GRAPHPNAS: LEARNING DISTRIBUTIONS OF GOOD NEURAL ARCHITECTURES VIA DEEP GRAPH GENERATIVE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Neural architectures can be naturally viewed as computational graphs. Motivated by this perspective, we, in this paper, study neural architecture search (NAS) through the lens of learning random graph models. In contrast to existing NAS methods which largely focus on searching for a single best architecture, *i.e.*, point estimation, we propose *GraphPNAS*, a deep graph generative model that learns a distribution of well-performing architectures. Relying on graph neural networks (GNNs), our GraphPNAS can better capture topologies of good neural architectures and relations between operators therein. Moreover, our graph generator leads to a learnable probabilistic search method that is more flexible and efficient than the commonly used RNN generator and random search methods. Finally, we learn our generator via an efficient reinforcement learning formulation for NAS. To assess the effectiveness of our GraphPNAS, we conduct extensive experiments on three search spaces, including the challenging RandWire on Tiny-ImageNet, ENAS on CIFAR10, and NAS-Bench-101. The complexity of RandWire is significantly larger than other search spaces in the literature. We show that our proposed graph generator consistently outperforms RNN based one and achieves better or comparable performances than state-of-the-art NAS methods.

1 INTRODUCTION

In recent years, we have witnessed a rapidly growing list of successful neural architectures that underpin deep learning, *e.g.*, VGG, LeNet, ResNets (He et al., 2016), Transformers (Dosovitskiy et al., 2020). Designing these architectures requires researchers to go through time-consuming trial and errors. Neural architecture search (NAS) (Zoph & Le, 2016; Elsken et al., 2018b) has emerged as an increasingly popular research area which aims to automatically find state-of-the-art neural architectures without human-in-the-loop.

NAS methods typically have two components: a search module and an evaluation module. The search module is expressed by a machine learning model, such as a deep neural network, designed to operate in a high dimensional search space. The search space, of all admissible architectures, is often designed by hand in advance. The evaluation module takes an architecture as input and outputs the reward, *e.g.*, performance of this architecture trained and then evaluated with a metric. The learning process of NAS methods typically iterates between the following two steps. 1) The search module produces candidate architectures and sends them to the evaluation module; 2) The evaluation module evaluates these architectures to get the reward and sends the reward back to the search module. Ideally, based on the feedback from the evaluation module, the search module should learn to produce better and better architectures. Unsurprisingly, this learning paradigm of NAS methods fits well to reinforcement learning (RL).

Most NAS methods (Liu et al., 2018b; White et al., 2020; Cai et al., 2019) only return a single best architecture (*i.e.*, a point estimate) after the learning process. This point estimate could be very biased as it typically underexplores the search space. Further, a given search space may contain multiple (equally) good architectures, a feature that a point estimate cannot capture. Even worse, since the learning problem of NAS is essentially a discrete optimization where multiple local minima exist, many local search style NAS methods (Ottelander et al., 2020) tend to get stuck in local minima.

From the Bayesian perspective, modelling the distribution of architectures is inherently better than point estimation, *e.g.*, leading to the ability to form ensemble methods that work better in practice. Moreover, modelling the distribution of architectures naturally caters to probabilistic search methods which are better suited for avoiding local optima, *e.g.*, simulated annealing. Finally, modeling the distribution of architectures allows to capture complex structural dependencies between operations that characterize good architectures capable of more efficient learning and generalization.

Motivated by the above observations and the fact that neural architectures can be naturally viewed as attributed graphs, we propose a probabilistic graph generator which models the distribution over good architectures using graph neural networks (GNNs). Our generator excels at generating topologies with complicated structural dependencies between operations. From the Bayesian inference perspective, our generator returns a distribution over good architectures, rather than a single point estimate, allowing to capture the multi-modal nature of the posterior distribution of good architectures and to effectively average or ensemble architecture (sample) estimates. Different from the Bayesian deep learning (Neal, 2012; Blundell et al., 2015; Gal & Ghahramani, 2016) that models distributions of weights/hidden units, we model distributions of neural architectures. Lastly, our probabilistic generator is less prone to the issue of local minima, since multiple random architectures are generated at each step during learning. In summary, our key contributions are as below.

- We propose a GNN-based graph generator for neural architectures which empowers a learnable probabilistic search method. To the best of our knowledge, we are the first to explore learning deep graph generative models as generators in NAS.
- We explore a significantly larger search space (*e.g.*, graphs with 32 operators) than the literature (*e.g.*, garphs with up to 12 operators) and propose to evaluate architectures under low-data regime, which altogether boost effectiveness and efficiency of our NAS system.
- Extensive experiments on three different search spaces show that our method consistently outperforms RNN-based generators and is slightly better or comparable to the state-of-the-art NAS methods. Also, it can generalize well across different NAS system setups.

2 Related Works

Neural Architecture Search. The main challenges in NAS are 1) the hardness of discrete optimization, 2) the high cost for evaluating neural networks, and 3) the lack of principles in the search space design. First, to tackle the discrete optimization, evolution strategies (ES) (Elsken et al., 2019; Real et al., 2019), reinforcement learning (RL) (Baker et al., 2017; Zhong et al., 2018; Pham et al., 2018; Liu et al., 2018a), Bayesian optimization (Bergstra et al., 2013; White et al., 2019) and continuous relaxations (Liu et al., 2018b) have been explored in the literature. We follow the RL path as it is principled, flexible in injecting prior knowledge, achieves the state-of-the-art performances (Tan & Le, 2019), and can be naturally applied to our graph generator. Second, the evaluation requires training individual neural architectures which is notoriously time consuming(Zoph & Le, 2016). Pham et al. (2018); Liu et al. (2018b) propose a weight-sharing supernet to reduce the training time. Baker et al. (2018) use a machine learning model to predict the performance of fully-trained architectures conditioned on early-stage performances. Brock et al. (2018); Zhang et al. (2018) directly predict weights from the search architectures via hypernetworks. Since our graph generator do not relies on specific choice of evaluation method, we choose to experiment on both oracle training(training from scratch) and supernet settings for completeness. Third, the search space of NAS largely determines the optimization landscape and bounds the best-possible performance. It is obvious that the larger the search space is, the better the best-possible performance and the higher the search cost would likely be. Besides this trade-off, few principles are known about designing the search space. Previous work (Pham et al., 2018; Liu et al., 2018b; Ying et al., 2019; Li et al., 2020) mostly focuses on cell-based search space. A cell is defined as a small (*e.g.*, up to 8 operators) computational graph where nodes (*i.e.*, operators like 3×3 convolution) are connected following some topology. Once the search is done, one often stacks up multiple cells with the same topology but different weights to build the final neural network. Other works (Tan et al., 2019; Cai et al., 2019; Tan & Le, 2019) typically fix the topology, *e.g.*, a sequential backbone, and search for layer-wise configurations (*e.g.*, operator types like 3×3 vs. 5×5 convolution and number of filters). In our method, to demonstrate our graph generator's ability in exploring large topology search space, we first explore on a challenging large cell space (32 operators), after which we experiment on ENAS Macro (Pham

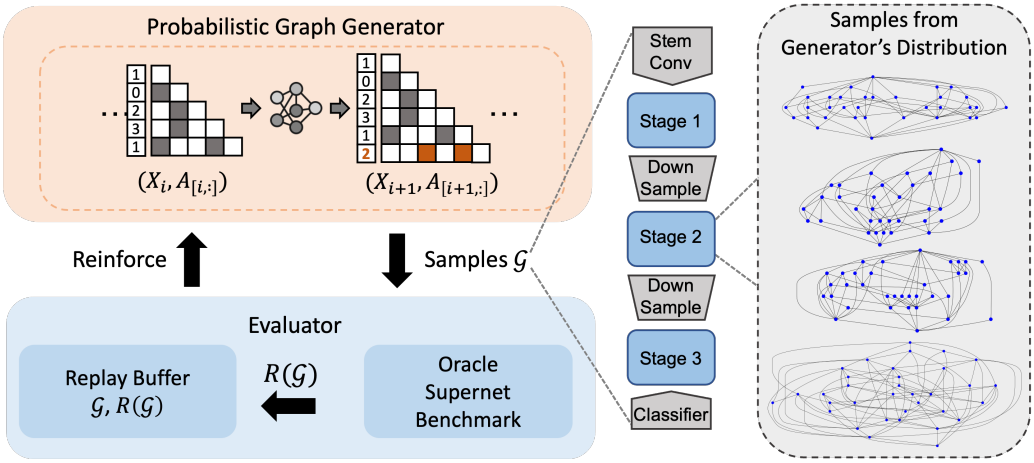


Figure 1: The pipeline of our NAS system. The core part is a GNN-based graph generator from which we sample graph representations of neural network $\mathcal{G}$. The corresponding model for each $\mathcal{G}$ is then sent to evaluator for evaluation. The evaluation result is first stored in a replay buffer and then used for learning the graph generator through Reinforcement Learning.

et al., 2018) and Nas-Benchmark-101(Ying et al., 2019) for more comparison with previous methods. Closely related to our work, Xie et al. (2019) explore different topologies of the similar cell space using non-learnable random graph models and You et al. (2020) subsequently investigate the relationship between topologies and performances. They are different from our work as we learn the graph generator to automatically explore the cell space.

Deep Graph Generative Models. Graph generative models date back to the Erdős–Rényi model (Erdös & Rényi, 1959), of which the probability of generating individual edges is the same. Other well-known graph generative models include the stochastic block model (Holland et al., 1983), the small-world model (Watts & Strogatz, 1998), and the preferential attachment model (Barabási & Albert, 1999). Recently, deep graph generative models instead parameterize the probability of generating edges and nodes using deep neural networks in, *e.g.*, the auto-regressive fashion (Li et al., 2018; You et al., 2018; Liao et al., 2019) or variational autoencoder fashion (Kipf & Welling, 2016; Grover et al., 2018; Liu et al., 2019). These models are highly flexible and can model complicated distributions of real-world graphs, *e.g.*, molecules (Jin et al., 2018), road networks (Chu et al., 2019), and program structures (Brockschmidt et al., 2018). Our graph generator builds on top of the state-of-the-art deep graph generative model in (Liao et al., 2019) with several important distinctions. First, instead of only generating nodes and edges, we also generate node attributes (*e.g.*, operator types in neural architectures). Second, since good neural architectures are actually latent, our learning objective maximizes the expected reward (*e.g.*, validation accuracies) rather than the simple log likelihood, thus being more challenging.

3 METHODS

The architecture of any feedforward neural network can be naturally represented as a directed acyclic graph (DAG), a.k.a., *computational graph*. There exist two equivalent ways to define the computational graph. First, we denote operations (*e.g.*, convolutions) as nodes and denote operands (*e.g.*, tensors) as edges which indicate how the computation flows. Second, we denote operands as nodes and denote operators as edges. We adopt the first view. In particular, a neural network $\mathcal{G}$ with N operations is defined as a tuple (A, X) where $A \in \{0,1\}^{N \times N}$ is an $N \times N$ adjacent matrix with $A_{ij} = 1$ indicates that the output of the j-th operator is used as the input of the i-th operator. For operator with multiple inputs, inputs are combined together (*e.g.*, using `sum` or `average` operator) before sending into the operator. X is a N-size *attribute* vector encoding operation types. For any operation i, its operation type X_i can only choose from a pre-defined list with length D, *e.g.*, 1×1, 3×3 or 5×5 convolutions. Note that for any valid feedforward architecture, $\mathcal{G}$ can not have loops. One sufficient condition to satisfy the requirement is to constrain A to be a lower triangu-

lar matrix with zero diagonal (*i.e.*, excluding self-loops). This formalism creates a search space of $D^N 2^{N(N-1)/2}$ possible architectures, which is huge even for moderately large number of operators N and number of operation types D. The goal of NAS is to find an architecture or a set of architectures within this search space that would perform well. For practical consideration, we search for cell graphs (*e.g.*, $N = 32$) and then replicate this cell several times to build a deep neural architecture. We also experiment on the ENAS Macro search space where $\mathcal{G}$ defines a entire network. More details for the corresponding search spaces can be found in Section 4.

3.1 Neural Architecture Search System

Before delving into details, we first give an overview of our NAS system, which consists of two parts: a generator and an evaluator. The system diagram is shown in Fig. 1. At each step, the probabilistic graph generator samples a set of cell graphs, which are further translated to neural architectures by replicating the cell graph multiple times and stacking them up. Then the evaluator evaluates these architectures, obtains rewards, and sends architecture-reward pairs to the replay buffer. The replay buffer is then used to improve the generator, effectively forming a reinforcement learning loop.

3.1.1 Probabilistic Generators for Neural Architectures

Now we introduce our probabilistic graph generator which is based on a state-of-the-art deep auto-regressive graph generative model in (Liao et al., 2019).

Auto-Regressive Generation. Specifically, we decompose the distribution of a cell graph along with attributes (operation types) in an auto-regressive fashion,

$$\mathbb{P}(A, X) = \prod_{i=1}^{N} \mathbb{P}\left(A_{i,:} | A_{i-1,:}, X_{i-1}, \cdots, A_{1,:}, X_1\right) \mathbb{P}\left(X_i | A_{i-1,:}, X_{i-1}, \cdots, A_{1,:}, X_1\right), \tag{1}$$

where $A_{i,:}$ and X_i denote the i-th row of the adjacency matrix A and the i-th operation type respectively. To ensure the generated graphs are DAGs, we constrain A to be lower triangular by adding a binary mask, *i.e.*, the i-th node can only be reached from the first $i-1$ nodes. We omit the masks in the equations for better readability. We further model the conditional distributions as follows,

$$\mathbb{P}\left(A_{i,:} | A_{i-1,:}, X_{i-1}, \cdots, A_{1,:}, X_1\right) = \sum_{k=1}^{K} \alpha_k \prod_{1 \le j < i} \theta_{k,i,j} \tag{2}$$

$$\mathbb{P}\left(X_i | A_{i-1,:}, X_{i-1}, \cdots, A_{1,:}, X_1\right) = \text{Categorical}\left(\beta_1, \cdots, \beta_D\right) \tag{3}$$

$$\alpha_1, \ldots, \alpha_K = \text{Softmax}\left(\sum\nolimits_{1 \le j < i} \text{MLP}_\alpha(h_i^S - h_j^S)\right) \tag{4}$$

$$\beta_1, \ldots, \beta_D = \text{Softmax}\left(\text{MLP}_\beta(h_i^S)\right) \tag{5}$$

$$\theta_{1,i,j}, \ldots, \theta_{K,i,j} = \text{Sigmoid}\left(\text{MLP}_\theta(h_i^S - h_j^S)\right), \tag{6}$$

where the distributions of the operation type and edges are categorical and K-mixture of Bernoulli respectively. D is again the number of operation types. MLP_α, MLP_β, and MLP_θ are different instances of two-layer MLPs with ReLU activations. Here h_i^S is the representation of i-th node returned by a GNN which has been executed S steps of message passing at each generation step. This auto-regressive construction breaks down the nice property of permutation invariance for graph generation. However, we do not find it as an issue in practice, partly due to the fact that the graph isomorphism becomes less likely to happen while considering both topology and operation types.

Message Passing GNNs. Each generation step $n \le N$ in auto-regressive generation above relies on representations of nodes up to and including n itself (see Eq. (4)–(6)). To obtain these node representations $\{h_i^S\}$, we exploit message passing GNNs (Gilmer et al., 2017) with an attention mechanism similar to (Liao et al., 2019). In particular, the s-th message passing step involves executing the following equations successively,

$$m_{ij}^s = f([h_i^s - h_j^s, \mathbf{1}_{ij}]) \tag{7}$$

$$\tilde{h}_i^s = [h_i^s, u_i] \tag{8}$$

$$a_{ij}^s = \text{Sigmoid}(g(\tilde{h}_i^s - \tilde{h}_j^s)) \tag{9}$$

$$h_i^{s+1} = \text{GRU}(h_i^s, \sum\nolimits_{j \in \mathcal{N}(i)} a_{ij}^s m_{ij}^s). \tag{10}$$

where $\mathcal{N}(i)$ is the set of node i along with its neighboring nodes. m_{ij}^s is the message sent from node i to j at the s-th message passing step. The connectivity for the propagation in GNN is given by $A_{1:i-1,1:i-1}$ with the last node (for which $A_{i,:}$ has not been generated yet) being fully connected. Note that message passing step is different from the generation step and we run multiple message passing steps per generation step in order to capture the structural dependency among nodes and edges. The f and g are two-layer MLPs. Since graphs are DAGs in our case rather than undirected ones as in (Liao et al., 2019), we add $\mathbf{1}_{ij}$ in Eq. (7), a one-hot vector for indicating the direction of the edge. We initialize the node representations h_i^0 (for $i < n$) as the corresponding one-hot encoded operation type vectors; h_n^0 is initialized to a special one-hot vector. Here u_i is an additional feature vector that helps distinguish i-th node from others. We found using one-hot-encoded incoming neighbors of i-th node and a positional encoding of the node index i work well in practice. We encourage readers to reference Fig. 4 for a detailed visualization of graph generation process.

Sampling. To sample from our generator, we first draw architectures following the standard ancestral sampling where each step involves drawing random samples from a categorical distribution and a mixture of Bernoulli distribution. At each step, this sampling process adds a new operator with a certain operation type and wire it to previously sampled operators.

3.1.2 Evaluator

Our design of generator and NAS pipeline do not rely on a specific choice of evaluator. Motivated by (Mnih et al., 2013), we use a replay buffer for storing the evaluated architectures. In our paper, based on specific datasets, we explore three types of evaluators, namely, oracle evaluator, supernet evaluator and benchmark evaluator, which are briefly introduced as follows.

Oracle evaluator. Given a sample from the generator, an oracle evaluator trains the corresponding network from scratch and tests it to get the validation performances. To reduce computation overhead, a common approach is to use early stopping (training with fewer epochs) as in (Tan et al., 2019; Tan & Le, 2019). In our experiment, we instead use a low-data evaluator similar to few-shot learning where we keep the same number of classes but use fewer samples per class to train.

SuperNet evaluator. Aiming at further reducing the amount of compute, this evaluator uses a weight-sharing strategy where each graph is a sub-graph of the supernet. We followed the single-path supernet setup used in (Pham et al., 2018) to compare with previous methods.

Benchmark evaluator. NAS benchmarks, *e.g.*, (Ying et al., 2019), provide accurate evaluation for architectures within the search space, which can be seen as oracle evaluators with full training budgets on target datasets.

3.2 Learning Method

Since we are dealing with discrete latent variables, *i.e.*, good architectures in our case, we train our NAS system using REINFORCE (Williams, 1992) algorithm with the control variate (a.k.a. baseline) to reduce the variance. In particular, the gradient of the loss or negative expected reward $\mathcal{L}$ w.r.t. the generator parameters ϕ is,

$$\nabla \mathcal{L}(\phi) = \mathbb{E}_{\mathbb{P}(\mathcal{G})} \left[-\frac{\partial \log \mathbb{P}(\mathcal{G})}{\partial \phi} \bar{R}(\mathcal{G}) \right], \tag{11}$$

where the reward $\bar{R}$ is standardized as $\bar{R}(\mathcal{G}) = (R(\mathcal{G}) - C)/\sigma$. Here the baseline C is the average reward of architectures in the replay buffer and σ is standard deviation of rewards in the replay buffer. The expectation in Eq. (11) is approximated by the Monte Carlo estimation. However, the score function (*i.e.*, the gradient of log likelihood w.r.t. parameters) in the above equation may numerically differ a lot for different architectures. For example, if a negative sample, *i.e.*, an architecture with a reward lower than the baseline, has a low probability $\mathbb{P}(\mathcal{G})$, it would highly likely to have an extremely large absolute score function value, thus leading to a negative reward with an extremely large magnitude. Therefore, in order to balance positive and negative rewards, we propose to use the reweighted log likelihood as follows,

$$\log \mathbb{P}(\mathcal{G}) = \beta \mathbf{1}_{\bar{R}(\mathcal{G}) \leq 0} \log(1 - \mathbb{P}(\mathcal{G})) + \mathbf{1}_{\bar{R}(\mathcal{G}) > 0} \log(\mathbb{P}(\mathcal{G})) \tag{12}$$

where β is a hyperparameter that controls the weighting between negative and positive rewards. $\mathbb{P}(\mathcal{G})$ is the original probability given by our generator.

Exploration vs. Exploitation Similar to many RL approaches, our NAS system faces the exploration vs. exploitation dilemma. We found that our NAS system may quickly collapse (*i.e.*, overly exploit) to a few good architectures due to the powerful graph generative model, thus losing the diversity and reducing to point estimate. Inspired by the epsilon greedy algorithm (Sutton & Barto, 2018) used in multi-armed bandit problems, we design a random explorer to encourage more exploration in the early stage. Specifically, at each search step, our generator samples from either itself or a prior graph distribution like the Watts–Strogatz model with a probability ϵ. As the search goes on, ϵ is gradually annealed to 0 so that the generator gradually has more exploitation over exploration.

4 EXPERIMENTS

In this section, we extensively investigate our NAS system on three different search spaces to verify its effectiveness. First, we adopt the challenging RandWire search space (Xie et al., 2019) which is significantly larger than common ones. To the best of our knowledge, we are the first to explore learning NAS systems in this space. Then we search on the ENAS Macro (Pham et al., 2018) and NAS-Bench-101(Ying et al., 2019) search spaces to further compare with previous literature. For all experiments, we set the number of mixture Bernoulli K to be 10, the number of message passing steps S to 7, hidden sizes of node representation h_i^s and message m_{ij}^s to 128. For RNN-based baselines, we follow the design in (Zoph et al., 2018) if not other specified.

4.1 RANDWIRE SEARCH SPACE ON TINY-IMAGENET

RandWire Search Space. Originally proposed in (Xie et al., 2019), a randomly wired neural network is a ResNet-like four-stage network with the cell graph $\mathcal{G}$ defines the connectivity of N convolution layers within each stage. At the end of each stage, the resolution is downsampled by 3×3 convolution with stride 2 whereas the number of channels is doubled. While following the RandWire small regime in (Xie et al., 2019), we share the cell graph G among last three stages for simplification. To keep the number of parameters roughly the same, we fix the node type to be separable 3×3 convolution. The number of nodes N within the cell graph $\mathcal{G}$ is set to 32 excluding the input and output nodes. This yields a search space of 2.1×10^{149} valid adjacency matrices, which is extremely large and renders the neural architecture search challenging. More details of the RandWire search space can be found in the Appendix B.1.

Methods	Cost (GPU Days)	Low Data (Search)		Full Data (Final)	
		Val Avg Acc	Std	Val Avg Acc	Std
ER-TopK	15.2	23.12	0.34	61.76	0.04
WS-TopK	15.6	22.39	0.91	62.24	0.34
ER-BEST	15.2	20.07	1.62	62.10	0.25
WS-BEST	15.6	18.68	1.41	62.16	0.92
RNN (Zoph et al., 2018)	17.2	18.46	**0.99**	61.73	0.77
Ours	16.7	**20.32**	1.12	**62.57**	**0.40**

Table 1: Comparisons on Tiny-ImageNet. The top and bottom blocks include random search and learning-to-search methods respectively. ER-TopK and WS-TopK refers to top (K=4) architectures found by all WS and ER models during search. ER-BEST and WS-BEST refer to the best ER and WS models found during search, *i.e.*, WS(k=4,p=0.75) and ER(p=0.1). Here Avg and Std of accuracies are computed based on 4 architectures sampled from generators.

Tiny-ImageNet w. Oracle Evaluator. To enable search on the RandWire space, we exploit the oracle evaluator on the Tiny-ImageNet dataset (Chrabaszcz et al., 2017). To save computation, we employ a low-data oracle evaluator where we sample $1/10$ of Tiny-ImageNet training set for training and use the rest for validation at each search step. Similar to the few-shot learning, we keep the number of classes unchanged but reduce the number of samples per class. After the search, we retrain our found architectures on the full training set and evaluate it on the original validation set. Specifically, for each model, the oracle evaluator trains for 300 epochs and uses the average validation accuracy of the last 3 epochs as the reward. Our total search budget is around 16 GPU days, which approximately amounts to 320 model evaluations, *e.g.*, 40 search steps and 8 samples evaluated per step. For random search baselines, we choose Erdős–Rényi (ER) and Watts–Strogatz (WS) models. Specifically, we first randomly draw hyperparameters from certain ranges, *i.e.*, $0.1 \leq$

$p \leq 0.5$ for ER and $(2, 0.2) \leq (k, p) \leq (6, 0.8)$ for WS, and then sample $\mathcal{G}$ from individual models. We set the reweight coefficient β to 0.05. For the random explorer, we choose WS model with the same hyperparameter range as a prior distribution and set $\epsilon = 0.6$ in the beginning and decay it by a factor of 0.2 every 10 search steps. We also find that gradually shrinking replay buffer size to keep 30% to 10% of top-performing architectures helps stabilize the training of the generator. At the search time, we reject samples that already appear in the replay buffer to avoid duplications. We apply the same setting to the RNN generator for a fair comparison.

Results. As shown in Table 1, we compare our NAS system with other random search methods and learning-to-search methods. We can see that our method outperforms the RNN-based generator and other random search methods in terms of average validation accuracy on the full dataset. Our generator also has a lower variance compared to the RNN-based one. Moreover, we observed that RNN based generator sometimes degenerates so that it frequently sample densely-connected graphs. This is probably due to the fact that RNN based generator does not effectively utilize the topology information. We can see that a high search reward (*i.e.*, a low-data validation accuracy) do not necessarily lead to better performances in full data training, which indicates a bias of the oracle evaluator within the low-data regime. Random search methods are prone to be biased as they select architectures solely based on the search reward. Nevertheless, our generator is less affected by the bias and able to learn a distribution of good architectures that perform well on full data training.

We also show results of the best architectures found within 4 samples in Table 2. It is clear that the best model found by our method outperforms those discovered by other methods by a considerable margin. Moreover, we scale up the best models (denoted as large) by adding more channels and one more computation stage (more details are in Appendix B.1). We can see that our searched architectures perform favorably against manually designed architectures like ResNet (He et al., 2016) and ResNeXt (Xie et al., 2017).

Model	Param (M)	Top1 / Top5 Acc	
Resnet18	11.68	$59.71_{\pm 0.09}$	$80.32_{\pm 0.10}$
Resnet50	25.56	$63.42_{\pm 0.30}$	$82.61_{\pm 0.15}$
Resnext50	27.56	$63.62_{\pm 0.07}$	$82.73_{\pm 0.08}$
FC	3.49	$60.82_{\pm 0.24}$	$82.29_{\pm 0.09}$
ER-Top1	3.23	$61.82_{\pm 0.09}$	$82.30_{\pm 0.18}$
RS-Top1	3.22	$62.55_{\pm 0.15}$	$82.64_{\pm 0.21}$
RNN	3.32	$62.29_{\pm 0.39}$	$82.16_{\pm 0.24}$
Ours	3.27	$\mathbf{63.23}_{\pm 0.18}$	$\mathbf{83.06}_{\pm 0.05}$
WS-Top1 Large	19.38	$63.84_{\pm 0.13}$	$82.61_{\pm 0.16}$
RNN Large	19.78	$63.69_{\pm 0.28}$	$82.74_{\pm 0.21}$
Ours Large	19.18	$\mathbf{64.45}_{\pm 0.26}$	$\mathbf{83.23}_{\pm 0.26}$

Table 2: Comparisons of best searched architectures (averaged over 3 runs per architecture) on Tiny ImageNet.

4.2 ENAS Macro Search Space on CIFAR10

ENAS Macro Search Space, originally proposed by Pham et al. (2018), is a search space which focuses on the entire network. $\mathcal{G}$ here defines the entire network with $N = 12$ nodes. The operation type (D=6)[1] per node is also searchable. $\mathcal{G}$ is guaranteed to contain a length-11 path, *i.e.*, $\forall i > 1$, $A_{i,i-1} = 1$. The goal is to search off-diagonal entries, *i.e.*, skip connections. This gives a search space of 1.6×10^{29} valid networks in total.

SuperNet Evaluator. For ENAS Macro search space, we experiment on CIFAR10 (Krizhevsky et al., 2009) dataset. For our generator, we use the ER model with $p = 0.4$ as our explorer, where ϵ decays from 1 to 0 in the first 100 search steps. For RNN based generator, we follow the setup in (Pham et al., 2018). We also adopt the weight-sharing mechanism in (Pham et al., 2018) to obtain a SuperNet evaluator that efficiently evaluates a model's performance. We use a budget of 300 search steps with around 100 architectures evaluated per step for all methods. After the search, we use a short-training of 100 epochs to evaluate the performances of 8 sampled architectures, after which top-4 performing ones are chosen for a 600-epoch full training. The best validation error rate among these 4 architectures is reported. For simplicity and a fair comparison, we do not use additional tricks (*e.g.*, adding entropy regularizer) in (Pham et al., 2018). More details are provided in Appendix C.

In Table 3, we compare the error rates and variances for different NAS methods. Note that this variance reflects the uncertainty of the distribution of architectures as it is computed based on sampled architectures. It is clear that our GraphPNAS achieves both lower error rates and lower variances compared to RNN based generator and is on par with the state-of-the-art NAS methods on other search spaces. We also see that the best architecture performance of our generator outperforms

[1] 1×1, 5×5 convolution, 1×1, 5×5 separable convolution, max pooling, avg pooling

Methods	Search Cost (days)	Params (M)	Best Error Rate	Top Samples Avg	Top Samples Std
Net Transform (Cai et al., 2018)	10	19.7	5.7	-	-
NAS (Zoph & Le, 2016)	22400	7.1	4.47	-	-
PNAS (Liu et al., 2018a)	225	3.2	3.41	-	-
Lemonade (Elsken et al., 2018a)	56	3.4	3.6	-	-
EPNAS-Macro (Perez-Rua et al., 2018)	1.2	38.8	4.01	-	-
RNN* (Pham et al., 2018)	0.9	19.64	4.18	4.47	0.282
RNN* Large	0.9	36.92	4.00	4.16	0.089
Ours	0.5	20.47	3.73	3.93	0.098
Ours Large	0.5	37.71	**3.55**	**3.62**	**0.050**

Table 3: Comparisons on CIFAR10 dataset. The bottom and top blocks include NAS methods with ENAS Macro and other search spaces respectively. *: our re-implementation. -: inapplicable.

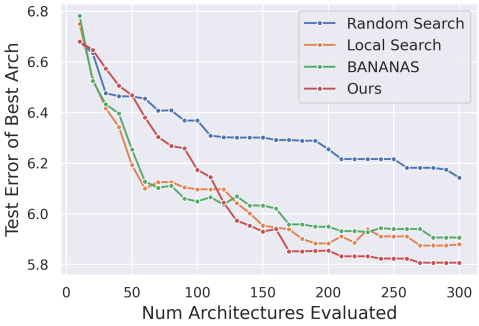


Figure 2: Performances (average over 10 runs) of best architectures vs. the number of architecture evaluations (search step).

Method	Avg Error	#Queries
RNN†	6.436	150
GCN Pred†	6.331	150
Evolution†	6.109	150
Ours	5.930 $\pm$0.143	150
Random Search	6.413 $\pm$0.422	300
Local Search	5.879 $\pm$0.371	300
BANANAS	5.906 $\pm$0.296	300
Ours	**5.807** $\pm$**0.072**	300

Table 4: Best model performances on NAS-Bench-101. † indicates numbers are taken from (White et al., 2019) Table 2.

RNN based generator by a significant margin. This verifies that our GraphPNAS is able to learn a distribution of well-performing neural architectures. Given that we only sample 8 architectures, the performances could be further improved with more computational budgets.

4.3 NAS-Bench-101

NAS-Bench-101 (Ying et al., 2019) is a tabulate benchmark containing 423K cell graphs, each of which is a DAG with up to 7 nodes and 9 edges including input and output nodes. We compare the performances of our GraphPNAS to open-source implementations of random search methods, local search methods, and BANANAS (White et al., 2019). The latter two are the best algorithms found by White et al. (2020) on NAS-Bench-101. For RNN based generator, GCN prediction and evolution methods, we use the score reported in (White et al., 2020). We give each NAS method the same budget of 300 queries and plot the curve of lowest test error as a function of the number of evaluated architectures. As shown in Fig. 2, our GraphPNAS is able to quickly find well-performing architectures. We also report the avg error rate over 10 runs in Table 4. Our GraphPNAS again outperforms RNN based generator by a significant margin and beats strong baselines like local search methods and BANANAS. Notably, our GraphPNAS has a much lower variance than other methods, thus being more stable across multiple runs.

5 Discussion & Conclusion

Qualitative Comparisons between RNN and GraphPNAS. In (You et al., 2020), the clustering coefficient and the average path length have been used to investigate distributions of graphs. Here we adopt the same metrics to visualize architectures (graphs) sampled by RNN based and our generators in RandWire experiments. Points in Fig. 3 refer to architectures sampled from both generators in the last 15 search steps where random explorers are disabled. The validation performances are color-coded. We can see that our GraphPNAS samples a set of graphs that have better validation accuracies while the ones of RNN generator have large variances in performances. Moreover, the

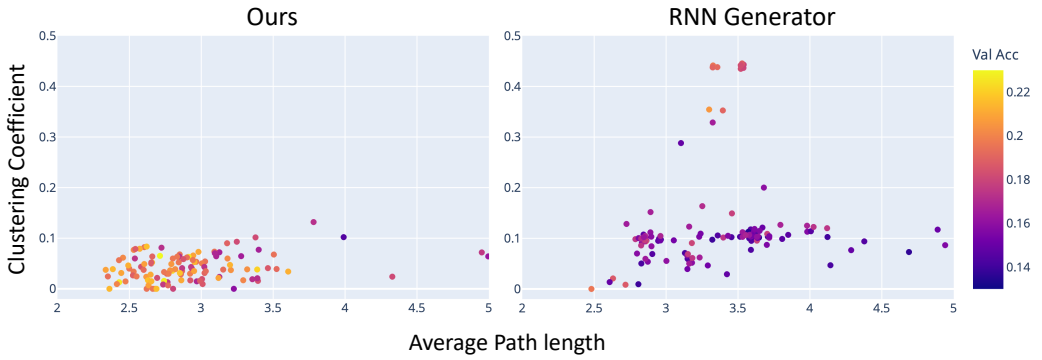


Figure 3: Visualization of architecutre explore sapce of GraphPNAS vs RNN. Each point in the figure denotes a model evaluation. Colors of each node denotes its validation accuracy returned by the low-data Oracle evaluator.

graphs in our case concentrate to those with smaller clustering coefficients, thus less likely being densely-connected. On the contrary, RNN generator tends to sample graphs that are more likely to be densely-connected. While RNN has been widely used for NAS, we show in our experiments that our graph generator consistently outperforms RNN over three search spaces on two different datasets. This is likely due to the fact that our graph generator better leverages graph topologies, thus being more expressive in learning the distribution of graphs.

Bias in Evaluator. In our experiments, we use SuperNet evaluator, low-data, and full-data Oracle evaluator to efficiently evaluate the model. From the computational efficiency perspective, one would prefer the SuperNet evaluator. However, it tends to give high rewards to those architectures used for training SuperNet. Although the low-data evaluator is more efficient than the full-data one, its reward is biased as discussed in Section 4.1. This bias is caused by the discrepancy between the data distributions in low-data and full-data regimes. We also tried to follow (Tan et al., 2019) to use early stopping to reduce the time cost of the full-data evaluator. However, we found that it assigns higher rewards to those shallow networks which converge much faster in the early stage of training. We show detailed results in Appendix B.4.

Search Space Design. The design of search space largely affects the performances of NAS methods. Our GraphPNAS successfully learns good architectures on the challenging RandWire search space. However, the search space is still limited as the cell graph across different stages is shared. A promising direction is to learn to generate graphs in a hierarchical fashion. For example, one can first generate a macro graph and then generate individual cell graphs (each cell is a node in the macro graph) conditioned on the macro graph. This will significantly enrich the search space by including the macro graph and untying cell graphs.

Conclusion. In this paper, we propose a GNN-based graph generator for NAS, called GraphPNAS. Our graph generator naturally captures topologies and dependencies of operations of well-performing neural architectures. It can be learned efficiently through reinforcement learning. We extensively study its performances on the challenging RandWire as well as two widely used search spaces. Experimental results show that our GraphPNAS consistently outperforms RNN-based generator on all datasets. Future works include exploring ensemble methods based on our GraphPNAS and hierarchical graph generation on even larger search spaces.

REFERENCES

Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. In *International Conference on Learning Representations ICLR*, 2017.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. In *International Conference on Learning Representations ICLR*, 2018.

Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286 (5439), 1999.

James Bergstra, Daniel Yamins, and David D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *Proceedings of the International Conference on Machine Learning, ICML*, 2013.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. In *International Conference on Learning Representations ICLR*, 2018.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490*, 2018.

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *International Conference on Machine Learning*, pp. 678–687. PMLR, 2018.

Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations ICLR*. OpenReview.net, 2019.

Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the cifar datasets. *arXiv preprint arXiv:1707.08819*, 2017.

Hang Chu, Daiqing Li, David Acuna, Amlan Kar, Maria Shugrina, Xinkai Wei, Ming-Yu Liu, Antonio Torralba, and Sanja Fidler. Neural turtle graphics for modeling city road layouts. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 4522–4530, 2019.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. *arXiv preprint arXiv:1804.09081*, 2018a.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018b.

Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *International Conference on Learning Representations ICLR*, 2019.

Paul Erdös and Alfréd Rényi. On random graphs i. *Publicationes Mathematicae Debrecen*, 6, 1959.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pp. 1050–1059, 2016.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

Aditya Grover, Aaron Zweig, and Stefano Ermon. Graphite: Iterative generative modeling of graphs. *arXiv preprint arXiv:1803.10459*, 2018.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social Networks*, 5(2), 1983.

Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Junction tree variational autoencoder for molecular graph generation. *arXiv preprint arXiv:1802.04364*, 2018.

Thomas N Kipf and Max Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

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

Wei Li, Shaogang Gong, and Xiatian Zhu. Neural graph embedding for neural architecture search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 4707–4714, 2020.

Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

Renjie Liao, Yujia Li, Yang Song, Shenlong Wang, Will Hamilton, David K Duvenaud, Raquel Urtasun, and Richard Zemel. Efficient graph generation with graph recurrent attention networks. In *Advances in Neural Information Processing Systems*, pp. 4255–4265, 2019.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 19–34, 2018a.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018b.

Jenny Liu, Aviral Kumar, Jimmy Ba, Jamie Kiros, and Kevin Swersky. Graph normalizing flows. In *Advances in Neural Information Processing Systems*, pp. 13578–13588, 2019.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

T Den Ottelander, Arkadiy Dushatskiy, Marco Virgolin, and Peter AN Bosman. Local search is a remarkably strong baseline for neural architecture search. *arXiv preprint arXiv:2004.08996*, 2020.

Juan-Manuel Perez-Rua, Moez Baccouche, and Stéphane Pateux. Efficient progressive neural architecture search. In *BMVC*, 2018.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *AAAI Conference on Artificial Intelligence*, 2019.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pp. 6105–6114. PMLR, 2019.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2820–2828, 2019.

Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684), 1998.

Colin White, Willie Neiswanger, and Yash Savani. Bananas: Bayesian optimization with neural architectures for neural architecture search. *arXiv preprint arXiv:1910.11858*, 2019.

Colin White, Sam Nolen, and Yash Savani. Local search is state of the art for neural architecture search benchmarks. *arXiv preprint arXiv:2005.02960*, 2020.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1492–1500, 2017.

Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1284–1293, 2019.

Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, pp. 7105–7114, 2019.

Jiaxuan You, Rex Ying, Xiang Ren, William L Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. *arXiv preprint arXiv:1802.08773*, 2018.

Jiaxuan You, Jure Leskovec, Kaiming He, and Saining Xie. Graph structure of neural networks. *arXiv preprint arXiv:2007.06559*, 2020.

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

Zhao Zhong, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu. Practical block-wise neural network architecture generation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2018.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 8697–8710, 2018.

A More details on Graph Generator

To more clearly illustrate the sampling process of our generator, we detailed probabilistic sampling process of our generator in Fig. 4.

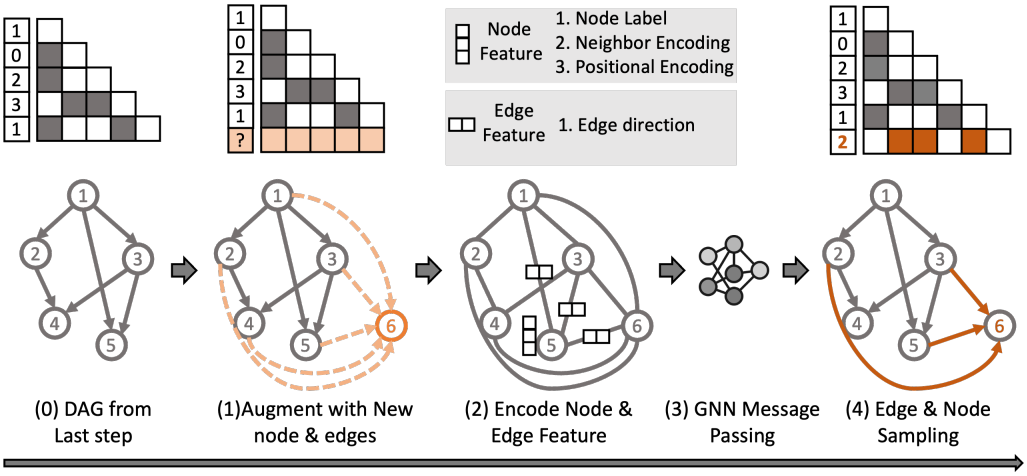


Figure 4: Detailed steps of auto-regressive generation with our graph generator.

B More details on the RandWire experiments

B.1 Details of RandWire search space

Here we provided more details on the RandWire search space shown in Table 5 and Fig. 5.

Stage	Output	Base		Large	
		Cell	Channels	Cell	Channels
conv_1	112×112	$\texttt{conv}_{3\times3}$	32	$\texttt{conv}_{3\times3}$	48
conv_2	56×56	$\texttt{conv}_{3\times3}$	64	$\texttt{conv}_{3\times3}$	96
conv_3	28×28	$\mathcal{G}$	64	$\mathcal{G}$	192
conv_4	14×14	$\mathcal{G}$	128	$\mathcal{G}$	288
				$\mathcal{G}$	384
conv_5	7×7	$\mathcal{G}$	256	$\mathcal{G}$	586
classifier	1×1	1×1 $\texttt{conv}_{1\times1}$, 1280-d global average pool, 200-d *fc*, softmax			

Table 5: Randwire search space with base and large settings. Base is the default setting for search while Large refers to the architecture of scaled up models in Table 2. `conv` denote a ReLU-SepConv-BN triplet . The input size is 224×224 pixels. The change of the output size implies a stride of 2 (omitted in table) in the convolutions that are placed at the end of each block. $\mathcal{G}$ is the shared cell graph that has $N = 32$ node.

B.2 Details for RandWire experiments

For experiment on Tiny-Imagenet, we resize image to 224×224 as showed in Table 5. We apply the basic data augmentation of horizontal random flip and random cropping with padding size 4. We provide detailed hyper-parameters for oracle evaluator training and learning for GraphPNAS in Table 6

B.3 Visualization of architectures from our generator

Here we visualize the top candidate architectures in Fig. 6

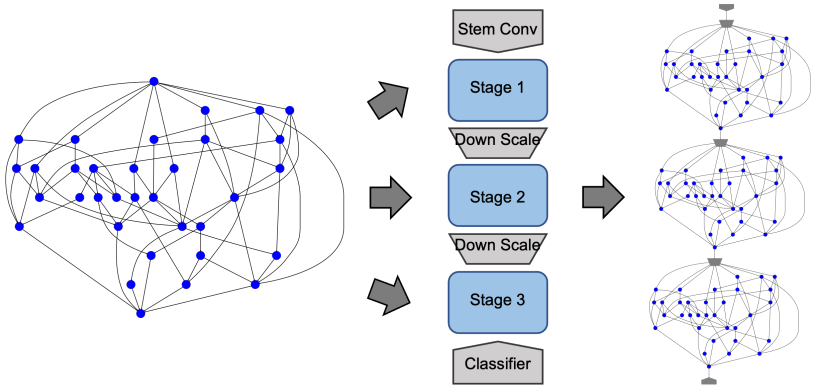


Figure 5: Visualization of RandWire search base space used in this paper. Different from (Xie et al., 2019), $\mathcal{G}$ here is shared across three stages.

Oracle Evaluator		Graph Controller	
batch size	256	graph batch size	16
optimizer	SGD	generator optimizer	Adam
learning rate	0.1	generator Learning rate	1e-4
learning rate deacy	consine lr decay	generator learning rate decay	none
weight decay	1e-4	generator weight decay	0.
grad clip	0.	generator gradient clip	1.0
training epochs	300	replay buffer fitting epochs	2000

Table 6: Hyperparameter setting for oracle evaluator and training our graph generator.

B.4 Bias for early stopping

As discussed in Section 5, using early stopping will lead to local minimal where the generator learns to generate shallow cell structure. We quantify this phenomenon in table 7, where we can see that with early stopping training, the generator will generate more shallow architectures with a shorter path from input to output. The corresponding average final validation accuracy also dropped by a large margin compared to the low data evaluator counter part.

evaluator	Final Val Acc	Average Path	Longest Path
early stopping	61.86	2.595	6.125
low data regime	62.57	3.046	8.75

Table 7: In the table we show ablation on the choice of oracle evaluator with our graph generator. The average Path and Longest path are computed as the average path length and longest path length from input to output over 8 samples from the corresponding generator.

C More details on ENAS Macro experiments

For ENAS Macro search space, we use a pytorch-based open source implementation[2] and follow the detailed parameters provided in (Pham et al., 2018) for RNN generator. Specifically, we follow (Pham et al., 2018) to train the SuperNet and update generator in an iterative style. At each search step, two sets of samples $\mathcal{G}_{\text{train}}$ and $\mathcal{G}_{\text{eval}}$ are sampled from the generator. $\mathcal{G}_{\text{train}}$ is used to learn the SuperNet's weights by back-propagating the training loss. The updated SuperNet is used for evaluating $\mathcal{G}_{\text{eval}}$, which is then used for updating the generator.

For our generator, we evaluate 100 architecture per step and update our generator every 5 epochs of SuperNet training. Instead of evaluating on a single batch, we reduce the number of models

[2]https://github.com/microsoft/nni/tree/v1.6/examples/nas/enas

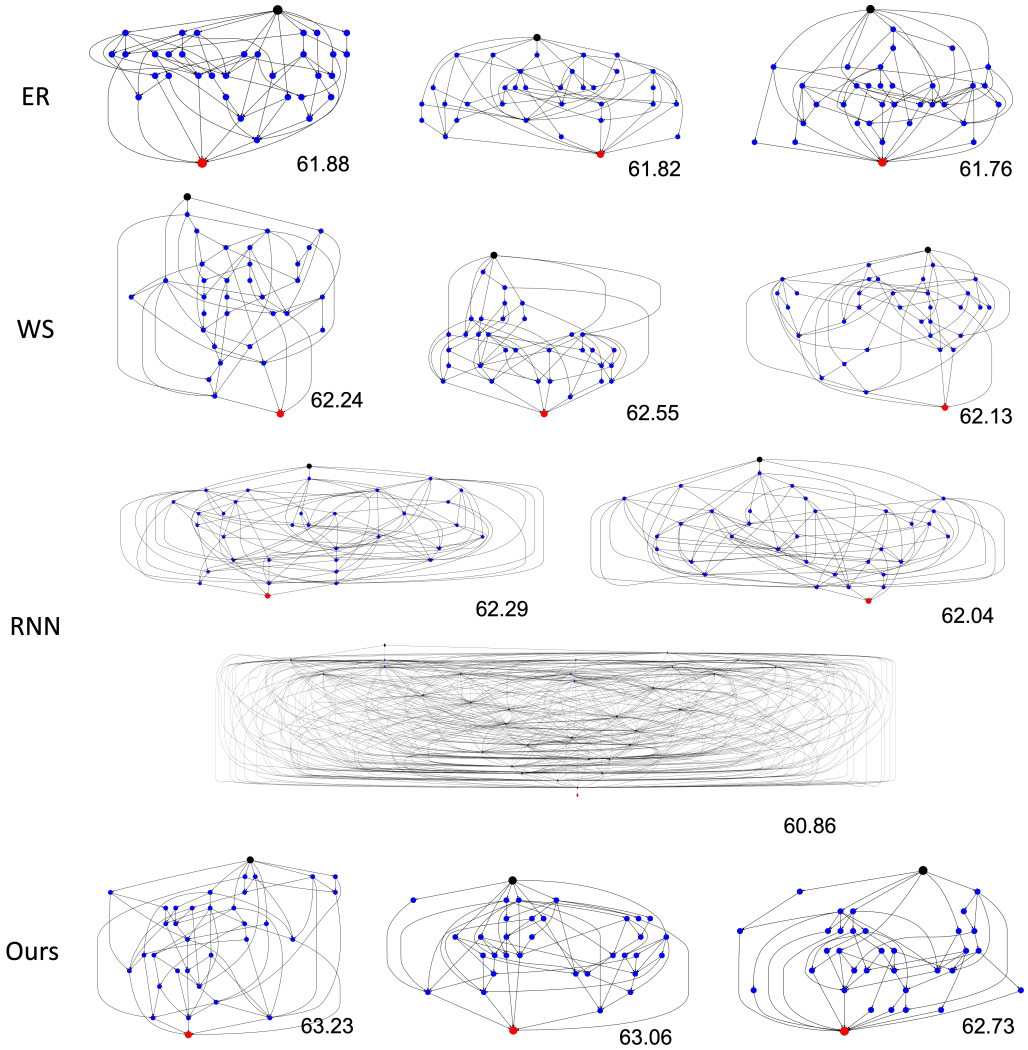
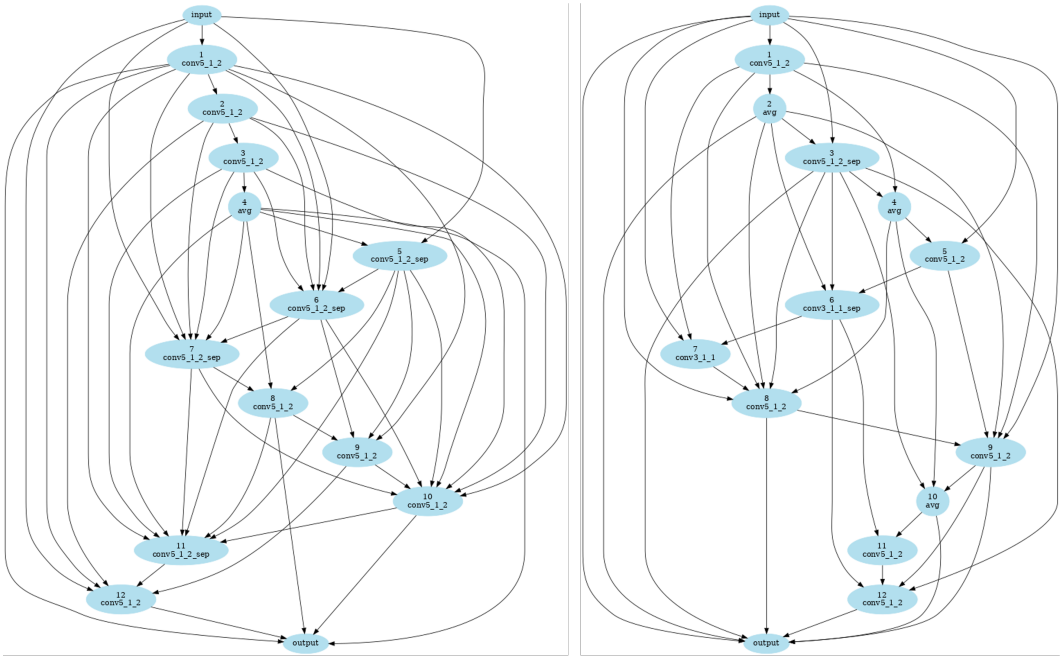


Figure 6: Visualization of Top 3 architectures sampled by each method. We observe that around 50% of samples from RNN generators are densely connected graphs or even fully connected graphs.



Ours Best: 3.55 Enas Best: 4.00

Figure 7: The best architecture found by GraphPNAS and RNN generator (Pham et al., 2018). Correspond to scores report in table 3. To get this architecture, we pre-evaluate 8 samples for both methods and select top-performing architecture.

evaluated per step and evaluate on the full test set. We found this stables the training of our generator while keeping evaluation costs the same. In the replay buffer, the top 20% of architectures is kept.

For training SuperNet and RNN generator, we follow the same hyper-parameter setting in (Pham et al., 2018) except learning rate decay policy is changed to cosine learning rate decay. For training our generator we use the same hyperparameter as in Table 6 with graph batch size changed to 32. For retraining the found the best architecture, we use a budget of 600 epoch training with a learning rate of 0.1, batch size 256, and weight decay 2e-4. We also apply a cutout with a probability of 0.4 to all the models when retraining the model.

C.1 Visualization of best architecture found

Here we visualize the best architecture found by GraphPNAS and RNN generator for Enas Macro search space in Fig. 7.

D More details on NAS-Bench-101

For sampling on NAS-Bench-101 (Ying et al., 2019), we first sample a 7-node DAG, then we remove any node that is not connected to the input or the output node. We reject samples that have more than 9 edges or don't have a path from input to output.

To train our generator on Nas-Bench-101, we use Erdős–Rényi with $p = 0.25$, ϵ is set to 1 in the beginning and decreased to 0 after 30 search steps. For the replay buffer, we keep the top 30 architectures. Our model is updated every 10 model evaluations, where we train 70 epochs on the replay buffer at each update time. The learning rate is set to 1e-3 for with a batch size of 2 on Nas-bench-101.