Towards Graph Pooling by Edge Contraction

Frederik Diehl [1] **Thomas Brunner** [1] **Michael Truong Le** [1] **Alois Knoll** [2]

Abstract

Graph Neural Networks (GNNs) research has concentrated on improving convolutional layers, with little attention paid to developing graph pooling layers. Yet pooling layers can enable GNNs to reason over abstracted groups of nodes instead of single nodes, thus increasing their generalization potential. To close this gap, we propose a graph pooling layer relying on the notion of edge contraction: EdgePool learns a localized and sparse pooling transform. We evaluate it on four datasets, finding that it increases performance on the three largest. We also show that EdgePool can be integrated in existing GNN architectures without adding any additional losses or regularization.

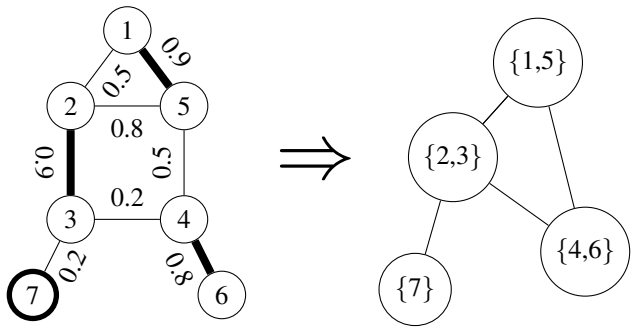

Figure 1. EdgePool: A score is computed for each edge in a graph (left). These edges are contracted in order of their score (edges $\{1,5\}$, $\{4,6\}$, $\{2,3\}$; chosen edges in bold), with nodes that have already been part of a pooled edge ignored (p.ex. edge $\{2,5\}$). Edges between contracted nodes are kept, as are left-over nodes (node 7). The resulting graph (right) is a pooled representation.

1. Introduction

In recent years, a fast-growing field of applying deep learning to graphs has emerged. Many of these works are inspired by Convolutional Neural Networks (CNNs). But while a multitude of different graph convolutional layers for these Graph Convolutional Networks (GCNs) have been proposed, the number of proposed pooling layers remains small.

Yet intelligent pooling on graphs holds significant promise: It might both identify clusters (feature- or structure-based) and reduce computational requirements by reducing the number of nodes. Together, these promise to abstract from nodes to sets of nodes. They are also a stepping stone towards enabling GNNs to modify graph structures instead of only node features.

We propose a new pooling layer based on edge contraction (EdgePool, see Fig. 1), which aims to correct existing weaknesses in previously proposed learned pooling layers. We do this by viewing the task not as choosing nodes but as choosing edges and pooling the connected nodes. This immediately and naturally takes the graph structure into account and ensures that we do not lose information.

Our main contributions are:

- We introduce a new learned pooling method.
- We show that it mostly outperforms existing methods.
- We show that it can be integrated into existing models without any changes.

Since submitting this paper, we have improved the method herein (Diehl, 2019) and applied it to node classification.

2. Related Work

There are two strategies to enable pooling on graphs: We can either integrate fixed pooling methods directly in the algorithm or learn how to pool. We concentrate on comparisons with learned pooling methods, since these appear to outperform fixed pooling methods.

Ying et al. (2018) were the first to propose a learned pooling layer. DiffPool learns to soft-assign each node to a fixed number of clusters based on their features. DiffPool works well, but suffers from three disadvantages: (a) The number of clusters has to be chosen in advance, which might cause performance issues when used on datasets with different

[1]fortiss GmbH, affiliated institute of Technische Universität München, Munich, Germany [2]Chair of Robotics, Artificial Intelligence and Real-time Systems, Technische Universität München, Germany. Correspondence to: Frederik Diehl <f.diehl@tum.de>.

Presented at the ICML 2019 Workshop on Learning and Reasoning with Graph-Structured Data.

graph sizes. (b) Since cluster assignment is based only on node features, nodes are assigned the same cluster based on their features, ignoring distances. (c) The cluster assignment matrix is dense, and in $\mathbb{R}^{n \times O(n)}$, i.e. size and number of operations scale quadratically with the number of nodes n. They also need several auxiliary objectives (link prediction, node feature ℓ_2 regularization, cluster assignment entropy regularization) to train well. In addition to that, the density makes integration into usually sparse GNNs difficult.

Graph U-Net, introduced by Gao & Ji (2018), uses a simple top-k choice of nodes for their gPool layer, learning a node score and dropping all but the top nodes. Cangea et al. (2018) later applied this to graph classification. While this approach is both sparse and variable in graph size, it introduces two new issues: (a) Adding random, unconnected nodes to a graph can change the pooling result of the whole graph. (b) Whole areas of a graph might see no node chosen, which loses information.

3. Edge Contraction Pooling

In the following, we consider a graph $G = (V, E)$, where each of the v nodes has f features $V \in \mathbb{R}^{v \times f}$. Edges are represented as pairs of nodes without weights of features.

While graph convolutional functions take that fixed graph and only transform the node features, pooling functions also transform the graph and reduce the number of nodes. The resulting graph is a coarse representation of the input graph.

3.1. Edge Contraction

Intuitively, edge contractions mean merging two nodes. Contracting the edge $e = \{v_i, v_j\}$ introduces the new vertex v_e and new edges such that v_e is adjacent to all nodes v_i or v_j has been adjacent to. v_i, v_j, and all their edges are deleted from the graph. This is written as G/e, and each such contraction reduces the number of nodes in a graph by 1.

Since edge contractions are commutative, we can also define an edge set contraction G/E', where $E' = \{e_1, \dots, e_n\} \subseteq E$. In this work, we avoid contracting edges which are incident to the same node.

3.2. Edge Contraction Pooling

Using edge contractions, we can now define a pooling methodology which chooses a set of edges and then uses edge contraction to produce a new graph. This requires us to decide on two architectural choices: How to choose which edges to pool and how to combine node features.

3.2.1. Choosing edges

Our procedure requires computing a score for each edge. The function to compute this can be freely chosen. We first compute raw scores for each edge as a simple linear combination of the concatenated node features, i.e. for an edge from node i to node j, we compute the raw score r as

$$r(e_{ij}) = W(n_i \| n_j) + b, \tag{1}$$

where n_i and n_j are the node features and W and b are learned parameters[1]. From these raw scores, we now compute actual node scores. We evaluate two different construction methods:

tanh Following the classical gate architecture, we simply compute node scores by using a tanh-nonlinearity, i.e. $s_{ij} = \tanh(r_{ij})$.

softmax The tanh score computation does not take neighboring edges into account. Intuitively, we'd much rather choose edges without any alternatives (e.g. one with a score of 0.4 vs one with a score of 0.1) than those which are close, yet with a high score (e.g. 0.95 vs 0.94). To take the neighborhood into account, we softmax-normalize edge scores over all edges which end in the same node, i.e. $s_{ij} = \text{softmax}_{r_{*j}}(r_{ij})$.

Given a node score, we can now begin the process of choosing edges to contract. We sort all edges by their score and successively choose the edge with the highest score whose two nodes have not yet been part of a contracted edge. By assuming a loop between non-contracted edges (by either adding self-loops or constructed once no more valid edges remain), we guarantee that the features from every node will be present in the pooled graph.

3.2.2. Combining node features

There are many strategies for combining the features of pairs of nodes. We found that taking the sum of the features works well. To enable the gradient to flow into the scores, we use gating, and multiply the combined node features by the edge score:

$$\hat{n}_{ij} = s_{ij}(n_i + n_j). \tag{2}$$

3.2.3. Advantages and Disadvantages

From the previous description, it becomes clear that EdgePool can easily operate on sparse representations. When

[1] This could also easily take edge features into account. However, we leave this and the question of how to merge edge features to future work.

doing so, runtime scales linearly in the number of edges. Therefore, it can avoid the runtime and memory issues of DiffPool, whose cluster assignment matrix size scales quadratically in the number of nodes. An evaluation for this can be found in Appendix B.1.

At the same time, the pooling algorithm ensures both that every node's features will be represented after pooling and that any change to the graph has only local influence on the chosen pooling. This has advantages both for changing graphs (only the neighborhood of changed nodes has to be recomputed) and large graphs (it can be parallelized with little overlap).

However, EdgePool as currently designed introduces a disadvantage: Each step roughly halves the number of nodes in the graph. That ratio is fixed and cannot be chosen by the user.

4. Experiments

With our experiments, we aim to answer two questions:

Q1: Does EdgePool outperform TopKPool and DiffPool?

Q2: Can EdgePool be used as a plug-and-play addition into any GNN?

4.1. General Setup

We evaluate our models on three graph classification datasets, and share most of the training procedures between all models.

4.1.1. Datasets

While there are many graph classification datasets available, most of these are small (in both nodes per graph and total graphs). As an example, the popular ENZYMES dataset contains only 600 graphs, making 10-fold crossvalidation (at a test set size of 60) very difficult.

We therefore evaluate on four larger datasets from the collection by Kersting et al. (2016): PROTEINS (Borgwardt et al., 2005) is the smallest at 1113 graphs, but has been used extensively as a benchmark dataset. The task is to predict whether a given protein is an enzymes. The two reddit-based datasets (Yanardag & Vishwanathan, 2015) model user responses in an online discussion. The task is to predict the subreddit, either binarily (REDDIT-BINARY) or from a set of 11 (REDDIT-MULTI-12K). Lastly, each COLLAB graph models scientific collaboration of one researcher. The task is to classify to which of three fields the researcher belongs. Neither COLLAB nor the two reddit-based datasets have node features.

4.1.2. Training

While we use different models to answer Q1 and Q2, several setup parameters have been chosen identically between the models. The models are trained for a total of 200 epochs using the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 10^{-2}, which is halved after every 50 epochs. 128 graphs are batched together at each step by treating them as a single unconnected graph.

We use 128 channels except for PROTEINS, where we used 64. This follows Ying et al. (2018). See Appendix A for a more detailed description.

4.2. Experimental Design

We design experiments to answer the questions above. To answer Q1, we first compare EdgePool with the DiffPool and TopKPool graph pooling methods. To answer Q2, we integrate EdgePool into a wide variety of different models.

4.3. Q1: Does EdgePool Outperform Alternatives?

For this comparison, we use the same architecture as used in DiffPool (Ying et al., 2018): The model has three SAGEConv blocks (Hamilton et al., 2017), whose outputs are globally mean-pooled and concatenated. Final classification occurs after two fully-connected layers. The default model does not pool nodes; any model which does so pools after every block (see Fig. 2). Note that DiffPool uses a siamese architecture, using separate blocks to compute cluster assignments. We restrict DiffPool to a maximum of 750 nodes per graph. TopKPool pools with a ratio of 0.5 to remain comparable to EdgePool.

In addition, we only use the cross-entropy loss to train the model. To ensure a fair comparison, we also do this for DiffPool, which originally used three additional auxiliary losses and tasks to stabilize training and precomputed additional features.

4.4. Q2: Can EdgePool be Integrated in Existing Architectures

To evaluate whether EdgePool can be integrated in pre-existing models, we follow the model configuration from *pytorch-geometric*'s benchmarks (Fey & Lenssen, 2019). Specifically, we use a total of five convolutional layers, followed by a global pooling layer and two fully-connected layers. If pooling is used, it is added between every second convolutional layer (i.e. there are two pooling layers).

The convolutional layers evaluated are GCN (Kipf & Welling), GIN and GIN0 (Xu et al., 2019), and GraphSAGE (Hamilton et al., 2017) both with and without accumulating intermediate results (GraphSAGE na).

Table 1. Accuracy (and standard deviation) on benchmark datasets in percent. Best results are marked bold. [*] Ying et al. (2018) use several additional techniques and auxiliary losses to stabilize training, and also include additional computed features. We report results without these.

	PROTEINS	RDT-B	RDT-12K	COLLAB
Base Model	$\mathbf{72.3 \pm 6.3}$	70.4 ± 4.1	37.4 ± 1.2	64.0 ± 2.6
DiffPool [*]	72.0 ± 4.3	83.2 ± 2.5	35.0 ± 1.4	69.7 ± 2.3
TopKPool	70.3 ± 3.8	71.1 ± 3.4	30.5 ± 1.1	63.1 ± 2.5
EdgePool (tanh)	71.5 ± 4.7	$\mathbf{87.4 \pm 2.3}$	$\mathbf{47.6 \pm 1.8}$	68.3 ± 2.2
EdgePool (softmax)	71.3 ± 2.3	86.4 ± 3.7	46.1 ± 1.0	$\mathbf{70.4 \pm 1.7}$

Table 2. Comparing performance differences between the baseline and added EdgePool layers. This is difference in accuracy in percentage points.

	PROTEINS	RDT-B	RDT-12K	COLLAB
GCN	$+2.4$	$+0.9$	-0.2	$+3.6$
GIN	$+0.6$	-1.2	-1.7	-1.3
GIN0	$+0.4$	-0.8	-1.2	-1.6
GraphSAGE	$+0.6$	$+5.0$	$+12.4$	-1.6
GraphSAGE na	-3.5	$+9.7$	$+9.6$	-2.2

5. Results and Discussion

Below, we report the results for our experiments. Every run used 10-fold cross-validation, and we report both performance mean and standard deviation.

5.1. Pooling Comparison

As the results in Table 1 show, EdgePool outperforms both the baseline and both alternative pooling methods on three out of four datasets.

On REDDIT-BINARY and REDDIT-MULTI-12K, EdgePool manages the most visible increase in performance. On COLLAB, EdgePool performs close to the DiffPool model, but both represent large increases in performance compared to the baseline. It only performs worse on PROTEINS. However, we note that results on PROTEINS are very noisy, making comparisons difficult.

This answers Q1: On no dataset, EdgePool performs significantly worse. On 3 of 4 datasets, it strongly outperforms the baseline and on two all other models.

5.2. Integration of EdgePool

Table 2 shows a summary of the results.

In general, we note that the performance increase of EdgePool depends on both convolutional layer used and dataset but performances. Ignoring PROTEINS due to the graph size and large variability in results, neither of the two GIN variants profit from introducing pooling. GCN, on the other hand, generally does.

For datasets, both reddit datasets show improvements most prominent for the GraphSAGE models (5-12.4 percentage points). COLLAB shows a decrease in performance except for the GCN model; PROTEINS a general increase except for the GraphSAGE na model.

This poses two interesting future questions: What is the interaction between GIN and EdgePool that decreases performance? What is the explanation for the performance differences between the reddit datasets and COLLAB[2]?

Detailed results can be found in Appendix B.

6. Conclusion

We introduced a sparse, learnable pooling method for graphs based on edge contraction. We have shown that it can be integrated into existing architectures without any changes to the training procedure, and that it outperforms other learned pooling methods.

Much work remains to be done: In particular, we would like to apply this to both node classification and graphs with edge features. We would also like to further investigate the quality of the found poolings.

We see EdgePool as a stepping stone towards enabling a class of GNNs which modify and create graph structures instead of only node features. Yet even today, EdgePool represents a capable drop-in pooling layer for GNNs, and we hope it will inspire more research into such layers.

[2]This might be based on edge density (< 2 edges per node for the reddit datasets and ≈ 32 for COLLAB).

References

Borgwardt, K. M., Ong, C. S., Schönauer, S., Vishwanathan, S. V. N., Smola, A. J., and Kriegel, H.-P. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl_1): i47–i56, June 2005. ISSN 1367-4803. doi: 10.1093/bioinformatics/bti1007.

Cangea, C., Veličković, P., Jovanović, N., Kipf, T., and Liò, P. Towards Sparse Hierarchical Graph Classifiers. In *NeurIPS 2018 Workshop on Relational Representation Learning*, November 2018.

Diehl, F. Edge Contraction Pooling for Graph Neural Networks. *arXiv:1905.10990 [cs, stat]*, May 2019.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Gao, H. and Ji, S. Graph U-Net. September 2018. URL https://openreview.net/forum?id=HJePRoAct7.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017.

Kersting, K., Kriege, N. M., Morris, C., Mutzel, P., and Neumann, M. Benchmark data sets for graph kernels, 2016. URL http://graphkernels.cs.tu-dortmund.de.

Kingma, D. and Ba, J. Adam: A method for stochastic optimization. *arXiv:1412.6980 [cs, stat]*, December 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR 2017*.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

Yanardag, P. and Vishwanathan, S. V. N. Deep Graph Kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1365–1374. ACM, October 2015. ISBN 978-1-4503-3664-2. doi: 10.1145/2783258.2783417.

Ying, R., You, J., Morris, C., Ren, X., Hamilton, W., and Leskovec, J. Hierarchical graph representation learning with differentiable pooling. In *Advances in Neural Information Processing Systems 31*, pp. 4800–4810. 2018.

Appendix

A. Training and Model Details

In the following section, we describe both our models and our training procedure in detail.

A.1. Models

We use two classes of models to answer Q1 and Q2.

A.1.1. SAGE-Style Models

SAGE-style models are used to answer Q1. This is equivalent to the model described by Ying et al. (2018), and is shown in Fig. 2. Its main features are the use of SAGE-Blocks, whose output is concatenated and mean-pooled to produce a graph-level feature vector. This is then used for classification.

The number of hidden units is identical to all layers, and has been set to 64 for PROTEINS and 128 for all other experiments. This follows Ying et al. (2018).

Experimentally, we found that the featureless datasets (where we have set all node features to a scalar 1) interact strangely with batch normalization, making the normal use of population statistics during evaluation not work. Instead, we also use mini-batch statistics during evaluation. We aim to further investigate this.

A.1.2. Benchmark-Style Models

Benchmark-Style models are used to answer Q2. These are almost straight adaptions of the benchmark models from *pytorch-geometric* (Fey & Lenssen, 2019), since they represent a good variety of different GCN models. We add both batch normalization and dropout. For pooling, we follow the example procedure from their TopKPooling implementation and pool after every second convolutional layer (see also Fig. 3) for a total of two pooling layers.

Here too, we use 64 units for PROTEINS and 128 for all other datasets.

A.2. Training Details

As described in in Section 4.1.2, we train all models for a total of 200 epochs using a batch size of 128. The initial learning rate for the Adam optimizer is 10^{-2}, and we halve it every 50 epochs.

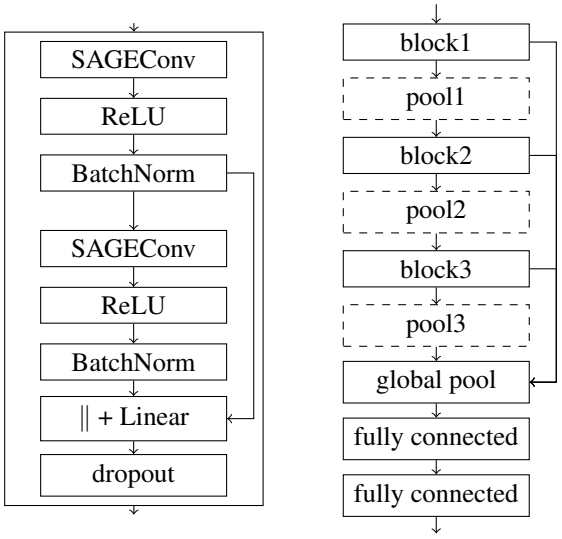

Figure 2. SAGE-style models used to answer Q1. Blocks are depicted to the left; the whole model to the right.

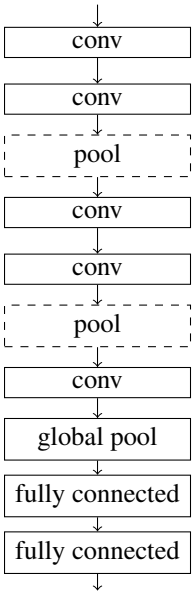

Figure 3. Benchmark-style models used to answer Q2.

Table 3. Accuracy (in percent) of benchmark models with and without EdgePool. GraphSAGE na means GraphSAGE without accumulating results.

	Model	No Pooling	TopKPool	EdgePool (tanh)	EdgePool (softmax)
PROTEINS	GCN	71.3 ± 5.0	71.3 ± 4.8	70.2 ± 5.4	$\mathbf{73.7 \pm 5.5}$
	GIN	70.8 ± 5.5	$\mathbf{72.8 \pm 4.3}$	71.4 ± 4.0	71.3 ± 4.8
	GIN0	71.4 ± 5.1	$\mathbf{73.2 \pm 4.5}$	71.8 ± 4.6	70.3 ± 3.9
	GraphSAGE	73.6 ± 4.4	73.7 ± 4.5	71.9 ± 5.4	$\mathbf{74.2 \pm 4.5}$
	GraphSAGE na	$\mathbf{72.5 \pm 5.3}$	69.8 ± 4.4	69.0 ± 3.8	67.0 ± 7.3
	Model	**No Pooling**	**TopKPool**	**EdgePool (tanh)**	**EdgePool (softmax)**
RDT-B	GCN	89.4 ± 1.7	80.5 ± 4.9	89.0 ± 2.7	$\mathbf{90.3 \pm 2.0}$
	GIN	$\mathbf{92.1 \pm 1.3}$	88.1 ± 2.8	90.9 ± 1.9	90.4 ± 2.5
	GIN0	$\mathbf{91.9 \pm 2.2}$	87.7 ± 5.1	91.1 ± 2.5	90.2 ± 2.6
	GraphSAGE	63.5 ± 5.1	48.6 ± 4.3	68.3 ± 3.8	$\mathbf{68.5 \pm 4.3}$
	GraphSAGE na	50.1 ± 5.7	53.5 ± 6.0	59.4 ± 4.3	$\mathbf{59.8 \pm 2.1}$
	Model	**No Pooling**	**TopKPool**	**EdgePool (tanh)**	**EdgePool (softmax)**
RDT-12K	GCN	$\mathbf{47.3 \pm 2.3}$	43.0 ± 2.3	46.8 ± 1.5	47.1 ± 1.9
	GIN	$\mathbf{49.9 \pm 1.6}$	47.8 ± 1.2	48.2 ± 1.5	48.0 ± 1.6
	GIN0	$\mathbf{49.5 \pm 1.3}$	46.7 ± 2.4	48.3 ± 1.7	48.3 ± 1.6
	GraphSAGE	23.0 ± 1.7	22.2 ± 1.2	35.2 ± 1.4	$\mathbf{35.4 \pm 2.0}$
	GraphSAGE na	23.8 ± 1.2	22.3 ± 1.4	33.2 ± 1.4	$\mathbf{33.4 \pm 1.5}$
	Model	**No Pooling**	**TopKPool**	**EdgePool (tanh)**	**EdgePool (softmax)**
COLLAB	GCN	64.4 ± 2.1	65.4 ± 2.1	$\mathbf{68.0 \pm 2.6}$	67.9 ± 1.6
	GIN	$\mathbf{71.0 \pm 1.5}$	69.9 ± 1.8	69.7 ± 1.9	69.5 ± 2.5
	GIN0	$\mathbf{71.5 \pm 1.3}$	70.1 ± 2.1	70.2 ± 2.7	69.2 ± 2.6
	GraphSAGE	$\mathbf{64.2 \pm 1.7}$	60.4 ± 2.1	62.6 ± 3.5	62.4 ± 2.7
	GraphSAGE na	$\mathbf{65.3 \pm 1.1}$	58.4 ± 3.4	63.1 ± 3.3	62.0 ± 3.6

We use ten-fold cross-validation for each dataset, designating one fold as test and another as validation data. We use the parameters which showed the highest performance on the validation dataset[3].

B. Additional Results

The expanded results from Section 5.2 can be found in Table 3.

B.1. Memory Usage

Fig. 4 shows the memory usage by number of nodes in the graph. Following Cangea et al. (2018), we construct Erdős-Rényi-Graphs with $|E| \approx 2\,|V|$. We use SAGE-Style models with 128 random node features and compute one forward and one backward pass. We evaluated this on a 1080 Ti GPU with 11GB memory. As the figure shows, both the sparse base model and EdgePool scale linearly in the number of nodes while DiffPool scales quadratically.

In particular, we note that DiffPool fails at more than $18k$ nodes, while both sparse models can be used for graphs of up to $250k$ nodes. EdgePool improves this yet again to $300k$. More pooling layers (in deeper models) would increase that difference.

In all of these models, batching can be done in linear memory: DiffPool's adjacency matrix can be replicated along a batch axis similar to images while sparse approaches simply assume unconnected graphs. Even so, a batch size of 128 as used in this work restricts DiffPool to graphs of at most $1.5k$ nodes[4] and the non-pooling model to graphs of $2k$ nodes. EdgePool increases this to $2.4k$ nodes.

However, note that the number of edges above play to EdgePool's strengths, since it scales in the number of edges. If we use $|E| \approx 64\,|V|$ (see Fig. 5), we see a larger memory use of the EdgePool model until about $5k$ nodes. However, DiffPool still fails at far fewer nodes ($13k$) compared to the sparse model ($28k$) or EdgePool ($23k$).

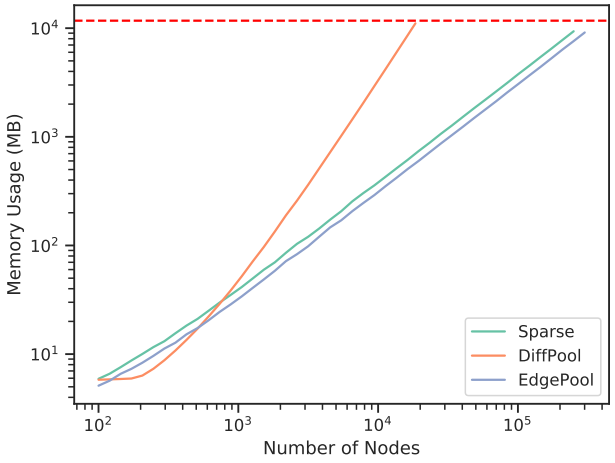

Figure 4. Memory usage comparison between DiffPool, a sparse base model, and an EdgePool model. Note the log-log scale on the graph. The dashed red line marks the limit of GPU memory. As can be seen, DiffPool's memory requirements scale quadratically while both sparse models only scale linearly in the number of nodes. Additionally, EdgePool requires less memory than the non-pooling base model.

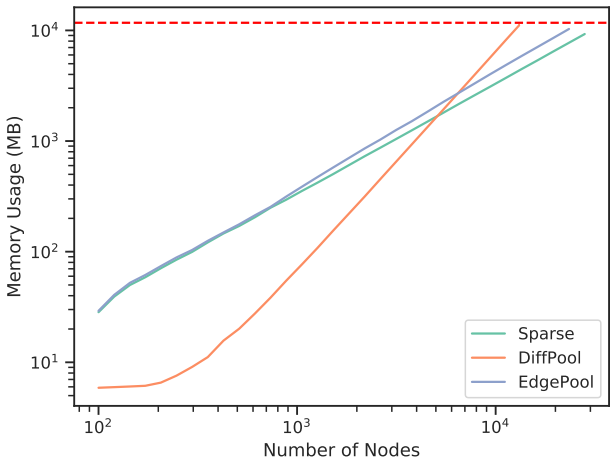

Figure 5. Memory usage comparison for 64 edges per node. As can be seen, DiffPool still scales identically to Fig. 4, while both sparse models need more memory but still scale linearly.

[3]Note that several other works do not do so but use a 9-1 train/test split without validation set, choosing the model that works best on the test set.

[4]Since DiffPool scales quadratically in the number of nodes, node counts are higher than expected from a simple linear extrapolation.