Chapter 17
Graph Neural Networks: AutoML

Kaixiong Zhou, Zirui Liu, Keyu Duan and Xia Hu

Abstract Graph neural networks (GNNs) are efficient deep learning tools to analyze networked data. Being widely applied in graph analysis tasks, the rapid evolution of GNNs has led to a growing number of novel architectures. In practice, both neural architecture construction and training hyperparameter tuning are crucial to the node representation learning and the final model performance. However, as the graph data characteristics vary significantly in the real-world systems, given a specific scenario, rich human expertise and tremendous laborious trials are required to identify a suitable GNN architecture and training hyperparameters. Recently, automated machine learning (AutoML) has shown its potential in finding the optimal solutions automatically for machine learning applications. While releasing the burden of the manual tuning process, AutoML could guarantee access of the optimal solution without extensive expert experience. Motivated from the previous successes of AutoML, there have been some preliminary automated GNN (AutoGNN) frameworks developed to tackle the problems of GNN neural architecture search (GNN-NAS) and training hyperparameter tuning. This chapter presents a comprehensive and up-to-date review of AutoGNN in terms of two perspectives, namely search space and search algorithm. Specifically, we mainly focus on the GNN-NAS problem and present the

Kaixiong Zhou
Department of Computer Science and Engineering, Texas A&M University, e-mail: zkxiong@tamu.edu

Zirui Liu
Department of Computer Science and Engineering, Texas A&M University, e-mail: tradigrada@tamu.edu

Keyu Duan
Department of Computer Science and Engineering, Texas A&M University, e-mail: k.duan@tamu.edu

Xia Hu
Department of Computer Science and Engineering, Texas A&M University, e-mail: hu@cse.tamu.edu

state-of-the-art techniques in these two perspectives. We further discuss the open problems related to the existing methods for future research.

17.1 Background

Graph neural networks (GNNs) have made substantial progress in integrating deep learning approaches to analyze graph-structured data collected from various domains, such as social networks (Ying et al, 2018b; Huang et al, 2019d; Monti et al, 2017; He et al, 2020), academic networks (Yang et al, 2016b; Kipf and Welling, 2017b; Gao et al, 2018a), and biochemical modular graphs (Zitnik and Leskovec, 2017; Aynaz Taheri, 2018; Gilmer et al, 2017; Jiang and Balaprakash, 2020). Following the common message passing strategy, GNNs apply spatial graph convolutional layer to learn a node's embedding representation via aggregating the representations of its neighbors and combining them to the node itself. A GNN architecture is then constructed by the stacking of multiple such layers and their inter-layer skip connections, where the elementary operations of a layer (e.g., aggregation & combination functions) and the concrete inter-layer connections are specified specifically in each design. To adapt to different real-world applications, a variety of GNN architectures have been explored, including GCN (Kipf and Welling, 2017b), GraphSAGE (Hamilton et al, 2017b), GAT (Veličković et al, 2018), SGC (Wu et al, 2019a), JKNet (Xu et al, 2018a), and GCNII (Chen et al, 2020l). They vary in how to aggregate the neighborhood information (e.g., mean aggregation in GCN versus neighbor attention learning in GAT) and the choices of skip connections (e.g., none connection in GCN versus initial connection in GCNII).

Despite the significant success of GNNs, their empirical implementations are usually accompanied with careful architecture engineering and training hyperparameter tuning, aiming to adapt to the different types of graph-structured data. Based on the researcher's prior knowledge and trial-and-error tuning processes, a GNN architecture is instantiated from its model space specifically and evaluated in each graph analysis task. For example, considering the underlying model GraphSAGE (Hamilton et al, 2017b), the various-size architectures determined by the different hidden units are applied respectively for citation networks and protein-protein interaction graphs. Furthermore, the optimal skip connection mechanisms in JKNet architectures (Xu et al, 2018a) vary with the real-world tasks. Except the architecture engineering, the training hyperparameters play important roles in the final model performance, including learning rate, weight decay, and epoch numbers. In the open repositories, their hyperparameters are manually manipulated to get the desired model performances. The tedious selections of GNN architectures and training hyperparameters not only burden data scientists, but also make it difficult for beginners to access the high-performance solutions quickly for their tasks on hand.

Automated machine learning (AutoML) has emerged as a prevailing research to liberate the community from the time-consuming manual tuning processes (Chen

et al, 2021). Given any task and based on the predefined search space, AutoML aims at automatically optimizing the machine learning solutions (or denoted with the term designs), including neural architecture search (NAS) and automated hyperparameter tuning (AutoHPT). While NAS targets the optimization of architecture-related parameters (e.g., the layer number and hidden units), AutoHPT indicates the selections of training-related parameters (e.g., the learning rate and weight decay). They are the sub-fields of AutoML. It has been widely reported that the novel neural architectures discovered by NAS outperform the human-designed ones in many machine learning applications, including image classification (Zoph and Le, 2016; Zoph et al, 2018; Liu et al, 2017b; Pham et al, 2018; Jin et al, 2019a; Luo et al, 2018; Liu et al, 2018b,c; Xie et al, 2019a; Kandasamy et al, 2018), semantic image segmentation (Chenxi Liu, 2019), and image generation (Wang and Huan, 2019; Gong et al, 2019). Dating back to 1900's (Kohavi and John, 1995), it has been commonly acknowledged that AutoHPT could improve over the default training setting (Feurer and Hutter, 2019; Chen et al, 2021). Motivated by the previous successful applications of AutoML, there have been some recent efforts on conjoining the researches of AutoML and GNNs (Gao et al, 2020b; Zhou et al, 2019a; You et al, 2020a; Ding et al, 2020a; Zhao et al, 2020a,g; Nunes and Pappa, 2020; Li and King, 2020; Shi et al, 2020; Jiang and Balaprakash, 2020). They generally define the automated GNN (AutoGNN) as an optimization problem and formulate their own working pipelines from three perspectives, as shown in Figure 17.1, the search space, search algorithm, and performance estimation strategy. The search space consists of a large volume of candidate designs, including GNN architectures and the training hyperparameters. On top of the search space, several heuristic search algorithms are proposed to solve the NP-complete optimization problem by iteratively approximating the well-performing designs, including random search (You et al, 2020a). The objective of performance estimation is to accurately estimate the task performance of every candidate design explored at each step. Once the search progress terminates, the best neural architecture accompanied with the suitable training hyperparameters is returned to be evaluated on the downstream machine learning task.

In this chapter, we will organize the existing efforts and illustrate AutoGNN framework with the following sections: notations, problem definition, and challenges of AutoGNN (in Sections 17.1.1, 17.1.2, and 17.1.3), search space (in Section 17.2), and search algorithm (in Section 17.3). We then present the open problems for future research in Section 17.4. Specially, since the community's interests mainly focus on discovering the powerful GNN architecture, we pay more attentions to GNN-NAS in this chapter.

17.1.1 Notations of AutoGNN

Following the previous expressions (You et al, 2020a), we use the term "design" to refer to an available solution of the optimization problem in AutoGNN. A design consists of a concrete GNN architecture and a specific set of training hy-

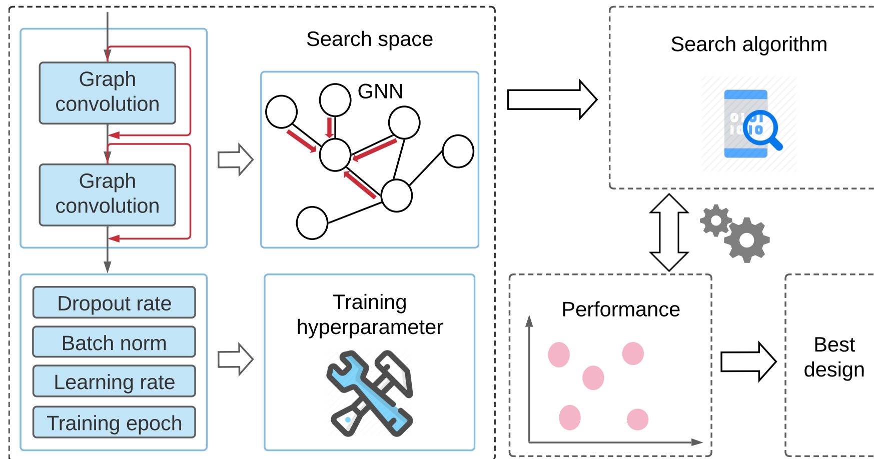

Fig. 17.1: Illustration of a general framework for AutoGNN. The search space consists of plenty of designs, including GNN architectures and the training hyperparameters. At each step, the search algorithm samples a candidate design from the search space and estimates its model performance on the downstream task. Once the search progress terminates, the best design accompanied with the highest performance on the validation set is returned and exploited for the real-world system.

perparameters. Specifically, the design is characterized by multiple dimensions, including architecture dimensions (e.g., the layer number, skip connections, aggregation, and combination functions) and hyperparameter dimensions (e.g., the learning rate and weight decay). Along each design dimension, there is a series of different elementary options provided to support the automated architecture engineering or training hyperparameter tuning. For example, we could have candidates $\{\mathrm{SUM}, \mathrm{MEAN}, \mathrm{MAX}\}$ at the aggregation function dimension, and use $\{1\text{e-}4, 5\text{e-}4, 1\text{e-}3, 5\text{e-}3, 0.01, 0.1\}$ at the learning rate dimension. Given the series of candidate options along each dimension, the search space in AutoGNN is constructed by Cartesian product of all the design dimensions. A design is instantiated by assigning concrete values to these dimensions, such as a GNN architecture with the aggregation function of MEAN and learning rate of 1e-3. Note that GNN-NAS and AutoHPT explore in the search spaces consisted of expansive GNN architectures and hyperparameter combinations, respectively; AutoGNN optimizes in a more comprehensive search space containing both of them.

17.1.2 Problem Definition of AutoGNN

Before diving into detailed techniques, we examine the essence of AutoGNN by formally defining its optimization problem. To be specific, let $\mathscr{F}$ be the search space. Let $\mathscr{D}_{\text{train}}$ and $\mathscr{D}_{\text{valid}}$ be the training and validation sets, respectively. Let M be the performance evaluation metric of a design in any given graph analysis task, e.g., F1 score or accuracy in the node classification task. The objective of AutoGNN is to find the optimal design $f^* \in \mathscr{F}$ in terms of M evaluated on the validation set $\mathscr{D}_{\text{valid}}$. Formally, AutoGNN requires solving the following bi-level optimization problem:

$$\begin{aligned} & f^* = \operatorname{argmax}_{f \in \mathscr{F}} M(f(\theta^*); D_{\text{valid}}), \\ \text{s.t.} \quad & \theta^* = \operatorname{argmin}_{\theta} L(f(\theta); D_{\text{train}}). \end{aligned} \tag{17.1}$$

where θ^* denotes the optimized trainable weights of design f and L denotes the loss function. For each design, AutoGNN will first optimize its associated weights θ by minimizing the loss on the training set through gradient descent, and then evaluates it on the validation set to decide whether this design is the optimal one. By solving the above optimization problem, AutoGNN automates the architecture engineering and training hyperparameter tuning procedure, and pushes GNN designs to examine a broad scope of candidate solutions. However, it is well known that such the bi-level optimization problem is NP-complete (Chen et al, 2021), thereby it would be extremely time-consuming for searching and evaluating the well-performing designs on large graphs with massive nodes and edges. Fortunately, there have been some heuristic search techniques proposed to locate the local optimal design (e.g., CNN or RNN architecture) as close as possible to the global one in the applications of image classification and natural language processing, including reinforcement learning (RL) (Zoph and Le, 2016; Zoph et al, 2018; Pham et al, 2018; Cai et al, 2018a; Baker et al, 2016), evolutionary methods (Liu et al, 2017b; Real et al, 2017; Miikkulainen et al, 2019; Xie and Yuille, 2017; Real et al, 2019), and Bayesian optimization (Jin et al, 2019a). They iteratively explore the next design and update the search algorithm based on the performance feedback of the new design, in order to move toward the global optimal solution. Compared with the previous efforts, the characteristics of AutoGNN problem could be viewed from two aspects: the search space and search algorithms tailored to identify the optimal design of GNN. In the following sections, we list the challenge details and the existing AutoGNN work.

17.1.3 Challenges in AutoGNN

The direct application of existing AutoML frameworks to automate GNN designs is non-trivial, due to the two major challenges as follows.

First, the search space of AutoGNN is significantly different from the ones in the AutoML literature. Taking NAS applied in discovering CNN architectures (Zoph and Le, 2016) as an example, the search space of convolution operation is mainly

specified by the convolutional kernel size. In contrast, considering the message-passing based graph convolution, the search space of spatial graph convolution is constructed by multiple key architecture dimensions, including aggregation, combination, and embedding activation functions. With the growing number of GNN model variants, it is important to formulate a good search space being both expressive and compact. On the one hand, the search space should cover the important architecture dimensions to subsume the existing human-designed architectures and adapt to a series of diverse graph analysis tasks. On the other hand, the search space should be compact by excluding the non-general dimensions and incorporating modest ranges of options along each dimension, in order to save the search time cost.

Second, the search algorithm should be tailored to discover the well-performing design efficiently based on the special search space in AutoGNN. The search controller determines how to iteratively explore the search space and update the search algorithm according to the performance feedbacks of sampled designs. A good controller needs to balance the trade-off between exploration and exploitation during the search progress, in order to avoid the premature sub-optimal region and quickly discover the well-performing designs, respectively. However, the previous search algorithms may be inefficient to the application of GNN-NAS. Specially, one of the key properties in GNN architectures is that the model performance may vary significantly with a slight modification along an architecture dimension. For example, it has been theoretically and empirically demonstrated that the graph classification accuracy could be improved by simply replacing the max pooling with summation in the aggregation function dimension of GNN (Xu et al, 2019d). The previous RL-based methods sample and evaluate the whole architecture at each search step. It would be hard for the search algorithms to learn the following relationship towards exploring better GNN: which part of the architecture dimension modifications improves or degrades the model performance. Another challenging problem is the surge of new graph analysis tasks, which requires huge computation resources to optimize GNN architectures. Instead of searching the optimal GNN from scratch, it is crucial to transfer the well-performing architectures discovered before to the new task to save the expensive computation cost.

17.2 Search Space

In this section, we summarize the search spaces in literature. As shown in Figure 17.2, the search spaces of designs in AutoGNN are differentiated according to GNN architectures and training hyperparameters, whose details are listed as below.

17.2.1 Architecture Search Space

Considering the existing AutoGNN frameworks (Gao et al, 2020b; Zhou et al, 2019a), GNN model is commonly implemented based on the spatial graph convolution mechanism. To be specific, the spatial graph convolution takes the input graph as a computation graph and learns node embeddings by passing messages along edges. A node embedding is updated recursively by aggregating the embedding representations of its neighbors and combining them to the node itself. Formally, the k-th spatial graph convolutional layer of GNN could be expressed as:

$$\begin{aligned} \mathbf{h}_i^{(k)} &= \text{AGGREGATE}(\{a_{ij}^{(k)} W^{(k)} \mathbf{x}_j^{(k-1)} : j \in \mathcal{N}(i)\}), \\ \mathbf{x}_i^{(k)} &= \text{ACT}(\text{COMBINE}(W^{(k)} \mathbf{x}_i^{(k-1)}, \mathbf{h}_i^{(k)})). \end{aligned} \tag{17.2}$$

$\mathbf{x}_i^{(k)}$ denotes the embedding vector of node v_i at the k-th layer. $\mathcal{N}(i)$ denotes the set of neighbors adjacent to node v_i. $W^{(k)}$ denotes the trainable weight matrix used to project node embeddings. $a_{ij}^{(k)}$ denotes the message-passing weight along edge connecting nodes v_i and v_j, which is determined by normalized graph adjacency matrix or learned from attention mechanism. Function AGGREGATE, such as mean, max, and sum pooling, is used to aggregate neighbor representations. Function COMBINE is used to combine neighbor embedding $\mathbf{h}_i^{(k)}$ as well as node embedding $\mathbf{x}_i^{(k-1)}$ from the last layer. Finally, function ACT (e.g., ReLU) is used to add non-linearity to the embedding learning.

As shown in Figure 17.2, GNN architecture consists of several graph convolutional layers defined in Eq. equation 17.2, and may incorporate skip connection between any two arbitrary layers similar to residual CNN (He et al, 2016a). Following the previous definitions in NAS, we use the term "micro-architecture" to represent a graph convolutional layer, including the specifications of hidden units and graph convolutional functions; we use the term "macro-architecture" to represent network topology, including the choices of layer depth, inter-layer skip connections, and pre/post-processing layers. The architecture search space contains a large volume of diverse GNN architectures, which could be categorized into the search spaces of micro-architectures as well as macro-architectures.

17.2.1.1 Micro-architecture Search Space

According to Eq. equation 17.2 and as shown in Figure 17.2, the micro-architecture of a graph convolutional layer is characterized by the following five architecture dimensions:

- **Hidden units:** Trainable matrix $W^{(k)} \in \mathbb{R}^{d^{(k-1)} \times d^{(k)}}$ maps node embeddings to a new space and learns to extract the informative features. $d^{(k)}$ is the number of hidden units and plays key role in the task performance. In the GNN-NAS

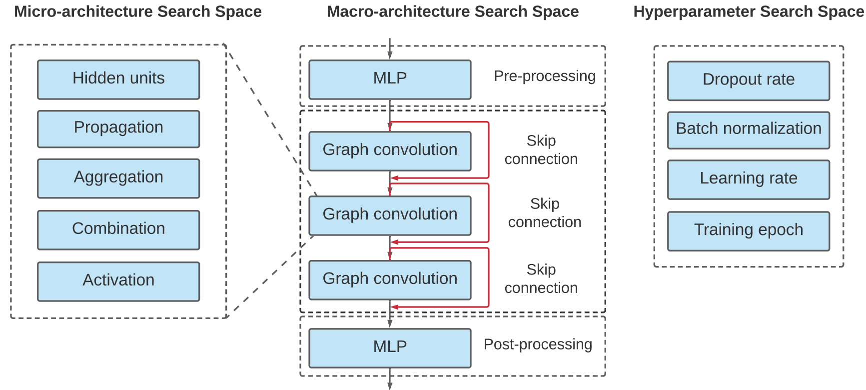

Fig. 17.2: Illustration of a comprehensive search space, which consists of micro-architecture, macro-architecture, and training hyperpameter search spaces. Each space is characterized by multiple dimensions, such as hidden units, propagation function, etc, in the micro-architecture search space. Each dimension provides a series of candidate options, and the search space is constructed by Cartesian product of all its dimensions. A discrete point in the comprehensive search space represents a specific design, which adopts one option at each dimension.

frameworks of GraphNAS (Gao et al, 2020b) and AGNN (Zhou et al, 2019a), $d^{(k)}$ is usually selected from set $\{4, 8, 16, 32, 64, 128, 256\}$.

- **Propagation function:** It determines the message-passing weight $a_{ij}^{(k)}$ to specify how node embeddings are propagated upon the input graph structure. In a wide variety of GNN models (Kipf and Welling, 2017b; Wu et al, 2019a; Hamilton et al, 2017b; Ding et al, 2020a), $a_{ij}^{(k)}$ is defined by the corresponding element from the normalized adjacency matrix: $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ or $\tilde{D}^{-1}\tilde{A}$, where $\tilde{A}$ is the self-loop graph adjacency matrix and $\tilde{D}$ is its degree matrix, respectively. Note that the real-world graph-structured data could be both complex and noisy (Lee et al, 2019c), which leads to the inefficient neighbor aggregation. GAT (Veličković et al, 2018) applies attention mechanism to compute $a_{ij}^{(k)}$ to attend on relevant neighbors. Based on the existing GNN-NAS frameworks (Gao et al, 2020b; Zhou et al, 2019a; Ding et al, 2020a), we list the common choices of propagation functions in Table 17.1.
- **Aggregation function:** Depending on the input graph structure, a proper application of aggregation function is important to learn the informative neighbor distribution (Xu et al, 2019d). For example, a mean pooling function takes the average of neighbors, while a max pooling only preserves the significant one. The aggregation function is usually selected from set $\{\mathrm{SUM}, \mathrm{MEAN}, \mathrm{MAX}\}$.
- **Combination function:** It is used to combine neighbor embedding $\mathbf{h}_i^{(k)}$ and projected embedding $W^{(k)}\mathbf{x}_i^{(k-1)}$ of the node itself. Examples of combination

function include sum and multiple layer perceptron (MLP), etc. While the sum operation simply adds the two embeddings, MLP further applies linear mapping based upon the summation or concatenation of these two embeddings.

- **Activation function:** The candidate activation function is usually selected from {Sigmoid, Tanh, ReLU, Linear, Softplus, LeakyReLU, ReLU6, ELU}.

Given the above five architecture dimensions and their associated candidate options, the micro-architecture search space is constructed by their Cartesian product. Each discrete point in the micro-architecture search space corresponds to a concrete micro-architecture, e.g., a graph convolutional layer with {Hidden units: 64, Propagation function: GAT, aggregation function: SUM, combination function: MLP, Activation function: ReLU}. By providing the extensive candidate options along each dimension, the micro-architecture search space covers most of layer implementations in the state-of-the-art models, such as Chebyshev (Defferrard et al, 2016), GCN (Kipf and Welling, 2017b), GAT (Veličković et al, 2018), and LGCN (Gao et al, 2018a).

Table 17.1: Propagation function candidates to compute weight $a_{ij}^{(k)}$ if nodes v_i and v_j are connected; otherwise $a_{ij}^{(k)} = 0$. Symbol $||$ denotes the concatenation operation, $\mathbf{a}$, $\mathbf{a}_l$ and $\mathbf{a}_r$ denote trainable vectors, and $W_G^{(k)}$ is a trainable matrix.

Propagation Types	Propagation functions	Equations
Normalized adjacency	$\tilde{A}$	1
	$\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$	$\frac{1}{\sqrt{\lvert\mathcal{N}(i)\rvert\lvert\mathcal{N}(j)\rvert}}$
	$\tilde{D}^{-1}\tilde{A}$	$\frac{1}{\lvert\mathcal{N}(i)\rvert}$
Attention mechanism	GAT	$\mathrm{LeakyReLU}(\mathbf{a}^\top(W^{(k)}\mathbf{x}_i^{(k-1)}\vert\vert W^{(k)}\mathbf{x}_j^{(k-1)}))$
	SYM-GAT	$a_{ij}^{(k)} + a_{ji}^{(k)}$ based on GAT
	COS	$\mathbf{a}^\top(W^{(k)}\mathbf{x}_i^{(k-1)}\vert\vert W^{(k)}\mathbf{x}_j^{(k-1)})$
	LINEAR	$\tanh(\mathbf{a}_l^\top W^{(k)}\mathbf{x}_i^{(k-1)} + \mathbf{a}_r^\top W^{(k)}\mathbf{x}_i^{(k-1)})$
	GERE-LINEAR	$W_G^{(k)}\tanh(W^{(k)}\mathbf{x}_i^{(k-1)} + W^{(k)}\mathbf{x}_i^{(k-1)})$

17.2.1.2 Macro-architecture Search Space

Besides the micro-architecture, another architectural level of GNN is its macro-architecture as shown in Figure 17.2, i.e., the network topology. The macro-architecture of GNN specifies the numbers of graph convolutional layers as well as pre/post-processing layers, and the choices of skip connections (You et al, 2020a; Li et al, 2018b, 2019c). We list the details of these four architecture dimensions in the following.

- **Graph convolutional layer depth:** The direct stacking of multiple layers is commonly adopted to improve the reception fields of nodes. Let l_{gc} denote the number of graph convolutional layers. l_{gc} is usually selected from range $[2, 10]$.
- **Pre-processing layer depth:** In real-world applications, the length of nodes' input features may be too large and leads to costly computation in hidden feature learning. The feature pre-processing is included in search space (You et al, 2020a) for the first time and conducted by MLP, whose layer number is denoted as l_{pre}. l_{pre} is sampled from candidates $\{0, 1, 2, 3\}$.
- **Post-processing layer depth:** Similarly, the post-processing layers of MLP are applied to project hidden embeddings into task-specific space, e.g., the embedding space with dimensions the same as class labels in the node classification task. Let l_{post} denote the layer number with examples $\{0, 1, 2, 3\}$.
- **Skip connections:** Following the residual deep CNNs in computer vision and the recent deep GNNs, skip connections have been incorporated in the search space of GNN-NAS frameworks (You et al, 2020a; Zhao et al, 2020g,a). To be specific, at layer l, the embeddings of up to $l-1$ previous layers could be sampled and combined to the current layer's output, leading to 2^{k-1} possible decisions at layer k. For the prior node embeddings that are connected to the current output, there have been a series of candidate options developed to combine them, namely $\{\text{SUM}, \text{CAT}, \text{MAX}, \text{LSTM}\}$. Specially, option SUM, CAT or MAX adds, concatenates or element-wisely max pools these connected embeddings. LSTM uses an attention mechanism to compute the importance score of each layer, and then obtain the weighted average of the connected embeddings (Xu et al, 2018a).

The entire architecture space is constructed by Cartesian product of the micro and macro-architecture search spaces, which is totally characterized by the nine architecture dimensions. It could be extremely huge and comprehensive to subsume the recent residual GNN models, such as JKNet (Xu et al, 2018a) and deeperGCN (Li et al, 2018b).

17.2.2 Training Hyperparameter Search Space

The training hyperparameters have significant impacts on the task performances of GNN architectures, and have been explored in AutoGNN frameworks (You et al, 2020a; Shi et al, 2020). We summarize four important dimensions of training hyperparameters in the following and show them in Figure 17.2.

- **Dropout rate**: At the beginning of each graph convolutional layer or pre/post-processing layer, a proper dropout rate is crucial to avoid the over-fitting issue. The widely-used examples are $\{\text{False}, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6\}$.
- **Batch normalization**: It is applied after graph convolutional layer or pre/post-processing layer to normalize node embeddings of the whole graph or a batch (Zhou et al, 2020d; Zhao and Akoglu, 2019; Ioffe and Szegedy, 2015). The candidate

normalization techniques include {False, BatchNorm ($IoffeandSzegedy$, 2015), PairNorm ($ZhaoandAkoglu$, 2019), DGN ($Zhouet\ al$, 2020d), NodeNorm ($Zhouet\ al$, 2020c), GraphNorm ($Caiet\ al$, 2020d)}.

- **Learning rate**: While a larger learning rate leads to a premature suboptimal solution, a smaller one will make the optimization process converge slowly. The candidate learning rates are {1e-4, 5e-4, 1e-3, 5e-3, 0.01, 0.1}.
- **Training epoch**: According to the common practice (You et al, 2020a; Kipf and Welling, 2017b), the training epoch examples are {100, 200, 400, 500, 1000}.

17.2.3 Efficient Search Space

Given the micro-architecture, macro-architecture, and training hyperparameters search spaces, in the practical systems, the applied search space is formulated by Cartesian product of any combination of them. Although a large search space subsumes the diverse GNN architectures and training environments to adapt to the different graph analysis tasks, it would be time-consuming to explore the optimal design. To make the search progress efficient, there are two mainstream simplifying search spaces applied in the existing AutoGNN frameworks.

- **Focus on GNN-NAS:** Instead of fully tuning the training hyperparameters, most of AutoGNN (or GNN-NAS) frameworks (Gao et al, 2020b; Zhou et al, 2019a; Zhao et al, 2020a,g; Ding et al, 2020a; Nunes and Pappa, 2020; Li and King, 2020; Jiang and Balaprakash, 2020) focus on tackling the problem of discovering the well-performing GNN architectures. Comparing with AutoHPT, it is commonly acknowledged that a novel architecture discovered from GNN-NAS is more important and challenging to the research community, which could motivate the data scientist to improve GNN model paradigms in the future. In GNN-NAS, the search space is thus reduced to the one containing only the neural architecture variants.
- **Simplify architecture search space:** Even in GNN-NAS, the plenty of architecture dimensions and their associated candidate options still make the search space complex. Based on the prior knowledge about the impacts of different modules on model performances, one would prefer to explore only along the crucial architecture dimensions in the practical systems. For example, it is found that the simplified search space (Zhao et al, 2020a) characterized by aggregation function and skip connections could generate the high-performance GNN architectures comparable to ones from the comprehensive search spaces (Gao et al, 2020b; Zhou et al, 2019a). Specially, since the decision cardinality of skip connections increases exponentially with layers, the simplified search space even only explores the skip connections in the last layer similar to JKNet (Xu et al, 2018a). In another simplified search space, the model-specific architecture dimensions are excluded and pre-defined based on expert experiences, including the hidden units, propagation function, and combination function.

17.3 Search Algorithms

Many different search strategies can be used to explore the search space in AutoGNN, including random search, evolutionary methods, RL, and differentiable search methods. In this section, we will introduce the basic concepts of these search algorithms and how to utilize them to explore candidate designs.

17.3.1 Random Search

Given a search space, random search randomly samples the various designs with equal probability. The random search is the most basic approach, yet it is quite effective in practice. In addition to serve as a baseline in AutoGNN works (Zhou et al, 2019a; Gao et al, 2020b), random search is the standard benchmark for comparing the effectiveness of different candidate options along a dimension in the search space (You et al, 2020a). Specially, suppose the dimension to be evaluated is batch normalization, whose candidate examples are given by $\{$False, BatchNorm$\}$. To comprehensively compare the effectiveness of these two options, a series of diverse designs are randomly sampled from the search space, where the batch normalization is reset to False and BatchNorm in each design, respectively. Each pair of designs (referred to Normalization=False and Normalization=BatchNorm) are compared in terms of their model performances on a downstream graph analysis task. It is found that the designs with Normalization=BatchNorm generally rank higher than the others, which indicates the benefit of including BatchNorm in the model design.

17.3.2 Evolutionary Search

Evolutionary methods evolve a population of designs, i.e., the set of different GNN architectures and training hyperparameters. In every evolution step, at least one design from the population is sampled and serves as a parent to generate a new child design by applying mutations to it. In the context of AutoGNN, the design mutations are local operations, such as changing the aggregation function from MAX to SUM, altering the hidden units, and altering a specific training hyperparameter. After training the child design, its performance is evaluated on the validation set. The superior design will be added to the population. Specifically, Shi et al (2020) proposes to select two parent designs and then crossover them along some dimensions. To generate the diverse child designs, Shi et al (2020) further mutates the above crossover designs.

17.3.3 Reinforcement Learning Based Search

RL (Silver et al, 2014; Sutton and Barto, 2018) is a learning paradigm concerned with how agents ought to take actions in an environment to maximize the reward. In the context of AutoGNN, the agent is the so-called "controller", which tries to generate promising designs. The generation of design can be regarded as the controller's action. The controller's reward is often defined as the model performance of generated design on the validation set, such as validation accuracy for the node classification task. The controller is trained in a loop as shown in Figure 17.3: the controller first samples a candidate design and trains it to convergence to measure its performance on the task of desire. Note that the controller is usually realized by RNN, which generates the design of GNN architecture and training hyperparameters as a string of variable strength. The controller then uses the performance as a guiding signal to update itself toward finding the more promising design in the future search progress.

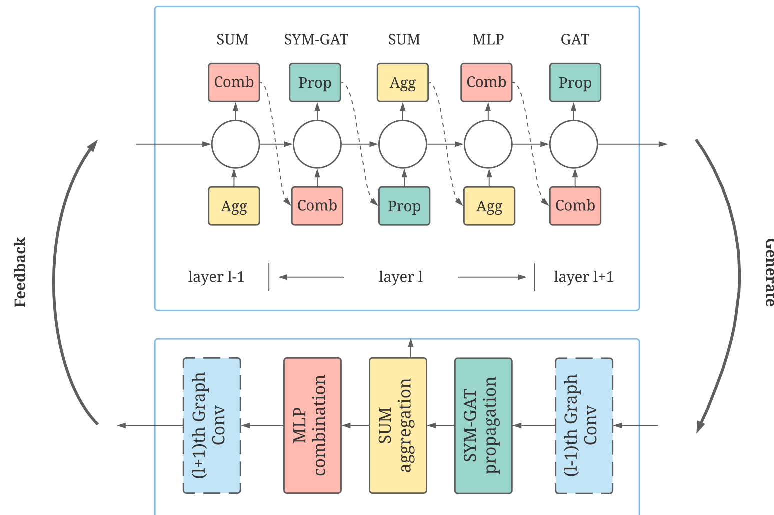

Fig. 17.3: A illustration of reinforcement learning based search algorithm. The controller (upper block) generates a GNN architecture (lower block) and tests it on the validation dataset. By treating the architecture as a string with variable length, the controller usually applies RNN to sequentially sample options in the different dimensions (e.g., combination, aggregation, and propagation functions) to formulate the final GNN architecture. The validation performance is then used as feedback to train the controller. Note that the architecture dimensions here are just used for the illustration purpose. Please refer to Section 17.2 for a complete introduction of the search space.

The existing RL-based AutoGNN frameworks target at the sub-field problem of GNN-NAS. Generally, in RL-based GNN-NAS, there are two sets of trainable parameters: the parameters of the controller, denoted by ω, and the parameters of a GNN architecture, denoted by θ. The training procedure consists of two interleaving phases, which alternatively solves the bi-level optimization problem as shown in Eq. equation 17.1. The first phase trains θ on the training data set $\mathscr{D}_{train}$ with a fixed number of epochs using standard back-propagation. The second phase trains ω to learn to sample high-performance GNN architectures evaluated on the validation set $\mathscr{D}_{valid}$. These two phases are alternated during the training. Specifically, in the first phase, the controller proposes a GNN architecture f and performs gradient descent on θ to minimize the loss function $\mathscr{L}(f(\theta);\mathscr{D}_{train})$, which is computed on the batches of training data. In the second phase, the optimized parameter θ^* is fixed to update the controller parameters ω, aiming to maximize the expected reward:

$$\omega^* = \text{argmax}_\omega\ \mathbb{E}_{f\sim\pi(f;\omega)}[\mathscr{R}(f(\theta^*);\mathscr{D}_{valid})]. \tag{17.3}$$

Here, $\pi(f;\omega)$ is the controller's policy parameterized by ω to sample and generate GNN architecture f. The reward $\mathscr{R}(f(\theta^*);\mathscr{D}_{valid})$ is the model performance defined by the task of desire, such as the accuracy for the node classification task. Furthermore, the reward is computed on the validation set, rather than on the training set, to encourage the controller to select architectures that generalize well. In most of the existing work, the gradient of the expected reward $\mathbb{E}_{f\sim\pi(f;\omega)}[\mathscr{R}(f(\theta^*);\mathscr{D}_{valid})]$ with respect to ω is computed using REINFORCE rule (Sutton et al, 2000).

Considering GNN-NAS efforts in literature, RL-based search algorithms differ in how they represent and train the controller. GraphNAS uses an RNN controller to sequentially sample from the multiple architecture dimensions and generate a string that encodes a GNN architecture (Gao et al, 2020b). Based on the expected reward signaling the quality of the whole architecture, the RNN controller has to optimize the sampling policies along all the dimensions. AGNN (Zhou et al, 2019a) is motivated by an observation that the minor modification to an architecture dimension can lead to abrupt change in performance. For example, the graph classification accuracy of GNN may be significantly improved by only changing the choice of aggregation function from MAX to SUM (Xu et al, 2019d). Based on this observation, AGNN proposes a more efficient controller consisted of a series of RNN sub-controllers, each corresponding to an independent architecture dimension. At each step, AGNN only applies one of the RNN sub-controllers to sample new options from the corresponding dimension, and uses these options to mutate the best architecture found so far. By evaluating such a slightly-mutated design, the RNN sub-controller can exclude the noises generated from the other architecture dimension modifications, and better trains the sampling policy of its own dimension.

17.3.4 Differentiable Search

There are several candidate options along each architecture dimension. For example, for the aggregation function at a particular layer, we have the option of applying either a SUM, a MEAN, or a MAX pooling. The common search approaches in GNN-NAS, such as random search, evolutionary algorithms, and RL-based search methods, treat selecting the best option as a black-box optimization problem over a discrete domain. At each search step, they sample and evaluate a single architecture from the discrete architecture search space. However, such the search process towards well-performing GNNs will be very time-consuming since the number of possible models is extremely large. Differentiable search algorithms relax the discrete search space to be continuous, which can be optimized efficiently by gradient descent. Specifically, for each architecture dimension, the differentiable search algorithms usually relax the hard choice from the candidate set into a continuous distribution, where each option is assigned with a probability. One example for illustrating the differentiable search along the aggregation function dimension is shown in Figure 17.4. At the k-th layer, the node embedding output of aggregation function can be decomposed and expressed as:

$$
\begin{aligned}
&\mathbf{h}_i^{(k)} = \begin{cases} \sum_m \alpha_m o_m(\mathbf{x}_j^{(k-1)} : j \in \mathcal{N}(i) \cup \{i\}), \\ \text{or} \\ \alpha_m o_m(\mathbf{x}_j^{(k-1)} : j \in \mathcal{N}(i) \cup \{i\}),\ m \sim p(\alpha_m), \end{cases} \\
&\text{s.t.}\ \sum_m \alpha_m = 1.
\end{aligned} \tag{17.4}
$$

o_m represents the m-th aggregation function option, and α_m is the sampling probability associated with the corresponding option. The probability distribution along a dimension is regularized to have the sum of one. The architecture distribution is then formulated by the union probability distribution of all the dimensions. At each search step, as shown in Eq.equation 17.4 (with the example of the aggregation function dimension), the real operation of a dimension in a new architecture could be generated by two different ways: weighted option combination and option sampling. For the case of weighted option combination, the real operation is represented by the weighted average of all candidate options. For the other case, the real operation is instead sampled from the probability distribution $p(\alpha_m)$ of the corresponding architecture dimension. In both cases, the adopted options are scaled by their sampling probabilities to support the architecture distribution optimization by gradient descent. The architecture distribution is then updated directly by backpropagating the training loss at each training step. During the testing, the discrete architecture can be obtained by retaining the strongest candidate with the highest probability α_m along each dimension. In contrast to black-box optimization, gradient-based optimization is significantly more data efficient, and hence greatly speeds up the search process.

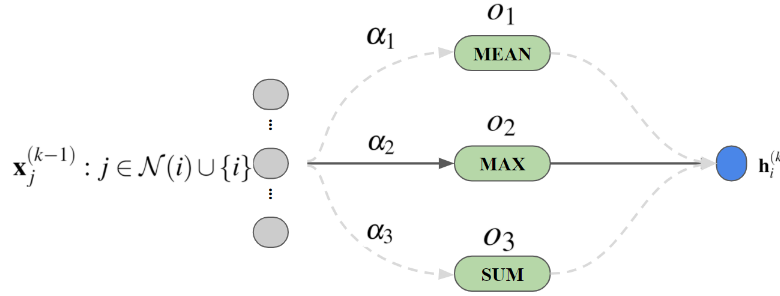

Fig. 17.4: One example for illustrating the differentiable search for the aggregation function. At a search step, the aggregation function is given by the weighted combination of the three candidates, or instead realized by one sampled option (e.g., MAX scaled with probability α_2). Once the search progress terminates, the option with the highest probability (e.g., MAX with solid arrow) is used in the final architecture to be evaluated on testing set.

Compared with RL-based search, differentiable search based algorithm is less popular in the GNN-NAS literature. PDNAS (Zhao et al, 2020g) relaxes the discrete search space into a continuous one by employing the Gumbel-sigmoid, enabling optimization via gradient descent. POSE focuses on searching the propagation function, whose discrete search space is relaxed by a softmax approximation.

17.3.5 Efficient Performance Estimation

To solve the bi-level optimization problem of AutoGNN, all the above search algorithms share a common two-stage working pipeline: sampling a new design and adjusting the search algorithm based on the performance estimation of the new design at each step. Once the search progress terminates, the optimal design with the highest model performance will be treated as the desired solution to the concerned optimization problem. Therefore, an accurate performance estimation strategy is crucial to AutoGNN framework. The simplest way of performance estimation is to perform a standard training for each generated design, and then obtain the model performance on the split validation set. However, such an intuitive strategy is computationally expensive given the long search progress and massive graph datasets.

Parameter sharing is one of the efficient strategies to reduce the cost of performance estimation, which avoids training from scratch for each design. Parameter sharing is first proposed in ENAS (Pham et al, 2018) to force all designs to share weights to improve efficiency. A new design could be immediately estimated by reusing the weights well trained before. However, such a strategy cannot be directly adopted in GNN-NAS since the GNN architectures in search space may have weights with different dimensions or shapes. To tackle the challenge, recent work

modified the parameter sharing strategy to customize for GNNs. GraphNAS (Gao et al, 2020b) categorizes and stores the optimized weights based on their shapes, and applies the one with the same shape to the new design. After parameter sharing, AGNN (Zhou et al, 2019a) further uses a few training epochs to fully adapt the transferred weights to the new design. In the differentiable GNN-NAS frameworks, the parameter sharing is conducted naturally between GNN architectures sharing the common computation options (Zhao et al, 2020g; Ding et al, 2020a).

17.4 Future Directions

We have reviewed various search spaces and search algorithms. Although some initial AutoGNN efforts have been paid, compared with the rapid development of AutoML in computer vision, AutoGNN is still in the preliminary research stage. In this section, we discuss several future directions, especially for research on GNN-NAS.

- **Search space.** The design of architecture search space is the most important portion in GNN-NAS framework. An appropriate search space should be comprehensive by covering the key architecture dimensions and their state-of-the-art primitive options to guarantee the performance of searched architecture for any given task. Besides, the search space should be compact by incorporating a moderate number of powerful options to make the search progress efficient. However, most of the existing architecture search spaces are constructed based on vanilla GCN and GAT, failing to consider the recent GNN developments. For example, graph pooling (Ying et al, 2018c; Gao and Ji, 2019; Lee et al, 2019b; Zhou et al, 2020e) has attracted increasing research interests to enable encoding the graph structures hierarchically. Based on the wide variety of pooling algorithms, the corresponding hierarchical GNN architectures gradually shrink the graph size and enhance the neighborhood reception field, empirically improving the downstream graph analysis tasks. Furthermore, a series of novel graph convolution mechanisms have been proposed from different perspectives, such as neighbor-sampling methods to accelerate computation (Hamilton et al, 2017b; Chen et al, 2018c; Zeng et al, 2020a), and PageRank based graph convolutions to extend neighborhood size (Klicpera et al, 2019a,a; Bojchevski et al, 2020b). With the development in GNN community, it is crucial to update the search space to subsume the state-of-the-art models.
- **Deep graph neural networks.** All the existing search spaces are implemented with shallow GNN architectures, i.e., the number of graph convolutional layers $l_{gc} \leq 10$. Unlike the widely adopted deep neural networks (e.g., CNNs and transformers) in computer vision and natural language processing, GNN architectures are usually limited with less than 3 layers (Kipf and Welling, 2017b; Veličković et al, 2018). As the layer number increases, the node representations will converge to indistinguishable vectors due to the recursive neighborhood aggregation and non-linear activation (Li et al, 2018b; Oono and Suzuki, 2020). Such phenomenon is recognized as the over-smoothing issue (NT and Maehara,

2019), which prevents the construction of deep GNNs from modeling the dependencies to high-order neighbors. Recently, many efforts have been proposed to relieve the over-smoothing issue and construct deep GNNs, including embedding normalization (Zhao and Akoglu, 2019; Zhou et al, 2020d; Ioffe and Szegedy, 2015), residual connection (Li et al, 2019c, 2018b; Chen et al, 2020l; Klicpera et al, 2019a), and random data augmentation (Rong et al, 2020b; Feng et al, 2020). However, most of them only achieve comparable or even worse performance compared to their corresponding shallow models. By incorporating these new techniques into the search space, GNN-NAS could effectively combine them and identify the novel deep GNN model, which unleashes the deep learning power for graph analytics.

- **Applications to emerging graph analysis tasks.** One limitation of GNN-NAS frameworks in literature is that they are usually evaluated on a few benchmark datasets, such as Cora, Citeseer, and Pubmed for node classification (Yang et al, 2016b). However, the graph-structured data is ubiquitous, and the novel graph analysis tasks are always emerging in real-world applications, such as property prediction of biochemical molecules (i.e., graph classification) (Zitnik and Leskovec, 2017; Aynaz Taheri, 2018; Gilmer et al, 2017; Jiang and Balaprakash, 2020), item/friend recommendation in social networks (i.e., link prediction) (Ying et al, 2018b; Monti et al, 2017; He et al, 2020), and circuit design (i.e., graph generation) (Wang et al, 2020b; Li et al, 2020h; Zhang et al, 2019d). The surge of novel tasks poses significant challenges for the future search of well-performing architectures in GNN-NAS, due to the diverse data characteristics and objectives of tasks and the expensive searching cost. On one hand, since the new tasks may do not resemble any of the existing benchmarks, the search space has to be re-constructed by considering their specific data characteristics. For example, in the knowledge graph with informative edge attributes, the micro-architecture search space needs to incorporate edge-aware graph convolutional layers to guarantee a desired model performance (Schlichtkrull et al, 2018; Shang et al, 2019). On the other hand, if the new tasks are similar to the existing ones, the search algorithms could re-exploit the best architectures discovered before to accelerate the search progress in the new tasks. For example, one can simply initialize the search progress with these sophisticated architectures and uses several epochs to explore the potentially good ones within a small region. Especially for the massive graphs with a large volume of nodes and edges, the reuse of well-performing architectures from similar tasks could significantly save the computation cost. The research challenge is how to quantify the similarities between the different graph-structured data.

Acknowledgements

This work is, in part, supported by NSF (#IIS-1750074 and #IIS-1718840). The views, opinions, and/or findings contained in this paper are those of the authors and should not be interpreted as representing any funding agencies.

Editor's Notes: Automated graph neural networks introduce automated machine learning to tackle the problem of GNN neural architecture search and hyperparameter search. Hence, this chapter is orthogonal to most of the other chapters in this book, which generally depend on expert experience to design specific models and tune hyperparameters. Neural architecture search space contains the components of manually designed models, such as kinds of aggregators introduced in chapter 4 and chapter 5. Automated graph neural networks support common graph analysis tasks, such as node classification (chapter 4), graph classification (chapter 9), and link prediction (chapter 10).