Advanced Machine Learning Project Open Graph Benchmark

Rigo Andrea

ABSTRACT

In this project several models are trained and evaluated, starting from the ones seen during the lectures on Graph Neural Networks (GNNs), on the *molhiv* dataset from the Open Graph Benchmark (OGB) [7, 8] on the task of graph classification. The best model finally undergoes the evaluation procedure indicated by OGB and is compared to the models on the OGB *molhiv* leaderboard. Section I will describe the problem and the dataset used, section II will give a brief overview of each of the approaches used in this work, section III will talk about the models trained, section IV will describe the experiments that were carried out and their results and finally section V will describe OGB's evaluation procedure and show the results compared to the leaderboard. The code is available at https://github.com/andrearigo-dev/aml-open-graph-benchmark.

I. PROBLEM FORMULATION AND DATASET

The task to be solved is graph classification. Given a graph $G = (V, E)$, where V is a set of nodes and E is a set of edges, the task is to assign a label to the entire graph based on nodes and edges features. In the case of *molhiv*, each graph represents a molecule and the task is to predict whether a molecule inhibits HIV virus replication or not, with binary labels. The dataset contains 41127 graphs, with an average of 25.5 nodes and 27.5 edges per graph. Each node is associated with a 9-dimensional feature vector and each edge with a 3-dimensional one. Given the limited computational resources available, this dataset was chosen because it is the smallest available for the task, making training and evaluation faster and therefore making development easier. For the *molhiv* dataset, OGB dictates the use of the Receiver Operating Characteristic Area Under the Curve (ROC-AUC) as evaluation metric. The ROC curve is a graph showing the performance of a classification model at all classification thresholds by plotting the true positive rate (recall) and the false positive rate. The area under the ROC curve measures the two-dimensional area underneath the entire the curve, which provides an aggregate measure of performance across all possible classification thresholds.

II. RELATED WORKS

GNNs, specifically Convolutional Graph Neural Networks (ConvGNNs), employ a series of Graph Convolution layers. The Graph Convolution is inspired by the convolution operation on pixels in Convolutional Neural Networks (CNNs), and given a node, it aggregates its neighbor nodes' features and combines them with the node's own features, producing a new feature representation for the node itself, which is then passed to next layer. The sequence of Graph Convolution layers in ConvGNNs gradually refines the nodes features with their neighbors'. ConvGNNs can also be seen as a case of Message Passing Neural Network (MPNN) [5], a general framework that groups many GNN architectures where neighboring nodes send their features through messages that get propagated along the edges of the graph and combined with the receiving node's features. In the case of graph classification, the features of all nodes at the last layer get pooled together in a single representation for the whole graph, which is then passed to a classifier to predict a label for the graph. This final global pooling is often called readout, and for example it can be the average of all the transformed nodes' feature vectors. The way the neighbor information is aggregated and used to update a node's representation, as well as the readout operation, can be defined in several different ways. This work makes use of PyTorch Geometric (PyG) [4], which provides many ConvGNNs components' implementations from different papers, ready for use. Because of the ease of use of such components thanks to the library, in this work several implementations were tested, plus a few attempts at combining different approaches. This section will briefly introduce how the various implementations work.

A. Graph Convolutional Networks

The first type of GNN used for this project are the Graph Convolutional Networks (GCNs) [11]. They are based on spectral convolutions on graphs [3], which suffer from a high computational complexity, and propose a fast approximation of it. GCNs define the convolution operation as:

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta} \tag{1}$$

where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ is the adjacency matrix with inserted self loops, $\hat{\mathbf{D}}$ is its diagonal matrix, $\mathbf{\Theta}$ is a learnable matrix and $\mathbf{X}$ is the nodes' features matrix. The adjacency matrix can also contain edge weights if needed.

B. Modified Graph Isomorphism Network

The graph isomorphism problem asks whether two graphs are topologically identical. This is a challenging problem as no polynomial-time algorithm to solve it is known. The Weisfeiler-Lehman (WL) [1] test of graph isomorphism is an effective and computationally efficient test that distinguishes a broad class of graphs. Xu et al. [17] studied the representational power of several GNN variants and postulated that no GNN can have a higher representational power than the

WL test. They then designed the Graph Isomorphism Network (GIN), a GNN as expressive as the test. [9] then proposed a modified version of GIN, that PyG calls GINE, that supports edge features. The convolution operator is defined as:

$$\mathbf{x}'_i = h_{\mathbf{\Theta}}\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \mathrm{ReLU}(\mathbf{x}_j + \mathbf{e}_{j,i})\right) \quad (2)$$

where $h_{\mathbf{\Theta}}$ is a neural network, i.e. a Multi Layer Perceptron (MLP), $\mathbf{x}_i$ are node features and $\mathbf{e}_{j,i}$ are edge features.

C. SAGE: SAmple and aggreGatE

Hamilton et al. [6] proposed a general framework called GraphSAGE (SAmple and aggreGatE), or simply SAGE. At at each iteration, SAGE aggregates information from the local neighbors of node v at layer k:

$$\mathbf{h}^k_{\mathcal{N}(v)} \leftarrow \mathrm{AGGREGATE}_k(\{\mathbf{h}^{k-1}_u, \forall u \in \mathcal{N}(v)\}) \quad (3)$$

where $\mathcal{N}(v)$ is the local node v neighborhood, $\mathbf{h}^k_{\mathcal{N}(v)}$ are the neighbors aggregated features at layer k and $\mathbf{h}^{k-1}_u$ is the feature vector of a neighbor node u at the previous layer $k-1$. Then the aggregated information is combined to the node own features:

$$\mathbf{h}^k_v \leftarrow \sigma\left(\mathbf{W}^k \cdot \mathrm{CONCAT}(\mathbf{h}^{k-1}_v, \mathbf{h}^k_{\mathcal{N}(v)})\right) \quad (4)$$

where $\mathbf{W}^k$ is a learnable weight matrix and σ is a non linearity. Each layer aggregates information from each node local neighbors, and as this process iterates, nodes incrementally gain more and more information from further reaches of the graph. The AGGREGATE operator can be defined in different ways. In the paper [6] the authors implement the mean aggregator, which simply takes the elementwise mean of the neighbors' feature vectors, a Long Short Term Memory Network (LSTM) and finally the *pooling* aggregator, which PyG simply calls *max*, where each neighbor's vector is independently fed through a fully-connected neural network, then all the transformed vectors are fed to a max-pooling operation.

D. Graph Attention Networks v2

Veličković et al. [15] proposed the Graph Attention Network (GAT), which uses self attention to allow each node to attend to each other nodes in its neighborhood. Given a query node i, the graph attention layer computes attention scores between the node and its neighbors, including i itself, then combines the neighbors information using the attention scores as weights to give more importance to the more relevant neighbors for i. Later, Brody et al. [2] discovered that the attention mechanism employed by GAT has some limitations, as it can't model situations where different keys have different relevance to different queries. So, they proposed a slightly modified GAT, GATv2, that solves this issue and makes GAT more expressive. The nodes' features are combined as follows:

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j \quad (5)$$

where $\alpha_{i,j}$ is the attention score between nodes i and j, $\mathbf{x}_i$ are node i features, $\mathbf{\Theta}$ is a learned matrix that transforms the features and $\mathcal{N}(i)$ is the neighborhood. The attention coefficients are computed as follows:

$$\alpha_{i,j} = \frac{\exp\left(\mathbf{a}^\top \mathrm{LeakyReLU}\left(\mathbf{\Theta}[\mathbf{x}_i \,\|\, \mathbf{x}_j \,\|\, \mathbf{e}_{i,j}]\right)\right)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}} \exp\left(\mathbf{a}^\top \mathrm{LeakyReLU}\left(\mathbf{\Theta}[\mathbf{x}_i \,\|\, \mathbf{x}_k \,\|\, \mathbf{e}_{i,k}]\right)\right)} \quad (6)$$

where $\mathbf{e}_{i,j}$ are multidimensional edge features. GATv2 also supports multi-head attention.

E. Attentional Aggregation

Li et al. [12] proposed an Attentional Aggregator, which aggregates nodes features using a sum weighted by attention scores:

$$\mathbf{r}_i = \sum_{n=1}^{N_i} \mathrm{softmax}\left(h_{\mathrm{gate}}(\mathbf{x}_n)\right) \cdot h_{\mathbf{\Theta}}(\mathbf{x}_n) \quad (7)$$

where h_{gate} and $h_{\mathbf{\Theta}}$ are two neural networks, for example MLPs, the former computes the attention scores and the latter transforms the features to the desired output dimension before combining them to the scores.

F. Differentiable Pooling

All GNNs seen above progressively refine node representations using neighbors information. However they don't learn any hierarchical structure between nodes, nor they use any pooling operation, both things that proved effective in CNNs to learn complex patterns. Ying et al. [19] proposed DiffPool, a differentiable graph pooling module that can generate hierarchical graph representations. DiffPool learns a differentiable soft cluster assignment for nodes at each layer of a deep GNN, mapping nodes to a set of clusters, which then form the coarsened input for the next GNN layer. Given the soft cluster assignment matrix $\mathbf{S}$, the nodes' features $\mathbf{X}$ and the graph adjacency matrix $\mathbf{A}$, a DiffPool layer coarsens the input graph, generating a new coarsened adjacency matrix $\mathbf{A}'$ and a new matrix of embeddings $\mathbf{X}'$ for each of the nodes/clusters in this coarsened graph by computing the weighted combination of cluster members:

$$\begin{aligned}\mathbf{X}' &= \mathrm{softmax}(\mathbf{S})^\top \cdot \mathbf{X} \\ \mathbf{A}' &= \mathrm{softmax}(\mathbf{S})^\top \cdot \mathbf{A} \cdot \mathrm{softmax}(\mathbf{S})\end{aligned} \quad (8)$$

The soft clustering matrix and node embeddings can be computed by any GNN module. Learning to cluster nodes using the graph label as the only source of supervision can be hard since the signal is very indirect. So the authors use two side objectives:

$$\begin{aligned}\mathcal{L}_{LP} &= \|\mathbf{A} - \mathrm{softmax}(\mathbf{S})\mathrm{softmax}(\mathbf{S})^\top\|_F, \\ \mathcal{L}_E &= \frac{1}{N}\sum_{n=1}^{N} H(\mathbf{S}_n)\end{aligned} \quad (9)$$

where $\mathcal{L}_{LP}$ is a link prediction loss encouraging nearby nodes to be pooled together, and $\mathcal{L}_E$ is the cluster entropy, encouraging hard assignments.

III. Models

This work first designs a base GNN, inspired by the PyG example code, then all following models are variations of it. The GNN architecture is illustrated in figure 1 and uses an Atom Encoder, which is a node encoder provided by OGB, followed by a block composed of a Graph Convolution layer, a Batch Norm [10] layer followed by the ReLU non-linearity, a Dropout layer with $p = 0.5$ and a residual connection that skips all the mentioned layers and propagates the signal as-is from the previous block or the Atom Encoder. Based on the parameter *layers* that determines the number of layers in the network, the block is repeated. Finally a readout operation is performed, combining all node embeddings into a single graph representation, which is then classified by being fed to a Linear layer and transformed into a probability with the Sigmoid function. Additionally, an MLP and MLP_{gate} networks are defined, which are necessary for some models. The former, is composed of a Linear layer doubling the hidden channels, a Batch Norm layer, ReLU activation function and another Linear layer that decreases the hidden channels back to the original dimension. The latter, has the same architecture, but decreases the hidden channels to 1 with the last Linear layer, and finally normalizes the output in the $[-1, 1]$ range using an hyperbolic tangent (Tanh). From now on, when referring to either of the two networks, it means a network with same architecture, not the same network as in the same trained weights.

The models used in this work, deriving from the base one, are the following:

1) GCN: This model is simply the base GNN, using GCN convolutions [11] (Section II-A) as Graph Convolution block.
2) EdgeGCN: The dataset comes with 3-dimensional edge features but GCN convolutions only support scalar weights, all set to 1 by default. So, in an attempt to make the GCN model aware of edge features, the edges' feature vectors are encoded by a Bond Encoder, which is OGB's provided edge encoder, and then fed to an MLP_{gate} that predicts a weight for the edge. The weight is than translated by two, bringing it into the $[1, 3]$ range:

$$\mathbf{w}_{i,j} = \mathrm{MLP}_{gate}(\mathrm{BondEnc}(\mathbf{e}_{i,j})) + 2 \qquad (10)$$

   where $\mathbf{w}_{i,j}$ is the resulting edge weight, $\mathbf{e}_{i,j}$ are the edge's features, and $\mathrm{BondEnc}$ is the Bond Encoder.
3) GINE: Base GNN using GINE convolutions [9] (Section II-B), with $h_{\mathbf{\Theta}} = \mathrm{MLP}$ to transform features. It also uses a Bond Encoder for edge features.
4) GAT: Base GNN using GATv2 convolutions [2] (Section II-D). Again, a Bond Encoder is used. The model is trained using 8 attention heads. Keep in mind however, that here for simplicity the model uses GNN Blocks with GATv2 convolutions, while in the original paper the authors only used GATv2 layers, with no Batch Norm or Dropout.
5) SAGE: Base GNN using SAGE convolutions [6] (Section II-C).

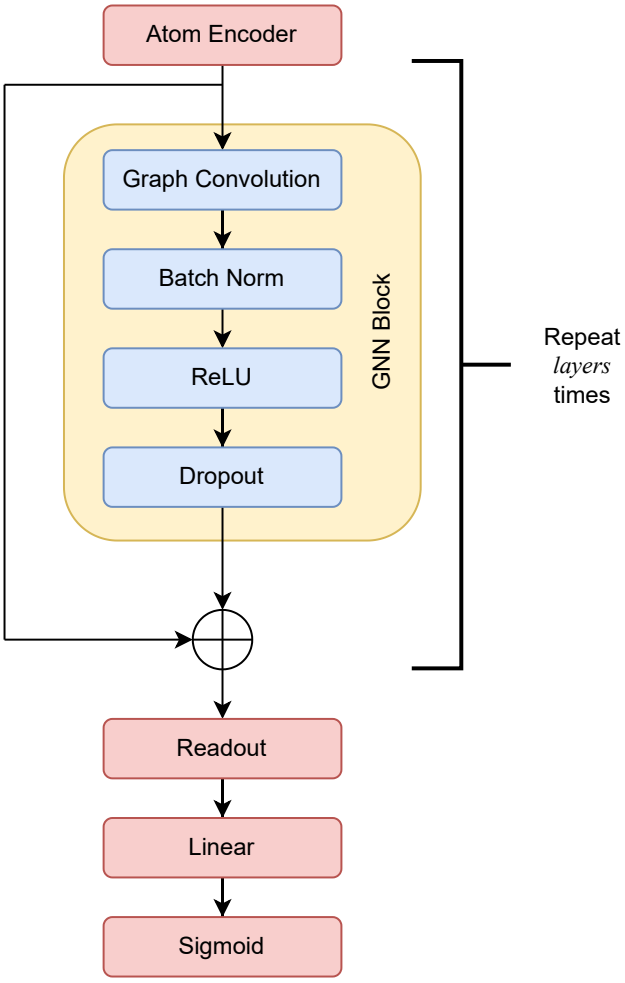

Figure 1. Base GNN scheme

6) DiffPool: Here, a DiffPool layer [19] (Section II-F) is placed every two GNN Blocks, except for the last layer. For example, for a 6 blocks network, there would be 3 DiffPool layers, and one would be after last GNN Block. This last DiffPool layer is not placed. As in the original paper, every GNN Block uses SAGE convolutions, and the DiffPool layers also use SAGE convolutions for transforming node features and for predicting the soft cluster assignments.

IV. Experiments

All the models have a final readout step, which uses an aggregator to combine all nodes features. SAGE convolutions aggregate features using an arbitrary aggregator function. The DiffPool model uses SAGE convolutions both in the GNN Blocks and in the pooling layers, therefore it also requires an aggregator function. Each aggregator in each model can either be a *mean* aggregator or a *max* aggregator, which simply takes the maximum of the features, except in the case of SAGE in which *max* refers to the *pooling* aggregator (Section II-C). Additionally, an Attentional Aggregator [12] (Section II-E) can be used in any of the above, adding an attentional component to models that wouldn't have it otherwise. Hence here all

the architectures described in previous sections are tested using different aggregators in the readout step, in the graph convolutions and in the pooling layers in order to explore which aggregators and models work best. In the case of the Attentional Aggregator, $h_{\text{gate}} = \text{MLP}_{gate}$ and $h_{\Theta} = \text{MLP}$ are used to compute the attention scores and transform features. All models were trained with the following parameters: batch size 64, 200 epochs, learning rate 0.001, 300 hidden channels for all the Graph Convolutions and 6 layers. The number of layers represents the number of GNN Blocks. In the case of DiffPool models, in addition to the 6 GNN Blocks there is a DiffPool layer every two of them. DiffPool layers also require an additional parameter, which is called *ReduceTo* in the code, that represents the number of clusters to assign nodes to, expressed as a percentage of number of clusters already present before applying the layer, or the number of nodes in the graph if no other pooling layer was applied before. Suppose a DiffPool layer groups nodes into 20 clusters, the next DiffPool must reduce the number of clusters to learn a hierarchical structure. So it will reduce the number of cluster to some percentage of the previous number. This parameter is set to 25%. In this example, the second DiffPool layer would cluster nodes to $20 \cdot 0.25 = 5$ clusters. One DiffPool model with a more aggressive clustering was also trained, with *ReduceTo* set to 50%. At every epoch the models were evaluated on the validation set, and at the end of the training the best epochs were taken for comparison.

Table II shows the results, and shows the trained model along with which type of aggregation was used by Graph Convolutions in the GNN Block and in the pooling layers as well as the readout. When the readout is set to *Pooling*, a DiffPool layer is used to group all nodes into a single cluster. The table also highlights the best model, which was DiffPool using *max* aggregation in the Graph Convolutions of the GNN Block, in the pooling layers and as readout.

The hyperparameters for the best model are then tuned using a grid search. The grid search trains and evaluates the model with all possible hyperparameters combinations from a predefined set of values and outputs the best configuration. Due to the computational resources available the search space is quite limited. Table I shows the possible parameters values.

Hyperparameter	Values
Batch size	32, 64
Number of layers	6, 8, 10
Learning rate	0.001
Hidden channels	300
Epochs	200

Table I
HYPERPARAMETERS SEARCH SPACE

The best configuration turned out to be initial one, described above.

V. OGB EVALUATION

The best model with the best configuration was then evaluated following OGB instructions: use OGB's provided evaluators to train and validate the model on both the test and validation set 10 times with 10 different random seeds, then report the average and standard deviation. The results are shown in table III. For comparison to the *molhiv* dataset leaderborad we will use the scores rounded to four decimals: 0.7712 ±0.0130 for test and 0.8548 ±0.0087 for validation. If it were to be submitted to the leaderboard at the time of writing this report, DiffPool would take the 27th place, as its test ROC-AUC is better than GIN+Virtual Node [17] but worse than Efficient Graph Convolution-Single (EGC-S) [13]. The former is a GIN where a virtual node connected to all the other nodes was added to the graph at the last layer. The neighborhood of the virtual node is the entire graph, so a GIN convolution will combine all nodes embeddings into a single final one before classification. The latter is an MPNN where messages don't depend on the target node, but only on the source, which also aims to be more computationally efficient than commonly used GNNs.

VI. CONCLUSION

Given the relative simplicity of the model and the scarce computational resources making deeper and larger architectures unfeasible, the 27th place on 33 is not a great achievement but I think it's enough for a small university project. For comparison, among the top of the leaderbaord there are much more complex and larger models, such as Graphormer [18], a Transformer [14] adapted to work on graph data and the best model, Pooling Architecture Search (PAS) [16], which uses Neural Architecture Search to search for an effective architecture for graph classification.

REFERENCES

[1] Boris Weisfeiler, A. L. (1968). A reduction of a graph to a canonical form and an algebra arising during this reduction. In *Nauchno-Technicheskaya Informatsia*.

[2] Brody, S., Alon, U., and Yahav, E. (2021). How attentive are graph attention networks?

[3] Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2014). Spectral networks and locally connected networks on graphs. In Bengio, Y. and LeCun, Y., editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*.

[4] Fey, M. and Lenssen, J. E. (2019). Fast graph representation learning with pytorch geometric.

[5] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry.

[6] Hamilton, W. L., Ying, R., and Leskovec, J. (2017). Inductive representation learning on large graphs. *CoRR*, abs/1706.02216.

[7] Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. (2020a). Open graph benchmark. https://ogb.stanford.edu/.

[8] Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. (2020b). Open graph benchmark: Datasets for machine learning on graphs.

[9] Hu, W., Liu, B., Gomes, J., Zitnik, M., Liang, P., Pande, V. S., and Leskovec, J. (2019). Pre-training graph neural networks. *CoRR*, abs/1905.12265.

[10] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167.

[11] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907.

[12] Li, Y., Gu, C., Dullien, T., Vinyals, O., and Kohli, P. (2019). Graph matching networks for learning the similarity of graph structured objects. *CoRR*, abs/1904.12787.

[13] Tailor, S. A., Opolka, F. L., Liò, P., and Lane, N. D. (2021). Do we need anisotropic graph neural networks?

[14] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need.

[15] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2017). Graph attention networks.

[16] Wei, L., Zhao, H., Yao, Q., and He, Z. (2021). Pooling architecture search for graph classification. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. ACM.

[17] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2018). How powerful are graph neural networks? *CoRR*, abs/1810.00826.

[18] Ying, C., Cai, T., Luo, S., Zheng, S., Ke, G., He, D., Shen, Y., and Liu, T.-Y. (2021). Do transformers really perform bad for graph representation? *CoRR*, abs/2106.05234.

[19] Ying, R., You, J., Morris, C., Ren, X., Hamilton, W. L., and Leskovec, J. (2018). Hierarchical graph representation learning with differentiable pooling. *CoRR*, abs/1806.08804.

Model	Readout	Aggregation	Pooling Aggregation	Validation ROC-AUC
GINE	Mean	-	-	0.7243
GCN	Mean	-	-	0.8202
GCN	Attentional	-	-	0.8078
EdgeGCN	Mean	-	-	0.8243
EdgeGCN	Attentional	-	-	0.8179
GAT	Mean	-	-	0.8257
SAGE	Mean	Mean	-	0.8354
SAGE	Max	Max	-	0.8415
SAGE	Mean	Max	-	0.8446
SAGE	Attentional	Max	-	0.8483
SAGE	Max	Attentional	-	0.8216
SAGE	Attentional	Attentional	-	0.8532
DiffPool	Mean	Mean	Mean	0.8382
DiffPool	Mean	Max	Attentional	0.8471
DiffPool	Pooling (Attentional)	Attentional	Attentional	0.8423
DiffPool, *ReduceTo* 50%	Attentional	Max	Max	0.8562
DiffPool	Attentional	Max	Max	0.8594
DiffPool	**Max**	**Max**	**Max**	**0.8597**

Table II
EVALUATION RESULTS

Seed	Test ROC-AUC	Validation ROC-AUC
0	0.7599528766488346	0.8641148805618286
1	0.7678228625504547	0.8619133234024048
2	0.774825701539234	0.8579328060150146
3	0.7743486741729273	0.8488297462463379
4	0.7651885899689064	0.834518313407898
5	0.7737577009984743	0.8518151044845581
6	0.7913246683018212	0.8569315671920776
7	0.7526603449274801	0.8628073930740356
8	0.7921377392379151	0.8527275919914246
9	0.7599007319569709	0.8560864925384521
AVG ±STD	**0.7711919546127319 ±0.012988988310098648**	**0.8547677993774414 ±0.008677446283400059**

Table III
DIFFPOOL OGB EVALUATION RESULTS