Appendices

The contents of the appendices are organized as follows:

- In Appendix A, we present more details about SLOG.
  - In Appendix A.1, we present a detailed proof of Proposition 3.1.
  - In Appendix A.2, we present an illustration of residual connections in SLOG(N).
  - In Appendix A.3, we present the pseudo code of SLOG(L).
  - In Appendix A.4, we present a complexity analysis of SLOG.
- In Appendix B, we present additional experiment results.
  - In Appendix B.1, we present ablation studies.
  - In Appendix B.2, we present comparison results between SLOG and additional baselines.
  - In Appendix B.3, we present experimental results on synthetic datasets.
  - In Appendix B.4, we present the convergence study.
  - In Appendix B.5, we present the results obtained using two alternative optimization methods.
    - In Appendix B.5.1, we present the results of Alternating Minimization.
    - In Appendix B.5.2, we present the results of Gaussian Process optimization.
- In Appendix C, we present the experimental details.
  - In Appendix C.1, we present statistics of datasets used.
  - In Appendix C.2, we present the details on model implementations.
  - In Appendix C.3, we present the details on METIS graph partition method.

A. Details of Proposed Model

A.1. Proof of Proposition 3.1

Proposition 3.1. The SLOG's filter with real-valued order, $\omega(\cdot)$, in the spectral domain can be regarded as the combination of two linear graph convolutional networks in the spatial domain: $\omega(\mathbf{L}_{\text{sym}}) = \mathbf{S}_1^p \cdot \mathbf{S}_2^q$, where $\mathbf{S}_1 = \frac{1}{2}(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})$ and $\mathbf{S}_2 = \mathbf{I} + (\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})^2$.

Proof. The filter with real-valued order, $\omega(\cdot)$, is defined as follows:

$$\omega(\mathbf{L}_{\text{sym}}) = (\mathbf{I} - \frac{1}{2}\mathbf{L}_{\text{sym}})^p(\mathbf{I} + (\mathbf{L}_{\text{sym}} - \mathbf{I})^2)^q, \tag{8}$$

We define $\mathbf{S}_1 = \frac{1}{2}(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})$, $\mathbf{S}_2 = \mathbf{I} + (\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})^2$. Since $\mathbf{L}_{\text{sym}} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$, we have:

$$\begin{aligned}
\mathbf{S}_1 &= \mathbf{I} - \frac{1}{2}\mathbf{L}_{\text{sym}} = \mathbf{U}(\mathbf{I} - \frac{1}{2}\mathbf{\Lambda})\mathbf{U}^\top \\
&= \mathbf{I} - \frac{1}{2}(\mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}) = \frac{1}{2}(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}), \\
\mathbf{S}_2 &= \mathbf{I} + (\mathbf{L}_{\text{sym}} - \mathbf{I})^2 = \mathbf{I} + (\mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} - \mathbf{I})^2 \\
&= \mathbf{I} + (\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})^2.
\end{aligned}$$

Then, we have:

$$\omega(\mathbf{L}_{\text{sym}}) = \mathbf{S}_1^p \cdot \mathbf{S}_2^q. \tag{9}$$

Similar to d in TeDGCN (Yan et al., 2023), p and q can be explained as two real-valued depths of the graph convolution networks $\mathbf{S}_1$ and $\mathbf{S}_2$. □

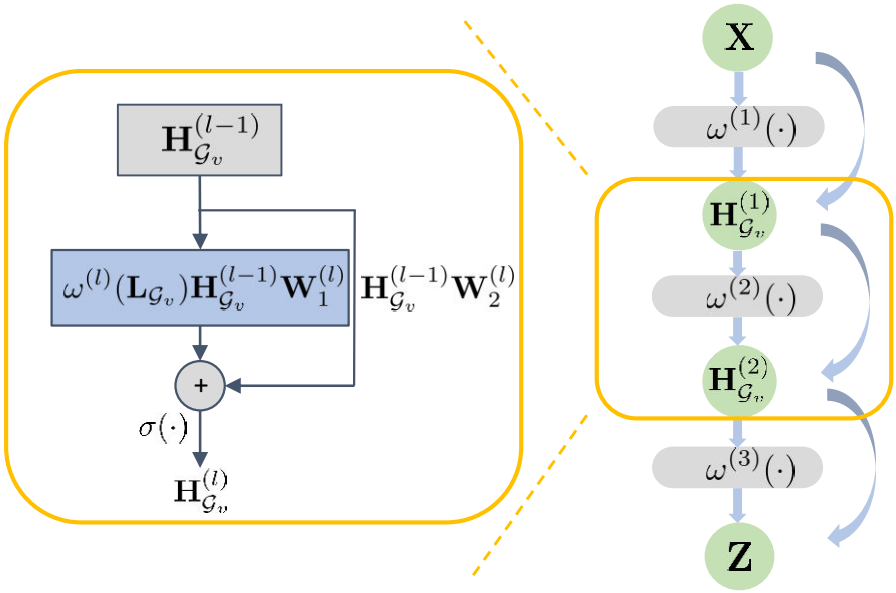


Figure 5. Residual connection in SLOG(N).

A.2. Illustration of Residual Connections in SLOG(N)

In Section 3.2, we introduce the SLOG(N), which adds more non-linearity and residual connections. In each layer, the representation of nodes is updated by Eq. (6). Here, we present a figure to further illustrate the residual connection structure, as shown in Figure 5.

A.3. Pseudo Code of SLOG(L)

Algorithm 2 SLOG(LB) Algorithm

Input: Training Graph $\mathcal{G}_T(\mathcal{V}_T, \mathcal{E}_T)$; evaluating nodes $\mathcal{V}_E$, corresponding edges $\mathcal{E}_E$; input features $\mathbf{X}_T, \mathbf{X}_E$; hop number of subgraphs K; maximum neighbor numbers of each depth $\{N_i\}$
Output: Vector representations $\mathbf{z}_v$ for node $v \in \mathcal{V}_E$
 # Step 1: *Graph partition*
Utilize METIS to partition $\mathcal{G}_T$ into M subgraphs $\{\mathcal{G}_{T_i}(\mathcal{V}_{T_i}, \mathcal{E}_{T_i})\}_{i=1}^M$.
Initialize parameters of global filter $(p_{\text{glo}}, q_{\text{glo}})$ and local filters $\{(p_i, q_i)\}_{i=1}^M$
 # Step 2: *Subgraph expansion and training*
for $i = 1$ to M **do**
  $\mathcal{G}'_{T_i} \leftarrow SAMPLE(\mathcal{G}_{T_i}, \mathcal{V}_{T_i}, K, \{N_i\})$
  Obtain corresponding filter ω_i by Eq. (7)
  Obtain nodes representations $\mathbf{Z}_{\mathcal{G}'_{T_i}}$ of $\mathcal{G}'_{T_i}$ by Eq. (5)
  Normalize representations in $\mathcal{G}_{T_i}$ with Softmax function and optimize Cross-entropy loss
end for
 # Step 3: *Minimum distance subgraph search & prediction*
$\mathcal{G} \leftarrow (\mathcal{V}_T \cup \mathcal{V}_E, \mathcal{E}_T \cup \mathcal{E}_E)$
for $v \in \mathcal{V}_E$ **do**
  Calculate distance from v to each $\mathcal{G}_i$ and get $\{dist_i\}$
  $\mathcal{G}_v \leftarrow \mathcal{G}_{T_i}$, **s.t.** $dist_i$ is the minimum
  $\mathcal{G}_v \leftarrow SAMPLE(\mathcal{G}, \{v\} \cup \mathcal{V}_{T_v}, K, \{N_i\})$
  Obtain corresponding filter ω_i by Eq. (7)
  Obtain nodes representations $\mathbf{Z}_{\mathcal{G}_v}$ of $\mathcal{G}_v$ by Eq. (5)
  $\mathbf{z}_v \leftarrow \mathbf{Z}_{\mathcal{G}_v}(v)$
end for
return $\{\mathbf{z}_v | v \in \mathcal{V}_E\}$

A.4. Complexity Analysis

In this subsection, we conduct a brief complexity analysis of SLOG(B) and SLOG(N). We base our analysis on a one-hop neighbor sampling strategy, setting the maximum number of sampled neighbors (k) empirically to no more than 25. This setup leads to a time complexity of $O(k)$ per batch. For SLOG(B), if we utilize a single node for one batch, we have N batches to train the model, where N is the number of nodes. The computational procedures for each batch can be categorized into two main steps: (1) eigenvalue decomposition and (2) message passing. Regarding (1), the eigenvalue decomposition of $\mathbf{L}_{\text{sym}}$ takes $O(k^3)$ time. In terms of (2), by Eq. (1), the computation of $\omega(\mathbf{L}_{\text{sym}})$ requires $O(k^3)$ time. Then, matrix multiplications cost $O(k^2 f + kfc)$ time, where f is the feature dimension, and c is the class number. Consequently, the time complexity for a single batch in SLOG(B) is $O(k^3 + k^2 f + kfc)$, and for the entire graph, it escalates to $O(N(k^3 + k^2 f + kfc))$. For SLOG(N) with L layers, the time complexity is $O(NL(k^3 + k^2 f + kfc))$.

B. Additional Experiment

B.1. Ablation studies

We conduct an ablation study on SLOG(B), which is detailed in Table 5, and assess the impact of each component. Results of the first two rows reveal that omitting any term from Eq. (4) reduces performance, emphasizing the importance of each element. Additionally, the sampling strategy plays a crucial role in enhancing performance. Without this strategy, the model would not only encounter a drop in performance but also become unsuitable for application to large-scale datasets, including Flickr, Ogbn-arxiv, and Reddit.

Table 5. Ablation Study for SLOG(B) Model.

Datasets	Cora	Chameleon	Minesweeper
SLOG(B)	**0.865±0.011**	**0.581±0.024**	**0.822±0.009**
w/o. $\mathbf{S}_1$	0.854±0.016	0.570±0.015	0.803±0.004
w/o. $\mathbf{S}_2$	0.838±0.013	0.576±0.030	0.818±0.008
w/o. sampling	0.856±0.017	0.568±0.013	0.802±0.008

To test the effectiveness of subgraph sampling strategy on other baseline methods, we equip ChebNet with subgraph sampling. The results are shown in Table 6. We can observe that the subgraph sampling strategy can indeed be beneficial to other baseline methods as well. However, the proposed SLOG still outperforms the baseline methods with subgraph sampling strategy like ChebNet.

Table 6. Evaluation results on different datasets in the inductive setting.

Dataset	Cora	Citeseer	Squirrel	Chameleon	Tolokers
ChebNet	0.804±0.004	0.740±0.009	0.350±0.004	0.535±0.005	0.783±0.003
w. sampling	0.829±0.013	0.754±0.010	0.356±0.013	0.467±0.000	0.784±0.003
SLOG(B)	0.865±0.011	0.766±0.026	0.392±0.006	0.581±0.024	0.796±0.005

B.2. Additional Baselines

In inductive setting, we compare SLOG(B)/SLOG(N) with CayleyNet (Levie et al., 2018), GIN (Xu et al., 2018), ARMA (Bianchi et al., 2021), ChebNetII (He et al., 2022), TeDGCN (Yan et al., 2023), PyGNN (Geng et al., 2023), FavardGNN (Guo & Wei, 2023), OptBasisGNN (Guo & Wei, 2023) with the same random split in Section 4. The results can be found in Table 7. It is observed that the latest baseline methods, such as ChebNetII, TeDGCN, can produce competitive results. However, SLOG still outperforms all the baselines among most of the datasets. It reveals the effectiveness of our proposed method.

B.3. Synthetic Datasets

To demonstrate the robustness of SLOG under different homophily/heterophily ratios, we generate synthetic datasets with varying heterophily ratios (h) from 0.1 to 0.9, following the method described in (Zhu et al., 2020). The higher the h value, the more heterophilic the graph is. Node features are sampled from the Ogbn-arxiv dataset (Hu et al., 2020).

Table 7. Evaluation results on different datasets in the inductive setting.

Datasets	Cora	Citeseer	Squirrel-filt.	Chameleon-filt.	Minesweeper	Tolokers	Amazon-ratings
CayleyNet	0.773±0.007	0.685±0.017	0.347±0.027	0.264±0.022	0.798±0.004	0.781±0.008	0.265±0.003
GIN	0.808±0.022	0.700±0.009	0.327±0.009	0.348±0.042	0.788±0.001	0.775±0.002	0.420±0.007
ARMA	0.757±0.026	0.735±0.009	0.364±0.010	0.343±0.034	0.794±0.012	0.779±0.007	0.428±0.004
ChebNetII	**0.868±0.011**	0.717±0.011	0.354±0.013	0.367±0.018	0.759±0.016	0.783±0.004	0.396±0.002
TeDGCN	0.825±0.005	0.751±0.003	0.409±0.031	0.422±0.022	0.793±0.001	0.796±0.002	0.425±0.002
PyGNN	0.863±0.011	0.742±0.007	0.349±0.011	0.397±0.030	0.787±0.003	0.789±0.001	**0.456±0.005**
FavardGNN	0.850±0.007	0.749±0.018	0.383±0.015	0.406±0.019	0.805±0.004	0.786±0.002	0.397±0.004
OptBasisGNN	0.856±0.005	0.753±0.006	0.392±0.007	0.400±0.043	0.789±0.001	0.775±0.000	0.368±0.000
SLOG(B)	0.865±0.011	**0.766±0.026**	**0.427±0.013**	0.420±0.023	0.822±0.009	0.796±0.005	0.451±0.007
SLOG(N)	0.761±0.010	0.675±0.026	0.376±0.025	**0.431±0.026**	**0.844±0.008**	**0.810±0.006**	**0.456±0.006**

To be specific, we initialize a small graph, and recursively add new nodes to the graph. The class assignment for the newly added nodes is determined randomly from a predefined set of class numbers, denoted as C. The probability of an edge between the new node and the existing nodes in the graph is determined by the classes that the nodes belong to, the heterophily ratio h, and the current degrees of existing nodes. Once the graph scale reaches the target scale, we assign each node a feature vector from a real dataset. It is ensured that nodes belonging to the same class in the generated graph have feature vectors originating from nodes in the same class in the real dataset. The statistics of generated datasets are shown in Table 8.

Table 8. The statistics of synthetic datasets.

Datasets	Nodes	Edges	Features	Classes	Heterophily
syn-arxiv	2,000	∼20,000	128	40	0.1 to 0.9

The experiments are conducted under inductive settings across 5 runs for each model on each dataset, and each dataset is randomly splited into 60%/20%/20% train/validation/test split. The results are shown in Table 9.

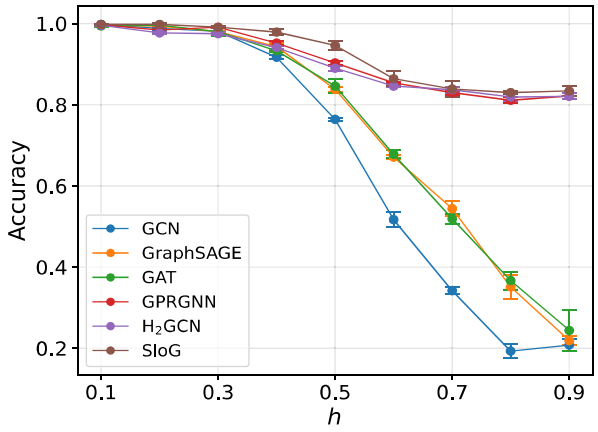


Figure 6. Performance comparison on synthetic datasets with different heterophily ratios h.

Figure 6 shows the performance comparison of SLOG(B) and several baselines on synthetic datasets of different h values. It can be seen that our method demonstrates superior performance on synthetic datasets across different heterophily ratios h, indicating its robustness to varying heterophilic/homophilic balances. Notably, general GNN methods (e.g., GCN, GraphSAGE, GAT) show high sensitivity to h, especially under high h values where their performance drops significantly, suggesting their limitation in heterophilic contexts. In contrast, heterophilic graph-oriented methods (e.g., GPRGNN, H_2GCN) show more robustness. Yet our method still outperforms them, demonstrating our method's ability to adaptively

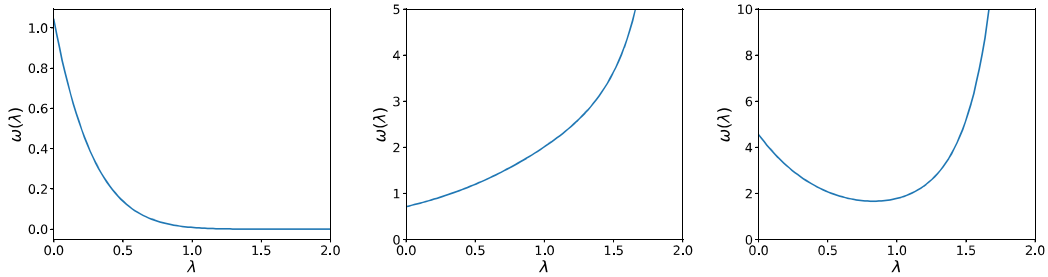
Table 9. Performance comparison on synthetic datasets.

h	0.1	0.2	0.3	0.4	0.5
MLP	0.771±0.018	0.803±0.005	0.826±0.007	0.817±0.004	0.857±0.006
GCN	**0.998±0.002**	0.988±0.005	0.981±0.007	0.917±0.005	0.764±0.003
ChebNet	0.845±0.004	0.823±0.002	0.856±0.019	0.818±0.005	0.817±0.005
GraphSAGE	0.996±0.003	0.994±0.002	0.980±0.002	0.943±0.009	0.838±0.006
GAT	0.994±0.003	0.996±0.002	0.980±0.010	0.933±0.003	0.846±0.017
APPAP	0.996±0.003	0.975±0.004	0.962±0.005	0.930±0.006	0.874±0.005
SGC	0.997±0.002	0.989±0.004	0.970±0.013	0.899±0.009	0.744±0.008
GATv2	0.995±0.001	0.982±0.003	0.983±0.009	0.952±0.007	0.901±0.008
GPRGNN	0.997±0.001	0.984±0.004	**0.991±0.003**	0.952±0.005	0.903±0.005
H_2GCN	0.996±0.001	0.977±0.002	0.975±0.004	0.941±0.003	0.890±0.006
FAGCN	0.993±0.004	0.981±0.004	0.973±0.014	0.952±0.003	0.905±0.004
BernNet	0.830±0.003	0.839±0.006	0.845±0.008	0.823±0.012	0.812±0.004
JacobiConv	0.213±0.022	0.187±0.020	0.213±0.031	0.235±0.039	0.201±0.038
SLOG(B)	**0.998±0.002**	**0.998±0.003**	**0.991±0.003**	**0.979±0.008**	**0.946±0.011**

h	0.6	0.7	0.8	0.9
MLP	0.813±0.005	0.825±0.006	0.828±0.002	0.810±0.006
GCN	0.517±0.019	0.342±0.008	0.193±0.017	0.208±0.014
ChebNet	0.805±0.006	0.814±0.014	0.782±0.007	0.804±0.006
GraphSAGE	0.671±0.005	0.544±0.019	0.351±0.030	0.219±0.011
GAT	0.678±0.010	0.519±0.012	0.367±0.022	0.244±0.050
APPAP	0.831±0.005	0.832±0.005	0.799±0.007	0.751±0.009
SGC	0.481±0.005	0.331±0.013	0.194±0.018	0.195±0.012
GATv2	0.852±0.016	0.776±0.019	0.654±0.068	0.413±0.046
GPRGNN	0.854±0.002	0.830±0.005	0.811±0.007	0.822±0.008
H_2GCN	0.846±0.002	0.837±0.007	0.819±0.009	0.821±0.007
FAGCN	0.846±0.006	0.828±0.002	0.820±0.007	0.822±0.005
BernNet	0.783±0.011	0.822±0.009	0.810±0.007	0.800±0.013
JacobiConv	0.235±0.051	0.341±0.047	0.378±0.080	0.384±0.101
SLOG(B)	**0.864±0.018**	**0.839±0.019**	**0.830±0.004**	**0.834±0.013**

capture both homophilic and heterophilic information.

Figure 7 shows the learned filters of our method on synthetic datasets with different h. Learned filters on synthetic datasets reveal adaptability to the heterophily ratio: a low-pass filter for small h (better for homophilic graphs) and a high-pass filter for large h (suited for heterophilic graphs). Especially, combining datasets with contrasting h values results in a band-stop filter, a hybrid of high-pass and low-pass filters. This is consistent with our intuition that the filter should be able to capture both homophilic and heterophilic information.



(a) h=0.1 (b) h=0.9 (c) h=0.1&0.9 combined

Figure 7. Learned filters on synthetic datasets with different h. Note that Figure 7(c) is the learned filter when two dataset with h=0.1 and h=0.9 are combined.

B.4. Convergence Study

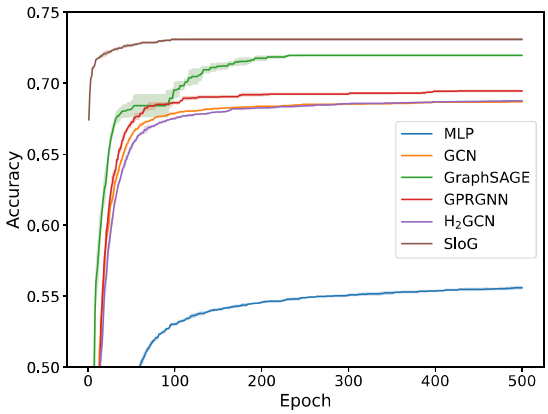


Figure 8. Convergence comparison on Ogbn-arxiv dataset.

To assess the convergence speed of SLOG(B), we performed experiments on the Ogbn-arxiv dataset, comparing it with various baselines. The results, depicted in Figure 8, demonstrate that SLOG(B) converges faster than the other methods. This rapid convergence is due to the method's simplicity, characterized by only two learnable exponents, p and q, along with several weight matrices.

B.5. Optimization Study

SLOG(B) comprises two groups of learnable parameters: those from the filter with real-valued order and those from weight matrices. in Section 4, we utilize Adam to optimize the orders of SLOG filter, p, q. However, the parameters p, q in the filter, being in the exponent position, are challenging to converge using traditional gradient descent methods or other commonly-used gradient-based optimizers. We evaluate two alternative optimization methods, Alternating Minimization in Section B.5.1 and Gaussian Process in Section B.5.2.

B.5.1. ALTERNATING MINIMIZATION

Alternating minimization can partially address this convergence issue, enabling the model to converge more rapidly (Li et al., 2019; Ghosh & Kannan, 2020). Specifically, this method sequentially optimizes one group of variables while fixing the others, then recursively repeats this optimization sequence. We apply alternating minimization to **SLOG(B)**, dividing the parameters into filter-related and weight-related groups to facilitate convergence.

We start by initializing all parameters. Then, we fix p and q in the filter and train all other parameters. After this, we fix all parameters except for p and q and continue the training. This process is repeated until the model converges or reaches the maximum number of iterations. The details are presented in Algorithm 3, where d denotes the tuple of (p, q).

We assess the algorithm using the same settings described in Section 4.2. We set the total number of minimization steps, $2TS = 500$, equal to the maximum iteration number used in Section 4.2. T is maximum iteration number of the outer loop, and S is that of the inner loop. For Squirrel and Squirrel-filt., we use $S = 5$. For Chameleon, we set $S = 10$, and for the remaining datasets, $S = 25$. The results are presented in Table 10.

Table 10. Performance of Gaussian Process optimization.

Datasets	Cora	Citeseer	Squirrel	Chameleon	Squirrel-filt.	Chameleon-filt.	Minesweeper
SLOG(B)	0.865±0.011	0.766±0.026	0.392±0.006	0.581±0.024	0.427±0.013	0.420±0.023	0.822±0.009
w. AM	**0.876±0.014**	**0.771±0.004**	**0.412±0.012**	**0.589±0.010**	**0.428±0.025**	**0.424±0.028**	**0.825±0.004**

The results indicate that alternating minimization enhances the performance of the original model, even with the same number of iterations. This suggests that the filter parameters, though challenging to train, can be effectively optimized with

Algorithm 3 SLOG(B) with Alternating Minimization

Input: Model $\mathcal{M}$; dataset $\mathcal{D}$; parameter space $\mathcal{X}_d$ and $\mathcal{X}_\theta$; loss function $\mathcal{L}$; optimizer $\mathcal{O}$; learning rate η; maximum alternation iteration number T; maximum optimization step number S.
Output: Optimal d^* and θ^*.
Initialize $\theta^{(0,0)}, d^{(0,0)}$ randomly
for $t = 0$ to $T-1$ **do**
  # Step 1: *optimization for* θ
  for $s = 1$ to S **do**
    $\theta^{(t,s)} \leftarrow \arg\min_{\theta \in \mathcal{X}_\theta} \mathcal{L}(\mathcal{M}(d^{(t,0)}, \theta^{(t,s-1)}), \mathcal{D})$ using Optimizer $\mathcal{O}$
  end for
  $\theta^{(t+1,0)} \leftarrow \theta^{(t,s)}$
  # Step 2: *optimization for* d
  for $s = 1$ to S **do**
    $d^{(t,s)} \leftarrow \arg\min_{d \in \mathcal{X}_d} \mathcal{L}(\mathcal{M}(d^{(t,s-1)}, \theta^{(t+1,0)}), \mathcal{D})$ using Optimizer $\mathcal{O}$
  end for
  $d^{(t+1,0)} \leftarrow d^{(t,s)}$
end for
$d^*, \theta^* \leftarrow d^{(T,0)}, \theta^{(T,0)}$
return d^*, θ^*

appropriate methods.

B.5.2. GAUSSIAN PROCESS

SLOG(B) includes parameters p, q (as in Eq. (4)) and other weight matrices. The placement of p, q in the exponent position makes training difficult. Large values of p, q can lead to excessively large gradients, risking gradient explosion, while small values may cause vanishing gradients. Both scenarios present challenges in achieving stable convergence. Therefore, we explore an alternative optimization method to tackle this issue.

We denote the tuple of (p, q) as d and all weight matrices as θ. The optimization problem for SLOG(B) can be formulated as:

$$d^*, \theta^* = \underset{d \in \mathcal{X}_d, \theta \in \mathcal{X}_\theta}{\arg\min}\ \mathcal{L}(\mathcal{M}(d, \theta), \mathcal{D}). \tag{10}$$

We approach this optimization using a bi-level strategy:

$$\begin{aligned} d^* &= \underset{d \in \mathcal{X}_d}{\arg\min} \mathcal{L}(\mathcal{M}(d, \theta^*), \mathcal{D}) \\ s.t.\ \theta^* &= \underset{\theta \in \mathcal{X}_\theta}{\arg\min} \mathcal{L}(\mathcal{M}(d, \theta), \mathcal{D}). \end{aligned} \tag{11}$$

The lower-level optimization (optimization of θ) can be solved using traditional methods such as gradient descent or advanced optimizers like Adam (Kingma & Ba, 2014), AdamW (Loshchilov & Hutter, 2017).

The high-level optimization resembles a hyper-parameter search problem. We employ Bayesian Optimization methods, such as Gaussian Process, for this purpose.

Here, we briefly introduce Gaussian Process. Gaussian Process is a stochastic search process, in which every data point searched at different timestamp obeys a multi-variant normal distribution. A key factor of the process is the covariance functions K, also known as Gaussian kernels, which determine the covariance of the variable collection. The most common covariance function is squared exponential kernel, defined as follows:

$$K(\mathbf{d}, \mathbf{d}') = \exp(-\frac{1}{2\ell^2}\left|\left|\mathbf{d} - \mathbf{d}'\right|\right|^2), \tag{12}$$

where ℓ is the characteristic length scale. Therefore, if we set the mean value of the multi-variant normal distribution as 0, then we can derive the prediction of newly searched data point $\mathbf{d}$ as:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{y}' \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(\mathbf{d}, \mathbf{d}) & K(\mathbf{d}, \mathbf{d}') \\ K(\mathbf{d}', \mathbf{d}) & K(\mathbf{d}', \mathbf{d}') \end{bmatrix}\right), \tag{13}$$

where $y = \mathcal{L}(\mathcal{M}(d, \theta), \mathcal{D})$ indicates the loss (also the performance) given the model parameters. From this, we can derive the following equation (Rasmussen et al., 2006):

$$\mathbf{y}'|\mathbf{y} \sim \mathcal{N}(K(\mathbf{d}', \mathbf{d})K(\mathbf{d}, \mathbf{d})^{-1}\mathbf{y}, K(\mathbf{d}', \mathbf{d}') - K(\mathbf{d}', \mathbf{d})K(\mathbf{d}, \mathbf{d})^{-1}K(\mathbf{d}, \mathbf{d}')). \tag{14}$$

Therefore, the most likely value of y' is given by:

$$\mathbf{y}' = K(\mathbf{d}', \mathbf{d})K(\mathbf{d}, \mathbf{d})^{-1}\mathbf{y}. \tag{15}$$

Thus, we alternate the optimization of model parameters θ and hyper-parameters d. For further details on Bayesian optimization and Gaussian Process, see (Rasmussen et al., 2006; Frazier, 2018). The optimization algorithm is detailed in Algorithm 4.

Algorithm 4 SLOG(B) with Gaussian Process optimization

1: **Input:** Model $\mathcal{M}$; dataset $\mathcal{D}$; parameter space $\mathcal{X}_d$ and $\mathcal{X}_\theta$; loss function $\mathcal{L}$; optimizer $\mathcal{O}$; learning rate η; maximum iteration number T.
2: **Output:** Optimal d^* and θ^*.
3: Initialize $\theta^{(0)}, d^{(0)}$ randomly
4: $\mathcal{Y} \leftarrow \emptyset$
5: **for** $t = 0$ to $T - 1$ **do**
6: **# Lower-level:** *optimization for* θ
7: $\theta^* \leftarrow \arg\min_{\theta \in \mathcal{X}_\theta} \mathcal{L}(\mathcal{M}(d^{(t)}, \theta), \mathcal{D})$ using Optimizer $\mathcal{O}$
8: $y^{(t)} \leftarrow \mathcal{L}(\mathcal{M}(d^{(t)}, \theta^*), \mathcal{D})$
9: $\mathcal{Y} \leftarrow \mathcal{Y} \cup \{d^{(t)}, y^{(t)}\}$
10: **# Upper-level:** *optimization for* d
11: $\mathcal{C} \leftarrow \{d_i \sim \mathcal{X}_d\}_{i=1}^{N}$
12: Calculate y_i according to Eq. (15) for each $d_i \in \mathcal{C}$
13: $idx \leftarrow \arg\max_{i \in |\mathcal{C}|} y_i$
14: $d^{(t+1)} \leftarrow d_{idx} \in \mathcal{C}$
15: **end for**
16: $k \leftarrow \arg\min_i \{y^{(i)} | (d^{(i)}, y^{(i)}) \in \mathcal{Y}\}$
17: **return** $d^{(k)}, \theta^{(k)}$

We evaluate the algorithm using the same settings described in Section 4.2, setting the maximum iteration number $T = 20$. The results are shown in Table 11.

Table 11. Performance of Gaussian Process optimization.

Datasets	Cora	Citeseer	Squirrel	Chameleon	Squirrel-filt.	Chameleon-filt.	Minesweeper
SLOG(B)	**0.865±0.011**	0.766±0.026	0.392±0.006	0.581±0.024	**0.427±0.013**	0.420±0.023	0.822±0.009
w. GP	0.853±0.022	**0.778±0.010**	**0.417±0.005**	**0.586±0.021**	0.402±0.017	**0.426±0.044**	**0.826±0.010**

This evaluation demonstrates that Gaussian Process optimization significantly enhances the performance of SLOG(B). As shown in Table 11, the model optimized with Gaussian Process outperforms that using the original settings across most of the datasets. Note that Gaussian Process will not increase the computational complexity of SLOG, since the total time complexity of Gaussian Process is $O(n^3 + n^2 m)$, where n is the number of elements in training set (i.e., $|\mathcal{Y}|$ in Algorithm 4) and m is the amount of samples to be predicted (i.e., $|\mathcal{C}|$ in Algorithm 4). However, due to the rather limited training set scale ($n \sim 10$) and prediction set scale ($m \sim 100$ in the experiments), the complexity of Gaussian Process (Line 10-14, Algorithm 4) is negligible in comparison to that of model training (Line 6-9, Algorithm 4).

C. Experimental Details

C.1. Dataset Statistics

For datasets like Cora, Citeseer, Chameleon, Squirrel, DBLP, Coauthor-CS, and Coauthor-Physics, we collect them from Pytorch Geometric library (Fey & Lenssen, 2019). For Chameleon-filtered, Squirrel-filtered, Minesweeper, Tolokers,

Amazon-ratings, Questions, we collect the raw data released from (Platonov et al., 2023)[8]. For Ogbn-arxiv, we collect it from Open Graph Benchmark (Hu et al., 2020). The statistics of these datasets are shown in Table 12.

Table 12. The statistics of the datasets.

Datasets	Nodes	Edges	Features	Classes	Heterophily
Cora	2,708	10,556	1,433	7	0.190
Citeseer	3,327	9,104	3,703	6	0.264
Chameleon	2,277	62,792	2,325	5	0.765
Squirrel	5,201	396,846	2,089	5	0.776
DBLP	17,716	105,734	1,639	4	0.172
Coauthor-CS	18,333	163,788	6,805	15	0.192
Coauthor-Physics	34,493	495,924	8,415	5	0.069
Chameleon-filtered	890	13,584	2,325	5	0.764
Squirrel-filtered	2,223	65,718	2,089	5	0.793
Minesweeper	10,000	39,402	7	2	0.317
Tolokers	11,758	519,000	10	2	0.405
Amazon-ratings	24,492	93,050	300	5	0.620
Questions	48,921	153,540	301	2	0.160
Flickr	89,250	899,756	500	7	0.681
Ogbn-arxiv	169,343	1,166,243	128	40	0.322
Reddit	232,965	114,615,892	602	41	0.244

C.2. Implementation Details

For small-scale datasets, we set the hidden dimension of all methods to 128, the learning rate to 0.01. For large-scale datasets, we set the hidden dimension as 512, and fine-tune the learning rate in the range of $[10^{-2}, 10^{-3}, 10^{-4}]$. We use Adam as the optimizer. For all datasets, we set the weight decay to 0.0005. For all datasets except Reddit, we limit the training epochs to 500. For Reddit, we set maximum training epochs as 30. The layer number L for SLOG(N) is set to 3. We determine the size of the subgraphs partitioned by SLOG(L) to be approximately 512, and the parameter β is set to 0.5. For hyper-parameters of baselines, we use the default configurations if available, otherwise, we use the same hyper-parameter space for SLOG. We run all the experiments on a Tesla-V100 GPU with 32G Memory.

C.3. Graph Partition: METIS Algorithm

In SLOG(L), METIS (Karypis & Kumar, 1998) is used for graph partition. Here, we provide more information about graph partition and METIS.

A k-way graph partition is defined as minimizing the edge-cut to conduct the following node partition (Karypis & Kumar, 1996): given a graph $\mathcal{G} = \{(\mathcal{V}, \mathcal{E})\}$ with $|V| = n$, partition the node set $\mathcal{V}$ into k subsets, $\mathcal{V}_1, \cdots, \mathcal{V}_k$, with the minimal number of edges whose incident verticles belong to different subset, subject to the constraint that $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ for $i \neq j$, $\cup_{i=1}^{k} \mathcal{V}_i = \mathcal{V}$ and $|\mathcal{V}_i| = n/k$ for $i = 1, \cdots, k$. To simplify the problem, we introduce bisection graph partition, which is a special case of k-way graph partition with $k = 2$. The METIS algorithm solve the problem with the following steps (Karypis & Kumar, 1998):

Step 1: Coarsening Phase. The original graph $\mathcal{G}_0$ will produce a sequence of graph $\mathcal{G}_1, \cdots, \mathcal{G}_m$, such that $|\mathcal{V}_0| > |\mathcal{V}_1| > \cdots > |\mathcal{V}_m|$. A set of nodes in $\mathcal{G}_i$ can be coarsened to a single *multinode* in $\mathcal{G}_{i+1}$. The coarsening strategy is related to *matching*, which is defined as a set of edges in which no two edges are incident on a same node. There are lots of matching algorithm, and METIS uses *heavy-edge matching* with a slight modification. One can refer to (Karypis & Kumar, 1998) for more details. The coarsening phase is repeated until the graph size is small enough.

Step 2: Partition Phase. We compute the bi-parition for $\mathcal{G}_m$, guaranteeing that each part after partition contains approximately half of the nodes in the original graph $\mathcal{G}_0$. Since the number of nodes in $\mathcal{G}_m$ is small, the time consumed in this step is rather small.

Step 3: Uncoarsening Phase. The partition of $\mathcal{G}_m$ is projected back to $\mathcal{G}_0$ by applying the same partition to the coarser graphs $\mathcal{G}_{m-1}, \cdots, \mathcal{G}_1$. The uncoarsening phase is repeated until the original graph $\mathcal{G}_0$ is reached.

[8]Github link: https://github.com/yandex-research/heterophilous-graphs/tree/main/data.

We implement graph partition with the help of PyMETIS[9] library, which is a python wrapper for METIS.

[9]Github link: https://github.com/inducer/pymetis.