ADAPTIVE GRAPH CAPSULE CONVOLUTIONAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

In recent years, many studies utilize Convolutional Neural Networks (CNNs) to deal with non-grid graph data, known as Graph Convolutional Networks (GCNs). However, there exist two main limitations of the prevalent GCNs. First, GCNs have a latent information loss problem since they use scalar-valued neurons rather than vector-valued ones to iterate through graph convolutions. Second, GCNs are presented statically with fixed architectures during training, which would restrict their representation power. To tackle these two issues, based on a capsule GCN model (CapsGNN) which encodes node embeddings as vectors, we propose Adaptive Graph Capsule Convolutional Networks (AdaGCCN) to adaptively adjust the model architecture at runtime. Specifically, we leverage Reinforcement Learning (RL) to design an assistant module for continuously selecting the optimal modification to the model structure through the whole training process. Moreover, we determine the architecture search space through analyzing the impacts of model's depth and width. To mitigate the computation overhead brought by the assistant module, we then deploy multiple workers to compute in parallel on GPU. Evaluations show that AdaGCCN achieves SOTA accuracy results and outperforms CapsGNN almost on all datasets in both bioinformatics and social fields. We also conduct experiments to indicate the efficiency of the paralleling strategy.

1 INTRODUCTION

Graph-structured data is ubiquitous in real-world scenarios such as biological networks, citation networks, recommender systems, and social networks. It has seen a surge in research on extending deep learning methods like Convolutional Neural Networks (CNNs) (LeCun & Bengio, 1998) or Recurrent Neural Networks (RNNs) (Mikolov et al., 2011) to process tasks on graphs. Graph Neural Networks (GNNs) were first proposed in (Gori et al., 2005) and further elaborated in (Scarselli et al., 2009) to handle graphs based on recursive neural networks (RecGNNs). RecGNNs learn to represent nodes by constantly exchanging neighbor information until reaching a stable equilibrium, which is computationally expensive. Encouraged by the success of CNNs, many studies (Bruna et al., 2014; Henaff et al., 2015; Defferrard et al., 2016; Kipf & Welling, 2017) borrowed the idea of convolutions and redefined them for graph data. These convolutional networks on graphs are known as Graph Convolutional Networks (GCNs) by stacking multiple graph convolutional layers to present high-level graph representation. Moreover, GCNs can save much time in computing compared to RecGNNs. In this paper, we focus on the research on GCNs.

SOTA GCNs methods (Zhang et al., 2018; Verma & Zhang, 2018) generate graph embeddings in the form of scalar, which may not help capture enough node features. Therefore, it is of great necessity to improve the information extraction ability of current GCNs. In 2011, (Hinton et al., 2011) pointed out that traditional pooling mechanism (Scherer et al., 2010) on CNNs may suffer from the information loss problem. Later (Sabour et al., 2017) proposed a novel neural network called CapsNet without using pooling. CapsNet encodes the features of images as vectors and uses a dynamic routing algorithm to obtain the right local-global relationship. Owing to its capability to capture inner relations among entities, CapsNet has been converged with GCNs to grasp higher quality graph information (Verma & Zhang, 2018; Xinyi & Chen, 2019).

CapsGNN (Xinyi & Chen, 2019) extracts multi-scale node features from different convolutional layers and represents the extracted features in the form of capsules. Compared to classic GCN

models, CapsGNN has been proven to have better representation ability on graph data. However, CapsGNN still uses a static model structure to conduct training, which inherently restricts its ability to cope with the enhancing abstraction level of graph information during the whole training process. Therefore, it is crucial to explore a method for dynamically adjusting the structure of CapsGNN to improve its classification performance.

Many research efforts (He et al., 2016; Szegedy et al., 2016; Huang et al., 2017; Szegedy et al., 2017; Li et al., 2018; Rong et al., 2019) have demonstrated that the number of convolutional layers (i.e., depth, denoted by D) and the number of neurons in each layer (i.e., width, denoted by W) impact significantly on the performance of neural networks. However, these methods are often inefficient in training due to overly complex model structures. In this paper, we simplify the architecture search problem and focus on the exploration of the best depth-width setting on particular dataset, considering the balance between efficiency and accuracy.

To evaluate the effects of the depth and width in CapsGNN, we change them to observe the performance of the model on two graph datasets, ENZYMES and IMDB-MULTI (Kersting et al., 2016), in bioinformatics and social fields, respectively. It is shown in Figure 1 that if the model has small D and W, it can not sufficiently extract the features of graphs. On the contrary, when the model structure is too complicated, overfitting occurs frequently. Thus, CapsGNN is sensitive to changes in depth and width just because of the vectorized graph representation in it. Since the impacts of D and W are distinct on different datasets, it is necessary to adaptively adjust the model architecture according to the latent characteristics of particular workloads.

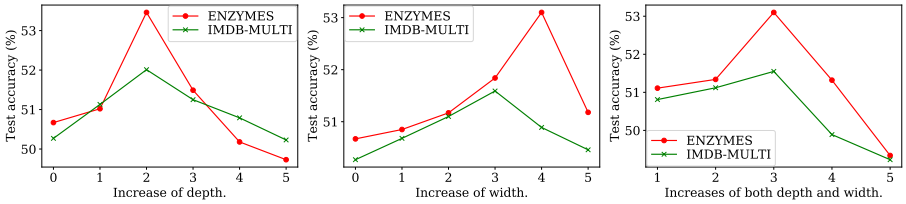


Figure 1: Results of classification accuracy when increasing D and W. The third subfigure depicts the changes of both D and W, and the x-axis represents the increase of either of them (for simplicity, the growth scales of D and W are set the same here).

With respect to the search for the optimal structure of a model, there has been increasing interest in designing automatic methods in Neural Architecture Search (NAS) (Zoph & Le, 2017; Liu et al., 2018; Luo et al., 2018; Real et al., 2019; Pham et al., 2018). There exist mainly three kinds of techniques to search for the best model architecture in NAS, i.e., Reinforcement Learning (RL) based, Evolution Algorithm (EA) based, and gradient-based methods. RL-based methods (Zoph & Le, 2017; Pham et al., 2018; Zoph et al., 2018) use RNN as the controller to train different architectures and select the optimal one according to their validation accuracy results. EA-based methods (Real et al., 2017; Xie & Yuille, 2017; Real et al., 2019; Shu et al., 2019) utilize the evolutionary algorithm to search for superior model structures from a set of initialized candidate networks. Gradient-based methods (Liu et al., 2019; Wu et al., 2019; Xie et al., 2019) construct a SuperNet and leverage the attention mechanism to remove weak connections after searching. The former two methods can be regarded as offline and are often time-consuming. Although the third method is an online method and shows good efficiency due to gradient descent optimization, it lacks variety in searching for the optimal architecture.

To achieve the balance between efficiency and accuracy, we propose Adaptive Graph Capsule Convolutional Networks (AdaGCCN) to address the issues discussed above. We utilize Reinforcement Learning (RL) to design an online assistant module for evaluating different changes to depth and width. Differing from the RL agent in an RL-based NAS task that considers the choices of various model structures as actions, we move the RL procedure into only one full training and choose one action (i.e., one alteration to the model structure) in a sliding epoch window at one time, according to not only the accuracy results on the validation data set but also the rate of reduction in training loss. However, the introduction of the assistant module would result in extra computational overhead. To accelerate the selection process described above, we further assign multiple workers to train in parallel on the GPU. Three main contributions are made in this paper:

- We optimize the model architecture at runtime by tuning depth and width of convolutional layers by observing both the change in validation accuracy and the reduction speed of training loss after each sliding epoch window. Following this scheme, we can dynamically refine our model while sharing the weights parameters learned at previous training stages.
- The assistant module is formalized as a Reinforcement Learning (RL) agent. We define the depth-width setting of graph convolutions as a state and consider the changes of the setting as actions. We leverage the RL mechanism to update the rewards corresponding to different actions.
- To simplify the search space of the assistant module, we discuss the adequacy of only adjusting depth and width of the convolution operations. To ease the computation burden that arises from the assistant module, we split the whole validation process into multiple tasks to run them synchronously.

We organize the rest of the paper as follows. Section 2 introduces the background of this work. Section 3 elaborates the fundamental architecture and implementation of AdaGCCN. In Section 4, we evaluate the classification performance of our model against the baseline methods. We discuss related work in Section 5 and conclude this paper in Section 6.

2 PRELIMINARIES

Here, we give a brief revisit to common GCNs and a capsule-based GCN, i.e., CapsGNN.

2.1 GRAPH CONVOLUTIONAL NETWORKS

A graph can be represented by $G = (V, X, A)$, in which $V = \{v_1, v_2, ..., v_N\}$ (N is the number of nodes) is the set of nodes in the graph, $A \in \{0, 1\}^{N \times N}$ is the adjacency matrix of G and $X \in R^{N \times d}$ represents the features of each node (d is the number of feature dimensions).

GCNs, inspired by CNNs, extract richer information of one node by aggregating features of the nodes from its neighborhood. At each layer of GCNs, each node and its neighbors are operated through convolutions. Then an activation function is applied to return the updated representation of each node. The procedure in GCNs can be written as:

$$Z^{l+1} = A^{'} Z^{l} W^{l}, X^{l+1} = f(Z^{l+1}) \tag{1}$$

where $Z^l \in R^{N \times d_l}$ implies node features at layer l ($Z^0 = X$), $W^l \in R^{d_l \times d_{l+1}}$ is a trainable weights matrix which serves as a convolution filter, $A^{'} \in R^{N \times N}$ is the regularized form of the adjacency matrix A, and f is a nonlinear activation function.

2.2 CAPSULE GRAPH NEURAL NETWORK

Although classic CNNs perform decent classification results in image processing, they face a severe robust problem. For example, when training with human faces, CNNs could learn good statistical representations of them, but can not tell whether the relative positions of different parts of a face are correct. The reason behind this is mainly due to the pooling mechanism in CNNs which extracts information through down-sampling, thus causing the losses on some essential features of images.

Unlike extracting image features as scalar values in CNNs, CapsNet was proposed (Hinton et al., 2011; Sabour et al., 2017; Hinton et al., 2018) to build vector-based capsules to learn the inherent part-whole relationships in images. Inspired by CapsNet, CapsGNN (Xinyi & Chen, 2019) fuses the capsule mechanism with GCNs to handle graph data. CapsGNN extracts multi-dimension node features from different graph convolutional layers to serve as basic node capsules. Then it combines the attention mechanism and dynamic routing to generate higher-level graph capsules. To this end, each graph can be abstracted as multiple graph capsules, and different graph capsules represent the properties of the graph from different aspects. At last, CapsGNN uses dynamic routing again to produce class capsules which are applied to do graph classifications. However, like traditional GCNs models, CapsGNN does not try to adjust the model architecture when more fine-grained graph information needs to be learned as training proceeds, which leaves potential in enhancing its representation power.

3 OVERVIEW OF ADAGCCN

In this section, we outline the overall design of AdaGCCN (depicted in Figure 2). We first introduce the *assistant module* (AM) and the principle for determining the search space in it. Then we elaborate on how AM dynamically changes the model structure to extract higher-quality graph representations. Finally, we describe the parallel execution in AdaGCCN which accelerates the search process in AM.

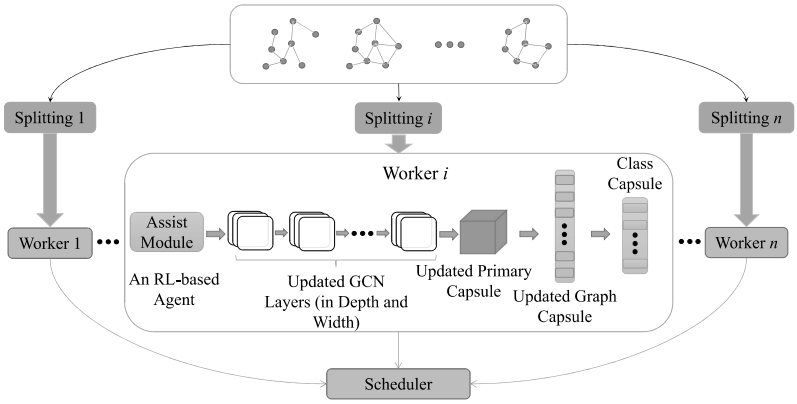


Figure 2: AdaGCCN Architecture.

3.1 THE ASSISTANT MODULE

We build an AM to select the changes to model structure from different branches. Each branch indicates an alteration to D and W. The changes of D and W are represented by ΔD and ΔW, respectively. We formalize the procedure of the assistant module as an RL procedure. RL usually uses an agent to interact with an environment by choosing actions at different periods. In AdaGCCN, the depth-width setting at the t-th ($t \geq 1$) training epoch is treated as the state in the RL agent (denoted by (D_t, W_t)). We specify two lists, the ΔD_list and the ΔW_list, for the changes of (D_t, W_t). The length of the ΔD_list and the ΔW_list are len_1 and len_2, respectively. Note that we focus on the impacts of D and W in this paper rather than the complex situation in common NAS, len_1 and len_2 can not be very large with considering both the overfitting problem and computation efficiency. An action a_i^t ($1 \leq i \leq m$) is defined as one of the m ($m = len_1 * len_2$) choices in changing (D_t, W_t) at the t-th epoch:

$$a_i^t = [\Delta D\_list[p], \Delta W\_list[q]],\ 0 \leq p \leq len_1 - 1,\ 0 \leq q \leq len_2 - 1. \tag{2}$$

3.2 ADAPTIVE D-W TUNING

There are two key issues we need to address upon building an AM:

- **Q1:** how to select the optimal ΔD and ΔW during training?
- **Q2:** how to share the model parameters derived at previous training epochs when D and W are updated?

Through training, we record the training loss (denoted by $loss_t$) and the accuracy on the validation hold (denoted by $valacc_t$) after epoch t ends. Unlike the original RL method, we can not predict which action performs the best before running the model with updated D and W. Thus we design a reward evaluation function to compare the loss reduction rate and the variation of validation accuracy over a sliding epoch window. Specifically, the length of this sliding window is set to three, and then the loss decrease speed is calculated within three consecutive epochs. We could broaden the size of the sliding window according to specific requirements. When the action with the maximal Q-value is obtained, we take it to update the model structure for training in the next sliding window.

We let $Q(a_i^t)$ represent the reward of taking the action a_i^t. The goal of AM is to constantly select the action a_{i*}^t with the maximal reward. The update on $Q(a_i^t)$ is determined whether a_i^t is chosen at

Algorithm 1: The updating strategy in the assistant module.

Input: The number of training epochs: E, the length of the sliding epoch window: 3, the reward discount rate: γ ($0 \leq \gamma \leq 0.5$), the threshold reflecting the reward for the loss decrease at the last two training epochs: θ ($0 \leq \theta \leq 0.5$), the ΔD_list (length: $len1$) for updating D, the ΔW_list (length: $len2$) for updating W, the set of losses during E epochs: $\{loss_t\}$, the set of accuracies on the validation data set during E epochs: $\{valacc_t\}$, the set of Q-values of m ($m = len1 * len2$) actions: $\{Q_i\}$ ($1 \leq i \leq m$).

Output: Optimal ΔD and ΔW.

for $3 \leq t < E$ **do**
  if $t \ mod \ 3 = 0$ **then**
    $\Delta loss_1 \leftarrow loss_{t-2} - loss_{t-1}$, $\Delta loss_2 \leftarrow loss_{t-3} - loss_{t-2}$;
    $eval(a_{i^*}^t) \leftarrow \Delta loss_1 / \Delta loss_2$;
    $\Delta valacc_1 \leftarrow valacc_{t-1} - valacc_{t-4}$, $\Delta valacc_2 \leftarrow valacc_{t-1} - valacc_{t-2}$;
    $\Delta valacc = \Delta valacc_1 + \Delta valacc_2$;
    if $\Delta valacc_1 * \Delta valacc_2 > 0$ **then**
      if $\Delta loss_1 * \Delta loss_2 > 0$ **then**
        $flag \leftarrow (Max(\Delta loss_1, 0) - Min(\Delta loss_1, 0))/\Delta loss_1$;
        if $eval(a_{i^*}^t) \geq 1$ **then**
          $Q[a_{i^*}^t] += flag * (1 - \gamma) * (eval(a_{i^*}^t) + 10 * \Delta valacc)$;
        else
          if $0 < \Delta loss_1 < \theta$ **then**
            $Q[a_{i^*}^t] -= (1 - \gamma) * (10 * eval(a_{i^*}^t) + 10 * \Delta valacc)$;
          else
            $Q[a_{i^*}^t] += \gamma * eval(a_{i^*}^t)$;
            if $\Delta loss_1 > \theta$ **then**
              $Q[a_{i^*}^t] += 10 * \gamma * \Delta valacc$;
      else
        if $eval(a_{i^*}^t) \leq -1$ **then**
          $Q[a_{i^*}^t] -= flag * (1 - \gamma) * (eval(a_{i^*}^t) + 10 * \Delta valacc)$;
        else
          $Q[a_{i^*}^t] += \gamma * (10 * eval(a_{i^*}^t))$;
          if $\Delta loss_1 < 0$ **then**
            $Q[a_{i^*}^t] += \gamma * (10 * \Delta valacc - 9 * eval(a_{i^*}^t))$;
    else
      $Q[a_{i^*}^t] += (1 - \gamma) * (10 * (Min(\Delta valacc_1, 0) + Min(\Delta valacc_2, 0)))$;
  $Q\_maxindex \leftarrow$ index of $Max\{Q_i\}$;
  $action\_maxindex \leftarrow actions[Q\_maxindex]$;
  return $\Delta D = action\_maxindex[0]$, $\Delta W = action\_maxindex[1]$;

epoch $t - s$ (s is the length of the sliding epoch window and we let $s = 3$ in this paper). Suppose we pick $a_{i^*}^{t-3} = [\Delta D\_list[p^*], \Delta W\_list[q^*]]$ at the $(t-3)$-th epoch, then we revise $Q(a_{i^*}^t)$ ($a_{i^*}^t = a_{i^*}^{t-3}$) by Algorithm 1 where $eval(a_{i^*}^t)$ describes the loss reduction speed at the last three epochs and $\Delta valacc$ implies how the validation accuracy varies within two consecutive sliding epoch windows. As to the actions that are not taken, the rewards of them would remain unchanged. Note that at the first three epochs, we random select an action $a_{i^*}^1$ from the action list, and initialize the Q-values of m actions to 0. After updating all the Q-values, if the maximal reward corresponds to more than one action, we randomly choose an action with the maximal Q-value. Once we acquire the updated ΔD and ΔW, D and W that AdaGCCN uses at epoch t are:

$$D_t = D_{t-3} + \Delta D\_list[p^*], \ W_t = W_{t-3} + \Delta W\_list[q^*]. \tag{3}$$

Suppose D and W are changed at the t-th epoch, the model would be reinitialized if we do not utilize the parameters produced at epoch $t-1$. In other words, the modifications to D and W would influence the shape of 2-D weight tensors in GCN layers. It urges us to consider how to maximize the retention of the weight parameters learned at the last epoch. Regarding the resampling techniques

in image processing, an interpolation solution called *bilinear interpolation* (BI) helps scale images using the distance weighted average of the four nearest pixel values to estimate a new pixel value. As opposed to other interpolation methods (e.g., nearest-neighbor interpolation) which would make some pixels appear larger than others in the resized image, BI reduces some of the visual distortions in image transformation. Thus we adopt BI in this paper to reshape the weight matrices upon tuning the D and W of the model without losing much information extracted in previous iterations.

3.3 Parallel Processing

We follow the same validation method as in CapsGNN, i.e., the 10-fold cross validation, which splits the original dataset into ten groups. Each time we take eight groups as train holds, one group as the validation hold, and the remained one group as the test hold. In the end, we would obtain 10 test accuracy results on 10 different splittings. We take the average of these results as the final test accuracy.

Since the ten splittings are independent to each other, to alleviate the computation overhead brought by AM, we dispatch these splittings to ten workers on one GPU card and explore a scheduling strategy (see Algorithm 2 in Appendix A) for dynamically assigning an appropriate number of workers through monitoring the running status of them. Specifically, we observe that the device memory required by an individual worker almost keeps constant, so we could estimate the memory requirement of a worker through pre-run, and then launch multiple workers in parallel while their total memory is not larger than the device memory.

However, when processing multiple workers concurrently on a single dataset, sometimes GPU memory is not fully utilized. Thus we extend Algorithm 2 to do scheduling on multiple datasets, i.e., combinatorilly assigning workers on different datasets to maximize the utilization of GPU memory, shown in Algorithm 3 in Appendix A.

4 Evaluation

We conduct elaborate experiments in this section on ten typical datasets to evaluate the performance of AdaGCCN against the baseline models.

4.1 Preparations for Experiments

Baseline models. AdaGCCN is compared with four graph kernel algorithms, GK (Shervashidze et al., 2009), WL (Shervashidze et al., 2011), DGK (Yanardag & Vishwanathan, 2015), and AWE (Ivanov & Burnaev, 2018)), four GNNs-based methods, PATCHY-SAN (PSCN) (Niepert et al., 2016), DGCNN (Zhang et al., 2018), GIN (Xu et al., 2019), and SOM-GCNN (Pasa et al., 2020), and two capsule-based GNNs methods, GCAPS-CNN (Verma & Zhang, 2018) and CapsGNN (Xinyi & Chen, 2019). We also take the results reported in (Errica et al., 2020) as a baseline, namely FGNN, which proposed a fair validation method of GNNs for graph classification.

Experimental settings. We evaluate AdaGCCN on a machine equipped with dual 2.40 GHz Intel Xeon Gold 6240R processors (24 cores in total), 256 GB main memory, and 1 NVIDIA Tesla V100 GPU (32 GB memory). The installed operating system is CentOS 7.5.1804, using CUDA 11.2 and cuDNN 7.6.5. PyTorch 1.8 (Paszke et al., 2019) and Python 3.6.5 are used for training.

Datasets. Since our work focuses on graph-level classifications (not node-level classifications), we experiment on the datasets frequently used in graph classification tasks including five biological datasets, ENZYMES, D&D, MUTAG, NCI1, and PROTEINS and five social network datasets, COLLAB, IMDB-BINARY, IMDB-MULTI, REDDIT-M5K, and REDDIT-M12K. The details of these datasets are described in Table 1.

4.2 AdaGCCN Performance

We first compare the classification accuracy between AdaGCCN and the baseline methods. Then we demonstrate the efficiency of the scheduling strategies applied in the parallel process.

Table 1: Datasets Information (Kersting et al., 2016).

Description	MUTAG	ENZYMES	NCI1	PROTEINS	D&D	IMDB-B	IMDB-M	COLLAB	REDDIT-M5K	REDDIT-M12K
Type	Bio	Bio	Bio	Bio	Bio	Social	Social	Social	Social	Social
Graphs	188	600	4110	1113	1178	1000	1500	5000	4999	11929
Classes	2	6	2	2	2	2	3	3	5	11
Nodes Avg.	17.93	32.46	29.87	39.06	284.31	19.77	13	74.49	508.5	391.4
Edges Avg.	19.79	63.14	32.30	72.81	715.65	193.06	131.87	4914.99	1189.74	456.89
Node Labels	7	6	23	4	82	-	-	-	-	-

4.2.1 THE EFFECTIVENESS OF AM

Based on the neural network method, AdaGCCN takes advantage of the multi-dimension representation ability of capsules, and further explores to strengthen the information extraction ability of the model by changing depth and width of graph convolutional layers. We reproduce CapsGNN in this paper and take the test accuracies reported in other baseline works for the comparison with our model.

It is shown in Table 2 that our model outperforms CapsGNN on 6 out of 7 datasets by up to 6.57% on ENZYMES except for the performance on PROTEINS. Note that the graphs on PROTEINS are more complex in inner relationships, and the ΔD_list and the ΔW_list in current AM are manually set, thus the extraction ability of AdaGCCN would be restricted somehow. However, the main point in this paper is to propose an RL-based methodology in adaptively adjusting the model structure to improve its representation ability on various workloads, some hyper-parameters defined in the assistant module could be tuned to present better model performance in the future. Besides, AdaGCCN achieves state-of-the-art results almost on all datasets including biological and social graphs, which further demonstrates the effectiveness of adjusting D and W during only one full training.

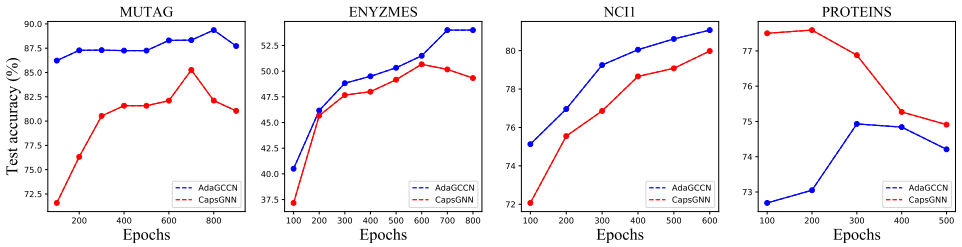
Table 2: Test accuracies on the benchmark datasets. The bold values represent the methods with top-2 performance on each dataset.

Model	MUTAG	ENZYMES	NCI1	PROTEINS	IMDB-BINARY	IMDB-MULTI	COLLAB
GK	81.58 ± 2.11	32.7 ± 1.20	62.49 ± 0.27	71.67 ± 0.55	65.87 ± 0.98	43.89 ± 0.38	72.84 ± 0.28
WL	82.05 ± 0.36	52.22 ± 1.26	82.19 ± 0.18	74.68 ± 0.49	73.40 ± 4.63	49.33 ± 4.75	79.02 ± 1.77
DGK	87.44 ± 2.72	53.43 ± 0.91	80.31 ± 0.46	75.68 ± 0.54	66.96 ± 0.56	44.55 ± 0.52	73.09 ± 0.25
AWE	87.87 ± 9.76	35.77 ± 5.93	-	-	$\mathbf{74.45 \pm 5.83}$	51.54 ± 3.61	73.93 ± 1.94
PSCN	88.95 ± 4.37	-	76.34 ± 1.68	75.00 ± 2.51	71.00 ± 2.29	45.23 ± 2.84	72.60 ± 2.15
DGCNN	85.83 ± 1.66	51.00 ± 7.29	74.44 ± 0.47	75.54 ± 0.94	70.03 ± 0.86	47.83 ± 0.85	73.36 ± 0.49
GIN	$\mathbf{89.40 \pm 5.60}$	-	$\mathbf{82.70 \pm 1.70}$	76.20 ± 2.80	$\mathbf{75.10 \pm 5.10}$	$\mathbf{52.30 \pm 2.80}$	$\mathbf{80.20 \pm 1.90}$
SOM-GCNN	-	50.01 ± 2.92	82.32 ± 0.52	75.22 ± 0.61	-	-	-
FGNN	-	$\mathbf{65.17 \pm 6.00}$	69.83 ± 2.20	75.75 ± 3.70	70.77 ± 5.00	49.09 ± 3.50	70.19 ± 1.50
GCAPS-CNN	-	$\mathbf{61.83 \pm 5.39}$	$\mathbf{82.72 \pm 2.38}$	$\mathbf{76.40 \pm 4.17}$	71.69 ± 3.40	48.50 ± 4.10	77.71 ± 2.51
CapsGNN	85.26 ± 5.43	50.67 ± 6.52	79.98 ± 1.69	$\mathbf{77.59 \pm 2.85}$	72.30 ± 4.57	50.27 ± 2.59	77.24 ± 2.79
AdaGCCN	$\mathbf{89.36 \pm 1.97}$	54.01 ± 2.78	81.07 ± 1.13	74.93 ± 1.54	74.20 ± 2.23	$\mathbf{52.27 \pm 1.76}$	$\mathbf{80.70 \pm 1.59}$

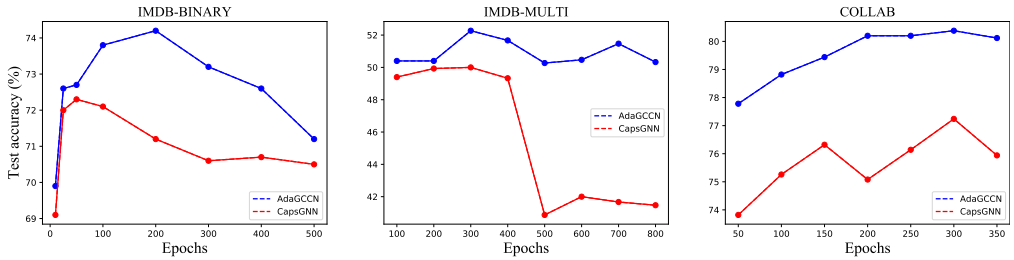
Figure 3 depicts the change of accuracies on the test fold when training with CapsGNN or AdaGCCN. Since the assistant module in AdaGCCN evaluates the loss reduction speed on the train fold and the accuracy change on the validation fold, as the training process goes on, it helps to select the ΔD and ΔW with the biggest accumulated reward. It is observed in the figure that AdaGCCN presents higher test accuracy than CapsGNN at arbitrary intermediate training stages. Besides, the dynamic adjustment in our model does not do any harm to its convergency compared to static CapsGNN. Note that the number of training epochs is set to be large on all datasets to ensure either CapsGNN or AdaGCCN achieves optimal classification performance during training.

4.2.2 PARALLEL EXECUTION

Taking into consideration the computational cost brought by the assistant module, we adaptively adjust the number of workers at runtime. It implies in Figure 4 that when with Algorithm 2, the parallel AdaGCCN on all datasets could averagely speedup the sequential AdaGCCN by up to 2.99×. To make full use of GPU memory, we adopt Algorithm 3 to let workers on different datasets run combinatorilly, which further decreases the computation time when handling the datasets one at a time, e.g., by 16.7% within 300 training epochs. Besides, the memory consumption of CapsGNN and AdaGCCN are displayed in Table 4 in Appendix A.

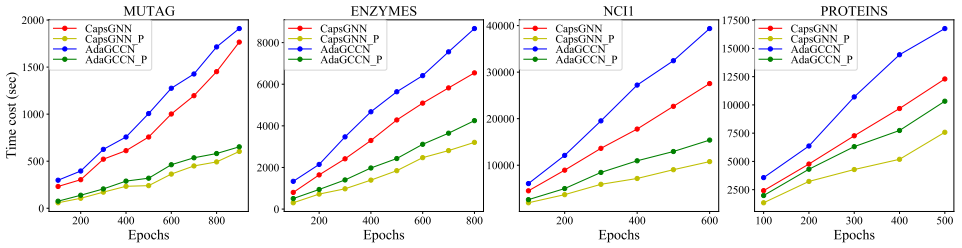


(a) Bioinformatics Datasets.

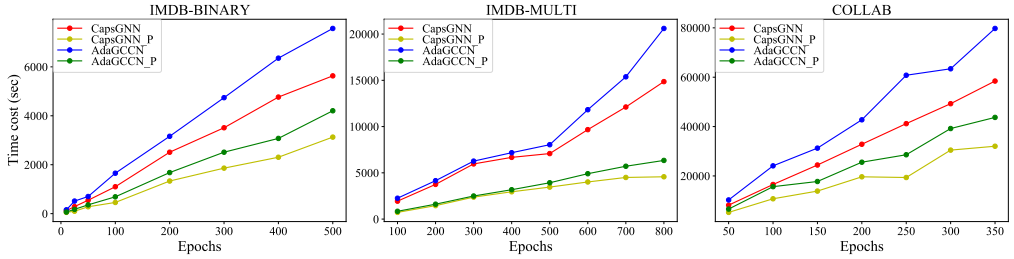


(b) Social Datasets.

Figure 3: The comparison of the test accuracies between CapsGNN and AdaGCCN at different training stages.



(a) Bioinformatics Datasets.



(b) Social Datasets.

Figure 4: Time costs of CapsGNN, AdaGCCN, and their parallel versions, i.e., CapsGNN_P and AdaGCCN_P.

4.3 Discussion

4.3.1 Generality and Explainability of the Assistant Module (AM)

The NAS nature of the RL-based AM enables it to be applied to other GNNs. We add an experiment on the effect of AM on GIN, the model with the best classification performance in all the baseline methods, and record its test accuracy results on different datasets in Table 3. Moreover, to show the explainability of AdaGCCN on different datasets, we compare the D and W selected by the assistant module with the original depth and width in CapsGNN in Table 5 in Appendix A.

Table 3: Test accuracies on GIN with or without the assistant module. GIN_AM means GIN applying AM in it.

Model	MUTAG	ENZYMES	NCI1	PROTEINS	IMDB-BINARY	IMDB-MULTI	COLLAB
GIN	89.40 ± 5.60	-	82.70 ± 1.70	76.20 ± 2.80	75.10 ± 5.10	52.30 ± 2.80	80.20 ± 1.90
GIN_AM	91.32 ± 2.53	-	83.24 ± 1.26	77.69 ± 2.52	76.27 ± 4.63	54.15 ± 2.37	81.64 ± 1.26

4.3.2 LIMITATIONS OF ADAGCCN

In our experiments, both CapsGNN and AdaGCCN can not run successfully on D&D, REDDIT-M5K, and REDDIT-M12K either due to GPU memory limitation or heavy time overhead. There are two reasons behind this phenomenon. First, CapsGNN and AdaGCCN have more complex structures than normal GNNs due to the vector-based capsule modules in them, resulting in more model parameters generated during training that the GPU can not hold. Second, both CapsGNN and our model set the batch size to 32 on small graphs, which is not appropriate on large ones. If we use a smaller batch size on D&D and REDDIT, the accuracy results would be affected compared to adopting a larger batch size, and the time costs would be unbearable, e.g., more than 3 GPU days on REDDIT-M12K. Here we provide probable directions in optimizing our work, e.g., designing fine-grained parallelism to improve the utilization of GPU, or refining the rewards evaluation function in the assistant module to increase the efficiency of the search process.

5 RELATED WORK

Following the success of neural networks on data with grid structures (LeCun & Bengio, 1998; Mikolov et al., 2011), considerable research interests have been devoted to non-grid graph data, i.e., Graph Neural Networks (GNNs) (Gori et al., 2005), especially Graph Convolutional Networks (GCNs) (Bruna et al., 2014; Henaff et al., 2015). GNNs have already obtained remarkable achievements in various tasks, e.g., graph classification (Defferrard et al., 2016), link prediction (Zhang & Chen, 2018), and node classification (Kipf & Welling, 2017). GCNs inherits the convolutional operations in CNNs, however, they are unable to learn graphs sufficiently which contains multiple attributes of nodes and complicated inner connections.

CapsNet was proposed by Hinton's team (Hinton et al., 2011) and improved by them (Sabour et al., 2017; Hinton et al., 2018) to represent local-global features of images. Inspired by the promising explainability of CapsNet, some studies (Verma & Zhang, 2018; Mallea et al., 2019; Xinyi & Chen, 2019) combine GCNs and CapsNet to extract multi-scale information in graphs. However, the static structure employed in these capsule-based GCNs would restrict their representation ability, which motivates us to explore dynamic adjustment of model structures at runtime.

The design of neural network structure is turning from manual efforts into automatic machine search. The landmark event of this process occurred in the paper (Zoph & Le, 2017) published by Google, which leverages reinforcement learning to search the optimal neural network structure, known as neural architecture search (NAS). Later, great efforts (Zoph & Le, 2017; Zoph et al., 2018; Chen et al., 2018; Gao et al., 2020) have been made to seek for the optimal architectures of CNNs, RNNs, or GNNs within a pre-determined search space according to the validation accuracy results. Although NAS has achieved impressive performance, the design of the search strategy is complicated and introduces numerous additional complete training processes.

6 CONCLUSION

In this work, we propose an adaptive framework AdaGCCN. We implement AdaGCCN by adjusting the model structure through an RL-based searching process, which improves the explainability of the model by finding the optimal depth and width of convolutions. To decrease the computation burden caused by the proposed assistant module, we assign multiple workers to execute in parallel. Once the device could not hold all the workers in it or has unused memory, AdaGCCN processes the workers on multiple datasets combinatorilly by monitoring the launches and exits of them. We demonstrate through experiments that AdaGCCN can adaptively refine its representation ability on different workloads, considering the trade-off between efficiency and accuracy.

REFERENCES

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann Lecun. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR)*, 2014.

Liang-Chieh Chen, Maxwell D. Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jonathon Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 8713–8724, 2018.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 3844–3852, 2016.

Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. In *International Conference on Learning Representations (ICLR)*, 2020.

Yang Gao, Hong Yang, Peng Zhang, Chuan Zhou, and Yue Hu. Graph neural architecture search. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1403–1409, 2020.

Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *IEEE International Joint Conference on Neural Networks (IJCNN)*, volume 2, pp. 729–734. IEEE, 2005.

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 (CVPR)*, pp. 770–778, 2016.

Mikael Henaff, Joan Bruna, and Yann LeCun. Deep convolutional networks on graph-structured data. *ArXiv*, abs/1506.05163, 2015.

Geoffrey E Hinton, Alex Krizhevsky, and Sida D Wang. Transforming auto-encoders. In *International Conference on Artificial Neural Networks (ICANN)*, pp. 44–51. Springer, 2011.

Geoffrey E Hinton, Sara Sabour, and Nicholas Frosst. Matrix capsules with EM routing. In *International conference on learning representations (ICLR)*, 2018.

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4700–4708, 2017.

Sergey Ivanov and Evgeny Burnaev. Anonymous walk embeddings. In *International Conference on Machine Learning (ICML)*, volume 80, pp. 2191–2200. PMLR, 2018.

Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016. `http://graphkernels.cs.tu-dortmund.de`.

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

Yann LeCun and Yoshua Bengio. *Convolutional Networks for Images, Speech, and Time Series*, pp. 255–258. MIT Press, 1998.

Qimai Li, Zhichao Han, and Xiao-Ming Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of the Thirty-Second Artificial Intelligence AAAI Conference (AAAI)*, pp. 3538–3545, 2018.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *European Conference on Computer Vision (ECCV)*, volume 11205, pp. 19–35. Springer, 2018.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *International Conference on Learning Representations (ICLR)*, 2019.

Renqian Luo, Fei Tian, Tao Qin, Enhong Chen, and Tie-Yan Liu. Neural architecture optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 7827–7838, 2018.

Marcelo Daniel Gutierrez Mallea, P. Meltzer, and P. Bentley. Capsule neural networks for graph classification using explicit tensorial graph representations. *ArXiv*, abs/1902.08399, 2019.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5528–5531. IEEE, 2011.

Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. In *International Conference on Machine Learning (ICML)*, pp. 2014–2023. JMLR.org, 2016.

Luca Pasa, Nicolò Navarin, and Alessandro Sperduti. SOM-based aggregation for graph convolutional neural networks. *Neural Computing and Applications*, pp. 1–20, 2020.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 8024–8035, 2019.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *International Conference on Machine Learning (ICML)*, volume 80, pp. 4092–4101. PMLR, 2018.

Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V. Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *International Conference on Machine Learning (ICML)*, volume 70, pp. 2902–2911. PMLR, 2017.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the Thirty-First Artificial Intelligence AAAI Conference (AAAI)*, pp. 4780–4789, 2019.

Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. DropEdge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations (ICLR)*, 2019.

Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 3856–3866, 2017.

F Scarselli, M Gori, AC Tsoi, M Hagenbuchner, and G Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2009.

Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International Conference on Artificial Neural Networksc (ICANN)*, pp. 92–101. Springer, 2010.

Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 5, pp. 488–495. JMLR.org, 2009.

Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

Han Shu, Yunhe Wang, Xu Jia, Kai Han, Hanting Chen, Chunjing Xu, Qi Tian, and Chang Xu. Co-evolutionary compression for unpaired image translation. In *International Conference on Computer Vision (ICCV)*, pp. 3234–3243, 2019.

Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

Christian Szegedy, S. Ioffe, V. Vanhoucke, and Alexander Amir Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the Thirty-First Artificial Intelligence AAAI Conference (AAAI)*, pp. 4278–4284, 2017.

Saurabh Verma and Zhi-Li Zhang. Graph capsule convolutional neural networks. In *Joint ICML and IJCAI Workshop on Computational Biology*, 2018.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. FBNet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 10734–10742, 2019.

Lingxi Xie and Alan L. Yuille. Genetic CNN. In *International Conference on Computer Vision (ICCV)*, pp. 1388–1397, 2017.

Sirui Xie, Hehui Zheng, Chunxiao Liu, and Liang Lin. SNAS: stochastic neural architecture search. In *International Conference on Learning Representations (ICLR)*, 2019.

Zhang Xinyi and Lihui Chen. Capsule graph neural network. In *International conference on learning representations (ICLR)*, 2019.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 1365–1374. ACM, 2015.

Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5165–5175, 2018.

Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the Thirty-Second Artificial Intelligence AAAI Conference (AAAI)*, pp. 4438–4445, 2018.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2017.

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 (CVPR)*, pp. 8697–8710, 2018.

A Appendix

The comparison of the memory consumption when experimenting with CapsGNN and our model are recorded in Table 4. Using the parallel strategies proposed in Section 3.3, as shown in Algorithm 2 amd Algorithm 3, AdaGCCN can achieve the balance between efficiency and accuracy.

Table 4: Memory consumption of CapsGNN and AdaGCCN.

Model	MUTAG	ENZYMES	NCI1	PROTEINS	IMDB-BINARY	IMDB-MULTI	COLLAB
CapsGNN	1521Mb	2279Mb	2695Mb	6135Mb	2501Mb	1975Mb	8111Mb
AdaGCCN	1990Mb	3162Mb	5526Mb	11120Mb	4692Mb	4730Mb	10572Mb

The D and W selected by the assistant module in AdaGCCN compared to the original depth and width in CapsGNN on different datasets are displayed in Table 5. Note that the optimal D and W determined by AM are different on the 10 splittings. We take the D and W on the splitting with the best test accuracy for the comparison against CapsGNN.

Algorithm 2: Scheduling within a dataset.

Input: Number of running workers: k, GPU memory allocated to each worker: M_s, available GPU memory: M_a, number of maximal concurrent workers: $K = \lfloor M_a / M_s \rfloor$.

```
Launch K workers and initialize k as K;
while k < K do
    Launch K − k workers from the unexecuted workers;
    Update k;
    if k = 0 then
        Break;
```

Algorithm 3: Scheduling among multiple datasets.

Input: The list of datasets: $\{ds_i\}$ (length: $len_{ds} \geq 2$), GPU memory allocated to each worker on ds_i: M_i, available GPU memory: M_a.

```
1  for i in [1, len_ds] do
2      if training on ds_i not begins then
3          U_i ← number of unexecuted splittings in ds_i;
4          K_i ← min(⌊M_a/M_i⌋, U_i);
5          Initialize K_i workers on ds_i;
6      for j in [i, len_ds] do
7          if training on ds_j not begins then
8              if M_a − K_i × M_i ≥ M_j then
9                  U_j ← number of unexecuted splittings in ds_j;
10                 K_j ← min(⌊(M_a − K_i × M_i)/M_j⌋, U_j);
11                 Initialize K_j workers in ds_j;
12 Repeat lines 1 to 11, and schedule on ds_i and ds_j with Algorithm 2, until all splittings from all datasets are executed.
```

Table 5: Comparison of (D, W) in CapsGNN and AdaGCCN.

Model	MUTAG	ENZYMES	NCI1	PROTEINS	IMDB-BINARY	IMDB-MULTI	COLLAB
CapsGNN	(5, 2)	(5, 2)	(5, 2)	(5, 2)	(5, 2)	(5, 2)	(5, 2)
AdaGCCN	(7, 4)	(7, 3)	(6, 4)	(6, 2)	(5, 3)	(7, 2)	(6, 3)