### **Solving OGB Function Name Prediction Using Data Augmentation and DiffPool Hierarchical Pooling**

*Stanford CS224W Project by “Nathanael” Yu Ren, Henry Ang, Lucy Zhu*

### **Introduction and Motivation**

#### **The problem**

In many real-world text-based applications, the ability to condense longer forms of text blocks into shorter and higher-level topologies is critical in being able to understand user-base trends and customer needs. For example, in the contact center space, the ability to summarize call transcripts into single-topic phrases allows business insight teams to understand the distribution of urgent customer concerns. This is a multi-million dollar opportunity that has not yet seen a widely accepted solution.

Clustering technology using word embeddings has demonstrated poor performance due to the high dimensionality of word embeddings, the presence of many outliers, and the difficulty in determining right seeding given the wide range of conversations. Typically such clustering techniques yield a precision rate of 10 to 30%. Other methods of using LLMs like GPT4 to zero-shot question-and-answer tag predictions generate too many isolated tags at a heavy price tag, which then requires a laborious manual process to cluster and review.

In this blog, we will talk about our experimentations with graph augmentation and hierarchical pooling to solve text condensation. The goal is to more accurately predict the final short-phrase topic tag of the paragraph text and hopefully along the way produce granular topic phrases at each level of the hierarchy that represents topology.

#### **Novelty**

DiffPool and GNNs at large have not been explored as potential methods for text summarization. However, when humans are asked to summarize a chunk of text with one phrase, the implicit exercise we do in our minds is most likely what commonality exists among all of the words. One way to model this is what node in a connected graph captures and can represent the subgraph of nodes. DiffPool is built for this purpose and is worthwhile to pursue.

In addition, as you’ll see in the following section, we picked a graph-level code OGB code summarization task where DiffPool has yet to be applied by the participants in the leaderboard. Furthermore, while exploring ways to outperform models on the leaderboard, we went above and beyond DiffPool implementation and tried several novel techniques to provide additional enhancements:

* CodeBERT embedding as the input into GNN in the pre-processing step
* RNN LSTM layers as post-processing
* Depth-biased pooling instead of global pooling

All listed above will be explored in further detail below.

### **Application Domain**

#### **Dataset**

For modeling, we chose to use the OGB graph-level prediction dataset [ogbg-code2](https://ogb.stanford.edu/docs/leader_graphprop/#ogbg-code2), which is a collection of 450k Python function definitions compiled from GitHub CodeSearchNet repositories. It is split into a training, validation, and test set where the training has approximately 410k samples, the validation has around 20k samples, and the test has around 20k samples. Each sample is a graph built from an AST hierarchy tree. Every node of the graph has 2-dimensional features where each dimension is an ID that maps to the raw [code type](https://docs.google.com/spreadsheets/d/1AxRRfu_Vzecs9b3iQS2HQ4lSumos7wn0/edit?usp=sharing&ouid=116932977220188674139&rtpof=true&sd=true) (e.g. for, while, list) and the raw [code attribute](https://docs.google.com/spreadsheets/d/1w_w6xvFRpmaGauB1DxnuNtpiCA95GaI6/edit?usp=sharing&ouid=116932977220188674139&rtpof=true&sd=true) (e.g. variable names). The nodes as input are sorted in DFS traversal order. The true labels y are the indices of the words that make up the method names.

We picked this dataset because it offers several advantages for this project. First, DiffPool has yet been applied to such a heterogeneous graph. Secondly, ogbg-code2 already converted all code text into directed AST graphs and is closest to a summarization task. These graphs have heterogeneous edges (e.g. conditional, else) and are in ready to use format, thereby reducing the scope of this project to be manageable. Thirdly, all submissions on the leaderboard have publications, so we can compare our methodology outcome.

#### **Task and Metric**

The task is often referred to as “code summarization,” whereby the model takes the code AST graph as input and predicts up to 5 words for the method name as the output. The words could be 1 of 5000 in the vocab list. This is a graph-level prediction task where, after all node embeddings are pooled into a prediction head, we perform a classification task that splits the prediction head into 5 outputs, each being the softmax probability the model assigns to the word at its index. Cross-entropy loss is used. An evaluator then calculates the F1 score for the accuracy of the model’s predicted method name.

While this is not exactly paragraph text condensation, there are lots of similarities — code summarization necessitates a form of text coarsening where code text is condensed. In fact, code summarization is harder because we have to predict the word sequence with just the code structure, which is often unrelated to the method naming. Solving the challenge for method name prediction will yield learnings that are generalizable for solving text summarization in general.

### **DiffPool Modeling**

****We created this illustration to explain how messaging and aggregation actually worked in our implementation

The idea for DiffPool is that at each layer, GraphSAGE or another GNN model is used to compute the node embeddings and then **S** coarsens these embeddings into fewer cluster nodes for the next layer. This process repeats until the entire graph is condensed into one node embedding that then is used for classification at the graph level (i.e. method name prediction). The model will have a differentiable assignment matrix that learns the clusters of higher hierarchy.



Here are code snippets for defining the DiffPool layers. Each DiffPool layer contains two configurable GNNs, generating node embeddings and assignment matrices respectively. They are subsequently fed into the DiffPool operator to generate the pooled nodes and their adjacency matrix.

class DiffPool(torch.nn.Module):

def \_\_init\_\_(self, gnn\_embed, gnn\_pool, number\_of\_nodes\_out):

super(DiffPool, self).\_\_init\_\_()

self.gnn\_embed = gnn\_embed

self.gnn\_pool = gnn\_pool

self.number\_of\_nodes\_out = number\_of\_nodes\_out

def forward(self, x, adj, mask=None):

# Compute embeddings and assignment matrix with GNNs.

z = self.gnn\_embed(x, adj)

s = self.gnn\_pool(x, adj)

# Use the DiffPool operator to get pooled graph.

x, adj, l1, e1 = dense\_diff\_pool(z, adj, s, mask)

return x, adj

# Define the DiffPool layers

self.diff\_pool\_layers = torch.nn.ModuleList()

for number\_of\_nodes\_out, gcn\_layer\_num in diff\_pool\_node\_number\_list:

gnn\_embed = GCN(input\_dim=emb\_dim, hidden\_dim=emb\_dim, output\_dim=emb\_dim, num\_layers=gcn\_layer\_num, dropout=drop\_ratio)

gnn\_pool = GCN(input\_dim=emb\_dim, hidden\_dim=emb\_dim, output\_dim=number\_of\_nodes\_out, num\_layers=gcn\_layer\_num, dropout=drop\_ratio)

diff\_pool\_layer = DiffPool(gnn\_embed=gnn\_embed, gnn\_pool=gnn\_pool, number\_of\_nodes\_out=number\_of\_nodes\_out)

self.diff\_pool\_layers.append(diff\_pool\_layer)

#### **Advantages of using DiffPool on AST**

One limitation of GCN, GraphSage and even GCN + virtual node is that they are flat when pooling for graph-level prediction. Flat hierarchy loses structural information. Oftentimes in text, chunks of text that are close to each other are related in their topic and can be condensed into one subtopic phrase, which themselves can further condense to higher-level abstractions — think outlines. An architecture like DiffPool that can distinguish the different hierarchies should have an advantage [7].

As previously mentioned, LLM-based models and clustering algorithms have underperformed. The Abstract Syntax Tree (AST)’s hierarchical structure of ogbg-code2 has unique properties that make hierarchical pool methods such as DiffPool likely to outperform classical GNNs. Logical components of a function are captured by distinct subtrees of the AST. Nodes closer to the root represent higher-level abstractions in the function, while nodes closer to leaves represent finer granular operations. Additionally, we are interested in generating different levels of coarseness when summarizing the AST, which requires hierarchical pooling.

### **Insights and results**

#### **Our Results vs OGBG Leaderboard**

In total, 32 experiments were conducted, each taking 6 hours to code and train on a V100 GPU. We were initially puzzled at why given the same models, we could not replicate the results seen on the [OGBG-code2 leaderboard](https://ogb.stanford.edu/docs/leader_graphprop/#ogbg-code2). For example, we only got to at most 12.8% with GCNs, and yet the leaderboard claims 14%. The difference is more pronounced when looking at GAT (our 12.8% vs 14.4%). What’s stranger is that when we set our hyperparameters to exactly to the GCN paper code, our results were 3% less on the GCN model (11.1% vs 14.0%). Same for others. After much exploration and review of our code, we realize that the difference lies in the hypermeter tuning. The leaderboard models were able to our more computing resources finding the optimal. Given that we have shown hyperparameters made a huge difference in our ablation studies, we guess that on average there is 2–3% of performance left on the table if we had all of the training resources in hand. However, given each experiment required a V100 GPU 6 hours to run, what we could do was quite expensive already for a small team. It is a shame though, because it would have meant that the methodology our team tried could have achieved as high as 18%, well within the top 5 of the leaderboard.

#### **DiffPool**

****

We went through a few iterations of DiffPool model implementations. In the initial implementation, in each DiffPool layer, we used 2 GCNs with a fixed number of layers to compute the node embeddings and assignment matrix. With some initial experimentations, we realized that the reduction in the number of nodes in each DiffPool layer meant the number of GCN layers may also need to be reduced to prevent over-smoothing. For example, if the 500 nodes of the original graph are pooled into 32 nodes, then the GCN layers needed to produce meaning for embeddings would need to decrease as well. The config for the pooling layers is presented as a sequence of (number of nodes after pooling, number of layers of GNN for the current DiffPool layer). For example, [(128, 5), (32,3)] indicates there are 2 DiffPool layers where the first pools the graph into 128 nodes using 5-layer GNNs, and the second pools the graph into 32 nodes using 3-layer GNNs.

During initial training, we also found that the GPU often encountered out-of-memory issues with larger batch sizes. We realized that the problem lies with the dense adjacency matrix required by the DiffPool operator. We modified the implementation to impose a max number of nodes to limit the adjacency matrix’s size. With the majority of the graphs having less than 500 nodes, we found that using 1500 as the max number of nodes prevented GPU OOMs with little impact on the performance.

The hyperparameter tuning process was difficult for the DiffPool model mainly due to the many floating pieces in the config of the DiffPool pooling layers. Combined with other hyperparameters choices like GNN type, learning rate and batch size, it was difficult to sufficiently explore the space of hyperparameters. With limited resources, we mainly focused on exploring the effect of having different pooling configurations and found that more pooling layers and deeper GNNs degraded the performance. We theorized that the resulting increased complexity of the model from more pooling layers and deeper GNNs resulted in more overfitting. Even with only a single layer of DiffPool with moderate GNN depth, the F1 scores during training were significantly higher than those of testing and validation.



The DiffPool experiments resulted in slightly higher performance than the baseline GCN models. The hierarchical pooling provided some performance improvements for the task. On the other hand, the complexity of the model makes training hard and prone to overfitting.

#### **Ablation studies**

****

For our ablation studies, we started with the most basic GNN model and incrementally added additional complexity to isolate the effect of each.

*Baseline GCN*

The starting point was a 3-layer GCN that summed all output node embeddings into a single prediction head. Both pre-and post-processing steps were simple linear transformations. This exercise not only established a baseline of 5.8% F1 score but also helped us figure out how to build a GNN training pipeline that works end-to-end on this graph-level classification task.

*Hypermeter Tuning*

After establishing a baseline, we optimized hyperparameters before trying more complex model changes. Changing hyperparameters yielded three insights:

1. *Oversmoothing***:** Fewer layers yielded better results. 3 layers performed universally better than 5 layers. This was true across all of the different GNN models. This result suggests to us that over-smoothing is a reality for these graphs because they are on average only 125 nodes.
2. *Lower dropout*: Starting at 0.5 dropout rate, we gradually lowered dropout rates and found that lowering the dropout rate induced more overfitting but also led to higher test-set F1 scores. It could be that the preservation of information outweighs overfitting for the validation and test sets.
3. *Learning adjustments***:** Decreasing the learning rate from 0.01 and a factor of 10 and weight decay by a factor of 1000x improved training stability.

*AST Encoder Pre-Processing and MLP Post-Processing*

Given GCN’s poor performance, we hypothesized that i) a linear transformation of input features was not expressive enough and ii) because the classification task involves global pooling all nodes into a single graph-level prediction head and then splitting it into 5 predictions each of which has thousands of classes, post-processing needs be very sensitive to minute differences in the prediction head.

Therefore, we added a basic AST encoder as a pre-processing step. The AST encoder converts each input feature — node type id, node attribute id, and node depth — into randomly initialized embeddings and then sums them together.

class ASTNodeEncoder(torch.nn.Module):

def \_\_init\_\_(self, emb\_dim, num\_nodetypes, num\_nodeattributes, max\_depth):

super(ASTNodeEncoder, self).\_\_init\_\_()

self.max\_depth = max\_depth

self.type\_encoder = torch.nn.Embedding(num\_nodetypes, emb\_dim)

self.attribute\_encoder = torch.nn.Embedding(num\_nodeattributes, emb\_dim)

self.depth\_encoder = torch.nn.Embedding(self.max\_depth + 1, emb\_dim)

def forward(self, x, depth):

depth[depth > self.max\_depth] = self.max\_depth

return self.type\_encoder(x[:,0]) + self.attribute\_encoder(x[:,1]) + self.depth\_encoder(depth)

AST encoding creates a learnable embedding for each node type and attribute, thereby doubling GCN performance to an F1 score of 12.87%.

Following this success, we had high hopes for what more robust post-processing could do. However, our first attempt with a 5-layer MLP significantly decreased performance. We hypothesized that MLP did not work well on this task because word sequence mattered for method name prediction. Modeling changes were needed as covered below.

*GNN Expressiveness*

Once we noticed a diminished margin of return from non-modeling changes, we carried over those learnings and applied them to more expressive models, namely GraphSAGE [3], GAT [4] (with 3 attention heads), GIN [5], DiffPool, DiffPool + RNN, and GraphSAGE + biased depth pooling. Overall, we noticed that the expressiveness of the models did not correlate one-to-one with gains in performance. In fact, GIN, in theory the most expressive GNN, relatively performed poorer. Several data points also point in this direction. For example, diffPool and extra convolution layers all brought comparable results if not worsened the result. We think that the reason behind this surprising finding is that model expressiveness is not the main challenge in this classification task. The code2 AST graphs are relatively small in size (~100 nodes) and yet their classification size can be one of 5000 vocabularies. The key challenge therefore might be unrelated to how well we capture code structure but instead the relationship between words in the name. The best model in the ablation studies was GraphSAGE with depth-biased pooling performed best at 15.34% F1 score. We then took this model and tried it with a number of augmentation methods to boost performance.

*Depth-biased Pooling*

Because the graphs in the dataset are Syntax Trees, we decided to make adjustments to the node pooling operator to make use of the special structure of the graphs. We hypothesized that within a syntax tree, the embedding of the nodes that have lower depth / closer to the root node should have higher importance because they represent higher-level constructs in a function. When pooling the node embeddings into the graph embedding, we gave nodes exponentially lower weights with respect to the depth of the node while preserving order invariance. Because the node depth is a node feature, the weight given to a node is the same regardless of node ordering.

Empirically, this change provided non-trivial improvement to the model performance, and the best-performing models in our experiments all incorporated this pooling mechanism.

*RNN Prediction Head*

One observation we made about the model’s predictions is that they often do not capture the sequential nature of function names well. To make the prediction sequence better learn the sequencing of words, we decided to add recurrent layers to the prediction head. We added LSTM layers that take in the graph embeddings to generate the prediction sequence, but it degraded the performance of the model. Additional techniques like teacher forcing didn’t yield improvements either.

### **Augmentation studies**

****

*CodeBERT Embeddings*

Witnessing that a basic AST encoding pre-processing step yielded almost 4% boost in F1 score, we wanted to invest further in more complex node augmentation techniques. One of the weaknesses of the AST encoder is that it is a simple shallow embedding of the ID, which loses all semantic information. CodeBERT is a pre-trained multimodal lingual model for programming languages [6]. It was pre-trained on 6 different languages, one of them being Python which is vital to our project as the ogbg-code2 dataset samples are written in Python. Our objective with incorporating CodeBERT is to use the token embeddings output by this model as part of the node embedding initialization in our model. We created a new class called SemanticNodeEncoder. Like the AST encoder, it leverages 3 embedding tables — self.type\_node, self.attribute\_encode, self\_depth\_encoder. Except, at initialization, instead of randomized features, SemanticNodeEncoder calls a get\_embedding function to essentially run all of the node type and node attribute strings through CoderBert to yield an embedding that carries semantic meaning in the embedding space.

class SemanticNodeEncoder(torch.nn.Module):

'''

Use CodeBERT to convert node type strings and node attributes strings into embeddings.

Depth is not used in this embedding.

Input:

emb\_dim: default node feature of N X D

node\_types: list of node type strings [98 type\_string]

node\_attributes: list of node attribute strings [10029 attribute\_string]

depth: The depth of the node in the AST.

Output:

BERTCode-based embedding, which sums the embeddings of the node type

and node attribute strings. Dim: N x 768

'''

def \_\_init\_\_(self, codeBert, tokenizer, emb\_dim, node\_type\_mapping, node\_attributes\_mapping):

super(SemanticNodeEncoder, self).\_\_init\_\_()

...

...

self.type\_encoder = torch.nn.Embedding(len(node\_type\_mapping), emb\_dim)

self.attribute\_encoder = torch.nn.Embedding(len(node\_attributes\_mapping), emb\_dim)

self.depth\_encoder = torch.nn.Embedding(self.max\_depth+1, emb\_dim)

self.type\_encoder.weight = torch.nn.Parameter(self.get\_embedding(node\_type\_mapping))

#self.attribute\_encoder.weight = torch.nn.Parameter(self.get\_embedding(node\_attributes\_mapping))

def get\_embedding(self, mapping):

'''

Input:

mapping: Either list of node type strings [98 type\_string] or list of node attribute strings [10029 attribute\_string]

Output:

BERTCode-based embedding of node attribute strings. Dim: N x 768

'''

...

...

# codeBert steps taken to transform strings into embeddings

tokens = self.tokenizer.tokenize(feature\_string)

tokens\_ids = self.tokenizer.convert\_tokens\_to\_ids(tokens)

tokens\_tensor = torch.tensor(tokens\_ids).to(device)[None,:]

node\_embedding = self.codeBert(tokens\_tensor)[0].view(tokens\_tensor.shape[1], -1)

node\_embedding = node\_embedding.sum(dim=0, keepdim=True) # sum embeddings if >1 word in the feature string

...

...

The resulting node\_encoder of SemanticNodeEncoder class is then passed into a GNN model of our own choosing to encode the node features. This method almost always outperformed the existing model. What we noticed particularly is an improvement in training set accuracy that reached as high as 52%, albeit the overfitting did not impact the testing set as largely. In addition, between node type, node attribute or an aggregation of the two, node attribute was slightly higher than node type and almost 1.5% higher than if we were to sum both together. We believe this because while every function has the same set of operators (e.g. for, while), they have different node attributes (e.g. variable names) that are more closely related to words chosen for the method name.

### **Concluding Lessons and Future Work**

As mentioned above, given that extensive and expensive hyperparameter tuning can make 3% of difference for the same models, the enhancements we have made in this project would mean an equivalent performance boost that is meaningful (F1 18.2%) and places our approach well within top 5 of the leaderboard. While DiffPool did not produce the step function improvement we had hoped, it was among the top performers overall and if given an actual text summarization dataset, its expressive power would have been more useful because the similarity of code structures and the lack of semantic meaning between the text and method name made expressiveness less useful.

It was a really fun and informative experience to work on the project. For the future, we would love to:

1. Use a transformer-based decoder to generate the target sequence of words. In our experiments, we found that LSTM based RNN did not perform well for the task. With self-attention of the transformer, it will likely outperform RNNs and produce more coherent sequences of function name predictions.
2. Further tuning of hyperparameters in the model to reduce overfitting and improve the overall performance.
3. Explore heterogeneous architectures like RGCN. In our experiments, the node types are embedded as node features. The different relationships between node types may be better learned through a heterogeneous model.

### **References**

[1] Alexandre Duval and Fragkiskos Malliaros. Higher-order clustering and pooling for graph neural networks. Conference on Information and Knowledge Management, 2022.

[2] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. Neural Information Pro-cessing Systems, 2018.

[3] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.<https://arxiv.org/abs/1706.02216>

[4] Petar Veliˇckovi ́c, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.<https://arxiv.org/abs/1710.10903>

[5] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2019.<https://arxiv.org/abs/1810.00826>

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.<https://arxiv.org/abs/2002.08155>

[7] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling, 2019. [https://arxiv.org/pdf/1806.08804](https://arxiv.org/pdf/1806.08804.pdf)