Graph Neural Architecture Search

Yang Gao [1,5], **Hong Yang** [2*], **Peng Zhang** [3], **Chuan Zhou** [4,5*] and **Yue Hu** [1,5]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]Centre for Artificial Intelligence, University of Technology Sydney, Australia
[3]Ant Financial Services Group, Hangzhou, China
[4]Academy of Mathematics and Systems Science, Chinese Academy of Sciences, Beijing, China
[5]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
gaoyang@iie.ac.cn, hong.yang@student.uts.edu.au, zhangpeng04@gmail.com,
zhouchuan@amss.ac.cn, huyue@iie.ac.cn

Abstract

Graph neural networks (GNNs) emerged recently as a powerful tool for analyzing non-Euclidean data such as social network data. Despite their success, the design of graph neural networks requires heavy manual work and domain knowledge. In this paper, we present a graph neural architecture search method (*GraphNAS*) that enables automatic design of the best graph neural architecture based on reinforcement learning. Specifically, GraphNAS uses a recurrent network to generate variable-length strings that describe the architectures of graph neural networks, and trains the recurrent network with policy gradient to maximize the expected accuracy of the generated architectures on a validation data set. Furthermore, to improve the search efficiency of GraphNAS on big networks, GraphNAS restricts the search space from an entire architecture space to a sequential concatenation of the best search results built on each single architecture layer. Experiments on real-world datasets demonstrate that GraphNAS can design a novel network architecture that rivals the best human-invented architecture in terms of validation set accuracy. Moreover, in a transfer learning task we observe that graph neural architectures designed by GraphNAS, when transferred to new datasets, still gain improvement in terms of prediction accuracy.

1 Introduction

Graph neural networks (GNNs) emerged recently as a powerful tool for analyzing non-Euclidean data such as social network data. The applications of GNNs span over online recommendation [Wu *et al.*, 2019b], traffic forecasting [Yu *et al.*, 2018] and action recognition [Yan *et al.*, 2018]. The basic idea of GNNs is to propagate feature information between neighboring nodes so that nodes can learn feature representations by using the locally connected graph architecture information. Popular GNNs include but not limited to GCN [Kipf and Welling, 2017], GraphSAGE [Hamilton *et al.*, 2017], GAT [Velickovic *et al.*, 2017] and APPNP [Klicpera *et al.*, 2019].

Despite the success of GNNs, the design of graph neural architectures requires both heavy manual work and domain knowledge. Similar to CNNs that contain many manual parameters such as the size of filters and the type of pooling layers, the results of GNNs heavily rely on the graph neural architectures including the receptive fields, message functions and aggregation functions.

Reinforcement learning has been successfully used to automatically design neural architectures for CNNs and RNNs. The pioneer model NAS [Zoph and Le, 2016] uses a recurrent network as the controller to generate network descriptions of CNNs and RNNs which are referred to as child networks, and then uses validation results of the child networks as reward of the controller to maximize the expected accuracy of the generated architectures of the CNNs and RNNs. According to their experiment results, the NAS search algorithm can improve CNNs and RNNs by a percentage of 0.09 on CIFAR-10 and 3.6 perplexity on the Penn Treebank dataset. Several new neural architecture search algorithms are proposed to improve the efficiency and accuracy of NAS, such as ENAS [Pham *et al.*, 2018] and ProxylessNAS [Cai *et al.*, 2019]. The promising results of using NAS to find the best neural architectures for CNNs and RNNs motivate to use reinforcement learning to find the best graph neural architectures for GNNs.

In this paper, we present a new *graph neural architecture search* method (*GraphNAS*) which can automatically design the best graph neural architecture using reinforcement learning. Specifically, we design a new search space for reinforcement learning that covers the operators from the state-of-the-art GNNs, such as GCN, GraphSAGE and GAT. Based on the search space, we use a RNN model as the controller to generate variable-length strings that describe the architectures of graph neural networks, and trains the recurrent network with policy gradient to maximize the expected accuracy of the generated architectures on a validation data set. To analyze big networks, we assume the layers of network architectures are independent and restrict the search space to each single layer. Then, the best results found from each single layer are sequentially concatenated to describe the entire architecture. Experiment results show that GraphNAS always obtains

*Corresponding authors

better results than the state-of-the-art methods. Experiment results also show that GraphNAS obtains performance improvement in a transfer learning task.

The contributions of the paper are summarized as follows:

- This is the first effort to study the challenging problem of using reinforcement learning to design the best graph neural architecture.
- We present a new model *GraphNAS* to enable the automatic search of the best graph neural architecture, where a new search space is designed that covers the operators from the state-of-the-art GNNs, and a policy gradient algorithm is used to iteratively solve the problem.
- We test *GraphNAS* on real-world datasets. The results show that our method is capable of designing graph neural architectures that outperform the best human-invented architectures in terms of validate set accuracy. We have released the python codes on Github[1] for comparison.

2 Related Work

2.1 Neural Architecture Search (NAS)

NAS has been popularly used to design convolutional architectures [Zoph and Le, 2016; Pham *et al.*, 2018; Xie *et al.*, 2019; Bello *et al.*, 2017; Liu *et al.*, 2018a; Cai *et al.*, 2019]. The basic idea of NAS is to use reinforcement learning to find the best neural architectures. Specifically, NAS uses a recurrent network to generate architecture descriptions of CNNs and RNNs. Based on NAS, evolution-based NAS [Real *et al.*, 2018] is proposed to use evolution algorithms to simultaneously optimize topology alongside with parameters. ENAS [Pham *et al.*, 2018] allows the sharing of parameters among child models, which enables the search speed 1000 times faster than the standard NAS and obtains a new convolution architecture in 0.45 GPU days. DARTS [Liu *et al.*, 2018a] formulates the task in a differentiable manner which shortens the search of high-performance convolution architectures within four GPU days. Following DARTS [Liu *et al.*, 2018a], GDAS [Dong and Yang, 2019] enables the search speed in four GPU hours, and Proxyless NAS [Cai *et al.*, 2019] claims that the search process can directly operate on the large-scale target tasks and the target hardware platforms. Due to NAS-based search algorithms achieve promising results for designing new architectures for CNNs and RNNs, we extend NAS to design graph neural architectures for GNNs in this paper.

2.2 Graph Neural Networks (GNNs)

GNNs are firstly discussed in the work [Gori *et al.*, 2005]. Convolutions of GNNs can be categorized into two groups, *spectral-based* [Kipf and Welling, 2017; Defferrard *et al.*, 2016; Bianchi *et al.*, 2019] and *spatial-based* [Velickovic *et al.*, 2017; Hamilton *et al.*, 2017; Niepert *et al.*, 2016; You *et al.*, 2019]. Spectral-based convolutions usually handle an entire graph, which is difficult to parallel and hardly scale to big graphs. In contrast, spatial-based convolutions aggregate feature information between neighboring nodes. Spatial-based graph neural architectures mainly consist of three types

[1]https://github.com/GraphNAS/GraphNAS

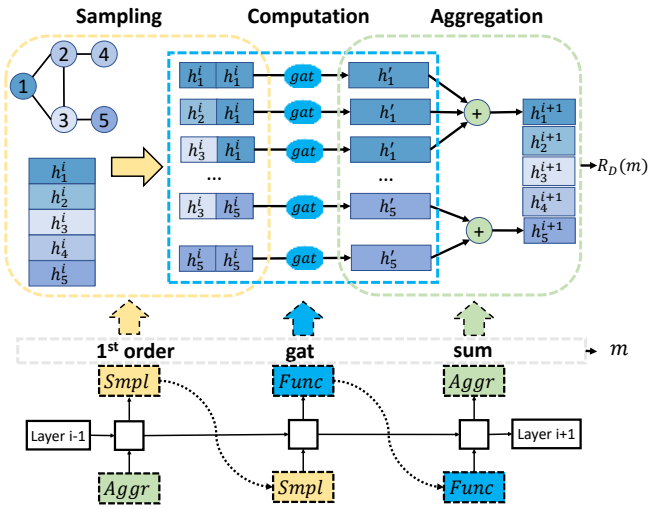

Figure 1: An illustration of GraphNAS. A recurrent network (Controller RNN) generates descriptions of graph neural architectures (child model GNNs). Once an architecture m is generated by the controller, GraphNAS trains m on a given graph G and test m on a validate set D. The validation result $R_D(m)$ is taken as the reward of the recurrent network.

of operators, i.e., neighbor sampling, message computation and information aggregation. Each layer of the architecture includes the combination of these operators. In this paper, we use reinforcement learning to search the best combination of the operators, instead of manually setting them as in the existing work.

3 Methods

In this section, we first formulate the problem of graph neural architecture search with reinforcement learning. Then, we introduce a new search space that covers the operators of the state-of-the-art GNNs. Next, we discuss the search algorithm based on policy gradient. At the last part, we discuss how to extend the search algorithm to big networks.

3.1 Problem Formulation

Given a search space of a graph neural architecture $\mathcal{M}$, and a validation set D, we aim to find the best architecture $m^* \in \mathcal{M}$ that maximizes the expected accuracy $\mathbb{E}[\mathcal{R}_D(m)]$ on the validate set D, i.e.,

$$m^* = \arg\max_{m \in \mathcal{M}} \ \mathbb{E}[R_D(m)]. \quad (1)$$

Figure 1 shows the reinforcement learning framework used to solve Eq.(1) by continuously sampling architectures $m \in \mathcal{M}$ and evaluating the accuracy (reward) R on the validation set D. First, the recurrent network generates a network description $m \in \mathcal{M}$ that corresponds to a GNN model. Then, the generated model m is trained on a given graph G and tested on the validate set D. The test result is taken as a reward signal R to update the reinforcement learning.

3.2 Search Space

As shown in Figure 1, we use a controller to generate the descriptions (architectures) of GNNs. The controller is a re-

current neural network with a search space. Similar to the search space of CNNs, each layer of GNNs associates with a search space of the following operators:

1. *Neighbor sampling operator* $Smpl$ [Hamilton *et al.*, 2017]. This operator selects the receptive field $N(v)$ for a target node v. For example, GraphSAGE [Hamilton *et al.*, 2017] uses sampling to obtain a fixed size of neighbors to handle large graphs.
2. *Message computation operator* $Func$ [You *et al.*, 2019]. This operator computes the feature information for each node u in the receptive field $N(v)$. Specifically, $Func(h_u, h_v)$ can be calculated by $e_{uv}Merg(h_u, h_v)$, where h_u and h_v are the features of nodes u and v respectively, and e_{uv}, as shown in Table 2, is the correlation coefficient [Velickovic *et al.*, 2017; Liu *et al.*, 2018b; Kipf and Welling, 2017; You *et al.*, 2019] between nodes u and v. The $Merg$ operator merges information of nodes u and v, such as the *CONCAT* used in [You *et al.*, 2019; Hamilton *et al.*, 2017].
3. *Message aggregation operator* $Aggr$ [Hamilton *et al.*, 2017]. This operator aggregates information from the receptive field $N(v)$, which is similar to the pooling operators in CNNs. Any permutation invariant operators, such as *mean*, *max*, *sum* and *mlp* [Xu *et al.*, 2018] can be used, and non-linear transformations are applied before and/or after the aggregation to achieve accurate expressive power [Zaheer *et al.*, 2017].
4. *Multi-head and readout operator* $Read$ [Velickovic *et al.*, 2017]. This operator is often used to stabilize the learning process of message computation operators. Similar to the attention mechanism in the work [Vaswani *et al.*, 2017], the multi-head mechanism repeats message computation operator $Func$ for k times with different initializations of $Func$. Readout operator $Read$ usually includes concatenation and averaging.

Table 1 summarizes the possible values of the above operators. Note that the values are collected from the state-of-the-art GNNs, such as GCN, GAT and GraphSAGE. Besides the above operators, we also add extra three operators that are popularly used in CNNs, i.e., the activation function σ, the number of multi-head k and the output dimension d.

We introduce an **example** of a simple graph neural architecture constructed with the operators given in Table 1. Consider a single layer of GAT with eight heads, 16 hidden units, and an activation function of elu, we describe the architecture by the operators as follows,

$$[first\_order, gat, sum, concat, 8, 16, elu],$$

where the first element $first\_order$ is an instance of the neighbor sampling operator $Smpl$, the second element gat is the message computation operator $Func = e^{gat}_{uv}h_u$, the third element sum is the message aggregation operator $Aggr$, the fourth element denotes that the architecture has eight heads, the fifth element denotes that the architecture has 16 hidden units, and the last element denotes $\sigma = elu$.

For an architecture with L layers, we concatenate the lists of operators built from each layer and generate an architecture

Operators	**Values**
$Smpl$	$first\_order$
$Func$	$e_{uv}h_u$
$Aggr$	sum, $mean$, max, mlp
$Read$	avg, for the last layer $concat$, otherwise
activate function σ	$sigmoid$, $tanh$, $relu$, $identity$, $softplus$, $leaky\_relu$, $relu6$, elu
multi-head k	1, 2, 4, 6, 8, 16
output dimension d	8, 16, 32, 64, 128, 256, 512

Table 1: Operators of search space $\mathcal{M}$

e_{uv}	**Values**
const	$e^{con}_{uv} = 1$
gcn	$e^{gcn}_{uv} = 1/\sqrt{d_u d_v}$
gat	$e^{gat}_{uv} = leaky\_relu((W_l * h_u + W_r * h_v))$
sym-gat	$e^{sym}_{uv} = e^{gat}_{vu} + e^{gat}_{uv}$
cos	$e^{cos}_{uv} =< W_l * h_u, W_r * h_v >$
linear	$e^{lin}_{uv} = tanh(sum(W_l * h_u))$
gene-linear	$e^{gan}_{uv} = W_a * tanh(W_l * h_u + W_r * h_v)$

Table 2: Correlation coefficients

of the entire L layers. For example, consider a GNN with two layers. The first layer consists of GCN with 16 hidden units and an activation function $relu$. The second layer consists of GAT with eight heads, 16 hidden units and an activation function elu. Then, the architecture is described by concatenating the operators of the two layers, which formulates a longer list of operators as follows:

$$[\ first\_order, gcn, sum, concat, 1, 16, relu,$$
$$first\_order, gat, sum, avg, 8, 16, elu\ ].$$

Because all the operators in the search space $\mathcal{M}$ given in Table 1 are independent, there will be 9408^L combinations in $\mathcal{M}$, where L is the number of layers. As the search space is too large, we set L to be only two layers in the experiments, which reduces the space to 8.8×10^7. If the architecture is deeper than two layers, a possible solution is to set a time-sensitive parameter to control the total search time over the search space.

3.3 Search Algorithm

A neural architecture that a controller predicts is a list of operators with length T, denoted by $m_{1:T}$, where each operator m_i $(1 \leq i \leq T)$ is sampled from the search space $\mathcal{M}$. We use an RNN model parameterized by θ to generate the operators, as shown at lines 3 to 8 in Algorithm 1.

In order to maximize the objective function given in Eq.(1), we use a policy gradient algorithm to update parameters θ, so that the controller generates better architectures over time. After the controller generates a list of operators $m_{1:T}$, we build a model m which returns an accuracy of $R_D(m)$ on D. We use the accuracy as a reward signal to train the controller. Since the reward signal R is non-differentiable, we iteratively update θ using REINFORCE [Williams, 1992] as follows,

Algorithm 1 GraphNAS search algorithm

Require: search space $\mathcal{M}$; controller RNN parameterized by θ; graph G; validation set D; # of operators T; # of models K; # of repeats N; # of samples S
Ensure: the best architecture m^*
// policy gradient
while the number of samples S is not met **do**
  $h_0 = \mathbf{0}$ // initial hidden state of RNN
  // sample operators $m_{1:T}$
  for $i = 1, ..., T$ **do**
    $x_i \leftarrow h_{i-1}$ // input of RNN
    $h_i \leftarrow RNN_\theta(x_i, h_{i-1})$;
    $P_i \leftarrow Softmax(h_{i-1})$;
    Sample m_i from $\mathcal{M}$ under P_i
  end for
  Design architecture m using operators $m_{1:T}$
  Train m on a given graph G
  Calcuate reward $R_D(m)$ on validation set D
  Update parameter θ *w.r.t.* $R_D(m)$
end while
// model selection
Select top K models *w.r.t.* validation accuracy
Re-train the K models for N times, select the best m^*
return m^*

$$\nabla_\theta \mathbb{E}_{P(m_{1:T};\theta)}[R] \tag{2}$$
$$= \sum_{t=1}^{T} \mathbb{E}_{P(m_{1:T};\theta)}[\nabla_\theta log P(m_t|m_{t-1:1};\theta)(R-b)],$$

where b is an exponential moving average of the previous architecture rewards. The training of a child model m is independent of the training of the controller. We choose cross-entropy loss function when training m. Considering the deviation of the validation accuracy, we select the top K models as candidates and repeatedly train them for N times to reduce variance. The algorithm is summarized in Algorithm 1.

3.4 Discussions

What will happen when an architecture goes deeper? Intuitively, the search space of GraphNAS grows exponentially with the number of layers. When the architecture goes deeper, constructing architectures by concatenating the operators will lead to an explosion of the search space. To solve the problem, we enforce three constraints to avoid exponential growth of the search space. First, we assume the layers are independent and design each layer independently. Second, we use domain knowledge of existing GNN architectures and reduce the number of combinations of the operators given in Table 1. Third, We allow the multi-head mechanism using different message computation operators.

In a deeper architecture, how to construct each layer efficiently? Figure 2 gives an example of constructing a single layer by GraphNAS. The layer can be represented as a DAG consisting of two input states O_1 and O_2, two intermediate states O_3 and O_4, and one output state O_5. Each state is a node in the DAG. Specifically, the intermediate states O_3 and O_4 are processed by a message computation operator $Func$, a sample operator $Smpl$, and an aggregation operator $Aggr$ from its previous state. This procedure of O_4 can be formulated as $O_4 = GAT(O_2)$, where GAT is used to represent the combination of $Smpl$, $Func$ and $Aggr$ used in GAT. The output state O_5 is processed by a $Read$ operator from all the intermediate states O_3 and O_4.

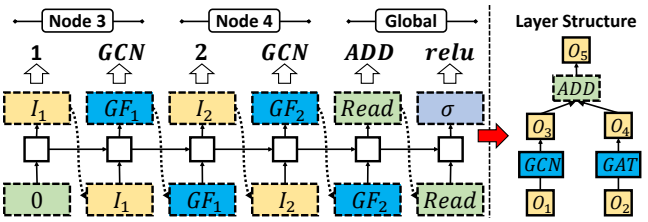

Figure 2: An illustration of GraphNAS constructing a single GNN layer at the right-hand side. The layer has two input states O_1 and O_2, two intermediate states O_3 and O_4, and an output state O_5. The controller at the left-hand side samples O_2 from $\{O_1, O_2, O_3\}$ and take O_2 as the input of O_4, and then samples GAT for processing O_2. The output state $O_5 = relu(O_3 + O_4)$ collects information from O_3 and O_4, and the controller assigns a readout operator add and an activation operator $relu$ for O_5. As a result, this layer can be described as a list of operators: [1,gcn, 2, gat, add, $relu$].

Based on the example, we can generalize the procedure of constructing each layer as follows,

$$O_{out} = \sigma(Read(O_i|3 \leqslant i \leqslant B+2)), \tag{3}$$

where B is the number of computation nodes. A controller needs to assign a previous state $O_j \in \{O_j|j < i\}$, a process operator for all the intermediate states $\{O_i|3 \leqslant i \leqslant B+2\}$, a read-out operator $Read$, and an activation function σ for the output state, as shown at the left-hand side of Figure 2. We restrict operator $Read$ to be only add, $multiply$ and $concat$. The operators will cover the following 12 choices:

- identity
- zeroize
- 8 head GAT
- 6 head GAT
- 4 head GAT
- 2 head GAT
- 1 head GAT
- GCN
- Chebnet
- Mean Sage
- ARMA
- SGC

where GAT stands for the combination of $Smpl$, $Func$ and $Aggr$ used in [Velickovic *et al.*, 2017], GCN for [Kipf and Welling, 2017], Chebnet for [Defferrard *et al.*, 2016], Mean Sage for GraphSage[Hamilton *et al.*, 2017] with the mean aggregator, ARMA for [Bianchi *et al.*, 2019], and SGC for [Wu *et al.*, 2019a].

4 Experiments

Datasets. We use three popular citation networks, i.e., Cora, Citeseer and Pubmed, as the testbed. To test the capability of transferring the architectures designed by GraphNAS, we use the co-author datasets of MS-CS and MS-Physics, and the product networks of Amazon Computers and Amazon Photos [Shchur *et al.*, 2018].

	Cora			Citeseer			Pubmed		
	semi	sup	rand	semi	sup	rand	semi	sup	rand
GCN	81.4±0.5	90.2±0.0	88.3±1.3	70.9±0.5	80.0±0.3	77.2±1.7	79.0±0.4	87.8±0.2	88.1±1.4
GAT	83.0±0.7	89.5±0.3	87.2±1.1	72.5±0.7	78.6±0.3	77.1±1.3	79.0±0.3	86.5±0.6	87.8±1.4
ARMA	82.8±0.6	89.8±0.1	88.2±1.0	72.3±1.1	79.9±0.6	76.7±1.5	78.8±0.3	88.1±0.2	88.7±1.0
APPNP	83.3±0.1	90.4±0.2	87.5±1.4	71.8±0.4	79.2±0.4	77.3±1.6	80.2±0.2	87.4±0.3	88.2±1.1
HGCN	79.8±1.2	89.7±0.4	87.7±1.1	70.0±1.3	79.2±0.5	76.9±1.3	78.4±0.6	88.0±0.5	88.0±1.6
GraphNAS-R	83.3±0.4	90.0±0.3	88.5±1.0	73.4±0.4	81.1±0.3	76.5±1.3	79.0±0.4	90.7±0.6	90.3±0.8
GraphNAS-S	81.4±0.6	90.1±0.3	88.5±1.0	71.7±0.6	79.6±0.5	77.5±2.3	79.5±0.5	88.5±0.2	88.5±1.1
GraphNAS	**83.7±0.4**	**90.6±0.3**	**88.9±1.2**	**73.5±0.3**	**81.2±0.5**	**77.6±1.5**	**80.5±0.3**	**91.2±0.3**	**91.1±1.0**

Table 3: Node classification results *w.r.t.* accuracy, where "semi" stands for semi-supervised learning experiments, "sup" for supervised learning experiments and "rand" for supervised learning experiments with randomly split data.

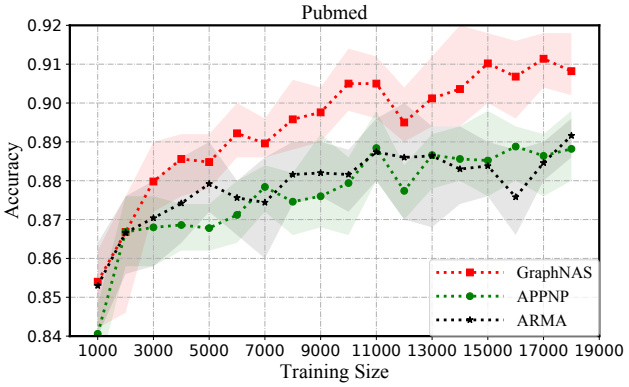

Figure 3: Comparisons *w.r.t.* the size of training data. When the training size exceeds 3,000, the model designed by GraphNAS always outperforms ARMA and APPNP.

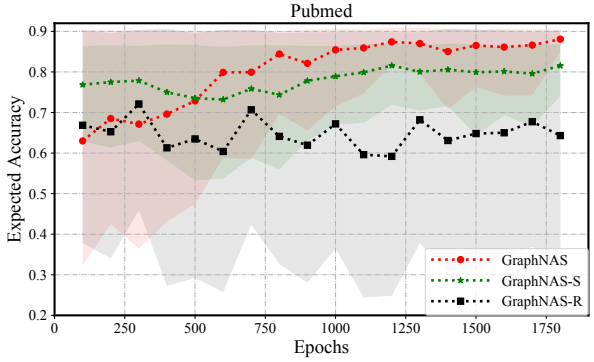

Figure 4: Comparisons *w.r.t.* the number of search epochs on Pubmed. The expected accuracy of the architecture designed by GraphNAS raises with the search epochs. GraphNAS outperforms Simple-NAS after 500 epochs.

Measures. We compare architectures designed by GraphNAS with the state-of-the-art GNNs, such as GCN, GAT, ARMA, APPNP [Klicpera *et al.*, 2019] and HGCN [Hu *et al.*, 2019], on node classification tasks. We use accuracy as the measure for comparison. All the results are the average scores for 100 runs with different random seeds.

4.1 Parameter Settings

The GNN architectures used in GraphNAS are implemented by PYG [Fey and Lenssen, 2019].

Hyper-parameters of the controller: The controller is a one-layer LSTM with 100 hidden units. It is trained with the ADAM optimizer with a learning rate of 0.00035. The weights of the controller are initialized uniformly between -0.1 and 0.1. To prevent premature convergence, we also use a tanh of 2.5 and a temperature of 5.0 for the sampling logits [Bello *et al.*, 2017], and add the controller's sample entropy to the reward, weighted by 0.0001. After GraphNAS searches $S = 2000$ architectures, we collect the top $K = 5$ architectures that achieve the best validation accuracy. Then we train those model for $N = 20$ times to choose the best models. Each GNN designed by GraphNAS contains $L = 2$ layers for fair comparisons.

Hyper-parameters of GNNs: Once the controller samples an architecture, a child model is constructed and trained for 300 epochs. We apply the L2 regularization with λ = 0.0005, dropout probability p = 0.6, and learning rate $lr = 0.005$ as the default parameters. To achieve the best results, the hyper-parameters of the GNN models are searched over the following search space:

- Hidden size: [8, 16, 32, 64, 128, 256, 512]
- Learning rate: [1e-2, 1e-3, 1e-4, 5e-3, 5e-4]
- Dropout: [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
- L_2 regularization strength: [0, 1e-3, 1e-4, 1e-5, 5e-5, 5e-4]

For GraphNAS, the hyper-parameters are predicted by the controller. While for the other models, the hyper-parameters is optimized by hyperopt[2].

4.2 Results of Node Classification

To validate the performance of node classification, we compare the models designed by GraphNAS with the benchmark GNNs in semi-supervised task and supervised task. The performance in terms of accuracy is shown in Table 3. The best performance of each column is highlighted with boldface.

In the **semi-supervised learning task**, the datasets follow the settings of [Kipf and Welling, 2017]. During training, only 20 labels per class are used for each citation network, 500 nodes in total for validation and 1,000 nodes for testing.

[2]https://github.com/hyperopt/hyperopt

Model	CS	Physics	Computers	Photo
GCN	95.5±0.3	98.3±0.2	88.0±0.6	95.4±0.3
GAT	95.5±0.3	98.1±0.2	89.1±0.6	95.6±0.3
ARMA	95.4±0.2	**98.5±0.1**	86.1±1.0	94.8±0.8
APPNP	95.6±0.2	**98.5±0.1**	89.8±0.4	95.8±0.3
GraphNAS	**97.1±0.2**	**98.5±0.2**	**92.0±0.4**	**96.5±0.4**

Table 4: Transferring architectures designed by GraphNAS on the citation networks to the other four datasets

In the **supervised learning task**, 500 nodes in each dataset are selected as the validation set and 500 nodes are selected as the test set. The rest of nodes are selected from the graph as training data. The architectures designed by GraphNAS are showed in Figure 5.

The work [Shchur *et al.*, 2018] claims that data split has an impact on the performance. Thus, we randomly split the datasets for 100 times to verify the models designed in the supervised learning task. Each split contains 500 nodes for evaluation, 500 nodes for test, and the rest for training.

From Table 3, we observe that the models designed by GraphNAS rival the best human-invented architectures. The model designed by GraphNAS always outperforms the state-of-the-art. The average improvement of GraphNAS is reasonable when the training data is small, such as on Cora and Citeseer. Moreover, GraphNAS performs significantly better than others when the number of training data is large, such as on Pubmed. Overall, the improvement of the model designed by GraphNAS on Pubmed is 3.1% in terms of accuracy.

The size of training data. In this part, we test the performance with respect to the number of training data. We split the training data ranging from 1,000 to 18,000. The results are shown in Figure 3. From the results, we observe that the model designed by GraphNAS always outperforms ARMA and APPNP when the number of training nodes exceeds 3,000.

4.3 Results of Transfer Learning

In this part, we apply the architectures discovered by GraphNAS on the citation networks to supervised node classification on different data sets, such as the coauthor networks MS-CS and MS-Physics, and the product networks of Amazon Computers and Amazon Photo. As shown in Table 4, the models designed by GraphNAS can obtain competitive results when transferred to new datasets.

4.4 Variants of GraphNAS

Based on the original GraphNAS, we construct two variants: *GraphNAS-R* that randomly selects graph neural network architectures from a given search space, and *GraphNAS-S* that simply searches for an entire graph neural architecture where each layer containing only one computational node and take the last layer as input.

Tables 3 presents the results of the architectures designed by the two variants. We observe that the combination of existing graph neural operators may achieve better results on bigger networks. The comparisons between *GraphNAS-R* and GraphNAS show that the search space we defined is reasonable, and the search algorithm converges to the optimal solution.

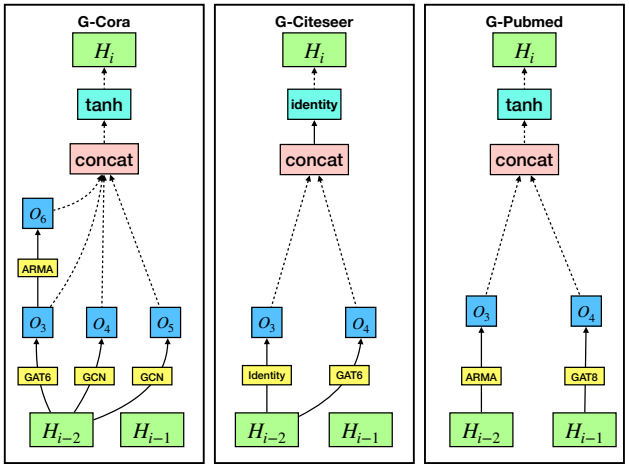

Figure 5: An example of the graph neural architectures designed by GraphNAS on the supervised learning task. The architecture *G-Cora* designed by GraphNAS on Cora is $[1, gat\_6, 1, gcn, 1, gcn, 3, arma, tanh, concat]$, the architecture *G-Citeseer* designed by GraphNAS on Citeseer is $[1, identity, 1, gat\_6, identity, concat]$, and the architecture *G-Pubmed* designed by GraphNAS on Pubmed is $[2, gat\_8, 1, arma, tanh, concat]$.

In order to verify the effectiveness of the search algorithm with reinforcement learning, we compare the expected validation accuracy over time during the architecture search, as shown in Figure 4.

In terms of computation time, for supervised learning the search time on Pubmed is 12 GPU hours, Cora and Citeseer 6 GPU hours. For semi-supervised learning, the search time on Pubmed is 9 GPU hours, Cora and Citeseer 2 GPU hours. The experiments are tested on a single NVIDIA 1080Ti.

5 Conclusions

In this paper, we study the challenging problem of graph neural architecture search using reinforcement learning. We present a new model *GraphNAS* which can automatically design the best graph neural architectures. A new search space is designed to include the operators from the state-of-the-art GNNs, and a policy gradient algorithm is used to iteratively solve the learning. Experiment results on real-world datasets show that GraphNAS can design a novel network architecture that rivals the best human-invented architecture in terms of validation set accuracy. Moreover, We release the python codes on Github for comparison.

Acknowledgements

This work was supported in part by the National Key Research and Development Program of China (No. 2017YFB0803300), the NSFC (No. 61872360), the Youth Innovation Promotion Association CAS (No. 2017210), and an Australian Government Research Training Program Scholarship.

References

[Bello *et al.*, 2017] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V. Le. Neural optimizer search with reinforcement learning. In *ICML*, 2017.

[Bianchi *et al.*, 2019] Filippo Maria Bianchi, Daniele Grattarola, Lorenzo Livi, and Cesare Alippi. Graph neural networks with convolutional arma filters. *ArXiv*, abs/1901.01343, 2019.

[Cai *et al.*, 2019] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019.

[Defferrard *et al.*, 2016] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*, 2016.

[Dong and Yang, 2019] Xuanyi Dong and Yi Yang. Searching for a robust neural architecture in four gpu hours. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1761–1770, 2019.

[Fey and Lenssen, 2019] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[Gori *et al.*, 2005] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks*, 2005.

[Hamilton *et al.*, 2017] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[Hu *et al.*, 2019] Fenyu Hu, Yanqiao Zhu, Shu Wu, Liang Wang, and Tieniu Tan. Hierarchical graph convolutional networks for semi-supervised node classification. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, (IJCAI)*, 2019.

[Kipf and Welling, 2017] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[Klicpera *et al.*, 2019] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Combining neural networks with personalized pagerank for classification on graphs. In *International Conference on Learning Representations*, 2019.

[Liu *et al.*, 2018a] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[Liu *et al.*, 2018b] Ziqi Liu, Chaochao Chen, Longfei Li, Jun Zhou, Xiaolong Li, and Le Song. Geniepath: Graph neural networks with adaptive receptive paths. *CoRR*, abs/1802.00910, 2018.

[Niepert *et al.*, 2016] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *ICML*, 2016.

[Pham *et al.*, 2018] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

[Real *et al.*, 2018] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018.

[Shchur *et al.*, 2018] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. 2018.

[Vaswani *et al.*, 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Lawrence Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.

[Velickovic *et al.*, 2017] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. *CoRR*, abs/1710.10903, 2017.

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

[Wu *et al.*, 2019a] Felix Wu, Amauri H. Souza, Tianyi Zhang, Christopher Fifty, Rui Zhang, and Kilian Q. Weinberger. Simplifying graph convolutional networks. In *ICML*, 2019.

[Wu *et al.*, 2019b] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. Session-based recommendation with graph neural networks. In *AAAI*, 2019.

[Xie *et al.*, 2019] Sirui Xie, H P Zheng, Chunxiao Liu, and Liang Lin. Snas: Stochastic neural architecture search. In *International Conference on Learning Representations*, 2019.

[Xu *et al.*, 2018] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *CoRR*, abs/1810.00826, 2018.

[Yan *et al.*, 2018] Sijie Yan, Yuanjun Xiong, and Dahua Lin. Spatial temporal graph convolutional networks for skeleton-based action recognition. In *AAAI*, 2018.

[You *et al.*, 2019] Jiaxuan You, Rex Ying, and Jure Leskovec. Position-aware graph neural networks. In *ICML*, 2019.

[Yu *et al.*, 2018] Bing Yu, Haoteng Yin, and Zhanxing Zhu. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *IJCAI*, 2018.

[Zaheer *et al.*, 2017] Manzil Zaheer, Satwik Kottur, Siamak Ravanbhakhsh, Barnabas Poczos, and Alexander Smola. Deep sets. 2017.

[Zoph and Le, 2016] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.