UNIVERSAL HYPERNETWORKS VIA STRUCTURED CONDITIONING ACROSS ARCHITECTURES AND TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Conventional hypernetworks are typically designed around a specific target-network parameterization, so changing the target architecture often requires redesigning and retraining the hypernetwork. We introduce the *Universal Hypernetwork* (UHN), a fixed-architecture generator that predicts weights from deterministic parameter, architecture, and task descriptors. By moving target specificity into these descriptors, UHN decouples the generator design from the structure of the target model. We evaluate UHN across vision, graph, text, and formula-regression tasks, showing that a fixed generator architecture can be applied across different target structures. In addition, a single UHN supports multi-model generalization within a model family and multi-task learning across heterogeneous models. Finally, by treating a UHN itself as a target network, the generation process can be applied recursively.

1 INTRODUCTION

Hypernetworks generate the parameters of target networks and have been applied to model compression (Ha et al., 2016), neural architecture search (Zhang et al., 2018), and multi-task learning (Navon et al., 2020). A common limitation, however, is that the hypernetwork design is often closely tied to the target architecture: changing the target model can require redesigning the generator structure and retraining the hypernetwork (Chen & Wang, 2022; Navon et al., 2020). This coupling makes it difficult to reuse one generator design across models that differ in architecture and scale.

We address this coupling by representing target-specific information through the inputs to the hypernetwork rather than through architecture-dependent outputs. Our *Universal Hypernetwork* (UHN) generates each scalar weight from deterministic descriptors of its parameter index, model structure, and task. Changing the target therefore changes the descriptors being queried rather than the generator architecture itself.

The four settings in Figure 1 illustrate the scope of UHN, spanning single-model, multi-model, multi-task, and recursive generation. Our contributions are as follows:

- **Generation across architectures and scales.** We show that the same UHN design can generate target models across diverse architecture families and task domains, while retaining the same generator architecture and parameter count as target-model size increases.
- **Shared generation across models and tasks.** We demonstrate that a single jointly trained UHN can parameterize multiple architectures within a model family and generalize to unseen architectures from that family. We further show that one shared UHN can parameterize target models with different architectures for heterogeneous tasks spanning vision, graph, text, and formula regression.
- **Recursive hypernetwork generation.** We show that UHN can generate another UHN that in turn generates a subsequent UHN or the final target model, demonstrating that the generation process can be applied recursively.

2 RELATED WORK

Hypernetworks generate the parameters of another neural network (Ha et al., 2016). By sharing a generator across target parameters, they have been applied to model compression, neural architecture

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

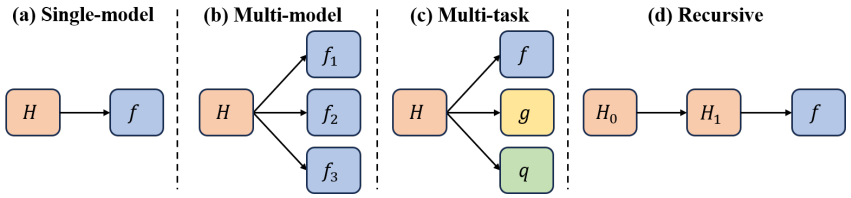


Figure 1: UHN settings: (a) single-model, UHN H generates a single base model f; (b) multi-model, UHN H generates a set of base models $\{f_i\}$ within one model family; (c) multi-task, UHN H generates base models $f, g, q, \ldots$ across potentially heterogeneous architectures for different tasks; and (d) recursive generation, in which the root UHN H_0 generates an intermediate UHN H_1, which in turn generates the base model f.

search, multi-task learning, personalization, and, more recently, LLM adaptation (Ha et al., 2016; Zhang et al., 2018; Navon et al., 2020; Ruiz et al., 2024; Liu et al., 2026). Two parameter-generation strategies are particularly relevant to UHN. Chunk- or component-based methods generate parameter blocks with a shared generator, often using learned embeddings to identify different blocks (Von Oswald et al., 2019; Gu & Yeung, 2025). In parallel, coordinate-conditioned methods generate individual weights as functions of positional queries (Stanley et al., 2009; Fernando et al., 2016; Liao et al., 2023). UHN is closest to this formulation, but extends the query beyond parameter coordinates: its structured descriptor identifies the layer, component type (e.g., weight or bias), and coordinate within that component, together with architecture and task context. This allows target-specific information to be represented in the query while retaining a fixed scalar output interface.

Beyond a single fixed target, hypernetworks have been developed to generate families of architectures using explicit architecture representations (Brock et al., 2017; Zhang et al., 2018), including unseen or diverse architectures (Knyazev et al., 2021; 2023), larger models (Zhou et al., 2024), and networks with different widths (Kim et al., 2026). These approaches accommodate architectural variation through structure representations and target-dependent generation mechanisms. UHN instead represents architectural variation through the queries themselves: changing the target architecture changes the parameter and structure descriptors while the generator architecture and scalar output interface remain unchanged.

In multi-task learning, hypernetworks generate task-specific weights, modules, or adapters (Navon et al., 2020; Mahabadi et al., 2021; Shamsian et al., 2021; Liu et al., 2022), typically while keeping target architectures fixed or structurally compatible. UHN instead jointly conditions on task and architecture to share one generator across heterogeneous task–model pairs.

Recursive hypernetworks allow a generated hypernetwork to serve as the generator for another network (Liao et al., 2023; Lutati & Wolf, 2021). We use this setting to test whether UHN's descriptor-conditioned formulation remains viable when repeatedly applied to generate hypernetworks and downstream targets.

Appendix A further discusses distributional weight-generation methods and implicit neural representations (INRs), whose coordinate-to-value formulation provides a conceptual connection to UHN.

3 Method

3.1 Universal Hypernetwork

3.1.1 Overview

Given a target specification (base model architecture and task), UHN generates each scalar parameter by querying a generator $H_{\boldsymbol{\theta}}$ with deterministic descriptors. At a high level, the generator is asked the same question for every scalar weight: *which parameter is this, in which model, and for which task?* Figure 2 illustrates the weight-generation process.

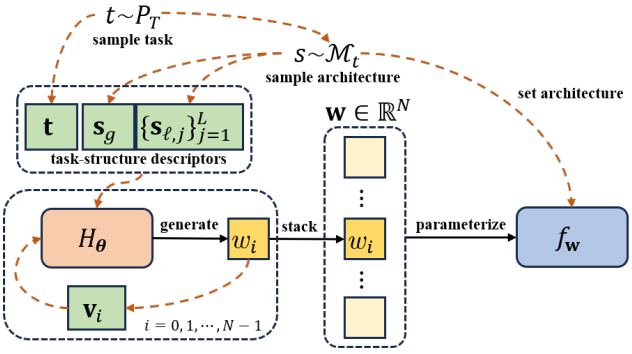


Figure 2: Method overview. Dark solid arrows denote differentiable paths through which gradients are backpropagated, while red dashed arrows indicate non-gradient paths. We sample the task and architecture of the base model $f_{\mathbf{w}}$ and collect their task-structure descriptors. For each parameter in $f_{\mathbf{w}}$, we collect its index descriptor $\mathbf{v}_i$ and feed it into UHN $H_{\boldsymbol{\theta}}$ along with the task-structure descriptors to obtain each individual weight w_i of f. These weights are then stacked into the weight vector $\mathbf{w}$, which parameterizes $f_{\mathbf{w}}$.

For a base model $f_{\mathbf{w}}$ with N parameters, we adopt a hierarchical view of its architecture: a *model* consists of L *layers*, and each layer consists of several *components* (e.g., a linear layer has a weight matrix and a bias vector).

The key idea is to move target-specificity into the *inputs* to $H_{\boldsymbol{\theta}}$ rather than into architecture-specific output heads. The descriptor representation should distinguish different target parameters and retain the structural and task information relevant to weight generation. Concretely, for each parameter indexed by i, we build four kinds of deterministic descriptors from the chosen architecture and task (detailed specifications are given in Appendix B.2):

- **Index descriptor** $\mathbf{v}_i$: tells the generator *which scalar parameter* is being queried, including its layer, component, and coordinate within that component.
- **Global structure descriptor** $\mathbf{s}_g$: tells the generator *what overall model* this parameter belongs to, such as model family, depth, or width configuration.
- **Local structure descriptors** $\{\mathbf{s}_{\ell,j}\}_{j=1}^{L}$: tell the generator *how each layer is configured*, such as layer type and input/output dimensions.
- **Task descriptor** $\mathbf{t}$: tells the generator *what task* the model is solving, such as image classification on MNIST or text classification on AG News.

Using these descriptors, UHN predicts each scalar parameter as

$$w_i = H_{\boldsymbol{\theta}}\big(\mathbf{v}_i,\, \mathbf{s}_g,\, \{\mathbf{s}_{\ell,j}\}_{j=1}^{L},\, \mathbf{t}\big). \tag{1}$$

In this sense, UHN models target-network parameters as a function of structured descriptors. A target model is parameterized by querying the descriptors of its individual weights through the same generator. Changing the target model or task therefore changes only the *queries* (the descriptors), not the architecture of $H_{\boldsymbol{\theta}}$.

Example. Consider a base model with a single linear layer (no bias) of shape $d_{\text{in}} \times d_{\text{out}}$ for MNIST (Lecun et al., 1998) classification. For each scalar weight connecting input unit p to output unit q, we construct $\mathbf{v}_i$ from coordinates such as (p, q) (together with a layer/component code), set $\mathbf{s}_g$ to encode global configuration (e.g., model family and depth) and set the single-layer local code $\mathbf{s}_{\ell,1}$ to include $(d_{\text{in}}, d_{\text{out}})$ and the layer type, and set $\mathbf{t}$ to encode the task/dataset. We query $H_{\boldsymbol{\theta}}$ for all parameters (efficiently via batching) and then reshape the outputs into $d_{\text{in}} \times d_{\text{out}}$ to parameterize the base model.

3.1.2 Unified generation settings

The same descriptor-conditioned generation procedure applies to single-model, multi-model, multi-task, and recursive generation by varying the target specifications and descriptors being queried.

Single-model, single-task. In this setting, structure/task conditioning can be disabled, and all parameters are generated by evaluating $H_{\boldsymbol{\theta}}$ on the corresponding index descriptors, i.e., $w_i = H_{\boldsymbol{\theta}}(\mathbf{v}_i)$.

Multi-model, single-task. Fix the task code $\mathbf{t}$, and vary the architecture descriptors $(\mathbf{s}_g, \{\mathbf{s}_{\ell,j}\}_{j=1}^{L})$ across different base-model architectures. For each architecture, we generate its parameters with the same $H_{\boldsymbol{\theta}}$.

Multi-task. Vary the task code $\mathbf{t}$ across tasks together with the architecture descriptors $(\mathbf{s}_g, \{\mathbf{s}_{\ell,j}\}_{j=1}^{L})$ corresponding to each task's target model. UHN then generates a task-specific model by conditioning on both the chosen task and architecture.

Recursive generation. Treat a hypernetwork itself as a target model. Given the specification of a target hypernetwork H_{k+1} and its index descriptors $\mathbf{v}_{k,i}$, we generate its parameters $\boldsymbol{\theta}_{k+1}$ using the same procedure as for base-model weights. The overall parameter generation process from the root UHN H_0 to the base model f generated by the intermediate UHN H_K can be denoted as a recursion chain $H_0 \to \cdots \to H_k \to \cdots \to H_K \to f$.

3.1.3 UHN ARCHITECTURE

UHN uses a two-stage forward pass: it first encodes the descriptors into fixed-length features and then maps them to a scalar weight. For inference pseudocode, see Algorithm 1.

Stage 1: encode descriptors. Given a target specification, we construct descriptors $(\mathbf{v}_i, \mathbf{s}_g, \{\mathbf{s}_{\ell,j}\}_{j=1}^{L}, \mathbf{t})$ for each parameter index i. We use Gaussian Fourier features (Tancik et al., 2020) to encode the index and task-structure descriptors, helping the generator represent higher-frequency variation in the weight field. Specifically, we form per-layer task-structure descriptors $\mathbf{u}_j = [\mathbf{s}_g; \mathbf{t}; \mathbf{s}_{\ell,j}]$ and compute

$$\boldsymbol{\phi}_i \;=\; \gamma_{\mathbf{B_v}}(\hat{\mathbf{v}}_i), \qquad \boldsymbol{\psi}_j \;=\; \gamma_{\mathbf{B_u}}(\hat{\mathbf{u}}_j). \tag{2}$$

For each $\mathbf{x} \in \{\mathbf{v}_i, \mathbf{u}_j\}$, we first compute its elementwise-normalized version $\hat{\mathbf{x}}$ (see Appendix B.4) and then apply

$$\gamma_{\mathbf{B}}(\hat{\mathbf{x}}) = \left[\cos(\mathbf{B}\hat{\mathbf{x}})^{\top},\ \sin(\mathbf{B}\hat{\mathbf{x}})^{\top}\right]^{\top}, \tag{3}$$

where $\hat{\mathbf{x}} \in \mathbb{R}^n$ and $\mathbf{B} \in \mathbb{R}^{m \times n}$ has entries sampled i.i.d. from $\mathcal{N}(0, \sigma^2)$ once and then kept fixed, yielding $\gamma_{\mathbf{B}}(\hat{\mathbf{x}}) \in \mathbb{R}^{2m}$.

Stage 2: map encodings to weights. UHN $H_{\boldsymbol{\theta}}$ contains an *index branch* that processes $\boldsymbol{\phi}_i$ and an optional *task-structure encoder* that aggregates $\{\boldsymbol{\psi}_j\}_{j=1}^{L}$ into a global conditioning feature. The index branch maps $\boldsymbol{\phi}_i$ through an input linear layer of width d followed by two pre-activation residual blocks (He et al., 2016); each block contains two ReLU (Nair & Hinton, 2010) $\to$ LayerNorm (Ba et al., 2016) $\to$ Linear layers of width d with a shortcut connection.

When architecture/task conditioning is enabled, a single-layer Transformer encoder (Vaswani et al., 2017) (h heads; feed-forward dimension $d_{\text{ff}} = \dim(\boldsymbol{\psi}_j)$; no dropout) processes $\{\boldsymbol{\psi}_j\}_{j=1}^{L}$, followed by mean pooling across the layer/token dimension and a two-layer MLP (Linear $\to$ ReLU $\to$ Linear, hidden/output width d, last layer zero-initialized). The resulting task-structure feature is added to the index-branch representation, and a final ReLU and linear layer predict the scalar weight w_i. Additional architecture hyperparameters are given in Appendix B.8.

3.2 TRAINING PROCEDURE

We train UHN by sampling a target specification (architecture and task), generating its parameters, evaluating the resulting base model, and backpropagating the task loss through the generation process (see Algorithm 2). In practice, we optionally begin with an initialization phase that matches simple parameter statistics (see Algorithm 3), followed by task-loss training.

3.2.1 TARGET SAMPLING

Let $\mathcal{T}$ denote the task set and P_T a distribution over tasks. For each task t, let $\mathcal{M}_t$ denote the distribution over its candidate model architectures. At each training iteration, we sample $t \sim P_T$ and then $s \sim \mathcal{M}_t$, represented by the task descriptor $\mathbf{t}$ and structure descriptors $(\mathbf{s}_g, \{\mathbf{s}_{\ell,j}\}_{j=1}^{L})$, respectively. In single-task settings, t is fixed; when only one target architecture is used, s is fixed as well.

3.2.2 INITIALIZATION STRATEGY

Before the main training phase, we optionally include an initialization phase to stabilize optimization by matching simple statistics of the generated parameters (e.g., mean and variance) to those of standard parameter initializations. Let G denote the set of all generated-model components, and let $g \in G$ denote a particular component (e.g., a weight matrix or bias vector) with empirical mean $\mu(g)$ and standard deviation $\sigma(g)$. We specify desired target statistics $(\mu^*(g), \sigma^*(g))$ for each component (see Appendix B.6) and minimize

$$\mathcal{L}_{\text{init}} = \frac{1}{2|G|} \sum_{g \in G} \Big[\Big(\mu(g) - \mu^*(g)\Big)^2 + \Big(\sigma(g) - \sigma^*(g)\Big)^2 \Big]. \tag{4}$$

Initialization for recursive generation. For recursive chains, we initialize generated models sequentially from shallow to deeper levels, starting with H_1 and ending with the leaf model f. We assign a fixed initialization-step budget to each level; the complete level-wise schedule is provided in Appendix F.

4 EXPERIMENTS

4.1 COMMON EXPERIMENTAL PROTOCOL

Unless otherwise stated, UHN is optimized with AdamW (Loshchilov & Hutter, 2017) and a cosine learning-rate schedule (Loshchilov & Hutter, 2016). We report mean $\pm$ standard deviation over multiple random seeds. Complete hardware, precision, batching, warmup, weight-decay, and common reporting details are provided in Appendix B.7; dataset splits, base-model architectures, model-family construction, and experiment-specific hyperparameter selection are provided in the corresponding experiment appendices.

4.2 SINGLE-MODEL GENERATION FOR HETEROGENEOUS MODEL FAMILIES

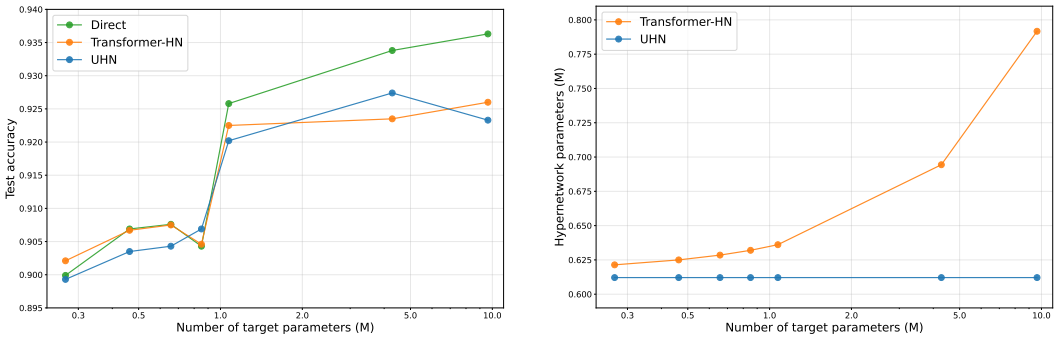
We first study the single-model setting: each experiment fixes one task and one target architecture and trains a separate UHN instance with the same generator design to produce that model's weights. This isolates whether the descriptor-conditioned generation mechanism can be applied across heterogeneous target families without redesigning the generator architecture. Because the task and target architecture are fixed within each run, we omit UHN's task-structure encoder in this setting.

Setup. We evaluate image classification on MNIST (Lecun et al., 1998) and CIFAR-10 (Krizhevsky et al., 2009), graph node classification on Cora, CiteSeer, and PubMed (Sen et al., 2008), text classification on AG News (Zhang et al., 2015) and IMDB (Maas et al., 2011), and formula regression on the 15-function Kolmogorov–Arnold network (KAN) benchmark (Liu et al., 2024). The target architectures span multilayer perceptrons (MLPs), convolutional neural networks (CNNs), graph convolutional networks (GCNs) (Kipf & Welling, 2016), graph attention networks (GATs) (Veličković et al., 2017), Transformers (Vaswani et al., 2017), and KANs. Complete base-model definitions, training protocols, hyperparameter selection, and full results are provided in Appendix C. We further investigate how the index-encoding method and generator capacity affect UHN's generation performance in Appendices G.1 and G.2.

Results. We compare UHN-generated weights against direct training (Direct). Table 1 reports representative classification results, with complete classification results in Table 9 and per-function formula-regression results in Table 10.

Table 1: Representative single-model results across heterogeneous classification tasks. Each row uses a separately trained UHN instance with the same generator design. #Params denotes parameter count; for UHN, it includes trainable network parameters, fixed Fourier-feature matrices, and descriptor normalization statistics. Acc. denotes test accuracy. Complete classification results are reported in Table 9.

Task type	Dataset	Model	#Params (Direct)	#Params (UHN)	Acc. (Direct)	Acc. (UHN)
Image	MNIST	MLP	118,282	158,613	0.9837 ± 0.0007	0.9841 ± 0.0006
Image	CIFAR-10	CNN-20	269,034	612,117	0.8999 ± 0.0027	0.8993 ± 0.0016
Graph	Cora	GCN	92,231	612,117	0.8172 ± 0.0039	0.7950 ± 0.0069
Graph	CiteSeer	GCN	237,446	612,117	0.6996 ± 0.0051	0.6529 ± 0.0190
Graph	PubMed	GAT	33,779	612,117	0.7700 ± 0.0033	0.7801 ± 0.0037
Text	AG News	Transformer-2L	378,372	612,117	0.9186 ± 0.0008	0.9099 ± 0.0005
Text	IMDB	Transformer-1L	377,858	612,117	0.8853 ± 0.0007	0.8638 ± 0.0005



(a) Test accuracy versus target-model parameter count.

(b) Hypernetwork parameter count versus target-model parameter count.

Figure 3: Target-size scaling on CIFAR-10 across depth- and width-scaled CNNs. Exact results are reported in Tables 15 and 16.

Analysis. Across the single-model experiments, UHN uses the same scalar-output generation interface for MLP, CNN, GCN, GAT, Transformer, and KAN targets; MNIST uses a smaller generator configuration for the proof-of-concept, while all other experiments use the 612,117-parameter design.

Performance relative to Direct varies by task. UHN is close to Direct on the MNIST and CIFAR-10 image-classification cases, while the differences are more substantial in several graph and text settings; on PubMed, UHN is slightly higher than Direct for both the GCN and GAT targets. For formula regression, UHN obtains lower RMSE than Direct on 6 of the 15 functions and higher RMSE on 9.

4.3 Target-Size Scaling under a Fixed Generator

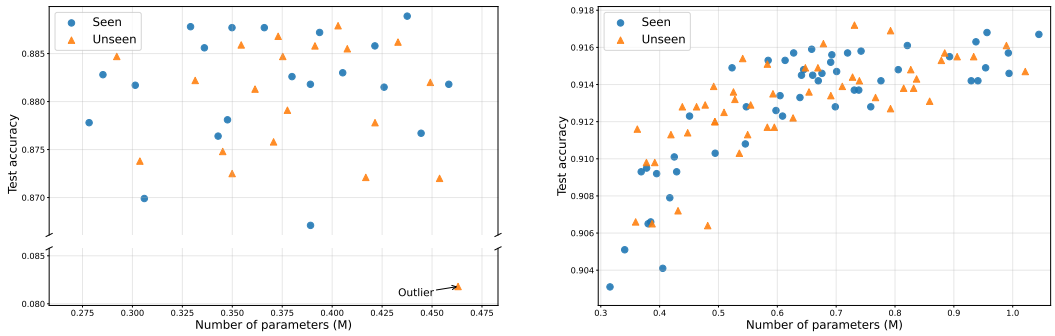
Setup. We study whether UHN can generate increasingly large target networks while keeping the generator architecture and parameter count fixed. On CIFAR-10, we scale the target CNN along two axes: depth (CNN-20 $\rightarrow$ CNN-32/44/56) and width (CNN-20 $\rightarrow$ CNN-20-2/4/6), reaching approximately 9.6M target-model parameters; architecture details are provided in Appendix C.2.

Results. We compare UHN with direct optimization (Direct) and three chunk-based hypernetwork baselines: Ha-HN (Ha et al., 2016), Chunked-HN (Von Oswald et al., 2019), and Transformer-HN. We use Transformer-HN as the representative hypernetwork baseline in the main scaling visualization because it provides a stronger chunk-based generator that jointly processes learned chunk embeddings with a Transformer encoder (Vaswani et al., 2017) before decoding the parameter chunks. Complete results and configurations for Ha-HN and Chunked-HN are reported in Appendix C.6, with computational measurements in Appendix C.7.

Analysis. Figure 3a shows that UHN remains close to Direct and Transformer-HN as CNN depth increases, while the gap to Direct becomes larger for the widest CNNs.

Table 2: Multi-model results. #Models is the size of the sampled model family, and Max target #Params is the maximum number of trainable parameters among target models in that family. Seen and Unseen report average test accuracy over the corresponding evaluation subsets.

Model family	#Models	Max target #Params	Seen	Unseen
CNN Mixed Depth	100	463,018	0.8842 ± 0.0031	0.8430 ± 0.0023
CNN Mixed Width	500	1,072,586	0.9145 ± 0.0014	0.9145 ± 0.0013
CNN Mixed Depth $\times$ Width	1000	1,372,110	0.9038 ± 0.0012	0.9040 ± 0.0016
Transformer Mixed	1000	1,086,212	0.9063 ± 0.0004	0.9066 ± 0.0002



(a) CNN Mixed Depth

(b) CNN Mixed Width

Figure 4: Seen and unseen target-model performance in the multi-model setting. Each point represents one target architecture and shows its parameter count and CIFAR-10 test accuracy.

Figure 3b shows that UHN retains the same generator architecture and 612,117-parameter count across the full target-size range, whereas Transformer-HN grows with target size. Ha-HN and Chunked-HN show the same qualitative growth trend in our implementations (Table 16); related scaling behavior of chunk-based hypernetworks is discussed in Appendix B.1.

However, a fixed generator size does not imply fixed computational cost. Appendix C.7 shows that UHN's training and generation costs rise more sharply than Transformer-HN's as target size increases, revealing a trade-off between parameter efficiency and computational efficiency.

4.4 Shared Generation Across Architectures Within a Model Family

4.4.1 Setup

We evaluate family-level learning by training one shared UHN per model family to generate multiple target architectures and testing it on both seen architectures included in the training model set and held-out unseen architectures from the same family. We consider three CNN families on CIFAR-10 (CNN Mixed Depth, CNN Mixed Width, and CNN Mixed Depth $\times$ Width) and one Transformer family on AG News (Transformer Mixed). UHN uses the task-structure encoder to represent target-architecture information, increasing its parameter count from 612,117 to 663,151. Full family definitions, model-set construction and splitting, and hyperparameter selection are provided in Appendix D; the contribution of task-structure conditioning is examined in Appendix G.3.

4.4.2 Results

We evaluate within-family generalization on seen architectures drawn from the training model set, $M'_{\text{train}} \subset M_{\text{train}}$, and unseen architectures in the held-out set M_{test}. Table 2 reports test accuracy averaged over each subset, with mean and standard deviation over $n = 3$ random seeds. Complete model-set statistics and split sizes are provided in Table 19.

Figure 4 shows per-architecture performance against target-model size for CNN Mixed Depth and CNN Mixed Width.

Table 3: Multi-task results. Perf. denotes test accuracy for classification (higher is better) and test RMSE for formula regression (lower is better). "UHN Single-task" denotes a separate UHN trained per task; "UHN Multi-task" denotes one shared UHN trained jointly on all tasks. UHN Single-task baselines are taken from Section 4.2.

Task type	Dataset	Model	Perf. (UHN Single-task)	Perf. (UHN Multi-task)
Image	MNIST	MLP	0.9841 ± 0.0006	0.9786 ± 0.0012
Image	CIFAR-10	CNN-44	0.9043 ± 0.0031	0.8927 ± 0.0011
Graph	Cora	GCN	0.7950 ± 0.0069	0.7930 ± 0.0053
Graph	PubMed	GAT	0.7801 ± 0.0037	0.7697 ± 0.0031
Text	AG News	Transformer-2L	0.9099 ± 0.0005	0.9062 ± 0.0005
Formula (RMSE)	kv	KAN-g5	0.0104 ± 0.0048	0.0172 ± 0.0112

Additional scatter plots for CNN Mixed Depth $\times$ Width and Transformer Mixed, together with comparisons between shared and separately trained UHNs, are reported in Appendix D.4.

4.4.3 Analysis

Seen and unseen accuracies are nearly identical for CNN Mixed Width, CNN Mixed Depth $\times$ Width, and Transformer Mixed, indicating strong within-family generalization in these three settings. The per-architecture scatter plots in Figures 4 and 5 show similar behavior across a range of target-model sizes.

CNN Mixed Depth exhibits a larger seen–unseen gap. This gap is largely driven by one unseen architecture with substantially lower accuracy. In our sampled model set, this architecture is deeper than every training architecture and corresponds to the largest target model shown in the Mixed Depth scatter in Figure 4a.

Table 22 further shows that the shared UHN achieves performance close to separately trained UHNs on representative CNN Mixed Depth $\times$ Width models. Overall, UHN generalizes well within the evaluated model families, while the Mixed Depth outlier indicates that performance can degrade for architectures beyond the training range.

4.5 Shared Generation Across Heterogeneous Tasks

4.5.1 Setup

We train one fixed-capacity UHN jointly across six task–model pairs spanning image, graph, text, and formula regression: MLP on MNIST, CNN-44 on CIFAR-10, GCN on Cora, GAT on PubMed, Transformer-2L on AG News, and KAN-g5 on kv, one of the 15 special functions in the KAN benchmark. Each task uses one base architecture, while architectural variation within a task is studied separately in the multi-model experiments. We use the task-structure encoder in this setting to incorporate task and base-model information; its effect is evaluated separately in Appendix G.3. Task sampling and hyperparameter selection are detailed in Appendix E.

4.5.2 Results

Table 3 compares the jointly trained UHN with separately trained single-task UHNs from the corresponding single-model experiments (Section 4.2); results are reported as mean $\pm$ standard deviation over $n = 3$ random seeds.

4.5.3 Analysis

A single jointly trained UHN generates models across all six heterogeneous task–model pairs, with task-dependent degradation relative to separately trained single-task UHNs. The difference is small for Cora and AG News and larger for CIFAR-10 and PubMed. For formula regression, the multi-task UHN obtains an RMSE of 0.0172, compared with 0.0104 for the single-task UHN.

4.6 Recursive Generation through Hypernetwork Composition

4.6.1 Setup

We evaluate recursive chains of the form

$$H_0 \to \cdots \to H_k \to \cdots \to H_K \to f, \tag{5}$$

where each arrow denotes "generate the weights of." The root UHN H_0 generates H_1; for $1 \leq k < K$, each generated UHN H_k generates H_{k+1}; and H_K generates the leaf model f. The leaf model is the same MNIST MLP used in the single-model experiment (Section 4.2), and all UHNs in the recursive chain use the task-structure encoder.

We evaluate recursive depths $K = 1, 2, 3$. Generated-UHN architectures and depth-specific optimization and stabilization procedures are provided in Appendix F.

4.6.2 Results

For recursive depths $K = 1, 2, 3$, leaf-model accuracy is 0.9825 ± 0.0007, 0.9795 ± 0.0011, and 0.9741 ± 0.0021, respectively. For comparison, the no-recursion baseline from the single-model MNIST MLP experiment achieves 0.9841 ± 0.0006 (Section 4.2). Results are reported as mean $\pm$ standard deviation over three random seeds. An initialization ablation, including the divergence observed when initialization is disabled in recursive training, is reported in Appendix G.4.

4.6.3 Analysis

Recursive training succeeds at all evaluated depths $K = 1, 2, 3$, while leaf-model accuracy decreases monotonically with recursive depth. This degradation may reflect errors accumulated across successive generation stages, the increasingly long optimization path, or both.

The initialization ablation further shows that recursive generation is sensitive to the initial parameter statistics of the generated UHNs. Without the initialization stage that aligns these statistics with standard network initialization, training diverges even at $K = 1$.

5 Discussion

UHN shifts target specificity from architecture-dependent generator outputs to structured parameter, architecture, and task descriptors. This allows the same scalar weight-generation mechanism to be used across heterogeneous target structures and enables a single trained generator to be shared across multiple architectures or task–model pairs.

The experiments also reveal the trade-offs of this formulation. UHN can keep its generator architecture and parameter count fixed as target-model size increases, but its computational cost still grows with the number of generated weights. Shared generation generalizes well within most evaluated model families, while performance can degrade for architectures beyond the training range and when one generator is shared across heterogeneous tasks.

Finally, recursive generation shows that a UHN can itself be treated as a target network, although increasing recursive depth makes optimization more difficult. Extended discussion of representation, task-structure conditioning, generated-weight initialization, limitations, and future directions is provided in Appendix H.

6 Conclusion

We introduced Universal Hypernetwork (UHN), a descriptor-conditioned approach that represents target-specific parameter, architecture, and task information through structured inputs rather than architecture-specific generator outputs. Across heterogeneous target structures, the same generation formulation can be applied without redesigning the output interface, while jointly trained UHNs can additionally share generation across multiple architectures or task–model pairs. UHN can also be applied recursively, with one UHN generating the parameters of another UHN, which can then continue the generation process. Together, these results establish descriptor-conditioned weight generation as a common formulation across different models, tasks, and recursive generation settings.

AI USE STATEMENT

In this work, we used generative AI tools to assist with paper editing and the discovery, filtering, and organization of related literature. All AI-assisted text and literature references were reviewed and verified by the author. We did not use generative AI tools to design the methodology or experiments. We take responsibility for the final content of this work, including text, claims, and artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide the model architectures, training configurations, hyperparameters, datasets, baseline settings, and experimental protocols needed to reproduce the reported results in the main text and appendix. Anonymous source code and experiment configurations are included in the supplementary material.

REFERENCES

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

JC de Borda. M'emoire sur les' elections au scrutin. *Histoire de l'Acad'emie Royale des Sciences*, 1781.

Andrew Brock, Theodore Lim, James M Ritchie, and Nick Weston. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*, 2017.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

Yinbo Chen and Xiaolong Wang. Transformers as meta-learners for implicit neural representations. In *European Conference on Computer Vision*, pp. 170–187. Springer, 2022.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 4(5):11, 2015.

Lior Deutsch, Erik Nijkamp, and Yu Yang. A generative model for sampling high-performance and diverse weights for neural networks. *arXiv preprint arXiv:1905.02898*, 2019.

Amer Essakine, Yanqi Cheng, Chun-Wun Cheng, Lipei Zhang, Zhongying Deng, Lei Zhu, Carola-Bibiane Schönlieb, and Angelica I Aviles-Rivero. Where do we stand with implicit neural representations? a technical and performance survey. *arXiv preprint arXiv:2411.03688*, 2024.

Chrisantha Fernando, Dylan Banarse, Malcolm Reynolds, Frederic Besse, David Pfau, Max Jaderberg, Marc Lanctot, and Daan Wierstra. Convolution by evolution: Differentiable pattern producing networks. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 109–116, 2016.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256. JMLR Workshop and Conference Proceedings, 2010.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*, 2017.

Jeffrey Gu and Serena Yeung. Foundation models secretly understand neural network weights: Enhancing hypernetwork architectures with foundation models. In *International Conference on Learning Representations*, pp. 58565–58577, 2025.

Saumya Gupta, Scott Biggs, Moritz Laber, Zohair Shafi, Robin Walters, and Ayan Paul. Deepweightflow: Re-basined flow matching for generating neural network weights. In *International Conference on Learning Representations*, 2026.

David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pp. 630–645. Springer, 2016.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pp. 448–456. pmlr, 2015.

Jiwoo Kim, Swarajh Mehta, Hao-Lun Hsu, Hyunwoo Ryu, Yudong Liu, and Miroslav Pajic. Nnit: Width-agnostic neural network generation with structurally aligned weight spaces. *arXiv preprint arXiv:2603.00180*, 2026.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Sylwester Klocek, Łukasz Maziarka, Maciej Wołczyk, Jacek Tabor, Jakub Nowak, and Marek Śmieja. Hypernetwork functional image representation. In *International Conference on Artificial Neural Networks*, pp. 496–510. Springer, 2019.

Boris Knyazev, Michal Drozdzal, Graham W Taylor, and Adriana Romero Soriano. Parameter prediction for unseen deep architectures. *Advances in Neural Information Processing Systems*, 34: 29433–29448, 2021.

Boris Knyazev, Doha Hwang, and Simon Lacoste-Julien. Can we scale transformers to predict parameters of diverse imagenet models? In *International Conference on Machine Learning*, pp. 17243–17259. PMLR, 2023.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, University of Toronto, 2009.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Isaac Liao, Ziming Liu, and Max Tegmark. Generating interpretable networks using hypernetworks. *arXiv preprint arXiv:2312.03051*, 2023.

Yen-Cheng Liu, Chih-Yao Ma, Junjiao Tian, Zijian He, and Zsolt Kira. Polyhistor: Parameter-efficient multi-task adaptation for dense vision tasks. *Advances in Neural Information Processing Systems*, 35:36889–36901, 2022.

Yewei Liu, Xiyuan Wang, Yansheng Mao, Yoav Gelbery, Haggai Maron, and Muhan Zhang. Shine: A scalable in-context hypernetwork for mapping context to lora in a single pass. In *International Conference on Machine Learning*, 2026.

Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks. *arXiv preprint arXiv:2404.19756*, 2024.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Shahar Lutati and Lior Wolf. Hyperhypernetwork for the design of antenna arrays. In *International Conference on Machine Learning*, pp. 7214–7223. PMLR, 2021.

Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pp. 142–150, 2011.

Rabeeh Karimi Mahabadi, Sebastian Ruder, Mostafa Dehghani, and James Henderson. Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks. *arXiv preprint arXiv:2106.04489*, 2021.

Ricardo Martin-Brualla, Noha Radwan, Mehdi SM Sajjadi, Jonathan T Barron, Alexey Dosovitskiy, and Daniel Duckworth. Nerf in the wild: Neural radiance fields for unconstrained photo collections. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 7210–7219, 2021.

Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1):99–106, 2021.

Thomas Müller, Alex Evans, Christoph Schied, and Alexander Keller. Instant neural graphics primitives with a multiresolution hash encoding. *ACM transactions on graphics (TOG)*, 41(4): 1–15, 2022.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

Aviv Navon, Aviv Shamsian, Gal Chechik, and Ethan Fetaya. Learning the pareto front with hypernetworks. *arXiv preprint arXiv:2010.04104*, 2020.

Keunhong Park, Utkarsh Sinha, Peter Hedman, Jonathan T Barron, Sofien Bouaziz, Dan B Goldman, Ricardo Martin-Brualla, and Steven M Seitz. Hypernerf: A higher-dimensional representation for topologically varying neural radiance fields. *arXiv preprint arXiv:2106.13228*, 2021.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pp. 1310–1318. PMLR, 2013.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Maria Pilligua, Danna Xue, and Javier Vazquez-Corral. Hypernvd: Accelerating neural video decomposition via hypernetworks. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 22933–22942, 2025.

Neale Ratzlaff and Li Fuxin. Hypergan: A generative model for diverse, performant neural networks. In *International Conference on Machine Learning*, pp. 5361–5369. PMLR, 2019.

Nataniel Ruiz, Yuanzhen Li, Varun Jampani, Wei Wei, Tingbo Hou, Yael Pritch, Neal Wadhwa, Michael Rubinstein, and Kfir Aberman. Hyperdreambooth: Hypernetworks for fast personalization of text-to-image models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 6527–6536, 2024.

Vishwanath Saragadam, Daniel LeJeune, Jasper Tan, Guha Balakrishnan, Ashok Veeraraghavan, and Richard G Baraniuk. Wire: Wavelet implicit neural representations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 18507–18516, 2023.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

Aviv Shamsian, Aviv Navon, Ethan Fetaya, and Gal Chechik. Personalized federated learning using hypernetworks. In *International conference on machine learning*, pp. 9489–9502. PMLR, 2021.

Vincent Sitzmann, Julien Martel, Alexander Bergman, David Lindell, and Gordon Wetzstein. Implicit neural representations with periodic activation functions. *Advances in neural information processing systems*, 33:7462–7473, 2020.

Bedionita Soro, Bruno Andreis, Hayeon Lee, Wonyong Jeong, Song Chong, Frank Hutter, and Sung Ju Hwang. Diffusion-based neural network weights generation. In *International Conference on Learning Representations*, pp. 99644–99675, 2025.

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

Kenneth O Stanley, David B D'Ambrosio, and Jason Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

Matthew Tancik, Pratul Srinivasan, Ben Mildenhall, Sara Fridovich-Keil, Nithin Raghavan, Utkarsh Singhal, Ravi Ramamoorthi, Jonathan Barron, and Ren Ng. Fourier features let networks learn high frequency functions in low dimensional domains. *Advances in neural information processing systems*, 33:7537–7547, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Johannes Von Oswald, Christian Henning, Benjamin F Grewe, and João Sacramento. Continual learning with hypernetworks. *arXiv preprint arXiv:1906.00695*, 2019.

Kai Wang, Dongwen Tang, Wangbo Zhao, Konstantin Schürholt, Zhangyang Wang, and Yang You. Scaling up parameter generation: A recurrent diffusion approach. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

Yuxin Wu and Kaiming He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 3–19, 2018.

Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*, pp. 40–48. PMLR, 2016.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Chris Zhang, Mengye Ren, and Raquel Urtasun. Graph hypernetworks for neural architecture search. *arXiv preprint arXiv:1810.05749*, 2018.

Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28, 2015.

Xinyu Zhou, Boris Knyazev, Alexia Jolicoeur-Martineau, and Jie Fu. Logah: Predicting 774-million-parameter transformers using graph hypernetworks with 1/100 parameters. *arXiv preprint arXiv:2405.16287*, 2024.

A Additional Related Work

A.1 Distributional Modeling of Neural-Network Parameters

A complementary line of work models distributions over neural-network parameters using approaches such as generative adversarial networks (GANs), diffusion models, and flow matching (Ratzlaff & Fuxin, 2019; Deutsch et al., 2019; Soro et al., 2025; Wang et al., 2025; Gupta et al., 2026). These methods enable different sets of network parameters to be sampled from the learned distribution, sometimes conditioned on task, dataset, or architecture. UHN addresses a different formulation of weight generation: for a specified architecture and task, it deterministically maps structured descriptors of individual parameters to their values.

A.2 Implicit Neural Representations

Implicit neural representations (INRs) model signals as continuous coordinate-to-value functions parameterized by neural networks (Essakine et al., 2024). Prior work has improved such mappings through specialized activation functions (Sitzmann et al., 2020; Saragadam et al., 2023) and more effective coordinate encodings (Tancik et al., 2020; Mildenhall et al., 2021; Müller et al., 2022).

This coordinate-to-value formulation of INRs can be naturally extended to neural-network weight generation: network parameters can be treated as values indexed by parameter coordinates or descriptors. UHN adopts this formulation by mapping each structured parameter descriptor to a scalar weight. The representation of these queries is important: our index-encoding ablation (Table 28) shows substantial differences between raw, positional, and Gaussian Fourier feature encodings. UHN further conditions this mapping on architecture and task descriptors, analogous in spirit to conditional INR formulations (Park et al., 2021; Martin-Brualla et al., 2021).

Related work also considers the opposite direction—using hypernetworks to generate the parameters of INRs (Klocek et al., 2019; Chen & Wang, 2022; Pilligua et al., 2025; Gu & Yeung, 2025). In that setting, the generated network itself represents a signal such as an image, video, or neural field; UHN instead uses an INR-like descriptor-to-value mapping to generate the parameters of target networks.

B Method

We begin with a scaling analysis of chunk-based hypernetworks (Ha et al., 2016; Von Oswald et al., 2019), then specify descriptor layouts, present inference/training/initialization algorithms, and close with descriptor normalization, base-model and recursive-hypernetwork structure definitions, initialization target statistics, and hyperparameter-selection details.

B.1 Scaling of Chunk-Based Hypernetworks

Target-dependent parameterization and scaling. Ha-HN, Chunked-HN, and Transformer-HN can all be viewed as chunk-based hypernetworks, but they differ in how target parameters are partitioned and how each chunk is generated. Ha-HN (Ha et al., 2016) uses kernel-wise chunking: learned kernel embeddings are mapped through linear networks to fixed-size convolution-kernel chunks, and larger kernels are assembled by concatenating multiple generated chunks. Chunked-HN (Von Oswald et al., 2019) and Transformer-HN instead flatten the target-model weights into a one-dimensional sequence and partition it into fixed-size chunks; Chunked-HN uses an MLP generator, whereas Transformer-HN contextualizes the chunk embeddings with a Transformer encoder before decoding the chunks.

To characterize the scaling induced by learned chunk embeddings, consider a chunk-based hypernetwork that generates N target parameters in chunks of size c and assigns each chunk r a learned embedding $\mathbf{e}_r \in \mathbb{R}^{d_{\text{emb}}}$, so $N_{\text{chunks}} = N/c$. Let o be the hidden width of the hypernetwork core just before the final linear output layer, and let $N_{H,0}$ denote the number of hypernetwork parameters that do not scale with $(N, c, N_{\text{chunks}})$. Then the total number of hypernetwork parameters

$$N_H = N_{H,0} + oc + N_{\text{chunks}} d_{\text{emb}}. \tag{6}$$

Table 4: Index descriptor layout $\mathbf{v}_i \in \mathbb{R}^{10}$.

Idx	Field	Description
0	layer_idx	Layer index.
1	layer_type	Layer type (categorical).
2	param_type	Parameter type within the layer (categorical).
3	out_idx	Output index / output channel / output embedding channel.
4	in_idx	Input index / input channel / input embedding channel.
5	kernel_h_idx	Convolution kernel height index (h coordinate).
6	kernel_w_idx	Convolution kernel width index (w coordinate).
7	embedding_idx	Token index.
8	sequence_idx	Token position.
9	grid_idx	KAN (Liu et al., 2024) grid/knots index.

Table 5: Global structure descriptor layout $\mathbf{s}_g \in \mathbb{R}^6$.

Idx	Field	Description
0	model_type	Model family type (categorical).
1	num_layers	Number of layers.
2	cnn_stage_num	Number of stages (CNN).
3	num_encoders	Number of encoder blocks (Transformer (Vaswani et al., 2017)).
4	num_structure_freqs	Number of frequencies for structure/task Fourier features (Tancik et al., 2020) (Recursive UHN).
5	num_index_freqs	Number of frequencies for index Fourier features (Recursive UHN).

The arithmetic–geometric mean inequality then yields the lower bound

$$oc + N_{\text{chunks}} d_{\text{emb}} = oc + \frac{N}{c} d_{\text{emb}} \geq 2\sqrt{o d_{\text{emb}}}\,\sqrt{N}, \tag{7}$$

so for fixed (o, d_{emb}), such hypernetworks satisfy $N_H = \Omega(\sqrt{N})$.

This lower bound applies to the learned-chunk construction above: as the number of target parameters grows, either the chunk output or the collection of learned chunk embeddings must grow. Ha-HN, Chunked-HN, and Transformer-HN all follow this general construction, although the details differ across methods. UHN instead queries individual weights with deterministic descriptors, so its generator parameter count does not grow with the number of generated target parameters.

B.2 Descriptor specifications

This section documents the descriptor layouts used in our implementation. Categorical attributes are deterministically mapped to integer IDs using a fixed implementation-defined mapping. Inapplicable *index* fields are set to -1, while inapplicable *structure* fields are set to 0. In our experiments, the *task* fields are always applicable. Detailed per-layer/per-model active-field conventions are provided in Appendix B.5.

Index descriptor $\mathbf{v}_i \in \mathbb{R}^{10}$ (Table 4). The index descriptor specifies the location and type of an individual parameter.

Global structure descriptor $\mathbf{s}_g \in \mathbb{R}^6$ (Table 5). The global structure descriptor encodes architecture-level metadata.

Local structure descriptor $\mathbf{s}_{\ell,j} \in \mathbb{R}^{21}$ (Table 6). The local descriptor describes layer j.

Task descriptor $\mathbf{t} \in \mathbb{R}^2$ (Table 7). The task descriptor encodes task-level information (task type and dataset).

Putting the above together, a single generated parameter w_i is determined by its index descriptor $\mathbf{v}_i$ (what parameter this is), the global structure descriptor $\mathbf{s}_g$ and a group of local structure descriptors $\{\mathbf{s}_{\ell,j}\}_{j=1}^{L}$ for a model with L layers (what model it belongs to), and the task descriptor $\mathbf{t}$ (what task it serves),

Table 6: Local structure descriptor layout $\mathbf{s}_{\ell,j} \in \mathbb{R}^{21}$.

Idx	Field	Description
0	layer_idx	Layer index.
1	layer_type	Layer type (categorical).
2	bias_type	Bias option (categorical).
3	norm_type	Normalization option (categorical).
4	shortcut_type	Shortcut (He et al., 2016) option (categorical).
5	output_size	Output dimension / channels / embedding dim.
6	input_size	Input dimension / channels / embedding dim.
7	activation_type	Activation function (categorical).
8	activation_param	Activation parameter (e.g., leaky rate).
9	dropout_rate	Dropout (Srivastava et al., 2014) probability.
10	input_pooling_reshape_type	Input pooling/reshape option (Linear layer; categorical).
11	group_num	GroupNorm (Wu & He, 2018) groups (Convolution layer).
12	kernel_size	Kernel size (Convolution layer).
13	stage_wise_pooling_type	Stage-wise pooling option (Convolution layer; categorical).
14	num_heads	Number of heads (GAT (Veličković et al., 2017) / Multi-Head Attention (MHA) (Vaswani et al., 2017)).
15	head_concat_type	Head aggregation option (GAT; categorical).
16	embedding_num	Vocabulary/embedding table size.
17	max_sequence_length	Maximum sequence length.
18	grid_size	KAN grid size.
19	spline_order	KAN spline order.
20	initialization_type	Initialization option (categorical).

Table 7: Task descriptor layout $\mathbf{t} \in \mathbb{R}^2$.

Idx	Field	Description
0	task_type	Task type (categorical).
1	dataset_type	Dataset type (categorical).

B.3 Algorithms

We summarize UHN inference, training, and initialization in pseudocode (Algorithms 1 to 3).

B.4 Descriptor normalization

Before applying Gaussian Fourier features (Tancik et al., 2020), we normalize each scalar attribute of the descriptor vectors independently (in particular, of the index descriptor $\mathbf{v}_i$ and the task-structure descriptors $\mathbf{u}_j$). For a given scalar attribute $x \in [x_{\min}, x_{\max}]$, where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values estimated from the training distribution, we apply

$$\hat{x} = \frac{x - \mu}{\sigma}, \qquad \mu = \frac{x_{\min} + x_{\max}}{2}, \qquad \sigma = \frac{x_{\max} - x_{\min}}{2\sqrt{3}}. \tag{8}$$

We apply this transform attribute-wise (i.e., per dimension) to descriptor vectors; in particular, we write $\hat{\mathbf{v}}_i$ and $\hat{\mathbf{u}}_j$ for the normalized versions of $\mathbf{v}_i$ and $\mathbf{u}_j$, respectively. This choice corresponds to standardizing a uniform distribution.

B.5 Base-Model and Recursive-Hypernetwork Structures

Overview. In our experiments, each generated base model is implemented as a stack of layers, where each layer composes several operations (e.g., a linear/conv operator together with optional normalization, activation, dropout (Srivastava et al., 2014), and shortcut (He et al., 2016)). In particular, whenever we say that UHN generates a model in our experiments, the generated parameters always instantiate one of the base-model families defined in this section, and that base model is the composition of the layer primitives specified below. As described in the main text, every scalar weight is represented by an *index descriptor*. Each layer is represented by a *local structure descriptor*, and the full model is represented by a *global structure descriptor* together with a *task descriptor*. We first define the layer types used throughout the paper, and then specify how these layers are composed into full models.

Algorithm 1 Inference (with recursive generation)

Input: root UHN parameters $\boldsymbol{\theta}$, recursion chain $H_0 \to \cdots \to H_k \to \cdots \to H_K \to f$
Set $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}$ {parameters of H_0}
for $k = 0$ **to** K **do**
  Gather the structure and task descriptors $(\mathbf{s}_{g,k}, \{\mathbf{s}_{\ell,k,j}\}_{j=1}^{L_k}, \mathbf{t}_k)$, and all index descriptors $\{\mathbf{v}_{k,i}\}_{i=1}^{N_k}$ for the target model H_{k+1} when $k < K$ or f when $k = K$
  Generate target model parameters $w_{k,i} \leftarrow H_{\boldsymbol{\theta}_k}\big(\mathbf{v}_{k,i}, \mathbf{s}_{g,k}, \{\mathbf{s}_{\ell,k,j}\}_{j=1}^{L_k}, \mathbf{t}_k\big)$ for all $i \in \{1, \ldots, N_k\}$
  if $k < K$ **then**
    Set $\boldsymbol{\theta}_{k+1} \leftarrow \mathbf{w}_k$ {parameters of generated H_{k+1}}
  else
    return $\mathbf{w}_k$ {parameters of generated f}
  end if
end for

Algorithm 2 Training (with recursive generation)

Input: root UHN parameters $\boldsymbol{\theta}$, total training steps S_{train}, initial task distribution P_{T_0}, recursive task distribution $P_{T_{k+1}|t_k}$, task-conditional model structure distribution $\mathcal{M}_{t_k}$
Initialize optimizer and learning-rate scheduler
for step $= 0$ **to** $S_{\text{train}} - 1$ **do**
  Sample root task $t_0 \sim P_{T_0}$ and root structure $s_0 \sim \mathcal{M}_{t_0}$
  Set $k \leftarrow 0$
  while t_k is recursive **do**
    Sample next task $t_{k+1} \sim P_{T_{k+1}|t_k}$ and structure $s_{k+1} \sim \mathcal{M}_{t_{k+1}}$
    $k \leftarrow k + 1$
  end while
  Let $K \leftarrow k$; the sampled chain $(t_0, s_0), \ldots, (t_K, s_K)$ defines $H_0 \to \cdots \to H_K \to f$
  Set $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}$ {parameters of H_0}
  for $k = 0$ **to** K **do**
    Gather the structure and task descriptors $(\mathbf{s}_{g,k}, \{\mathbf{s}_{\ell,k,j}\}_{j=1}^{L_k}, \mathbf{t}_k)$, and all index descriptors $\{\mathbf{v}_{k,i}\}_{i=1}^{N_k}$ for the target model H_{k+1} when $k < K$ or f when $k = K$
    Generate target model parameters $w_{k,i} \leftarrow H_{\boldsymbol{\theta}_k}\big(\mathbf{v}_{k,i}, \mathbf{s}_{g,k}, \{\mathbf{s}_{\ell,k,j}\}_{j=1}^{L_k}, \mathbf{t}_k\big)$ for all $i \in \{1, \ldots, N_k\}$
    if $k < K$ **then**
      Set $\boldsymbol{\theta}_{k+1} \leftarrow \mathbf{w}_k$ {parameters of generated H_{k+1}}
    else
      Acquire the corresponding dataset $\mathcal{D}_{t_K}$ and loss function ℓ_{t_K} for leaf task t_K
      Sample a mini-batch $(\mathbf{X}, \mathbf{Y}) \sim \mathcal{D}_{t_K}$
      Compute loss $\mathcal{L} \leftarrow \ell_{t_K}\big(f_{\mathbf{w}_k}(\mathbf{X}), \mathbf{Y}\big)$
      Backpropagate $\mathcal{L}$ through the generation path and update $\boldsymbol{\theta}$
    end if
  end for
end for

B.5.1 LAYER PRIMITIVES

Shared conventions. For graph and KAN layers (e.g., GCN (Kipf & Welling, 2016) / GAT (Veličković et al., 2017) / KAN (Liu et al., 2024)), let $X \in \mathbb{R}^{n \times d_{\text{in}}}$ be the input (a batch of n row vectors) and $Y \in \mathbb{R}^{n \times d_{\text{out}}}$ its output. For graph layers with n nodes, let $A \in \{0,1\}^{n \times n}$ be the adjacency matrix and let $\tilde{A} = A + I$ be the adjacency matrix with added self-loops. Let $\tilde{D}$ be the degree matrix of $\tilde{A}$. We write $\mathcal{E}$ for the directed edge set induced by $\tilde{A}$. For a node index $v \in \{1, \ldots, n\}$, we write $\Gamma(v) = \{u \mid (u \to v) \in \mathcal{E}\}$ for the set of source neighbors of v. For MHA (Vaswani et al., 2017) layers, we use $X \in \mathbb{R}^{n \times T \times d}$ and $Y \in \mathbb{R}^{n \times T \times d}$, where T is the sequence length. For embedding layers, let V be the vocabulary size and use token indices $X \in \{1, \ldots, V\}^{n \times T}$ as input and $Y \in \mathbb{R}^{n \times T \times d}$ as output. For convolution layers, let

Algorithm 3 Initialization (with recursive generation)

Input: root UHN parameters $\boldsymbol{\theta}$, total initialization steps S_{init}, per-initialization-level steps S_{lvl}, initial task distribution P_{T_0}, recursive task distribution $P_{T_{k+1}|t_k}$, task-conditional model structure distribution $\mathcal{M}_{t_k}$
for step $= 0$ **to** $S_{\text{init}} - 1$ **do**
  if step $\bmod S_{\text{lvl}} = 0$ **then**
    Initialize (or reset) optimizer and learning-rate scheduler for the new initialization level
  end if
  Sample root task $t_0 \sim P_{T_0}$ and root structure $s_0 \sim \mathcal{M}_{t_0}$
  Set $k \leftarrow 0$
  while t_k is recursive **do**
    Sample next task $t_{k+1} \sim P_{T_{k+1}|t_k}$ and structure $s_{k+1} \sim \mathcal{M}_{t_{k+1}}$
    $k \leftarrow k + 1$
  end while
  Let $K \leftarrow k$; the sampled chain $(t_0, s_0), \ldots, (t_K, s_K)$ defines $H_0 \to \cdots \to H_K \to f$
  Set $\boldsymbol{\theta}_0 \leftarrow \boldsymbol{\theta}$ {parameters of H_0}
  Set $r \leftarrow \min(K, \lfloor \text{step}/S_{\text{lvl}} \rfloor)$ {current initialization level}
  for $k = 0$ **to** r **do**
    Gather the structure and task descriptors $(\mathbf{s}_{g,k}, \{\mathbf{s}_{\ell,k,j}\}_{j=1}^{L_k}, \mathbf{t}_k)$, and all index descriptors $\{\mathbf{v}_{k,i}\}_{i=1}^{N_k}$ for the target model H_{k+1} when $k < K$ or f when $k = K$
    Generate target model parameters $w_{k,i} \leftarrow H_{\boldsymbol{\theta}_k}\big(\mathbf{v}_{k,i}, \mathbf{s}_{g,k}, \{\mathbf{s}_{\ell,k,j}\}_{j=1}^{L_k}, \mathbf{t}_k\big)$ for all $i \in \{1, \ldots, N_k\}$
    if $k < r$ **then**
      Set $\boldsymbol{\theta}_{k+1} \leftarrow \mathbf{w}_k$ {parameters of generated H_{k+1}}
    else
      Compute initialization loss $\mathcal{L}_{\text{init}}$ on $\mathbf{w}_k$ via Equation (4)
      Backpropagate $\mathcal{L}_{\text{init}}$ through the generation path and update $\boldsymbol{\theta}$
    end if
  end for
end for

$X \in \mathbb{R}^{n \times c_{\text{in}} \times h \times w}$ and $Y \in \mathbb{R}^{n \times c_{\text{out}} \times h' \times w'}$ (channels-first). For linear layers, the input X may be a matrix ($X \in \mathbb{R}^{n \times d_{\text{in}}}$), a sequence tensor ($X \in \mathbb{R}^{n \times T \times d_{\text{in}}}$), or an image tensor ($X \in \mathbb{R}^{n \times c_{\text{in}} \times h \times w}$). The output Y is either a matrix ($Y \in \mathbb{R}^{n \times d_{\text{out}}}$) or a sequence tensor ($Y \in \mathbb{R}^{n \times T \times d_{\text{out}}}$). We never produce a 4D output from a linear layer: for image inputs we always apply adaptive pooling to reduce X to 2D before the linear map.

We write $\alpha(\cdot)$ for the chosen activation, $\mathcal{R}(\cdot)$ for dropout and $\mathcal{N}(\cdot)$ for the chosen (optional) normalization. When enabled, $\mathcal{N}$ corresponds to LayerNorm (Ba et al., 2016) for non-convolutional layers, and to GroupNorm (Wu & He, 2018) for convolutional layers; otherwise, $\mathcal{N}$ corresponds to identity map. We write $\langle \cdot, \cdot \rangle$ for the standard Euclidean inner product.

We optionally apply a deterministic pooling operator $\mathcal{P}(\cdot)$ to the layer input. For linear layers, $\mathcal{P}$ is either the identity, reshaping input to a matrix, adaptive pooling (for image-like inputs), averaging over the sequence dimension or selection of the first token (for sequence inputs).

For convolution layers, we use either (i) explicit pooling, where $\mathcal{P}$ is identity/average/max pooling applied to the input, or (ii) stride-2 convolution (in this case we set $\mathcal{P}$ to identity and perform downsampling via the stride).

The shortcut branch is optional: when enabled, $\text{Skip}(X) = X$, and when disabled we set $\text{Skip}(X) = 0$.

When adding a bias vector to a batched tensor, broadcasting is implied.

Linear layer (f_{lin}). A linear layer with bias computes

$$Y = \text{Skip}(\mathcal{P}(X)) + \mathcal{R}\Big(\mathcal{N}(\alpha(\mathcal{P}(X)))W + b\Big), \tag{9}$$

where $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ and $b \in \mathbb{R}^{d_{\text{out}}}$.

- Generated parameters: W, b.
- Active index descriptor fields: layer_idx, layer_type, param_type, out_idx, in_idx.
- Active local structure descriptor fields: layer_idx, layer_type, bias_type, norm_type, shortcut_type, output_size, input_size, activation_type, activation_param, dropout_rate, input_pooling_reshape_type, initialization_type.

Convolution layer (f_{conv}). For a 2D convolution with kernel size $k \times k$, we compute

$$Y = \mathrm{Skip}(X) + \mathcal{R}\Big(\mathrm{Conv}\big(\mathcal{P}(\mathcal{N}(\alpha(X)));\, W\big) + b\Big). \tag{10}$$

where $W \in \mathbb{R}^{c_{\mathrm{out}} \times c_{\mathrm{in}} \times k \times k}$ and $b \in \mathbb{R}^{c_{\mathrm{out}}}$.

- Generated parameters: W, b.
- Active index descriptor fields: layer_idx, layer_type, param_type, out_idx, in_idx, kernel_h_idx, kernel_w_idx.
- Active local structure descriptor fields: layer_idx, layer_type, bias_type, norm_type, shortcut_type, output_size, input_size, activation_type, activation_param, dropout_rate, group_num, kernel_size, stage_wise_pooling_type, initialization_type.

GCN layer ($f_{\mathrm{gcn},A}$). A GCN layer with bias computes

$$Y = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} \Big(\mathcal{R}\big(\mathcal{N}(\alpha(X))\big) W\Big) + b, \tag{11}$$

where $W \in \mathbb{R}^{d_{\mathrm{in}} \times d_{\mathrm{out}}}$ and $b \in \mathbb{R}^{d_{\mathrm{out}}}$.

- Generated parameters: W, b.
- Active index descriptor fields: layer_idx, layer_type, param_type, out_idx, in_idx.
- Active local structure descriptor fields: layer_idx, layer_type, bias_type, norm_type, output_size, input_size, activation_type, activation_param, dropout_rate, initialization_type.

GAT layer ($f_{\mathrm{gat},A}$). A GAT layer with bias applies a linear projection $W \in \mathbb{R}^{d_{\mathrm{in}} \times d_{\mathrm{out}}}$ and reshapes into h heads,

$$Z = \mathcal{R}\big(\mathcal{N}(\alpha(X))\big) W, \qquad Z_v^{(r)} \in \mathbb{R}^{d_h},\ r \in \{1, \ldots, h\}. \tag{12}$$

where $d_h = d_{\mathrm{out}}/h$ denote the per-head output dimension. For each directed edge $(u \to v) \in \mathcal{E}$ and head r, with attention vectors $a_{\mathrm{src}}, a_{\mathrm{dst}} \in \mathbb{R}^{h \times d_h}$,

$$\begin{aligned} e_{u \to v}^{(r)} &= \mathrm{LeakyReLU}_{0.2}\Big(\langle a_{\mathrm{src}}^{(r)}, Z_u^{(r)} \rangle + \langle a_{\mathrm{dst}}^{(r)}, Z_v^{(r)} \rangle\Big), \\ \beta_{u \to v}^{(r)} &= \frac{\exp\big(e_{u \to v}^{(r)}\big)}{\sum_{u' \in \Gamma(v)} \exp\big(e_{u' \to v}^{(r)}\big)}. \end{aligned} \tag{13}$$

The head output is

$$Y_v^{(r)} = \sum_{u \in \Gamma(v)} \beta_{u \to v}^{(r)}\, Z_u^{(r)}, \tag{14}$$

We combine heads by concatenation or averaging and add bias,

$$Y_v = \begin{cases} \mathrm{Concat}\big(Y_v^{(1)}, \ldots, Y_v^{(h)}\big) + b, & \text{(concat)}, \\ \frac{1}{h} \sum_{r=1}^{h} Y_v^{(r)} + b, & \text{(avg)}. \end{cases} \tag{15}$$

where $\mathrm{Concat}(\cdot)$ denotes concatenation across heads and $b \in \mathbb{R}^{d_{\mathrm{out}}}$ (concat) or $b \in \mathbb{R}^{d_h}$ (avg).

- Generated parameters: W, b, a_{src}, a_{dst}.
- Active index descriptor fields: layer_idx, layer_type, param_type, out_idx, in_idx.
- Active local structure descriptor fields: layer_idx, layer_type, bias_type, norm_type, output_size, input_size, activation_type, activation_param, dropout_rate, num_heads, head_concat_type, initialization_type.

Embedding layer (f_{emb}). An embedding layer returns

$$Y = \mathcal{R}(\text{Emb}(X) + P_{1:T}) \,. \tag{16}$$

Here $\text{Emb}(X) \in \mathbb{R}^{n\times T\times d}$ denotes embedding lookup in E and $P_{1:T} \in \mathbb{R}^{T\times d}$ is broadcast over the batch dimension, where $E \in \mathbb{R}^{V\times d}$ is the token embedding table and $P \in \mathbb{R}^{T_{\max}\times d}$ the positional embedding table, with $T_{\max}$ the maximum supported sequence length.

- Generated parameters: E, P.
- Active index descriptor fields: layer_idx, layer_type, param_type, out_idx, embedding_idx, sequence_idx.
- Active local structure descriptor fields: layer_idx, layer_type, output_size (embedding dim), dropout_rate, embedding_num, max_sequence_length, initialization_type.

Multi-head attention (MHA) layer (f_{mha}). An MHA layer with h heads (head dimension $d_h = d/h$) first applies a pre-activation formulation:

$$\bar{X} = \mathcal{N}(\alpha(X)). \tag{17}$$

We then compute projections

$$Q = \bar{X}W_Q + b_Q, \qquad K = \bar{X}W_K + b_K, \qquad V = \bar{X}W_V + b_V, \tag{18}$$

reshape Q, K, V into h heads (each in $\mathbb{R}^{n\times T\times d_h}$), and apply scaled dot-product attention per head:

$$A^{(r)} = \mathcal{R}\Big(\text{softmax}\Big(\tfrac{Q^{(r)}(K^{(r)})^\top}{\sqrt{d_h}}\Big)\Big) V^{(r)}, \qquad r \in \{1,\dots,h\}. \tag{19}$$

We concatenate head outputs $\text{Concat}(A^{(1)},\dots,A^{(h)}) \in \mathbb{R}^{n\times T\times d}$ and apply the output projection,

$$Y = \text{Skip}(X) + \mathcal{R}\Big(\text{Concat}(A^{(1)},\dots,A^{(h)})W_O + b_O\Big). \tag{20}$$

- Generated parameters: $W_Q, W_K, W_V, b_Q, b_K, b_V, W_O, b_O$, where $W_Q, W_K, W_V, W_O \in \mathbb{R}^{d\times d}$ and $b_Q, b_K, b_V, b_O \in \mathbb{R}^d$.
- Active index descriptor fields: layer_idx, layer_type, param_type, out_idx, in_idx.
- Active local structure descriptor fields: layer_idx, layer_type, bias_type, norm_type, shortcut_type, output_size (embedding dim), input_size (also set to embedding dim), activation_type, activation_param, dropout_rate, num_heads, initialization_type.

KAN layer (f_{kan}). A KAN layer combines a base linear map and a spline expansion. We compute

$$Y = \alpha(X)W_{\text{base}} + b_{\text{base}} + \Phi(X;\, g_{\min}, \Delta, \kappa, p)\widetilde{W}_{\text{spline}}, \tag{21}$$

where $W_{\text{base}} \in \mathbb{R}^{d_{\text{in}}\times d_{\text{out}}}$ and $b_{\text{base}} \in \mathbb{R}^{d_{\text{out}}}$. Concretely, let the spline weights be a tensor $W_{\text{spline}} \in \mathbb{R}^{d_{\text{in}}\times(G+p)\times d_{\text{out}}}$ and let spline scales be $S \in \mathbb{R}^{d_{\text{in}}\times d_{\text{out}}}$, where G is the grid size and p is the spline order. We form the scaled spline weight matrix

$$\widetilde{W}_{\text{spline}} = \text{reshape}\big(W_{\text{spline}} \odot S,\ (d_{\text{in}}(G+p),\, d_{\text{out}})\big), \tag{22}$$

where the elementwise multiplication $W_{\text{spline}} \odot S$ uses broadcasting of S along the grid dimension (of size $G+p$). The basis map $\Phi(\cdot)$ stacks B-spline basis functions of order p; for the above batch input $X \in \mathbb{R}^{n\times d_{\text{in}}}$, it returns $\Phi(X;\, g_{\min}, \Delta, \kappa, p) \in \mathbb{R}^{n\times d_{\text{in}}(G+p)}$. We parameterize the knot grid via trainable parameters $g_{\min} \in \mathbb{R}^{d_{\text{in}}}$, $\Delta \in \mathbb{R}^{d_{\text{in}}}$, and $\kappa \in \mathbb{R}^{d_{\text{in}}\times(G+2\,p+1)}$ and set

$$g = g_{\min} + \exp(\Delta) \odot \text{cumsum}(\text{softmax}(\kappa)), \tag{23}$$

where g is the monotonically increasing knot sequence (with $\exp(\Delta)$ broadcast along the grid axis for the elementwise product) and cumsum denotes cumulative summation along the grid axis.

- Generated parameters: W_{base}, b_{base}, W_{spline}, S, $g_{\min}$, Δ, κ.
- Active index descriptor fields: layer_idx, layer_type, param_type, out_idx, in_idx, grid_idx.
- Active local structure descriptor fields: layer_idx, layer_type, bias_type, output_size, input_size, activation_type (base), activation_param, grid_size, spline_order, initialization_type.

B.5.2 MODEL FAMILIES

Model definition. Let the base model be the composition of L layers (functions) $\{f_j\}_{j=1}^{L}$:

$$f(X) = f_L \circ f_{L-1} \circ \cdots \circ f_1(X). \tag{24}$$

In our experiments, generating a base model means generating all trainable parameters that appear in these constituent layers. We represent each base model with a *global structure descriptor* and a *task descriptor*. The shared global structure descriptor fields (active for all models) are model_type and num_layers. Additional global structure descriptor fields are activated depending on the model family: CNNs use cnn_stage_num, Transformers (Vaswani et al., 2017) use num_encoders, and recursive hypernetworks use num_structure_freqs and num_index_freqs. The task descriptor is shared across all models and includes task_type and dataset_type. Below we summarize the model families used in our experiments in a compact mathematical form.

MLP / CNN model. A MLP/CNN is a stack of convolutional layers followed by linear layers

$$f(X) = f_{\text{lin}}^{(J)} \circ \cdots \circ f_{\text{lin}}^{(1)}\big(f_{\text{conv}}^{(K)} \circ \cdots \circ f_{\text{conv}}^{(1)}(X)\big), \tag{25}$$

where K is the number of conv layers and J is the number of linear layers (including the output layer). $f_{\text{lin}}^{(1)}$ includes reshaping/pooling needed to map the its input to a matrix. When $K = 0$, the model reduces to an MLP.

GCN model. A GCN is a composition of graph convolution layers

$$f(X, A) = f_{\text{gcn},A}^{(L)} \circ \cdots \circ f_{\text{gcn},A}^{(1)}(X), \tag{26}$$

GAT model. A GAT is a composition of graph attention layers

$$f(X, A) = f_{\text{gat},A}^{(L)} \circ \cdots \circ f_{\text{gat},A}^{(1)}(X), \tag{27}$$

Transformer model. A Transformer is a composition of an embedding layer, a stack of J encoder blocks, and an output linear layer. Let $\text{Enc}^{(j)}(\cdot)$ denote the jth Transformer encoder block (one MHA layer + K_{j} linear layers)

$$\text{Enc}^{(j)}(X) = f_{\text{lin}}^{(K_{\text{j}})} \circ \cdots \circ f_{\text{lin}}^{(1)} \circ f_{\text{mha}}(X). \tag{28}$$

Then

$$f(X) = f_{\text{lin}}\Big(\text{Enc}^{(J)} \circ \cdots \circ \text{Enc}^{(1)}(f_{\text{emb}}(X))\Big), \tag{29}$$

where the output linear layer f_{lin} includes selecting the first token representation.

KAN model. A KAN model is a composition of KAN layers

$$f(X) = f_{\text{kan}}^{(L)} \circ \cdots \circ f_{\text{kan}}^{(1)}(X), \tag{30}$$

B.5.3 RECURSIVE HYPERNETWORK

A recursive hypernetwork (i.e., a generated hypernetwork) is represented as a conditional mapping

$$H\big(\mathbf{v}_i, \{\mathbf{u}_j\}_{j=1}^{L'}\big) = f_{\text{lin}}\Big(h_{\mathbf{v}}\big(\gamma_{\mathbf{B_v}}(\hat{\mathbf{v}}_i)\big) + h_{\mathbf{u}}\big(\{\gamma_{\mathbf{B_u}}(\hat{\mathbf{u}}_j)\}_{j=1}^{L'}\big)\Big), \tag{31}$$

where $\mathbf{v}_i$ is the index descriptor, $\mathbf{u}_j = [\mathbf{s}_g; \mathbf{t}; \mathbf{s}_{\ell,j}]$ are L' per-layer task-structure descriptors, and $\hat{\mathbf{v}}_i$ and $\hat{\mathbf{u}}_j$ denote the corresponding attribute-wise normalized descriptors. Here $\gamma_{\mathbf{B}}(\cdot)$ denotes the fixed Gaussian Fourier feature map (Tancik et al., 2020), with fixed matrices $\mathbf{B_v}$ and $\mathbf{B_u}$. We write $\boldsymbol{\phi}_i := \gamma_{\mathbf{B_v}}(\hat{\mathbf{v}}_i)$ for the index Fourier features and $\boldsymbol{\psi}_j := \gamma_{\mathbf{B_u}}(\hat{\mathbf{u}}_j)$ for the per-layer token Fourier features.

For generality, we write the index branch $h_{\mathbf{v}}$ as a composition of K linear layers,

$$h_{\mathbf{v}}(\boldsymbol{\phi}_i) = f_{\text{lin},\mathbf{v}}^{(K)} \circ \cdots \circ f_{\text{lin},\mathbf{v}}^{(1)}(\boldsymbol{\phi}_i), \tag{32}$$

Similarly, we write the task-structure branch $h_{\mathbf{u}}$ as a MHA layer followed by J linear layers,

$$h_{\mathbf{u}}\big(\{\boldsymbol{\psi}_j\}_{j=1}^{L'}\big) = f_{\mathrm{lin},\mathbf{u}}^{(J)} \circ \cdots \circ f_{\mathrm{lin},\mathbf{u}}^{(1)} \circ f_{\mathrm{mha}}\big(\{\boldsymbol{\psi}_j\}_{j=1}^{L'}\big), \tag{33}$$

where one of the linear layers in the task-structure branch includes averaging over the token (sequence) dimension. Finally, the fused representation is passed through the output linear layer f_{lin} to predict the weight scalar.

In the recursive setting, we reuse the same descriptor normalization statistics and Fourier matrices as the root UHN.

B.6 Initialization target statistics

We define an initialization target for each generated *parameter group* g (i.e., each layer *component* such as a weight matrix, bias vector, or attention projection (Vaswani et al., 2017; Veličković et al., 2017)) in terms of a desired mean and standard deviation $(\mu^*(g), \sigma^*(g))$. During the initialization phase, we match the parameter statistics $(\mu(g), \sigma(g))$ of the generated component g to this target in order to stabilize optimization of the main training phase. Throughout this subsection, $\mathcal{U}(a, b)$ denotes the continuous uniform distribution on $[a, b]$, and $\mathcal{N}(\mu, \sigma^2)$ denotes a Gaussian distribution with mean μ and variance σ^2.

Default vs. zero targets. We support two initialization modes: (i) DEFAULT, where targets mirror common library initializations (Paszke et al., 2019; Fey & Lenssen, 2019), and (ii) ZERO, where $(\mu^*, \sigma^*) = (0, 0)$ for all parameter groups. If not specified otherwise, we use DEFAULT.

Default target statistics by layer. For DEFAULT, we use the following targets. For Linear, convolution, embedding, and MHA (Vaswani et al., 2017) layers, these targets are chosen to match PyTorch default initializations (Paszke et al., 2019). For GCN (Kipf & Welling, 2016) and GAT (Veličković et al., 2017) layers, we match the defaults used by PyTorch Geometric (Fey & Lenssen, 2019). For KAN layers, we follow the original KAN paper with several modifications (Liu et al., 2024), as described below.

- **Linear layer.** For weights and biases we use $\mu^* = 0$ and $\sigma^* = \frac{1}{\sqrt{3\, d_{\mathrm{in}}}}$, where d_{in} is the input dimension.
- **Convolution layer.** For a $k \times k$ convolution with c_{in} input channels we use $\mu^* = 0$ and $\sigma^* = \frac{1}{\sqrt{3\, c_{\mathrm{in}} k^2}}$ (for both weights and biases).
- **GCN layer.** For weights we use $\mu^* = 0$ and $\sigma^* = \sqrt{\frac{2}{d_{\mathrm{in}} + d_{\mathrm{out}}}}$, where d_{in} and d_{out} are the input/output dimensions; biases are set to zero, i.e., $(\mu_{\mathrm{b}}^*, \sigma_{\mathrm{b}}^*) = (0, 0)$.
- **GAT layer.** For the main weights we use $\mu^* = 0$ and $\sigma^* = \sqrt{\frac{2}{d_{\mathrm{in}} + h\, d_{\mathrm{out}}}}$, where d_{in} is the input dimension, d_{out} is the output dimension per head, and h is the number of heads; biases are set to zero. For attention parameters (source and target) we use $\mu^* = 0$ and $\sigma^* = \sqrt{\frac{2}{d_{\mathrm{out}} + h}}$.
- **Embedding layer.** Embedding weights use $(\mu^*, \sigma^*) = (0, 1)$, while positional embeddings use $(0, 0)$.
- **Multi-head attention (MHA).** For Q/K/V projection weights we use a Glorot-style (Glorot & Bengio, 2010) target $\mu^* = 0$ and $\sigma^* = \sqrt{\frac{2}{d+d}} = \sqrt{\frac{1}{d}}$, where d is the MHA dimension; all projection biases are set to zero. For the output projection weight we match $\mathcal{U}(-1/\sqrt{d}, 1/\sqrt{d})$, i.e., $\mu^* = 0$ and $\sigma^* = \frac{1}{\sqrt{3d}}$, and set its bias to zero.
- **KAN layer.** Base weights and spline scales match $\mathcal{U}(-1/\sqrt{d_{\mathrm{in}}}, 1/\sqrt{d_{\mathrm{in}}})$ (thus $\mu^* = 0$, $\sigma^* = 1/\sqrt{3\, d_{\mathrm{in}}}$, with d_{in} the layer input dimension). Base biases are set to zero. Spline weights follow $\mathcal{N}(0, 0.1^2)$. Grid parameters are initialized deterministically with zero variance: grid lower bounds are set to -1, grid lengths are set to $\log 2$, and grid knots are set to 0.

B.7 Common Training and Reporting Details

All experiments are implemented in PyTorch (Paszke et al., 2019) and run on a single NVIDIA RTX 4090 GPU. We enable automatic mixed precision (AMP, FP16) to reduce memory usage and speed up training.

When training UHN, we use AdamW (Loshchilov & Hutter, 2017) with a cosine learning-rate schedule (Loshchilov & Hutter, 2016) for both the initialization and main training phases. In the main training phase, we use 5 linear learning-rate warmup (Goyal et al., 2017) epochs for single-model and multi-model experiments and 1000 warmup steps for multi-task and recursive experiments. For classification tasks (image/graph/text), we use mini-batches of size 256; for formula regression, we use full-batch training. Across UHN, direct training, and the hypernetwork baselines, we use no weight decay in any experiment.

We report mean $\pm$ standard deviation over multiple random seeds, with exact seed or run counts stated in the corresponding experiment or table caption. Experiment-specific dataset splits, architectures, model-family sampling, and hyperparameter choices are documented in the corresponding appendix sections.

B.8 Hyperparameter selection

Here we summarize the hyperparameters and sweep protocol used across experiments.

Architecture hyperparameters. The main model (architecture) hyperparameters are the number of Fourier-feature (Tancik et al., 2020) frequencies $F_{\mathbf{v}}$ for index encoding, the Fourier-feature scale σ (for both index encoding and structure/task encoding), the UHN hidden dimension d, and the number of UHN residual blocks N_{blk}. When used, the task-structure encoder introduces additional hyperparameters, including the number of Fourier-feature frequencies $F_{\mathbf{u}}$ for task-structure descriptors and the number of attention heads (Vaswani et al., 2017) h. In all experiments, we fix $\sigma = 100$, and (when using the task-structure encoder) we set $(F_{\mathbf{u}}, h) = (32, 4)$.

Unless otherwise specified, we use $(F_{\mathbf{v}}, d, N_{\mathrm{blk}}) = (1024, 64, 2)$ for the single-model MNIST (Lecun et al., 1998) experiment and for recursive experiments, and $(F_{\mathbf{v}}, d, N_{\mathrm{blk}}) = (2048, 128, 2)$ for all other experiments.

Swept hyperparameters. For each experimental setting, we sweep the initialization hyperparameters $(S_{\mathrm{init}}, \eta_{\mathrm{init}})$, where S_{init} denotes the number of initialization steps and η_{init} denotes the initialization learning rate. For the training phase, we sweep either $(S_{\mathrm{train}}, \eta_{\mathrm{train}})$ when using step-based training budgets (multi-task/recursive settings) or $(E_{\mathrm{train}}, \eta_{\mathrm{train}})$ when using epoch-based training budgets (single-/multi-model settings), where S_{train} denotes the number of training steps, E_{train} denotes the number of training epochs, and η_{train} denotes the training learning rate.

Sweep protocol. To reduce search cost, we use a staged (rather than fully joint) sweep:

1. Fix the training budget and sweep η_{train} with $S_{\mathrm{init}} = 0$.
2. Fix the selected training hyperparameters and sweep $(S_{\mathrm{init}}, \eta_{\mathrm{init}})$.
3. Fix the selected initialization hyperparameters and refine the training hyperparameters.

We denote the fixed training budget in the first sweeping stage as E_0 (epochs) in single-/multi-model settings and as S_0 (steps) in multi-task/recursive settings.

Validation/selection criterion. In the single-model setting, validation is performed on the single target model. For multi-model experiments, we select hyperparameters by average validation performance over architectures in M_{val}. For multi-task experiments, we compare configurations using a rank-based aggregate across tasks (i.e., Borda count (Borda, 1781)) with task-specific guardrails. The selection criterion for recursive experiments follows the multi-task protocol.

Table 8: Citation-network dataset statistics, as reported in (Yang et al., 2016), used in our experiments.

	Cora	CiteSeer	PubMed
Nodes	2708	3327	19717
Edges	5429	4732	44338
Features	1433	3703	500
Classes	7	6	3

C SINGLE-MODEL EXPERIMENTS

This section provides implementation and evaluation details supporting the single-model experiments in Section 4.2, including datasets, base model architectures, additional results and the hyperparameter selection.

C.1 DATASETS

MNIST. MNIST (Lecun et al., 1998) is an image classification dataset with 10 classes, 60,000 training samples, and 10,000 test samples (28×28 grayscale). During hyperparameter sweeps we reserve 20% of the training set for validation; final results are obtained by training on the full training set. We apply standard per-pixel normalization.

CIFAR-10. CIFAR-10 (Krizhevsky et al., 2009) is an image classification dataset with 10 classes, 50,000 training samples, and 10,000 test samples (32×32 RGB). During sweeps we reserve 10% of the training set for validation. For training we use augmentation following (Zagoruyko & Komodakis, 2016): 4-pixel reflection padding, random horizontal flip, random 32×32 crops, and per-channel normalization; for testing we use only normalization.

Citation networks (Cora, CiteSeer, PubMed). Cora, CiteSeer, and PubMed (Sen et al., 2008) are citation-network benchmarks for graph node classification. Table 8 summarizes dataset statistics, as reported in (Yang et al., 2016), where nodes correspond to documents, edges correspond to citations, features are bag-of-words vectors. We use the standard splits with 140/500/1000 nodes for train/val/test on Cora, 120/500/1000 on CiteSeer, and 60/500/1000 on PubMed following (Kipf & Welling, 2016). We normalize node features for both training and evaluation.

AG News. AG News (Zhang et al., 2015) is a text classification dataset with 4 classes, 120,000 training samples, and 7,600 test samples. During sweeps we reserve 20% of the training set for validation. We use a simple English tokenizer (lowercasing + regex tokenization), build a vocabulary of the $5{,}000$ most frequent tokens, including `<CLS>` (classification token), `<PAD>` (padding), `<UNK>` (unknown), and use a maximum sequence length of 128 (including the first `<CLS>` token). We apply dynamic padding per batch and use a padding mask.

IMDB. IMDB (Maas et al., 2011) is a binary text classification dataset with 25,000 training samples and 25,000 test samples. During sweeps we reserve 20% of the training set for validation. We use the same tokenizer and vocabulary construction as AG News, but use a maximum sequence length of 512.

Formula regression (special functions). We evaluate formula regression on 15 special functions (following the KAN benchmark (Liu et al., 2024); listed in Table 10). For each function, we sample 1,000 training and 1,000 test inputs uniformly over $(x_1, x_2) \in [0, 1)^2$ (except for special functions yv and kv, where we sample $x_2 \in [\epsilon, 1)$ to avoid numerical instabilities; we use $\epsilon = 0.1$). For special function ellipj the target is $\mathbb{R}^4$; otherwise it is scalar. We reserve 20% of the training set for validation, use full-batch training, and normalize inputs to $[-1, 1)$.

C.2 BASE-MODEL ARCHITECTURES

The following base models are instances of the model families in Appendix B.5.2; their parameters are generated by UHN. All base-model input/output dimensions are adapted to the corresponding dataset.

Table 9: Complete single-model results across heterogeneous classification tasks. Each row uses a separately trained UHN instance with the same generator design. #Params denotes parameter count; Acc. denotes test accuracy.

Task type	Dataset	Model	#Params (Direct)	#Params (UHN)	Acc. (Direct)	Acc. (UHN)
Image	MNIST	MLP	118,282	158,613	0.9837 ± 0.0007	0.9841 ± 0.0006
Image	MNIST	CNN-8	74,762	158,613	0.9938 ± 0.0004	0.9944 ± 0.0010
Image	CIFAR-10	CNN-20	269,034	612,117	0.8999 ± 0.0027	0.8993 ± 0.0016
Graph	Cora	GCN	92,231	612,117	0.8172 ± 0.0039	0.7950 ± 0.0069
Graph	CiteSeer	GCN	237,446	612,117	0.6996 ± 0.0051	0.6529 ± 0.0190
Graph	PubMed	GCN	32,259	612,117	0.7691 ± 0.0028	0.7815 ± 0.0075
Graph	Cora	GAT	92,373	612,117	0.8132 ± 0.0094	0.7981 ± 0.0091
Graph	CiteSeer	GAT	237,586	612,117	0.6999 ± 0.0051	0.6819 ± 0.0068
Graph	PubMed	GAT	33,779	612,117	0.7700 ± 0.0033	0.7801 ± 0.0037
Text	AG News	Transformer-2L	378,372	612,117	0.9186 ± 0.0008	0.9099 ± 0.0005
Text	IMDB	Transformer-1L	377,858	612,117	0.8853 ± 0.0007	0.8638 ± 0.0005

- **MLP (MNIST).** Two hidden layers of width 128; bias enabled; activation: LeakyReLU (slope 0.1); normalization: LayerNorm (Ba et al., 2016); no dropout (Srivastava et al., 2014); residual shortcuts (He et al., 2016) enabled.
- **CNN-8 (MNIST).** Four-stage CNN; layers per stage 1/2/2/2; widths 16/16/32/64; bias enabled; kernel size 3; downsampling (via stride-2 convolution) in the third and fourth stages; GroupNorm (Wu & He, 2018) with 4 groups; average pooling after convolutions; LeakyReLU (0.1); no dropout; residual shortcuts enabled; classifier is a final linear layer (overall, a lightweight pre-activation ResNet-style (He et al., 2016) variant).
- **CNN-20/32/44/56 (CIFAR-10).** Same as CNN-8 (MNIST), but with layers per stage $1/n/n/n$ and $n \in \{6, 10, 14, 18\}$.
- **CNN-20-2/4/6 (CIFAR-10).** Same as CNN-20 (CIFAR-10), but with widths $16k/16k/32k/64k$ and $k \in \{2, 4, 6\}$.
- **GCN (Kipf & Welling, 2016).** One hidden layer of width 64; bias enabled; activation: LeakyReLU (0.1); dropout 0.5; no normalization.
- **GAT (Veličković et al., 2017).** One hidden layer with 8 heads and head dimension 8; 8 output heads for PubMed and 1 otherwise; bias enabled; activation: ELU (Clevert et al., 2015); dropout 0.6; no normalization.
- **Transformer-2L (Vaswani et al., 2017) (AG News).** Two encoder blocks; 2 attention heads; 2 feed-forward layers per encoder; embedding dimension 64; bias and residual shortcuts enabled; normalization: LayerNorm; activation: LeakyReLU (0.1); dropout 0.2.
- **Transformer-1L (IMDB).** Same as Transformer-2L (AG News), but with 1 encoder block and dropout 0.4.
- **KAN-g5/g10 (Liu et al., 2024).** One hidden layer of width 5; grid size 5 by default (KAN-g5) and 10 for special functions jv and yv (KAN-g10); spline order 3; bias enabled; base activation: SiLU (Hendrycks & Gimpel, 2016).

C.3 Complete classification results (heterogeneous single-model experiment)

Table 9 reports the complete classification results corresponding to the representative cases summarized in the main text. For image and text tasks, results are reported over 3 random seeds; graph results are reported over 10 random seeds. All entries report mean $\pm$ standard deviation of test accuracy.

C.4 Formula-regression results (heterogeneous single-model experiment)

Table 10 reports the full per-function formula-regression results for the heterogeneous single-model experiment, as a continuation of Section 4.2. Results are reported as mean $\pm$ standard deviation of test RMSE over 3 random seeds.

Table 10: Heterogeneous single-model results for formula regression on the 15 special functions following the KAN benchmark (Liu et al., 2024). Model (#Params) reports the base model architecture and its parameter count; UHN has 612,117 parameters for all functions. RMSE denotes test root-mean-squared error (lower is better).

Function	Model (#Params)	RMSE (Direct)	RMSE (UHN)
Jacobian elliptic functions (ellipj)	KAN-g5 (407)	$6.78 \times 10^{-4} \pm 1.32 \times 10^{-4}$	$6.44 \times 10^{-4} \pm 5.20 \times 10^{-5}$
Incomplete elliptic integral of the first kind (ellipkinc)	KAN-g5 (254)	$5.37 \times 10^{-4} \pm 1.40 \times 10^{-4}$	$5.99 \times 10^{-4} \pm 9.32 \times 10^{-5}$
Incomplete elliptic integral of the second kind (ellipeinc)	KAN-g5 (254)	$6.02 \times 10^{-4} \pm 2.58 \times 10^{-4}$	$5.23 \times 10^{-4} \pm 4.51 \times 10^{-5}$
Bessel function of the first kind (jv)	KAN-g10 (364)	$1.80 \times 10^{-3} \pm 8.04 \times 10^{-4}$	$2.32 \times 10^{-3} \pm 1.12 \times 10^{-3}$
Bessel function of the second kind (yv)	KAN-g10 (364)	$1.09 \times 10^{-2} \pm 4.05 \times 10^{-3}$	$8.19 \times 10^{-3} \pm 3.67 \times 10^{-3}$
Modified Bessel function of the second kind (kv)	KAN-g5 (254)	$2.11 \times 10^{-2} \pm 1.26 \times 10^{-2}$	$1.04 \times 10^{-2} \pm 4.77 \times 10^{-3}$
Modified Bessel function of the first kind (iv)	KAN-g5 (254)	$2.94 \times 10^{-3} \pm 1.74 \times 10^{-3}$	$2.91 \times 10^{-3} \pm 9.40 \times 10^{-4}$
Associated Legendre function ($m = 0$) (lpmv0)	KAN-g5 (254)	$7.06 \times 10^{-4} \pm 1.57 \times 10^{-4}$	$7.28 \times 10^{-4} \pm 6.04 \times 10^{-5}$
Associated Legendre function ($m = 1$) (lpmv1)	KAN-g5 (254)	$2.22 \times 10^{-3} \pm 1.42 \times 10^{-4}$	$1.57 \times 10^{-3} \pm 3.48 \times 10^{-4}$
Associated Legendre function ($m = 2$) (lpmv2)	KAN-g5 (254)	$2.74 \times 10^{-4} \pm 2.95 \times 10^{-5}$	$9.61 \times 10^{-4} \pm 1.16 \times 10^{-4}$
Spherical harmonics ($m = 0, n = 1$) (sph_harm01)	KAN-g5 (254)	$1.62 \times 10^{-4} \pm 2.66 \times 10^{-5}$	$3.70 \times 10^{-4} \pm 1.64 \times 10^{-4}$
Spherical harmonics ($m = 1, n = 1$) (sph_harm11)	KAN-g5 (254)	$2.11 \times 10^{-4} \pm 4.92 \times 10^{-5}$	$4.52 \times 10^{-4} \pm 1.79 \times 10^{-4}$
Spherical harmonics ($m = 0, n = 2$) (sph_harm02)	KAN-g5 (254)	$2.22 \times 10^{-4} \pm 9.92 \times 10^{-5}$	$4.40 \times 10^{-4} \pm 1.85 \times 10^{-4}$
Spherical harmonics ($m = 1, n = 2$) (sph_harm12)	KAN-g5 (254)	$2.44 \times 10^{-4} \pm 1.34 \times 10^{-5}$	$5.44 \times 10^{-4} \pm 1.63 \times 10^{-4}$
Spherical harmonics ($m = 2, n = 2$) (sph_harm22)	KAN-g5 (254)	$1.54 \times 10^{-4} \pm 4.90 \times 10^{-6}$	$5.33 \times 10^{-4} \pm 8.24 \times 10^{-5}$

Table 11: UHN hyperparameter sweep grids (heterogeneous single-model experiment). Table entries list the sweep-grid values for η_{init}, S_{init}, η_{train}, and E_{train}; we use "All" when the same grid applies to all datasets/models within that task.

Task	Dataset	Model	E_0	η_{init}	S_{init}	η_{train}	E_{train}
Image	MNIST	MLP	100	{5e−5, 1e−4, 2e−4}	{50, 100, 200}	{5e−5, 1e−4, 2e−4}	{50, 100, 200}
Image	MNIST	CNN-8	50	{5e−5, 1e−4, 2e−4}	{50, 100, 200}	{5e−5, 1e−4, 2e−4}	{30, 50, 100}
Image	CIFAR-10	CNN-20	200	{5e−5, 1e−4, 2e−4}	{50, 100, 200}	{5e−5, 1e−4, 2e−4}	{200, 400, 800}
Text	AG News	Transformer-2L	50	{5e−5, 1e−4, 2e−4}	{50, 100, 200}	{5e−5, 1e−4, 2e−4}	{30, 50, 100}
Text	IMDB	Transformer-1L	100	{5e−5, 1e−4, 2e−4}	{50, 100, 200}	{5e−5, 1e−4, 2e−4}	{50, 100, 200}
Graph	All	All	200	{5e−5, 1e−4, 2e−4}	{50, 100, 200}	{2e−5, 5e−5, 1e−4}	{100, 200, 400}
Formula	All	All	4000	{5e−5, 1e−4, 2e−4}	{50, 100, 200}	{2e−5, 5e−5, 1e−4}	{4000}

C.5 Hyperparameter selection

C.5.1 Heterogeneous single-model experiment

For UHN, we select all optimization hyperparameters by grid search, following the sweep protocol described in Appendix B.8. Concretely, we sweep (i) the initialization-stage learning rate and steps $(\eta_{\text{init}}, S_{\text{init}})$ and (ii) the training-stage learning rate and epochs $(\eta_{\text{train}}, E_{\text{train}})$. Table 11 summarizes the sweep grids, and Table 12 reports the best hyperparameters selected by the grid search.

For the direct-training baseline, we use AdamW (Loshchilov & Hutter, 2017) with a cosine learning-rate schedule (Loshchilov & Hutter, 2016) and 5 linear warmup epochs (Goyal et al., 2017). We select the optimizer learning rate and the number of training epochs $(\eta_{\text{train}}, E_{\text{train}})$ via grid search. Table 13 summarizes the grids used, and Table 14 reports the best configuration selected for each experiment.

C.6 Target-Size Scaling: Complete Results and Baseline Details

Complete predictive results. Table 15 reports the complete CIFAR-10 test-accuracy results used in the target-size scaling study.

Parameter counts. Table 16 reports the parameter counts for Direct and all four hypernetwork methods. For Ha-HN, the reported count includes its hypernetwork parameters, learned embeddings, and directly optimized target-network parameters.

Baseline configurations and training protocol. Unless otherwise specified, trainable network parameters use PyTorch default initialization (Paszke et al., 2019). All methods use AdamW with a cosine learning-rate schedule and 5 linear warmup epochs in this scaling study. For UHN and Direct,

Table 12: UHN best hyperparameters selected by grid search (heterogeneous single-model experiment).

Task	Dataset	Model	η_{init}	S_{init}	η_{train}	E_{train}
Image	MNIST	MLP	$2e{-4}$	100	$2e{-4}$	100
Image	MNIST	CNN-8	$2e{-4}$	200	$1e{-4}$	100
Image	CIFAR-10	CNN-20	0	0	$2e{-4}$	800
Graph	Cora	GCN	$1e{-4}$	200	$2e{-5}$	400
Graph	CiteSeer	GCN	$1e{-4}$	50	$1e{-4}$	200
Graph	PubMed	GCN	$1e{-4}$	200	$5e{-5}$	200
Graph	Cora	GAT	$2e{-4}$	100	$5e{-5}$	400
Graph	CiteSeer	GAT	$2e{-4}$	200	$5e{-5}$	400
Graph	PubMed	GAT	$2e{-4}$	100	$2e{-5}$	400
Text	AG News	Transformer-2L	0	0	$5e{-5}$	50
Text	IMDB	Transformer-1L	$5e{-5}$	100	$1e{-4}$	100
Formula	ellipj	KAN-g5	$1e{-4}$	200	$1e{-4}$	4000
Formula	ellipkinc	KAN-g5	$2e{-4}$	50	$1e{-4}$	4000
Formula	ellipeinc	KAN-g5	$1e{-4}$	50	$1e{-4}$	4000
Formula	jv	KAN-g10	$2e{-4}$	200	$5e{-5}$	4000
Formula	yv	KAN-g10	$2e{-4}$	100	$1e{-4}$	4000
Formula	kv	KAN-g5	$2e{-4}$	200	$1e{-4}$	4000
Formula	iv	KAN-g5	$5e{-5}$	100	$1e{-4}$	4000
Formula	lpmv0	KAN-g5	$5e{-5}$	200	$5e{-5}$	4000
Formula	lpmv1	KAN-g5	$5e{-5}$	200	$5e{-5}$	4000
Formula	lpmv2	KAN-g5	$2e{-4}$	200	$1e{-4}$	4000
Formula	sph_harm01	KAN-g5	0	0	$1e{-4}$	4000
Formula	sph_harm11	KAN-g5	$5e{-5}$	50	$5e{-5}$	4000
Formula	sph_harm02	KAN-g5	$2e{-4}$	200	$2e{-5}$	4000
Formula	sph_harm12	KAN-g5	$2e{-4}$	100	$1e{-4}$	4000
Formula	sph_harm22	KAN-g5	$2e{-4}$	200	$2e{-5}$	4000

Table 13: Direct-training hyperparameter sweep grids (heterogeneous single-model experiment). Table entries list the sweep-grid values for η_{train} and E_{train}.

Task	Dataset	Model	η_{train}	E_{train}
Image	MNIST	MLP	$\{0.002, 0.005, 0.01\}$	$\{50, 100, 200\}$
Image	MNIST	CNN-8	$\{0.005, 0.01, 0.02\}$	$\{30, 50, 100\}$
Image	CIFAR-10	CNN-20	$\{0.002, 0.005, 0.01\}$	$\{100, 200, 400\}$
Graph	Cora	GCN	$\{0.002, 0.005, 0.01\}$	$\{100, 200, 400\}$
Graph	CiteSeer	GCN/GAT	$\{0.001, 0.002, 0.005\}$	$\{100, 200, 400\}$
Graph	PubMed	GCN	$\{0.002, 0.005, 0.01\}$	$\{100, 200, 400\}$
Graph	Cora	GAT	$\{0.005, 0.01, 0.02\}$	$\{50, 100, 200\}$
Graph	PubMed	GAT	$\{0.01, 0.02, 0.05\}$	$\{100, 200, 400\}$
Text	AG News	Transformer-2L	$\{0.0005, 0.001, 0.002\}$	$\{50, 100, 200\}$
Text	IMDB	Transformer-1L	$\{0.0002, 0.0005, 0.001\}$	$\{50, 100, 200\}$
Formula	ellipj/ellipkinc/ellipeinc	KAN-g5	$\{0.01, 0.02, 0.05\}$	$\{4000\}$
Formula	jv/yv	KAN-g10	$\{0.005, 0.01, 0.02\}$	$\{4000\}$
Formula	kv/iv	KAN-g5	$\{0.005, 0.01, 0.02\}$	$\{4000\}$
Formula	lpmv0/lpmv1/lpmv2	KAN-g5	$\{0.02, 0.05, 0.1\}$	$\{4000\}$
Formula	sph_harm01/sph_harm02/sph_harm11	KAN-g5	$\{0.02, 0.05, 0.1\}$	$\{4000\}$
Formula	sph_harm12/sph_harm22	KAN-g5	$\{0.02, 0.05, 0.1\}$	$\{4000\}$

we reuse the hyperparameters selected on CNN-20 in the cross-family experiment (Tables 12 and 14) for all six larger targets, CNN-32/44/56 and CNN-20-2/4/6, without additional re-sweeps.

Ha-HN follows the HyperNetwork construction of Ha et al. (2016), using an embedding size of 188 in our implementation. Learned kernel embeddings are mapped through linear networks to fixed-size convolution-kernel chunks, with larger kernels formed by concatenating multiple generated chunks; the first convolution layer, final linear layer, and biases of the intermediate convolution layers are directly optimized. We sweep $(\eta_{\text{train}}, E_{\text{train}})$ on CNN-20 and reuse the selected setting for all remaining depth- and width-scaled targets.

For Chunked-HN (Von Oswald et al., 2019), we use chunk size 6000, hidden dimension 100, embedding dimension 64, and 2 hidden layers. We sweep $(\eta_{\text{train}}, E_{\text{train}})$ on CNN-20 and reuse the selected setting for all remaining targets.

Table 14: Direct-training best hyperparameters selected by grid search (heterogeneous single-model experiment).

Task	Dataset	Model	η_{train}	E_{train}
Image	MNIST	MLP	0.01	200
Image	MNIST	CNN-8	0.005	50
Image	CIFAR-10	CNN-20	0.005	400
Graph	Cora	GCN	0.01	100
Graph	CiteSeer	GCN	0.005	100
Graph	PubMed	GCN	0.005	200
Graph	Cora	GAT	0.02	100
Graph	CiteSeer	GAT	0.002	200
Graph	PubMed	GAT	0.02	200
Text	AG News	Transformer-2L	0.0005	100
Text	IMDB	Transformer-1L	0.001	50
Formula	ellipj/ellipkinc/ellipeinc	KAN-g5	0.02	4000
Formula	jv/yv	KAN-g10	0.02	4000
Formula	kv/iv	KAN-g5	0.02	4000
Formula	lpmv0	KAN-g5	0.05	4000
Formula	lpmv1/lpmv2	KAN-g5	0.02	4000
Formula	sph_harm01/sph_harm02/sph_harm11	KAN-g5	0.1	4000
Formula	sph_harm12/sph_harm22	KAN-g5	0.05	4000

Table 15: Complete target-size scaling results on CIFAR-10. Entries report test accuracy. "Failed" indicates that Ha-HN did not train successfully on CNN-20-6 using the hyperparameters selected on CNN-20 and reused throughout the scaling experiment.

Model	Direct	Ha-HN	Chunked-HN	Transformer-HN	UHN
CNN-20	0.8999 ± 0.0027	0.9070 ± 0.0039	0.9010 ± 0.0008	0.9021 ± 0.0019	0.8993 ± 0.0016
CNN-32	0.9069 ± 0.0028	0.9124 ± 0.0021	0.9010 ± 0.0002	0.9067 ± 0.0013	0.9035 ± 0.0026
CNN-44	0.9076 ± 0.0010	0.9161 ± 0.0017	0.9063 ± 0.0030	0.9075 ± 0.0018	0.9043 ± 0.0031
CNN-56	0.9043 ± 0.0036	0.9159 ± 0.0004	0.9027 ± 0.0021	0.9046 ± 0.0006	0.9069 ± 0.0015
CNN-20-2	0.9258 ± 0.0010	0.9212 ± 0.0025	0.9213 ± 0.0012	0.9225 ± 0.0011	0.9202 ± 0.0013
CNN-20-4	0.9338 ± 0.0014	0.9277 ± 0.0028	0.9241 ± 0.0021	0.9235 ± 0.0002	0.9274 ± 0.0013
CNN-20-6	0.9363 ± 0.0021	Failed	0.9281 ± 0.0024	0.9260 ± 0.0027	0.9233 ± 0.0037

For Transformer-HN, learned chunk embeddings are processed as a sequence by a Transformer encoder (Vaswani et al., 2017); the contextualized representations are decoded by a shared linear layer and concatenated to form the target-network weights. The embedding dimension and feed-forward dimension are both 100, the encoder uses 4 attention heads with dropout set to 0, and the chunk size is 5500. We keep $E_{\text{train}} = 800$, sweep $\eta_{\text{train}} \in \{0.001, 0.002, 0.005\}$ on CNN-20, and reuse the selected learning rate 0.002 for all remaining depth- and width-scaled targets.

For CNN-20-4 and CNN-20-6, generating all target weights in a single UHN forward pass exceeds the memory available on the RTX 4090. We therefore divide the parameter queries into batches, each corresponding to at most 1.28M generated weights, then evaluate UHN on the batches sequentially and concatenate the outputs to instantiate the complete target model. During training, we additionally use gradient checkpointing (Chen et al., 2016) to reduce activation memory; the same sequential generation is used when generating the final target model. Because these UHN runs are substantially more time-consuming, CNN-20-4 and CNN-20-6 are evaluated over 2 seeds rather than 3.

None of the reported Ha-HN, Chunked-HN, Transformer-HN, or UHN results uses post-generation fine-tuning.

C.7 Computational Cost of Target-Size Scaling

The main target-size scaling experiment focuses on generator size and predictive performance. Table 18 reports training and model-generation time for Direct, Transformer-HN, and UHN.

Table 16: Parameter counts for the target-size scaling study. UHN retains the same generator architecture and a generator parameter count of 612,117 as the target model grows.

Model	Direct	Ha-HN	Chunked-HN	Transformer-HN	UHN
CNN-20	269,034	619,306	625,480	621,400	612,117
CNN-32	463,018	635,546	627,592	625,000	612,117
CNN-44	657,002	651,786	629,640	628,500	612,117
CNN-56	850,986	668,026	631,688	632,000	612,117
CNN-20-2	1,072,586	686,490	634,056	636,100	612,117
CNN-20-4	4,283,274	951,706	668,296	694,400	612,117
CNN-20-6	9,632,074	1,391,386	725,384	791,700	612,117

Table 17: Hyperparameter sweeps used for the hypernetwork baselines in the target-size scaling experiment.

Hypernetwork	η_{train} grid	E_{train} grid	Best η_{train}	Best E_{train}
Ha-HN	$\{0.002, 0.005, 0.01\}$	$\{200, 400, 800\}$	0.002	800
Chunked-HN	$\{0.001, 0.002, 0.005\}$	$\{200, 400, 800\}$	0.002	800
Transformer-HN	$\{0.001, 0.002, 0.005\}$	$\{800\}$	0.002	800

D Multi-Model Experiments

This section provides additional experimental details supporting the multi-model experiments in Section 4.4, including model-set sampling and splitting, hyperparameter selection, supplementary family-level scatter plots, and the comparison between shared and separately trained UHNs.

D.1 Model Sampling Details

For each multi-model experiment, we construct a finite model set M by sampling architectures from the corresponding family under simple constraints (e.g., divisibility for GroupNorm (Wu & He, 2018)). In total, we construct four model sets: three CNN model sets for Exp-1–3 and one Transformer (Vaswani et al., 2017) model set for Exp-4.

D.1.1 CNN family (Exp-1–3; CIFAR-10)

We use a 4-stage CNN with stages indexed by $s \in \{0, 1, 2, 3\}$. Let K_s denote the number of convolutional layers in stage s and c_s the stage width (channels); under shortcuts (He et al., 2016) (our default), c_s is the (constant) number of output channels for all convolution layers within stage s. Throughout this subsection, $\text{Unif}\{a, \dots, b\}$ denotes the discrete uniform distribution over integers $a, a+1, \dots, b$. We consider the following experiment-specific sampling schemes:

- **Mixed depth (Exp-1).** Sample $K_1, K_2, K_3 \sim \text{Unif}\{6, \dots, 10\}$ independently, with $K_0 = 1$, and fix $(c_0, c_1, c_2, c_3) = (16, 16, 32, 64)$. We sample $|M| = 100$ models.
- **Mixed width (Exp-2).** Fix $(K_0, K_1, K_2, K_3) = (1, 6, 6, 6)$ and sample $c_0 \sim \text{Unif}\{16, \dots, 32\}$, then set $c_1 = c_0$, set $c_2 \sim \text{Unif}\{32, \dots, 64\}$, and set $c_3 \sim \text{Unif}\{64, \dots, 128\}$. We sample $|M| = 500$ models.
- **Mixed depth $\times$ width (Exp-3).** Sample $K_1, K_2, K_3 \sim \text{Unif}\{6, \dots, 8\}$ and sample $c_0 \sim \text{Unif}\{16, \dots, 32\}$, then set $c_1 = c_0$, with $c_2 \sim \text{Unif}\{32, \dots, 64\}$ and $c_3 \sim \text{Unif}\{64, \dots, 128\}$. We sample $|M| = 1000$ models.

Implementation details. Unless otherwise stated, all CNNs use 3×3 convolutions and enable bias in both convolution and linear layers. We use LeakyReLU with slope 0.1 and GroupNorm with $g = 4$ groups for all convolution layers except the first convolution layer. We use no dropout (Srivastava et al., 2014) in either convolution or linear layers. Since GroupNorm with $g = 4$ groups is used, we require $c_s \equiv 0 \pmod g$. Residual shortcuts are enabled for all eligible layers except the output linear layer and the first convolution layer of stages $s \in \{0, 2, 3\}$ (so the first layer of stage 1 uses a shortcut and we enforce $c_0 = c_1$). We only apply stage-wise downsampling at the first layer of stages

Table 18: Training and generation time for Direct, Transformer-HN, and UHN on representative CNN variants. Train reports total training time in hours, and Gen reports per-model generation time in milliseconds.

Model	Base model params	Direct Train	Transformer Train	UHN Train	Transformer Gen	UHN Gen
CNN-20	269,034	0.73 h	1.35 h	1.30 h	0.159 ms	11.154 ms
CNN-20-2	1,072,586	0.68 h	1.35 h	3.57 h	0.152 ms	37.995 ms
CNN-20-4	4,283,274	0.68 h	1.45 h	19.32 h†	0.176 ms	145.475 ms†
CNN-20-6	9,632,074	1.12 h	2.55 h	42.61 h†	0.155 ms	324.865 ms†

† For CNN-20-4 and CNN-20-6, UHN training and generation use sequential batched weight generation due to the memory limit of an RTX 4090; training additionally uses gradient checkpointing.

Table 19: Model-family statistics and model-set splits for the multi-model experiments. Avg./Max target #Params are the average/maximum numbers of trainable parameters among target models in M.

	Model-family statistics			Model-set split			
Model family	$\lvert M\rvert$	Avg. target #Params	Max target #Params	$\lvert M_{\text{train}}\rvert$	$\lvert M_{\text{test}}\rvert$	$\lvert M_{\text{val}}\rvert$	$\lvert M'_{\text{train}}\rvert$
CNN Mixed Depth	100	366,995	463,018	80	20	20	20
CNN Mixed Width	500	624,987	1,072,586	450	50	50	50
CNN Mixed Depth $\times$ Width	1000	734,206	1,372,110	950	50	50	50
Transformer Mixed	1000	523,650	1,086,212	950	50	50	50

2 and 3, implemented via convolutional pooling (stride 2). The classifier head consists of adaptive average pooling followed by a single linear output layer (no activation, no normalization).

D.1.2 Transformer family (Exp-4; AG News)

In Exp-4, we sample a Transformer encoder stack by drawing the number of encoders $J \sim \mathrm{Unif}\{1,\dots,4\}$ and the number of heads $h \sim \mathrm{Unif}\{1,\dots,8\}$. We sample the embedding dimension $d \sim \mathrm{Unif}\{32,\dots,128\}$ subject to $d \equiv 0 \pmod h$, and for each encoder block $j \in \{1,\dots,J\}$ we sample the feed-forward depth $K_{\mathrm{j}} \sim \mathrm{Unif}\{1,\dots,3\}$. We sample $|M| = 1000$ models.

Implementation details. We use vocabulary size $V = 5000$ and maximum sequence length $T_{\max} = 128$. All MHA layers and all linear layers use bias. We apply dropout with rate 0.2 in the embedding layer and in all MHA and feed-forward layers. Residual shortcuts are enabled for all MHA and feed-forward layers. The final output linear layer uses neither dropout nor a shortcut. Except for the first MHA layer and the final output linear layer, we apply LeakyReLU with slope 0.1 and LayerNorm (Ba et al., 2016); otherwise we use no activation and no normalization. The final classifier head extracts the first token representation and applies a linear output layer. Because shortcuts are used, the output dimension of all MHA and intermediate linear layers equals the embedding dimension d.

D.2 Model-set construction and splitting

For each multi-model experiment, we sample a finite model set M once from its model family detailed in Appendix D.1. M is further split into 2 subsets M_{train} and M_{test} for training and held-out evaluation. During hyperparameter selection, we reserve a validation subset $M_{\text{val}} \subset M_{\text{train}}$.

We also measure the hold-in performance. However, since evaluating every model in M_{train} is time consuming, we only use a hold-in subset $M'_{\text{train}} \subset M_{\text{train}}$ for evaluation, where $M'_{\text{train}} \cap M_{\text{val}} = \emptyset$.

Table 19 summarizes the model-family sizes, target-model parameter scales, and split sizes.

D.3 Hyperparameter selection

For each experiment, we sweep (i) the initialization-stage learning rate and steps $(\eta_{\text{init}}, S_{\text{init}})$ and (ii) the training-stage learning rate and epochs $(\eta_{\text{train}}, E_{\text{train}})$, following the sweep protocol described in Appendix B.8. Table 20 lists the sweep grids, and Table 21 lists the selected values.

Table 20: Hyperparameter sweep grids for the multi-model experiments. Table entries list the sweep-grid values for η_{init}, S_{init}, η_{train}, and E_{train}.

Model family	E_0	η_{init}	S_{init}	η_{train}	E_{train}
CNN Mixed Depth	200	$\{5e{-}5, 1e{-}4, 2e{-}4\}$	$\{3200, 6400, 12800\}$	$\{2e{-}5, 5e{-}5, 1e{-}4\}$	$\{800, 1600, 3200\}$
CNN Mixed Width	200	$\{5e{-}5, 1e{-}4, 2e{-}4\}$	$\{3200, 6400, 12800\}$	$\{2e{-}5, 5e{-}5, 1e{-}4\}$	$\{800, 1600, 3200\}$
CNN Mixed Depth × Width	200	$\{5e{-}5, 1e{-}4, 2e{-}4\}$	$\{3200, 6400, 12800\}$	$\{2e{-}5, 5e{-}5, 1e{-}4\}$	$\{800, 1600, 3200\}$
Transformer Mixed	50	$\{5e{-}5, 1e{-}4, 2e{-}4\}$	$\{2000, 4000, 8000\}$	$\{2e{-}5, 5e{-}5, 1e{-}4\}$	$\{50, 100, 200\}$

Table 21: Best hyperparameters selected by grid search for the multi-model experiments.

Model family	η_{init}	S_{init}	η_{train}	E_{train}
CNN Mixed Depth	$2e{-}4$	12800	$5e{-}5$	3200
CNN Mixed Width	$1e{-}4$	12800	$1e{-}4$	3200
CNN Mixed Depth × Width	$2e{-}4$	3200	$1e{-}4$	3200
Transformer Mixed	$2e{-}4$	8000	$5e{-}5$	100

D.4 Additional Multi-Model Results

The aggregate results in the main paper are complemented here by per-architecture diagnostics for the remaining model families and by a direct comparison between shared and separately trained UHNs.

Shared versus separately trained UHNs. To quantify the performance cost of sharing generator parameters across architectures, we compare six representative architectures from the seen evaluation subset M'_{train} and six representative architectures from the unseen evaluation set M_{test} of CNN Mixed Depth × Width under two settings. In the shared setting, we reuse the shared-UHN runs trained on M_{train} in the main multi-model experiment (Section 4.4) and evaluate the models they generate for these 12 architectures. In the separate setting, we train one dedicated UHN for each selected architecture. These separate UHNs omit the task-structure encoder and reuse the UHN hyperparameters selected for CNN-20 in the cross-family experiment (Table 12). Detailed architecture specifications are provided in Appendix D.5.

The separately trained UHNs attain higher average accuracy on both representative subsets, indicating that parameter sharing across architectures incurs an average accuracy gap of roughly one percentage point in this comparison.

D.5 Architectures used for separate-UHN comparison in the multi-model setting

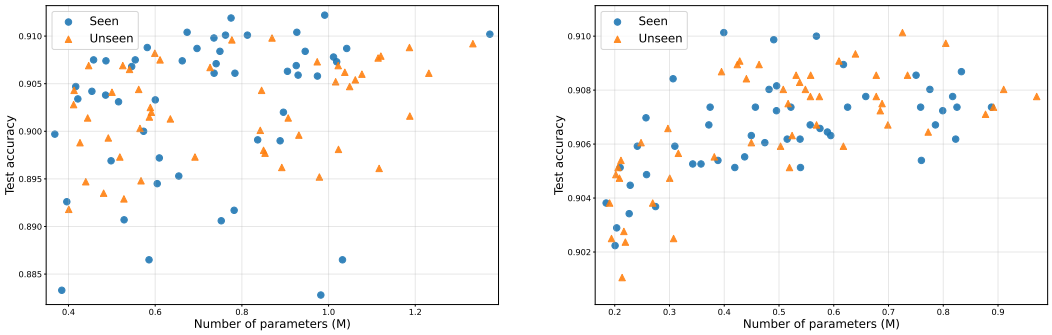
To compare shared and separately trained UHNs in the multi-model setting, we select a representative subset of target architectures spanning different depths, widths, and parameter scales in the CNN Mixed Depth × Width setting. The subset includes six representative architectures from the seen evaluation subset M'_{train} (Table 23) and six representative architectures from the unseen evaluation set M_{test} (Table 24). Where each architecture is specified by: (1) total network depth, (2) channel widths of the four stages, and (3) the number of layers in each stage.

E Multi-Task Experiment

This section provides additional training, task-sampling, and hyperparameter-selection details for the multi-task experiment in Section 4.5.

E.1 Training and task sampling

Following the task-sampling procedure in Section 3.2.1, at each training step we sample a task $t \sim P_T$ and train on that task. The sampling distribution assigns probabilities CIFAR-10 (0.55), AG News (0.18), kv (0.11), MNIST (0.08), Cora (0.04), and PubMed (0.04).



(a) CNN Mixed Depth × Width

(b) Transformer Mixed

Figure 5: Additional seen–unseen target-model performance plots. Each point represents one target architecture and shows its parameter count and test accuracy. Seen and unseen models exhibit similar performance across both CNN Mixed Depth×Width and Transformer Mixed.

Table 22: Shared-versus-separate UHN comparison on six representative architectures from M'_{train} and six representative architectures from M_{test} in CNN Mixed Depth × Width. For each random seed, test accuracy is first averaged separately over the six architectures in each subset; we report the mean ± standard deviation of these averages over 2 seeds for Separate UHN and 3 seeds for Shared UHN.

Method	Seen Avg. Acc.	Unseen Avg. Acc.
Separate UHN	0.9172 ± 0.0002	0.9153 ± 0.0022
Shared UHN	0.9052 ± 0.0015	0.9045 ± 0.0013

E.2 Hyperparameter selection

Our goal is to select a single multi-task hyperparameter configuration that works well across all tasks. To avoid solutions that perform well on average but fail on an individual task, we enforce task-specific validation guardrails (Table 25) and discard any configuration that violates *any* task's guardrail. For metrics where higher is better (e.g., classification accuracy), we require the validation metric to be at least the guardrail; for metrics where lower is better (e.g., RMSE), we require it to be at most the guardrail.

We sweep (i) the initialization-stage learning rate and steps $(\eta_{\text{init}}, S_{\text{init}})$ and (ii) the training-stage learning rate and steps $(\eta_{\text{train}}, S_{\text{train}})$, following the sweep protocol described in Appendix B.8. Table 26 summarizes the sweep grid and the selected hyperparameters.

F Recursive Experiments

This section provides implementation details for the recursive experiments in Section 4.6, including (i) the architecture of generated UHNs H_i $(i \geq 1)$, (ii) additional training details, and (iii) hyperparameter selection.

F.1 Generated UHN architecture

Each generated hypernetwork H_i $(i \geq 1)$ follows the architectural template in Appendix B.5.3 with the following settings.

Two-branch parameter generator Each generated UHN consists of two branches whose outputs are summed before the final readout.

Index branch The index branch takes Gaussian Fourier features (Tancik et al., 2020) of the normalized index descriptor with $F_{\mathbf{v}} = 1024$ frequencies. It applies an input linear layer followed by

Table 23: Representative seen architectures used for the separate-UHN comparison.

ID	Total Depth	Stage Widths	Stage Layers
S1	22	32 / 32 / 44 / 64	1 / 6 / 7 / 7
S2	25	20 / 20 / 48 / 84	1 / 7 / 8 / 8
S3	25	28 / 28 / 60 / 80	1 / 7 / 8 / 8
S4	21	28 / 28 / 64 / 116	1 / 7 / 6 / 6
S5	23	32 / 32 / 60 / 128	1 / 6 / 7 / 8
S6	26	32 / 32 / 32 / 116	1 / 8 / 8 / 8

Table 24: Representative unseen architectures used for the separate-UHN comparison.

ID	Total Depth	Stage Widths	Stage Layers
U1	25	32 / 32 / 48 / 64	1 / 8 / 7 / 8
U2	22	16 / 16 / 40 / 104	1 / 7 / 7 / 6
U3	24	24 / 24 / 36 / 112	1 / 7 / 8 / 7
U4	23	28 / 28 / 44 / 128	1 / 7 / 8 / 6
U5	26	20 / 20 / 44 / 120	1 / 8 / 8 / 8
U6	25	20 / 20 / 56 / 128	1 / 7 / 8 / 8

4 linear layers with hidden dimension $d = 64$, each with LeakyReLU (slope 0.1), LayerNorm (Ba et al., 2016), and a shortcut connection (He et al., 2016).

Task-Structure branch The task-structure branch takes Gaussian Fourier features of the normalized task-structure descriptors with $F_{\mathbf{u}} = 32$ frequencies. It applies a multi-head attention (Vaswani et al., 2017) layer with $h = 4$ heads (with a shortcut connection), followed by two linear layers with LeakyReLU, LayerNorm, and shortcut connections. The resulting sequence is average-pooled across the sequence dimension, followed by a linear layer with output size 64 and then a second linear layer with LeakyReLU.

Readout We sum the index branch and task-structure branch features and apply a final linear layer with LeakyReLU to predict each target scalar weight.

We share descriptor normalization statistics and Fourier matrices across recursion levels (i.e., all H_i use the same ones as the root H_0).

F.2 Additional training details

Since deeper recursion increases training instability, we apply gradient clipping (Pascanu et al., 2013) with maximum norm 0.01 in *both* the initialization and training phases for all recursion depths (including $K = 1$).

Level-wise initialization schedule. For a recursive chain $H_0 \to \cdots \to H_k \to \cdots \to H_K \to f$, we initialize the generated models in a top-down order: first H_1, then H_2, and so on, until finally initializing the leaf model f. We allocate S_{lvl} initialization steps to each stage of this sequence. At initialization iteration step, the stage currently being optimized is indexed by

$$k = \min\left(K, \lfloor \mathrm{step}/S_{\mathrm{lvl}} \rfloor\right). \tag{34}$$

Here, $\mathrm{step} = 0, 1, 2, \ldots$ counts initialization iterations from the start of the recursive initialization procedure. When $k < K$, the current stage initializes the parameters of H_{k+1} generated by H_k; when $k = K$, it initializes the leaf model f generated by H_K. We reset the optimizer and learning-rate scheduler whenever the procedure advances to the next stage.

F.3 Hyperparameter selection

We sweep (i) the initialization-stage learning rate and steps $(\eta_{\mathrm{init}}, S_{\mathrm{init}})$ and (ii) the training-stage learning rate and steps $(\eta_{\mathrm{train}}, S_{\mathrm{train}})$ for recursion depth $K = 1$, following the sweep protocol

Table 25: Task-specific validation guardrails used during hyperparameter selection in the multi-task setting.

Task	Dataset	Model	Guardrail (val)
Image	MNIST	MLP	0.95
Image	CIFAR-10	CNN-44	0.85
Graph	Cora	GCN	0.75
Graph	PubMed	GAT	0.75
Text	AG News	Transformer-2L	0.85
Formula	kv	KAN-g5	5e−2

Table 26: Hyperparameter sweep grid and selected hyperparameters for multi-task training (subject to the guardrails in Table 25).

	η_{init}	S_{init}	η_{train}	S_{train}
Grid ($S_0 = 40000$)	$\{5\text{e}{-5}, 1\text{e}{-4}, 2\text{e}{-4}\}$	$\{500, 1000, 2000\}$	$\{1\text{e}{-5}, 2\text{e}{-5}, 5\text{e}{-5}\}$	$\{100000, 200000, 300000\}$
Selected	$1\text{e}{-4}$	500	$2\text{e}{-5}$	200000

described in Appendix B.8. We set the number of initialization steps allocated to each recursion level as $S_{\text{lvl}} = S_{\text{init}}/2$, and reuse the same S_{lvl} for all recursion depths. Table 27 summarizes the sweep grid and the selected best configuration.

Let $S_{\text{init}}^{(K=1)}$, $S_{\text{train}}^{(K=1)}$, $\eta_{\text{init}}^{(K=1)}$, and $\eta_{\text{train}}^{(K=1)}$ denote the selected step budgets and learning rates at $K = 1$. For depths $K > 1$, we set the total initialization steps to $S_{\text{init}} = (K+1)\, S_{\text{init}}^{(K=1)}/2$, reuse $S_{\text{train}} = S_{\text{train}}^{(K=1)}$, and reduce learning rates to maintain stability. During initialization, we use a per-level learning rate schedule

$$\eta_{\text{init}}^{(k)} = \begin{cases} \eta_{\text{init}}^{(K=1)}/5, & k \in \{0, 1\}, \\ \eta_{\text{init}}^{(K=1)}/10, & k = 2, \\ \eta_{\text{init}}^{(K=1)}/40, & k = 3, \end{cases} \tag{35}$$

where $k \in \{0, \ldots, K\}$ indexes the recursion level, and $\eta_{\text{init}}^{(k)}$ denotes learning rate for level-k initialization. We use 1000 warmup steps at each initialization level. During training, we use a reduced training-stage learning rate for deeper recursion: $\eta_{\text{train}}^{(K=1)}/2$ for $K = 2$ and $\eta_{\text{train}}^{(K=1)}/8$ for $K = 3$.

G Ablation Experiment

This appendix collects the ablation experiments referenced from the main paper: index encoding, UHN capacity, task-structure conditioning, and initialization. These analyses test representation and optimization choices separately from the main cross-family, scaling, shared-generation, and recursive experiments.

G.1 Index Encoding

G.1.1 Setup

UHN generates scalar parameters by conditioning on index descriptors. Since MLPs exhibit a spectral bias towards low-frequency functions (Tancik et al., 2020), the choice of index encoding can be critical when the target parameter field varies rapidly across indices. Note that the index encoding is applied to the normalized index descriptor $\hat{\mathbf{v}}_i \in \mathbb{R}^{10}$ (computed as in Appendix B.4). We test the effect of different index encoding schemes by applying an encoding $\gamma(\cdot)$ to $\hat{\mathbf{v}}_i$:

1. **Raw encoding (no mapping).** $\gamma_{\text{raw}}(\hat{\mathbf{v}}_i) = \hat{\mathbf{v}}_i$.
2. **Positional encoding (Mildenhall et al., 2021).**

$$\gamma_{\text{pos}}(\hat{\mathbf{v}}_i) = \left[\cos(\omega^0 \hat{\mathbf{v}}_i)^\top,\ \sin(\omega^0 \hat{\mathbf{v}}_i)^\top,\ \ldots,\ \cos(\omega^{n-1} \hat{\mathbf{v}}_i)^\top,\ \sin(\omega^{n-1} \hat{\mathbf{v}}_i)^\top\right]^\top,$$

Table 27: Hyperparameter sweep grid and selected hyperparameters for the recursive setting with recursion depth $K = 1$.

	η_{init}	S_{init}	η_{train}	S_{train}
Grid ($S_0 = 30000$)	$\{5\text{e}{-5}, 1\text{e}{-4}, 2\text{e}{-4}\}$	$\{1000, 2000, 4000\}$	$\{2\text{e}{-5}, 5\text{e}{-5}, 1\text{e}{-4}\}$	$\{15000, 30000, 60000\}$
Selected	$1\text{e}{-4}$	4000	$2\text{e}{-5}$	30000

Table 28: Index-encoding ablation results. #Params denotes UHN parameter count; Acc. denotes test accuracy.

Encoding type	#Params	Acc.
Raw	68,629	0.6642 ± 0.0199
Positional	149,301	0.8677 ± 0.0034
GFF	612,117	0.8993 ± 0.0016

where $\omega^j = \sigma^{j/n}$ for $j \in \{0, \dots, n-1\}$; we set the number of frequencies $n = 32$, and scale $\sigma = 100$.

3. **Gaussian Fourier features (GFF) (Tancik et al., 2020).** $\gamma_{\mathbf{B_v}}(\hat{\mathbf{v}}_i) = \left[\cos(\mathbf{B_v}\hat{\mathbf{v}}_i)^\top, \ \sin(\mathbf{B_v}\hat{\mathbf{v}}_i)^\top\right]^\top$, where $\mathbf{B_v} \in \mathbb{R}^{F_{\mathbf{v}} \times 10}$ with every entry sampled from $\mathcal{N}(0, \sigma^2)$; we set the number of Fourier-feature frequencies $F_{\mathbf{v}} = 2048$ and scale $\sigma = 100$.

We run the ablation using the single-model experiment setting (Section 4.2) on CIFAR-10 where the base model is CNN-20, and only change the encoding scheme.

Additional training details. Gaussian Fourier features are the default index encoding scheme in our single-model experiment; its result in Table 28 therefore corresponds to the single-model experiment setting. For Raw and Positional encodings, we re-sweep the training learning rate η_{train} over $\{5\text{e}{-5}, 1\text{e}{-4}, 2\text{e}{-4}\}$ and select $2\text{e}{-4}$. Since Raw and Positional encodings are less stable in our experiments, we apply gradient clipping (Pascanu et al., 2013) with maximum norm 1.0 during the training phase for these two settings (no clipping during the initialization phase).

G.1.2 Results.

Table 28 reports test accuracy (mean $\pm$ std over $n = 3$ seeds) and the UHN parameter count.

G.1.3 Analysis

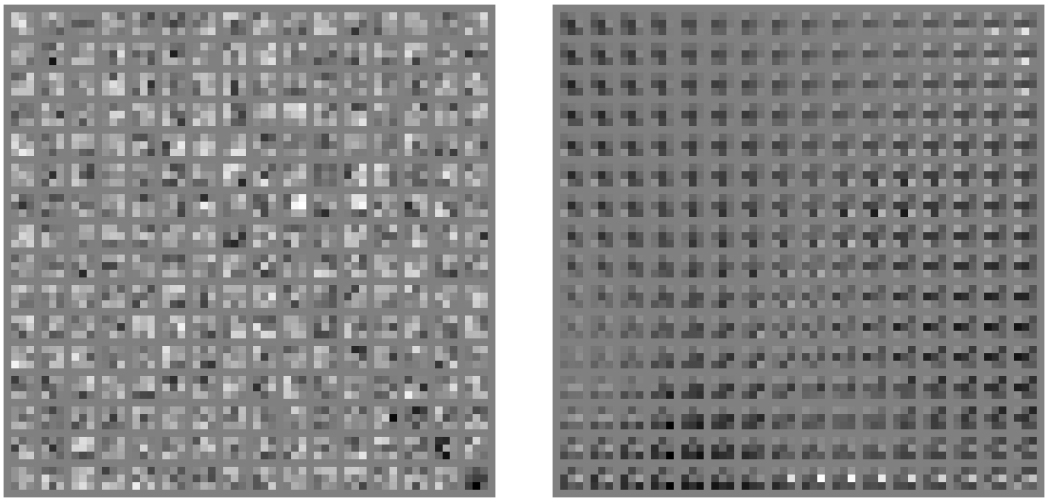
Gaussian Fourier features substantially outperform both Raw and Positional encodings in this experiment. Because the three encoding configurations also differ in UHN parameter count, the comparison alone does not fully separate the effect of encoding from generator capacity. However, the capacity ablation in Table 29 provides complementary evidence: a smaller GFF configuration with $F_{\mathbf{v}} = 256$ uses $135{,}445$ parameters yet achieves 0.8894 accuracy, compared with 0.8677 for the larger $149{,}301$-parameter Positional model, supporting a genuine benefit from the encoding itself rather than generator size alone.

We also visualize the kernels of the third convolution layer of the generated base model (Figure 6). Raw encoding produces lower-variance kernels, whereas GFF produces more structured and diverse kernels. This is consistent with the intuition that Gaussian Fourier features expose higher-frequency basis functions (Tancik et al., 2020) to the hypernetwork, mitigating spectral bias and enabling it to model rapidly varying parameter fields over index space.

G.2 UHN Capacity

G.2.1 Setup

We study how UHN performance scales with generator capacity, testing whether accuracy improves predictably as we increase (i) the richness of the index encoding, (ii) the width of the generator,



(a) Gaussian Fourier features (b) Raw encoding

Figure 6: Generated third-layer convolution kernels under different index encodings. Gaussian Fourier features (GFF) produce more structured and diverse kernels, whereas raw index encoding produces lower-variance kernels.

Table 29: Capacity ablation results. We vary one hyperparameter at a time: the number of Fourier-feature frequencies $F_{\mathbf{v}}$ for index encoding (left), the UHN hidden dimension d (middle), and the number of UHN residual blocks N_{blk} (right). Underlined entries indicate the single-model experiment setting (Section 4.2). #Params denotes UHN parameter count; Acc. denotes test accuracy.

$F_{\mathbf{v}}$			d			N_{blk}		
Value	#Params	Acc.	Value	#Params	Acc.	Value	#Params	Acc.
256	135,445	0.8894 ± 0.0020	32	156,117	0.8900 ± 0.0037	0	545,045	0.8901 ± 0.0028
512	203,541	0.8955 ± 0.0005	64	299,925	0.8960 ± 0.0015	1	578,581	0.8995 ± 0.0008
1024	339,733	0.8964 ± 0.0012	<u>128</u>	<u>612,117</u>	<u>0.8993 ± 0.0016</u>	<u>2</u>	<u>612,117</u>	<u>0.8993 ± 0.0016</u>
<u>2048</u>	<u>612,117</u>	<u>0.8993 ± 0.0016</u>	256	1,334,805	0.9019 ± 0.0041	3	645,653	0.9009 ± 0.0010
4096	1,156,885	0.9018 ± 0.0012	–	–	–	–	–	–

or (iii) its depth. We run the UHN capacity ablation in the single-model setting (Section 4.2) on CIFAR-10 where the base model is CNN-20. Starting from the default configuration $(F_{\mathbf{v}}, d, N_{\text{blk}}) = (2048, 128, 2)$, we vary one of three capacity-related hyperparameters while keeping the other two fixed: the number of Fourier-feature (Tancik et al., 2020) frequencies $F_{\mathbf{v}}$ for index encoding, the UHN hidden dimension d, and the number of UHN residual blocks N_{blk}.

G.2.2 Results

For each configuration, we report test accuracy (mean $\pm$ std over $n = 3$ seeds) and the UHN parameter count. Table 29 summarizes the results.

G.2.3 Analysis

Increasing the number of Fourier-feature frequencies $F_{\mathbf{v}}$ and the UHN hidden dimension d consistently improves accuracy over the evaluated configurations. These trends support the view that richer high-frequency index features and higher-dimensional hidden representations are important for accurate weight generation. By contrast, increasing the number of residual blocks N_{blk} produces its largest improvement from 0 to 1 block, with only small differences beyond one block, suggesting that additional depth provides limited benefit here.

Table 30: Task-structure-encoder (TSE) ablation in the multi-model setting. "w/" and "w/o" indicate whether the task-structure encoder is enabled or disabled. #Params denotes UHN parameter count. "Seen Acc." averages test accuracy over M'_{train}, and "Unseen Acc." averages over M_{test}.

Model family	TSE	#Params	Seen Acc.	Unseen Acc.
CNN Mixed Width	w/	663,151	0.9145 ± 0.0014	0.9145 ± 0.0013
CNN Mixed Width	w/o	612,117	0.9143 ± 0.0012	0.9141 ± 0.0013

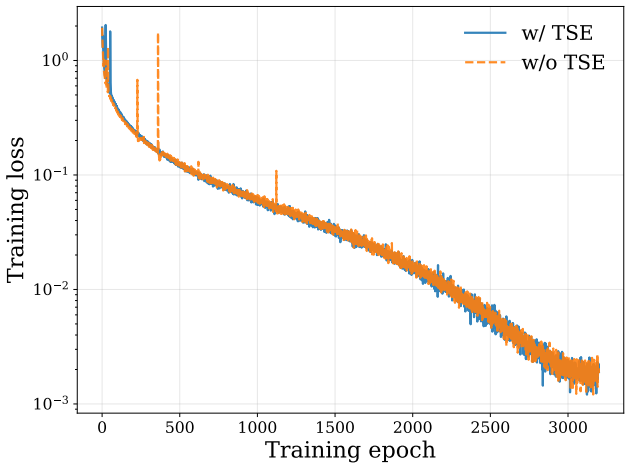


Figure 7: TSE ablation in the multi-model CNN Mixed Width setting, comparing training with and without the task-structure encoder ("w/ TSE" and "w/o TSE", respectively). Removing TSE introduces repeated transient loss spikes, whereas using TSE produces a smoother optimization trajectory. Training loss is averaged over all samples within each epoch and shown on a logarithmic y-axis.

The default configuration $(F_{\mathbf{v}}, d, N_{\text{blk}}) = (2048, 128, 2)$ provides a good accuracy–parameter balance among the evaluated configurations. Overall, the ablation shows that UHN performance can be improved through standard capacity choices while retaining the same generation mechanism.

G.3 Task-Structure Encoder

G.3.1 Setup

UHN can condition on explicit descriptors of the target architecture and task via a *task-structure encoder*. To assess the contribution of this conditioning component, we disable the task-structure encoder while keeping the remaining components and training protocol unchanged, and evaluate in **multi-model training** on CNN Mixed Width following Section 4.4. We additionally evaluate the task-structure encoder in (i) multi-model training on Transformer Mixed following Section 4.4 and (ii) multi-task training across heterogeneous tasks following Section 4.5. These additional settings complement the CNN Mixed Width ablation by testing the same conditioning component under a different architecture family and under heterogeneous task sharing.

G.3.2 Multi-Model Results

For multi-model training, we report test accuracy averaged over architectures in the seen evaluation subset M'_{train} and the unseen evaluation set M_{test}, which is held out from training. Tables 30 and 31 report the mean and standard deviation of these averages over 3 random seeds.

G.3.3 Multi-Task Results

For multi-task training, we use UHN to generate each task's base model and evaluate test performance on that task. Table 32 reports the mean and standard deviation over 3 random seeds. For classification

Table 31: Task-structure-encoder (TSE) ablation in the multi-model setting. "w/" and "w/o" indicate whether the task-structure encoder is enabled or disabled. #Params denotes UHN parameter count. "Seen Acc." averages test accuracy over M'_{train}, and "Unseen Acc." averages over M_{test}.

Model family	TSE	#Params	Seen Acc.	Unseen Acc.
Transformer Mixed	w/	663,151	0.9063 ± 0.0004	0.9066 ± 0.0002
Transformer Mixed	w/o	612,117	0.9073 ± 0.0021	0.9074 ± 0.0023

Table 32: Task-structure-encoder (TSE) ablation in the multi-task setting. "w/ TSE" and "w/o TSE" indicate whether the task-structure encoder is enabled or disabled. "Perf." denotes test accuracy for classification (higher is better) and test RMSE for formula regression (lower is better).

Task	Dataset	Model	UHN perf. (multi-task, w/ TSE)	UHN perf. (multi-task, w/o TSE)
Image	MNIST	MLP	0.9786 ± 0.0012	0.9783 ± 0.0011
Image	CIFAR-10	CNN-44	0.8927 ± 0.0011	0.8895 ± 0.0015
Graph	Cora	GCN	0.7930 ± 0.0053	0.7927 ± 0.0021
Graph	PubMed	GAT	0.7697 ± 0.0031	0.7703 ± 0.0042
Text	AG News	Transformer-2L	0.9062 ± 0.0005	0.9057 ± 0.0002
Formula (RMSE)	kv	KAN-g5	0.0172 ± 0.0112	0.0082 ± 0.0018

tasks, we report *test accuracy* (higher is better); for formula regression, we report *test RMSE* (lower is better).

G.3.4 Analysis

The task-structure encoder has a modest and mixed effect on final performance. For CNN Mixed Width, seen and unseen accuracy changes only slightly when the encoder is removed, while the training-loss curve in Figure 7 shows larger fluctuations without it. For Transformer Mixed, removing the encoder slightly improves both seen and unseen accuracy.

The multi-task results show a similarly mixed pattern. The clearest classification improvement from the encoder occurs on CIFAR-10, while MNIST, Cora, and AG News change only slightly; PubMed accuracy is marginally lower with the encoder. For formula regression, enabling the encoder increases RMSE from 0.0082 to 0.0172.

Taken together, these results suggest that the task-structure encoder is best viewed as an auxiliary conditioning component rather than the primary source of UHN's shared-generation capability. In most of these comparisons, final performance changes relatively little without the encoder, suggesting that the index-based generator already provides most of the information needed for parameter generation across different models and tasks. The task-structure encoder provides additional context about which architecture and task a queried parameter belongs to, helping disambiguate parameters that may occupy similar index positions across different targets. This additional conditioning can make optimization smoother in some cases, as suggested by the CNN Mixed Width training curve, while its effect on final performance depends on the model family and task.

G.4 Initialization

G.4.1 Setup

Mismatches between the initial weight distribution of the base model weights generated by UHN and standard neural network initializations (e.g., Glorot initialization (Glorot & Bengio, 2010)) can destabilize optimization. We therefore include an *initialization stage* that calibrates the generated weights to a target distribution before the main training begins (by default, we match PyTorch's standard parameter initialization (Paszke et al., 2019) for the corresponding layer unless otherwise specified).

We evaluate the effect of this stage by disabling it (setting the initialization-stage learning rate and steps $(\eta_{\text{init}}, S_{\text{init}})$ to 0) while keeping all other settings unchanged. We report results in three settings: (i) **multi-model training** on two representative model families (CNN Mixed Width and

Table 33: Initialization (init) ablation in the multi-model setting. "w/" and "w/o" indicate whether the initialization is enabled or disabled. "Seen Acc." averages test accuracy over M'_{train}, and "Unseen Acc." averages over M_{test}.

Model family	init	Seen Acc.	Unseen Acc.
CNN Mixed Width	w/	0.9145 ± 0.0014	0.9145 ± 0.0013
CNN Mixed Width	w/o	0.9130 ± 0.0011	0.9130 ± 0.0011
Transformer Mixed	w/	0.9063 ± 0.0004	0.9066 ± 0.0002
Transformer Mixed	w/o	0.8966 ± 0.0095	0.8963 ± 0.0096

Table 34: Initialization (init) ablation in the multi-task setting. "w/ init" and "w/o init" indicate whether the initialization is enabled or disabled. "Perf." denotes test accuracy for classification (higher is better) and test RMSE for formula regression (lower is better).

Task	Dataset	Model	UHN perf. (multi-task, w/ init)	UHN perf. (multi-task, w/o init)
Image	MNIST	MLP	0.9786 ± 0.0012	0.9780 ± 0.0016
Image	CIFAR-10	CNN-44	0.8927 ± 0.0011	0.8875 ± 0.0023
Graph	Cora	GCN	0.7930 ± 0.0053	0.7750 ± 0.0040
Graph	PubMed	GAT	0.7697 ± 0.0031	0.7690 ± 0.0017
Text	AG News	Transformer-2L	0.9062 ± 0.0005	0.9051 ± 0.0025
Formula (RMSE)	kv	KAN-g5	0.0172 ± 0.0112	0.0196 ± 0.0085

Transformer Mixed) following Section 4.4; (ii) **multi-task training** across heterogeneous tasks following Section 4.5; and (iii) **recursive training** with recursion depth $K = 1$ following Section 4.6.

G.4.2 Results

For multi-model training, we report test accuracy averaged over architectures in the seen evaluation subset M'_{train} and the unseen evaluation set M_{test}, which is held out from training. Table 33 reports the mean and standard deviation of these averages over 3 random seeds.

For multi-task training, we use UHN to generate each task's base model and evaluate test performance on that task. Table 34 reports the mean and standard deviation over 3 random seeds. For classification tasks, we report *test accuracy* (higher is better); for formula regression, we report *test RMSE* (lower is better).

For recursive training, we observe that disabling initialization causes training to diverge.

G.4.3 Analysis

The initialization stage affects both optimization dynamics and final performance, with the magnitude depending on the setting. In multi-model training, initialization improves final performance on both CNN Mixed Width and Transformer Mixed. The gain is small for CNN Mixed Width, whereas Transformer Mixed shows a larger improvement in final accuracy and substantially lower early training loss with initialization (Figure 8). In multi-task training, initialization improves test performance for all six task–model pairs, although the differences vary across tasks. In the recursive setting, removing the initialization stage causes training to diverge at $K = 1$, showing that recursive generation is particularly sensitive to the initial parameter statistics of the generated UHN. Overall, initialization is essential for stable recursive generation and generally benefits optimization and final performance in the multi-model and multi-task settings, although the magnitude of the improvement varies across architectures and tasks.

H Extended Discussion

This section provides additional empirical insights, limitations, and future directions.

UHN representation and capacity. UHN benefits from expressive index encodings and sufficient generator capacity. Fourier features are particularly effective for representing higher-frequency variation in the parameter field, supporting more accurate weight generation.

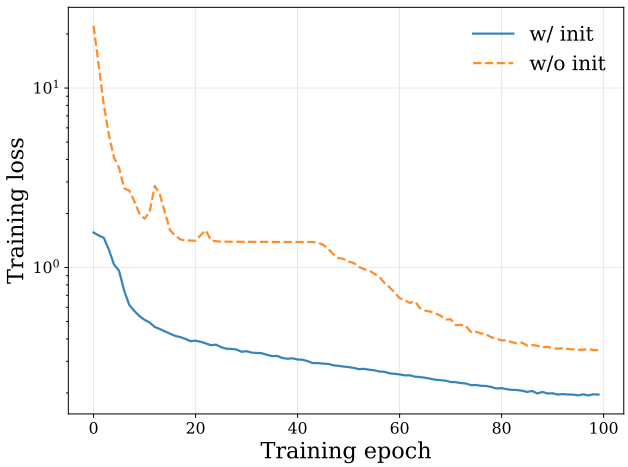


Figure 8: Initialization ablation in the multi-model Transformer Mixed setting, comparing training with and without the initialization stage ("w/ init" and "w/o init", respectively). Initialization leads to faster convergence and consistently lower training loss over the course of training. Training loss is averaged over all samples within each epoch and shown on a logarithmic y-axis.

Task-structure conditioning and generated-weight initialization. The task-structure encoder provides auxiliary architecture and task context rather than serving as the primary source of shared-generation capability, and its effect varies across models and tasks. In contrast, initialization of the generated weights with appropriate parameter statistics plays a more consistent role in optimization and is essential for recursive generation.

H.1 Limitations

Performance relative to direct training. UHN does not consistently match directly trained models across all tasks and architectures, and the performance gap varies across settings.

Scope of generalization. The current experiments establish within-family generalization to unseen architectures and joint generation across heterogeneous task–model pairs, but do not evaluate zero-shot generation for arbitrary architectures or tasks outside the supported descriptor schema.

Non-trainable model state. UHN currently generates trainable parameters but not auxiliary non-trainable states such as BatchNorm (Ioffe & Szegedy, 2015) running statistics, so architectures that depend on such state require additional handling.

Computational cost. Although the generator parameter count remains fixed as target size grows, training and generation costs increase with the number of generated parameters and can substantially exceed those of direct training and the chunk-based hypernetwork baselines.

H.2 Future Work

Scaling to larger targets and deeper recursion. Extending UHN to substantially larger target models and longer recursive generation chains would further test the scalability of the descriptor-conditioned formulation.

Improving shared generation. Future work could improve parameter sharing across heterogeneous models and tasks through better optimization, task balancing, and conditioning, particularly for extreme unseen architectures and settings with stronger task interference.

More expressive parameter encodings. More expressive representations of parameter indices and structure, such as hash encodings (Müller et al., 2022), may further improve weight generation for complex parameter fields.

Support for non-trainable model state. Extending UHN to generate auxiliary non-trainable state, such as BatchNorm (Ioffe & Szegedy, 2015) running statistics, would broaden the range of architectures it can handle directly.