Towards Autoregressive Neural Network Parameter Generation

Anonymous authors
Paper under double-blind review

Abstract

Large Language Models (LLMs) have recently established a new paradigm in machine learning, demonstrating exceptional zero-shot generalization and complex reasoning capabilities. Concurrently, neural network parameter generation has emerged as a promising avenue for rapid task adaptation and model synthesis. However, current weight generation techniques struggle with out-of-distribution generalization. Training a unified model on a diverse corpus of pretrained weights remains highly challenging due to the continuous, high-dimensional nature of parameter spaces and the difficulty of conditioning the generation on novel tasks. To mitigate these limitations, we propose a novel, LLM-inspired approach that formulates network parameter generation as an autoregressive modeling problem. By leveraging a decoder-only transformer architecture, our method treats sequential parameter spaces analogously to linguistic tokens, enabling scalable and conditionally accurate weight synthesis. Extensive experiments on multiple vision datasets demonstrate that our method consolidates diverse pretrained models into a single flexible generative framework. The synthesized parameters achieve competitive or superior performance compared with state-of-the-art baselines, with particular advantages in scalability and efficiency when applied to larger architectures.

1 Introduction

The advent of Large Language Models (LLMs) has catalyzed a paradigm shift in machine learning. By leveraging scalable decoder-only transformer architectures (Vaswani et al., 2017) and vast corpora of sequential data, LLMs exhibit remarkable zero-shot generalization and complex reasoning (Brown et al., 2020). A core driver of this success is the autoregressive modeling of discrete tokens, which lets these models capture the syntactic and semantic structure of language with a single, unified objective. In contrast, adapting a neural network to a novel task still relies overwhelmingly on iterative, gradient-based optimization, which is data and compute-hungry and must be repeated from scratch for every new task or architecture. Despite the proliferation of public model repositories (Schürholt et al., 2022c) such as Hugging Face[1] and TIMM (Wightman, 2019), reusing these pretrained models still demands manual selection and extensive fine-tuning.

Prior work in transfer learning, meta-learning, and knowledge distillation (Gou et al., 2021; Yang et al., 2021; Elsken et al., 2019) has predominantly focused on adapting *individual* models, often overlooking the cross-task insights embedded in large-scale model collections. A distinct line of work instead treats trained weights as data: hyper-representation learning (Schürholt et al., 2021; Schürholt et al., 2022b;a; Wang et al., 2024) seeks to learn distributions over network weights to enhance initialization. These methods, however, are typically *unconditional* and limited to single-task scenarios, neglecting the benefits of incorporating pretrained dataset embeddings during training. A few studies have explored task-specific parameter generation (Soro et al., 2024), but a unified and flexible solution remains lacking.

We take inspiration from LLMs to address this gap. If network weights, like language, possess intrinsic and hierarchical structure regularities across layers, channels, and parameter roles analogous to grammar and syntax then autoregressive *sequence* modeling offers a more principled route to weight synthesis than one-shot

[1] https://huggingface.co/

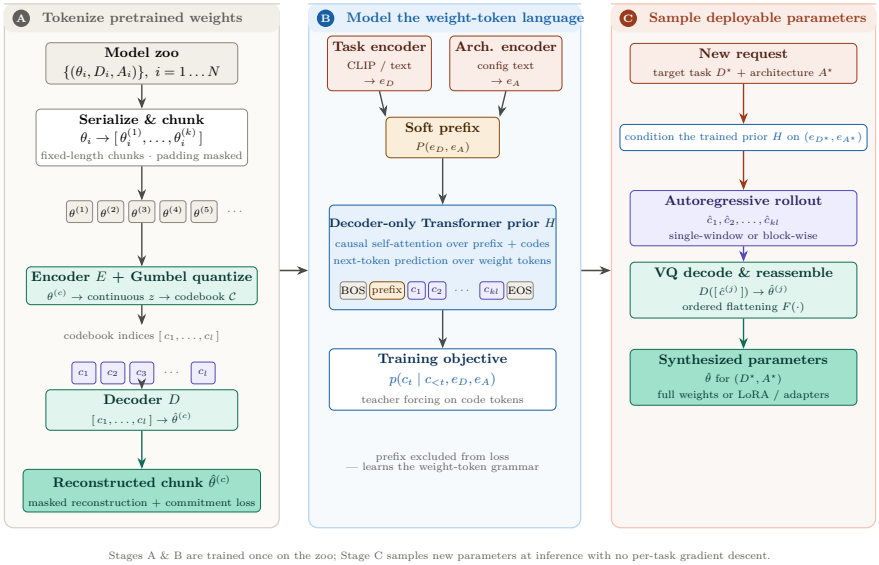

Figure 1: The IGPG pipeline. **(A)** A Gumbel VQ-VAE tokenizes flattened, chunked pretrained weights: the encoder and quantizer map each chunk $\theta^{(c)}$ to codebook indices $[c_1, \dots, c_l]$, and the decoder D reconstructs $\hat{\theta}^{(c)}$ under a masked reconstruction plus commitment objective. **(B)** A decoder-only transformer prior H models the discrete weight sentence (BOS, $c_1, \dots, c_{kl}$, EOS) by next-token prediction, conditioned on a soft prefix $P(e_D, e_A)$ from a task encoder (CLIP) and an architecture encoder; the prefix is excluded from the loss. **(C)** At inference, H is conditioned on a new request $(D^\star, A^\star)$, rolls out codebook tokens autoregressively (single-window or block-wise), and the VQ decoder reassembles them via $F(\cdot)$ into synthesized parameters—full weights or LoRA/adapters.

continuous regression. Serializing the parameter space converts an ill-posed, high-dimensional continuous-generation problem into a tractable, step-by-step conditional one, in which each token is predicted from all preceding context and the target task specification. This formulation directly addresses the missing intra-network dependencies: by construction, every generated parameter token is conditioned on those already produced, preserving inter-layer or chunks coherence.

Building on this view, we show that rather than attempting to generate arbitrary neural networks from unconstrained prompts, we can trained a conditional autoregressive prior over discretized pretrained-weight distributions. By converting continuous parameters into reusable codebook tokens with a Vector Quantized Variational Autoencoder (VQ-VAE) (van den Oord et al., 2017), we propose an instruction guided autoregressive method that enables to, retrieve, and sample task- or architecture-conditioned parameter initializations. This positions autoregressive weight-token modeling as a practical step toward foundation models for neural parameter generation.

The framework jointly encodes three elements: task representations, using dataset embeddings or natural-language instructions to capture target-task semantics; architecture specifications, explicitly representing network designs to enable cross-architecture generation; and inter-layer/chunks dependencies, employing autoregressive modeling to preserve coherence across layers. Concretely, a VQ-VAE compresses high-dimensional weights into a discrete codebook, and a decoder-only transformer then models the resulting token sequence autoregressively. This joint conditioning lets a single generator synthesize coherent, task-optimized parameters across heterogeneous tasks and architectures, reducing reliance on extensive fine-tuning.

Our contributions are as follows:

- **Task-conditioned generation.** We generate network parameters directly from dataset descriptors and natural-language instructions, providing intuitive task control over weight synthesis.

- **Architecture-agnostic framework.** A single model generates parameters across diverse architectures by conditioning on textual architecture specifications, leveraging knowledge pooled from many pretrained models.
- **Autoregressive coherence.** By modeling layer-wise dependencies as a token sequence, the framework produces internally consistent parameters that accelerate convergence and improve transfer.
- **Extensive evaluation.** The synthesized parameters achieve competitive performance across small-to-medium vision architectures and show promising scalability through chunk-wise tokenization..

2 Related Work

Neural network parameter generation. Approaches to neural network parameter generation can be broadly grouped into hypernetwork-based synthesis and generative modeling over weight space. Hypernetwork methods generate parameters directly, ranging from early evolutionary and neural generation schemes (Stanley & Miikkulainen, 2002; Deutsch, 2018; Ratzlaff & Fuxin, 2019) to graph-based (Zhang et al., 2019; Knyazev et al., 2021) and transformer-based formulations (Zhmoginov et al., 2022). Although these methods establish that neural networks can be produced by other neural networks, the generated parameters are typically used as initializations that still require downstream optimization.

A complementary line of work instead learns representations or distributions over *pretrained* weights. This includes adaptive transfer from model zoos (Shu et al., 2021), hyper-representations of pretrained model populations (Schürholt et al., 2022b), and diffusion-based generation of network checkpoints (Peebles et al., 2022). Collectively, these results demonstrate that pretrained weights contain reusable structure that can be exploited for sampling and transfer. Our work adopts this perspective but departs from prior formulations by casting parameter generation as autoregressive sequence modeling over discrete weight tokens, which supports conditional generation while explicitly capturing dependencies within and across layers.

Applications and implications. Generative modeling of pretrained parameters has direct implications for efficient model adaptation. By sampling task-adapted parameters rather than training from scratch, it can reduce the cost of transferring models to new tasks. It also provides a mechanism for producing compact, task-specific adaptations, including parameter-efficient modules such as LoRA (Huang et al., 2023; Addair & Rishi, 2024). Retrieval-based approaches have explored selecting or composing LoRA experts for downstream adaptation (Zhao et al., 2024), and recent work on LLM-guided model generation indicates that task or instruction information can steer model synthesis (Tang et al., 2024). In contrast, our method learns an instruction-conditioned autoregressive prior over tokenized parameters, enabling scalable sampling of either full weights or adaptation modules from pretrained weight distributions.

3 Method

We now describe the ptoposed method, which we refer to as Instruction-Guided Parameter Generation (IGPG). We first set up the problem and give a high-level overview (Section 3.1), then detail the two stages: a VQ-VAE weight tokenizer (Section 3.2) and a conditional autoregressive transformer prior (Section 3.3). Figure 1 illustrates the full pipeline and Figure 2 the token format.

3.1 Problem Setup

Let $\mathcal{D}$ denote the space of datasets/tasks, $\mathcal{A}$ the space of architectures, and Θ the pretrained parameter space. Given a set of pretrained models(model zoo) $\{\theta_i\}_{i=1}^N$ with corresponding descriptions $\{\mathcal{D}_i\}$, we learn a generative mapping H that operates in two phases: during training $H : \mathcal{D} \times \mathcal{A} \times \Theta \to \Theta$, and at inference $H : \mathcal{D} \times \mathcal{A} \to \Theta$. Thus, given a dataset $\mathcal{D}_i$ and architecture $\mathcal{A}_i$, the trained mapping produces a tailored initialization $\hat{\theta}_i = H(\mathcal{D}_i, \mathcal{A}_i)$. We construct a model zoo by vectorizing each network's parameters. In the *layer-wise* setting, each layer's weights and biases are flattened into a single vector, yielding per-layer samples across all networks. In the *architecture-wise* setting, the flattened layers of a model are concatenated in order

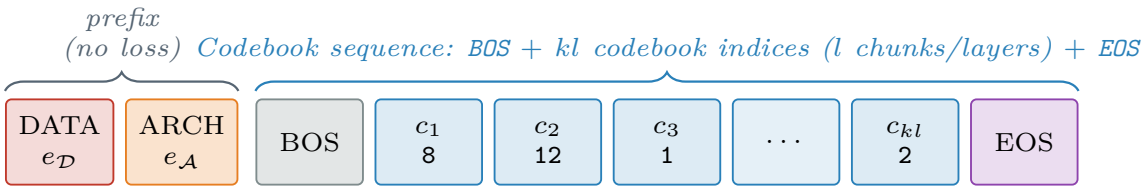

Figure 2: Token format (one sequence per model). A continuous conditioning prefix dataset embedding $e_{\mathcal{D}}$ and, when the zoo spans architectures, an architecture embedding $e_{\mathcal{A}}$ is followed by a `BOS` token, the kl VQ-VAE codebook indices (grouped internally into l chunks/layers with$k$ code each but not individually delimited), and a final `EOS`. The transformer is trained by next-token prediction with the prefix and padding excluded from the loss.

into one global vector per model. Both produce uniform representations $\Theta \in \mathbb{R}^D$ suitable for the generative mapping.

3.2 Weight Tokenization with a Gumbel VQ-VAE

As the first stage, we encode the vectorized parameters with a Gumbel VQ-VAE (van den Oord et al., 2017). The autoencoder operates on fixed-length inputs, so we must first map each parameter vector $\theta \in \mathbb{R}^L$ to a fixed-size representation. Two regimes arise depending on the model zoo. When the zoo is homogeneous or small models sharing a single architecture and hence a common parameter count L padding every vector to one fixed dimension $L_{\max} = \max_i L_i$ suffices, and the encoder consumes each model as a single input. When the zoo instead contains large architectures, or architectures with differing parameter counts, a single fixed dimension is impractical: it would force $L_{\max}$ to the largest model and waste capacity on padding for the rest. In this case we instead apply *chunking* splitting each vector into fixed-size pieces so that models of different sizes are represented by different *numbers* of equally sized chunks. Concretely, θ is padded to length $L' = \lceil L/K \rceil K$ and split into $n = L'/K$ chunks $\{\theta^{(1)}, \ldots, \theta^{(n)}\}$ of size K, which are tokenized independently and later reassembled in the original parameter ordering. Padding is the special case $K = L_{\max}$, $n = 1$.

The autoencoder comprises an encoder $\mathbf{E}$, a decoder $\mathbf{D}$, and a quantizer $\mathbf{Q}$ with a learned codebook $e = \{e_1, \ldots, e_m\}$ of m entries. The encoder maps a chunk $\theta^{(c)}$ to a continuous latent $z = \mathbf{E}(\theta^{(c)})$, whose components act as logits $\pi_1, \ldots, \pi_m$ over the codebook. Rather than select the single nearest entry, $\mathbf{Q}$ draws a soft assignment via Gumbel-Softmax sampling, producing a quantized latent z_q as a convex combination of codebook vectors:

$$z_q = \sum_{j=1}^{m} y_j e_j, \qquad y_j = \frac{\exp((\log \pi_j + g_j)/\tau)}{\sum_{i=1}^{m} \exp((\log \pi_i + g_i)/\tau)}, \tag{1}$$

where g_j are i.i.d. Gumbel noise samples and τ is a temperature controlling the sharpness of the assignment. The decoder reconstructs the chunk as $\hat{\theta}^{(c)} = \mathbf{D}(z_q)$, and the model minimizes

$$\mathcal{L} = \underbrace{\|\mathcal{M} \odot (\theta^{(c)} - \hat{\theta}^{(c)})\|_2^2}_{\text{reconstruction}} + \gamma \underbrace{\|z - \mathrm{sg}[z_q]\|_2^2}_{\text{commitment}} + \beta \underbrace{\|\mathrm{sg}[z] - z_q\|_2^2}_{\text{codebook}}, \tag{2}$$

where $\mathcal{M}$ masks padding values, $\mathrm{sg}[\cdot]$ is the stop-gradient, and $\{\beta, \gamma\}$ balance the terms. This stochastic, differentiable quantization prepares parameter vectors for subsequent autoregressive modeling. In the homogeneous, single-chunk case the same objective applies with $n = 1$.

3.3 Conditional Autoregressive Modeling

As the second stage, we model the discrete token sequences with a decoder-only transformer prior conditioned on the target task and, when it varies across the zoo, the architecture. For a labeled dataset $\mathcal{D}$, we draw a balanced subset of reference images (five per class) and embed each with CLIP (Radford et al., 2021); the resulting per-class embeddings form a conditioning set $\bar{e}_{\mathcal{D}}$. The subset is *re-sampled* each time an example is drawn during training, which acts as conditioning augmentation and prevents the model from memorizing

Algorithm 1 Autoregressive Parameter Generation

Require: architecture embedding $e_{\mathcal{A}}$, dataset embedding $e_{\mathcal{D}}$, parameter length L, chunk size K, tokens-per-chunk l, context limit $N_{\max}$, context width w
Ensure: parameters $\theta_{\mathcal{A}} \in \mathbb{R}^L$

$k \leftarrow \lceil L/K \rceil; \quad T \leftarrow k \cdot l$ ▷ token-sequence length
if $T \leq N_{\max}$ **then**
  $\boldsymbol{s} \leftarrow \mathcal{H}(e_{\mathcal{A}}, e_{\mathcal{D}})$ ▷ single-pass rollout
else
  $J \leftarrow \lceil T/N_{\max} \rceil$
  $\boldsymbol{s}^{(1)} \leftarrow \mathcal{H}(e_{\mathcal{A}}, e_{\mathcal{D}})$
  for $j = 2$ to J **do**
    $\boldsymbol{s}^{(j)} \leftarrow \mathcal{H}\big(e_{\mathcal{A}}, e_{\mathcal{D}}, \boldsymbol{s}_{\text{ctx}}^{(j-1)}\big)$ ▷ $\boldsymbol{s}_{\text{ctx}}^{(j-1)}$: last w tokens of block $j-1$
  end for
  $\boldsymbol{s} \leftarrow \text{concat}(\boldsymbol{s}^{(1)}, \ldots, \boldsymbol{s}^{(J)})_{1:T}$
end if
split $\boldsymbol{s}$ into $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_k$, each of length l
$\theta_{\mathcal{A}} \leftarrow \mathcal{F}\big([\mathbf{D}(\boldsymbol{s}_1); \ldots; \mathbf{D}(\boldsymbol{s}_k)]\big)$ ▷ decode and flatten
return $\theta_{\mathcal{A}}$

a fixed prompt. When the zoo spans multiple architectures, we additionally render each architecture to a textual description $\text{desc}(\mathcal{A})$, encode it with an instruction-tuned LLM (llama 3 team, 2024) to obtain $e_{\mathcal{A}}$, and include it in the conditioning; when the architecture is fixed (as in the homogeneous tiny-model zoos experiments), this term is omitted and conditioning reduces to $\bar{e}_{\mathcal{D}}$.

Rather than fuse the conditioning into the discrete vocabulary, we adopt a *prefix-LM* design (Radford et al., 2019; Esser et al., 2020): the continuous conditioning embeddings are mapped into the transformer width by a small projection and placed as a soft prefix *before* the token sequence, where causal self-attention can read them. This mirrors prompt/prefix tuning in language models and exactly matches how the model is queried at inference—supply a task, then roll out a codebook—without a separate encoder or cross-attention.

Let $c = (c_1, \ldots, c_T)$ denote the codebook-index sequence of a model ($T = kl$). We form the teacher-forcing input by prepending a beginning-of-sequence token, $(\texttt{BOS}, c_1, \ldots, c_T)$, and the target by appending an end-of-sequence token, $(c_1, \ldots, c_T, \texttt{EOS})$, so the model predicts each position from all preceding ones. The conditioning prefix occupies positions ahead of BOS and is never a prediction target. Writing $e = (\bar{e}_{\mathcal{D}}, e_{\mathcal{A}})$ for the conditioning, the prior is trained by minimizing the masked next-token cross-entropy

$$\mathcal{L}_{\text{prior}} = -\,\mathbb{E}_{(c,e)} \sum_{t=1}^{T+1} \log p_\theta(\tilde{c}_t \mid \tilde{c}_{<t},\, e)\,, \qquad p_\theta(c \mid e) = \prod_{t=1}^{T} p_\theta(c_t \mid c_{<t}, e)\,, \tag{3}$$

where $\tilde{c} = (c_1, \ldots, c_T, \texttt{EOS})$ is the target sequence and the loss is taken only over code and EOS positions (padding is masked out). The factorization on the right makes explicit that each parameter token is generated conditioned on those already produced, so the prior captures the inter-layer dependencies exposed by the architecture-wise serialization of Section 3.1. Because the sequence is short and the corpus modest, a compact transformer (tens of millions of parameters) suffices and trains quickly.

3.4 Generation and Scaling

Consider a target architecture $\mathcal{A}$ whose parameters form a vector $\theta_{\mathcal{A}} \in \mathbb{R}^L$. As in Section 3.2, we split $\theta_{\mathcal{A}}$ into $k = \lceil L/K \rceil$ chunks of size K; the learned tokenizer $\mathcal{T} : \mathbb{R}^K \to \mathcal{V}^l$ maps each chunk to l codebook tokens, so the full vector corresponds to a token sequence of length kl. The decoder $\mathbf{D} : \mathcal{V}^l \to \mathbb{R}^K$ recovers real-valued chunks, and a flattening operator $\mathcal{F}$ reassembles the k decoded chunks, in order, into $\theta_{\mathcal{A}}$.

Let $N_{\max}$ be the maximum sequence length seen during training, and let $\mathcal{H}$ denote the trained autoregressive generator conditioned on $(e_{\mathcal{A}}, e_{\mathcal{D}})$. Two regimes arise, mirroring the tokenization regimes of Section 3.2.

Single pass ($kl \leq N_{\max}$). This is the homogeneous case with same architecture, or models small enough to fit in one window, as in the tiny-model zoos. The generator emits the whole sequence $\boldsymbol{s} = \mathcal{H}(e_{\mathcal{A}}, e_{\mathcal{D}}) \in \mathcal{V}^{kl}$ in one rollout. Splitting $\boldsymbol{s}$ into k segments $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_k$ of length l, decoding each with $\mathbf{D}$, and flattening with $\mathcal{F}$ yields $\theta_{\mathcal{A}}$.

Chunk-wise ($kl > N_{\max}$). For large architectures the sequence exceeds the context window, so $\mathcal{H}$ cannot emit all kl tokens at once. We instead generate $J = \lceil kl/N_{\max} \rceil$ blocks autoregressively: an initial block $\boldsymbol{s}^{(1)} \in \mathcal{V}^{N_{\max}}$, then $\boldsymbol{s}^{(2)}, \ldots, \boldsymbol{s}^{(J)}$, each conditioned on $(e_{\mathcal{A}}, e_{\mathcal{D}})$ and a context window carried over from the previous block. Concatenating the blocks and truncating to the first kl tokens recovers the full sequence, which is then decoded and flattened as above. Algorithm 1 summarizes both regimes.

By limiting each step to $N_{\max}$ tokens, the chunked scheme scales generation to arbitrarily large architectures while preserving the model's capacity to represent high-dimensional parameter vectors; the single-pass case is its natural specialization when one window suffices.

4 Experiments

Implementation. All experiments run on a single NVIDIA V100 GPU. We train with AdamW and a linear learning-rate schedule starting from 1×10^{-4}, using a cyclical Gumbel-Softmax temperature annealed from 1 to 1×10^{-4}. We evaluate on pretrained models gathered from public repositories spanning diverse architectures and datasets. Instructions consist of textual architecture descriptions combined with dataset embeddings.

4.1 Benchmarking on the Tiny Model Zoo

We first evaluate on the Tiny Model Zoo (Schürholt et al., 2022c), comprising compact CNNs trained on MNIST, SVHN, CIFAR-10, and STL-10. We use a 2-layer CNN (2,464 parameters) for MNIST and SVHN and a 3-layer CNN (10,853 parameters) for CIFAR-10 and STL-10. Following prior work, pretrained weights are drawn from epochs 21–25 of a 50-epoch schedule, with 70/15/15 train/validation/test splits. Unlike methods that require a separate model per dataset (Schürholt et al., 2022; Sch"urholt et al., 2024), our framework learns a single generator across all architectures and tasks. We compare against training from scratch, SKDE30 Schürholt et al. (2022), SANE with a KDE prior and its subsampled variant (Sch"urholt et al., 2024), and the diffusion-based D2NWG (Soro et al., 2024).

Table 1: Comparison of weight initialization methods trained on pretrained from epochs 21–25. We compare: (1) training from scratch (tr. fr. scratch), (2) S_{KDE30} (Schürholt et al., 2022b), (3) SANE with $KDE30$, (4) subsampled SANE_{SUB} (aligned with IGPG), and (5) D2NWG (Soro et al., 2024).

Ep.	Method	MNIST	SVHN	CIFAR-10	STL
0	tr. fr. scratch	∼10 /%	∼10 /%	∼10 /%	∼10 /%
	S_{KDE30}	68.6±6.7	54.5±5.9	*n/a*	*n/a*
	$SANE_{KDE30}$	84.8±0.8	70.7±1.4	56.3±0.5	39.2±0.8
	$SANE_{SUB}$	**86.7±0.8**	**72.3±1.6**	57.9±0.2	43.5±1.0
	D2NWG	80.52±0.82	66.6±0.7	58.80±0.1	44.50±0.1
	IGPG	83.20±0.01	67.10±0.4	**58.3±0.1**	**44.41±0.1**
1	tr. fr. scratch	20.6±1.6	19.4±0.6	37.2±1.4	21.3±1.6
	S_{KDE30}	83.7±1.3	69.9±1.6	*n/a*	*n/a*
	$SANE_{KDE30}$	85.5±0.8	71.3±1.4	58.2±0.2	43.5±0.7
	$SANE_{SUB}$	**87.5±0.6**	73.3±1.4	**59.1±0.3**	44.3±1.0
	D2NWG	87.8±0.4	73.6±1.3	59.2±0.3	44.8±0.2
	IGPG	86.10±0.1	**74.0±0.3**	58.7±0.3	**44.9±0.1**
5	tr. fr. scratch	36.7±5.2	23.5±4.7	48.5±1.0	31.6±4.2
	S_{KDE30}	92.4±0.7	57.3±12.4	*n/a*	*n/a*
	$SANE_{KDE30}$	87.5±0.7	72.2±1.2	58.8±0.4	45.2±0.6
	$SANE_{SUB}$	89.0±0.4	73.6±1.5	59.6±0.3	45.3±0.9
	D2NWG	**92.5±0.9**	74.0±0.1	60.3±0.1	45.4±0.1
	IGPG	91.7±0.8	**74.5±0.5**	**60.3±0.1**	**45.7±0.1**
25	tr. fr. scratch	83.3±2.6	66.7±8.5	57.2±0.8	44.0±1.0
	S_{KDE30}	93.0±0.7	74.2±1.4	*n/a*	*n/a*
	$SANE_{KDE30}$	92.0±0.3	74.7±0.8	60.2±0.6	48.4±0.5
	$SANE_{SUB}$	92.3±0.4	75.1±1.0	61.2±0.1	48.0±0.4
	D2NWG	**96.2±0.3**	75.7±0.5	**64.1±1.0**	48.7±0.5
	IGPG	94.5±0.1	**76.9±0.1**	63.9±0.0	**49.06±0.2**
50	tr. fr. scratch	91.1±2.6	70.7±8.8	61.5±0.7	47.4±0.9

Results. Table 1 shows that our method outperforms the sequential generation of SANE (Sch"urholt et al., 2024) while matching the convergence of the diffusion-based D2NWG.

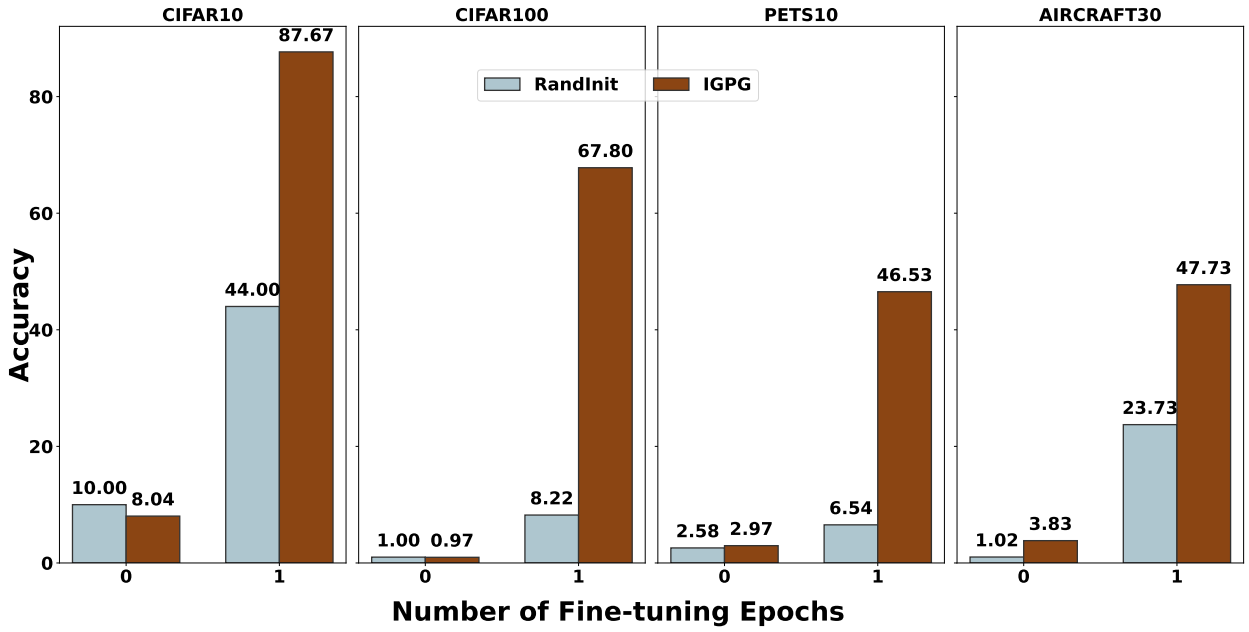

Figure 3: Transfer learning evaluation on novel datasets: CIFAR100, CIFAR10, Aircraft30, and PETS10 compared to random initialization.

Crucially, it preserves both zero-shot accuracy and fine-tuning performance relative to prior work, generating robust initial weights that adapt efficiently across multiple architectures and datasets within a single unified framework.

4.2 Transfer Learning on Unseen Datasets

We next train a single generator on 30 diverse datasets from Meta-Album (Ullah et al., 2022), spanning a broad range of class distributions, and sample parameters for CIFAR-10, CIFAR-100, Oxford Pets (PETS10), and Aircraft (AIRCRAFT30). Training and target datasets are disjoint to ensure a fair assessment. Since the architecture is fixed (a MobileNetV3 subnet from OFA, with 224-pixel inputs), we encode only dataset information.Table 8 lists the training image datasets. As shown in Figure 3, sampled models begin near random-initialization performance but improve by over 50% relative within a single fine-tuning epoch. e.g. CIFAR-10 rises from 8.0% to 87.7% and CIFAR-100 from 1.0% to 67.8% after one epoch. This underscores the advantage of leveraging broad pretraining data: although zero-shot performance starts near baseline, it rapidly surpasses random initialization, reflecting the framework's ability to generate meaningful parameters. IGPG does not solve direct zero-shot weight synthesis in this setting, but it provides an initialization that greatly accelerates early adaptation

4.3 Cross-Architecture Benchmarking

We evaluate cross-architecture generation on CIFAR-10 using 125 randomly sampled ResNet-56 variants spanning 200k–700k parameters, with block configurations from $[4, 4, 4]$ to $[8, 8, 8]$. Each variant is trained for 100 epochs, and we collect the last five epochs' weights as training data. We set the maximum token-sequence length to 768 and train a VQ-VAE to encode these parameters, then condition a GPT-2 prior on instruction templates (preprocessed with an instruction-tuned LLaMA-3 model) to generate codebook tokens. For autoregressive pretraining on ResNet model zoos built from a single dataset, we omit dataset conditioning because the dataset identity is shared by all training samples. The model is therefore conditioned only on the architecture, specified by the ResNet block configuration. Under this setting, the autoregressive prior learns to generate the sequence of discrete VQ-VAE codebook indices associated with the target model weights. The prompt format is illustrated below:

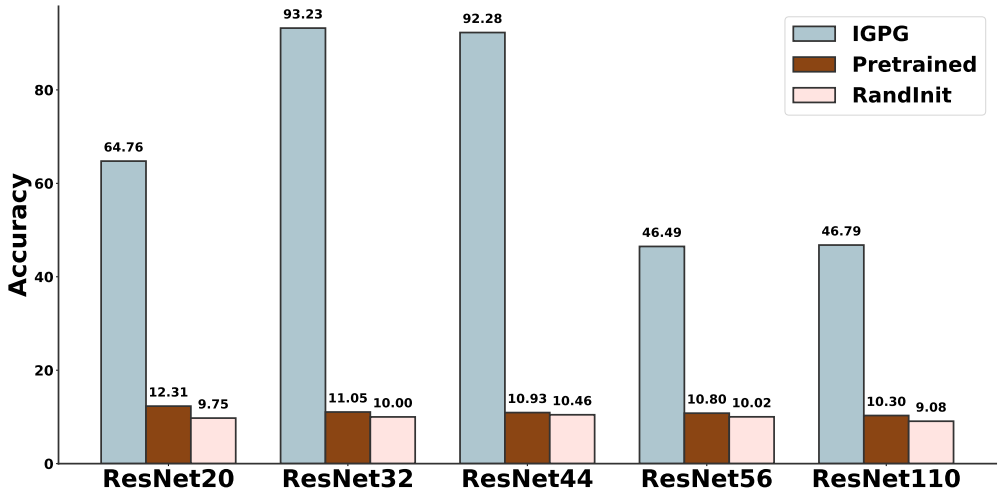

Figure 4: Performance evaluation with seen and unseen ResNet architectures on CIFAR-10 against models pretrained on CIFAR-100 and Random Initialization.

Remark 1. ***Instruction:*** *You are a codebook generator.* ***Input:*** *Generate the full codebook sequence of length* 512 *for a ResNet with* `num_blocks` $= [6, 7, 6]$. ***Output:*** $[995, 66, 750, 391, 809, 830, \ldots]$.

We test five ResNet architectures: two in-distribution variants (directly sampled) and three out-of-distribution (ResNet-20, ResNet-56, ResNet-110). On in-distribution architectures, generated weights match pretrained baselines; on out-of-distribution networks, our method attains up to 64% on ResNet-20 and 46% on both ResNet-56 and ResNet-110, outperforming random initialization and CIFAR-100-pretrained weights. These results highlight strong cross-architecture generalization without additional fine-tuning.

4.4 Weight Retrieval Across Heterogeneous Architectures

We further test versatility by jointly encoding 19 publicly available architectures from GitHub[2] spanning ResNet, VGG, MobileNetV2, ShuffleNetV2, and RepVGG pretrained on CIFAR-10 and CIFAR-100, with parameter counts from 0.27M to 27M (over 100× range). We train a VQ-VAE with a chunk size of ∼256k parameters, each encoded into 64 tokens; for the largest models (roughly 10 chunks) this yields a maximum sequence length of 640 tokens. We condition on both architecture descriptions and dataset encodings (a CLIP image encoder over five images per class) Table 2 reports per-family averages: generated weights track pretrained accuracy almost exactly (global averages of 93.74 vs. 93.76 on CIFAR-10 and 73.36 vs. 73.42 on CIFAR-100), with near-perfect Pearson correlation between generated and pretrained accuracy (0.9999 on CIFAR-10, 0.9991 on CIFAR-100). This confirms the framework's capacity to learn from diverse architecture–dataset pairs and faithfully approximate original performance, pointing to its potential for model selection and fast adaptation.

4.5 Extension to LoRA Parameter Generation

To show the framework can learn distributions of diverse low-rank adapters, we evaluate on six vision benchmarks: Oxford-IIIT Pets, Stanford Cars, CIFAR-10, EuroSAT, DTD, and FGVC Aircraft. We fine-tune a ViT-Base with LoRA modules following Gao et al. (2024), retain the top-five performing models per dataset, learn their parameter distribution, and generate novel LoRA parameters.

Table 3 shows the generated LoRA parameters deliver up to a 10% average improvement over the FourierFT baseline (Gao et al., 2024) at the same parameter budget (72K), with especially large gains on DTD and Stanford Cars. This demonstrates the value of distribution learning for uncovering higher-performing adapter configurations within the existing parameter space.

[2] https://github.com/chenyaofo/pytorch-cifar-models

Table 2: Performance of weights generation from diverses models pretrained on CIFAR-10 and CIFAR-100 datasets

Model	CIFAR-10			CIFAR-100		
	IGPG	Pretrained	Params (M)	IGPG	Pretrained	Params (M)
MobileNetV2 (×0.5)	93.16	93.12	0.70	71.05	71.14	0.82
MobileNetV2 (×0.75)	94.16	94.08	1.37	74.15	74.06	1.48
MobileNetV2 (×1.0)	93.98	94.05	2.24	74.41	74.30	2.35
MobileNetV2 (×1.4)	94.26	94.22	4.33	75.97	76.29	4.50
Average (MobileNetV2)	**93.89**	**93.87**	**2.16**	**73.90**	**73.95**	**2.29**
RepVGG-A0	94.44	94.47	7.84	75.32	75.29	7.96
RepVGG-A1	94.93	94.93	12.82	76.51	76.45	12.94
RepVGG-A2	95.36	95.27	26.82	77.52	77.50	26.94
Average (RepVGG)	**94.91**	**94.89**	**15.83**	**76.45**	**76.41**	**15.95**
ResNet-20	92.47	92.59	0.27	68.92	68.84	0.28
ResNet-32	93.47	93.53	0.47	69.98	70.15	0.47
ResNet-44	93.92	94.01	0.66	71.39	71.64	0.67
ResNet-56	94.56	94.37	0.86	72.43	72.60	0.86
Average (ResNet)	**93.61**	**93.63**	**0.57**	**70.68**	**70.81**	**0.57**
ShuffleNetV2 (×0.5)	90.61	90.65	0.35	67.90	67.80	0.44
ShuffleNetV2 (×1.0)	92.75	93.30	1.26	72.55	72.59	1.36
ShuffleNetV2 (×1.5)	93.58	93.57	2.49	74.22	74.22	2.58
ShuffleNetV2 (×2.0)	94.02	93.99	5.37	75.39	75.49	5.55
Average (ShuffleNetV2)	**92.74**	**92.88**	**2.37**	**72.52**	**72.53**	**2.48**
VGG11-BN	92.85	92.78	9.76	70.76	70.78	9.80
VGG13-BN	94.01	94.00	9.94	74.40	74.62	9.99
VGG16-BN	94.19	94.15	15.25	73.98	74.00	15.30
VGG19-BN	94.09	93.91	20.57	73.82	73.87	20.61
Average (VGG)	**93.79**	**93.71**	**13.88**	**73.24**	**73.32**	**13.93**
Global Average (All Models)	**93.74**	**93.76**	**6.96**	**73.36**	**73.42**	**7.00**

Table 3: Performance evaluation on LoRA weights generation conditioned on the dataset

Model	Method	# Trainable Parameters	OxfordPets	StanfordCars	CIFAR10	DTD	EuroSAT	FGVC	Average
ViT-Base	LP	-	$90.28_{\pm 0.43}$	$25.76_{\pm 0.28}$	$96.41_{\pm 0.02}$	$69.77_{\pm 0.67}$	$88.72_{\pm 0.13}$	$17.44_{\pm 0.43}$	64.73
	FF	85.8M	$93.14_{\pm 0.40}$	$\mathbf{79.78}_{\pm 1.15}$	$\mathbf{98.92}_{\pm 0.05}$	$77.68_{\pm 1.21}$	$\mathbf{99.05}_{\pm 0.09}$	$\mathbf{54.84}_{\pm 1.23}$	**83.90**
	LoRA	581K	$93.19_{\pm 0.36}$	$45.38_{\pm 0.41}$	$98.78_{\pm 0.05}$	$74.95_{\pm 0.40}$	$98.44_{\pm 0.15}$	$25.16_{\pm 0.16}$	72.65
	FourierFT(Gao et al., 2024)	72K	$\mathbf{93.21}_{\pm 0.26}$	$46.11_{\pm 0.24}$	$98.58_{\pm 0.07}$	$75.09_{\pm 0.37}$	$98.29_{\pm 0.04}$	$27.51_{\pm 0.64}$	73.13
	IGPG	72K	$92.84_{\pm 0.30}$	$57.67_{\pm 0.52}$	$98.45_{\pm 0.95}$	$\mathbf{88.74}_{\pm 0.08}$	$98.70_{\pm 0.23}$	$36.63_{\pm 0.11}$	78.83

4.6 Learning Distributions over Models Pretrained on Diverse Datasets Size

Finally, we collect ViT-Small and MobileNetV3-Small weights pretrained on datasets ranging from small to large scale, including ImageNet-1k. We train the VQ-VAE on the combined weights and the transformer on its codebook using five samples per class and architecture descriptions as instructions, then evaluate both reconstruction and autoregressive sampling in-distribution.

Table 4 shows that both VQ-VAE reconstruction and sampled weights attain accuracy nearly indistinguishable from the original pretrained models across all datasets, with the best samples often matching or slightly

Table 4: Learning distribution of combined ViT-Small (CIFAR-10, CIFAR-100, CINIC-10, SVHN, Tiny-ImageNet) and MobileNetV3-Small (ImageNet). We report model size (Parameters), Pretrained accuracy, VQVAE Reconstruction accuracy, and the best among five samples (Best Sampled Weights).

Dataset	Architecture	Parameters	Pretrained (%)	VQVAE Reconstruction (%)	IGPG(%)
CIFAR-10	ViT-Small	2697610	93.67	93.67	**93.70**
CIFAR-100	ViT-Small	2714980	72.29	72.28	**72.30**
CINIC-10	ViT-Small	2697610	83.23	83.23	**83.24**
SVHN	ViT-Small	2697610	97.74	97.74	**97.76**
Tiny-ImageNet	ViT-Small	2771144	53.44	53.44	**53.45**
ImageNet	MobileNetV3-Small	2542856	**67.67**	67.66	67.65

Table 5: Performance comparison of conditional sampling with chunk-based and chunk-free approach. **C-10** and **C-100** refer to respectively CIFAR-10 and CIFAR-100 datasets

		PRETRAINED		CHUNK-FREE		CHUNK-BASED		
MODELS	#PARAMS	TOP-1	TOP-5	TOP-1	TOP-5	TOP-1	TOP5	DATASETS
RESNET-56	860026	94.37	**99.83**	**94.40**±0.01	99.82±0.00	94.19±0.01	99.87± 0.00	C-10
RESNET-56	865876	**72.63**	91.94	72.45±0.01	**91.96±0.01**	72.41 ± 0.2	91.80 ±0.03	C-100
MOBILENETV2_X0_5	834324	70.88	91.72	**71.14**±0.01	92.48±0.04	70.93 ±0.00	**92.55 ±0.02**	C-100

exceeding the baseline. This confirms the framework preserves performance while offering the flexibility to sample diverse parameter configurations.

4.7 Ablation Study

Full Model vs Layer-Wise Sampling We compare two parameter generation approaches: full model-wise encoding and layer-wise encoding. In layer-wise encoding, we assemble parameters from each layer into separate training datasets, applying chunking and padding per layer. While both methods perform well on in-distribution parameters, layer-wise encoding shows superior generalization to novel architectures, suggesting better adaptability and robustness.

Impact of Chunking Strategy We evaluate the effect of chunking network weights versus using complete weight vectors. Using pretrained weights from PyTorchHub[3], we assess ResNet56 and MobileNetV2 on CIFAR-10 and CIFAR-100. Results in Table 5 show that while chunking offers no significant advantage for medium-sized architectures, it becomes crucial for larger models where chunk-free approaches struggle to maintain performance.

Can Pretrained LLMs Generate Weight Codebooks? We investigate whether pretrained language models can be adapted to generate neural network parameters through discrete VQ-VAE codebook sequences. We fine-tune LLaMA-3.2-1B-Instruct with LoRA and train GPT-2 for codebook-sequence generation. The results are mixed. LLaMA-3.2-1B-Instruct reproduces the initial codebook tokens accurately, but its generations become unstable as the sequence length increases. GPT-2 with greedy decoding, corresponding to top-(k) sampling with (k=1), can match pretrained codebooks for small parameter vectors when trained fromch scratch on the codebook. These results suggest that directly adapting existing pretrained LLMs to weight-codebook generation remains challenging: the models often fail to maintain long-range code consistency and may emit spurious natural-language tokens instead of valid codebook indices, with errors compounding over long rollouts.

Limitations and Future Work

Although our results are promising, the current study is limited to relatively small architectures. Scaling autoregressive parameter generation to larger and more heterogeneous models remains an open challenge. Our method also does not yet reach the broad generalization ability observed in LLMs, partly because

[3] https://github.com/chenyaofo/pytorch-cifar-models

weight tokens lack the natural semantic ordering of text. A second limitation is the reliance on a VQ-VAE tokenizer. While discretization enables autoregressive modeling, the tokenizer can bottleneck reconstruction and generation quality. Future work should explore alternative tokenization schemes, such as direct bfloat16 or integer-based representations. We also found that instruction-tuned chat models are difficult to adapt directly for codebook generation. For example, LoRA tuning an instruction-tuned LLaMA-3.2-1B often produced short or inconsistent codebooks, whereas GPT-2-small trained on codebook sequences and then instruction-tuned generated valid codebooks. Future work will study stronger pretraining strategies and broader autoregressive backbones for neural parameter generation.

5 Conclusion

We presented an LLM-inspired approach for neural network parameter generation that combines a VQ-VAE tokenizer with a conditional autoregressive transformer prior. By treating serialized network weights analogously to linguistic tokens, we show that pretrained neural network weights can be discretized into reusable codebook tokens and modeled with a conditional autoregressive prior. This enables a single model to reconstruct, retrieve, and sample task/architecture-conditioned initializations that accelerate adaptation and approximate pretrained performance across several weight-space learning settings. Across tiny CNN zoos, transfer to unseen datasets, cross-architecture ResNet generation, LoRA synthesis, and large multi-dataset model collections, the synthesized weights match or exceed strong baselines and converge faster than random initialization, while scaling efficiently to large architectures. These results position autoregressive token modeling as a scalable paradigm for compressing and transferring the collective knowledge of pretrained model repositories, advancing neural architecture search and transfer learning.

References

Justin Zhao Timothy Wang Wael Abid Geoffrey Angus Arnav Garg Jeffery Kinnison Piero Molino Travis Addair and Devvret Rishi. Lora land: 310 fine-tuned llms that rival gpt-4, a technical report, April 2024. URL https://predibase.com/blog/lora-land-fine-tuned-open-source-llms-that-outperform-gpt-4.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901, 2020.

Lior Deutsch. Generating neural networks with neural networks. *ArXiv*, abs/1801.01952, 2018.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *ArXiv*, abs/1808.05377, 2019.

Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis, 2020.

Ziqi Gao, Qichao Wang, Aochuan Chen, Zijing Liu, Bingzhe Wu, Liang Chen, and Jia Li. Parameter-efficient fine-tuning with discrete fourier transform, 2024.

Jianping Gou, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 129(6):1789–1819, mar 2021.

Chengsong Huang, Qian Liu, Bill Yuchen Lin, Tianyu Pang, Chao Du, and Min Lin. Lorahub: Efficient cross-task generalization via dynamic lora composition, 2023.

Boris Knyazev, Michal Drozdzal, Graham W. Taylor, and Adriana Romero-Soriano. Parameter prediction for unseen deep architectures. *ArXiv*, abs/2110.13100, 2021.

llama 3 team. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

William S. Peebles, Ilija Radosavovic, Tim Brooks, Alexei A. Efros, and Jitendra Malik. Learning to learn with generative models of neural network checkpoints. *ArXiv*, abs/2209.12892, 2022.

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

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

Neale Ratzlaff and Li Fuxin. HyperGAN: A generative model for diverse, performant neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 5361–5369. PMLR, 09–15 Jun 2019.

Konstantin Schürholt, Boris Knyazev, Xavier Giró-i Nieto, and Damian Borth. Hyper-representations as generative models: Sampling unseen neural network weights. In *Advances in Neural Information Processing Systems*, volume 35, pp. 27906–27920, 2022.

Konstantin Schürholt, Boris Knyazev, Xavier Giró i Nieto, and Damian Borth. Hyper-representations as generative models: Sampling unseen neural network weights. In *Advances in Neural Information Processing Systems*, 2022a.

Konstantin Schürholt, Boris Knyazev, Xavier Giró i Nieto, and Damian Borth. Hyper-representation for pre-training and transfer learning. In *First Workshop on Pre-training: Perspectives, Pitfalls, and Paths Forward at ICML 2022*, 2022b.

Konstantin Schürholt, Diyar Taskiran, Boris Knyazev, Xavier Giró i Nieto, and Damian Borth. Model zoos: A dataset of diverse populations of neural network models. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022c.

Konstantin Sch"urholt, Michael W. Mahoney, and Damian Borth. Towards scalable and versatile weight space learning. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

Konstantin Schürholt, Dimche Kostadinov, and Damian Borth. Self-supervised representation learning on neural network weights for model characteristic prediction. In *Advances in Neural Information Processing Systems (NeurIPS 2021)*, Sydney, Australia, 2021.

Yang Shu, Zhi Kou, Zhangjie Cao, Jianmin Wang, and Mingsheng Long. Zoo-tuning: Adaptive transfer from a zoo of models. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 9626–9637. PMLR, 18–24 Jul 2021.

Bedionita Soro, Bruno Andreis, Hayeon Lee, Song Chong, Frank Hutter, and Sung Ju Hwang. Diffusion-based neural network weights generation, 2024.

Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10:99–127, 2002.

Zihao Tang, Zheqi Lv, Shengyu Zhang, Fei Wu, and Kun Kuang. Modelgpt: Unleashing llm's capabilities for tailored model generation, 2024.

Ihsan Ullah, Dustin Carrion, Sergio Escalera, Isabelle M Guyon, Mike Huisman, Felix Mohr, Jan N van Rijn, Haozhe Sun, Joaquin Vanschoren, and Phan Anh Vu. Meta-album: Multi-domain meta-dataset for few-shot image classification. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://meta-album.github.io/.

Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *NIPS*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Kai Wang, Zhaopan Xu, Yukun Zhou, Zelin Zang, Trevor Darrell, Zhuang Liu, and Yang You. Neural network diffusion, 2024.

Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models, 2019.

Jian Yang, Gang Xiao, Yulong Shen, Wei Jiang, Xinyu Hu, Ying Zhang, and Jinghui Peng. A survey of knowledge enhanced pre-trained models. *ArXiv*, abs/2110.00269, 2021.

Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. *ArXiv*, abs/1810.05749, 2019.

Ziyu Zhao, Leilei Gan, Guoyin Wang, Yuwei Hu, Tao Shen, Hongxia Yang, Kun Kuang, and Fei Wu. Retrieval-augmented mixture of lora experts for uploadable machine learning, 2024.

Andrey Zhmoginov, Mark Sandler, and Max Vladymyrov. Hypertransformer: Model generation for supervised and semi-supervised few-shot learning. *ArXiv*, abs/2201.04182, 2022.

Use of Large Language Models

Large language models were used only for writing assistance, including grammar correction, clarity improvement, and language polishing. All scientific ideas, methodological contributions, experiments, results, analyses, and conclusions are our own. All figures were created by the authors. In particular, the concept figures were directly implemented in the main paper using the `TikZ` package.

A Vectorizing Neural Network Parameters

To enable our generative mapping function H to learn from diverse pretrained models, we introduce a standardized parameter vectorization scheme that transforms weights and biases into a uniform vector representation. For a network with L layers, fully connected layers are vectorized by reshaping the weight matrix $\theta^{(l)} \in \mathbb{R}^{d_{l-1} \times d_l}$ into $\mathrm{vec}(\theta^{(l)}) \in \mathbb{R}^{d_{l-1} d_l}$ and appending the bias $b^{(l)} \in \mathbb{R}^{d_l}$, yielding $d_{l-1} d_l + d_l$ elements. Similarly, convolutional layers with kernel $\theta^{(l)} \in \mathbb{R}^{k_h \times k_w \times c_{\text{in}} \times c_{\text{out}}}$ are flattened to $\mathrm{vec}(\theta^{(l)}) \in \mathbb{R}^{k_h k_w c_{\text{in}} c_{\text{out}}}$ and concatenated with the bias $b^{(l)} \in \mathbb{R}^{c_{\text{out}}}$ to produce $k_h k_w c_{\text{in}} c_{\text{out}} + c_{\text{out}}$ elements. We consider two aggregation strategies: an architecture-wise vectorization that concatenates all layers into a single vector $\Theta = \bigoplus_{l=1}^{L} \left[\mathrm{vec}(\theta^{(l)}) \oplus b^{(l)} \right]$, and a layer-wise encoding that preserves each layer's representation.

A.1 Model Overview

Our VQVAE model is a modified implementation based on the VQGAN codebase. Table 6 summarizes the architecture details, while Table 7 provides an overview of the generative model along with its parameter counts. We optimize the model using Adam with a learning rate of 1×10^{-4} and employ a cyclical temperature schedule for the Gumbel-Softmax, annealing the temperature from 1 to 1×10^{-4}. We also illustrate an example template 7 for experimental results in Figure 4, where only architecture embeddings are used since all architectures were pretrained on the same dataset.

B Additional Results and Tables

In this section, we provide a comprehensive report of the experimental results such as those presented in section **??**. The primary objective of these results is to demonstrate the ability to encode diverse architectures across a variety of datasets while maintaining retrieval performance comparable to the pretrained models. Table 2 showcases representative results for different architectures, along with their corresponding parameter counts. The conditioning has been applied to both architectural configurations and datasets. Additionally, we include example configuration samples used for instruction encoding in Table 9.

Index	Name	Type	Parameters
0	encoder	Encoder	197 M
1	decoder	Decoder	198 M
2	loss	VQLoss	0
3	quantize	GumbelQuantize	132 K
4	quant_conv	Conv2d	4.2 K
5	post_quant_conv	Conv2d	4.2 K

Table 6: Pretrained weights configuration for the VQVAE model. The table details the layer-wise parameters of the VQVAE model, highlighting its encoder, decoder, and quantization components optimized for downstream tasks.

Index	Name	Type	Parameters
0	first_stage_model	GumbelVQNoDisc	48.5 M
1	cond_stage_model	CondStage	524 K
2	ArchCond	Identity	0
3	transformer	GPT	27.8 M

Table 7: Layer-wise configuration of the model, including the first-stage model, conditional stage model, and transformer components. The table specifies the type and parameter count of each component. This table shows the structure of the model used for the experiments in Section

Domain	**Original Dataset Name**	**Number of Classes**
Large Animals	Birds	315
	Dogs	120
	Animals with Attributes	50
Small Animals	Plankton	102
	Insects 2	102
	Insects	117
Plants	Flowers	102
	PlantNet	25
	Fungi	25
Plant Diseases	Plant Village	38
	Medicinal Leaf	26
	PlantDoc	27
Microscopy	Bacteria	33
	PanNuke	19
	Subcellular Human Protein	21
Remote Sensing	RESISC	45
	RSICB	45
	RSD	43
Vehicles	Cars	196
	Airplanes	21
	Boats	26
Manufacturing	Textures	64
	Textures DTD	47
	Textures ALOT	250
Human Actions	Sports	73
	Stanford 40 Actions	40
	MPII Human Pose	29
OCR	Omniprint-MD-mix	706
	Omniprint-MD-5-bis	706
	Omniprint-MD-6	703

Table 8: Diverse pretrained datasets grouped by domain, showcasing their variety in classes for cross-dataset adaptation tasks.

Table 9: Examples of architecture configurations for various neural network families.

Architecture Name	Configuration
vgg11_bn	architecture: A, layers: [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'], batch_norm: true
mobilenetv2_x0_5	width_multiplier: 0.5, final_layer_channels: 640
repvgg_a0	width_multiplier: [0.75, 0.75, 0.75, 2.5]
shufflenetv2_x0_5	stages_repeats: [4, 8, 4], stages_out_channels: [24, 48, 96, 192, 1024]
resnet20	layers: [3, 3, 3]

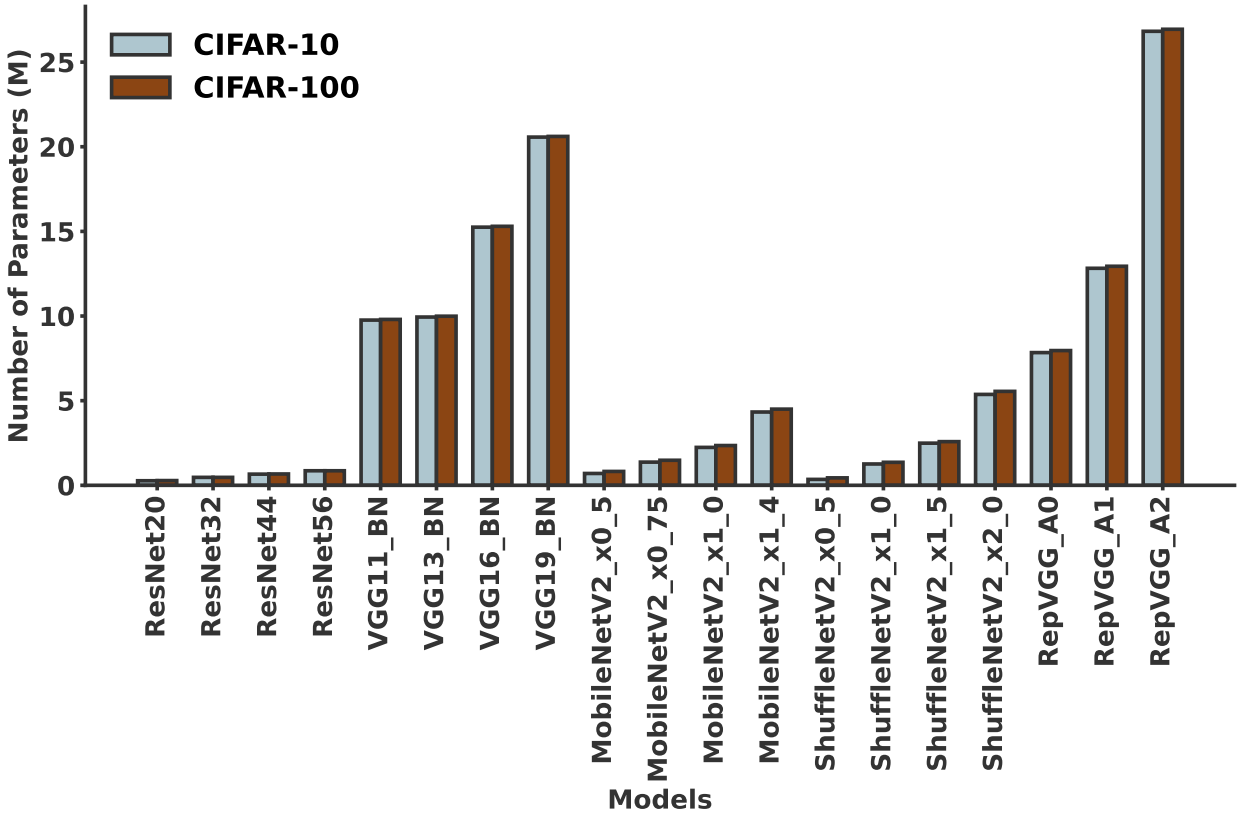

Figure 5: Parameters distribution of diverse architectures pretrained on CIFAR-10 and CIFAR100 all jointly encoded by IGPG