Large Language Model Compression with Neural Architecture Search

Rhea Sanjay Sukthanker[1] **Benedikt Staffler**[3] **Frank Hutter**[1,2] **Aaron Klein**[4]

[1]University of Freiburg [2]ELLIS Institute Tübingen [3]BCAI [4]ScaDS.AI

Abstract

Large language models (LLMs) exhibit remarkable reasoning abilities, allowing them to generalize across a wide range of downstream tasks, such as commonsense reasoning or instruction following. However, as LLMs scale, inference costs become increasingly prohibitive, accumulating significantly over their life cycle. This poses the question: **Can we compress pre-trained LLMs to meet diverse size and latency requirements?** We leverage *Neural Architecture Search (NAS)* to compress LLMs by pruning structural components, such as attention heads, neurons, and layers, aiming to achieve a Pareto-optimal balance between performance and efficiency. While NAS already achieved promising results on small language models in previous work, in this paper we propose various extensions that allow us to scale to LLMs. Compared to structural pruning baselines, we show that NAS improves performance up to 3.4% on MMLU with an on-device latency speedup.

1 Introduction

Large language models (LLMs) represent a significant advancement in artificial intelligence and are increasingly deployed in products such as chatbots and coding assistants. However, their substantial parameter count leads to high latency and significant computational demands during inference. This renders deployment in time-sensitive or resource-constrained settings, such as embedded systems, often impractical, or results in elevated costs per user query, for example in web services.

LLM providers typically offer various-sized models, such as LLama-2 with 7B, 13B, 34B and 70B parameters [Touvron et al., 2023], enabling users to balance performance against costs. While the training cost scales with model size, the pre-training process is so computationally intensive that even smaller models incur substantial costs; for instance, while Llama 2 70B model required up to 1720320 GPU hours, the 13B model still required 368640 GPU hours for pre-training.

A more efficient alternative is to compress large models while striving to maintain its performance. Distillation techniques [Hinton et al., 2015] train a smaller student model to replicate the predictions of a larger teacher model. Although this still requires training a model from scratch, it generally converges faster than traditional pre-training. Pruning methods [Frantar and Alistarh, 2023a, Ashkboos et al., 2024] remove components from the network that do not contribute to the overall performance. A primary challenge for both pruning and distillation is determining the extent to which model capacity can be reduced without causing a substantial decline in performance.

Neural architecture search (NAS) proved to be an efficient method for compressing neural networks [Klein et al., 2024, Muralidharan et al., 2024] by identifying sparse sub-networks that optimally trade-off between downstream performance and efficiency. Specifically, two-stage NAS [Yu et al., 2020], treats the pre-trained network as a super-network composed of a finite number of sub-networks. By modifying the training strategy to update only sub-networks at each step, we avoid co-adaptation, enhancing their performance when extracted from the super-network. After fine-tuning, any black-box optimization method can be employed to select the optimal set of sub-networks that achieve the desired balance of performance and efficiency. Unlike other methods, NAS approximates the Pareto set of sub-networks, capturing the non-linear relationship between sparsity and performance.

Machine Learning and Compression Workshop at NeurIPS 2024.

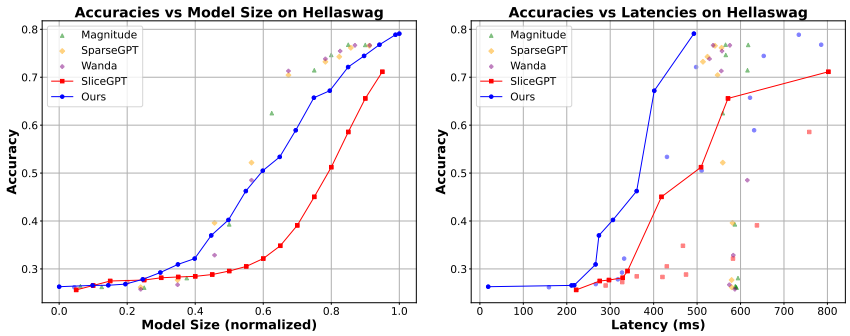


Figure 1: Comparison of Accuracy v/s Latency and Parameter Pareto-Fronts for NAS and pruning baselines on Hellaswag for Llama-3.1-8B

Although two-stage NAS is effective in compressing smaller encoder-only transformer models [Klein et al., 2024], its scalability to larger models poses a challenge. In this paper, we identify gaps in the applicability of NAS for pruning and propose extensions to address these issues:

- **Efficient super-network training:** The total number of sub-networks increases exponentially with the size of the super-network. Standard super-network training strategies, such as the sandwich rule [Yu et al., 2020, 2019], sample sub-networks uniformly at random from the search space. However, this oversamples tiny models that are unlikely to learn effectively due to gradient conflicts [Xu et al., 2022]. We propose a novel sampling strategy to allocate more compute to promising sub-networks.
- **Parameter-efficient fine-tuning:** Fine-tuning LLMs with more than 7B parameters require access to large-scale compute. Building on previous work [Munoz et al., 2024], we empirically investigate the combination of two-stage NAS with state-of-the-art parameter-efficient fine-tuning methods, such as LoRA [Hu et al., 2021], to scale to larger models.
- **On Device Latency Speedups:** We show that weight-sharing based NAS allows us to obtain a Pareto optimal set of architectures with varying latency trade-offs as depicted in Figure 1, that are able to maintain higher downstream performance than models pruned by structural pruning approaches across typical benchmarks for commonsense reasoning.

Section 2 provides an overview of related work on compressing LLMs based on distillation, structural pruning, as well as NAS. Section 3 introduces two stage NAS, and Section 4 provides an overview of our approach to scale weight-sharing based NAS to LLMs. We provide an empirical analysis of our approach across baseline methods from the structural pruning literature and an in-depth analysis of our approach in Section 5.

2 Related Work

Pruning removes neurons or weights of a trained neural networks while maintaining its predictive power. We distinguish between *unstructured pruning*, which removes individual weights [Sun et al., 2024, Han et al., 2015a, Frantar and Alistarh, 2023b], and *structured pruning*, which eliminates groups of weights, such as layers or attention heads [Cai et al., Ashkboos et al., 2024]. A popular unstructured pruning approach is weight magnitude pruning, which masks individual weights by their magnitude [Han et al., 2015a]. Structured pruning, in contrast, targets specific components, such as the embedding dimension [Ashkboos et al., 2024], heads or layers [Muralidharan et al., 2024]. While unstructured pruning often preserves performance at high sparsities, compared to structured approaches, it leads to sparse weight matrices which do not improve on-device latency. Semi-structured N:M pruning [Zhou et al.] aims to combine both methods, offering latency gains, such as 2:4 sparsity on NVIDIA Ampere architectures. In this work, we focus on structured pruning, aiming to identify optimal sparsity blocks for inference improvements.

Neural Architecture Search (NAS) automates the design of deep neural networks in a data-driven manner [Elsken et al., 2019b, White et al., 2023], often optimizing multiple objectives jointly, such as hardware efficiency and predictive performance [Muralidharan et al., 2024, Cai et al., 2020, Wang et al., 2020, Sukthanker et al., 2024b,a, Cai et al.]. Two-stage NAS [Yu et al., 2020] trains a single super-network composing of a finite number of sub-networks. We can consider two-stage NAS as a structural pruning method that identifies sparse sub-parts of the network to balance between

efficiency and performance. For example, Klein et al. [2024] demonstrated its potential to outperform standard structured pruning baselines on small-scale encoder models. Recently, Flextron [Cai et al., Muralidharan et al., 2024] applied NAS to design Pareto-optimal, low-latency architectures by learning routing mechanisms in Llama-2-7b. Similarly, Minitron [Muralidharan et al., 2024] utilized NAS along with importance computation and knowledge distillation to develop smaller, efficient versions of Llama-3.1-8b. However, as the fine-tuning pipelines and datasets used in these works are not publicly available, our study compares against established structured pruning baselines.

3 Background and Notations

This section defines the notations and terminologies used throughout the paper. We begin with an overview of the modular components in transformers in 3.1, followed by an introduction to the various building blocks for NAS in 3.2.

3.1 Transformer Architecture

The transformer architecture [Vaswani et al., 2017] underpins many leading LLMs. We focus on decoder-only architectures [Radford et al., 2019], prevalent in state-of-the-art models. A transformer comprises of an embedding layer followed by L blocks, each containing a self-attention and a fully connected layer, with layer normalization [Ba et al., 2016] or RMS normalization [Zhang and Sennrich, 2019] between layers. We delineate the key components for our search space in Section 4.

Embedding Layer: Each input token is mapped to a $\mathbb{R}^{d_{model}}$ vector using an embedding matrix $\boldsymbol{W}_{emb} \in \mathbb{R}^{V \times d_{model}}$, which is defined by the vocabulary size V and the embedding dimension d_{model}. This results in an input sequence $\boldsymbol{X} \in \mathbb{R}^{N \times d_{model}}$ of length N. Without loss of generality we assume that the weights of the embedding layer and the prediction head are shared by weight-tying [Press and Wolf, 2016].

Attention Layer: Multi-head attention consists of H heads, where each head $i \in [0, H-1]$ has key, query, and value matrices $\boldsymbol{W}_K^{(i)}, \boldsymbol{W}_Q^{(i)}, \boldsymbol{W}_V^{(i)} \in \mathbb{R}^{d_{head} \times d_{model}}$. The head computes $X_i = Att(\boldsymbol{W}_Q^{(i)}, \boldsymbol{W}_K^{(i)}, \boldsymbol{W}_V^{(i)}, \boldsymbol{X})$, leading to the final output $X = Concat(X_0, \ldots, X_{H-1}) \cdot \boldsymbol{W}_O$, where $\boldsymbol{W}_O \in \mathbb{R}^{Hd_{head} \times d_{model}}$.

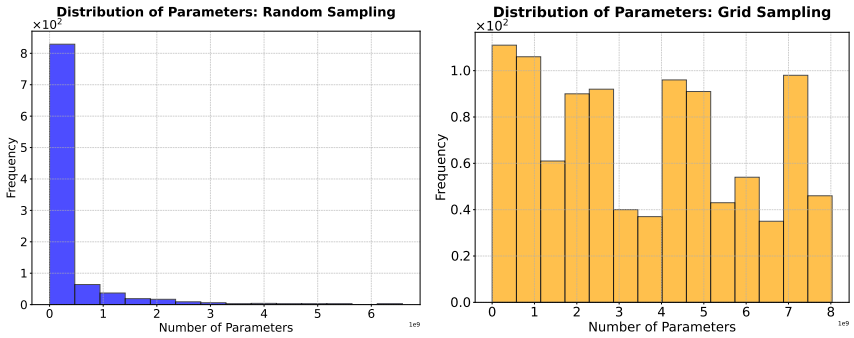
Fully Connected Layer: The FFN layer is defined as $FFN(\boldsymbol{X}) = \boldsymbol{W}_{fc1}\sigma(\boldsymbol{W}_{fc0}\boldsymbol{X})$, with $\boldsymbol{W}_{fc0} \in \mathbb{R}^{U \times d_{model}}$, $\boldsymbol{W}_{fc1} \in \mathbb{R}^{d_{model} \times U}$, and $U = r \cdot d_{model}$. $\sigma(\cdot)$ represents a non-linear activation function.

3.2 Neural Architecture Search

Given a search space Θ composed of architectural design choices, for example the number of heads or layers, NAS (see Elsken et al. [2019b] for an overview) finds the optimal architecture $\theta^\star \in argmin_{\theta \in \Theta} f(\theta)$ that minimizes some error metric $f(\theta)$, such as the validation error $f(\theta) = \mathcal{L}_{valid}(\theta, w_\theta^\star)$ after training the network $w_\theta^\star = \arg\min \mathcal{L}_{train}(w_\theta)$ with a set of weights $w_\theta \in \mathbb{R}^n$. The search space Θ is usually large but finite. We can extend this to multiple objectives $min_{\theta \in \Theta} \{f_0(\theta), ..., f_k(\theta)\}$, such as for example validation error, inference latency or parameter count. In the multi-objective setting we do not have a single solution $\theta_\star$ that simultaneously optimizes all objectives. Instead, we try to approximate the *Pareto Set*: $P_f = \{\theta \in \Theta | \nexists \theta' \in \Theta : \theta' \succ \theta\}$ of points that dominate all other points in the search space in at least one objective, where we define $\theta \succ \theta'$ iff $f_i(\theta) \leq f_i(\theta'), \forall i \in [k]$ and $\exists i \in [k] : f_i(\theta) < f_i(\theta')$.

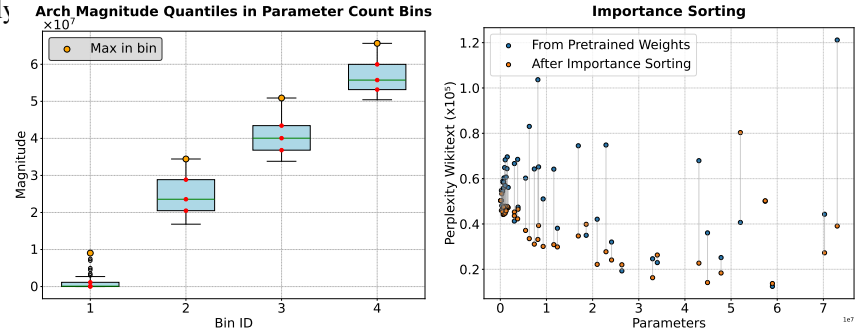
Two-stage NAS. Classical NAS approaches propose evolutionary algorithms [Real et al., 2019] or reinforcement learning methods [Zoph and Le, 2017] to tackle this optimization problem. However, since every evaluation of $f(\theta)$ involves training and validating a neural network model, these methods consume a substantial amount of compute making them infeasible in practical settings. *Two-stage NAS* [Yu et al., 2020] trains a *super-network* $\mathbf{w}_\Theta^\star = \arg\min \mathcal{L}_{train}(\mathbf{w}_\Theta)$ with a single set of weights $\mathbf{w}_\Theta \subset \mathbb{R}^d$ The super-network is defined such that it contains all possible networks $\theta \in \Theta$ in the search space. After training the super-network, we can compute the validation error $f(\theta) = \mathcal{L}_{valid}(\theta, \hat{\mathbf{w}}_\theta)$ of an architecture θ by a subset of the weights of the super-network $\hat{\mathbf{w}}_\theta \subseteq \mathbf{w}_\Theta$, reducing the overall compute by orders of magnitude. We refer to a network described by θ that uses a subset of the super-network weights as *sub-network*.

Super-network Training. To train a neural network, we iteratively update the current weight vector $\mathbf{w}_{t+1} = \mathbf{w}_t + \lambda\delta_t$, where λ represents the learning rate by the gradient $\delta_t = \nabla_{\mathbf{w}_t}\mathcal{L}_{train}$ of the



a) **b)**

Figure 2: Parameter count of sub-networks sampled from Llama 3.1-8B model. **a)** Uniform random sampling from Θ, which oversamples tiny models and **b)** Grid sampling, which samples more uniformly



a) **b)**

Figure 3: **c)** Calibration procedure based on architecture magnitude for subnetworks, and **d)** sub-network perplexities before and after applying importance sorting to the model.

training loss with respect to all weights. However, if we treat our network as a super-network, we need to ensure that sub-networks should still perform well in isolation. To avoid co-adaptation of sub-networks, the sandwich rule [Yu et al., 2019] uses the following weight update:

$$\delta_t = \nabla_{\mathbf{w}} \mathcal{L}_{train}(\mathbf{w}_\Theta) + \sum_{i=1}^{k} \nabla_{\mathbf{w}} \mathcal{L}_{train}(\mathbf{w}_{\theta_i}) + \nabla_{\mathbf{w}} \mathcal{L}_{train}(\mathbf{w}_{\theta_{min}}) \tag{1}$$

Here θ_{min} represents the smallest sub-network in the search space, and $\theta_i \sim \Theta$ is sampled uniformly at random from the search space. Intuitively, the sandwich rule ensures that the same amount of computation is spend to larger and smaller models in the search space.

4 Model Compression via Neural Architecture Search

Inspired by previous work [Klein et al., 2024] for smaller encoder-only networks, we treat the pre-trained LLM as super-network with weights $\mathbf{w}_\Theta$ and fine-tuned on some dataset using causal language modelling cross-entropy loss. Afterwards, we slice off sub-networks to balance between latency or parameter count and down-stream performance.

Following Sukthanker et al. [2024a], we factorize our search space Θ as $\Theta_d \times \Theta_H \times \Theta_r \times \Theta_L$, where $\Theta_d = [1, d_{model}]$ controls the embedding dimension, $\Theta_H = [1, H]$ the number of heads for all attention layers, the ratio r to compute the FFN hidden dimension, $\Theta_r = [1, r]$ and $\Theta_L = [0, L]$ the total number of transformer blocks. Here, H, r, d_{model}, and L represent the number of heads, the MLP ratio, the embedding dimension, and the number of layers of the pre-trained model, respectively.

4.1 Sampling Distribution

The sandwich rule (see Section 3.2) samples sub-networks uniformly at random from the search space in each update step. This results in a highly skewed distribution towards tiny model for the search space Θ described above, and we allocate too many updates to sub-networks that are not able to learn

effectively. Figure 2a illustrates this for a Llama-3.1-8B model. Reducing Θ is not straightforward, as it introduces bias into the search-space design process and presents an explore-exploit tradeoff. This becomes worse with larger models which contain much more sub-networks.

To better exploit our computational budget, we aim to allocate more update steps to promising sub-networks with uniformly distributed parameter counts. Let θ_{min} and θ_{max} represent the smallest and largest sub-network in Θ, and their corresponding parameter counts by $params_{min} = params(\theta_{min})$ and $params_{max} = params(\theta_{max})$. We discretize the range of parameters, and define a equally sized grid $\mathbb{G} = \{params_{min}, ..., params_{max}\}$ with K bins. We use a rejecting sampling based approach to create a discreet set of sub-networks. More specifically, for each bin $g_i \in \mathbb{G}$, we sample a set of of M sub-networks $\mathbb{S}_i = \{\theta_1, ...\theta_M\}$ such that for each $\theta \in \mathbb{S}_i$ we have $g_{i-1} \leq params(\theta) \leq g_i$. Now let $mag(\theta) = \|\mathbf{w}_\theta\|$ denote the weight magnitude of all weights of the sub-network described by θ. We select for each bin g_i the sub-networks with the highest weight magnitude $\hat{\theta}_i = \arg\min_{\theta \in \mathbb{S}_i} mag(\theta)$ from the set $\mathbb{S}_i$ and form our sampling grid $\mathbb{Q} = \{\hat{\theta}_0, ..., \hat{\theta}_K\}$. Now, in each update step of the sandwich rule, we sample $\theta_i \in \mathbb{Q}$ to compute Equation 1.

4.2 Importance scoring

If we select sub-networks from the super-network we always select the first few neorons or layers. For example, remember that $\Theta = \Theta_d \times \Theta_H \times \Theta_r \times \Theta_L$, for $\theta = [768, 8, 2, 9]$ we select the first $d_{model} = 768$ entries of the embedding vector, the first $H = 8$ heads of the multi-head attention layers, the first $U = d_{model} * 2$ units in the FFN layer and the first $L = 9$ layers. This convention ensures a bijective mapping from Θ to sub-networks [Klein et al., 2024], however it reduces the flexibility of selecting sub-networks.

Transformers are invariant to a fixed permutation of neurons throughout the network. Leveraging this, we rearrange the different components of the pre-trained super-network based on their importance score. Following [Muralidharan et al., 2024], we compute for each component the feature importance score on a small calibration dataset, for example Wikitext-2 test set. Figure 3b show how this simple permutation significantly reduces subnetwork perplexity, leading to better initialization for super-network training (Section 3.2).

4.3 Parameter Efficient Fine-Tuning for Two-stage NAS

Full-Fine-Tuning (FFT) of LLMs is often prohibitevly expensive, both in terms of compute costs and GPU memory consumption. For example we can go up to only a model size of $3B$ parameters with FFT on a single A100 GPU. To scale to larger models we adopt parameter-efficient-fine-tuning (PEFT) to fine-tune larger model sizes. Specifically we adopt LoNAS[Munoz et al., 2024], with two major changes. Firstly, we apply LoRA to all weight matrices in a transformer i.e., the prediction head, attention layers, FFN layers and the language model heads, in contrast to LoNAS which focuses only on attention and FFN layers. Secondly, since we also select the total number of blocks, we dynamically drop obsolete LoRA models. Compared to LoNAS, this allows us to also prune the embedding dimension and entire transformer blocks, to further reduce latency.

5 Experiments

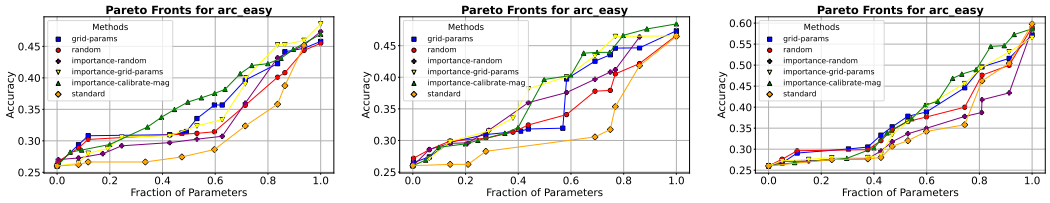
For the empirical evaluation of our model we consider Pythia-410M, Pythia-1B, and Pythia-2.8B from the *Pythia* family [Biderman et al., 2023], LLaMA-2-7B from the *LLaMA-2* family [Touvron et al., 2023], along with the newly introduced LLaMA-3.1-8B as well as Phi-2 and Phi-3 [Abdin et al., 2024]. We evaluate sub-networks on five commonsense reasoning tasks in a *zero-shot* manner using LM-Eval-Harness [Gao et al., 2024]. We fine-tune super-networks (see Section 4.2) on the Alpaca [Taori et al., 2023] dataset.

5.1 Comparison to Structural Pruning Baselines

We compare against the following structural pruning baselines on the Phi-2, Phi-3, Llama-2-7B and the Llama-3.1-8B model: **SliceGPT** [Ashkboos et al., 2024] is a post-training structural pruning technique based on Principal Component Analysis that reduces embedding dimensions by removing rows and columns from weight matrices. **N:M** [Zhou et al.] prunes N out of every M consecutive weights based on their magnitude. **SparseGPT** [Frantar and Alistarh, 2023b] prunes weights iteratively to a target sparsity by using a pre-computed Hessian to quantify the sensitivity of each neuron. **Wanda** [Sun et al., 2024] similar to SparseGPT but uses a simpler pruning metric based on weight magnitude and input activation norms.

Dataset	Metric	Magnitude			SparseGPT			Wanda			SliceGPT			Ours		
		Sparsity	Latency	Accuracy	Sparsity	Latency	Accuracy	Sparsity	Latency	Accuracy	Sparsity	Latency	Accuracy	Sparsity	Latency	Accuracy
phi-2																
Winogrande	Accuracy			62.43			68.03			68.11			69.38			**70.48**
ARC Challenge	Accuracy (norm)			35.41			34.55			37.372			40.44			**49.23**
MMLU	Accuracy	2:4	243.357	23.64	50% (2:4)	269.99	28.12	2:4	254.45	25.402	15.00%	228.80	26.03	15.90%	216.81	**33.45**
Hellaswag	Accuracy (norm)			54.07			54.74			50.77			62.44			**65.11**
TruthfulQA	mc2			43.54			43.211			41.47			**42.89**			42.04
phi-3																
Winogrande	Accuracy			54.14			62.90			63.61			65.900			**69.29**
ARC Challenge	Accuracy (norm)			32.93			33.36			35.83			47.01			**47.70**
MMLU	Accuracy	2:4	298.71	28.01	2:4	339.53	31.93	2:4	335.73	33.91	10.00%	319.83	49.68	10.12%	398.0301	**52.21**
Hellaswag	Accuracy (norm)			35.95			53.11			49.81			68.03			**70.72**
TruthfulQA	mc2			51.68			46.64			46.24			50.02			**54.88**
Llama-2-7B																
Winogrande	Accuracy			58.64			64.01			62.27			**65.82**			65.51
ARC Challenge	Accuracy (norm)			29.35			30.29			31.1433			38.40			**42.15**
MMLU	Accuracy	2:4	459.21	23.89	2:4	530.32	25.55	2:4	529.82	25.14	10.00%	508.12	30.94	10.44 %	496.63	**34.82**
Hellaswag	Accuracy (norm)			52.73			57.38			55.51			68.33			**73.47**
TruthfulQA	mc2			47.94			37.18			38.73			37.82			**45.29**
Llama-3.1-8B																
Winogrande	Accuracy			53.75			63.77			61.09			65.98			**66.61**
ARC Challenge	Accuracy (norm)			31.23			29.35			29.44			39.08			**45.563**
MMLU	Accuracy	2:4	493.54	23.73	2:4	596.23	27.04	2:4	592.91	27.02	10%	502.76	43.15	10.36%	409.88	**50.750**
Hellaswag	Accuracy (norm)			39.29			52.18			48.516			65.58			**74.45**
TruthfulQA	mc2			46.61			43.36			41.46			40.65			**54.15**

Table 1: Zero-Shot Evaluation on Downstream Tasks. We compute all latencies on a single A100 GPU with batch size of 2 and 512 as block size N. Note for N:M semi-structured baselines we use 2:4 sparsity as only that leads to on-device speedups in practice. Latencies reported are in ms.



a) Pythia-410m **b)** Pythia-1b **c)** Pythia-2.8b

Figure 4: Comparison of different sampling schemes on Pythia-410m, Pythia-1b and Pythia-2.8b

Table 1 shows the comparison of pruning baselines to our NAS method. We use the 2:4 sparsity of semi-structured baselines to yield on-device speedup. Using NAS we obtain sub-networks that outperform or perform competitively to these baselines across a wide variety of datasets. Moreover, the sub-networks found by NAS achieve lower latency (see Figure 1 for a comparison on Llama 3.1).

5.2 Ablation of Sampling Scemes

We perform an ablation study of our sampling strategy by comparing against the following alternatives: **Standard** fine-tuning which updates all weights of the network. **Random** samples a single sub-network uniformly at random in each update step. **Grid-params** uses the binning strategy as described in Section 4 but samples a random sub-network from each bin. **Importance-random**, **Importance-grid-params** and **Importance-calibrate-mag** combines importance scoring to reorder the sub-networks with random sampling, our binning strategy either selecting sub-network per bin at random or based on their weight magnitude, respectively.

Figure 4 shows results for Pythia 410m, Pythia 1b and Pythia2.8b using full fine-tuning instead of PEFT. *Standard* fine-tuning leads to a sharp drop in accuracy with increasing sparsity levels. On the other hand, super-network, and more specifically the importance-calibrated-mag scheme makes the network more amenable to pruning leading to better performing sub-networks.

6 Discussion and Conclusion

We investigate two-stage NAS for model compression and scale it up to LLMs. We adapt the sampling procedure during super-network training to allocate more update steps to promising sub-networks, in an equi-spaced parameter grid. Given the implicit constraint given by the search space to always select the first components, we propose to re-shuffle entries based on their importance score.

Compared to classical structural pruning approaches, NAS provides a Pareto set of sub-networks to allow practitioners to select the optimal sub-network matching their constraints. However, compared to some structural pruning methods, NAS requires an additional fine-tuning step to account for training the super-network.

In future work, we plan to extend our setting to dynamic benchmarks. Also we plan to incorporate techniques from the distillation literature to improve sub-network performance.

Acknowledgments

This research was partially supported by the following sources: TAILOR, a project funded by EU Horizon 2020 research and innovation programme under GA No 952215; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under grant number 417962828; the European Research Council (ERC) Consolidator Grant "Deep Learning 2.0" (grant no. 101045765). Robert Bosch GmbH is acknowledged for financial support. The authors acknowledge support from ELLIS and ELIZA. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the ERC. Neither the European Union nor the ERC can be held responsible for them. Aaron Klein acknowledges the financial support by the Federal Ministry of Education and Research of Germany and by Sächsische Staatsministerium für Wissenschaft, Kultur und Tourismus in the programme Center of Excellence for AI-research „Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig", project identification number: ScaDS.AI. Frank Hutter acknowledges the financial support of the Hector Foundation.

.

References

Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL `https://arxiv.org/abs/2404.14219`.

S. Ashkboos, M. L. Croci, M. Gennari do Nascimento, T. Hoefler, and J. Hensman. SliceGPT: Compress large language models by deleting rows and columns. In *The Twelfth International Conference on Learning Representations*, 2024.

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv:1607.06450 [stat.ML]*, 2016.

G. Bender, P. Kindermans, B. Zoph, V. Vasudevan, and Q. Le. Understanding and simplifying one-shot architecture search. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*, 2018.

S. Biderman, H. Schoelkopf, Q. G. Anthony, H. Bradley, K. O'Brien, E. Hallahan, M. A. Khan, S. Purohit, S. Prashanth, E. Raff, et al. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning (ICML'23)*, 2023.

H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations (ICLR'20)*, 2020.

R. Cai, S. Muralidharan, G. Heinrich, H. Yin, Z. Wang, J. Kautz, and P. Molchanov. Flextron: Many-in-one flexible large language model. In *International Conference on Machine Learning (ICML'24)*.

R. Cai, S. Muralidharan, G. Heinrich, H. Yin, Z. Wang, J. Kautz, and P. Molchanov. Flextron: Many-in-one flexible large language model. *arXiv:2406.10260 [cs.CL]*, 2024.

T. Elsken, J. H. Metzen, and F. Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *International Conference on Learning Representations (ICLR'19)*, 2019a.

T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 2019b.

William Fedus, Jeff Dean, and Barret Zoph. A review of sparse expert models in deep learning. *arXiv preprint arXiv:2209.01667*, 2022.

E. Frantar and D. Alistarh. SparseGPT: Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*, 2023a.

Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International Conference on Machine Learning*, pages 10323–10337. PMLR, 2023b.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 07 2024. URL `https://zenodo.org/records/12608602`.

Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pages 544–560. Springer, 2020.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015a.

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. *Advances in neural information processing systems*, 28, 2015b.

G. Hinton, O. Vinyals, and J. Dean. Distilling the knowledge in a neural network. *arXiv:1503.02531 [stat.ML]*, 2015.

E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. Lora: Low-rank adaptation of large language models. *arXiv:2106.09685 [cs.CL]*, 2021.

A. Klein, J. Golebiowski, X. Ma, V. Perrone, and C. Archambeau. Structural pruning of pre-trained language models via neural architecture search. *Transactions on Machine Learning Research*, 2024.

Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. Shortgpt: Layers in large language models are more redundant than you expect. *arXiv preprint arXiv:2403.03853*, 2024.

P. Michel, O. Levy, and G. Neubig. Are sixteen heads really better than one? In *Proceedings of the 32th International Conference on Advances in Neural Information Processing Systems (NeurIPS'19)*, 2019.

Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. Accelerating sparse deep neural networks. *arXiv preprint arXiv:2104.08378*, 2021.

J Pablo Muñoz, Jinjie Yuan, and Nilesh Jain. Shears: Unstructured sparsity with neural low-rank adapter search. *arXiv preprint arXiv:2404.10934*, 2024.

Juan Pablo Munoz, Jinjie Yuan, Yi Zheng, and Nilesh Jain. Lonas: Elastic low-rank adapters for efficient large language models. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 10760–10776, 2024.

Saurav Muralidharan, Sharath Turuvekere Sreenivas, Raviraj Joshi, Marcin Chochowski, Mostofa Patwary, Mohammad Shoeybi, Bryan Catanzaro, Jan Kautz, and Pavlo Molchanov. Compact language models via pruning and knowledge distillation. *arXiv preprint arXiv:2407.14679*, 2024.

Ofir Press and Lior Wolf. Using the output embedding to improve language models. *arXiv preprint arXiv:1608.05859*, 2016.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019.

E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the Conference on Artificial Intelligence (AAAI'19)*, 2019.

Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. On the effect of dropping layers of pre-trained transformer models. *Computer Speech & Language*, 77:101429, 2023.

Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. *Advances in neural information processing systems*, 29, 2016.

Sharath Nittur Sridhar, Souvik Kundu, Sairam Sundaresan, Maciej Szankin, and Anthony Sarah. Instatune: Instantaneous neural architecture search during fine-tuning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1523–1527, 2023.

R. S. Sukthanker, A. Zela, B. Staffler, A. Klein, L. Purucker, J. K. H. Franke, and F. Hutter. Hw-gpt-bench: Hardware-aware architecture benchmark for language models. *arXiv:2405.10299 [cs.LG]*, 2024a.

Rhea Sanjay Sukthanker, Arber Zela, Benedikt Staffler, Samuel Dooley, Josif Grabocka, and Frank Hutter. Multi-objective differentiable neural architecture search. *arXiv:2402.18213 [cs.LG]*, 2024b. URL `https://arxiv.org/abs/2402.18213`.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. A simple and effective pruning approach for large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=PxoFut3dWW`.

Chen Tang, Li Lyna Zhang, Huiqiang Jiang, Jiahang Xu, Ting Cao, Quanlu Zhang, Yuqing Yang, Zhi Wang, and Mao Yang. Elasticvit: Conflict-aware supernet training for deploying fast vision transformer on diverse mobile devices. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5829–5840, 2023.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. Alpaca: A strong, replicable instruction-following model. *Stanford Center for Research on Foundation Models. https://crfm. stanford. edu/2023/03/13/alpaca. html*, 3(6):7, 2023.

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

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS'17)*, 2017.

Dilin Wang, Meng Li, Chengyue Gong, and Vikas Chandra. Attentivenas: Improving neural architecture search via attentive sampling. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6418–6427, 2021.

H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han. Hat: Hardware-aware transformers for efficient natural language processing. In *Annual Meeting of the Association for Computational Linguistics*, 2020.

C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter. Neural architecture search: Insights from 1000 papers. *arXiv:2301.08727 [cs.LG]*, 2023.

Jin Xu, Xu Tan, Kaitao Song, Renqian Luo, Yichong Leng, Tao Qin, Tie-Yan Liu, and Jian Li. Analyzing and mitigating interference in neural architecture search. In *International Conference on Machine Learning*, pages 24646–24662. PMLR, 2022.

Yifei Yang, Zouying Cao, and Hai Zhao. Laco: Large language model pruning via layer collapse. *arXiv preprint arXiv:2402.11187*, 2024.

J. Yu, L. Yang, N. Xu, J. Yang, and T. Huang. Slimmable neural networks. In *International Conference on Learning Representations (ICLR'19)*, 2019.

J. Yu, P. Jin, H. Liu, G. Bender, P. J. Kindermans, M. Tan, T. Huang, X. Song, R. Pang, and Q. Le. BigNAS: Scaling Up Neural Architecture Search with Big Single-Stage Models. In *The European Conference on Computer Vision (ECCV'20)*, 2020.

Jiahui Yu and Thomas S Huang. Universally slimmable networks and improved training techniques. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1803–1811, 2019.

B. Zhang and R. Sennrich. Root mean square layer normalization. In *Proceedings of the 32th International Conference on Advances in Neural Information Processing Systems (NeurIPS'19)*, 2019.

Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n: M fine-grained structured sparse neural networks from scratch.

Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*, 2023.

B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR'17)*, 2017.

A Extended Related Work

Model compression reduces the size or computational complexity of a neural network model while minimizing loss in performance. It involves techniques such as quantization (reducing the precision of weights and activations), pruning (removing neurons or connections from a model), knowledge distillation (transferring knowledge from a large to a small network e.g. in the form of representations or outputs), low-rank factorization, efficient model design and NAS (see e.g. Zhu et al. [2023] for an overview). In the following, we focus on pruning and NAS.

Pruning removes weights and connections of a network to reduce the number of parameters and accelerate inference. Unstructured pruning removes arbitrary weights while structural pruning considers entire groups of parameters such as attention heads Michel et al. [2019] or layers [Sajjad et al., 2023] for removal which is better suited for model acceleration on hardware optimized for dense computation [Mishra et al., 2021]. Pruning and in particular structured pruning approaches can result in a loss of accuracy and most pruning methods include a retraining phase to recover as much accuracy as possible. Recent work focused on pruning LLMs to tackle the particular challenges that come with their large number of parameters, the high computational complexity, and the often limited availability of data for retraining. Methods such as ShortGPT [Men et al., 2024] and LaCo [Yang et al., 2024] use importance scores to prune or merge layers of LLMs. SparseGPT [Frantar and Alistarh, 2023a] approximates the optimal weights in a pruning mask using a row-wise iterative update scheme to do unstructured and semi-structure pruning of generative pretrained transformers. Wanda [Sun et al., 2024] extends magnitude pruning [Han et al., 2015b] by including the activation values on a small calibration set to do unstructured and N:M structured pruning. Flextron [Cai et al., 2024] propose a procedure that allows to extract models for different deployment scenarios by first making by combining an elastic model (cf. Cai et al. [2020]) with methods from mixture of experts (see e.g. Fedus et al. [2022] for a recent review). The router networks can take static information such as a target latency into account but also input-adaptive routing. Probably the closest work to our approach is Minitron [Muralidharan et al., 2024], which uses activation-based importance scores to prune models and knowledge distillation from an uncompressed teacher for retraining. However, they need to considerably reduce the number of architectures that are compared to reduce training time. In our work, we consider much larger search spaces and leverage one-shot NAS for efficient training including knowledge distillation and incorporate importance scores during the architecture sampling procedure. This allows us to combine everything into a single one-step training procedure and calculate the full Pareto front instead of single architectures.

Neural Architecture Search automates the design of deep neural networks in a data-driven manner (see e.g. Elsken et al. [2019b], White et al. [2023] for an overview). NAS has been extended to a multi-optimization problem taking also efficiency on a target hardware platform into account such as latency or energy consumption [Elsken et al., 2019a, Cai et al., 2020, Wang et al., 2020] making it closely related to model compression. To tackle the enormous computational cost of early NAS methods [Zoph and Le, 2017, Real et al., 2019], weight-sharing based NAS [Saxena and Verbeek, 2016, Bender et al., 2018] trains a single super-network from which the weights can be inherited to all architectures in a search space for performance evaluation without further training. A particularly prominent approach to use NAS for model compression is two-stage NAS, which has a dedicated super-network training and multi-objective search stage [Bender et al., 2018, Guo et al., 2020, Cai et al., 2020]. Most two-stage methods use the notation of elastic layers [Cai et al., 2020] that can dynamically adjust its size (e.g. width) during training. The training of the supermodel is typically done as proposed in Yu and Huang [2019] using the sandwich rule, which aggregates the gradients of multiple sub-networks from the super-network, as well as in-place distillation, which uses the outputs of the largest network in a super-network as targets for smaller ones. Furthermore, Tang et al. [2023] and Wang et al. [2021] proposed different strategies how to sample models from supermodel to better cover the Pareto front. A detailed study of NAS for structural pruning has been conducted in Klein et al. [2024] that shows that NAS is a competitive technique to other pruning approaches and highlights in particular the increased flexibility and automation potential of NAS methods that allow to estimate the full Pareto front [Cai et al., 2020, Sukthanker et al., 2024b] instead of having to set a single threshold for pruning. However, even two-stage NAS methods still have a large computational overhead compared to regular model training. To further reduce the computational complexity of NAS, several works proposes to leverage pretrained weights as well as parameter efficient fine-tuning methods. InstaTune [Sridhar et al., 2023] uses a pretrained model to initialize and train a super-network on a fine-tuning task. LoNAS [Munoz et al., 2024] freezes the weights

of a pretrained backbone and introduces elastic LoRA adapters [Hu et al., 2021]. Similarly, Shears [Muñoz et al., 2024] combines unstructured pruning of a pretrained model with NAS to search for elastic LoRA adapters to mitigate the performance loss of pruning.

B Experimental details

In this section we provide details on the hyperparameters, search spaces and datasets used for fine-tuning different pretrained models studied in the paper. Our hyperparameter settings largely follow litgpt configs [1].

B.1 Fine-tuning Dataset

We fine-tune all models with the Alpaca[Taori et al., 2023] dataset, which contains 51,759 instruction pairs. Alpaca dataset is generated instruction-following demonstrations by building upon the self-instruct method, which uses an existing strong language model to automatically generate instruction data.

B.2 Pythia

Hyperparameters. We use a learning rate of 2e-5 for all pythia models and sampling schemes. Furthermore we use a batch size of 4 for pythia-410m and pythia-1b and a batch size of 1 for full fine-tuning of pythia-2.8b. We use the AdamW optimizer with 300 warmup steps and cosine annealing of the learning rate for all models. We fine-tune all the models for 5 epochs.

Search Space. We define range of choices for search space of pythia models in Table 2, where model_embedding_dim corresponds to the embedding dimension of the pretrained model, the model_number_of_heads corresponds to the number of heads in the pretrained model and model_num_layers corresponds to number of layers in the pretrained model. All other architecture dimensions are same as the respective pretrained models.

Table 2: Configuration Search Space Parameters for Pythia Models

Parameter	**Type**	**Choice Range [max, min]**
embed_dim	Logarithmic	[1, model_embedding_dim]
num_heads	Random Integer	[1, model_number_of_heads]
mlp_expansion_ratio	Random Integer	[1, 4]
depth	Random Integer	[1, model_num_layers]

B.3 Llama

Hyperparameters. We use a learning rate of 0.0002 for all llama models and sampling schemes. Furthermore we use a batch size of 8 for Llama-3.1-8B and 16 for Llama-2-7B and use the AdamW optimizer with 10 warmup steps with cosine annealing of the learning rate. We fine-tune all the models for 5 epochs. We set the LoRA rank to 32, the LoRA alpha to 16 and the dropout to 0.05. Furthermore, we add LoRA layers to all attention layers, mlp layers and the prediction head.

Search Space. We define range of choices for search space of Llama-3.1-8B models in Table 3 and for Llama-2-7B in Table 4, where model_embedding_dim corresponds to the embedding dimension of the pretrained model, and model_num_layers corresponds to number of layers in the pretrained model. All other architecture dimensions (including number of heads) are same as the respective pretrained models.

[1] litgpt config hub

Table 3: Configuration Search Space Parameters for Llama-3.1-8B

Parameter	Type	Choice Range [max, min]
embed_dim	Logarithmic Integer	$[1, \text{model\_embedding\_dim}]$
mlp_expansion_ratio	Random Choice	$\{1.0, 2.0, 3.0, 3.5\}$
depth	Random Integer	$[1, \text{model\_num\_layers}]$

Table 4: Configuration Search Space Parameters for Llama-2-7B

Parameter	Type	Choice Range [max, min]
embed_dim	Logarithmic Integer	$[1, \text{model\_embedding\_dim}]$
mlp_expansion_ratio	Random Choice	$\{1.0, 2.0, 2.5, 2.6875\}$
depth	Random Integer	$[1, \text{model\_num\_layers}]$

B.4 Phi

Hyperparameters. We use a learning rate of 0.0002 for all phi models and sampling schemes. Furthermore we use a batch size of 32 and use the AdamW optimizer with 10 warmup steps with cosine annealing of the learning rate. We fine-tune all the models for 1 epoch. We set the LoRA rank to 8, the LoRA alpha to 16 and the dropout to 0.05. Furthermore, we add LoRA layers to all attention layers, mlp layers and the prediction head.

B.5 Search Space.

We define the search space for Phi-2 in a manner similar to Pythia models, Table 2 and define the search space for Phi-3 in Table 5, where model_embedding_dim corresponds to the embedding dimension of the pretrained model, and model_num_layers corresponds to number of layers in the pretrained model. All other architecture dimensions (including number of heads for Phi-3) are same as the respective pretrained models.

Table 5: Configuration Search Space Parameters for Phi-3

Parameter	Type	Choice Range [max, min]
embed_dim	Logarithmic	$[1, \text{model\_embedding\_dim}]$
mlp_expansion_ratio	Random Choice	$\{1.0, 2.0, 2.5, 2.6666666666666665\}$
depth	Random Integer	$[1, \text{model\_num\_layers}]$

B.6 Grid Sampling Scheme

We use uniform size of 22 for the architecture grid for **Grid-params**, **Importance-grid-params** and **Importance-calibrate-mag** schemes as defined in 5.2. Furthermore when doing rejection sampling based on parameter count to obtain architectures in different parameter bins, we allow for at most 1000 architecture sampling trials.