LiteTransformerSearch: Training-free On-device Search for Efficient Autoregressive Language Models

Mojan Javaheripi[1] **Gustavo Henrique de Rosa**[2] **Subhabrata Mukherjee**[2] **Shital Shah**[2]
Tomasz Lukasz Religa[3] **Caio Cesar Teodoro Mendes**[2] **Sebastien Bubeck**[2]
Farinaz Koushanfar[1] **Debadeepta Dey**[2]

[1]University of California San Diego, [2]Microsoft Research, [3]Microsoft

Abstract The Transformer architecture is ubiquitously used as the building block of large-scale autoregressive language models. However, finding architectures with the optimal trade-off between task performance (perplexity) and hardware constraints like peak memory utilization and latency is non-trivial. We leverage the somewhat surprising empirical observation that the number of non-embedding parameters in autoregressive Transformers has a high rank correlation with task performance, irrespective of the architecture topology. This observation organically induces a simple search algorithm that can be directly run on target devices. We rigorously show that the pareto-frontier of perplexity versus different hardware costs such as latency and memory can be found without need for any model training, using non-embedding parameters as a proxy for perplexity. We evaluate our method, dubbed Lightweight Transformer Search (LTS) on diverse devices from ARM CPUs to NVIDIA GPUs and two popular autoregressive Transformer backbones: GPT-2 and Transformer-XL. Results show that the perplexity of 16-layer GPT-2 and Transformer-XL can be achieved with up to 1.6×, 2.5× faster runtime and 1.3×, 2× lower peak memory utilization. LTS extracts the pareto-frontier in under 3 hours, running on a commodity laptop. We effectively remove the carbon footprint of training during search for hundreds of GPU hours, offering a strong simple baseline for future NAS methods in autoregressive language modeling.

1 Introduction

The Transformer architecture [31] is the de-facto building block of most pre-trained language models like GPT [3]. A critical problem arises when creating smaller Transformer models with strict memory and latency constraints: the architectural hyperparameters, e.g., number of layers and number of attention heads, are not known. This problem is exacerbated if each individual Transformer layer is allowed to have different values for these settings. This results in a combinatorial explosion of architectural hyperparameter choices and a large heterogeneous search space. For instance, the search space considered in this paper consists of over 10^{54} possible architectures.

Neural Architecture Search (NAS) is an organic solution due to its ability to automatically search through candidate models with multiple conflicting objectives like latency vs. task performance. The central challenge in NAS is prohibitively expensive function evaluation, i.e., evaluating each architecture requires training it on the dataset at hand. Thus it is often infeasible to evaluate more than a handful of architectures during the search phase. Supernets [23] have emerged as a dominant paradigm in NAS which combine all possible architectures into a single graph and jointly train them using weight-sharing. Nevertheless, supernet training imposes constraints on the expressiveness of the search space [21] and is often memory-hungry [4, 38, 40] as it creates large networks during search. Additionally, training supernets is non-trivial as children architectures may interfere with each other and rank correlation between sub-architectures is not preserved [21][1].

We propose a training-free proxy that provides a highly accurate ranking of candidate architectures during NAS without need for costly training or supernets. Our scope is NAS for efficient Transformer-based autoregressive language models[2]. We design a lightweight search that is target hardware-aware and outputs a gallery of models on the pareto-frontier of perplexity versus

[1]See [21] for a comprehensive treatment of the difficulties of training supernets.
[2]For preliminaries on autoregressive Transformers, please see Appendix A.

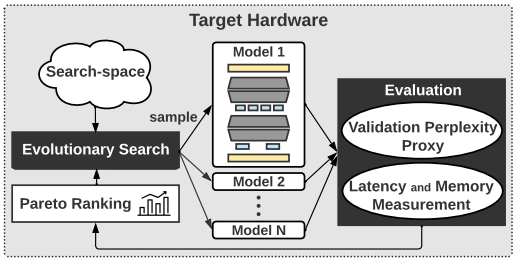


Figure 1: High-level overview of LTS. We propose a training-free proxy for evaluating the validation perplexity of candidate architectures. Pareto-frontier search is powered by evolutionary algorithms which use the proposed proxy along with real latency and memory measurements on the target hardware to evaluate sampled architectures.

hardware metrics. We term this method Lightweight Transformer Search (LTS). LTS relies on our surprising observation: *the number of non-embedding parameters, i.e., parameters enclosed in decoder layers, has a high rank correlation with the perplexity of fully trained autoregressive Transformers.*

Given a set of Transformer-based autoregressive language models, one can accurately rank them using non-embedding parameter count as the proxy for perplexity. Our observations are also well-aligned with the power laws in [15], shown for homogeneous autoregressive Transformers, i.e., when all decoder layers have the same configuration. We provide rigorous experiments to empirically establish the high rank correlation between perplexity and non-embedding parameter count for *both* homogeneous and heterogeneous search spaces.

The above phenomenon coupled with the fact that a candidate architecture's hardware performance can be measured on the target device leads to a training-free search procedure: *pick one's favorite discrete search algorithm (e.g. evolutionary search), sample candidate architectures from the search space; count their decoder parameters as a proxy for task performance (i.e., perplexity); measure their hardware performance (e.g., latency and memory) directly on the target device; and progressively create a pareto-frontier estimate.*

Building upon these insights, Figure 1 shows a high-level overview of LTS. We design the first training-free Transformer search that is performed entirely on the target (constrained) platform. As such, LTS easily performs a multi-objective NAS where several underlying hardware performance metrics, e.g., latency and peak memory utilization, are simultaneously optimized. Using our training-free proxy, we extract the 3-dimensional pareto-frontier of perplexity versus latency and memory in a record breaking time of < 3 hours on a commodity Intel Core i7 CPU. Notably, LTS eliminates the carbon foot print from hundreds of GPU hours of training associated with legacy NAS methods.

To evaluate our proxy, we train over 2800 Transformer architectures on two large language modeling datasets, i.e., WikiText-103 [20] and One Billion Word [5]. We use LTS to search for pareto-optimal architectural hyperparameters in two popularly used autoregressive Transformer backbones, i.e., Transformer-XL [6] and GPT-2 [24]. We believe decoder parameter count provides a competitive baseline for Transformer NAS in terms of ranking capabilities and easy computation.

2 Related Work

To the best of our knowledge, there exist only one prior work that focuses on NAS for autoregressive (decoder-only) transformer architectures. So et al. [26] searches over TensorFlow programs that implement an autoregressive language model via evolutionary search. Since most random sequences of programs either have errors or underperform, the search has to be seeded with the regular transformer architecture, termed "Primer". As opposed to "Primer" which uses large computation to search a much more general space, our aim is to efficiently search the "backbone" of traditional decoder-only transformers. Additionally, the objective in "Primer" is to finds models that train faster. Our objective for NAS, however, is to deliver pareto-frontier models for the inference phase with respect to task metrics and hardware constraints, e.g., latency.

We provide further literature review of NAS for other Transformer architectures and whose target applications are language domains in Appendix B. We refer the eager reader to extensive surveys on NAS [9, 37] for a more broad overview of the field.

3 Lightweight Transformer Search

We perform an evolutionary search over candidate architectures to extract models that lie on the pareto-frontier, which spans a wide range of latency, peak memory utilization, and perplexity char-

acteristics. To evaluate candidate models during the search, LTS uses a training-free proxy for the validation perplexity. By incorporating training-free evaluation metrics, LTS, for the first time, performs the entire search directly on the target (constrained) hardware. Therefore, we can use real measurements of hardware performance during the search. Algorithm 1 outlines the iterative process performed in LTS for finding candidate architectures in the search space ($\mathcal{D}$)[3], that lie on the 3-dimensional pareto-frontier ($\mathbb{F}$) of perplexity versus latency and memory. At each iteration, a set of points ($\mathbb{F}'$) are subsampled from the current pareto-frontier. A new batch of architectures ($\mathbb{S}_N$) are then sampled from $\mathbb{F}'$ using evolutionary algorithms ($EA(.)$)[4]. The new samples are evaluated by measuring the latency ($\mathcal{L}$) and peak memory utilization ($\mathcal{M}$) directly on the target hardware and estimating the validation perplexity ($\mathcal{P}$) using our accurate and training-free proxy. Finally, the pareto-frontier is recalibrated using the lower convex hull of all sampled architectures. In the context of multi-objective NAS, pareto-frontier points are those where no single metric (e.g., perplexity, latency, and memory) can be improved without degrading another metric [13].

Algorithm 1: LTS's training-free NAS

Input: Search space $\mathcal{D}$, n_{iter}
Output: Perplexity-latency-memory pareto-frontier $\mathbb{F}$

$\mathcal{L}, \mathcal{M}, \mathcal{P}, \mathbb{F} \leftarrow \emptyset, \emptyset, \emptyset, \emptyset$
while $N \leq n_{iter}$ **do**
  $\mathbb{F}' \leftarrow$ Subsample($\mathbb{F}$)
  $\mathbb{S}_N \leftarrow EA(\mathbb{F}', \mathcal{D})$
  `// hardware profiling`
  $\mathcal{L} \leftarrow \mathcal{L} \bigcup$ Latency($\mathbb{S}_N$)
  $\mathcal{M} \leftarrow \mathcal{M} \bigcup$ Memory($\mathbb{S}_N$)
  `// estimate perplexity`
  $\mathcal{P} \leftarrow \mathcal{P} \bigcup$ Proxy($\mathbb{S}_N$)
  `// update the pareto front`
  $\mathbb{F} \leftarrow$ LowerConvexHull($\mathcal{P}, \mathcal{L}, \mathcal{M}$)

3.1 Training-free Architecture Ranking

► **Low-cost Ranking Proxies**. Recently, Abdelfattah et al. [1] utilize the summation of pruning scores over model weights as the ranking proxy for Convolutional Neural Networks (CNNs), where a higher score corresponds to a higher rank. White et al. [36] find that no particular low-cost (pruning-based) proxy performs consistently well over various tasks, while number of parameters is a quite competitive proxy. However they did not include Transformer-based search spaces in their analysis. Note that one cannot naively apply these proxies to language models. Specifically, since the embedding layer in Transformers is equivalent to a lookup operation, special care must be taken to omit this layer from the proxy computation. Using this insight, we perform the first systematic study of low-cost proxies for NAS on autoregressive Transformers in language domain.

We use various pruning metrics, namely, `grad_norm` , `snip` [16], `grasp` [33], `fisher` [29], and `synflow` [27]. We also study `jacob_cov` [19] and `relu_log_det` [18] which are low-cost scoring mechanisms proposed for NAS on CNNs in vision tasks. While these low-cost techniques do not perform model training, they require forward and backward passes over the architecture to compute the proxy, which can be time consuming on low-end hardware. Additionally, these techniques, by definition, include the final softmax projection in their score assessment which can skew the evaluation for autoregressive Transformers with a large output vocabulary.

► **Decoder Parameter Count as a Proxy**. We empirically establish a strong correlation between the parameter count inside decoder layers and final model performance in terms of validation perplexity. We evaluate 200 architectures sampled uniformly at random from the search space of two autoregressive Transformer backbones, namely, Transformer-XL

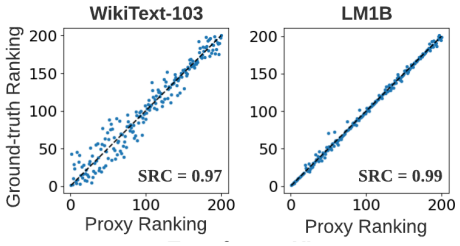


Figure 2: Decoder parameter count proxy is highly correlated with the (ground-truth) perplexity after full training.

[3]Details of our search-space are included in Appendix C.

[4]While we have chosen a reasonable search algorithm inspired by [8, 14], one can plug-and-play any pareto-frontier search method such as those in [13].

and GPT-2. Full training of these architectures consumes over **25000** GPU-hours on NVIDIA A100 and V100 nodes. We compare the ranking obtained using our zero-cost proxy against the ground-truth ranking based on validation perplexity after full training in Figure 2 and Figure 7 in the Appendix. On WikiText-103, ranking using the decoder parameter count obtains a Spearman's Rank Correlation (SRC) of 0.97 and 0.98 with full training for Transformer-XL and GPT-2 backbones, respectively. SRC increases to 0.99 for the more complex LM1B dataset. As shown, decoder parameter count is strongly correlated with model perplexity, providing a reliable proxy for NAS.

4 Experiments

Details of our experimental setup is included in Appendix E. Our experiments seek answers to the following critical questions: ❶ How well can training-free proxies perform compared to training-based methods for estimating the performance of Transformers (Section 4.1)? ❷ How does model topology affect the decoder parameter count proxy (Appendix I)? ❸ Which models are on the pareto-frontier of perplexity, latency, and memory for different hardware (Section 4.2)? ❹ Can our training-free decoder parameter count proxy be integrated inside a search algorithm to estimate the pareto-frontier? How accurate is such an estimation of the pareto (Appendix H)?

4.1 How do training-free proxies perform compared to training-based methods?

We benchmark several proxy methods for estimating the rank of candidate architectures.

► **Partial Training.** We stop the training after $\tau \in [1.25\%, 87.5\%]$ of the total iterations needed for convergence and analyze the relationship between the obtained perplexity versus that of full training. As shown in Figure 8 in the Appendix, more training iterations result in a more accurate estimate of the final perplexity. Nevertheless, the increased wall-clock time prohibits training during search and also imposes the need for GPUs. Interestingly, very few training iterations (1.25%) provides a very good proxy for final perplexity with an SRC of > 0.9. Our training-free proxy, i.e., decoder parameter count, also shows competitive SRC compared to partial training.

► **Low-cost Proxies.** We benchmark various low-cost methods introduced in Section 3.1 on randomly sampled architectures from the Transformer-XL backbone. Figure 3 shows the SRC between low-cost proxies and the ground-truth ranking after full training on WikiText-103. We measure the cost of each proxy in terms of floating point operations (FLOPs). The evaluated low-cost proxies have a strong correlation with the ground-truth ranking (even the lowest performing `relu_log_det` has > 0.8 SRC)[5]. Our analysis on randomly selected models with homogeneous decoder layers also shows a strong correlation between the low-cost proxies and perplexity, with decoder parameter count outperforming other proxies (see Appendix G).

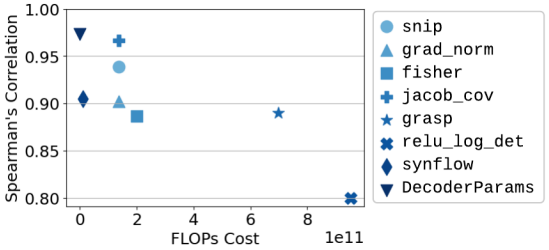


Figure 3: SRC between low-cost proxies and the ground-truth ranking after full training of 200 randomly sampled Transformers. The decoder parameter count obtains the best SRC with zero cost.

► **Parameter Count.** As shown in Figure 3 and Figure 8 in the Appendix, our zero-cost proxy outperforms all higher cost proxies. We thus propose using the decoder parameter count as the ranking proxy for the search. We provide a more detailed breakdown of the performance of this proxy in Appendices G and F.

4.2 Pareto-frontier Models for Various Hardware Platforms

We benchmark the Transformer-XL (base) and GPT-2 (small) models with homogeneous layers $\in [1, 16]$ as our baseline and show that LTS finds models with better validation perplexity and/or lower latency and peak memory utilization (better pareto). All models are trained using the same setup[6] in Appendix E for fair comparison. We compare the pareto-frontier architectures found by

[5]The lower performance of `relu_log_det` can be attributed to the fundamental difference in the frequency of ReLU activations in CNNs, where the method was originally developed, compared to Transformers.

[6]The best reported result in the literature for GPT-2 or Transformer-XL might be different based on the specific training hyperparameters, which is orthogonal to our investigation.

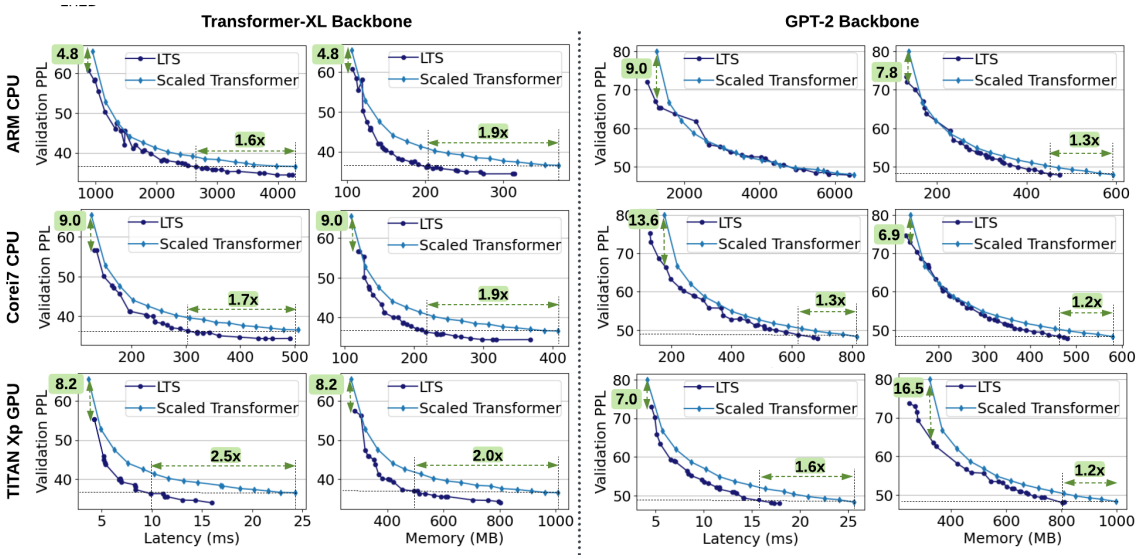


Figure 4: 2D visualization of the perplexity versus latency and memory pareto-frontier found by LTS, versus the scaled backbone models with varying number of layers, trained on the LM1B dataset. Architectural parameters for models shown here are detailed in Appendix L.

LTS with the baseline in Figure 4. Here, all models are trained on the LM1B dataset (See Figure 16 in Appendix K for results on WikiText-103). Note that the pareto-frontier search is performed in a 3-dimensional space (see Appendix J) but here we plot 2-dimensional slices of the pareto-frontier.

As seen, in the low-latency regime, LTS consistently finds models that have significantly lower perplexity compared to naive scaling of the baseline Transformer-XL or GPT-2. On the Transformer-XL backbone, LTS finds models with an average of 19.8% (resp. 25.8%) and 28.8% (resp. 30.0%) lower latency and memory and similar perplexity compared to the baseline on ARM CPU (resp. Corei7 CPU). The savings are even higher on the GPU device, where the NAS-generated models achieve the same perplexity as the baseline with average 30.5% lower latency and 27.0% less memory. Specifically, an LTS model with the same perplexity as the 16-layer Transformer-XL base has 2.5× lower latency and consumes 2.0× less peak memory on TITAN Xp.

On the GPT-2 backbone, the NAS-generated models utilize on average 11.8% less memory while achieving the same validation perplexity and latency on an ARM CPU. The benefits are larger on Corei7 and TitanXP where the latency savings are pushed to 13.8% and 11.9%, respectively. The peak memory utilization is also decreased by 9.7% and 12.9%, on average, compared to the baseline scaled GPT-2s on Corei7 and TITAN Xp, respectively.

Search Efficiency. The main component in LTS search time is the latency/peak memory utilization measurement for candidate architectures since evaluating the model perplexity is instant using the decoder parameter count. Our lightweight search, therefore, finishes in a few hours on commodity hardware, i.e., 0.9, 2.6, and 17.2 hours on a TITAN Xp GPU, Corei7 CPU, and an ARM core, respectively. To provide more context, full training of just one 16-layer Transformer-XL base model on LM1B using a machine with 8× NVIDIA V100 GPUs takes 15.8 hours.

5 Limitations and Future Work

Decoder parameter count provides a simple yet accurate proxy for ranking autoregressive models. This should serve as a strong baseline for future works on Transformer NAS. Our focus is entirely on autoregressive, decoder-only transformers. We therefore, study perplexity as the commonly used metric for language modeling tasks. Nevertheless, recent literature on scaling laws for Transformers suggest a similar correlation between parameter count and task metrics may exist for encoder only (BERT-style) Transformers or encoder-decoder models used in neural machine translation (NMT) [12]. Additionally, recent findings [28] show specific scaling laws exist between model size and downstream task metrics, e.g., GLUE [32]. Investigating such scaling laws for NAS on heterogeneous BERT-style or NMT models with new metrics is an important future direction.

References

[1] M. S. Abdelfattah et al. “Zero-Cost Proxies for Lightweight NAS”. In: *International Conference on Learning Representations*. 2020.

[2] A. Baevski and M. Auli. “Adaptive Input Representations for Neural Language Modeling”. In: *International Conference on Learning Representations*. 2018.

[3] T. B. Brown et al. “Language models are few-shot learners”. In: *arXiv preprint arXiv:2005.14165* (2020).

[4] H. Cai et al. “Once-for-All: Train One Network and Specialize it for Efficient Deployment”. In: *International Conference on Learning Representations*. 2019.

[5] C. Chelba et al. “One billion word benchmark for measuring progress in statistical language modeling”. In: *arXiv preprint arXiv:1312.3005* (2013).

[6] Z. Dai et al. “Transformer-XL: Attentive Language Models beyond a Fixed-Length Context”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. 2019, pp. 2978–2988.

[7] J. Devlin et al. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *NAACL-HLT (1)*. 2019.

[8] T. Elsken, J. H. Metzen, and F. Hutter. “Efficient Multi-Objective Neural Architecture Search via Lamarckian Evolution”. In: *International Conference on Learning Representations*. 2019.

[9] T. Elsken, J. H. Metzen, and F. Hutter. *Neural Architecture Search: A Survey*. 2019. arXiv: 1808.05377 [stat.ML].

[10] H. Face. *OpenAI GPT2 by Hugging Face*. https://huggingface.co/docs/transformers/model_doc/gpt2.

[11] J. Gao et al. *AutoBERT-Zero: Evolving BERT Backbone from Scratch*. 2021. arXiv: 2107.07445 [cs.CL].

[12] B. Ghorbani et al. “Scaling laws for neural machine translation”. In: *arXiv preprint arXiv:2109.07740* (2021).

[13] J. Guerrero-Viu et al. “Bag of baselines for multi-objective joint neural architecture search and hyperparameter optimization”. In: *arXiv preprint arXiv:2105.01015* (2021).

[14] H. Hu et al. “Efficient Forward Architecture Search”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/6c468ec5a41d65815de23ec1d08d7951-Paper.pdf.

[15] J. Kaplan et al. “Scaling laws for neural language models”. In: *arXiv preprint arXiv:2001.08361* (2020).

[16] N. Lee, T. Ajanthan, and P. Torr. “SNIP: SINGLE-SHOT NETWORK PRUNING BASED ON CONNECTION SENSITIVITY”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=B1VZqjAcYX.

[17] H. Liu, K. Simonyan, and Y. Yang. “DARTS: Differentiable Architecture Search”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=S1eYHoC5FX.

[18] J. Mellor et al. “Neural architecture search without training”. In: *International Conference on Machine Learning*. PMLR. 2021, pp. 7588–7598.

[19] J. Mellor et al. *Neural Architecture Search without Training*. 2021. URL: https://openreview.net/forum?id=g4E6SAAvACo.

[20] S. Merity et al. “Pointer sentinel mixture models”. In: *arXiv preprint arXiv:1609.07843* (2016).

[21] X. Ning et al. “Evaluating Efficient Performance Estimators of Neural Architectures”. In: *Advances in Neural Information Processing Systems* 34 (2021).

[22] NVIDIA. *Transformer-XL For PyTorch*. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/Transformer-XL.

[23] H. Pham et al. “Efficient neural architecture search via parameters sharing”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 4095–4104.

[24] A. Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.

[25] D. R. So, C. Liang, and Q. V. Le. *The Evolved Transformer*. 2019. arXiv: 1901.11117 [cs.LG].

[26] D. R. So et al. *Primer: Searching for Efficient Transformers for Language Modeling*. 2021. arXiv: 2109.08668 [cs.LG].

[27] H. Tanaka et al. “Pruning neural networks without any data by iteratively conserving synaptic flow”. In: *Advances in Neural Information Processing Systems* 33 (2020).

[28] Y. Tay et al. “Scale efficiently: Insights from pre-training and fine-tuning transformers”. In: *arXiv preprint arXiv:2109.10686* (2021).

[29] L. Theis et al. “Faster gaze prediction with dense networks and fisher pruning”. In: *arXiv preprint arXiv:1801.05787* (2018).

[30] H. Tsai et al. *Finding Fast Transformers: One-Shot Neural Architecture Search by Component Composition*. 2020. arXiv: 2008.06808 [cs.LG].

[31] A. Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[32] A. Wang et al. “GLUE: A multi-task benchmark and analysis platform for natural language understanding”. In: *arXiv preprint arXiv:1804.07461* (2018).

[33] C. Wang, G. Zhang, and R. Grosse. “Picking Winning Tickets Before Training by Preserving Gradient Flow”. In: *International Conference on Learning Representations*. 2020. URL: https://openreview.net/forum?id=SkgsACVKPH.

[34] H. Wang et al. “HAT: Hardware-Aware Transformers for Efficient Natural Language Processing”. In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020, pp. 7675–7688.

[35] H. Wang et al. “DeepNet: Scaling Transformers to 1,000 Layers”. In: *arXiv preprint arXiv:2203.00555* (2022).

[36] C. White et al. “A Deeper Look at Zero-Cost Proxies for Lightweight NAS”. In: *ICLR Blog Track*. https://iclr-blog-track.github.io/2022/03/25/zero-cost-proxies/. 2022. URL: https://iclr-blog-track.github.io/2022/03/25/zero-cost-proxies/.

[37] M. Wistuba, A. Rawat, and T. Pedapati. *A Survey on Neural Architecture Search*. 2019. arXiv: 1905.01392 [cs.LG].

[38] J. Xu et al. *Analyzing and Mitigating Interference in Neural Architecture Search*. 2021. arXiv: 2108.12821 [cs.CL].

[39] J. Xu et al. “NAS-BERT”. In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (Aug. 2021). URL: http://dx.doi.org/10.1145/3447548.3467262.

[40] Y. Xu et al. “PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search”. In: *International Conference on Learning Representations*. 2019.

[41] Y. Yin et al. “Autotinybert: Automatic hyper-parameter optimization for efficient pre-trained language models”. In: *arXiv preprint arXiv:2107.13686* (2021).

[42] Y. You et al. “Large batch optimization for deep learning: Training bert in 76 minutes”. In: *arXiv preprint arXiv:1904.00962* (2019).

[43] Y. Zhao et al. *Memory-Efficient Differentiable Transformer Architecture Search*. 2021. arXiv: 2105.14669 [cs.LG].

A Preliminaries on Autoregressive Transformers

Perplexity. Perplexity is a widely used metric for evaluating language models. This metric encapsulates how well the model can predict a word. Formally, perplexity of a language model M is derived using the entropy formula as:

$$Perplexity(M) = 2^{H(L,M)} = 2^{-\sum_x L(x).log(M(x)))} \tag{1}$$

where L represents the ground-truth words. As seen, the perplexity is closely tied with the cross-entropy loss of the model, i.e., $H(L, M)$.

Parameter count. Contemporary autoregressive Transformer architectures comprise three main components, namely, the input embedding layer, hidden layers, and the final (softmax) projection layer. The embedding head often comprises look-up table based modules which map the input language tokens to vectors. These vectors then enter a stack of multiple hidden layers a.k.a, the decoder blocks. Each decoder block is made up of an attention layer and a feed-forward network. Once the features are extracted by the stack of decoder blocks, the final prediction is generated by passing through a softmax projection layer. When counting the number of parameters in an autoregressive Transformer, the total parameters enclosed in the hidden layers is dubbed the decoder parameter count or equivalently, the non-embedding parameter count. These parameters are architecture dependent and do not change based on the underlying tokenization or the vocabulary size. The embedding parameter count, however, accounts for the parameters enclosed in the input embedding layer as well as the softmax projection layer, both of which are closely tied to the tokenization and vocabulary size. We visualize an autoregressive Transformer in Figure 5, where the orange blocks contain the decoder parameters and grey blocks hold the embedding parameters.

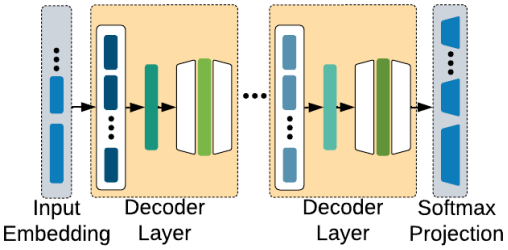


Figure 5: High-level visualization of different components in autoregressive Transformers. Here, the parameters enclosed in the orange blocks are counted as decoder parameters, while the parameters contained in the gray boxes denote the embedding parameter count.

B Additional Related Work

NAS for Encoder-only Architectures. Relative to decoder-only autoregressive language models, encoder-only architectures like BERT [7] have received much more recent attention from the NAS community. NAS-BERT [39] trains a supernet to efficiently search for masked language models (MLMs) which are compressed versions of the standard BERT, Such models can then be used in downstream tasks as is standard practice. Similar to NAS-BERT, Xu et al. [38] also train a supernet to conduct architecture search with the aim of finding more efficient BERT variants. They find interesting empirical insights into supernet training issues like differing gradients at the same node from different child architectures and different tensors as input and output at every node in the supernet. The authors propose fixes that drastically improves supernet training. Gao et al. [11], Tsai et al. [30], and Yin et al. [41] also conduct variants of supernet training with the aim of finding more efficient BERT models.

NAS for Encoder-Decoder Architectures: Applying the well-known DARTS [17] approach to Transformer search spaces leads to memory-hungry supernets. To mitigate this issue, [43] propose

a multi-split reversible network and a memory efficient backpropagation algorithm. One of the earliest papers that applied discrete NAS to Transformer search spaces was by So et al. [25], who use a modified form of evolutionary search. Due to the expense of directly performing discrete search on the search space, this work incurs extremely high computation overhead. Follow-up work by [34] uses the Once-For-All [4] approach to train a supernet for encoder-decoder architectures used in machine translation. Search is performed on subsamples of the supernet that inherit weights to estimate task accuracy. For each target device, the authors train a small neural network regressor on thousands of architectures to estimate latency. As opposed to using a latency estimator, LTS evaluates the latency of each candidate architecture on the target hardware. Notably, by performing the search directly on the target platform, LTS can easily incorporate various hardware performance metrics, e.g., peak memory utilization, for which accurate estimators may not exist. To the best of our knowledge, such holistic integration of multiple hardware metrics in Transformer NAS has not been explored previously.

C Search Space

Figure 6 shows all elastic parameters in LTS search space, namely, number of layers (n_{layer}), number of attention heads (n_{head}), decoder output dimension (d_{model}), inner dimension of the feed forward network (d_{inner}), embedding dimension (d_{embed}), and the division factor (k) of adaptive embedding [2]. These architectural parameters are compatible with popularly used autoregressive Transformer backbones, e.g., GPT. We adopt a heterogeneous search space where the backbone parameters are decided on a per-layer basis. This is in contrast to the homogeneous structure commonly used in Transformers [3, 6], which reuses the same configuration for all layers. Compared to homogeneous models, the flexibility associated with heterogeneous architectures enables them to obtain much better hardware performance under the same perplexity budget (see Section 4.2).

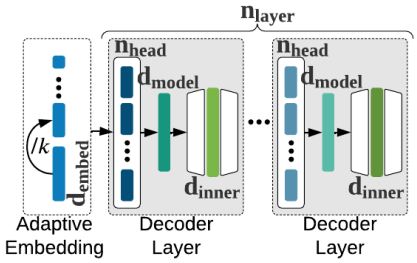


Figure 6: Elastic parameters in LTS search space.

Heterogeneous search space was previously explored in [34]. However, due to the underlying supernet structure, not all design parameters can change freely. As an example, the dimensionality of the *Q, K, V* vectors inside the encoder and decoder layers is fixed to 512 to accommodate inheritance from the supernet. Our search space, however, allows exploration of all internal dimensions without constraints. By not relying on the supernet structure, our search space easily encapsulates various Transformer backbones with different configurations of the input/output embedding layers and elastic internal dimensions.

LTS searches over the following values for the architectural parameters in our backbones: $n_{layer} \in \{2, \ldots, 16|1\}$[7], $d_{model} \in \{128, \ldots, 1024|64\}$, $d_{inner} \in \{128, \ldots, 4096|64\}$, and $n_{head} \in \{2, 4, 8\}$. Additionally we explore adaptive input embedding [2] with $d_{embed} \in \{128, 256, 512\}$ and factor $k \in \{1, 2, 4\}$. Once a d_{model} is sampled, we adjust the lower bound of the above range for d_{inner} to $2 \times d_{model}$. Encoding this heuristic inside the search ensures that the acquired models will not suffer from training collapse. Our heterogeneous search space encapsulates more than 10^{54} different architectures. Such high dimensionality further validates the critical need for training-free NAS.

D Decoder Parameter Count Proxy

We randomly sample 200 architectures from the GPT-2 backbone and train them fully on WikiText-103 and LM1B. We then compare the (ground-truth) ranking of these architectures based on their final validation perplexity with the ranking obtained from the decoder parameter count proxy. As shown in Figure 7, our training-free decoder parameter proxy holds a high SRC of 0.98 with final perplexity after full training for both datasets.

[7] We use the notation $\{v_{min}, \ldots, v_{max}|\text{step size}\}$ to show the valid range of values.

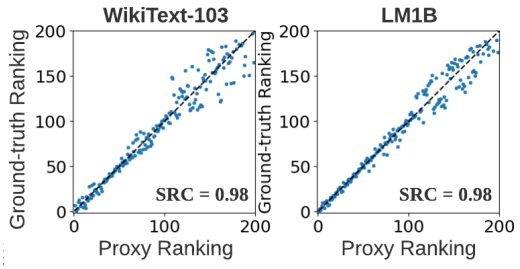


Figure 7: Our zero-cost proxy is highly correlated with the (ground-truth) perplexity after full training on WikiText-103 and LM1B, achieving an SRC of 0.98. Experiment is conducted on 200 models randomly sampled from the GPT-2 backbone.

E Experimental Setup

Datasets. We conduct experiments on two datasets, WikiText-103 and LM1B. The datasets are tokenized using word-level and byte-pair encoding for models with Transformer-XL and GPT-2 backbones, respectively.

Training and Evaluation. We adopt the open-source code by [10] and [22] to implement the GPT-2 and Transformer-XL backbones, respectively. For each backbone and dataset, we use the same training setup for all models generated by NAS. Table 1 encloses the hyperparameters used for training all models in the experiments section of the paper. We follow the training hyperparameters, i.e., batch size, optimizer, learning rate values and scheduler, provided in NVIDIA's open-source repository [22]. Validation perplexity is measured over a sequence length of 192 and 32 tokens for WikiText-103 and LM1B datasets, respectively. Inference latency and peak memory utilization are measured on the target hardware for a sequence length of 192, averaged over 10 measurements. We utilize PyTorch's native benchmarking interface for measuring the latency and memory utilization of candidate architectures.

Table 1: LTS training hyperparameters for different backbones. Here, DO is the abbreviation used for dropout layers.

Backbone	Dataset	Tokenizer	# Vocab	Optim.	# Steps	Batch size	LR	Scheduler	Warmup	DO	Attn DO
Transformer-XL	WT103	Word	267735	LAMB [42]	4e4	256	1e-2	Cosine	1e3	0.1	0.0
	LM1B	Word	267735	Adam	1e5	224	2.5e-4	Cosine	2e4	0.0	0.0
GPT-2	WT103	BPE	50257	LAMB [42]	4e4	256	1e-2	Cosine	1e3	0.1	0.1
	LM1B	BPE	50257	LAMB [42]	1e5	224	2.5e-4	Cosine	2e4	0.1	0.1

Search Setup. Evolutionary search is performed for 30 iterations with a population size of 100; the parent population accounts for 20 samples out of the total 100; 40 mutated samples are generated per iteration from a mutation probability of 0.3; and 40 samples are created using crossover.

Backbones. We apply our search on two widely used autoregressive Transformer backbones, namely, Transformer-XL [6] and GPT-2 [24] that are trained from scratch with varying architectural hyperparameters. The internal structure of these backbones are quite similar in that they contain decoder blocks with attention and feed-forward layers. The difference between the backbones lies mainly in their dataflow structures; the models derived from Transformer-XL backbone adopt a recurrence methodology over past states, which coupled with relative positional encoding, allows for modeling longer term dependencies.

Performance Criteria. To evaluate the ranking performance of various proxies, we first establish a ground-truth ranking of candidate architectures by training them until full convergence. This ground-truth ranking is then utilized to compute two performance criteria as follows:

► Common Ratio (CR): We define CR as the percentage overlap between the ground-truth ranking of architectures versus the ranking obtained from the proxy over the top-k% of architectures. CR quantifies the ability of the proxy ranking to identify the top-k% architectures.

► Spearman's Rank Correlation (SRC): We use this metric to measure the correlation between the proxy ranking and the ground-truth. Ideally, the proxy ranking should have high correlation with the ground-truth over the entire search space as well as high-performing candidate models.

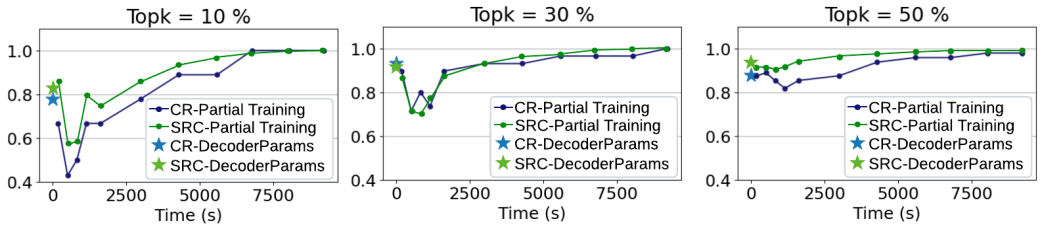
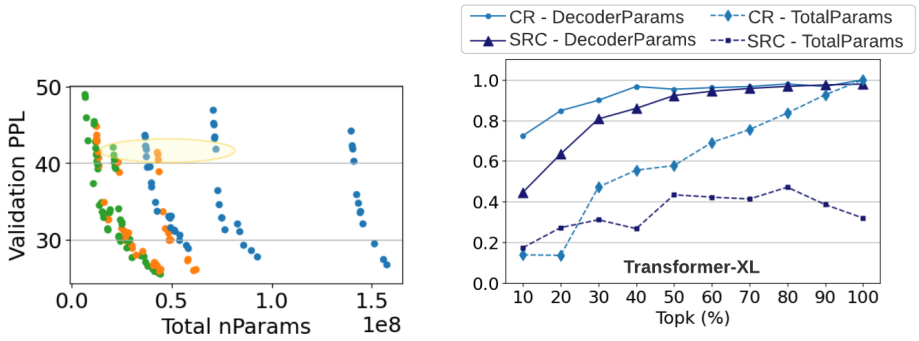


Figure 8: Comparison between partial training and our zero-cost proxy, i.e., decoder parameter count, in terms of ranking performance and timing overhead. Experiment is conducted on 100 randomly selected models from the Transformer-XL backbone, trained on WikiText-103. The horizontal axis denotes the average time required for τ iterations of training across all sampled models.

F Decoder Parameter Count versus Total Parameters

Figure 9a demonstrates the final validation perplexity versus total number of model parameters for 300 randomly sampled architectures from the Transformer-XL backbone. This figure contains two important observations: (1) the validation perplexity has a downward trend as the number of parameters increase, (2) The discontinuity is caused by the dominance of embedding parameters when moving to the small Transformer regime. We highlight several example points in Figure 9a where the architectures are nearly identical but the adaptive input embedding factor k is changed. Changing $k \in \{1, 2, 4\}$ (shown with different colors in Figure 9a) varies the total parameter count without much influence on the validation perplexity.

The above observations motivate us to evaluate two proxies, i.e., total number of parameters and only decoder parameter count. Figure 9b demonstrates the CR and SRC metrics evaluated on the 300 randomly sampled models divided into top-k% bins based on their validation perplexity. As shown, the total number of parameters has a lower SRC with the validation perplexity, compared to decoder parameter count. This is due to the masking effect of embedding parameters. The total number of decoder parameters, however, provides a highly accurate, zero-cost proxy with an SRC of 0.97 with the perplexity after full training. We further show the high correlation between decoder parameter count and validation perplexity for Transformer architectures with homogeneous decoder blocks in Appendix G.



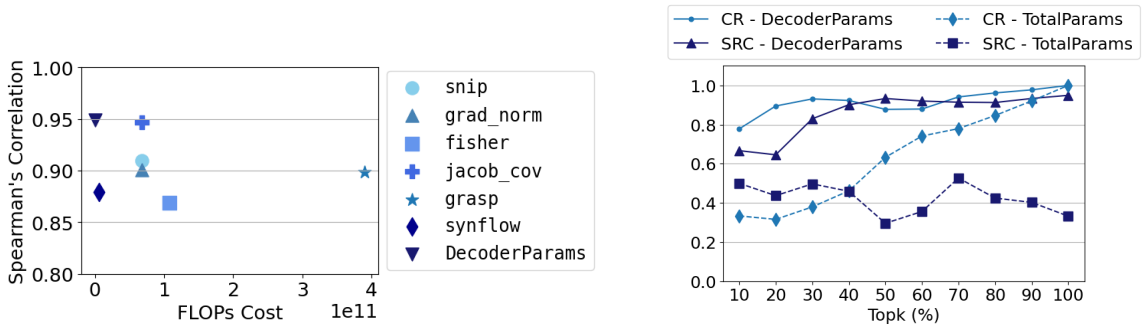
(a) (b)

Figure 9: (a) Validation perplexity after full training versus total parameters. The clear downward trend suggests a strong correlation between parameter count and perplexity. (b) Ranking performance of parameter count proxies. The decoder parameter count provides a very accurate ranking proxy while the total parameter count has a low SRC and CR, due to the masking effect of embedding parameter count. Experiments are conducted on 300 architectures randomly sampled from the Transformer-XL backbone and trained on WikiText-103.

G Analysis on Homogeneous Models

In this section, we evaluate the efficacy of the proposed proxies on the homogeneous search space, i.e., when all decoder layers have the same parameter configuration. In this scenario, the parameters are sampled from the valid ranges in Section 3 to construct one decoder block. This block is then replicated based on the selected n_{layer} to create the Transformer architecture. In what follows, we provide experimental results gathered on 100 randomly sampled Transformer models from the Transformer-XL backbone with homogeneous decoder blocks, trained on WikiText-103.

► **Low-cost Proxies.** Figure 10a demonstrates the SRC between various low-cost methods and the validation perplexity after full training. On the horizontal axis, we report the total computation required for each proxy in terms of FLOPs. Commensurate with the findings on the heterogeneous models, we observe a strong correlation between the low-cost proxies and validation perplexity, with decoder parameter count outperforming other proxies. Note that we omit the `relu_log_det` method from Figure 10a as it provides a low SRC of 0.42 due to heavy reliance on ReLU activations.



(a) (b)

Figure 10: Experiments conducted on 100 randomly sampled Transformers with homogeneous decoder blocks, trained on WikiText-103. (a) SRC between ranking obtained from low-cost proxies and the ground-truth ranking after full training. The decoder parameter count obtains the best SRC with zero cost. (b) Performance of parameter count proxies. The decoder parameter count provides a very accurate ranking proxy with an SRC of 0.95 over all models.

► **Parameter Count.** As seen in Figure 10b, the total parameter count has a low SRC with the validation perplexity while the decoder parameter count provides an accurate proxy with an SRC of 0.95 over all architectures. These findings on the homogeneous search space are well-aligned with the observations in the heterogeneous space.

H How good is the decoder parameters proxy for pareto-frontier search?

Before we use the decoder parameter count as a proxy for perplexity in the inner loop of pareto-frontier search, we validate whether this proxy will actually help find pareto-frontiers which are close to the groundtruth. We first fully train all 1200 architectures sampled from the Transformer-XL backbone during evolutionary search (See Algorithm 1). Using the validation perplexity obtained after full training, we rank all sampled architectures and extract the ground-truth pareto-frontier of perplexity versus latency. We train the models on the WikiText-103 dataset and benchmark Intel Xeon E5-2690 CPU as our target hardware platform for latency measurement in this experiment.

Figure 11 represents a scatter plot of the validation perplexity (after full training) versus latency for all sampled architectures during the search. The ground-truth pareto-frontier, by definition, is the lower convex hull of the dark navy dots, corresponding to models with the lowest validation perplexity for any given latency constraint. We mark the pareto-frontier points found by the training-free proxy with orange color. As shown, the architectures that were selected as the pareto-frontier by the proxy method are either on or very close to the ground-truth pareto-frontier.

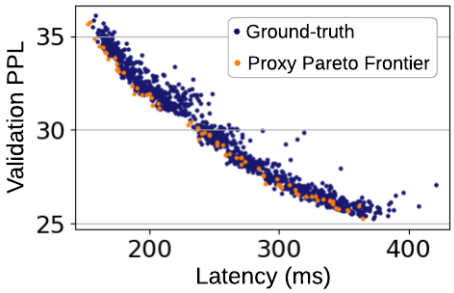


Figure 11: Perplexity versus latency pareto obtained from full training of 1200 architectures sampled during NAS on Transformer-XL backbone. Orange points are the pareto-frontier extracted using decoder parameter count proxy, which lies closely to the actual pareto-frontier. Decoder parameter count holds a SRC of 0.98 with the ground-truth perplexity after full training.

We define the mean average perplexity difference as a metric to evaluate the distance (d_{avg}) between the proxy and ground-truth pareto-frontier:

$$d_{avg} = \frac{1}{N} \sum_{i=1}^{N} \frac{|p_i - p_{gt,i}|}{p_{gt,i}} \tag{2}$$

Here, p_i denotes the i-th point on the proxy pareto front and $p_{gt,i}$ is the closest point, in terms of latency, to p_i on the ground-truth pareto front. The mean average perplexity difference for Figure 11 is $d_{avg} = 0.6\%$. This low difference further validates the effectiveness of our zero-cost proxy in correctly ranking the sampled architectures and estimating the true pareto-frontier. In addition to the low distance between the pareto-frontier estimated using decoder parameter count proxy and the ground-truth, our zero-cost proxy holds a high SRC of 0.98 over the entire pareto, i.e., all 1200 sampled architectures. We provide the ranking performance for different evaluation techniques along with their cost in Table 2. As seen, our proposed evaluation proxy, i.e., decoder parameter count, provides the highest SRC with the ground-truth ranking of the sampled architectures during NAS, while removing the extremely high overhead of training. This proxy can hence be effectively used in the inner loop of search for NAS.

Table 2: Comparison between training-based and the proposed training-free evaluation proxy on a real NAS benchmark. The SRC is computed on the entire population set of 1200 models visited during the search. The training time is reported on the Nvidia V100 GPU.

	Train Iter	GPU Hours	CO_2e (lbs)	SRC
Full Training	40,000	19,024	5433	1.0
Partial Training	500	231	66	0.92
	5,000	2690	768	0.96
# Decoder Params	**0**	**0**	**~0**	**0.98**

We further study the decoder parameter proxy in scenarios where the range of model sizes provided for search is limited. We categorize the total 1200 sampled architectures into different bins based on the decoder parameters. Figure 12 demonstrates the SRC between decoder parameter count proxy and the validation perplexity after full training for different model sizes. The proposed proxy provides a highly accurate ranking of candidate architectures even when exploring a small range of model sizes.

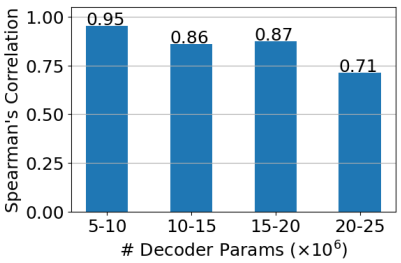


Figure 12: SRC between the decoder parameter count proxy and validation perplexity. Results are gathered on 1200 models grouped into four bins based on their decoder parameter count. Our proxy performs well even when exploring within a small range of model sizes.

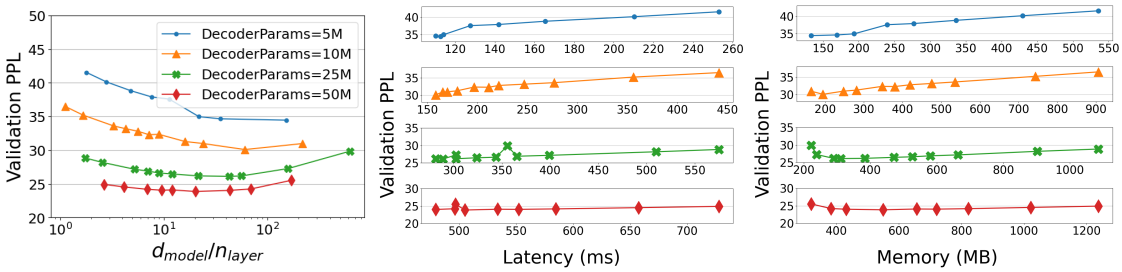


Figure 13: Validation perplexity after full training versus (a) the width-to-depth aspect ratio, (b) latency, and (c) peak memory utilization. Models are randomly generated from the GPT-2 backbone and trained on WikiText-103. For a given decoder parameter count, we observe low variation in perplexity across different models, regardless of their topology. The topology, however, significantly affects the latency (up to 2.8×) and peak memory utilization (up to 5.5×) for models with the same perplexity.

I How does model topology affect the decoder parameter count proxy?

The low-cost proxies introduced in Section 3.1, rely on forward and backward passes through the network. As such, they automatically capture the topology of the underlying architecture by relying on the dataflow to compute the final proxy score. The decoder parameter count proxy, however, is topology-agnostic. In this section, we investigate the effect of topology on the performance of our proposed decoder parameter count proxy. Specifically, we seek to answer whether for a given decoder parameter count budget, the aspect ratio of the architecture, i.e., trading-off the width versus the depth, can affect the final validation perplexity.

We define the aspect ratio of the architecture as d_{model} (=width), divided by n_{layer} (=depth). This metric has been used in previous work which study scaling laws for language models [15] to quantify the skewness of the topology. For a given decoder parameter count budget, we generate several random architectures from the GPT-2 backbone with a wide range of the width-to-depth aspect ratios[8]. The generated models span wide, shallow topologies (e.g., d_{model}=1256, n_{layer}=2) to narrow, deep topologies (e.g., d_{model}=112, n_{layer}=100). Figure 13a shows the validation perplexity of said architectures after full training on WikiText-103 versus their aspect ratio. The maximum deviation (from median) of the validation perplexity is $< 12.8\%$ for a given decoder parameter count, across a wide range of aspect ratios $\sim 1 - 630$. Our findings on the heterogeneous models complement the empirical results by [15] where decoder parameter count largely determines perplexity for homogeneous Transformer models, irrespective of shape (see Fig 5 in [15]).

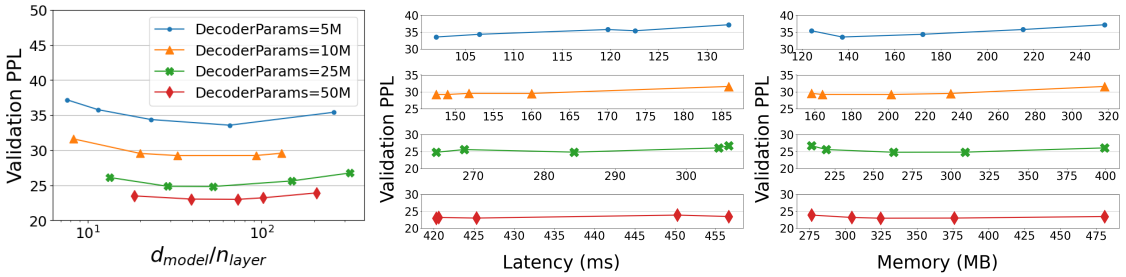
We observe stable training when scaling models from the GPT-2 backbone up to 100 layers, with the perplexity increasing only when the aspect ratio nears 1. Nevertheless, such deep models are not part of our search space as they have a high latency and are unsuitable for lightweight inference. For the purposes of hardware-aware and efficient Transformer NAS, decoder parameter count proxy holds a very high correlation with validation perplexity, regardless of the architecture topology as shown in Figure 13a.

Note that while models with the same parameter count have very similar validation perplexities, the topology in fact affects their hardware performance, i.e., latency (up to 2.8×) and peak memory utilization (up to 5.5×), as shown in Figure 13b and 13c. This motivates the need for incorporating hardware metrics in NAS to find the best topology.

We further validate the effect of topology on decoder parameter count proxy for the Transformer-XL backbone in Figure 14. Here, the generated models span wide, shallow topologies (e.g., d_{model}=1024, n_{layer}=3) to narrow, deep topologies (e.g., d_{model}=128, n_{layer}=35). Our results demonstrate less than 7% deviation (from the median) in validation perplexity for aspect ratios ranging from $\sim 8 - 323$. Nevertheless, for the same decoder parameter count budget, the latency can vary by 1.3× and the peak memory utilization by 2.0× as shown in Figure 14b and 14c, respectively.

[8]We control the aspect ratio by changing the width, i.e., d_{model} while keeping d_{inner}=2×d_{model} and n_{head}=8. The number of layers is then derived such that the total parameter count remains the same.

For deeper architectures (more than 40 layers) with the Transformer-XL backbone, we observe an increase in the validation perplexity, which results in a deviation from the pattern in Figure 14a. This observation is associated with the inherent difficulty in training deeper architectures, which can be mitigated with proposed techniques in the literature [35]. Nevertheless, to facilitate generation of lightweight lightweight Transformers, our search-space contains architectures with less than 16 layers. In this scenario, the decoder parameter count proxy, independent of the topology, highly correlates with validation perplexity as seen in Figure 14a.



(a) (b) (c)

Figure 14: Validation perplexity after full training versus (a) the width-to-depth aspect ratio, (b) latency, and (c) peak memory utilization. Models are randomly generated from the Transformer-XL backbone and trained on WikiText-103. For a given decoder parameter count, we observe low variation in perplexity across different models, regardless of their topology. The topology, however, significantly affects the latency and peak memory utilization for models with the same perplexity.

J 3D Pareto Visualization

Figure 15 visualizes the 3-dimensional pareto for the GPT-2 backbones. Here, the black and blue points denote the regular and pareto-frontier architectures, respectively. The pair of red dots are architectures which match in both memory and decoder parameter count (~ perplexity). However, as shown, their latency differs by 2×. The pair of green points correspond to models with the same decoder parameter count (~ perplexity) and latency, while the memory still differs by 30MB, which is non-negligible for memory-constrained application. In a 2-objective pareto-frontier search of perplexity versus memory (or latency), each pair of red (or green) dots will result in similar evaluations. While in reality, they have very different characteristics in terms of the overlooked metric. This experiment validates the need for multi-objective pareto-frontier search, which simultaneously takes into account multiple hardware performance metrics.

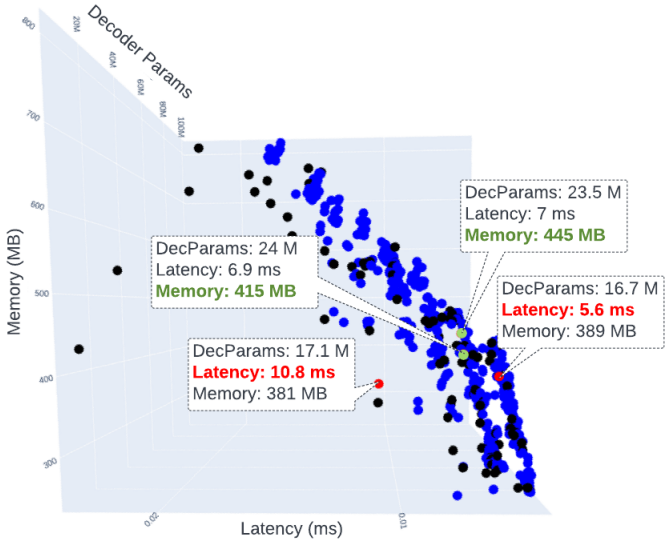


Figure 15: 3D visualization of our multi-objective NAS for the GPT-2 backbone on TITAN Xp GPU. Architectures with similar memory and decoder parameter count can result in drastically different runtimes (up to 2× difference). Similarly, architectures with similar decoder parameter count and latency may have different peak memory utilization. Therefore, it is important to perform multi-objective NAS where several hardware characteristics are simultaneously taken into account when extracting the pareto-frontier.

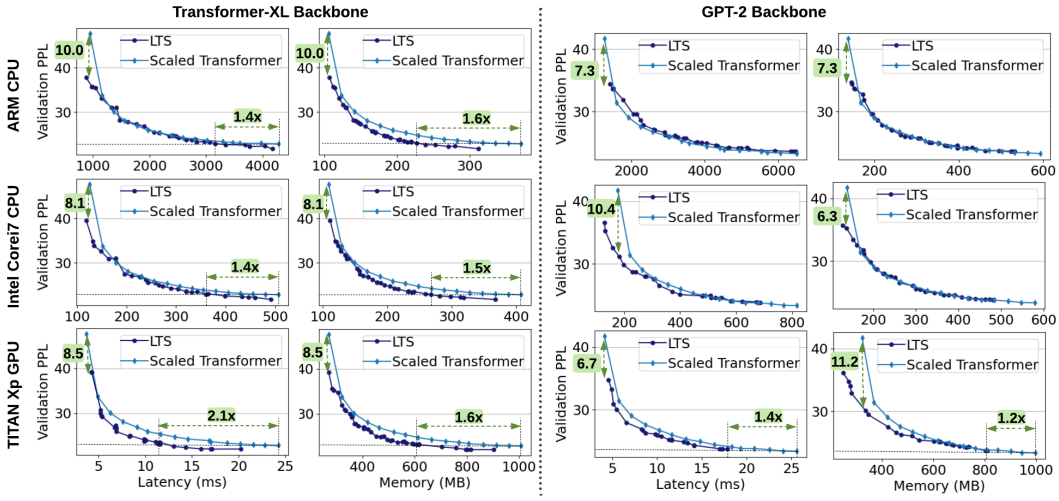


Figure 16: 2D visualization of the perplexity versus latency and memory pareto-frontier found by LTS and scaled backbone models with varying number of layers. All models are trained on the WikiText-103 dataset. The architectural parameters for all models are enclosed in Appendix L.

K LTS Performance Comparison on WikiText-103

We compare the pareto-frontier architectures found by LTS with the baseline after full training on the WikiText-103 dataset in Figure 16. Commensurate with the findings on the LM1B dataset, the NAS-generated models outperform the baselines in at least one of the three metrics, i.e., perplexity, latency, and peak memory utilization. We note that the gap between the baseline models and those obtained from NAS is larger when training on the LM1B dataset. This is due to the challenging nature of LM1B, which exceeds the WikiText-103 dataset size by $\sim 10\times$. Thus, it is harder for hand-crafted baseline models to compete with the optimized LTS architectures on LM1B.

On the Transformer-XL backbone, the models on LTS pareto-frontier for the ARM CPU have, on average, 3.8% faster runtime and 20.7% less memory under the same validation perplexity budget. On the Corei7, the runtime and memory savings increase to 13.2% and 19.6%, respectively, while matching the baseline perplexity. We achieve our highest benefits on TITAN Xp GPU where the pareto-frontier of LTS has on average 31.8% lower latency and 21.5% less memory. Notably, the validation perplexity of the baseline 16-layer Transformer-XL base can be achieved with a lightweight model with $2.1\times$ less latency while consuming $1.6\times$ less memory at runtime.

On the GPT-2 backbone, LTS achieves $6.3 - 11.2$ lower perplexity in the low-latency-and-memory regime. As we transition to larger models and higher latency, our results show that the GPT-2 architecture is nearly optimal on WikiText-103 when performing inference on a CPU. The benefits are more significant when targeting a GPU; For any given perplexity achieved by the baseline, LTS pareto-frontier on TITAN Xp delivers, on average, 9.0% lower latency and 4.5% lower memory. Therefore, the perplexity and memory of the baseline 16-layer GPT-2 can be achieved by a new model that runs $1.4\times$ faster and consumes $1.2\times$ less memory on TITAN Xp.

L Architecture Details

Tables 3, 4, 5, 6 enclose the architecture parameters for the baseline and NAS-generated models in Figure 4 for Transformer-XL and GPT-2 backbones. For each target hardware, the rows of the table are ordered based on increasing decoder parameter count (decreasing validation perplexity). For all models, $d_{head}=d_{model}/n_{head}$, the adaptive input embedding factor is set to $k = 4$, and $d_{embed}=d_{model}$.

Table 3: Detailed architectural parameters for all models in Figure 4 with Transformer-XL backbone.

		n_{layer}	d_{model}	n_{head}	d_{inner}	DecoderParams (M)
baseline		ϵ[1,16]	512	8	2048	-
ARM	M1	2	512	[2, 2]	[1216, 1280]	3.2
	M2	3	320	[2, 4, 2]	[1472, 2368, 3392]	5.5
	M3	2	512	[2, 2]	[2560, 2176]	5.5
	M4	2	512	[2, 2]	[3904, 1792]	6.5
	M5	2	640	[2, 2]	[3520, 3456]	9.8
	M6	2	832	[2, 2]	[3264, 3968]	13.1
	M7	2	704	[8, 2]	[3904, 3968]	13.4
	M8	2	960	[2, 2]	[3648, 3968]	15.9
	M9	2	960	[2, 2]	[3904, 3968]	16.4
	M10	3	960	[2, 2, 2]	[1856, 2368, 3392]	16.5
	M11	3	960	[2, 4, 2]	[3328, 2368, 3200]	19.6
	M12	3	832	[2, 2, 2]	[3904, 3968, 3008]	19.7
	M13	3	960	[2, 2, 2]	[3904, 3584, 3456]	22.9
	M14	3	960	[4, 2, 2]	[3648, 3584, 3584]	23.3
	M15	3	960	[2, 2, 8]	[4032, 3968, 3904]	26.6
	M16	4	896	[4, 2, 8, 2]	[3904, 3008, 3520, 3584]	29.7
	M17	4	960	[2, 2, 2, 2]	[3840, 3904, 3520, 3072]	30.0
	M18	4	960	[2, 2, 2, 2]	[4032, 3648, 3136, 4032]	31.0
	M19	4	960	[2, 2, 4, 2]	[3904, 3968, 3840, 3584]	32.5
	M20	4	960	[8, 8, 8, 4]	[4032, 3968, 2880, 3200]	35.7
	M21	4	960	[8, 2, 4, 8]	[4032, 3584, 3840, 3584]	35.7
	M22	5	960	[2, 2, 2, 2, 2]	[3904, 3968, 3264, 3456, 3200]	37.3
	M23	5	960	[2, 2, 2, 8, 2]	[3904, 3648, 3136, 3648, 3840]	39.9
	M24	6	960	[2, 2, 2, 2, 2, 8]	[3328, 2624, 3392, 2944, 3008, 3904]	42.5
	M25	6	960	[2, 4, 2, 2, 2, 2]	[2112, 3840, 3328, 3264, 3968, 3648]	43.1
	M26	6	960	[2, 2, 2, 2, 2, 4]	[3968, 3968, 3456, 3456, 3776, 2432]	44.8
	M27	6	960	[2, 2, 4, 2, 8, 8]	[3584, 2624, 3392, 3968, 3008, 3328]	46.3
	M28	6	960	[2, 4, 2, 2, 8, 2]	[3904, 3008, 3392, 3648, 3392, 3584]	46.4
	M29	6	960	[8, 8, 2, 4, 2, 4]	[3904, 3648, 3136, 3648, 3200, 3840]	49.7
	M30	6	960	[2, 4, 8, 4, 2, 8]	[3904, 3008, 3392, 3200, 3968, 3904]	49.7
	M31	6	960	[8, 4, 8, 4, 2, 8]	[3904, 3648, 3392, 3200, 3968, 3840]	52.7
	M32	8	896	[4, 2, 2, 4, 4, 2, 4, 8]	[3584, 3968, 3392, 3904, 2240, 1856, 2560, 3264]	53.1
	M33	8	896	[4, 2, 2, 2, 4, 4, 4, 2]	[3584, 3584, 3520, 2368, 2752, 4032, 3520, 3264]	54.7
	M34	8	960	[2, 4, 4, 4, 4, 8, 2, 2]	[3968, 3584, 3520, 3072, 3968, 4032, 1856, 3712]	62.5
	M35	9	896	[4, 2, 4, 4, 8, 2, 8, 8, 2]	[3840, 3136, 3520, 2880, 3200, 3008, 3328, 2560, 3136]	63.4
	M36	9	960	[4, 4, 8, 2, 2, 2, 8, 8, 2]	[2112, 3008, 3520, 3648, 3968, 4032, 1984, 3200, 3520]	68.0
	M37	9	960	[8, 2, 4, 2, 8, 8, 8, 2, 2]	[3968, 3008, 3520, 3200, 3200, 4032, 1984, 2816, 3520]	69.8
	M38	12	832	[2, 4, 4, 2, 2, 8, 8, 8, 4, 4, 2, 8]	[3136, 2112, 2112, 2368, 2752, 2432, 2432, 2176, 3456, 3712, 2880, 3712]	70.4
	M39	12	832	[4, 4, 4, 2, 2, 8, 4, 8, 2, 8, 2, 8]	[3136, 3968, 2112, 2368, 3072, 2240, 2624, 2112, 3456, 3072, 2880, 3264]	72.1
Corei7	M1	2	384	[2, 2]	[896, 2816]	3.4
	M2	2	576	[2, 2]	[1792, 2816]	6.1
	M3	2	832	[2, 2]	[1728, 1536]	6.5
	M4	2	576	[2, 2]	[1408, 3776]	6.7
	M5	2	768	[2, 2]	[2112, 3584]	9.7
	M6	2	768	[2, 2]	[3776, 1920]	9.7
	M7	2	832	[2, 2]	[3776, 3392]	13.0
	M8	2	960	[2, 4]	[1984, 3840]	13.0
	M9	2	832	[2, 2]	[3968, 3584]	13.7
	M10	2	960	[2, 2]	[3904, 3904]	16.2
	M11	2	960	[8, 8]	[3968, 3584]	19.4
	M12	3	960	[2, 2, 4]	[2176, 3840, 2880]	19.6
	M13	3	896	[2, 2, 2]	[2304, 3904, 3904]	19.9
	M14	3	960	[2, 2, 4]	[3776, 2880, 3904]	22.8
	M15	3	960	[2, 8, 2]	[3840, 3840, 3904]	26.0
	M16	3	960	[2, 2, 8]	[3968, 3904, 3904]	26.3
	M17	3	960	[2, 8, 8]	[3904, 3840, 3904]	27.9
	M18	4	960	[2, 4, 2, 2]	[3904, 2112, 4032, 3584]	29.3
	M19	4	960	[2, 2, 2, 4]	[2112, 3840, 3904, 3904]	29.5
	M20	4	960	[2, 2, 2, 4]	[3904, 3776, 3904, 3904]	32.9
	M21	4	960	[2, 4, 8, 4]	[3776, 3392, 3520, 3904]	33.6
	M22	5	960	[2, 2, 2, 2, 2]	[3776, 1984, 3904, 3904, 3456]	35.8
	M23	5	960	[2, 4, 2, 4, 2]	[3968, 3584, 3520, 3904, 3200]	39.3
	M24	5	960	[2, 4, 4, 4, 2]	[3776, 3840, 3904, 3904, 3968]	42.2
	M25	6	960	[2, 4, 2, 4, 2, 4]	[3776, 2112, 4032, 3584, 3200, 4032]	45.4
	M26	6	960	[2, 4, 4, 2, 2, 4]	[3776, 3840, 3904, 3904, 3008, 2304]	45.4
	M27	6	960	[2, 4, 2, 4, 4, 4]	[3776, 3840, 3904, 4032, 3648, 2432]	47.7
	M28	6	960	[4, 2, 8, 4, 2, 2]	[3840, 3712, 3520, 4032, 3200, 4032]	49.7
	M29	8	960	[2, 2, 2, 4, 2, 4, 8, 2]	[3392, 1792, 3904, 3904, 3200, 2432, 1792, 2496]	52.1
	M30	7	960	[2, 2, 8, 4, 2, 2, 4]	[3776, 3840, 3904, 1856, 3072, 3648, 4032]	53.8
	M31	8	960	[2, 2, 4, 4, 2, 4, 8, 2]	[3776, 3008, 4032, 3904, 3520, 3136, 1984, 3648]	60.5
	M32	8	960	[8, 2, 2, 4, 8, 4, 4, 8]	[3776, 3008, 3904, 3904, 2176, 4032, 4032, 3648]	67.1
	M33	9	960	[4, 2, 4, 4, 4, 4, 8, 8, 2]	[3840, 3136, 3520, 4032, 3200, 4032, 3648, 2112, 2368]	69.8
	M34	9	960	[8, 2, 8, 8, 2, 4, 8, 2, 2]	[3520, 3008, 2880, 4032, 3200, 2432, 4032, 3904, 3136]	71.5
	M35	13	768	[2, 8, 2, 4, 2, 2, 4, 2, 2, 8, 8, 8, 4]	[3776, 2112, 1600, 3904, 3840, 2880, 2304, 3200, 2048, 2944, 2816, 3328, 3968]	73.3

Table 4: Detailed architectural parameters for all models in Figure 4 with Transformer-XL backbone.

		n_{layer}	d_{model}	n_{head}	d_{inner}	DecoderParams (M)
baseline		$\in$[1,16]	8	2048	512	-
TITAN Xp	M1	2	384	[2, 2]	[1152, 2432]	3.3
	M2	2	576	[2, 2]	[2048, 1728]	5.1
	M3	2	512	[2, 2]	[2368, 3072]	6.2
	M4	2	448	[8, 2]	[2944, 3008]	6.8
	M5	2	832	[8, 2]	[3264, 3072]	13.2
	M6	2	768	[2, 2]	[3968, 4032]	13.3
	M7	2	896	[8, 4]	[4032, 2880]	15.8
	M8	2	960	[2, 2]	[3840, 3968]	16.2
	M9	2	960	[4, 8]	[3968, 3008]	17.1
	M10	2	960	[4, 8]	[3968, 3648]	18.3
	M11	3	960	[2, 2, 2]	[3584, 3072, 2624]	19.7
	M12	3	896	[2, 2, 2]	[3840, 2880, 3840]	20.7
	M13	3	896	[8, 4, 8]	[4032, 2112, 3392]	22.9
	M14	3	960	[4, 2, 2]	[3840, 3008, 3840]	23.0
	M15	3	960	[2, 2, 8]	[3584, 4032, 4032]	26.1
	M16	3	960	[2, 2, 8]	[4032, 4032, 3840]	26.6
	M17	3	960	[8, 2, 8]	[4032, 4032, 3520]	27.8
	M18	3	960	[8, 4, 8]	[4032, 4032, 4032]	29.4
	M19	4	896	[4, 4, 8, 8]	[4032, 3456, 3328, 3392]	32.4
	M20	4	960	[4, 2, 8, 8]	[3840, 3008, 3328, 3584]	33.2
	M21	4	960	[4, 2, 4, 4]	[3840, 4032, 3904, 4032]	34.7
	M22	4	960	[2, 2, 8, 8]	[4032, 3968, 3904, 3840]	36.4
	M23	5	960	[4, 2, 4, 4, 8]	[3840, 3008, 3392, 2496, 4032]	39.0
	M24	5	960	[2, 2, 4, 4, 4]	[3968, 4032, 3328, 4032, 2752]	39.7
	M25	5	960	[2, 4, 2, 2, 8]	[3968, 3968, 3840, 4032, 3904]	43.4
	M26	5	960	[4, 2, 8, 8, 8]	[3840, 3008, 3840, 3328, 3968]	43.8
	M27	5	960	[8, 2, 8, 8, 4]	[4032, 3008, 3840, 3904, 3968]	45.3
	M28	6	896	[2, 2, 4, 4, 2, 2]	[3840, 3968, 3840, 3328, 3904, 3904]	45.5
	M29	6	896	[8, 4, 8, 4, 8, 8]	[3328, 2112, 3392, 3904, 3328, 3264]	46.2
	M30	6	960	[4, 2, 2, 4, 2, 8]	[3840, 3008, 3840, 3904, 4032, 3392]	49.1
	M31	6	960	[4, 8, 8, 4, 8, 4]	[3072, 3584, 3392, 3840, 3328, 3712]	51.3
	M32	6	960	[2, 4, 8, 8, 4, 2]	[3840, 3968, 3840, 3328, 4032, 3776]	52.4
	M33	6	960	[4, 8, 8, 8, 4, 4]	[3840, 3584, 3392, 3328, 3968, 3776]	53.1
	M34	6	960	[4, 8, 8, 8, 8, 2]	[3840, 3840, 3392, 3840, 3328, 3712]	53.9
	M35	7	960	[4, 8, 8, 8, 8, 2, 8]	[3840, 3968, 3840, 3328, 3968, 3328, 4032]	64.7
	M36	8	960	[4, 2, 8, 8, 8, 4, 8, 8]	[3840, 3968, 3840, 3328, 3072, 3328, 4032, 3072]	70.1
	M37	10	896	[8, 8, 8, 2, 8, 2, 2, 2, 8, 2]	[3840, 3072, 3840, 2560, 3648, 3328, 3840, 3008, 2880, 3328]	74.2
	M38	9	960	[8, 8, 8, 4, 4, 8, 8, 4, 2]	[2752, 3456, 2880, 3904, 2752, 3904, 4032, 3264, 3136]	74.4
	M39	10	896	[8, 4, 8, 8, 8, 2, 8, 2, 4, 8]	[4032, 3008, 3840, 2560, 3904, 3904, 3072, 3264, 2368, 2496]	75.4
	M40	12	832	[2, 4, 8, 8, 8, 8, 8, 8, 8, 8, 4, 2]	[3840, 2816, 2112, 3584, 3648, 2432, 2304, 3008, 2880, 1664, 2432, 3776]	77.7

Table 5: Detailed architectural parameters for all models in Figure 4 with GPT-2 backbone.

		n_{layer}	d_{model}	n_{head}	d_{inner}	DecoderParams (M)
baseline		$\in$[1,16]	1024	12	3072	-
ARM	M1	2	512	[2, 2]	[1920, 1920]	6.0
	M2	3	320	[8, 2, 4]	[1920, 1920, 3712]	6.1
	M3	2	576	[2, 2]	[1344, 3200]	7.9
	M4	3	384	[2, 8, 2]	[3840, 2368, 3328]	9.1
	M5	5	384	[4, 4, 2, 4, 4]	[2880, 1920, 960, 2496, 1280]	10.3
	M6	2	768	[2, 2]	[1600, 2240]	10.6
	M7	5	320	[4, 2, 2, 4, 2]	[1344, 2240, 3776, 3008, 3648]	11.0
	M8	3	768	[2, 2, 4]	[1856, 1792, 1920]	15.7
	M9	3	704	[2, 2, 2]	[3136, 2112, 1920]	16.1
	M10	2	960	[4, 2]	[3584, 2304]	18.7
	M11	6	448	[4, 4, 2, 2, 4, 2]	[3072, 2112, 4032, 2688, 1600, 3072]	19.7
	M12	3	960	[4, 4, 2]	[2368, 2560, 2048]	24.5
	M13	4	704	[4, 8, 4, 2]	[3008, 3776, 2560, 3648]	26.3
	M14	5	704	[4, 2, 4, 2, 8]	[3584, 3136, 3776, 3072, 1856]	31.7
	M15	3	960	[2, 2, 2]	[3392, 3648, 3840]	32.0
	M16	4	960	[4, 2, 8, 2]	[2048, 3328, 1984, 1856]	32.5
	M17	7	704	[2, 4, 4, 4, 8, 2, 2]	[3008, 2560, 1920, 1856, 2112, 1728, 3136]	36.9
	M18	4	960	[2, 2, 4, 8]	[3392, 3456, 2432, 2304]	37.0
	M19	5	832	[4, 4, 4, 4, 4]	[3840, 1920, 4032, 3072, 3968]	41.9
	M20	5	960	[8, 4, 2, 2, 4]	[2560, 2048, 3648, 1728, 2304]	42.1
	M21	5	960	[4, 4, 2, 2, 2]	[3072, 2240, 1984, 2176, 3520]	43.4
	M22	5	960	[2, 4, 4, 4, 2]	[2496, 3648, 3328, 3392, 2112]	47.2
	M23	6	832	[4, 2, 4, 4, 2, 4]	[2496, 3200, 1664, 3904, 3520, 3840]	47.7
	M24	6	960	[8, 2, 2, 2, 8, 4]	[2304, 3328, 3456, 1856, 1792, 2112]	50.7
	M25	5	960	[4, 8, 2, 4, 4]	[3264, 2688, 4032, 3968, 3712]	52.4
	M26	6	960	[2, 4, 4, 2, 2, 2]	[3008, 2624, 4032, 2688, 3520, 2624]	57.7
	M27	6	960	[2, 4, 4, 2, 8, 2]	[2304, 3648, 3328, 3648, 3904, 1728]	57.8
	M28	6	960	[4, 4, 2, 4, 2, 2]	[3072, 2368, 4032, 4032, 3776, 3264]	61.6
	M29	7	960	[2, 2, 2, 8, 4, 8, 4]	[3008, 2304, 1920, 1984, 3520, 2816, 3712]	62.9
	M30	7	960	[2, 4, 4, 4, 4, 2, 2]	[3200, 4032, 2048, 2624, 2112, 2752, 2880]	63.6
	M31	7	960	[2, 4, 4, 4, 4, 2, 4]	[3584, 3648, 3328, 3392, 3200, 1984, 3200]	68.8
	M32	7	960	[2, 4, 8, 8, 2, 2, 8]	[3008, 3648, 3584, 3648, 3008, 1728, 3712]	68.8
	M33	7	960	[4, 4, 2, 4, 4, 8, 4]	[3584, 3840, 3328, 3392, 3136, 2944, 2496]	69.5
	M34	8	960	[8, 2, 2, 8, 2, 2, 8, 2]	[3008, 3648, 1792, 1984, 3008, 2816, 3712, 3520]	74.7
	M35	8	960	[2, 2, 2, 2, 8, 4, 4, 2]	[3008, 2304, 1792, 3008, 3520, 2880, 3712, 3456]	75.1
	M36	8	960	[2, 2, 2, 2, 2, 2, 4, 8]	[3008, 1792, 3840, 3392, 3520, 3136, 3712, 3520]	79.4
	M37	9	960	[2, 2, 4, 4, 8, 8, 4, 2, 4]	[1664, 1792, 2240, 3904, 3648, 3264, 2176, 3712, 1856]	79.9
	M38	11	832	[8, 4, 2, 4, 4, 2, 8, 4, 4, 8, 8]	[3072, 2368, 4032, 3968, 1664, 3968, 2176, 2624, 3840, 2176, 2112]	83.8
	M39	9	960	[4, 2, 4, 8, 2, 2, 4, 2, 4]	[2496, 3648, 3328, 3392, 3648, 1728, 2880, 3520, 2368]	85.1
	M40	9	960	[4, 2, 4, 8, 4, 2, 4, 2, 4]	[3072, 2816, 4032, 2560, 3648, 1728, 3840, 3264, 3456]	87.8
	M41	10	960	[8, 2, 4, 4, 2, 2, 4, 8, 2, 4]	[3648, 1792, 2432, 1856, 3392, 2304, 3776, 2944, 3136, 3904]	93.0
	M42	10	960	[8, 2, 2, 4, 2, 2, 2, 4, 2, 2]	[3264, 2048, 3520, 3904, 3840, 3840, 2624, 3072, 3776, 2304]	98.8
	M43	12	896	[4, 4, 4, 2, 4, 2, 4, 8, 8, 2, 4, 2]	[2048, 3136, 4032, 1792, 3584, 1728, 3136, 3008, 2560, 3200, 3648, 1728]	98.9
	M44	10	960	[4, 2, 8, 4, 2, 8, 4, 4, 4, 2]	[3584, 3968, 3328, 3904, 2368, 2112, 3904, 3520, 3328, 2688]	99.8
	M45	10	960	[8, 2, 4, 4, 4, 4, 4, 2, 2, 8]	[2688, 3200, 3840, 3392, 3520, 3136, 3392, 3520, 2880, 3200]	99.9
Corei7	M1	2	384	[2, 2]	[3840, 2432]	6.0
	M2	3	320	[2, 2, 2]	[2176, 3072, 2496]	6.2
	M3	2	512	[2, 2]	[1408, 2624]	6.2
	M4	3	384	[2, 2, 2]	[3264, 3456, 3584]	9.7
	M5	2	576	[2, 2]	[3136, 3648]	10.5
	M6	3	448	[2, 2, 2]	[4032, 3648, 4032]	12.9
	M7	4	448	[2, 2, 4, 4]	[3072, 3648, 4032, 1792]	14.5
	M8	2	768	[2, 2]	[3968, 3328]	15.9
	M9	4	576	[2, 2, 2, 2]	[3072, 2752, 3456, 3136]	19.6
	M10	2	960	[2, 2]	[3840, 3264]	21.0
	M11	4	640	[2, 2, 2, 2]	[2176, 3648, 3584, 1920]	21.1
	M12	3	960	[2, 2, 2]	[2176, 3264, 2432]	26.2
	M13	4	768	[2, 2, 2, 2]	[3584, 2112, 3392, 1920]	26.4
	M14	4	768	[2, 2, 2, 2]	[3584, 2560, 3776, 1536]	27.1
	M15	4	832	[2, 2, 2, 2]	[3904, 1984, 3392, 3136]	31.8
	M16	3	960	[2, 2, 2]	[3968, 4032, 2880]	32.0
	M17	5	768	[2, 2, 4, 2, 2]	[3648, 3072, 3392, 1984, 2944]	34.9
	M18	4	960	[2, 2, 2, 2]	[3136, 1984, 3392, 2944]	36.8
	M19	4	960	[2, 2, 2, 4]	[3968, 3456, 3584, 3136]	42.0
	M20	6	768	[4, 2, 2, 4, 2, 4]	[3584, 2112, 3456, 3136, 3840, 2560]	42.9
	M21	7	768	[2, 4, 2, 4, 4, 4, 2]	[2624, 1984, 2496, 3968, 2880, 2112, 4032]	47.5
	M22	5	960	[2, 2, 4, 2, 4]	[2176, 3264, 3392, 3008, 3328]	47.6
	M23	6	960	[4, 4, 2, 4, 2, 2]	[2048, 2624, 3520, 1984, 2880, 2624]	52.3
	M24	6	960	[2, 4, 4, 4, 2, 2]	[1792, 3456, 2752, 2240, 1664, 3840]	52.4
	M25	6	960	[4, 2, 2, 2, 4, 4]	[2176, 1664, 3648, 3136, 3968, 3904]	57.7
	M26	7	960	[2, 2, 4, 4, 2, 2, 8]	[2816, 1792, 3968, 1728, 1664, 3328, 2944]	60.9
	M27	7	896	[2, 2, 4, 2, 2, 2, 2]	[3904, 3264, 3328, 3968, 1728, 2624, 4032]	63.5
	M28	7	960	[4, 2, 4, 2, 2, 2, 2]	[3584, 2560, 1792, 1920, 3968, 2112, 3968]	64.1
	M29	8	960	[2, 2, 2, 4, 2, 2, 2, 4]	[3328, 2432, 2624, 2752, 1664, 2240, 2304, 2816]	68.3
	M30	7	960	[4, 2, 4, 2, 2, 2, 2]	[3904, 2304, 2368, 3584, 3264, 2880, 3904]	68.5
	M31	8	960	[4, 2, 4, 2, 2, 4, 2, 4]	[2560, 3648, 2624, 2112, 3328, 2112, 1792, 3328]	70.9
	M32	8	960	[4, 4, 4, 2, 2, 4, 2, 4]	[2560, 2304, 2624, 4032, 2688, 2624, 3840, 2816]	74.7
	M33	9	960	[2, 4, 2, 4, 2, 4, 2, 2, 4]	[3072, 3264, 2944, 1984, 2880, 3520, 2112, 2624, 1728]	79.6
	M34	10	896	[2, 2, 4, 2, 2, 2, 2, 2, 4, 2]	[2816, 3264, 3584, 1792, 3136, 3584, 2240, 2240, 1920, 2752]	81.2
	M35	9	960	[8, 2, 2, 2, 4, 4, 2, 4, 4]	[3904, 3648, 2432, 3136, 3264, 2816, 2240, 3072, 3840]	87.7
	M36	10	960	[4, 4, 2, 2, 4, 4, 2, 4, 4, 2]	[2176, 3264, 2752, 3136, 3968, 3520, 3776, 3328, 1728, 2496]	94.9
	M37	10	960	[4, 2, 4, 2, 2, 2, 2, 4, 2, 2]	[3904, 2112, 2496, 3968, 3968, 2624, 3904, 2304, 3200, 3840]	99.0
	M38	11	960	[4, 2, 2, 4, 2, 4, 2, 2, 4, 4, 4]	[2176, 4032, 3264, 3840, 2688, 1984, 1728, 2944, 1920, 2368, 3840]	99.8

Table 6: Detailed architectural parameters for all models in Figure 4 with GPT-2 backbone.

		n_{layer}	d_{model}	n_{head}	d_{inner}	DecoderParams (M)
baseline		$\in$[1,16]	8	2048	512	-
TITAN Xp	M1	3	256	[2, 2, 2]	[3072, 3776, 3904]	6.3
	M2	2	448	[2, 2]	[3456, 3776]	8.1
	M3	2	448	[2, 4]	[4032, 3904]	8.7
	M4	3	384	[2, 2, 2]	[3072, 2176, 4032]	8.9
	M5	2	576	[2, 2]	[3456, 3584]	10.8
	M6	4	448	[2, 2, 2, 2]	[4032, 3904, 1920, 3072]	14.8
	M7	4	512	[2, 2, 4, 2]	[3904, 3136, 1280, 2624]	15.4
	M8	2	832	[8, 2]	[3456, 3584]	17.3
	M9	2	960	[2, 8]	[3456, 3648]	21.0
	M10	2	960	[2, 2]	[3968, 3584]	21.9
	M11	5	640	[2, 2, 2, 2, 2]	[4032, 2560, 2176, 2304, 3136]	26.4
	M12	3	832	[2, 8, 4]	[3840, 3840, 3776]	27.4
	M13	5	704	[2, 2, 2, 4, 4]	[2368, 3648, 1856, 3712, 3200]	30.8
	M14	3	960	[2, 2, 2]	[3584, 3648, 4032]	32.7
	M15	3	960	[2, 2, 2]	[3904, 3520, 4032]	33.1
	M16	6	640	[2, 2, 2, 2, 2, 2]	[2624, 2560, 2880, 3776, 3648, 3840]	34.6
	M17	4	896	[2, 2, 4, 2]	[4032, 3712, 3328, 3072]	38.2
	M18	5	832	[2, 2, 2, 4, 4]	[3392, 3648, 2880, 3712, 3200]	41.9
	M19	4	960	[2, 2, 4, 2]	[3904, 3136, 3328, 3776]	42.0
	M20	4	960	[8, 8, 2, 4]	[3904, 3712, 4032, 3776]	44.4
	M21	6	832	[2, 2, 4, 2, 2, 2]	[3904, 3456, 4032, 1792, 3072, 2496]	47.9
	M22	5	896	[4, 2, 2, 2, 4]	[3968, 3200, 3840, 3328, 3648]	48.3
	M23	5	960	[2, 2, 2, 2, 2]	[3904, 3264, 3328, 3776, 3392]	52.4
	M24	5	960	[2, 2, 4, 2, 2]	[3584, 3456, 3776, 2944, 4032]	52.7
	M25	5	960	[2, 8, 2, 4, 2]	[3904, 3648, 4032, 3776, 3968]	55.6
	M26	6	960	[8, 8, 2, 2, 2, 2]	[3904, 2560, 2880, 3776, 2240, 3840]	59.1
	M27	6	960	[2, 2, 2, 4, 2, 2]	[2496, 3456, 3328, 3904, 3968, 2944]	60.8
	M28	6	960	[4, 2, 4, 4, 2, 8]	[4032, 3456, 3328, 3776, 4032, 2752]	63.2
	M29	6	960	[2, 2, 2, 4, 4, 4]	[3968, 3648, 3840, 3776, 3584, 2624]	63.4
	M30	7	960	[2, 2, 2, 4, 2, 4, 2]	[3904, 2368, 4032, 3008, 3520, 2944, 2496]	68.7
	M31	7	960	[2, 2, 4, 2, 2, 2, 4]	[3072, 3648, 3520, 3584, 3136, 1984, 3584]	69.1
	M32	7	960	[4, 2, 2, 2, 8, 2, 2]	[3712, 3648, 3584, 3520, 2752, 3008, 3392]	71.2
	M33	8	960	[2, 4, 4, 2, 2, 2, 2, 2]	[3904, 2816, 3072, 1920, 3328, 3456, 2304, 2368]	74.1
	M34	8	960	[2, 2, 2, 4, 2, 2, 8, 2]	[3520, 2368, 4032, 1792, 3200, 3776, 3200, 3648]	78.6
	M35	8	960	[4, 2, 4, 4, 8, 8, 4, 2]	[3520, 3712, 3328, 3776, 3200, 2752, 3200, 2112]	78.7
	M36	8	960	[8, 4, 2, 8, 2, 2, 2, 2]	[3520, 3840, 3328, 3776, 3200, 3776, 3968, 3648]	85.4
	M37	10	960	[2, 8, 2, 4, 2, 2, 4, 2, 8, 8]	[3648, 2560, 3776, 1792, 3968, 2752, 3200, 2368, 4032, 2368]	95.5
	M38	10	960	[2, 4, 2, 2, 4, 2, 4, 2, 4, 8]	[3840, 2240, 3328, 3776, 3648, 3200, 2944, 2368, 3968, 2880]	98.8
	M39	10	960	[2, 4, 2, 2, 2, 2, 4, 2, 4, 8]	[3840, 2240, 3328, 3776, 3200, 3200, 3968, 2368, 3968, 2816]	99.8