

We organize our appendix as follows.

Discussion with More Related Works

- Section A.1: Overview of comparisons.
- Section A.2: Discussion with Hyper-Representations.
- Section A.3: Discussion with HyperNetworks.
- Section A.4: Details and limitations of G.pt.
- Section A.5: Details and limitations of p-diff.
- Section A.6: More related works.
- Section A.7: Practical advantages of RPG in real-world applications.

Experimental Setting and Other Details

- Section B.1: Training recipe.
- Section B.2: Detailed structure of recurrent diffusion.
- Section B.3: Description of datasets.

Additional Experimental Results

- Section C.1: Effectiveness of permutation state.
- Section C.2: More results of Section 4.
- Section C.3: Training memory cost analysis.
- Section C.4: Inference memory cost & sampling time.
- Section C.5: Parameter sensitivity vs. performance.
- Section C.6: Details of trained checkpoints collection.
- Section C.7: Impact of the diffusion process.
- Section C.8: Computational cost for training.
- Section C.9: Why not auto-regression?
- Section C.10: Generating from noise vs. mapping from embedding.

A Discussion with More Related Works

A.1 Overview of comparisons

We compare RPG and four other methods from three aspects: scalability, performance, and generalization. Only our RPG excels in all three aspects simultaneously.

method	scalability	performance	generalization
S_{KDE30} [55]	✗	✗	✗
p-diff [68]	✗	✓	✗
SANE [57]	✓	✗	✗
D2NWG [63]	✓	✓	✗
HyperNetwork [24]	✗	✓	✓
HyperFields [2]	✗	✓	✓
ATT3D [42]	✗	✓	✓
HyperNetwork (MAML) [3]	✓	✗	✓
RPG (ours)	✓	✓	✓

Table 14: Comparison of RPG and existing methods on key aspects: scalability, performance, and generalization.

A.2 Discussion with Hyper-Representations

We mainly compare with three HyperRepresentation methods [55, 57, 56]. These methods use an autoencoder to learn the latent features of trained models, so they call the latent feature Hyper-Representation. This HyperRepresentation is then used for analyzing the model’s performance or characteristics, or for sampling to generate new models or pre-trained parameters.

- [55] utilizes kernel density estimation (KDE) to sample model parameters on the learned HyperRepresentation space. They also emphasize the importance of layer-wise loss normalization in the learning process of HyperRepresentation. This work achieves parameter generation in small CNNs from Model Zoos [58] with 2864 parameters.

- [56] focuses on using HyperRepresentation to sample the pre-trained model parameters. They also evaluate the ability of transfer learning by using a trained parameter autoencoder to initialize on unseen dataset. This work can be regarded as a cheap parameter initialization method.
- [57] divides the neural network weights into subsets and utilizes a sequential autoencoder for neural embeddings (SANE) on these subsets with a sliding window. This work can generate the entire parameter of ResNet-18. However, its generation process does not directly derive parameters from noise. Instead, it relies on a half-trained model for Kernel Density Estimation (KDE) sampling.

We summarize the main differences as follows:

- *Hyper-representations as generative models* is hard to achieve comparable results as their original models that are used for training, but our approach obtains comparable results.
- *Hyper-representation for pre-training and transfer learning* focuses on parameter initialization while our approach targets to learn the distribution of high-performing neural network parameters.
- *SANE* is the latest method among these three HyperRepresentation methods. However, SANE uses a sliding window to model the relationship of a small part of trained parameters. Our approach uses a recurrent model among all parameters.
- Our approach can synthesize many popular vision and language parameters, such as ConvNeXt-L and LoRA parameters of LLaMA-7B, with a maximum parameter count of approximately 200M, which is much larger than previous works.

Additionally, the parameters generated by our model can directly achieve almost peak performance without any finetuning. And the generation process is entirely synthesized from noise, eliminating the need for a few prompt examples that are trained for a few epochs, as required by the S_{KDE} [57] sampling.

The capability of large-scale and high-accuracy generation makes our method more applicable in practical scenarios, significantly bridging the gap between theoretical parameter generation and practical application.

A.3 Discussion with HyperNetworks

The early HyperNetworks [24] can only generate a small number of parameters and cannot scale up, which limits their applications in many domains. However, in certain areas, researchers improve HyperNetworks to develop some interesting works.

- HyperFields [2] introduces a dynamic HyperNetwork that maps text directly to NeRF parameters, enabling zero-shot or few-shot 3D scene generation. By combining NeRF distillation with this architecture, it achieves faster training and broader generalization than traditional optimization-based methods.
- ATT3D [42] trains a single model across many text prompts, avoiding per-prompt optimization. This makes text-to-3D generation faster, more generalizable, and capable of smooth interpolations for new assets and animations.
- HyperNetwork with MAML [3] improves HyperNetworks’ design by fixing gradient scaling, regularization, and momentum issues. The resulting HyperNetworks achieve higher accuracy and stronger robustness in meta-learning than standard models.

We summarize the main differences as follows:

- HyperFields and ATT3D aim to generate small, domain-specific networks tailored to particular functions. In contrast, our work explores the unified generation of large models—such as vision transformers and LoRA adapters for large language models—that has emerged in recent years.
- HyperNetwork with MAML focuses on meta-learning and tends to produce a better initialization rather than directly generating good parameters. In contrast, our work aims to directly generate high-quality parameters.

A.4 Details and limitations of G.pt

A primary limitation of G.pt [50] is the training data collection cost. By default, they collect 23 million checkpoints to train the parameter generator. Besides, they only evaluate the effectiveness of G.pt on small architectures, such as a low-dimensional MLP layer or a Convolutional layer with limited channels. The maximum number of generated parameters does not exceed 10,000.

A.5 Details and limitations of p-diff

P-diff [68] directly flattens all parameters into a 1D vector, disregarding the inter-layer parameter relationships. Furthermore, p-diff faces challenges in scaling up to large-scale parameter generation.

A.6 More related works

Diffusion models. Diffusion models [26, 48, 14] gain increasing popularity in recent years, due to their superiority in image generation. Ever since its advent, many works have been done focusing on improving the generation quality and efficiency of diffusion models. For generation quality, [53] propose to conduct diffusion in the latent space, enabling high-resolution image synthesis. [51] leverage the transformer [67] to explore scalability of diffusion models, proving the possibility of generating higher quality images by increasing model size. As for efficiency problem, efficient samplers [61, 43, 62], efficiency models [18, 59, 73], and global acceleration approaches [44, 49] are proposed to increase diffusion models’ efficiency. These methods facilitate generating high quality images with less computational and/or memory cost. Although diffusion models for image generation have achieved great success, improving quality and efficiency in large-scale parameter generation remains to be explored.

A.7 Practical advantages of RPG in real-world applications

RPG offers several practical advantages in real-world scenarios. We list some of them below:

- *Efficient initialization:* RPG provides superior parameter initialization compared to random weights, significantly reducing training time and computational costs.
- *Few-shot learning:* For tasks with limited training data, RPG-generated parameters serve as informed priors that enable better performance than training from scratch.
- *Rapid model deployment:* RPG enables near-instantaneous model adaptation for new tasks without requiring extensive training, making it valuable for real-time applications where quick deployment is critical.

B Experimental Setting and Other Details

B.1 Training recipe

In this section, we provide detailed training recipes and supplementary information. The number of parameters generated by our approach ranges from approximately 3K to 200M. The significant disparity necessitates different training settings. Generally, as the number of parameters increases, the learning process becomes more challenging, requiring higher training costs, particularly for generating parameters beyond 50 million. Therefore, our training settings are divided into two categories: the default setting and the setting for parameters over 50 million, shown in Tab. 15.

Data parallelism: When the number of parameters is less than 50 million, we adopt a single GPU to run the training process. For larger number of parameters, we employ distributed data parallelism to facilitate the training.

Diffusion batch size: In our approach, the diffusion model is shared across all tokens. Typically, all tokens can be fed as a single batch into the diffusion model for training. However, in practice, we randomly select a subset of tokens from a long sequence for training, rather than feeding all parts at once. This approach significantly reduces memory usage without compromising performance. The ‘diffusion batch size’ in Tab. 15 refers to the number of tokens fed into the diffusion model during a single training iteration.

training setting	number parameters < 50M	number parameters > 50M
batch size	16	8
optimizer	AdamW	AdamW
learning rate	3e-5	1e-5
training steps	80,000	120,000
weight decay	1e-5	1e-5
mixed precision	bfloat16	bfloat16
diffusino batch size	1024	512

Table 15: Training recipe in detail.

B.2 Detailed structure of recurrent diffusion

In this section, we provide specific details about the proposed recurrent model and diffusion model in RPG. More detailed configurations can be found in Tab. 16.

module	setting	RPG-Tiny	RPG-Small	RPG-Base	RPG-Large
adequate number of parameters		<50K	50K~10M	5M~50M	>50M
recurrent (Mamba)	d_model of 1st layer	256	4096	8192	12288
	d_model of 2nd layer	256	4096	8192	16384
	d_state	32	128	128	128
	d_conv	4	4	4	4
	expand	2	2	2	2
	parameter counts	1.3M	256M	1018M	3076M
diffusion (1D CNN)	encoder channels	(1, 32, 64, 128)	(1, 32, 64, 128)	(1, 32, 64, 128)	(1, 64, 96)
	decoder channels	(128, 64, 32, 1)	(128, 64, 32, 1)	(128, 64, 32, 1)	(96, 64, 1)
	token size	256	4096	8192	16384
	kernel size	7	7	7	7
	default solver	DDPM	DDPM	DDPM	DDIM
	sampling steps	1000	1000	1000	60
	β -start & β -end	(0.0001, 0.02)	(0.0001, 0.02)	(0.0001, 0.02)	(0.0001, 0.02)
	betas schedule	linear	linear	linear	linear
	number time steps	1000	1000	1000	1000
	parameter counts	0.3M	17M	69M	273M

Table 16: Detailed information about four different sizes of recurrent diffusion. The *adequate number of parameters* implies that our model is usually adequate to generate parameters in that scale, which is empirical results instead of an exact rule. It also necessitates considering other factors such as parameter sensitivity.

Details of recurrent model. By default, the recurrent model consists of two Mamba layers [22]. As the increasing of parameters to generate, we need a larger recurrent model to capture the information in these parameters. The size of the recurrent model is mainly determined by the token size, which varies according to the number of parameters to be generated. Based on the token size, we categorize our model into four versions: Tiny, Small, Base, and Large.

Details of diffusion model. Following p-diff [68], our diffusion model adopts a 1D convolutional architecture. The parameters of the diffusion model are significantly fewer than those of the recurrent model. We feed the prototypes from the recurrent model as conditions into the diffusion model by directly adding them to the feature map.

The setting of our main experiments. In our main experiments, ViT-Base, ConvNeXt-Large, ADE20K, COCO, and DoRA rank 64 used the setting of RPG-Large. CNN (s) and CNN (m) used the setting of RPG-Tiny. And all other experiments used the setting of RPG-Base.

B.3 Description of datasets

In this section, we introduce the datasets used in the paper, including those for classification, semantic segmentation, object detection&instance segmentation, and commonsense reasoning.

Classification: ImageNet-1k [13] is a large-scale visual database for visual object recognition research. It contains over 1 million images across 1000 categories and is widely used for training and benchmarking deep learning models. **CIFAR-10** [33] dataset consists of 60,000 32×32 colorful images in 10 different classes. It is commonly used for training machine learning and computer vision algorithms, providing a standard benchmark for image classification task.

Semantic segmentation: ADE20K [77] is a dataset for semantic segmentation and scene parsing, containing over 20,000 images annotated with pixel-level labels for 150 object categories. It is used to train models to understand and segment various objects and scenes in an image, making it valuable for applications in autonomous driving, robotics, and image editing.

Instance segmentation & object detection: COCO [40] dataset is a large-scale object detection, segmentation, and captioning dataset. It contains over 330,000 images in various resolutions, with more than 200,000 labeled instances across 80 object categories. COCO is widely used for training and evaluating models in object detection, segmentation, and image captioning tasks.

Commonsense reasoning: BoolQ [10]: Yes/No questions based on natural passages. **PIQA** [5]: Questions about physical tasks and actions. **SIQA** [54]: Questions about social interactions. **HellaSwag** [75]: Choosing the correct ending for stories. **ARC** [11]: Multiple-choice science questions. **OBQA** [46]: Questions requires multi-step reasoning, commonsense knowledge, and rich text comprehension.

C Additional Experimental Results

C.1 Effectiveness of permutation state

The effect of permutation state. RPG incorporates a permutation state operation to address parameter symmetries, which become particularly pronounced when collecting checkpoints from multiple training runs. To evaluate this, we collect checkpoints from different numbers of training runs (1, 3, and 10) and compare the performance with and without permutation state. These results demonstrate that permutation state operation effectively addresses parameter symmetries and enables stable results even when incorporating checkpoints from multiple training runs.

collected runs	original	w/o permutation state	w/ permutation state
1	88.2	88.0	88.1
3	88.3	fail	88.1
10	88.5	fail	88.2

Table 17: Permutation state effectively mitigates parameter symmetries when collecting checkpoints from different training runs.

Analysis of LMC for evaluating model similarity. Linear mode connectivity [19, 40] (LMC) is a technique used to assess the similarity between deep neural network models by examining the performance of linear interpolations between their weight vectors. This metric would actually suggest diversity in the sense of models belonging to different basins [1] which is more meaningful.

In this experiment, we use ResNet-18 on CIFAR-10 with different training and sampling schemes, and test the performance of linear model weights interpolation. The results in Tab. 18 suggest that LMC can be broken in sampled models when multiple permutation states are applied. This further implies that RPG can learn the distribution of parameters, and its generating process is not merely a linear interpolation of the training data.

method \ accuracy (%)	A	$0.5A + 0.5B$	B
trained on checkpoints from the same basin	95.1	95.0	95.0
trained on checkpoints from 2 basins and sampled with 1 permutation state	95.0	94.8	94.9
trained on checkpoints from 2 basins and sampled with 2 permutation states	94.9	18.4	94.9

Table 18: LMC of two RPG-generated models under different training and sampling settings.

C.2 More results of Section 4

Results of generating models for unseen tasks. In Section 4, we show the potential of our approach in generating models for unseen tasks. In this part, we provide more results. First, we compare the performance of original and generated models using all unseen embeddings in Tab. 19. Results demonstrate that our approach consistently achieves good results in unseen tasks.

unseen binary embeddings	original	best accuracy	average accuracy	standard deviation
0 1 0 0 0 1 0 1 1 1	97.3	94.4	93.9	0.6
0 1 1 1 1 1 0 1 1 0	98.1	96.6	94.9	2.1
0 0 1 1 1 0 1 1 1 0	97.4	95.0	94.2	1.1
0 1 0 1 1 1 1 1 1 1	98.4	96.1	95.8	0.3
0 0 1 0 0 0 0 0 0 0	98.9	96.6	95.2	2.3
0 0 0 1 1 0 0 1 0 1	96.7	92.9	91.6	1.1
1 1 1 1 1 0 1 0 0 1	97.6	94.8	94.1	0.7
1 0 1 0 0 0 0 0 1 1	98.1	95.7	91.8	3.7
0 1 0 0 0 1 0 1 1 0	97.1	93.6	90.7	4.3
1 1 0 0 0 1 1 0 0 1	97.0	94.0	90.1	3.6
1 0 1 0 0 0 1 1 0 1	97.3	91.3	90.7	0.8
0 1 1 1 1 0 0 0 1 0	96.3	95.4	89.4	6.3
1 1 0 1 1 1 0 1 0 0	97.6	92.6	90.5	3.2
0 1 1 1 0 0 1 1 1 0	96.3	90.8	89.1	1.9
0 1 0 0 1 1 1 0 1 0	96.3	91.9	88.4	4.4
0 0 1 0 0 0 1 1 0 1	97.5	93.7	88.0	5.6
0 0 0 1 1 0 1 1 1 1	96.5	90.8	85.5	7.0
1 0 0 1 0 0 1 1 0 1	96.4	86.7	83.7	3.6
1 0 0 1 0 1 0 0 0 0	97.7	85.6	83.2	2.0
0 0 1 1 0 0 0 1 0 1	96.3	90.2	79.2	9.6

Table 19: Performance comparisons between original and generated models on unseen tasks. Specifically, we generated 10 models for each unseen task, with binary embeddings in the unseen set as condition. The results consistently show that our generated models perform comparably to the original models.

More experiments for ViT-Small on Coarse ImageNet-1K [64]. To verify the generalization potential of RPG on larger-scale datasets and models, we evaluated RPG on the ImageNet-1K dataset (coarse-grained 10 classes) [64] using the ViT-Small architecture. In Tab. 20, experimental results show that, despite not using any training data when generating parameters for new tasks, RPG achieves performance comparable to that of traditionally trained models. Moreover, RPG demonstrates strong generalization ability and adaptability across increasingly large datasets and models (from CIFAR10 to ImageNet, and from ViT-Tiny to ViT-Small), further validating its potential for practical applications.

unseen tasks (embeddings)	original	RPG
0 0 0 0 0 1 0 1 0 1	91.5	89.7
0 1 1 0 1 0 0 0 0 1	88.5	87.0
1 1 1 1 1 1 0 0 1 0	89.1	88.1
1 1 0 0 0 0 1 0 1 1	89.3	87.3
1 0 1 0 1 0 1 1 0 1	91.4	90.4
1 1 0 1 1 1 0 0 0 0	88.6	87.2
1 0 1 0 1 1 1 1 1 0	92.5	90.1
0 1 1 1 1 0 1 0 1 0	89.5	87.5
1 0 1 1 0 1 0 0 0 1	90.1	87.9
1 1 0 1 1 0 0 0 0 0	90.2	89.2

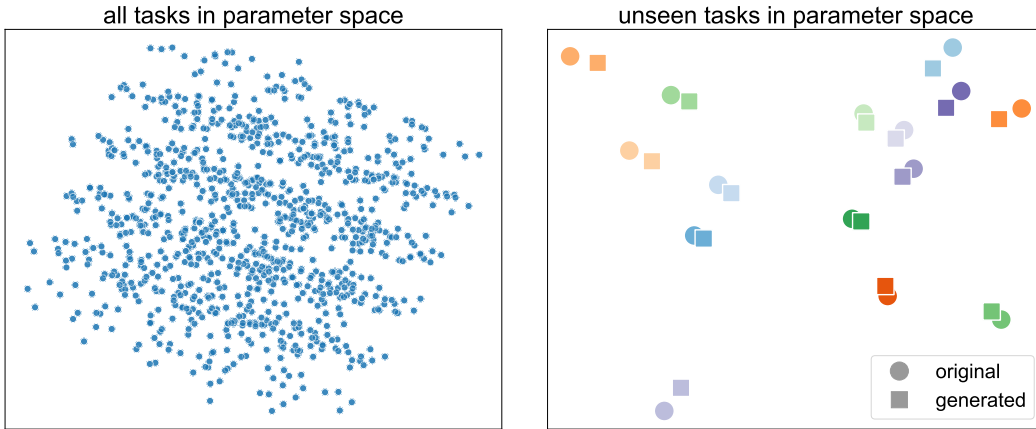
Table 20: Result comparisons between original and generated models for generated ViT-Small on Coarse ImageNet-1K [64].

More experiments for YOLOv8n [66] on PASCAL VOC [17]. For more complex scenarios, we employ YOLOv8n to demonstrate RPG’s ability to generate models for unseen tasks. We use the PASCAL VOC dataset, which contains 20 classes, and design 20-bit positional embeddings to specify the target detection requirements (‘1’ indicates that a class must be detected, and ‘0’ otherwise). We then train 500 YOLOv8n models on 500 randomly sampled tasks and collect their checkpoints. Performance is evaluated using mAP50-95, comparing models fine-tuned from pretrained weights against those initialized with generated parameters. The results in Tab. 21 show that RPG provides strong initializations for new tasks, enabling faster adaptation and reducing the need for extensive training. This confirms our method’s scalability and effectiveness on more complex tasks.

unseen tasks (embeddings)																epoch-0	epoch-1	epoch-2	epoch-5					
0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	1	0.00 / 0.29	0.01 / 0.30	0.06 / 0.33	0.18 / 0.39
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0.01 / 0.25	0.02 / 0.30	0.05 / 0.32	0.21 / 0.39
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0.01 / 0.28	0.09 / 0.33	0.12 / 0.35	0.32 / 0.38
0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0.00 / 0.24	0.04 / 0.25	0.04 / 0.29	0.13 / 0.35
0	0	0	1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0.00 / 0.19	0.03 / 0.22	0.11 / 0.26	0.21 / 0.33

Table 21: Result comparisons between original and generated models for generated YOLOv8n [66] on PASCAL VOC [17].

PCA visualization of classification head parameters. We also provide a visualization of the parameters of the classification head (a two-layer fully connected structure with total 38,976 parameters) for 1022 tasks as described in Section 4 using Principal Component Analysis (PCA), which presents the structure of the parameter space in Fig. 6a. Our generated model achieves an average accuracy of 91.2% across all binary classification tasks, which indicates that our method has effectively learned this structure. Furthermore, we evaluate the parameters corresponding to unseen tasks and compared their positions in Fig. 6b between the original and generated parameters. It is noteworthy that, even though the original parameters of these tasks are not included in the training data, the generated parameters consistently appeared in close proximity to the original ones. This observation further highlights the capability of our method to model the structure of the parameter space, even for tasks not previously encountered.



(a) Visualization of the classification head of all 1022 tasks. This reveals that there is an inherent structure among the high-dimensional parameters, which can be captured by deep learning models.

(b) Visualization of the classification head in some unseen tasks. Parameters associated with the same task are indicated by a consistent color. Our method can capture the structure of the parameter space.

Figure 6: Principal Component Analysis (PCA) visualization of the classification head. The figures demonstrate the presence of an inherent structure in the parameter space and highlight our method’s effectiveness in capturing this structure for unseen tasks.

Conditional generation guided by LLM. To demonstrate the application scenarios of our model, we utilize large language model to generate binary embeddings to guide RPG in generating corresponding classification models. For the first example in Fig 7, we give the LLM a prompt: ‘Give me a model to select all living things.’ With the binary embedding provided by the LLM, our RPG then generates a ViT-Tiny classifier. After that, We use images in CIFAR-10 to evaluate the model’s accuracy. The model should classify ‘bird’, ‘cat’, ‘deer’, ‘dog’, ‘frog’, and ‘horse’ to the positive class, which we used as the ground truth. The result is 97.1%. Some other examples are in Tab 22. This experiment demonstrates our method’s capability to generate neural network parameters based on natural language guidance, highlighting the potential applications of our method.

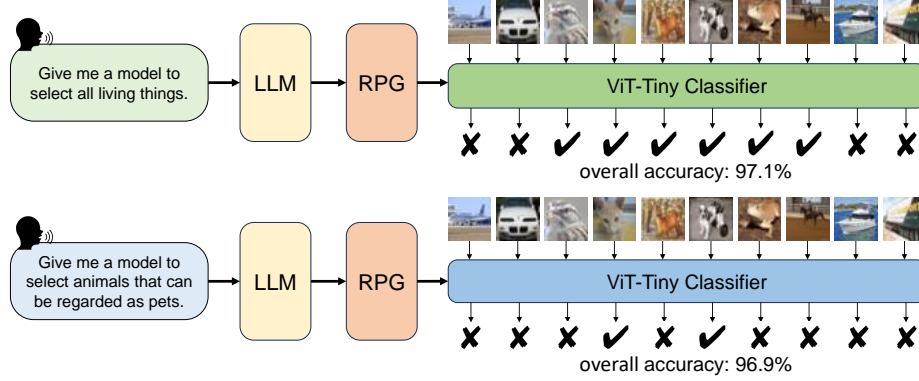


Figure 7: Illustration of RPG-generated models guided by binary embeddings from a large language model (Qwen2.5-3B [72]), demonstrating parameter generation conditioned by natural language.

prompt	expected embedding	acc. (%)
Give me a model to select all living things.	0,0,1,1,1,1,1,0,0	98.7
Find all vehicles that operate on roads.	0,1,0,0,0,0,0,0,1	95.9
Classify all mammals.	0,0,0,1,1,1,0,1,0,0	97.6
Select all man-made objects.	1,1,0,0,0,0,0,0,1,1	97.7
Find all things that are both man-made and can operate on water.	0,0,0,0,0,0,0,0,1,0	98.4
Select all animals that can be regarded as pets.	0,0,0,1,0,1,0,0,0,0	96.8
Select all things that can fly.	1,0,1,0,0,0,0,0,0,0	87.7
Find all animals with fur.	0,0,1,1,1,1,0,1,0,0	70.4
Select all pets commonly found in households.	0,0,1,1,0,1,0,0,0,0	83.3
Identify all cold-blooded animals.	0,0,0,0,0,0,1,0,0,0	98.9

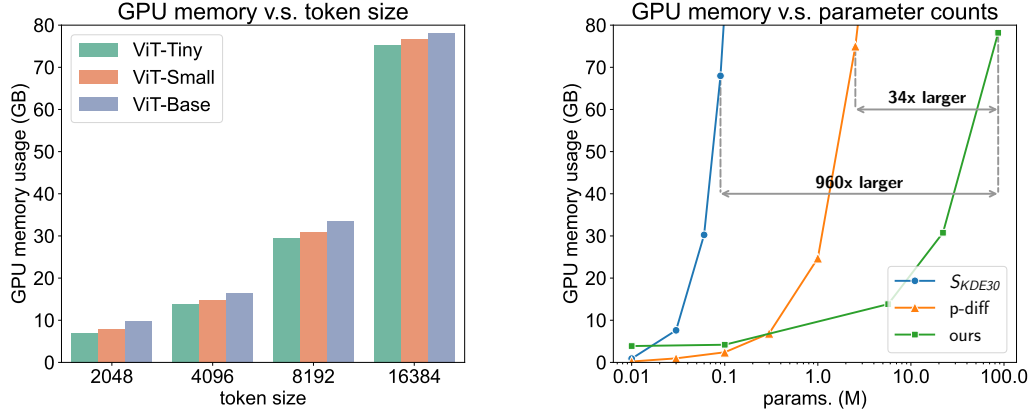
Table 22: The table demonstrates examples of generating from abstract prompts using LLM and RPG. We are able to achieve high accuracy on abstract prompts. The *expected embedding* is used for evaluating the accuracy.

C.3 Training memory cost analysis

In this section, we analyze the GPU memory utilization during training. GPU memory consumption is usually highly correlated with two factors: i) the size of the generative model and ii) the size of generated parameters. We analyzed the impact of these two factors on the GPU memory utilization during the training of our approach.

GPU Memory vs. Token Size. In Tab. 5, we present the relationship between token size and RPG’s learning capability. Despite the fact that larger token sizes (i.e., larger models) carry a greater volume of information, they also lead to substantial GPU memory costs, as illustrated in Fig. 8a. This implies that, when the performance of the generated models is comparable, we prefer to use models with smaller token sizes.

GPU memory v.s. parameter counts. We conduct experiments to show the relationship between GPU memory and generated parameter counts in Fig. 8b. In previous methods, the relationship between GPU memory consumption and the number of parameters in the generated model was quadratic [55] or directly proportional [68]. This limits their practicality and application range. In contrast, our approach demonstrates remarkable efficiency: with equivalent GPU memory usage, it can generate models with 34 to 960 times more parameters compared to previous methods.



(a) Visualization of GPU memory v.s. token size. GPU memory usage increases proportionally to the token size. Thus, the token size cannot get larger infinitely; we need to choose a proper token size.

(b) Visualization of GPU memory v.s. parameter counts. Our method can generate much more parameters than existing approaches e.g. S_{KDE30} [55] using a single NVIDIA H100 80G GPU.

Figure 8: Training memory cost analysis. *Left*: GPU memory v.s. token size. *Right*: GPU memory v.s. parameter counts.

C.4 Inference memory cost & sampling time

In this section, we present more information about the sampling, including memory usage, inference time, and the balance between sequential and parallel inference. In Tab. 9, we show the sampling time and memory usage for ViT-Base and ConvNeXt-L. Here, we present the sampling time and memory usage for other models. In Tab. 23, we adopt DDPM as the solver and conduct 1000-step sampling. Since the diffusion model in RPG is shared among all the parameter tokens, we can adopt different inference modes to find a balance between memory usage and inference speed:

- **fully parallel:** All tokens are fed into the diffusion model simultaneously. This approach results in a high memory usage but achieves a high generation speed.
- **sequential:** Tokens are fed into the diffusion model one by one. This approach significantly reduces memory usage, as the model only occupies memory for inferring a single token at a time. This enable us to generate parameters of models listed on a GPU with less than 8GB of memory .
- **partially parallel (default):** In partial parallel mode, we set 256 tokens as a batch for the diffusion model inference. This approach significantly boosts speed with a slight increase in GPU memory usage, reaching an optimal trade-off between memory and speed. We adopt this as the default setting.

Based on the results in Tab. 23, our approach can be flexibly applied to many other GPUs as it can achieve a good trade-off between memory and time.

metrics	inference mode	ResNet-18	ResNet-50	ViT-Tiny	ViT-Small	ConvNeXt-A
time (minute)	sequential	18.6	38.0	9.8	33.8	6.8
	partially parallel	1.8	3.3	1.1	2.9	0.9
	fully parallel	1.7	3.3	1.1	2.9	0.9
memory cost (GB)	sequential	6.3	6.4	6.2	6.4	6.2
	partially parallel	10.3	10.5	10.3	10.5	10.3
	fully parallel	30.8	50.5	19.4	45.9	15.2

Table 23: We show the inference time and memory cost under different inference modes. All information in this table is collected from a single NVIDIA H100 80G GPU. We report the time and memory required to generate a single model.

C.5 Parameter sensitivity vs. performance

According to conventional understanding, larger parameter quantities are generally more challenging to learn. However, our experiments reveal that this rule is not absolute and demonstrates instability in learning some small model parameters.

This motivates us to investigate the relationship between parameter sensitivity and generation quality. Specifically, we add Gaussian noise with weights of 0.01, 0.10, and 1.00 to the original parameters to measure model sensitivity, as shown in Tab. 24. We observe that as noise weight increases, performance decreases for all models, with smaller models being more affected than larger ones. This indicates that smaller models are relatively more sensitive. Additionally, we notice that the performance gap between the original and generated models widens as sensitivity of the model increases. This demonstrates a strong correlation between a model’s sensitivity and the difficulty of generating its parameters.

model	params. (M)	sensitivity	accuracy decline			
			ours	noise (0.01)	noise (0.10)	noise (1.00)
ConvNeXt-A	3.7	+++	0.85	62.83	0.60	0.03
ResNet-18	11.7	++	0.39	53.56	0.46	0.00
ViT-Base	86.6	+	0.09	5.39	0.02	0.00

Table 24: The accuracy decline reflects the accuracy gap between the original model and the generated model or the model after adding noise. We add Gaussian noise with weights of 0.01, 0.10, and 1.00 to the parameters to measure model sensitivity. Results demonstrate that smaller models are relatively more sensitive than larger ones. The more plus signs (+), the higher the sensitivity.

C.6 Details of trained checkpoint collection

This section mainly supplements the collection of checkpoints in Section 3. First, we obtain the pre-trained models, which either come from model libraries (such as timm [70]) or are trained by ourselves. Then, we fine-tune the full model for one epoch, and save 50 checkpoints during this epoch uniformly. In Tab 17, the term *collected runs* refers to the number of times the entire process, from pre-training to fine-tuning and saving, is repeated. This is done without fixing the seed, resulting in checkpoints from entirely different permutations.

In addition, we also compared the impact of the number of collected checkpoints on the similarity between models. Based on the results in Tab. 25, the analysis reveals that using too few checkpoints lead to low diversity. However, as the number of checkpoints increases, the similarity initially decreases and then stabilizes around 0.84 when the count exceeds 50.

number of checkpoints	1	10	50	200	400
similarity	0.93	0.89	0.84	0.83	0.84

Table 25: Comparison between the number of checkpoints and similarity.

C.7 Impact of the diffusion process

We conduct experiments to analyze the effect of the diffusion process. We compare the performance of our method with and without the diffusion process. The results are obtained by training with ViT-Tiny. The findings are as follows: (1) using the diffusion process allows for generating an infinite number of models, whereas the version without the diffusion process can produce only one model; (2) incorporating the diffusion process increases the diversity between the original and generated models (0.84 vs. 0.91); (3) the model generated without the diffusion process tends to memorize the ensemble of original models more strongly, as indicated by a high similarity score of 0.93.

diffusion process	number of generated models	similarity	similarity with ensembled original models	accuracy (%)
✓	infinite	0.84	0.85	75.3
✗	only one model	0.91	0.93	75.3

Table 26: Comparison of with and without diffusion processes.

C.8 Computational cost for training

Training RPG requires computational resources. However, all experiments can be completed within 144 GPU hours ($8 \text{ GPUs} \times 18 \text{ hours}$), as shown in Tab. 27.

model	number of parameters	training cost (GPU hours)
ViT-Tiny	6M	4
DoRA (rank4)	8M	4
ViT-Small	22M	10
ViT-Base	86M	54
DoRA (rank64)	113M	73
ADE20K	177M	132
ConvNeXt-L	198M	144
ViT-Tiny (Section-4)	6M	40

Table 27: Training cost of RPG for various model architectures.

C.9 Why not auto-regression?

It is worth noting that our approach does not employ an auto-regressive method, *i.e.*, we do not feed the output back as input again. Our method takes position embedding and permutation state as inputs and synthesizes neural network parameters as outputs, forming a standard recurrent neural network. We have attempted to train our model using an auto-regressive approach such as a decoder-only transformer architecture, whose results are shown in Tab 6. To ensure a fair comparison, we also applied a causal mask to the transformer encoder-only structure, ensuring that information can only be passed in one direction. Due to the accumulation of errors during the auto-regressive process in inference, the parameters generated at the end of sequence become nearly indistinguishable from noise, leading to poor performance. In contrast, in RPG, noise is only introduced in the diffusion model and does not accumulate in the recurrent process, ensuring stable parameter generation.

C.10 Generating from noise vs. mapping from embedding

When generating neural network parameters, methods that synthesize weights from noise—such as RPG—offer significant advantages in memory efficiency and scalability over approaches that map learnable embeddings directly to full parameter tensors.

- Lower memory footprint: Embedding-based decoders must produce the entire parameter tensor in one go, causing memory usage to scale linearly with model size. In contrast, RPG generates weights sequentially, drastically reducing peak memory consumption.
- Superior scalability: RPG’s autoregressive generation process naturally accommodates large models without requiring storage or computation of massive, dense parameter tensors at once.
- Favorable compute-memory trade-off: While RPG incurs a modest increase in compute due to sequential generation, it achieves orders-of-magnitude memory savings—making it far more practical for deployment in resource-constrained or large-scale settings.