Learning Large Graph Property Prediction via Graph Segment Training

Kaidi Cao [1] * Phitchaya Mangpo Phothilimthana [2] Sami Abu-El-Haija [2] Dustin Zelle [2] Yanqi Zhou [2]
Charith Mendis [3] * Jure Leskovec [1] Bryan Perozzi [2]

Abstract

Learning to predict properties of a large graph is challenging because each prediction requires the knowledge of an entire graph, while the amount of memory available during training is bounded. Here we propose Graph Segment Training (GST), a general framework that utilizes a divide-and-conquer approach to allow learning large graph property prediction with a constant memory footprint. GST first divides a large graph into segments and then backpropagates through only a few segments sampled per training iteration. We refine the GST paradigm by introducing a historical embedding table to efficiently obtain embeddings for segments not sampled for backpropagation. To mitigate the staleness of historical embeddings, we design two novel techniques. First, we finetune the prediction head to fix the input distribution shift. Second, we introduce *Stale Embedding Dropout* to drop some stale embeddings during training to reduce bias. We evaluate our complete method GST+EFD (with all the techniques together) on large graph property prediction benchmark MalNet. Our experiments show that GST+EFD is both memory-efficient and fast, while offering a slight boost on test accuracy over a typical full graph training regime.

1. Introduction

The popular graph property prediction tasks deal with relatively small graphs, so the scalability issue arises only from a large number of (small) graphs. However, graph property prediction tasks also face another scalability challenge, which arises due to the large size of each individual graph, as some graphs can have millions or even billions of nodes and edges (Freitas et al., 2021). Training typical Graph Neural Networks (GNNs) to classify such large graphs can be computationally infeasible, as the memory needed scales at least linearly with the size of the graph (Zhang et al., 2022). This presents a challenge as even most powerful GPUs, which are optimized for handling large amounts of data, only have a limited amount of memory available.

In this paper, we address the problem of property prediction of large graphs. We propose Graph Segment Training (GST), which is able to train on large graphs with constant (GPU) memory footprint. Our approach partitions each large graph into smaller segments with a controlled size in the pre-processing phase. During the training process, a random subset of segments is selected to update the model at each step, rather than using the entire graph. This way, we need to maintain intermediate activations for only a few segments for backpropagation; embeddings for the remaining segments are created without saving their intermediate activations. The embeddings of all segments are then combined to generate an embedding for the original large graph, which is used for prediction. Therefore, each large graph has an upper bound on memory consumption during training regardless of its original size. This allows us to train the model on large graphs without running into an out-of-memory (OOM) issue, even with limited computational resources.

To accelerate the training process further, we introduce a historical embedding table to efficiently produce embeddings for graph segments that do not require gradients, as historical embeddings eliminate additional computation on such segments. However, the historical embeddings induce staleness issues during training, so we design two techniques to mitigate such issues in practice. First, we characterize the input distribution mismatch issue between training and test stages of the prediction head, and propose to finetune only the prediction head at the end of training to close the input distribution gap. Second, we identify bias in the loss function due to stale historical embeddings, and introduce *Stale Embedding Dropout* to drop some stale embeddings during training to reduce this bias. Our final proposed method, called GST+EFD, is both memory-efficient and fast.

We evaluate our method on the following datasets: MalNet-

* Work partially done while at Google. [1]Stanford University, California, USA [2]Google, California, USA [3]UIUC, Illinois, USA. Correspondence to: Kaidi Cao <kaidicao@cs.stanford.edu>.

Proceedings of the 2nd Annual Workshop on Topology, Algebra, and Geometry in Machine Learning (TAG-ML) at the 40th International Conference on Machine Learning, Honolulu, Hawaii, USA. 2023. Copyright 2023 by the author(s).

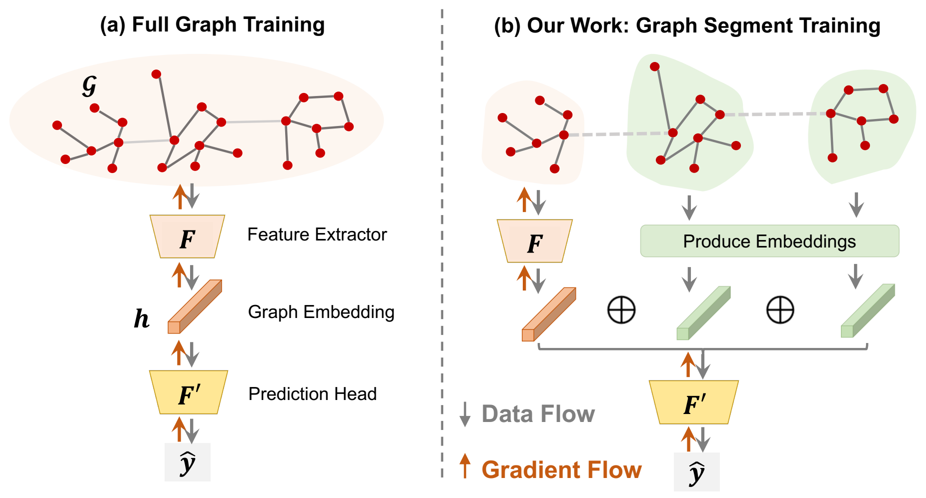

Figure 1. (a) **Full Graph Training**: Classically, models are trained using the entire graph, meaning all nodes and edges of the graph are used to compute gradients. For large graphs, this might be computationally infeasible. (b) **Graph Segment Training**: Our solution is to partition each large graph into smaller segments and select a random subset of segments to update the model; embeddings for the remaining segments are produced without saving their intermediate activations. The embeddings of all segments are combined to generate an embedding for the original large graph, which is then used for prediction.

Tiny, MalNet-Large. A typical full graph training pipeline (Full Graph Training) can only train on MalNet-Tiny, and unavoidably reaches OOM on MalNet-Large. On the contrary, we empirically show that the proposed GST framework successfully trains on arbitrarily large graphs using a single NVIDIA-V100 GPU with only 16GB of memory for MalNet-Large, while maintaining comparable performance with Full Graph Training. We finally demonstrate that our complete method GST+EFD slightly outperforms GST by another 1-2% in terms of final evaluation metric, and simultaneously being 3x faster in terms of training time.

2. Our Method: GST+EFD

2.1. Graph Segment Training (GST)

Given a training graph dataset $\mathcal{D}_{\text{train}} = \{(\mathcal{G}^{(i)}, y^{(i)})\}_{i=1}^{n}$, a common SGD update step requires calculating the gradient:

$$\nabla_\theta \sum_{(\mathcal{G}^{(i)}, y^{(i)}) \in \mathcal{B}} \mathcal{L}((F' \circ F)(\mathcal{G}^{(i)}), y^{(i)})$$

where θ is trainable weights in $F' \circ F$, and $\mathcal{B}$ is a sampled minibatch. Graphs can differ in size (the number of nodes $|\mathcal{V}^{(i)}|$), with some being too large to fit into the device's memory. This is because the memory required to store all intermediate activations for computing gradients is proportional to the number of nodes and edges in the graph.

To address the above issue, we propose to partition each original input graph into a collection of graph segments, *i.e.*,

$$\mathcal{G}^{(i)} \approx \bigoplus \mathcal{G}_j^{(i)} \quad \text{for } j \in \{1, 2, \ldots, J^{(i)}\}$$

The approximation $\approx$ is due to removal of inter-segment edges. An example of a partition algorithm is METIS (Karypis & Kumar, 1997). This preprocessing step will result in a training set $\mathcal{D}_{\text{train}} = \{(\bigoplus_{j \le J^{(i)}} \mathcal{G}_j^{(i)}, y^{(i)})\}_{i=1}^{n}$. Number of partitions $J^{(i)}$ can vary across graphs, but the size of each graph segment can be bounded by a controlled size ($|\mathcal{V}_j^{(i)}| < m_{\text{GST}}, \forall (i,j)$) so that a batch of a fixed number of graph segments can always fit within the device's memory.

When processing graph segment $\mathcal{G}_j^{(i)}$, we can obtain its segment embedding through the backbone: $\boldsymbol{h}_j^{(i)} = F(\mathcal{G}_j^{(i)})$. The prediction head F' requires information from the whole graph to make the prediction, thus we propose to aggregate all segment embeddings to recover the full graph embedding: $\boldsymbol{h}^{(i)} = \bigoplus \boldsymbol{h}_j^{(i)}$. A simple realization of this aggregation is mean pooling. Note that naïvely applying the prediction head F' on the aggregated graph embedding — $\hat{y}^{(i)} = F'(\bigoplus \boldsymbol{h}_j^{(i)})$ — would not provide any reduction in peak memory consumption, as we need to keep track of the activations of all graph segments $\{\mathcal{G}_j^{(i)}\}_{j \in J^{(i)}}$ to perform backpropagation.

Thus, we propose to perform backpropagation on only a few randomly sampled graph segments $\mathcal{S}^{(i)} \subseteq \{1, \ldots, J^{(i)}\}$ and generate embeddings without requiring gradients for the rest. We hereby denote $\boldsymbol{h}_s$ and $\bar{\boldsymbol{h}}_j$ as segment embeddings that require and do not require gradient, respectively. An entire graph embedding is then: $\boldsymbol{h}^{(i)} \approx \{\boldsymbol{h}_s^{(i)}\}_{s \in \mathcal{S}^{(i)}} \bigoplus \{\bar{\boldsymbol{h}}_j^{(i)}\}_{j \notin \mathcal{S}^{(i)}} \triangleq \boldsymbol{h}_s^{(i)} \bigoplus \bar{\boldsymbol{h}}_j^{(i)}$. We name this general pipeline as GST and summarize it in Algorithm 1.

Algorithm 1 General Framework of GST

```
Require: A preprocessed training graph dataset $\mathcal{D}_{\text{train}} = \{(\bigoplus \mathcal{G}_j^{(i)}, y^{(i)})\}_{i=1}^n$. A parameterized backbone $F$ and a prediction head $F'$.
1: for $t$ = 1 to $T_0$ do
2:     $\mathcal{B} \leftarrow$ SampleMiniBatch($\mathcal{D}_{\text{train}}$)
3:     for $(\mathcal{G}^{(i)}, \widehat{y}^{(i)})$ in $\mathcal{B}$ do
4:         $\{\mathcal{G}_s^{(i)}\}_{s\in\mathcal{S}^{(i)}} \leftarrow$ SampleGraphSegments($\mathcal{G}^{(i)}$)
5:         $\bar{\boldsymbol{h}}_j^{(i)} \leftarrow$ ProduceEmbedding($\mathcal{G}_j^{(i)}$) for $j \notin \mathcal{S}^{(i)}$
6:         $\boldsymbol{h}_s^{(i)} \leftarrow F(\mathcal{G}_s^{(i)})$   for $s \in \mathcal{S}^{(i)}$
7:     end for
8:     SGD on loss $\leftarrow \frac{1}{|\mathcal{B}|}\sum_i \mathcal{L}\left(F'(\boldsymbol{h}_s^{(i)} \bigoplus \bar{\boldsymbol{h}}_j^{(i)}), \widehat{y}^{(i)}\right)$
9: end for
```

One implementation of ProduceEmbedding($\cdot$) in Algorithm 1 is to use the same feature encoder F to forward all the segments in $\{\mathcal{G}_j^{(i)}\}_{j\notin\mathcal{S}^{(i)}}$ without storing any intermediate activation (by stopping gradient).

2.2. GST with Historical Embedding Table

Calculating $\bar{\boldsymbol{h}}_j$ by stopping gradient guarantees an upper bound on peak memory consumption. However, since we do not need gradients for segments $\{\mathcal{G}_j^{(i)}\}_{j\notin\mathcal{S}^{(i)}}$, computing forward pass on these segments can be avoided to make training faster. To achieve this, we use historical embeddings acquired in previous training iterations $\tilde{\boldsymbol{h}}_j^{(i)} = \mathcal{T}(i, j)$. With an embedding table $\mathcal{T}$, one can implement ProduceEmbedding($\cdot$) by fetching the corresponding embedding from the table without any computation. We update the embedding table after conducting the forward pass on a graph segment. We optimize the following loss $\mathcal{L}(F'(\boldsymbol{h}_s^{(i)} \bigoplus \tilde{\boldsymbol{h}}_j^{(i)}), y^{(i)})$ during training. We name the embedding version of our algorithm as GST+E.

2.3. Prediction Head Finetuning

Let's compare the input-output distribution of the prediction head F' during the training and inference stage. We have training distribution $\mathcal{P}_{\text{train}}(\boldsymbol{h}, y) = \mathcal{P}(\boldsymbol{h}_s \bigoplus \tilde{\boldsymbol{h}}_j, y)$ and test distribution $\mathcal{P}_{\text{test}}(\boldsymbol{h}, y) = \mathcal{P}(\bigoplus \boldsymbol{h}_j, y)$. Regardless of the innate distribution shift between the training and test stage of the dataset, we note that stale historical embeddings can further widen the gap between the training and test distributions. In this case, the minimizer of the expected training loss does not minimize the expected test loss. To mitigate the distribution misalignment, we introduce the Prediction Head Finetuning technique. Concretely, at the end of training, we update each embedding $\boldsymbol{h}_j^{(i)}$ in the embedding table $\mathcal{T}$ by forwarding each graph segment in the training set with the most current feature encoder F. We then finetune only the prediction head F' with all the input embeddings up-to-date. We use GST+EF to denote GST+E refined with the Prediction Head Finetuning technique.

2.4. Stale Embedding Dropout

The staleness introduces an additional source of bias and variance to the stochastic optimization; the loss function calculated with historical embeddings is no longer an unbiased estimation of its true value. To mitigate the negative impact of historical embeddings on loss function estimation, we propose the second technique, *Stale Embedding Dropout* (SED). Unlike a standard Dropout, which uniformly drops elements and weighs up the rest, we propose to drop only stale segment embeddings and weigh up only segment embeddings that are up-to-date. Concretely, assume with the keep probability p, the weight η for each segment is defined as:

$$\eta^{(i)} = \begin{cases} p + (1-p)\frac{J^{(i)}}{S^{(i)}} & \text{for } \mathcal{G}_s^{(i)} \\ 0 & \text{for } \mathcal{G}_j^{(i)}, \text{ with prob. } (1-p) \\ 1 & \text{for } \mathcal{G}_j^{(i)}, \text{ with prob. } p \end{cases} \tag{1}$$

Please refer to the theoretical analysis in the appendix. By combining the two proposed techniques, we denote our final algorithm as GST+EFD.

3. Experiments

3.1. Empirical Results on MalNet

To demonstrate the general applicability of our proposed GST framework, we consider three backbones, namely, GCN (Kipf & Welling, 2016), SAGE (Hamilton et al., 2017), and GraphGPS (Rampášek et al., 2022). GCN and SAGE are two popular GNN architectures. GraphGPS is a Graph Transformer that recently achieves state-of-the-art performance on many graph-level tasks, but is well-known for its issue on scalability. We report the top-1 test accuracy of various methods on MalNet-Tiny and MalNet-Large in Table 1. We include MalNet-Tiny in this study because its graphs are relatively small so that it is still possible to run Full Graph Training.

Notably, we observe that GST slightly outperforms Full Graph Training in terms of test accuracy on MalNet-Tiny. GST has exactly the same number of weight parameters with Full Graph Training. This implies that GST potentially has a better hierarchical graph pooling mechanism that leads to better generalization. As we step from MalNet-Tiny to MalNet-Large, Full Graph Training strategy can no longer fit the large graphs on a GPU, so we report OOM in the table. GST's estimation on graph segment embeddings $\bar{\boldsymbol{h}}_j^{(i)}$ that do not require gradients is accurate, and thus does not suffer from staleness issues. Therefore, we use GST as an

Table 1. Test accuracy on MalNet-Tiny and MalNet-Large. We report the standard deviation over five runs. GST+EFD achieves better accuracy than Full Graph Training, and GST, while being much more memory efficient and computationally faster.

Dataset	MalNet-Tiny			MalNet-Large		
Backbone	GCN	SAGE	GraphGPS	GCN	SAGE	GraphGPS
Full Graph Training	87.84±1.37	88.08±1.68	90.82±0.59	OOM	OOM	OOM
GST	88.26±0.80	88.42±1.03	91.03±0.81	88.35±1.14	88.62±0.82	91.39±0.85
GST-One	71.62±3.85	72.64± 4.73	77.63±3.15	60.41±6.29	57.13±7.36	66.82±4.71
GST+E	86.53±1.18	86.82±0.93	89.75±0.89	48.42±6.61	43.28±7.01	62.47±3.19
GST+EF	87.67±0.78	87.83±0.81	90.52±0.71	84.83±0.96	85.26±0.87	91.33±0.65
GST+ED	88.18±0.48	88.50±0.74	90.96±0.68	82.17±4.74	71.83±6.31	89.46±1.36
GST+EFD	88.78±0.45	89.24±0.53	92.46± 0.66	89.67±0.71	89.78±0.68	92.52±0.58

estimation for the performance of Full Graph Training on MalNet-Large.

Naïvely training on only one graph segment (GST-One) yields inferior performance than Full Graph Training and GST, showing that it is essential to aggregate embeddings from all graph segments. Solely introducing the historical embedding table (GST+E) significantly deteriorates the optimization due to the staleness issue. Each of the proposed techniques (Prediction Head Finetuning and SED) individually is beneficial in combating the staleness issue. The combination of our two techniques (GST+EFD) achieves the best performance, slightly outperforming GST by another 1-2% in terms of final evaluation metric.

3.2. Runtime Analysis

Next, we empirically compare runtime of different variants under the proposed GST framework. We summarize an average time for one forward-backward pass during training on MalNet-Large dataset in Table 2. Since GST runs inference for the graph segments that do not require gradients, the runtime of GST is significantly higher than others'. We also found that GST+E's and GST+EFD's runtime are very close to GST-One's; this means the overhead of fetching embeddings from the embedding table $\mathcal{T}$ is minimal. Moreover, GST+EFD's runtime is slightly lower than GST+E's because in the implementation, we can skip the fetching process if an embedding is set to be dropped. This result demonstrates that our proposed GST+EFD not only is efficient in terms of memory usage but also reduces training time significantly.

Table 2. Runtime analysis (average training time per iteration in ms) on MalNet-Large dataset.

	GCN	SAGE	GraphGPS
GST	720.8	706.3	1285.7
GST-One	242.2	239.6	441.4
GST+E	258.4	253.6	451.9
GST+EFD	247.9	244.7	448.2

References

Freitas, S., Dong, Y., Neil, J., and Chau, D. H. A large-scale database for graph representation learning. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

Karypis, G. and Kumar, V. Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 1997.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. 2016.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Rampášek, L., Galkin, M., Dwivedi, V. P., Luu, A. T., Wolf, G., and Beaini, D. Recipe for a general, powerful, scalable graph transformer. *arXiv preprint arXiv:2205.12454*, 2022.

Wei, C., Kakade, S., and Ma, T. The implicit and explicit regularization effects of dropout. In *International conference on machine learning*, pp. 10181–10192. PMLR, 2020.

You, J., Ying, Z., and Leskovec, J. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33:17009–17021, 2020.

Zhang, H., Yu, Z., Dai, G., Huang, G., Ding, Y., Xie, Y., and Wang, Y. Understanding gnn computational graph: A coordinated computation, io, and memory perspective. *Proceedings of Machine Learning and Systems*, 4:467–484, 2022.

A. Proofs and Derivations

Theorem A.1. *Under proper a condition that $W \cdot \tilde{\boldsymbol{h}}_j^{(i)} \approx 0$ and $W \cdot \boldsymbol{h}_j^{(i)} \approx 0$, where W is the first linear transformation in F', SED with a keep ratio p ensures to reduce bias term introduced by historical embeddings by a factor of p, while introducing another regularization term.*

Proof. Let $\delta^{(i)} \triangleq \boldsymbol{h}_s^{(i)} \bigoplus \tilde{\boldsymbol{h}}_j^{(i)} - \bigoplus \boldsymbol{h}_j^{(i)}$ be the perturbation on the graph embedding. We use ET to denote using the embedding table without applying SED. For GST+E, we have

$$\delta_j^{(i)^{\text{ET}}} = \begin{cases} 0 & \text{with prob. } \frac{S^{(i)}}{J^{(i)}} \\ \tilde{\boldsymbol{h}}_j^{(i)} - \boldsymbol{h}_j^{(i)} & \text{with prob. } \frac{J^{(i)}-S^{(i)}}{J^{(i)}} \end{cases}$$

The randomness above comes from the fact that each segment $\mathcal{G}_j^{(i)}$ is selected for backpropagation with probability $\frac{S^{(i)}}{J^{(i)}}$.

For SED, there are two folds of randomness when training on graph $\mathcal{G}^{(i)}$: randomly selecting $S^{(i)}$ segments to train and randomly select stale embedding $\tilde{\boldsymbol{h}}_j^{(i)}$ to drop. Thus we can rewrite $\delta_j^{(i)}$ as

$$\delta_j^{(i)^{\text{SED}}} = \begin{cases} \frac{(1-p)(J^{(i)}-S^{(i)})}{S^{(i)}} \boldsymbol{h}_j^{(i)} & \text{with prob. } \frac{S^{(i)}}{J^{(i)}} \\ -\boldsymbol{h}_j^{(i)} & \text{with prob. } \frac{(1-p)(J^{(i)}-S^{(i)})}{J^{(i)}} \\ \tilde{\boldsymbol{h}}_j^{(i)} - \boldsymbol{h}_j^{(i)} & \text{with prob. } \frac{p(J^{(i)}-S^{(i)})}{J^{(i)}} \end{cases}$$

We apply Taylor expansion around $\delta_j^{(i)} = 0$ on the final loss to analyze the effect of this perturbation. In Section A.2 of Wei et al. (2020), when $W \cdot \tilde{\boldsymbol{h}}_j^{(i)} \approx 0$ and $W \cdot \boldsymbol{h}_j^{(i)} \approx 0$, the perturbation of $\delta^{(i)}$ to the loss function might not be too large, so it supports the use of Taylor Expansion.

$$\begin{aligned} &\mathcal{L}(F'(\boldsymbol{h}_s^{(i)} \bigoplus \tilde{\boldsymbol{h}}_j^{(i)})) - \mathcal{L}(F'(\bigoplus \boldsymbol{h}_j^{(i)})) \\ =&\mathcal{L}(F'(\bigoplus(\boldsymbol{h}_j^{(i)} + \delta_j^{(i)}))) - \mathcal{L}(F'(\bigoplus \boldsymbol{h}_j^{(i)})) \\ \approx& \sum_j D_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}]\delta_j^{(i)} + \frac{1}{2}\delta_j^{(i)^\top}(D^2_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}])\delta_j^{(i)} \end{aligned}$$

Note that we randomly select segments with index s during training, we can then derive an approximation of the expected difference during training as

$$\begin{aligned} &\mathbb{E}_s \mathcal{L}(F'(\boldsymbol{h}_s^{(i)} \bigoplus \tilde{\boldsymbol{h}}_j^{(i)})) - \mathcal{L}(F'(\bigoplus \boldsymbol{h}_j^{(i)})) \\ \approx& \sum_j \mathbb{E}_{\delta_j^{(i)}} \underbrace{D_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}]\delta_j^{(i)}}_{B} + \underbrace{\frac{1}{2}\delta_j^{(i)^\top}(D^2_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}])\delta_j^{(i)}}_{R} \end{aligned} \tag{2}$$

We can then compare the effect of SED by substituting the two versions of $\delta_j^{(i)}$ into Eq. 2.

So for the first term, we have

$$\begin{aligned} \mathbb{E}_{\delta_j^{(i)^{\text{ET}}}}[B] &= \langle D_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}], \mathbb{E}\delta_j^{(i)} \rangle \\ &= \langle D_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}], \frac{J^{(i)} - S^{(i)}}{J^{(i)}} \mathbb{E}(\tilde{\boldsymbol{h}}_j^{(i)} - \boldsymbol{h}_j^{(i)}) \rangle \\ \mathbb{E}_{\delta_j^{(i)^{\text{SED}}}}[B] &= \langle D_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}], \mathbb{E}\delta_j^{(i)} \rangle \\ &= \langle D_{\boldsymbol{h}_j^{(i)}}(\mathcal{L} \circ F')[\boldsymbol{h}_j^{(i)}], \frac{J^{(i)} - S^{(i)}}{J^{(i)}} \mathbb{E}(\tilde{\boldsymbol{h}}_j^{(i)} - \boldsymbol{h}_j^{(i)}) * p \rangle \end{aligned}$$

whereas for the second term, we have

$$\begin{aligned}
\mathbb{E}_{\delta_j^{(i)\mathrm{ET}}}[R] =& \langle D^2_{\boldsymbol{h}_j^{(i)}}(\mathcal{L}\circ F')[\boldsymbol{h}_j^{(i)}], \frac{\mathbb{E}\delta_j^{(i)}\delta_j^{(i)^\top}}{2}\rangle \\
=& \langle D^2_{\boldsymbol{h}_j^{(i)}}(\mathcal{L}\circ F')[\boldsymbol{h}_j^{(i)}], \frac{J^{(i)}-S^{(i)}}{2J^{(i)}}(\tilde{\boldsymbol{h}}_j^{(i)}-\boldsymbol{h}_j^{(i)})^{\odot 2}\rangle \\
\mathbb{E}_{\delta_j^{(i)\mathrm{SED}}}[R] =& \langle D^2_{\boldsymbol{h}_j^{(i)}}(\mathcal{L}\circ F')[\boldsymbol{h}_j^{(i)}], \frac{\mathbb{E}\delta_j^{(i)}\delta_j^{(i)^\top}}{2}\rangle \\
=& \langle D^2_{\boldsymbol{h}_j^{(i)}}(\mathcal{L}\circ F')[\boldsymbol{h}_j^{(i)}], (\frac{(J^{(i)}-S^{(i)})p}{2J^{(i)}}(\tilde{\boldsymbol{h}}_j^{(i)}-\boldsymbol{h}_j^{(i)})^{\odot 2} \\
&+\frac{(J^{(i)}-S^{(i)})(1-p)(J^{(i)}-pJ^{(i)}+pS^{(i)})}{2J^{(i)}S^{(i)}}\boldsymbol{h}_j^{(i)\odot 2})\rangle
\end{aligned}$$

It is easy to check that the statement satisfies given the value calculated. □

B. Implementation Details

Datasets. MalNet (Freitas et al., 2021) is a large-scale graph representation learning dataset, with the goal to predict the category of a function call graph. MalNet is the largest public graph database constructed to date in terms of average graph size. Its widely-used split is called *MalNet-Tiny*, containing 5,000 graphs across balanced 5 types, with each graph containing at most 5,000 nodes. To evaluate our approach on the regime where the graph size is large, we construct an alternative split from the original MalNet dataset, which we named *MalNet-Large*. *MalNet-Large* also contains 5,000 graphs across balanced 5 types. *MalNet-Large*'s average graph size reaches 47k with the largest graph containing 541k nodes. We will release our experimental split for *MalNet-Large* to promote future research.

Methods. We test combinations of the following proposed techniques and some baselines. (1) Full Graph Training: we train on all graphs in their original scale without applying any partitioning beforehand. (2) GST-One: we partition the original graph into a collection of graph segments $\mathcal{G}^{(i)} \approx \bigoplus \mathcal{G}_j^{(i)}$, but we randomly select only one segment $\mathcal{G}_j^{(i)}$ for each graph to train every iteration. (3) GST: following the general GST framework described in Algorithm 1, we replace ProduceEmedding$(\cdot)$ by using the same feature encoder F to forward all the segments in $\{\mathcal{G}_j^{(i)}\}_{j\notin\mathcal{S}^{()}}$ without storing any intermediate activation. We set $S^{(i)}=1$ in our experiments. (4) E: we introduce an embedding table $\tilde{\boldsymbol{h}}_j^{(i)} = \mathcal{T}(i,j)$ to store the historical embedding of each graph segment, and we fetch the embedding from $\mathcal{T}$ if we do not need to calculate gradient for the corresponding segment. (5) F: in addition to introducing the embedding table $\mathcal{T}$, we finetune the prediction head F' with all up-to-date segment embeddings at the end of training. (6) D: we apply SED defined in Eq. 1 during training.

When these techniques are combined, we concatenate the acronyms with a "+" to GST as an abbreviation. We conduct all the experiments on MalNet with a single NVIDIA-V100 GPU with 16GB of memory. Please refer to Appendix B for additional implementation details.

Table 3. Overview of the graph datasets used in this study.

	Avg. # nodes	Min. # nodes	Max. # nodes	Avg. # edges	Min. # edges	Max. # edges
MalNet-Tiny	1,410	5	4,994	2,860	4	20,096
MalNet-Large	47,838	3,374	541,571	225,474	20,597	3,278,318

We follow GraphGym (You et al., 2020) to represent design spaces of GNN as (message passing layer type, number of pre-process layers, number of message passing layers, number of post-process layers, activation, aggregation). Our code is implemented in PyTorch (Paszke et al., 2017). We will make source code public at the time of publication.

Implementation details for MalNet-Large. We consider three model variations for the MalNet-Large dataset. Please refer to their hyperparameters in Table 4. We use Adam optimizer (Kingma & Ba, 2014) with the base learning rate of 0.01 for GCN and SAGE. For GraphGPS, we use AdamW optimizer (Loshchilov & Hutter, 2017) with the cosine scheduler

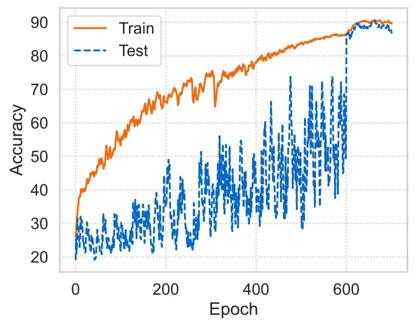

Figure 2. Accuracy curve on MalNet-Large of GST+EFD with SAGE backbone. We start Prediction Head Finetuning at epoch 600.

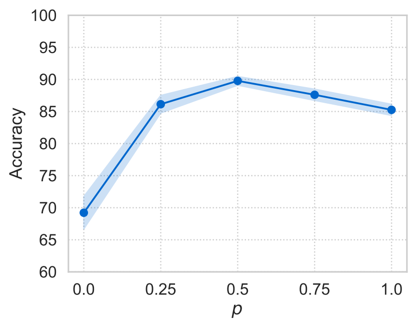

Figure 3. Ablation study on the keep ratio p in SED. We report test accuracy of GST+EFD with SAGE backbone on MalNet-Large for 5 runs.

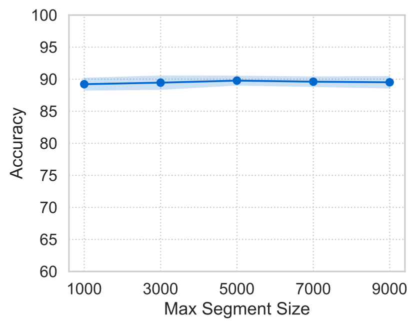

Figure 4. Ablation study on maximum segment size. We report test accuracy of GST+EFD with SAGE backbone on MalNet-Large for 5 runs.

and the base learning rate of 0.0005. We use L2 regularization with a weight decay of 1e-4. We train for 600 epochs until convergence. For Prediction Head Finetuning, we finetune for another 100 epochs. We limit the maximum segment size to 5,000 nodes, and use a keep probability $p = 0.5$ if not otherwise specified. We train with CrossEntropy loss.

Table 4. Detailed GNN/Graph Transformer designs used in MalNet-Tiny and MalNet-Large.

model	GCN	SAGE	GraphGPS
message passing layer type	GCNConv	SAGEConv	GatedGCN+Performer
pre-process layer num.	1	1	0
message passing layer num.	2	2	5
post-process layer num.	1	1	3
hidden dimension	300	300	64
activation	PReLU	PReLU	ReLU
aggregation	mean	mean	mean

Implementation details for MalNet-Tiny. We use the same model architectures/training schedules as in the MalNet-Large dataset. The only difference is that as graphs in MalNet-Tiny have no more than 5000 nodes, so we limit maximum segment size to 500 here.

C. Additional Results

C.1. Ablation Studies

Effect of finetuning. We visualize the training/test accuracy curve of GST+EFD over time in Figure 2. The staleness introduced by historical embeddings drastically hurts generalization, as shown for the first 600 epochs. We start finetuning at epoch 600, and the gap between training and test accuracy decreases by a large margin instantly.

Ablation study on segment dropout ratio. To analyze the effect of the keep ratio p in SED, we vary its value from 0 to 1 and visualize the results in Figure 3. When $p = 1$, GST+EFD degrades back to using the historical embedding table without SED, as the performance decreases due to staleness. When $p = 0$, GST+EFD becomes GST-One, where we drop all the stale historical embeddings. This extreme case introduces too heavy regularization that impedes the model from fitting the training data, leading to a decrease in test performance ultimately. We found that $p = 0.5$ achieves a satisfactory tradeoff between fitting the training data and adding a proper amount of regularization.

Ablation study on segment size. We also alter the maximum segment size and visualize the results in Figure 4. A smaller maximum segment size will result in much more number of segments. Interestingly, we found that the proposed GST+EFD is very robust to the choice of the maximum segment size, as long as the segment size is reasonally large.

C.2. Runtime Analysis

Next, we empirically compare runtime of different variants under the proposed GST framework. We summarize an average time for one forward-backward pass during training on MalNet-Large dataset in Table 2. Since GST runs inference for the graph segments that do not require gradients, the runtime of GST is significantly higher than others'. We also found that GST+E's and GST+EFD's runtime are very close to GST-One's; this means the overhead of fetching embeddings from the embedding table $\mathcal{T}$ is minimal. Moreover, GST+EFD's runtime is slightly lower than GST+E's because in the implementation, we can skip the fetching process if an embedding is set to be dropped. This result demonstrates that our proposed GST+EFD not only is efficient in terms of memory usage but also reduces training time significantly.