-only GPT-2 causal language model with 124M parameters (Radford et al., 2019). Within each experiment, we initialize the model with GPT-2’s original configuration (12 transformer layers, 12 attention heads per layer, and a default hidden size of 768) and finetune its pretrained checkpoint for 3 epochs on the next token prediction task. We train the model using the Adam optimizer with a learning rate of 1×10^{-4} .

For TwIST, we set $|C| = 0$ for all layers; and share the first two and last two layers (i.e. $N_{\text{sub}} = N_{\text{full}} = 12$). The remaining 10 layers are partitioned and have the same number of blocks across all subnetworks. For these 10 layers, we vary N_{sub} across experiments. For each configuration, we evaluate three subsampling settings: attention-only (`attn`), feedforward-only (`ffn`), and combined (`both`), where subsampling is applied exclusively to the respective area. We fix the repartition interval at 15 training batches.

For the training dataset, we use the official training split of WikiText-103-raw-v1, a standard benchmark for language modeling (Merity et al., 2016). By default, we use the GPT-2 BPE tokenizer to preprocess the dataset, ensure each training sequence consists of 1024 consecutive tokens, and set the batch size to 2.

Baseline Pipelines and Evaluation Metrics. We apply TwIST in a **two-stage pipeline** where we first train the

Setting	Ratio (κ)	Data Parallelism					TwIST (Ours)	
		SparseGPT	Wanda	Z-Pruner	Block Prune	SE	SE	SE _{6/12}
attn	8/12	16.14	16.42	16.27	21.90	22.91	17.35	17.29
	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
both	8/12	16.60	18.30	17.27	66.44	54.19	19.67	20.02
	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
ffn	8/12	16.41	17.55	16.85	33.83	31.60	18.06	18.48
	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

Table 1. Performance of subnets in terms of Perplexity (PPL).

model by TwIST and then extract a random subnet from the model using the module introduced in Section 4.2. For our two-stage pipeline baselines, we first train the model by the standard data-parallel method (Li et al., 2020a) and then obtain a subnet either by random subnet extraction or by the following post-training pruning methods:

- **SparseGPT** is a one-shot pruning approach that 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** is a one-shot pruning approach that ranks weights by combining their magnitudes with input activation statistics. Pruning decisions are made using a small calibration set as well(Sun et al., 2024).
- **Z-Pruner** is a zero-shot structured pruning technique that does not rely on calibration data. It estimates weight importance using analytic, data-free proxies (e.g., norm-based sensitivity) and removes less important channels or heads in a single step (Bhuiyan et al., 2025b).
- **Block Prune** is a one-shot pruning method that prunes on weight matrices corresponding to attention and feed-forward projections by partitioning each matrix into parameter blocks, and estimating the empirical Fisher of each block using a full pass over the dataset. In which the block with the lowest Fisher information are masked out (Michel et al., 2019).

5.1 Subnet Quality

We evaluate subnet quality using evaluation perplexity on the WikiText-103 test split and report the results in Table 1. For subnet extraction (SE), each attention and feedforward

layer is divided into 12 parameter blocks, following the same block partitioning procedure described in Section 3.1. To ensure a fair comparison, all pipelines prune the same parameter types (attn, ffn, or both) to identical sparsity levels. We define the subnet ratio, κ , by the ratio $X/12$, where X indicates the number of the remaining parameter blocks after pruning or SE . A smaller ratio corresponds to higher sparsity (i.e., fewer active parameters in the network). When the training and extraction sparsity differ, we denote the non-default setting as $SE_{X/12}$, where $X/12$ represents the sparsity ratio used during training. For example, $SE_{6/12}$ indicates that the TwIST model was trained with a sparsity level of 6/12, while retaining the capability to deploy subnets at different sparsity levels.

TwIST achieves perplexity scores remarkably close to State-Of-The-Art (SOTA) one-shot methods like SparseGPT (Frantar & Alistarh, 2023), but at effectively *zero post-training cost*. The SOTA PTP baselines incur substantial overhead, requiring calibration data and complex, layer-wise reconstructions involving costly inverse Hessian (H^{-1}) computations or approximations (e.g., $O(d_{\text{model}}^3)$) (Frantar & Alistarh, 2023). In sharp contrast, TwIST bypasses this entirely, using a simple random sampling strategy with negligible overhead, yet achieves highly competitive results.

Furthermore, TwIST’s advantage emerges strongly under aggressive pruning ($\kappa \leq 6/12$), where it consistently and significantly outperforms all baselines. For instance, at $\kappa = 4/12$ (both), TwIST (23.14 PPL) drastically outperforms SparseGPT (31.64 PPL), while other methods collapse (PPL > 100). Critically, unlike the PTP baselines, TwIST is a **structured pruning** method that removes entire blocks. This creates genuinely smaller, dense matrices, enabling tangible inference speedups and memory savings on commodity hardware. Thus, TwIST delivers high-sparsity subnets that are not only accurate but also genuinely faster in real-world deployments.

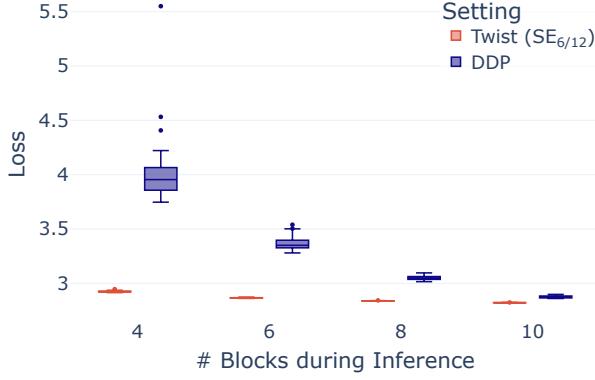


Figure 4. Distribution of eval loss for randomly generated subnets in the `attn` configuration. The distributions for TwIST ($SE_{6/12}$) are compared against a DDP baseline across various subnet ratios. The $SE_{6/12}$ variant of TwIST is presented for a direct comparison, as both this method and DDP involve only a single training pass.

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.

These findings suggest that the performance of DDP subnets is sensitive to the specific subnet initialization, a characteristic consistent with the LTH. Conversely, the low variance of TwIST suggests that its method identifies subnets that are more structurally consistent and less dependent on the random generation process. This stability is crucial: it indicates that the performance of any single sampled subnet is a strong proxy for the performance of other subnets, making the results from TwIST highly reliable and reproducible. We note that similar distributional trends were observed in the `ffn` and both settings (See Appendix B.1).

Architectural Robustness. Next, we investigate architectural robustness, the model’s ability to generalize to subnet configurations it was not trained for. Figure 5 presents a heatmap of subnet performance across various mismatched training and evaluation sparsity targets for the `attn` setting. See Appendix B.2 for the both and `ffn` settings.

In Figure 5, the horizontal coordinate (X) represents the number of parameter blocks (out of 12) used per layer during the training phase of TwIST. The vertical coordinate (Y) represents the number of blocks used for evaluation. A mismatch occurs when $Y \neq X$, with the diagonal representing the matched baseline (where $Y = X$).

The heatmaps show similar trends across pruned parameter types. As expected, performance is generally optimal along

the diagonal ($Y = X$). Yet, two key asymmetries emerge. First, “inferencing downward” (training on a large subnet and evaluating on a smaller one, $X > Y$) causes a sharp performance degradation. This is most severe when the mismatch is large (e.g., $X = 10$, $Y = 4$), where PPL can increase by ≈ 42 points (from 18 to 60). Second, “inferencing upward” (training on a small subnet and evaluating on a larger one, $X < Y$) often maintains or even improves performance compared to the matched baseline. In other words, we find that it is more difficult to prune (go smaller) than to expand (go bigger) at inference time. In practice, this implies one should be more conservative when selecting a training size for an unknown target device.

Most interestingly, training at a mid-level sparsity (e.g., $X = 6$) yields the most uniformly low perplexity across all evaluation sizes (Y). This identifies a “sweet spot” for training: if the target deployment sparsity is unknown, training TwIST at a mid-level sparsity provides a highly robust parent model that minimizes worst-case degradation, transferring well to both leaner and denser subnet configurations.

The practical benefits of this robustness are significant: a single TwIST-trained parent model can deploy a spectrum of “tickets” tailored to the computational budget of each device, enabling a flexible and efficient distribution paradigm that moves away from a one-size-fits-all deployment. When combined with the system stability in Figure 4, TwIST makes models resilient to hardware failures in distributed, model-parallel deployments (Shoeybi et al., 2019; Lepikhin et al., 2020). In such large-scale settings, the failure of a node (which holds certain parameters) is analogous to “inferencing downward” to a smaller subnet. Our results suggest models trained with TwIST can gracefully handle such failures, a critical feature for large-scale training and inference (Wright et al., 2024; Nebius, 2025).

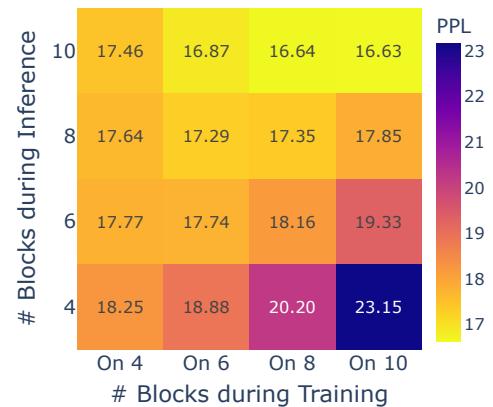


Figure 5. Heatmap of subnet robustness for the `attn` setting. Brighter colors (yellow) signify lower PPL (better performance), and darker colors (blue) signify higher PPL.

5.2 Training Efficiency

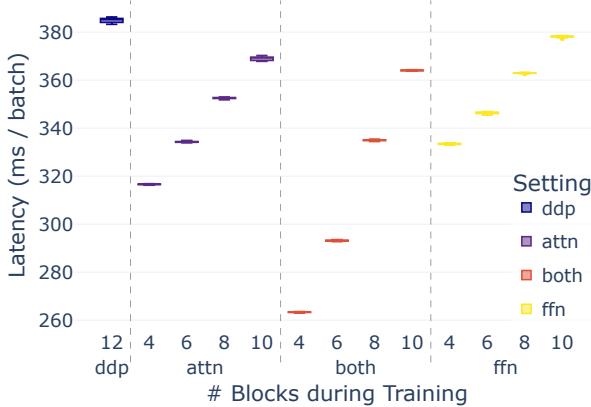


Figure 6. Training latency ablation.

Setting	κ	memory	comm	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	1.5	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 2. Tabulation of memory (GB), communication (TB), and wall clock (HH:MM) across pruning settings.

Theoretically, TwIST reduces the number of model parameters, which is expected to decrease memory usage and communication costs (see Appendix C 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.

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 2 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 C. 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 save 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 computational per subnetwork.

6 CONCLUSION AND FUTURE WORK

We introduced TwIST, a distributed system where subnets are trained in parallel and periodically aggregated. This method is motivated by the *golden lottery ticket hypothesis* and validates that randomly sampled subnets from a TwIST-trained network can achieve high performance *without* finetuning. As shown in Table 1, this enables robust, zero-cost pruning at deployment, achieving perplexity scores competitive with SOTA methods. TwIST effectively shifts the search for sparse models from an expensive post-training step (which often requires calibration data and complex operations like inverting the Hessian matrix) to the training process itself.

TwIST’s true advantage emerges under aggressive pruning ($\kappa \leq 6/12$), where it consistently outperforms all baselines (e.g., 23.14 PPL vs. 31.64 for SparseGPT at $\kappa = 4/12$). Critically, as a *structured pruning* method, TwIST removes entire parameter blocks. This design produces genuinely smaller, dense matrices that translate directly to tangible inference speedups and memory savings on commodity hardware (e.g., CPUs) that lack efficient sparse computation support, offering a practical path to high-sparsity models.

This study was limited by budget, precluding validation on multibillion-parameter models. Future work should explore more sophisticated subnet assignment strategies, optimal aggregation scheduling, and the use of dynamic architectures.

REFERENCES

- Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- Ahmadian, A., Dash, S., Chen, H., Venkitesh, B., Gou, Z. S., Blunsom, P., Üstün, A., and Hooker, S. Intriguing properties of quantization at scale. *Advances in Neural Information Processing Systems*, 36:34278–34294, 2023.
- Beck, A. and Tetruashvili, L. On the convergence of block coordinate descent type methods. *SIAM journal on Optimization*, 23(4):2037–2060, 2013.
- Bhuiyan, S. B., Adib, M. S. H., Bhuiyan, M. A., Kabir, M. R., Farazi, M., Rahman, S., and Mohammed, N. Z-pruner: Post-training pruning of large language models for efficiency without retraining. *arXiv preprint arXiv:2508.15828*, 2025a.
- Bhuiyan, S. B., Adib, M. S. H., Bhuiyan, M. A., Kabir, M. R., Farazi, M., Rahman, S., and Mohammed, N. Z-pruner: Post-training pruning of large language models for efficiency without retraining, 2025b. URL <https://arxiv.org/abs/2508.15828>.
- Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.
- Bommarito II, M. and Katz, D. M. Gpt takes the bar exam. *arXiv preprint arXiv:2212.14402*, 2022.
- Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- Caruana, R. Multitask learning. In *Machine Learning*, volume 28, pp. 41–75, 1997. doi: 10.1023/A:1007379606734.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Chi, T.-C., Fan, T.-H., Chen, L.-W., Rudnicky, A. I., and Ramadge, P. J. Latent positional information is in the self-attention variance of transformer language models without positional embeddings. *arXiv preprint arXiv:2305.13571*, 2023. Accepted by ACL 2023.
- Dean, J., Corrado, G., Monga, R., Chen, K., Devin, M., Mao, M., Ranzato, M., Senior, A., Tucker, P., Yang, K., et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.
- Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems*, 35:30318–30332, 2022.
- Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. 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.
- Dun, C., Wolfe, C. R., Jermaine, C. M., and Kyriolidis, A. ResIST: Layer-wise decomposition of resnets for distributed training. In *Uncertainty in Artificial Intelligence*, pp. 610–620. PMLR, 2022.
- Dun, C., Hipolito, M., Jermaine, C., Dimitriadis, D., and Kyriolidis, A. Efficient and light-weight federated learning via asynchronous distributed dropout. In *International Conference on Artificial Intelligence and Statistics*, pp. 6630–6660. PMLR, 2023.
- Evci, U., Pedregosa, F., Gomez, A., and Elsen, E. The difficulty of training sparse neural networks. *arXiv preprint arXiv:1906.10732*, 2019.
- Evci, U., Gale, T., Menick, J., Castro, P. S., and Elsen, E. Rigging the lottery: Making all tickets winners. In *International conference on machine learning*, pp. 2943–2952. PMLR, 2020.
- Farber, P. and Asanovic, K. Parallel neural network training on multi-spert. In *Proceedings of 3rd International Conference on Algorithms and Architectures for Parallel Processing*, pp. 659–666. IEEE, 1997.
- Fort, S., Hu, H., and Lakshminarayanan, B. Deep ensembles: A loss landscape perspective. *arXiv preprint arXiv:1912.02757*, 2019.
- Frankle, J. and Carbin, M. The lottery ticket hypothesis: Finding sparse, trainable neural networks. *arXiv preprint arXiv:1803.03635*, 2018.
- Frankle, J., Dziugaite, G. K., Roy, D. M., and Carbin, M. Pruning neural networks at initialization: Why are we missing the mark? *arXiv preprint arXiv:2009.08576*, 2020.
- Frantar, E. and Alistarh, D. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International conference on machine learning*, pp. 10323–10337. PMLR, 2023.
- Frantar, E., Ashkboos, S., Hoefer, T., and Alistarh, D. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

- Gadhikar, A. H., Mukherjee, S., and Burkholz, R. Why random pruning is all we need to start sparse. In *International Conference on Machine Learning*, pp. 10542–10570. PMLR, 2023.
- Gale, T., Elsen, E., and Hooker, S. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.
- Gupta, A. K., Smith, K. G., and Shalley, C. E. 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>.
- Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.
- Han, S., Pool, J., Tran, J., and Dally, W. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015b.
- Hassibi, B., Stork, D. G., and Wolff, G. J. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pp. 293–299. IEEE, 1993.
- He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.
- Hu, E., Tang, Y., Kyrillidis, A., and Jermaine, C. Federated learning over images: vertical decompositions and pre-trained backbones are difficult to beat. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 19385–19396, 2023.
- Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.
- Hugging Face Documentation Team. Model memory estimator. https://huggingface.co/docs/accelerate/en/usage_guides/model_size_estimator, 2025. Accessed: 21 October 2025.
- Kedia, A., Zaidi, M. A., Khyalia, S., Jung, J., Goka, H., and Lee, H. Transformers get stable: An end-to-end signal propagation theory for language models. *arXiv preprint arXiv:2403.09635*, 2024. Accepted at ICML 2024.
- Kim, B.-K., Kim, G., Kim, T.-H., Castells, T., Choi, S., Shin, J., and Song, H.-K. Shortened llama: Depth pruning for large language models with comparison of retraining methods. *arXiv preprint arXiv:2402.02834*, 2024. URL <https://arxiv.org/abs/2402.02834>.
- Kumar, T., Luo, K., and Sellke, M. No free prune: Information-theoretic barriers to pruning at initialization. *arXiv preprint arXiv:2402.01089*, 2024.
- LeCun, Y., Denker, J., and Solla, S. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.
- Lee, N., Ajanthan, T., and Torr, P. H. S. SNIP: Single-shot network pruning based on connection sensitivity. In *International Conference on Learning Representations (ICLR)*, 2019. URL <https://arxiv.org/abs/1810.02340>. Published on arXiv as 1810.02340.
- Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. GShard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.
- Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., and Chintala, S. Pytorch distributed: Experiences on accelerating data parallel training, 2020a. URL <https://arxiv.org/abs/2006.15704>.
- Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020b.
- Liu, S., Chen, T., Chen, X., Atashgahi, Z., Yin, L., Kou, H., Shen, L., Pechenizkiy, M., Wang, Z., and Mocanu, D. C. Sparse training via boosting pruning plasticity with neuroregeneration. *Advances in Neural Information Processing Systems*, 34:9908–9922, 2021.
- Liu, S., Chen, T., Chen, X., Shen, L., Mocanu, D. C., Wang, Z., and Pechenizkiy, M. The unreasonable effectiveness of random pruning: Return of the most naive baseline for sparse training. *arXiv preprint arXiv:2202.02643*, 2022.
- Liu, Z., Sun, M., Zhou, T., Huang, G., and Darrell, T. Rethinking the value of network pruning. *arXiv preprint arXiv:1810.05270*, 2018.
- McMahan, H. B., Moore, E., Ramage, D., Hampson, S., and Agüera y Arcas, B. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54, pp. 1273–1282, 2017. URL <https://arxiv.org/abs/1602.05629>.

- Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models, 2016.
- Michel, P., Levy, O., and Neubig, G. Are sixteen heads really better than one?, 2019. URL <https://arxiv.org/abs/1905.10650>.
- Nebius. Fault-tolerant training: How we build reliable clusters for distributed AI workloads. Nebius Blog, August 2025. URL <https://nebius.com/blog/posts/how-we-build-reliable-clusters>.
- Pang, T., Xu, K., Du, C., Chen, N., and Zhu, J. Improving adversarial robustness via promoting ensemble diversity. In *International Conference on Machine Learning*, pp. 4970–4979. PMLR, 2019.
- Panigrahi, A., Saunshi, N., Lyu, K., Miryoosefi, S., Reddi, S., Kale, S., and Kumar, S. Efficient stagewise pre-training via progressive subnetworks. *arXiv preprint arXiv:2402.05913*, 2024.
- Prasanna, S., Rogers, A., and Rumshisky, A. When bert plays the lottery, all tickets are winning. *arXiv preprint arXiv:2005.00561*, 2020.
- Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. 2019.
- Raina, R., Madhavan, A., and Ng, A. Y. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880, 2009.
- Renda, A., Frankle, J., and Carbin, M. Comparing rewinding and fine-tuning in neural network pruning. *arXiv preprint arXiv:2003.02389*, 2020.
- Sanh, V., Wolf, T., and Rush, A. Movement pruning: Adaptive sparsity by fine-tuning. *Advances in neural information processing systems*, 33:20378–20389, 2020.
- Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.
- Su, J., Chen, Y., Cai, T., Wu, T., Gao, R., Wang, L., and Lee, J. D. Sanity-checking pruning methods: Random tickets can win the jackpot. *Advances in neural information processing systems*, 33:20390–20401, 2020.
- Sun, M., Liu, Z., Bair, A., and Kolter, J. Z. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*, 2023.
- Sun, M., Liu, Z., Bair, A., and Kolter, J. Z. A simple and effective pruning approach for large language models, 2024. URL <https://arxiv.org/abs/2306.11695>.
- Taboga, M. Chebyshev’s weak law of large numbers, 2021. URL <https://www.statlect.com/asymptotic-theory/law-of-large-numbers>.
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- Wang, S., Sun, T., Xu, S., Wu, T., Li, L., Xu, S., and Wu, P. GraSP: Gradient signal preservation for pruning neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. URL <https://arxiv.org/abs/2002.07376>. Published on arXiv as 2002.07376.
- Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.
- Wolfe, C. R., Yang, J., Liao, F., Chowdhury, A., Dun, C., Bayer, A., Segarra, S., and Kyriolidis, A. GIST: Distributed training for large-scale graph convolutional networks. *Journal of Applied and Computational Topology*, 8(5):1363–1415, 2024.
- Wright, L., Huang, H., Huang, C.-C., Cala, M., and Petersen, E. Fault tolerant llama: training with 2000 synthetic failures every 15 seconds and no checkpoints on Crusoe L40S. PyTorch Blog, 2024. URL <https://pytorch.org/blog/fault-tolerant-llama-training-with-2000-synthetic-failures-every-15-seconds-and-no-checkpoints-on-crusoe-l40s/>.
- Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T.-Y. On layer normalization in the transformer architecture. *arXiv preprint arXiv:2002.04745*, 2020.
- Yang, Y., Zhen, K., Ganesh, B., Galstyan, A., Huybrechts, G., Müller, M., Kübler, J. M., Swaminathan, R. V., Mouchtaris, A., Bodapati, S. B., et al. Wanda++: Pruning large language models via regional gradients. *arXiv preprint arXiv:2503.04992*, 2025.
- Yao, Z., Yazdani Aminabadi, R., Zhang, M., Wu, X., Li, C., and He, Y. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in neural information processing systems*, 35: 27168–27183, 2022.

Yuan, B., Wolfe, C. R., Dun, C., Tang, Y., Kyrillidis, A., and Jermaine, C. M. Distributed learning of deep neural networks using independent subnet training. *arXiv preprint arXiv:1910.02120*, 2019.

Zheng, Y., Ren, Y., Xia, X., Xiao, X., and Xie, X. Dense2moe: Restructuring diffusion transformer to moe for efficient text-to-image generation. *arXiv preprint arXiv:2510.09094*, 2025. URL <https://arxiv.org/abs/2510.09094>.

A BACKGROUND AND NOTATION

This section introduces the fundamental components and notation for the transformer architecture, primarily following the conventions from the paper by Panigrahi et al. (2024).

A.1 Key Components

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

Multi-Head Attention. From Vaswani et al. (2017), the scaled dot-product attention function for a single head is defined as

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

where $\mathbf{Q} \in \mathbb{R}^{N \times d_{\text{head}}}$, $\mathbf{K} \in \mathbb{R}^{N \times d_{\text{head}}}$, and $\mathbf{V} \in \mathbb{R}^{N \times d_{\text{head}}}$. 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. This is a significant improvement over single-head attention. A multi-head attention layer, denoted as f_{attn} , processes a sequence of input vectors $\{\mathbf{x}_n\}_{n=1}^N$ and outputs a sequence $\{\mathbf{y}_n\}_{n=1}^N$. The layer uses H attention heads, where each head h has its own parameter matrices $\{\mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}\}$.

The output is the concatenation of the outputs from each head, projected by a final matrix $\mathbf{C}^{\text{attn}} \in \mathbb{R}^{Hd_{\text{head}} \times d_{\text{model}}}$:

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

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

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

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

A.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

$\mathbf{X} \in \mathbb{R}^{N \times d_{\text{model}}}$, the layer produces an output matrix $\mathbf{Y} \in \mathbb{R}^{N \times d_{\text{model}}}$ through the following steps:

1. **Attention Layer Normalization:** The input vectors are first normalized. Let $\mathbf{X}_{\text{attnln}}$ be the matrix where each row is the result of applying layer normalization to the corresponding row of \mathbf{X} : $\mathbf{X}_{\text{attnln}} = f_{\text{ln}}(\mathbf{X}; \gamma_{\text{attn}}, \mathbf{b}_{\text{attn}})$.
2. **Multi-Head Attention:** The normalized vectors are passed through the multi-head attention layer: $\mathbf{Y}_{\text{attn}} = f_{\text{attn}}(\mathbf{X}_{\text{attnln}})$.
3. **Residual Connection:** A residual connection is added: $\mathbf{Y}_{\text{attnlayer}} = \mathbf{X} + \mathbf{Y}_{\text{attn}}$.
4. **FFN Layer Normalization:** The result is normalized before the FFN: $\mathbf{X}_{\text{ffnlm}} = f_{\text{ln}}(\mathbf{Y}_{\text{attnlayer}}; \gamma_{\text{ffn}}, \mathbf{b}_{\text{ffn}})$.
5. **FFN Function:** The normalized vectors are processed by the FFN. Since the FFN operates on individual vectors, this is applied row-wise: $\mathbf{Y}_{\text{ffn}} = f_{\text{ffn}}(\mathbf{X}_{\text{ffnlm}})$.
6. **Final Output:** A final residual connection yields the output of the layer: $\mathbf{Y} = \mathbf{Y}_{\text{attnlayer}} + \mathbf{Y}_{\text{ffn}}$.

A.3 Initialization

The weights for the transformer layer are initialized following He et al. (2015):

$$\begin{aligned} \mathbf{W}_h^Q, \mathbf{W}_h^K, \mathbf{W}_h^V &\sim \mathcal{N}\left(0, \frac{1}{d_{\text{model}}} \mathbf{I}\right) \\ \mathbf{C}^{\text{attn}} &\sim \mathcal{N}\left(0, \frac{1}{Hd_{\text{head}}} \mathbf{I}\right) \\ \mathbf{W}^{\text{ffn}} &\sim \mathcal{N}\left(0, \frac{2}{d_{\text{model}}} \mathbf{I}\right) \\ \mathbf{C}^{\text{ffn}} &\sim \mathcal{N}\left(0, \frac{1}{d_{\text{inner}}} \mathbf{I}\right) \end{aligned}$$

The layer normalization parameters γ and \mathbf{b} for both $f_{\text{LN}}^{\text{attn}}$ and $f_{\text{LN}}^{\text{ffn}}$ are initialized to 1 and 0, respectively.

For much of the proofs below, we follow in He's convention (He et al., 2015) and assume \mathbf{X} has i.i.d. components.

Lemma A.1. Let $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ be a random vector with i.i.d. components. Let $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ be a random matrix with i.i.d. components, $\mathbf{W}_{ji} \sim \mathcal{N}(0, \sigma_W^2)$. If we let $\mathbf{y} := \mathbf{W}\mathbf{x}$, then the expected squared norm of the output is

$$\mathbb{E}[\|\mathbf{y}\|^2] = d_{\text{out}}\sigma_W^2 \mathbb{E}[\|\mathbf{x}\|^2]$$

Proof. We expand the squared norm and use the linearity of

expectation:

$$\begin{aligned}\mathbb{E}[\|\mathbf{y}\|^2] &= \mathbb{E}\left[\sum_{j=1}^{d_{\text{out}}}\left(\sum_{i=1}^{d_{\text{in}}} \mathbf{W}_{ji} \mathbf{x}_i\right)^2\right] \\ &= \sum_{j=1}^{d_{\text{out}}} \mathbb{E}\left[\sum_{i,k} \mathbf{W}_{ji} \mathbf{W}_{jk} \mathbf{x}_i \mathbf{x}_k\right].\end{aligned}$$

Since the components of \mathbf{W} are i.i.d. with zero mean and are independent of \mathbf{x} , the cross-terms vanish on expectation: $\mathbb{E}[\mathbf{W}_{ji} \mathbf{W}_{jk} \mathbf{x}_i \mathbf{x}_k] = \mathbb{E}[\mathbf{W}_{ji}] \mathbb{E}[\mathbf{W}_{jk}] \mathbb{E}[\mathbf{x}_i \mathbf{x}_k] = 0$. We are left only with the terms where $i = k$:

$$\begin{aligned}\mathbb{E}[\|\mathbf{y}\|^2] &= \sum_{j=1}^{d_{\text{out}}} \sum_{i=1}^{d_{\text{in}}} \mathbb{E}[\mathbf{W}_{ji}^2 \mathbf{x}_i^2] \\ &= \sum_{j=1}^{d_{\text{out}}} \sum_{i=1}^{d_{\text{in}}} \mathbb{E}[\mathbf{W}_{ji}^2] \mathbb{E}[\mathbf{x}_i^2] \\ &= \sum_{j=1}^{d_{\text{out}}} \sum_{i=1}^{d_{\text{in}}} \sigma_W^2 \mathbb{E}[\mathbf{x}_i^2] \\ &= d_{\text{out}} \sigma_W^2 \sum_{i=1}^{d_{\text{in}}} \mathbb{E}[\mathbf{x}_i^2] \\ &= d_{\text{out}} \sigma_W^2 \mathbb{E}[\|\mathbf{x}\|^2].\end{aligned}\quad \square$$

Lemma A.2. Let $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$ be a random vector with i.i.d. components. Let $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ be a random matrix with i.i.d. components, $\mathbf{W}_{ji} \sim \mathcal{N}(0, \sigma_W^2)$. If $\sigma_{\text{relu}}(\cdot)$ is the element-wise ReLU activation function, then the expected squared norm of the output is

$$\mathbb{E}[\|\sigma_{\text{relu}}(\mathbf{W}\mathbf{x})\|^2] = \frac{1}{2} d_{\text{out}} \sigma_W^2 \mathbb{E}[\|\mathbf{x}\|^2]$$

Proof. Let $\mathbf{y} = \mathbf{W}\mathbf{x}$. Since each \mathbf{W}_{ji} is drawn from a distribution symmetric about 0 and is independent of \mathbf{x} , each pre-activation component $\mathbf{y}_j = \sum_i \mathbf{W}_{ji} \mathbf{x}_i$ also has a distribution symmetric about 0. For any such random variable, this implies $\mathbb{E}[\max(0, \mathbf{y}_j)^2] = \frac{1}{2} \mathbb{E}[\mathbf{y}_j^2]$. The expected squared norm is then

$$\begin{aligned}\mathbb{E}[\|\sigma_{\text{relu}}(\mathbf{y})\|^2] &= \sum_{j=1}^{d_{\text{out}}} \mathbb{E}[\max(0, \mathbf{y}_j)^2] \\ &= \frac{1}{2} \sum_{j=1}^{d_{\text{out}}} \mathbb{E}[\mathbf{y}_j^2] = \frac{1}{2} \mathbb{E}[\|\mathbf{y}\|^2].\end{aligned}$$

The result follows by substituting $\mathbb{E}[\|\mathbf{y}\|^2] = d_{\text{out}} \sigma_W^2 \mathbb{E}[\|\mathbf{x}\|^2]$ from Lemma A.1. \square

Theorem A.3. Let $\mathbf{y}_{\text{ffn}} = \mathbf{C}^{\text{ffn}} \sigma_{\text{relu}}(\mathbf{W}^{\text{ffn}} \mathbf{x})$ be a two-layer FFN with inner dimension d_{inner} and input $\mathbf{x} \in \mathbb{R}^{d_{\text{model}}}$ with

i.i.d. components. Let \mathbf{y}'_{ffn} be the output after reducing the inner dimension to $d'_{\text{inner}} \leq d_{\text{inner}}$ by taking a subset of the original weights \mathbf{W}^{ffn} and \mathbf{C}^{ffn} . The expected squared output norm scales as

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

Proof. Let $\mathbf{W}' \in \mathbb{R}^{d'_{\text{inner}} \times d_{\text{model}}}$ and $\mathbf{C}' \in \mathbb{R}^{d_{\text{model}} \times d'_{\text{inner}}}$ be the submatrices of the original weights. By our initialization, the variances are $\text{Var}(\mathbf{W}_{ji}^{\text{ffn}}) = \frac{2}{d_{\text{model}}}$ and $\text{Var}(\mathbf{C}_{kj}^{\text{ffn}}) = \frac{1}{d_{\text{inner}}}$. Since \mathbf{W}' and \mathbf{C}' are subsets of these weights, their elements retain the same variances. Now, we derive the expected squared norm of the modified network's output using Lemma A.1 and A.2.

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

For the original network, we can set $d'_{\text{inner}} = d_{\text{inner}}$ in the above derivation, which yields $\mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|^2] = \mathbb{E}[\|\mathbf{x}\|^2]$. Substituting this back into the previous result gives the theorem. \square

Lemma A.4. Let $\mathbf{x} \in \mathbb{R}^{d_{\text{model}}}$ be a random vector with i.i.d. components. Let \mathbf{y}_{ffn} and \mathbf{y}'_{ffn} be as in Thm. A.3. Then for each $i \in \{1, \dots, d_{\text{model}}\}$,

$$\text{Var}(\mathbf{y}'_{\text{ffn}_i}) = \frac{d'_{\text{inner}}}{d_{\text{inner}}} \text{Var}(\mathbf{y}_{\text{ffn}_i}).$$

Proof. The output components have zero mean (e.g., $\mathbb{E}[\mathbf{y}_{\text{ffn}_i}] = 0$) because the second-layer weights \mathbf{C}^{ffn} are initialized with zero mean and are independent of the first-layer activations. Thus, the variance is the expected squared value: $\text{Var}(\mathbf{y}_{\text{ffn}_i}) = \mathbb{E}[(\mathbf{y}_{\text{ffn}_i})^2]$.

Since the output components are identically distributed, we can relate the variance of a single component to the expected squared norm of the full output vector as

$$\text{Var}(\mathbf{y}_{\text{ffn}_i}) = \frac{1}{d_{\text{model}}} \sum_{k=1}^{d_{\text{model}}} \text{Var}(\mathbf{y}_{\text{ffn}_k}) = \frac{1}{d_{\text{model}}} \mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|^2].$$

The same reasoning holds for \mathbf{y}'_{ffn} . From Thm. A.3, we

know that $\mathbb{E}[\|\mathbf{y}'_{\text{ffn}}\|^2] = \frac{d'_{\text{inner}}}{d_{\text{inner}}} \mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|^2]$. It follows that:

$$\begin{aligned}\text{Var}(\mathbf{y}'_{\text{ffn}_i}) &= \frac{1}{d_{\text{model}}} \mathbb{E}[\|\mathbf{y}'_{\text{ffn}}\|^2] \\ &= \frac{d'_{\text{inner}}}{d_{\text{inner}}} \left(\frac{1}{d_{\text{model}}} \mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|^2] \right) \\ &= \frac{d'_{\text{inner}}}{d_{\text{inner}}} \text{Var}(\mathbf{y}_{\text{ffn}_i}).\end{aligned}\quad \square$$

Lemma A.5. Let $\mathbf{x} \in \mathbb{R}^{d_{\text{model}}}$ be a random vector with i.i.d. components, zero mean, and unit variance. Let the output of a ffn block be $\mathbf{y}_{\text{ffn}} = \mathbf{C}^{\text{ffn}} \sigma_{\text{relu}}(\mathbf{W}^{\text{ffn}} \mathbf{x})$, where weights are initialized as $\mathbf{W}^{\text{ffn}} \sim \mathcal{N}(0, \frac{2}{d_{\text{model}}} \mathbf{I})$ and $\mathbf{C}^{\text{ffn}} \sim \mathcal{N}(0, \frac{1}{d_{\text{inner}}} \mathbf{I})$. Then for each component i and large d_{model} ,

$$\text{Var}([\mathbf{y}_{\text{ffn}}]_i) = 1.$$

Proof. Let $\mathbf{z} = \sigma_{\text{relu}}(\mathbf{W}^{\text{ffn}} \mathbf{x})$. The i -th output is $[\mathbf{y}_{\text{ffn}}]_i = \sum_j \mathbf{C}_{ij}^{\text{ffn}} \mathbf{z}_j$. Since the weights $\mathbf{C}_{ij}^{\text{ffn}}$ are i.i.d., zero-mean, and independent of the activations \mathbf{z}_j , we can compute the variance. By the Bienaymé formula, the variance of a sum of independent variables is the sum of their variances. Furthermore, for two independent variables X and Y where $\mathbb{E}[X] = 0$, the variance of their product is $\text{Var}(XY) = \text{Var}(X)\mathbb{E}[Y^2]$. This gives

$$\begin{aligned}\text{Var}([\mathbf{y}_{\text{ffn}}]_i) &= \sum_{j=1}^{d_{\text{inner}}} \text{Var}(\mathbf{C}_{ij}^{\text{ffn}} \mathbf{z}_j) \\ &= \sum_{j=1}^{d_{\text{inner}}} \text{Var}(\mathbf{C}_{ij}^{\text{ffn}}) \mathbb{E}[\mathbf{z}_j^2] \\ &= \sum_{j=1}^{d_{\text{inner}}} \frac{1}{d_{\text{inner}}} \mathbb{E}[\mathbf{z}_j^2] = \mathbb{E}[\mathbf{z}_1^2].\end{aligned}$$

The last equality holds as the activations \mathbf{z}_j are identically distributed.

Let $\mathbf{a}_j = [\mathbf{W}^{\text{ffn}} \mathbf{x}]_j$ be the j -th pre-activation. Its variance is $\text{Var}(\sum_{k=1}^{d_{\text{model}}} \mathbf{W}_{jk}^{\text{ffn}} \mathbf{x}_k)$. Since the terms in the sum are independent, the variance is the sum of the variances. Moreover the weights $\mathbf{W}_{jk}^{\text{ffn}}$ and inputs \mathbf{x}_k are independent and zero-mean, so the variance of their product is the product of their variances, $\text{Var}(\mathbf{W}_{jk}^{\text{ffn}} \mathbf{x}_k) = \text{Var}(\mathbf{W}_{jk}^{\text{ffn}}) \text{Var}(\mathbf{x}_k)$. We now have

$$\begin{aligned}\text{Var}(\mathbf{a}_j) &= \sum_{k=1}^{d_{\text{model}}} \text{Var}(\mathbf{W}_{jk}^{\text{ffn}} \mathbf{x}_k) \\ &= \sum_{k=1}^{d_{\text{model}}} \text{Var}(\mathbf{W}_{jk}^{\text{ffn}}) \text{Var}(\mathbf{x}_k) \\ &= \sum_{k=1}^{d_{\text{model}}} \frac{2}{d_{\text{model}}} \cdot 1 = 2.\end{aligned}$$

For large d_{model} , the Central Limit Theorem implies $\mathbf{a}_j \sim \mathcal{N}(0, 2)$. For a zero-mean Gaussian $v \sim \mathcal{N}(0, \sigma^2)$, the second moment of its ReLU activation is $\mathbb{E}[(\sigma_{\text{relu}}(v))^2] = \frac{1}{2} \text{Var}(v)$. Thus,

$$\mathbb{E}[\mathbf{z}_1^2] = \mathbb{E}[(\sigma_{\text{relu}}(\mathbf{a}_1))^2] = \frac{1}{2} \text{Var}(\mathbf{a}_1) = \frac{2}{2} = 1.$$

Substituting this back yields $\text{Var}([\mathbf{y}_{\text{ffn}}]_i) = 1$. \square

Theorem A.6 (FFN Scale Factor). Let $\mathbf{x} \in \mathbb{R}^{d_{\text{model}}}$ be a random vector with i.i.d. components, zero mean, and unit variance. Given a two layer ffn as defined in Section A

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

with inner dimension d_{inner} , changing the inner dimension to $d'_{\text{inner}} \in \{0, 1, 2, \dots, d_{\text{inner}}\}$ without resampling the weights (i.e., taking a subset of the parameters) scales the expected output norm as follows with high probability:

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

for sufficiently large d_{model} .

Proof. From Theorem A.3, we have an exact relation for the expected squared norms:

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

Taking the square root of both sides gives:

$$\sqrt{\mathbb{E}[\|\mathbf{y}'_{\text{ffn}}\|^2]} = \sqrt{\frac{d'_{\text{inner}}}{d_{\text{inner}}}} \sqrt{\mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|^2]} \quad (1)$$

We argue that the approximation $\mathbb{E}[\|\cdot\|] = \sqrt{\mathbb{E}[\|\cdot\|^2]}$ holds for high-dimensional random vectors like \mathbf{y}_{ffn} and \mathbf{y}'_{ffn} . For a vector $\mathbf{z} \in \mathbb{R}^{d_{\text{model}}}$, the squared norm $\|\mathbf{z}\|^2 = \sum_{i=1}^{d_{\text{model}}} z_i^2$ is a sum of a large number of random variables. As established in Lemma A.5 and A.4, the components of \mathbf{y}_{ffn} and \mathbf{y}'_{ffn} have well-behaved variances. Moreover, as an intermediate result we have Uncorrelated Output Components from A.5. So, for a sufficiently large d_{model} , Chebyshev's Weak Law of Large Numbers (Taboga, 2021) implies that the sum $\|\cdot\|^2$ will be sharply concentrated around its expected value, $\mathbb{E}[\|\cdot\|^2]$.

When a random variable is highly concentrated, its value is very close to its mean with high probability. By the continuity of the square-root function, if $\|\cdot\|^2$ is concentrated around $\mathbb{E}[\|\cdot\|^2]$, then its square root, $\|\cdot\|$, must be concentrated around $\sqrt{\mathbb{E}[\|\cdot\|^2]}$. The expectation of a tightly

concentrated random variable is nearly equal to the value it is concentrated around. Therefore, for large d_{model} , we have:

$$\begin{aligned}\mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|] &\approx \sqrt{\mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|^2]} \\ \mathbb{E}[\|\mathbf{y}'_{\text{ffn}}\|] &\approx \sqrt{\mathbb{E}[\|\mathbf{y}'_{\text{ffn}}\|^2]}\end{aligned}$$

Substituting these high-probability approximations into Equation (1) directly yields the desired result:

$$\mathbb{E}[\|\mathbf{y}'_{\text{ffn}}\|] = \sqrt{\frac{d'_{\text{inner}}}{d_{\text{inner}}}} \mathbb{E}[\|\mathbf{y}_{\text{ffn}}\|] \quad \square$$

Theorem A.7. *Given a Multi-Head Attention layer with output $\mathbf{Y}_{\text{attn}} = \text{Concat}(\text{head}_1, \dots, \text{head}_H) \mathbf{C}^{\text{attn}}$, reducing the number of heads to $H' \leq H$ by selecting a subset scales the expected squared norm of the output rows as:*

$$\mathbb{E}[\|[\mathbf{Y}'_{\text{attn}}]_i\|^2] = \frac{H'}{H} \mathbb{E}[\|[\mathbf{Y}_{\text{attn}}]_i\|^2].$$

Proof. Let $\mathbf{A} = \text{Concat}(\text{head}_1, \dots, \text{head}_H)$. The projection matrix \mathbf{C}^{attn} is initialized independently from \mathbf{A} with zero-mean entries, so $\mathbb{E}[[\mathbf{Y}_{\text{attn}}]_i] = 0$. The expected squared norm of an output row is:

$$\begin{aligned}\mathbb{E}[\|[\mathbf{Y}_{\text{attn}}]_i\|^2] &= \sum_{j=1}^{d_{\text{model}}} \text{Var} \left(\sum_{k=1}^{Hd_{\text{head}}} \mathbf{A}_{ik} \mathbf{C}_{kj}^{\text{attn}} \right) \\ &= \sum_j \sum_k \text{Var}(\mathbf{A}_{ik} \mathbf{C}_{kj}^{\text{attn}}) \\ &= \sum_j \sum_k \mathbb{E}[\mathbf{A}_{ik}^2] \text{Var}(\mathbf{C}_{kj}^{\text{attn}}).\end{aligned}$$

The second equality holds because the entries of \mathbf{C}^{attn} are i.i.d. and zero-mean, causing covariance terms from the Bienaymé formula to vanish. The third holds due to the independence of \mathbf{A} and \mathbf{C}^{attn} . Let $\sigma_C^2 = \text{Var}(\mathbf{C}_{kj}^{\text{attn}})$. As all heads are statistically identical due to i.i.d. initialization, we can rewrite the sum over the concatenated dimension k as a factor of H :

$$\mathbb{E}[\|[\mathbf{Y}_{\text{attn}}]_i\|^2] = H \left(\sigma_C^2 \sum_{j=1}^{d_{\text{model}}} \sum_{l=1}^{d_{\text{head}}} \mathbb{E}[\mathbf{A}_{i,l}^2] \right),$$

where the sum over l is across a single head's dimensions. The term in parentheses is an invariant with respect to the number of heads. For a layer with H' heads, the leading factor is simply H' . The theorem follows by taking the ratio.

Lemma A.8. *Let $\mathbf{X} \in \mathbb{R}^{N \times d_{\text{model}}} \sim \mathcal{N}(0, \mathbf{I})$. For a Multi-Head Attention layer with H heads, $\mathbf{Y}_{\text{attn}} = \text{Concat}(\text{head}_1, \dots, \text{head}_H) \mathbf{C}^{\text{attn}}$, reducing the number of heads to $H' \leq H$ by taking a subset scales the output variance as*

$$\text{Var}([\mathbf{Y}'_{\text{attn}}]_{ij}) = \frac{H'}{H} \text{Var}([\mathbf{Y}_{\text{attn}}]_{ij}).$$

Proof. At initialization, we approximate the softmax as attending to all tokens equally (Kedia et al., 2024; Chi et al., 2023; Xiong et al., 2020). Thus, $\text{head}_h \approx \frac{1}{N} \mathbf{1}_N \mathbf{1}_N^\top \mathbf{X} \mathbf{W}_h^V$. Let $\tilde{\mathbf{X}} = \frac{1}{N} \mathbf{1}_N \mathbf{1}_N^\top \mathbf{X}$. Since $\mathbf{X}_{lk} \sim \mathcal{N}(0, 1)$ are i.i.d., the elements of $\tilde{\mathbf{X}}$ have $\mathbb{E}[\bar{x}_{ik}] = 0$ and $\text{Var}(\bar{x}_{ik}) = 1/N$.

The elements of a single head, $[\text{head}_h]_{ij} = \sum_{k=1}^{d_{\text{model}}} \bar{x}_{ik} [\mathbf{W}_h^V]_{kj}$, are zero-mean. By our initialization, the weights $[\mathbf{W}_h^V]_{kj}$ are i.i.d. with variance σ_V^2 and are independent of $\tilde{\mathbf{X}}$. So, the variance $\text{Var}([\text{head}_h]_{ij})$ is

$$\sum_{k=1}^{d_{\text{model}}} \text{Var}(\bar{x}_{ik}) \text{Var}([\mathbf{W}_h^V]_{kj}) = \frac{d_{\text{model}}}{N} \sigma_V^2.$$

Let $\mathbf{Z} = \text{Concat}(\text{head}_1, \dots, \text{head}_H)$, and let σ_C^2 be the variance of $\mathbf{C}_{kj}^{\text{attn}}$. Notice the elements of the output layer $[\mathbf{Y}_{\text{attn}}]_{ij} = \sum_{k=1}^{Hd_{\text{head}}} \mathbf{Z}_{ik} \mathbf{C}_{kj}^{\text{attn}}$ are zero-mean. Since each head is independent, the variance is

$$\begin{aligned}\text{Var}([\mathbf{Y}_{\text{attn}}]_{ij}) &= \sum_{k=1}^{Hd_{\text{head}}} \text{Var}(\mathbf{Z}_{ik}) \text{Var}(\mathbf{C}_{kj}^{\text{attn}}) \\ &= (Hd_{\text{head}}) \left(\frac{d_{\text{model}}}{N} \sigma_V^2 \right) \sigma_C^2.\end{aligned}\quad (2)$$

For a layer with a subset of H' heads, the derivation is identical, replacing H with H' . Taking the ratio with Eq. (2) gives the result. \square

Lemma A.9. *Let $\mathbf{X} \in \mathbb{R}^{N \times d_{\text{model}}} \sim \mathcal{N}(0, \mathbf{I})$. Given a Multi-Head Attention layer as defined in Section A. Then for each component of the attention layer output $i \in \{0, 1, \dots, N-1\}$, $j \in \{0, 1, \dots, d_{\text{model}}\}$,*

$$\text{Var}([\mathbf{Y}_{\text{attn}}]_{ij}) = \frac{1}{N}$$

for sufficiently large d_{model} .

Proof. By Equation 2 of Lemma A.8

$$\begin{aligned}\text{Var}([\mathbf{Y}_{\text{attn}}]_{ij}) &= Hd_{\text{head}} \frac{d_{\text{model}}}{N} \sigma_x^2 \sigma_V^2 \sigma_C^2 \\ &= Hd_{\text{head}} \frac{d_{\text{model}}}{N} (1) \left(\frac{1}{d_{\text{model}}} \right) \left(\frac{1}{Hd_{\text{head}}} \right) \\ &= \frac{1}{N}\end{aligned}\quad \square$$

Theorem A.10 (Attention Scale Factor). *Let $\mathbf{X} \in \mathbb{R}^{N \times d_{\text{model}}} \sim \mathcal{N}(0, \mathbf{I})$. For a Multi-Head Attention layer with H heads, $\mathbf{Y}_{\text{attn}} = \text{Concat}(\text{head}_1, \dots, \text{head}_H) \mathbf{C}^{\text{attn}}$, reducing the number of heads to $H' \leq H$ by taking a subset scales the expected output norm as*

$$\mathbb{E}[\|[\mathbf{Y}'_{\text{attn}}]_i\|] = \sqrt{\frac{H'}{H}} \mathbb{E}[\|[\mathbf{Y}_{\text{attn}}]_i\|].$$

Proof. We argue similarly to Theorem A.6. From Theorem A.7, the expected squared norms are exactly related by $\mathbb{E}[\|\mathbf{Y}'_{\text{attn}}\|_i^2] = \frac{H'}{H} \mathbb{E}[\|\mathbf{Y}_{\text{attn}}\|_i^2]$. Taking the square root gives

$$\sqrt{\mathbb{E}[\|\mathbf{Y}'_{\text{attn}}\|_i^2]} = \sqrt{\frac{H'}{H}} \sqrt{\mathbb{E}[\|\mathbf{Y}_{\text{attn}}\|_i^2]}. \quad (3)$$

For large d_{model} , the squared norm $\|\mathbf{Y}_{\text{attn}}\|_i^2$ is a sum over many uncorrelated components with finite variance (Lemmas A.9 and A.8). By the Law of Large Numbers, the squared norm concentrates around its mean. The continuous mapping theorem then implies that the norm concentrates around the square root of the mean, justifying the approximation $\mathbb{E}[\|\cdot\|] \approx \sqrt{\mathbb{E}[\|\cdot\|^2]}$. Substituting this into Eq. (3) yields the result. \square

B ADDITIONAL EXPERIMENTS

B.1 Additional System Stability Experiments

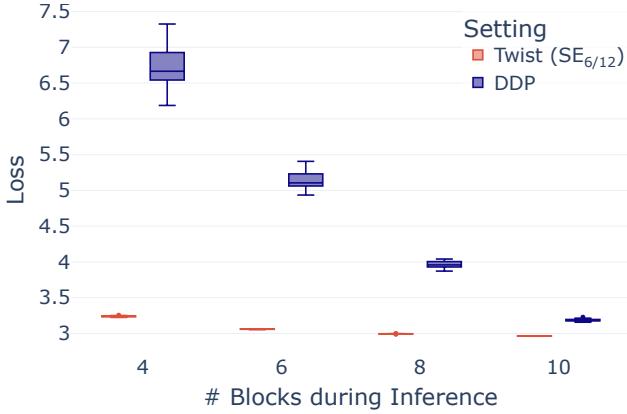


Figure 7. Distribution of eval loss for randomly generated subnets in the `both` configuration. The distributions for TwIST ($SE_{6/12}$) are compared against a DDP baseline across various subnet ratios. The $SE_{6/12}$ variant of Twist is presented for a direct comparison, as both this method and DDP involve only a single training pass.

Figure 7 presents a detailed breakdown of the evaluation loss distributions for the TwIST ($SE_{6/12}$) method compared against the DDP baseline. This analysis was performed on randomly generated subnets, varying the number of active blocks during inference (4, 6, 8, and 10). The results demonstrate the significant stability and superior performance of the TwIST method. Across all tested subnet configurations, TwIST maintains a consistent and low evaluation loss (approx. 3.1-3.2) with negligible variance. In contrast, the DDP baseline exhibits both substantially higher loss and high variance, particularly with fewer active blocks. The median loss for DDP is highest at 4 blocks (approx. 6.8) and progressively improves as the number of blocks increases, eventually approaching the loss of TwIST at 10 blocks

(approx. 3.2). This comparison highlights that TwIST provides robust performance regardless of the active subnet width, a significant advantage over the more variable and width-dependent DDP baseline.

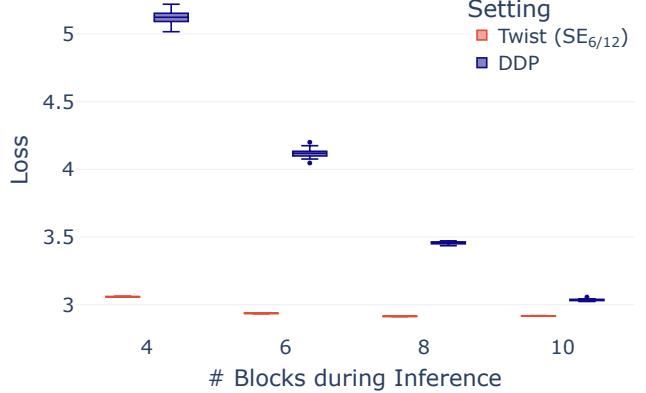


Figure 8. Distribution of eval loss for randomly generated subnets in the `ffn` configuration. The distributions for TwIST ($SE_{6/12}$) are compared against a DDP baseline across various subnet ratios. The $SE_{6/12}$ variant of Twist is presented for a direct comparison, as both this method and DDP involve only a single training pass.

This finding is further reinforced by the analysis of the `ffn` configuration, shown in Figure 8. The results exhibit a virtually identical pattern: the TwIST ($SE_{6/12}$) method again achieves a consistently low loss (approx. 3.0) with near-zero variance, irrespective of the number of active blocks. The DDP baseline, while demonstrating lower overall losses in this configuration (e.g., a median loss of approx. 5.1 at 4 blocks, compared to 6.8 in Figure 7), displays the same critical dependency on network width. Its loss and variance are highest at the leanest width and only converge with TwIST's superior performance when the full 10 blocks are utilized. Taken together, Figures 7 and 8 provide strong evidence that the TwIST methodology produces subnets that are robustly performant at arbitrary widths, a key advantage over the width-sensitive DDP baseline.

B.2 Additional Architectural Robustness Experiments

Figures 9 and 10 further investigate the issue of subnet robustness by analyzing the mismatch between training and inference depths. Figure 9 provides a perplexity (PPL) heatmap for the `both` setting, where lower PPL (brighter yellow) is better. The x-axis represents the number of blocks used during training (e.g., “On 4”), and the y-axis represents the number of blocks used during inference. The plot reveals a severe performance degradation when models trained on wide subnets are inferred on lean ones. For instance, the model trained “On 10” blocks, while performing well at 10-block inference (17.97 PPL), experiences a catastrophic failure at 4-block inference (60.05 PPL). Conversely, the model trained “On 4” blocks shows remarkable robustness,

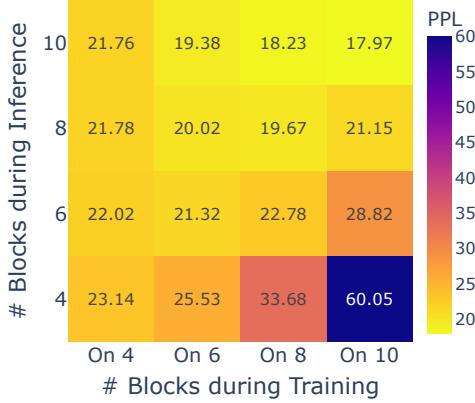


Figure 9. Heatmap of subnet robustness for the both setting. Brighter colors (yellow) signify lower PPL (better performance), and darker colors (blue) signify higher PPL.

maintaining a stable PPL (between 21.76 and 23.14) across all inference depths.

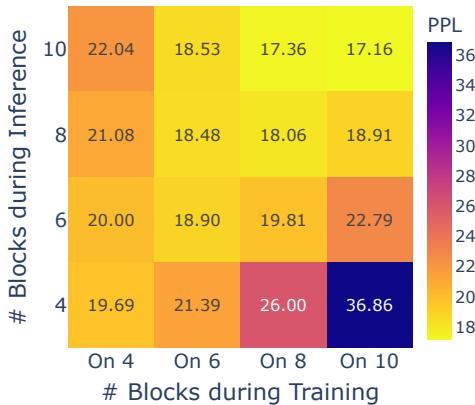


Figure 10. Heatmap of subnet robustness for the ffn setting. Brighter colors (yellow) signify lower PPL (better performance), and darker colors (blue) signify higher PPL.

Figure 10 confirms this exact finding in the ffn configuration. The identical trend is observed: models trained on wide subnets are “brittle” and not robust to leaner inference. The model trained “On 10” blocks sees its PPL degrade from 17.16 (at 10-block inference) to 36.86 (at 4-block inference). In stark contrast, the model trained “On 4” blocks again proves the more robust, with its PPL remaining in a tight, low range (19.69 to 22.04) regardless of the number of blocks used at inference. Both heatmaps strongly indicate that considering target sparsity from the start of training is crucial for building models that are robust to dynamic inference depths, whereas a traditional training strategy leads to significant performance collapse on smaller subnets.

C COMPUTING MEMORY AND COMMUNICATION

We present formula used for computing the number of parameters in GPT-2 style transformer. Let $N_{\text{embed}} = (N_{\text{vocab}})(d_{\text{model}})$ be the number of parameters in the embedding where N_{vocab} is the size of the vocabulary. Let $N_{\text{attn}} = 4(d_{\text{attn}})(d_{\text{model}}) + 3(d_{\text{attn}}) + d_{\text{model}}$ be the number of parameters in an attention layer where d_{attn} is computed as the number of heads in that layer multiplied by the head dimension. The first term is the total number of parameters in the $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V, \mathbf{C}^{\text{attn}}$ weights. The following two terms are the number of parameters in the biases for the $\{\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V\}$ and \mathbf{C}^{attn} weights respectively. Let $N_{\text{ffn}} = 2(d_{\text{ffn}})(d_{\text{model}}) + d_{\text{ffn}} + d_{\text{model}}$ be the number of parameters in a feedforward layer. Let $N_{\text{ln}} = 2d_{\text{model}}$ be the number of parameters in a layer norm. Then, the number of parameters in a transformer layer is $N_{\text{layer}}^l = N_{\text{ln}} + \alpha_l N_{\text{attn}} + \beta_l N_{\text{ffn}}$ where α is the sparsity of the attention module and β is the sparsity of the feedforward module. Moreover, the total number of parameters in the transformer is

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

where N_{proj} is the number of parameters in the final transformer projection. This depends on the task. For text generation, $N_{\text{proj}} = 0$ assuming tied weights. For text classification, $N_{\text{proj}} = (d_{\text{model}})(N_{\text{labels}})$ where N_{labels} is the number of categories.

The number of parameters is roughly proportional to the amount of space a model takes up on a hardware accelerator both at model load time (where dtype determines bytes per param) and at train time (where it is about four times the model size when using Adam) (Hugging Face Documentation Team, 2025). Since every parameter is both sent to and from a worker in a communication round, the total communication cost is proportional to the number of communication rounds multiplied by the number of parameters.