Doctoral Dissertation

Compositionality-Aware Graph Representation Learning

Takeshi D. Itoh
Program of Information Science and Engineering
Graduate School of Science and Technology
Nara Institute of Science and Technology

Supervisor: Prof. Kazushi Ikeda
Mathematical Informatics Lab. (Division of Information Science)

Submitted on March 15, 2022

A Doctoral Dissertation
submitted to Graduate School of Science and Technology,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of Engineering

Takeshi D. Itoh

Thesis Committee:

Supervisor Kazushi Ikeda
(Professor, Division of Information Science)

Kenichi Matsumoto
(Professor, Division of Information Science)

Junichiro Yoshimoto
(Visiting Professor, Division of Information Science)

Takatomi Kubo
(Associate Professor, Division of Information Science)

Makoto Fukushima
(Assistant Professor, Division of Information Science)

Chie Hieida
(Assistant Professor, Division of Information Science)

Hayaru Shouno
(Professor, University of Electro-Communications)

Compositionality-Aware Graph Representation Learning*

Takeshi D. Itoh

Abstract

Graph neural networks (GNNs) have been widely used to learn vector representations of graph-structured data and have achieved better task performance than conventional graph machine learning methods. The foundation of GNNs is the message passing procedure, which propagates information from a node to its neighbors. Therefore, message passing GNNs can exploit the stationarity and locality of graphs.

However, existing GNN methods have limitations in capturing and exploiting the compositionality of graphs while learning graph representations. Because message passing is executed one step at a time for each layer, the range of the information propagation among nodes is small in the lower layers, and it expands toward the higher layers. Therefore, a GNN model must be deep enough to capture the global structural information in a graph. By contrast, it is known that deep GNN models suffer from performance degradation because they lose the local information of nodes, which would be essential for good model performance, because of the large number of message passing steps. In other words, there is a trade-off between using deep GNNs to capture global graph information and using shallow ones to focus on local information.

In this dissertation, we propose multi-level attention pooling (MLAP) for graph representation learning (GRL), which can adapt to both local and global structural information in a graph. It has an attention pooling layer for each message passing step and computes the final graph representation by unifying layer-wise graph representations. The MLAP architecture allows models to utilize the

*Doctoral Dissertation, Graduate School of Science and Technology, Nara Institute of Science and Technology, March 15, 2022.

structural information in graphs with multiple levels of localities because it preserves layer-wise information before losing them. Results of our experiments show that the MLAP architecture improves performance compared with the baseline architectures, both in graph classification tasks and graph-to-sequence (graph2seq) tasks. In addition, analyses of the layer-wise graph representations indicate that aggregating information from multiple levels of localities is indeed beneficial for learning discriminative graph representations. Although the exploitation of compositionality in neural network studies is just in its infancy, we believe that analyzing compositionality is key to building high-performance and interpretable machine learning models, both in GRL and other machine learning domains.

Keywords:

Graph representation learning (GRL), graph neural network (GNN), compositionality, attention

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graph-structured data can be found in many fields. A wide variety of natural and artificial objects can be expressed with graphs, such as molecular structural formulae, biochemical reaction pathways, brain connection networks, social networks, and abstract syntax trees of computer programs. Similar to other forms of natural data (e.g., images), graphs have three frequently observed properties: stationarity, locality, and compositionality [1, 2]. Stationarity implies that one can observe similar statistical properties of the signal across the entire graph. Locality indicates that we can extract certain information by looking at only a part of a graph. Compositionality implies that the global information on a graph is composed of information from multiple local (smaller) subgraphs. In this dissertation, we explore a graph representation learning (GRL) technique using graph neural networks (GNNs) that makes more effective use of the compositional property of graphs.

As graphs are found ubiquitously around the world, machine learning techniques on graphs have been intensively researched for decades. Owing to rich information underlying the structure, graph machine learning techniques have shown remarkable performances in various tasks. For example, the PageRank algorithm [3] measures the importance of each node in a directed graph based on the number of inbound edges to the node. Shervashidze et al. [4] used a graph kernel method [5] to predict chemical molecules' toxicity as a graph classification task. Despite these promising applications, classical machine learning techniques on graphs require difficult and costly processes for manually designing

graph features or kernels.

To solve the problem, the GRL approach has attracted the attention from the research field [6]. A GRL model learns a mapping from a node or a graph to a vector representation. The learned representation provided by the mapping can then be used as an input feature for task-specific models (e.g., classifiers or regressors) and thus a model becomes free from the problem of inflexible hand-crafted features. The mapping is trained so that the geometric relationships among embedded representations reflect the similarity of structural information in graphs; that is, nodes with similar local structures have similar representations [7, 8]. However, the early GRL techniques learned a unique vector for each node without sharing parameters among nodes, leading to high computational costs and the risk of overfitting. Furthermore, as these techniques learn a specific representation for each node, learned models cannot be applied for prediction on novel graphs or nodes that do not appear in the training phase [9, Section 3.4].

Since around 2015, GNNs have rapidly gained interest as a new framework for GRL (we refer readers to Zhang et al. [10] and Wu et al. [11] for review papers; see Section 2.2 for related works). Unlike aforementioned non-GNN GRL techniques, which learn node-specific representations, GNNs learn *how* to compute the node representation from the structural information around a node. Hence, GNNs do not suffer from the problem faced by earlier GRL methods, where the computation cost and the number of parameters increased linearly to the number of nodes. Furthermore, the learned models generalize to graphs or nodes that were unknown during training.

The foundation of GNNs is the message passing procedure propagating the information in a node to its neighbor nodes, each of which is directly connected to the source node by an edge (Figure 1.1a; see Section 2.3.1 for detail). The message passing procedure is designed to function well when the signals on graphs exhibit stationarity and locality. That is, stationarity allows same message passing parameters to be reused across the entire graph, resulting in reduction of the number of parameters to be trained. Also, collecting information from neighbor nodes for updating the node representation is possible because of the information locality.

However, existing message passing GNNs have suffered to capture the third

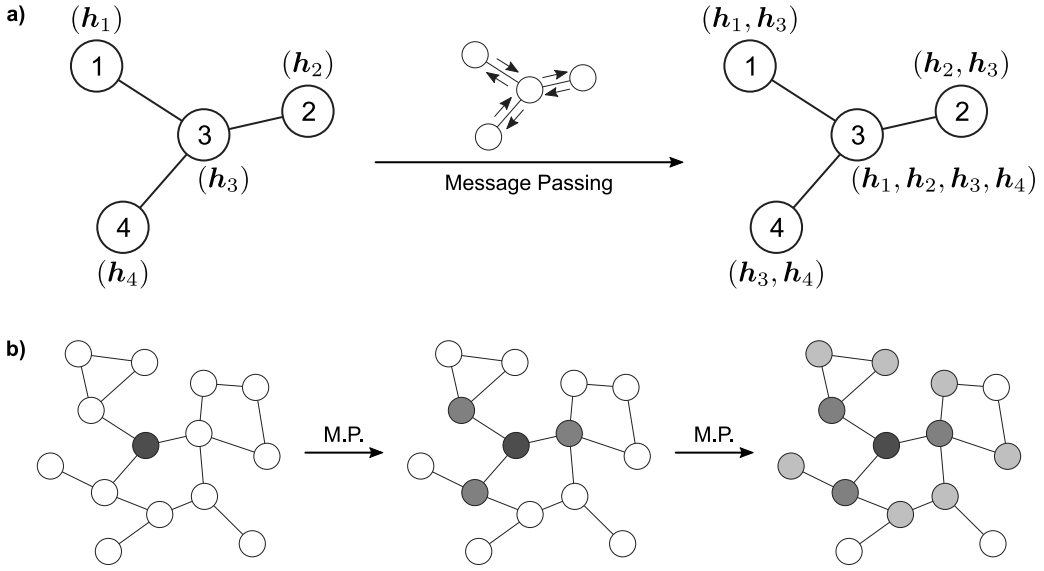

Figure 1.1: a) Schematic illustration of the message passing procedure. The i-th node has its original node information, $\boldsymbol{h}_i$ $(i = 1, \ldots, 4)$, at the beginning (left). The message passing procedure propagates node information between each pair of connected nodes (center). As a result, each node has its own information and neighbors' information after the message passing (right). **b)** The scope of the information propagation expands along the message passing process. The black node in the middle of a graph has only its original node information at the beginning (left). This node obtains information in broader subgraphs through message passing, i.e., dark gray nodes after one message passing step (center) and light gray nodes after two message passing steps (right). *M.P.*: message passing.

property: the compositionality. As the procedure proceeds one step per layer, the range of the information propagation among nodes is small in the lower layers, and it expands toward the higher layers—i.e., the node representations in the higher layers collect information from broader subgraphs (Figure 1.1b). In other words, each layer has hierarchically organized level of information locality. Existing GNN architectures could not effectively utilize such hierarchy or compositionality of the information in graphs because they commonly used the graph representation after a fixed number of message-passing steps, ignoring the information in lower layers. It is difficult to determine an appropriate depth (number of layers) for a GNN model that allows the model to capture both local and composited global information. Hence, it may be beneficial to develop a GNN model capable of using information at multiple levels in the compositionality.

Overall, our fundamental questions in this dissertation are as follows:

- *Does a compositionality-aware GNN architecture improve the performances in graph machine learning tasks?*
- *Is it effective for GNNs to explicitly utilize the graph information from various levels of localities to learn more discriminative graph representations?*

We answer *"yes"* to both of the questions by proposing multi-level attention pooling (MLAP) architecture.

1.1 Contributions

This dissertation makes the following contributions.

- It presents a novel compositionality-aware GNN architecture called multi-level attention pooling (MLAP).
- It demonstrates that the MLAP architecture is effective both in real-world graph classification and graph to sequence (graph2seq) tasks.
- It provides detailed analyses of the layer-wise graph representations in MLAP models and demonstrates the effectiveness of utilizing information at multiple levels of localities.

1.2 Outline

The rest of the dissertation consists of the following chapters: Chapter 2 introduces the MLAP architecture and apply it to multiple synthetic and real-world graph classification problems. It shows performance improvement by MLAP from baseline architectures, as well as provides detailed analyses of the layer-wise graph representations. Chapter 3 extends the MLAP architecture for graph2seq tasks. The application of the technique to a source code summarization task is discussed and it is demonstrated that our proposed model achieves the state-of-the-art (SoTA) performance. Finally, Chapter 4 reviews the findings and the future direction of the study, and then provides concluding statements.

Chapter 2

Multi-Level Attention Pooling for Graph Classification

In this chapter, we introduce the multi-layer attention pooling (MLAP) architecture for GNN, which utilizes the compositionality in graph information. We demonstrate that aggregating graph representations from multiple message-passing layers improves the model performance in graph classification tasks.

2.1 Introduction

Initial stages of the machine learning research had focused on data in fixed form—e.g., scalars, vectors, or matrices. However, there exist many types of data that can take arbitrary and unfixed form. Graphs and networks are the most important examples of such arbitrary-shaped data. Even though we need machine learning techniques specifically designed to deal with graphs having unfixed form, they would benefit from rich information in their structure in addition to the signals on graphs.

Discriminating graphs into multiple classes is one of the most fundamental problems in graph machine learning. Judging toxicity of a molecule from the chemical formula, determining the type of an object from point cloud, or finding attributes of a person from a subgraph of social networks are all good examples of real-world graph classification tasks. As we discussed in Chapter 1, message passing GNN has been designed to perform well when graphs have stationarity

and locality, and it certainly has achieved a significant success in this field [10, 11].

However, there exists a well-known problem with GNNs where the model performance degrades as the number of layers increases. This is because deep GNN models lose the nodes' local information, which would be essential for good model performance, during many message-passing steps. This phenomenon is known as *oversmoothing* [12]. Because graphs are compositional, a GNN model needs to be capable of capturing both local structural information and global structural information. Capturing global structural information requires a GNN model to be deep (i.e., having many message passing steps), but oversmoothing prohibits a model from being deep. In other words, there has been a tradeoff between making deep GNNs to capture global graph information and using shallow ones to focus on local information.

In this chapter, we present a compositionality-aware GNN technique to learn more discriminative graph representation by using multiple graph representations in different localities. Previous studies typically computed the graph representation by a graph pooling layer that collects node representations after the last message passing layer. Therefore, deeper models cannot utilize nodes' local information in computing the graph representation because local information is lost through many message-passing steps due to oversmoothing. Although many previous methods have addressed the oversmoothing problem (see Section 2.2.2), our approach—using information with multiple levels of localities to compute graph representations—does not aim to directly solve the oversmoothing problem itself, but rather focuses on improving the discriminability of learned representations by explicitly utilizing the compositionality.

To this end, we propose the multi-level attention pooling (MLAP) architecture. In summary, the MLAP architecture introduces an attention pooling layer [13] for each message passing step to compute layer-wise graph representations. Then, it aggregates them to compute the final graph representation, inspired by the jumping knowledge network [14]. As a result, the MLAP architecture can focus on different nodes (or different subgraphs) in each layer with different levels of information localities, which leads to better modeling of both local structural information and global structural information. In other words, introducing layer-wise attention pooling prior to aggregating layer-wise

representation would improve the graph-level classification performance. Our experiments show performance improvements in GNN models with the MLAP architecture. Furthermore, analyses on the layer-wise graph representations suggest that MLAP has the potential to learn graph representations with improved class discriminability by aggregating information with multiple levels of localities.

The rest of this chapter is organized as follows: Section 2.2 summarizes related studies on graph neural networks, Section 2.3 introduces the MLAP framework, Section 2.4 describes the experimental setups, Section 2.5 demonstrates the results, and Section 2.6 discusses the findings and concludes this chapter by highlighting the contributions of the study.

2.2 Related Works

Gori et al. [15] and Scarselli et al. [16] first introduced the idea of GNNs, and Bruna et al. [17] and Defferrard et al. [1] elaborated the formulation in the graph Fourier domain using spectral filtering. Based on these early works, Kipf and Welling [18] proposed the graph convolution network (GCN), which made a foundation of today's various GNN models [19–23]. Gilmer et al. [24] summarized these methods as a framework named neural message passing, which computes node representations iteratively by collecting neighbor nodes' representation using differentiable functions (Figure 1.1a, see Section 2.3.1 for mathematical formulation).

In this study, we focus on methods for computing the graph representation from node-wise representations in GNN models. We first summarize the studies on graph pooling methods and then review the recent trends in *deep* GNN studies. Finally, we summarize previous studies that aggregate layer-wise representation to compute the final node or graph representation and elaborate the idea behind our proposed method.

2.2.1 Graph Pooling Methods

Techniques for learning *graph* representations are usually based on techniques for learning *node* representations. A graph-level model first computes the representation for each node in a graph and then collects the node-wise representations into

a single graph representation vector. This collection procedure is called a pooling operation. Although there are various pooling methods, they can be divided into two categories: the *global* pooling approach and the *hierarchical* pooling approach.

The *global* pooling approach collects all node representations in a single computation. The simplest example of the global pooling method is sum pooling, which merely computes the sum of all node representations. Duvenaud et al. [19] introduced sum pooling to learn embedded representations of molecules from a graph where each node represents an atom. Similarly, we can compute an average or take the maximum elements as a pooling method. Li et al. [13] introduced attention pooling, which computes a weighted sum of node representations based on a softmax attention mechanism [25]. Vinyals et al. [26] proposed set2set by extending the sequence to sequence (seq2seq) approach for a set without ordering. Zhang et al. [27] introduced the SortPooling, which sorts the node representations according to their topological features and applies one-dimensional convolution. These global pooling methods are simple and computationally lightweight, but they cannot use the structural information of graphs in the pooling operation.

By contrast, *hierarchical* pooling methods segment the entire graph into a set of subgraphs hierarchically and compute the representations of subgraphs iteratively. Bruna et al. [17] introduced the idea of hierarchical pooling, or graph coarsening, based on hierarchical agglomerative clustering. Although some previous studies such as Defferrard et al. [1] also applied similar approaches, clustering-based hierarchical pooling requires the clustering algorithm to be deterministic—that is, the hierarchy of subgraphs is fixed throughout the training. To overcome this limitation, Ying et al. [28] proposed DiffPool, which learns the subgraph hierarchy itself along with the message passing functions. They proposed to use a neural network to estimate which subgraph a node should belong to in the next layer. Gao and Ji [29] extended U-Net [30] for graph structure to propose graph U-Nets. Original U-Net introduced down-sampling and up-sampling procedures for semantic image segmentation tasks. Based on the U-Net, graph U-Nets are composed of a gPool network to shrink the graph size hierarchically and a gUnpool network to restore the original graph structure. Furthermore, Lee et al. [31] employed a self-attention mechanism to define a hierarchy of subgraph structures.

Hierarchical pooling can adapt to multiple localities of graph substructures during step-wise shrinkage of graphs. Although these hierarchical pooling methods effectively utilize the compositionality in graphs, they are often computationally heavy because, as discussed in Cangea et al. [32], they have to learn the dense *assignment matrix* for each layer, relating a node in a layer to a node in the shrunk graph in the next layer. Thus, they require longer computational time and consume more memory.

2.2.2 Oversmoothing in Deep Graph Neural Networks

Kipf and Welling [18] first reported that deep GNN models with many message passing layers performed worse than shallower models. Li et al. [12] investigated this phenomenon and found that deep GNN models converged to an equilibrium point where connected nodes have similar representations. As the nodes with similar representations are indistinguishable from each other, such convergence degrades the performance in node-level prediction tasks. This problem is called *oversmoothing*. In graph-level prediction tasks, oversmoothing occurs independently for each graph. Oversmoothing in each graph damages GNN models' expressivity and results in performance degradation [33, 34].

Studies addressing the oversmoothing problem mainly fall into three categories: modifying the message passing formulation, adding residual connections, and normalization. Anyhow, the objective of these studies is to retain discriminative representations even after many steps of message passing.

Studies modifying the message passing formulation aim to retain high-frequency components in graph signals during message passing steps, whereas message passing among nodes generally acts as a low-pass filter for the signals. Min et al. [35] proposed scattering GCN, which adds a circuit for band-pass filtering of node representations. DropEdge [36] randomly removes some edges from the input graph, alleviating the low-pass filtering effect of the graph convolution. In addition, although not explicitly stated, the graph attention network (GAT) [22] is known to mitigate the oversmoothing problem because it can focus on specific nodes during message passing.

Adding residual connections is a more straightforward way to retain node-local representation up to deeper layers. Residual connections, or ResNet architecture,

were first used in convolutional neural networks (CNNs) for computer vision tasks, where they achieved a SoTA performance [37]. Kipf and Welling [18] applied the residual connections in the graph convolutional network and reported that residual connections mitigated the performance degradation in deep GNN models. Later, Li et al. [38], Zhang and Meng [39], and Chen et al. [40] applied similar residual architectures on GNNs and showed performance improvement.

Normalization in deep learning gained attention owing to the success of previous studies such as BatchNorm [41] and LayerNorm [42]. Although these general normalization techniques are also applicable and effective in GNNs, graph-specific normalization methods have been recently proposed. PairNorm [43], NodeNorm [44], GraphNorm [45], and differentiable group normalization [46] are representative examples of graph-specific normalization methods.

These studies succeeded in overcoming the oversmoothing problem and allowing deep GNN models to retain discriminative representations. However, directly using local representations in computing the final graph representation would lead to more performance improvement, owing to the compositionality in graph information.

2.2.3 Aggregating Layer-Wise Representations in GNN

The studies summarized in the previous subsection directly addressed the oversmoothing problem. They sought techniques to retain discriminative representations even after multiple steps of message passing. Instead, we search for a technique to learn more discriminative representation by aggregating multiple representations in different localities.

Jumping knowledge (JK) network [14] proposed to compute the final node representation by aggregating intermediate layer-wise node representations. Thus, JK can adapt the locality of the subgraph from which a node gathers information. After JK was proposed, many studies adopted JK-like aggregation of layer-wise representation to improve the learned representation. Wang et al. [47] adopted JK in recommendation tasks on knowledge graphs. Cangea et al. [32] adopted a JK-like aggregation of layer-wise pooled representation on gPool [29] network to learn graph-level tasks. A similar combination of hierarchical graph pooling and JK-like aggregation was also proposed by Ranjan et al. [48]. Dehmamy et al. [49]

proposed aggregating layer-wise representation from a modified GCN architecture and showed performance improvement.

Our proposed MLAP technique is motivated by the same idea of these studies that GNNs should be capable of aggregating information in multiple levels of localities. Here, we utilize an intuition on graph-level prediction tasks: a model should focus on different nodes as the message passing proceeds through layers and the locality of information extends. That is, the importance of a node in global graph pooling would differ depending on the locality of the information. Therefore, in this study, we propose a method that uses an attention-based global pooling in each layer and aggregates all layer-wise graph representations to compute the final graph representation.

2.3 Methods

We propose the MLAP architecture, which aggregates graph representation from multiple levels of localities. In this section, we first summarize the fundamentals of GNNs, particularly the message passing procedure, and then introduce the MLAP architecture.

2.3.1 Preliminaries: Graph Neural Networks

Let $G = (\mathcal{N}, \mathcal{E})$ be a graph, where $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ is a set of edges. $n \in \mathcal{N}$ denotes a node and $e_{n_{\mathrm{src}},n_{\mathrm{dst}}} \in \mathcal{E}$ denotes a directed edge from a source node n_{src} to a destination node n_{dst}. A graph may have either node features or edge features, or both. If a graph has node features, each node n has a node feature vector $\boldsymbol{p}_n$. Similarly, if a graph has edge features, each edge $e_{n_{\mathrm{src}},n_{\mathrm{dst}}}$ has an edge feature vector $\boldsymbol{q}_{n_{\mathrm{src}},n_{\mathrm{dst}}}$.

There are three types of tasks commonly studied for GNNs: graph-level prediction, node-level prediction, and edge-level prediction. In this study, we focus on the graph-level prediction tasks. For a set of graph $\mathcal{G} = \left\{G_1, \ldots, G_{|\mathcal{G}|}\right\}$ and their labels $\mathcal{Y} = \left\{y_1, \ldots, y_{|\mathcal{G}|}\right\}$, we want to learn a graph representation vector $\boldsymbol{h}_G$ used for predicting the graph label $\hat{y}_G = g\left(\boldsymbol{h}_G\right)$, where g denotes a predictor function.

Suppose we have a GNN with L layers. Each layer in a GNN propagates the node representation $\boldsymbol{h}_n$ along the edges (*message passing*). Let $\boldsymbol{h}_n^{(l)} \in \mathbb{R}^d$ be the representation of n after the message passing by the l-th layer, where d is the dimension of the vector representations. In general, the propagation by the l-th layer first computes the *message* $\boldsymbol{m}_n^{(l)}$ for each node n from its neighbor nodes $\mathrm{NBR}\,(n)$, as in

$$\boldsymbol{m}_n^{(l)} = f_{\mathrm{col}}^{(l)}\left(\left\{ f_{\mathrm{msg}}^{(l)}\left(\boldsymbol{h}_{n'}^{(l-1)}, \boldsymbol{q}_{n',n}\right) \;\middle|\; n' \in \mathrm{NBR}\,(n)\right\}\right), \tag{2.1}$$

where $f_{\mathrm{msg}}^{(l)}$ is a message function to compute the message for each neighbor node from the neighbor representation and the feature of the connecting edge, and $f_{\mathrm{col}}^{(l)}$ is a function to collect the neighbor node-wise messages. Then, the layer *updates* the node representation $\boldsymbol{h}_n^{(l)}$ as

$$\boldsymbol{h}_n^{(l)} = f_{\mathrm{upd}}^{(l)}\left(\boldsymbol{m}_n^{(l)}, \boldsymbol{h}_n^{(l-1)}\right), \tag{2.2}$$

where $f_{\mathrm{upd}}^{(l)}$ is an update function.

After L steps of message passing, a graph pooling layer computes a graph representation vector $\boldsymbol{h}_G$ from the final node representations $\boldsymbol{h}_n^{(L)}$ for each $n \in \mathcal{N}$, as in

$$\boldsymbol{h}_G = \mathrm{Pool}\left(\left\{\boldsymbol{h}_n^{(L)} \;\middle|\; n \in \mathcal{N}\right\}\right). \tag{2.3}$$

2.3.2 Multi-Level Attention Pooling

Graph-level prediction tasks require the models to use both local information in nodes and global information as the entire graphs to achieve good performances. However, typical GNN implementations first execute the message passing among nodes for a certain number of steps L *and then* pool the node representations into a graph representation, as shown in Eq. (2.3) (Figure 2.1a). This formulation damages GNN models' expressivity because it can only use the information in a fixed locality to compute the graph representation.

To fix this problem, we introduce a novel GNN architecture named multi-level attention pooling (MLAP; Figure 2.1c). In the MLAP architecture, each message passing layer has a dedicated pooling layer to compute layer-wise graph representations, as in

$$\boldsymbol{h}_G^{(l)} = \mathrm{Pool}^{(l)}\left(\left\{\boldsymbol{h}_n^{(l)} \;\middle|\; n \in \mathcal{N}\right\}\right) \quad \forall\, l \in \{1, \ldots, L\}. \tag{2.4}$$

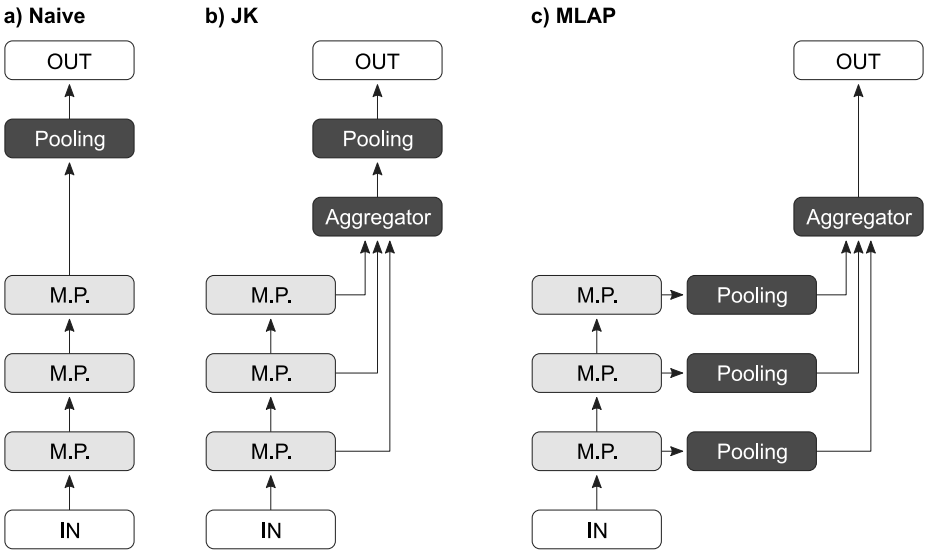

Figure 2.1: a) Naive GNN architecture. A pooling layer computes the graph representation from the node representations after the last message passing. **b)** Jumping knowledge (JK) network architecture. The aggregator collects the layer-wise *node* representation, and then a pooling layer computes the graph representation from the aggregated node representation. **c)** Proposed multi-level attention pooling (MLAP) architecture. There is a dedicated pooling layer for each message passing layer to compute layer-wise *graph* representation. The aggregator computes the final graph representation from the layer-wise graph representations. *M.P.*: message passing.

Here, we used the attention pooling [13] as the pooling layer. Thus,

$$\boldsymbol{h}_G^{(l)} = \sum_{n \in \mathcal{N}} \mathrm{softmax}\left(f_{\mathrm{gate}}^{(l)}(\boldsymbol{h}_n^{(l)})\right) \boldsymbol{h}_n^{(l)} \tag{2.5}$$

$$= \sum_{n \in \mathcal{N}} \frac{\exp\left(f_{\mathrm{gate}}^{(l)}(\boldsymbol{h}_n^{(l)})\right)}{\sum_{n' \in \mathcal{N}} \exp\left(f_{\mathrm{gate}}^{(l)}(\boldsymbol{h}_{n'}^{(l)})\right)} \boldsymbol{h}_n^{(l)}, \tag{2.6}$$

where $f_{\mathrm{gate}}^{(l)}$ is a function used to compute the attention score, for which a two-layer neural network was used. By introducing such layer-wise attention pooling operations, MLAP can focus on different nodes at different information localities.

Then, an aggregation function computes the final graph representation by unifying the layer-wise representations as follows:

$$\boldsymbol{h}_G = f_{\mathrm{agg}}\left(\left\{\boldsymbol{h}_G^{(l)} \;\middle|\; l \in \{1, \ldots, L\}\right\}\right), \tag{2.7}$$

where f_{agg} is an aggregation function. We can use an arbitrary function for f_{agg}. In this study, we tested two types of aggregation functions: *Sum* and *Weighted*.

MLAP-Sum

One of the simplest ways to aggregate the layer-wise graph representations is to calculate their sum, as in

$$\boldsymbol{h}_G = \sum_{l=1}^{L} \boldsymbol{h}_G^{(l)}. \tag{2.8}$$

This formulation expresses an assumption that the representation in each layer is equally important in computing the final graph representation.

MLAP-Weighted

Each layer-wise representation may have varied importance depending on the layer index. If this is the case, computing a weighted sum would be adequate to learn the importance of layers, as in

$$\boldsymbol{h}_G = \sum_{l=1}^{L} w^{(l)} \boldsymbol{h}_G^{(l)}, \tag{2.9}$$

where $\left\{ w^{(l)} \;\middle|\; l \in \{1, \ldots, L\} \right\}$ is a trainable weight vector.

2.4 Experiments

Our experimental evaluation aims to answer these research questions:

RQ1 Does the MLAP architecture improve the GNN performance in graph classification tasks?

RQ2 Does aggregating representations from multiple layers aid in learning discriminative graph representation?

We conducted experiments using four graph classification datasets: a synthetic dataset and three real-world datasets.

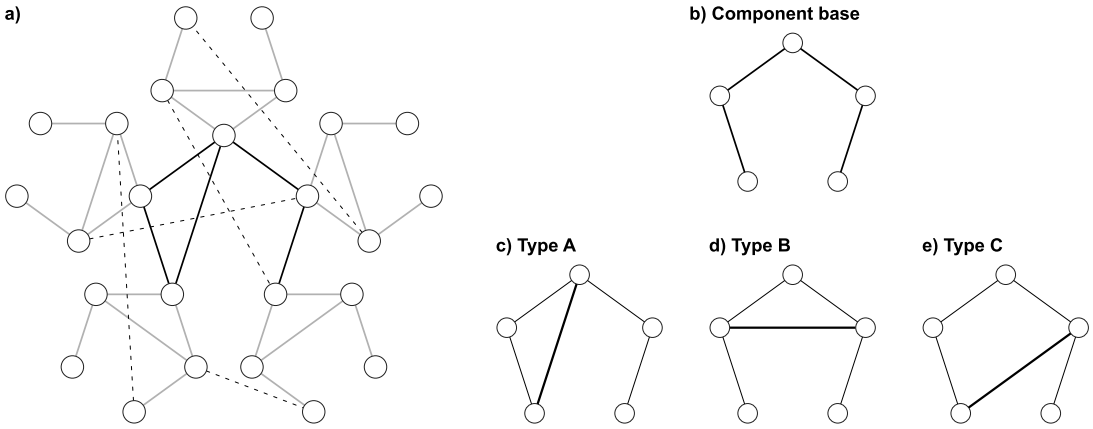

Figure 2.2: a) A graph in the synthetic dataset. It consists of the *center* component (black edges), five *peripheral* components (gray edges), and five additional random edges (dotted edges). The class of this graph is determined by the combination of the types of the center component (type A) and the peripheral components (type B). **b)** Basic structure of a component. **c–e)** Three types of components.

2.4.1 Synthetic Dataset

We created a synthetic dataset to show the effectiveness of MLAP using multi-level representation in a graph-level classification task. We designed the dataset in such a way that its graph features are represented in both local and global graph structures.

A graph in the dataset consists of six 5-node components: one *center* component surrounded by five *peripheral* components, each of which shares a node with the center component (Figure 2.2a). The basic structure of a component is five sequentially connected nodes (Figure 2.2b) with an extra edge. Based on how the extra edge is appended, there are three types of components (Figure 2.2c–e). The class of a graph is determined by the combination of the type of the center component and the type of the peripheral components. We note that the five peripheral components share the same type. Therefore, there are $3 \times 3 = 9$ classes of graphs. With this design, accurately classifying the graphs in this dataset requires a model to learn both the local substructures in a graph and the global structure as an entire graph (i.e., the combination of the types of local substructures). Neither nodes nor edges in the graphs have features.

We generated 1,000 unique graphs for each class by randomly appending five

edges between arbitrarily selected pairs of nodes. Hence, there were 9,000 instances in the dataset in total, and we applied a random 8:1:1 split to provide training, validation, and test sets. Model performance was evaluated by the error rate ($1 - \text{Accuracy}$).

2.4.2 Real-World Benchmark Datasets

We used two datasets from the open graph benchmark (OGB) [50] and the MCF-7 dataset from the TU graph dataset collection [51].

ogbg-molhiv

ogbg-molhiv is a dataset for a molecular property prediction task, originally introduced in Wu et al. [52]. Each graph in the dataset represents a molecule. Each node in a graph represents an atom and has a 9-dimensional discrete-valued feature containing the atomic number and other atomic properties. Each edge represents a chemical bond between two atoms and has a 3-dimensional discrete-valued feature containing the bond type and other properties. This dataset has a relatively small sample size (41,127 graphs in total), with 25.5 nodes and 27.5 edges per graph on average. The task is a binary classification to identify whether a molecule inhibits the human immunodeficiency virus (HIV) from replication. Model performance is evaluated by the area under the curve value of the radar operator characteristics curve (ROC-AUC). We followed the standard dataset splitting procedure provided by the OGB.

ogbg-ppa

The ogbg-ppa dataset contains a set of subgraphs extracted from protein-protein association networks of species in 37 taxonomic groups, originally introduced in Szklarczyk et al. [53]. Each node in a graph represents a protein without node features. Each edge represents an association between two proteins and has a 7-dimensional real-valued feature describing the biological meanings of the association. This dataset has a medium sample size (158,100 graphs in total), with 243.4 nodes and 2266.1 edges per graph on average. The task is a classification to identify from which taxonomic group among 37 classes an association graph

originates. The performance of a model is evaluated by the overall classification accuracy. We followed the standard dataset splitting procedure provided by the OGB.

MCF-7

MCF-7 is a chemical molecule dataset originally extracted from PubChem[1]. Each graph in the dataset represents a molecule, and the task is a binary classification of whether a molecule inhibits the growth of a human breast tumor cell line. Each node in a graph represents an atom and has a 1-dimensional discrete-valued feature describing the atomic number. Each edge represents a chemical bond between two atoms and has a 1-dimensional discrete-valued feature describing the bond type. This dataset has a relatively small sample size (27,770 graphs in total), with 26.4 nodes and 28.5 edges per graph on average. Model performance is evaluated by ROC-AUC. Because the TU dataset does not provide a standard data split, we applied a random 8:1:1 split into training, validation, and test sets.

2.4.3 Model Configurations

We used the graph isomorphism network (GIN) [23] as the message passing layer[2] following the OGB's reference implementation shown in Hu et al. [50], i.e., in Eqs. (2.1) and (2.2),

$$\boldsymbol{m}_n^{(l)} = \sum_{n' \in \mathrm{NBR}(n)} \mathrm{ReLU}\left(\boldsymbol{h}_{n'}^{(l-1)} + f_{\mathrm{edge}}^{(l)}(\boldsymbol{q}_{n',n})\right), \tag{2.10}$$

$$\boldsymbol{h}_n^{(l)} = f_{\mathrm{NN}}^{(l)}\left((1+\epsilon^{(l)}) \cdot \boldsymbol{h}_n^{(l-1)} + \boldsymbol{m}_n^{(l)}\right), \tag{2.11}$$

where $f_{\mathrm{edge}}^{(l)}$ is a trainable function to encode edge features into a vector, $f_{\mathrm{NN}}^{(l)}$ is a two-layer neural network for transforming node representations, and $\epsilon^{(l)}$ is a trainable scalar weight modifier.

We varied the number of GIN layers L from 1 to 10 to investigate the impact of depth in model performance. We fixed the node representation dimension d

[1] `https://pubchem.ncbi.nlm.nih.gov`

[2] We note that the MLAP architecture is applicable to any GNN models irrespective of the type of message passing layers.

to 200 and added a dropout layer for each GIN layer with a dropout ratio of 0.5. In addition, each message passing operation is followed by a GraphNorm operation [45] before dropout under the GraphNorm (+) configuration. We optimized the model using the Adam optimizer [54].

The dataset-specific settings are detailed below.

Synthetic Dataset

As the graphs in the synthetic dataset do not have the node features or edge features, we set $\boldsymbol{p}_n = 0$ and $\boldsymbol{q}_{n_{\mathrm{src}},n_{\mathrm{dst}}} = 0$. Each GIN layer had an edge feature encoder that returned a constant d-dimensional vector.

In addition to GNN, each model learned an embedded class representation matrix $\boldsymbol{E} \in \mathbb{R}^{9\times d}$. The probability with which a graph belongs to the class c was computed by the softmax function:

$$P(c|G) = \mathrm{softmax}\left(\boldsymbol{E}_c \cdot \boldsymbol{h}_G + b_c\right) = \frac{\exp\left(\boldsymbol{E}_c \cdot \boldsymbol{h}_G + b_c\right)}{\sum_{c'=1}^{9} \exp\left(\boldsymbol{E}_{c'} \cdot \boldsymbol{h}_G + b_{c'}\right)}, \tag{2.12}$$

where $\boldsymbol{E}_c$ is the c-th row vector of $\boldsymbol{E}$, and b_c is the bias term for the class c.

The models were trained against a cross-entropy loss function for 65 epochs. The initial learning rate was set to 10^{-3} and decayed by $\times 0.2$ for every 15 epochs. The batch size was 50.

ogbg-molhiv

We used the OGB's atom encoder for computing the initial node representation $\boldsymbol{h}_n^{(0)}$ from the 9-dimensional node feature. We also used the OGB's bond encoder as $f_{\mathrm{edge}}^{(l)}$, which takes the 3-dimensional edge feature as its input.

After computing the graph representation $\boldsymbol{h}_G$, a linear transformation layer followed by a sigmoid function computes the probability with which each graph belongs to the *positive* class, as in

$$P(\mathrm{positive}|G) = \sigma\left(\boldsymbol{w}_{\mathrm{prob}} \cdot \boldsymbol{h}_G + b\right), \tag{2.13}$$

where σ is a sigmoid function, and $\boldsymbol{w}_{\mathrm{prob}}$ is a trainable row vector with the same dimension d as the graph representation vectors. b is the bias term.

The models were trained against a binary cross-entropy loss function for 50 epochs. The initial learning rate was set to 10^{-4} and decayed by $\times 0.5$ for every 15 epochs. The batch size was set to 20 to avoid overfitting.

ogbg-ppa

We set $\boldsymbol{p}_n = 0$ because this dataset does not have node features. We used a two-layer neural network as $f_{\text{edge}}^{(l)}$ to embed the edge feature.

The multi-class classification procedure and the hyperparameters for optimization were identical to those used for the synthetic dataset, except that the number of classes was 37 and that the models were trained for 50 epochs.

MCF-7

We trained a vector embedding for node features as $\boldsymbol{h}_n^{(0)}$ and another vector embedding for $f_{\text{edge}}^{(l)}$ in Eq. (2.10).

The binary classification procedure and the hyperparameters for optimization were identical to those used for ogbg-molhiv.

2.4.4 Performance Evaluation (RQ1)

Baseline Models

We compared the performance of GNN models using our MLAP framework (Figure 2.1c) with two baseline models. One was a naive GNN model that simply stacked GIN layers, wherein the representation of a graph was computed by pooling the node representations after the last message passing (Figure 2.1a), as in

$$\boldsymbol{h}_G = \text{Pool}\left(\left\{\boldsymbol{h}_n^{(L)} \;\middle|\; n \in \mathcal{N}\right\}\right). \tag{2.14}$$

We used the same attention pooling as MLAP, that is,

$$\boldsymbol{h}_G = \sum_{n \in \mathcal{N}} \text{softmax}\left(f_{\text{gate}}(\boldsymbol{h}_n^{(L)})\right) \boldsymbol{h}_n^{(L)}. \tag{2.15}$$

The other was the JK architecture [14], which first computed the final node representations by aggregating layer-wise node representations, and the graph

representation was computed by pooling the aggregated node representations (Figure 2.1b) [23], as in

$$\boldsymbol{h}_G = \mathrm{Pool}\left(\left\{\boldsymbol{h}_n^{(\mathrm{JK})} \;\middle|\; n \in \mathcal{N}\right\}\right). \tag{2.16}$$

Here, $\boldsymbol{h}_n^{(\mathrm{JK})}$ is the aggregated node representation computed from the layer-wise node representation, as in

$$\boldsymbol{h}_n^{(\mathrm{JK})} = f_{\mathrm{JK}}\left(\left\{\boldsymbol{h}_n^{(l)} \;\middle|\; l \in \{1, \ldots, L\}\right\}\right), \tag{2.17}$$

where f_{JK} is the JK's aggregation function, for which we tested all three variants proposed in Xu et al. [14]—*Concatenation*, *MaxPool*, and *LSTM-Attention*—and *Sum* used in the OGB's reference implementation, defined as

$$f_{\mathrm{JK}}\left(\left\{\boldsymbol{h}_n^{(l)} \;\middle|\; l \in \{1, \ldots, L\}\right\}\right) = \sum_{l=1}^{L} \boldsymbol{h}_n^{(l)}. \tag{2.18}$$

Finally, the graph representation was computed using the attention pooling, as in

$$\boldsymbol{h}_G = \sum_{n \in \mathcal{N}} \mathrm{softmax}\left(f_{\mathrm{gate}}(\boldsymbol{h}_n^{(\mathrm{JK})})\right) \boldsymbol{h}_n^{(\mathrm{JK})}. \tag{2.19}$$

For each architecture, we trained models with varying depth (1–10).

Statistical Analyses

We trained models using 30 different random seeds, except for ogbg-ppa, for which we used 10 seeds because the dataset is bigger than others and requires a long time for training. The performance of an architecture with a certain depth was evaluated by the mean and the standard error.

Among each of the naive, JK, and MLAP architecture, we selected the best model configuration in terms of depth, type of aggregator, and GraphNorm (+) or (−). Then, we compared the performance of the best MLAP model to the best naive models and the best JK models using the Mann-Whitney U-test. In addition, we computed the effect size. Given the test statistic z from the U-test, the effect size r was computed as $r = z/\sqrt{N}$, where N is the total number of samples.

2.4.5 Analyses on Layer-Wise Representations (RQ2)

We analyzed the layer-wise graph representations to investigate the effectiveness of the MLAP architecture in learning discriminative graph representations. First, we computed the layer-wise graph representations and the final graph representation after MLAP aggregation for each graph in the datasets. We conducted two different analyses on these embedded representations.

t-SNE Visualization

We visualized the distribution of those representations in a two-dimensional space using t-SNE [55]. The t-SNE hyperparameters were as follows: the learning rate was 50, the number of iterations was 3000, and the perplexity was 20.

Training Layer-Wise Classifiers

We trained layer-wise classifiers to evaluate the *goodness* of the layer-wise representations quantitatively. We followed the classifier implementations in Eqs. (2.12) and (2.13), but the graph representation terms $\boldsymbol{h}_G$ in those equations were replaced by the layer-wise representations $\boldsymbol{h}_G^{(l)}$. These classifiers were trained on the representations of the training set. The classification performances were tested against the representations of the test set. The classifiers were optimized by the Adam optimizer for 30 epochs with a learning rate of 10^{-3}.

2.5 Results

2.5.1 Model Performances (RQ1)

We first selected the best model in terms of depth, type of aggregator, and GraphNorm (+) or (−) based on the validation performance summarized in Figures 2.3–2.6[3]. Then, we evaluated the selected models' performances using the test set (Table 2.1) and performed statistical analyses (Table 2.2).

[3]For legibility, we only plotted the results of naive architecture, the best one among four JK architectures, and the best one between two MLAP architectures in Figures 2.3–2.6. The full results are available in Appendix A.1.

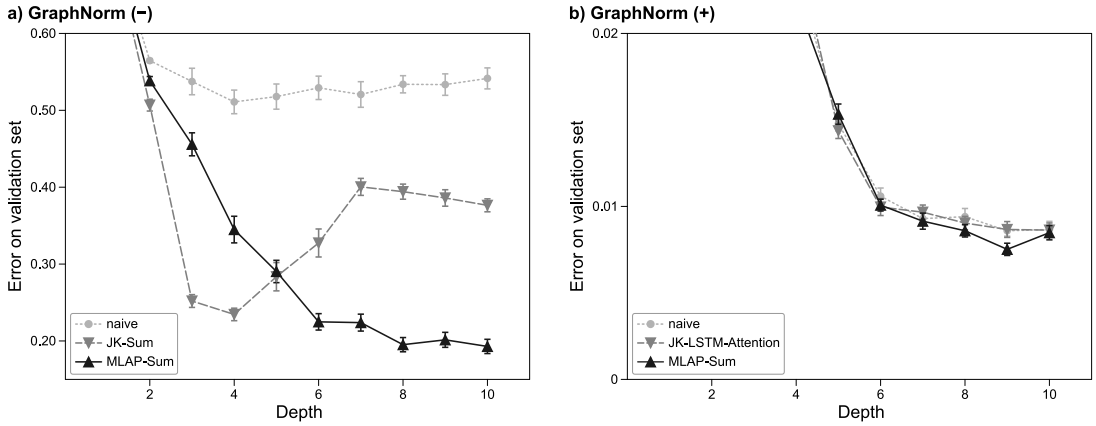

Figure 2.3: The validation performances for the synthetic dataset. Full results are provided in Appendix Table A.2.

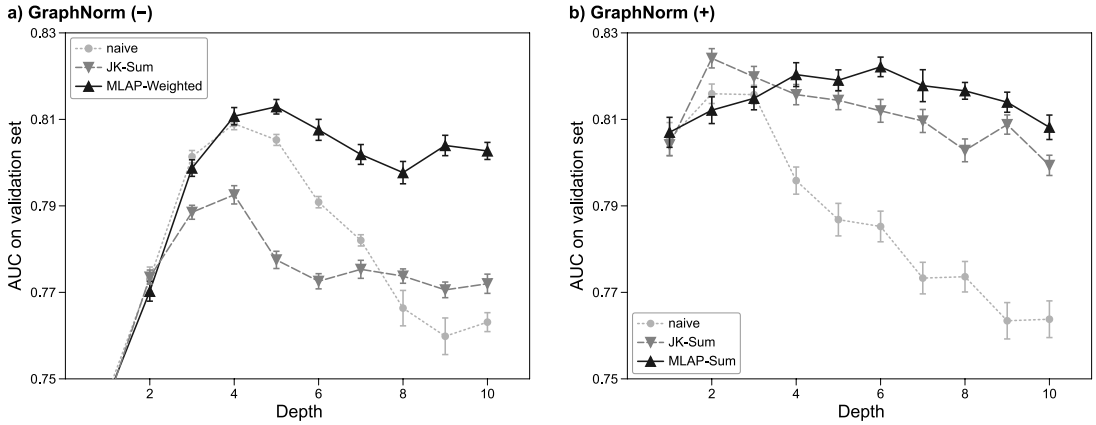

Figure 2.4: The validation performances for the ogbg-molhiv dataset. Full results are provided in Appendix Table A.3.

Synthetic Dataset

The *Synthetic* column of Table 2.1 summarizes the test performance in the synthetic dataset experiments. The 9-layer MLAP-Sum with GraphNorm model performed the best (0.0150 ± 0.0006). It was better than the best performance of the baseline models: 0.0163 ± 0.0005 for 10-layer JK-LSTM-Attention with GraphNorm. In other words, the error rate was decreased by 8.4%. The statistical tests showed that MLAP performed significantly better than both the naive and the JK architectures (Table 2.2). The effect sizes (0.345 and 0.226)

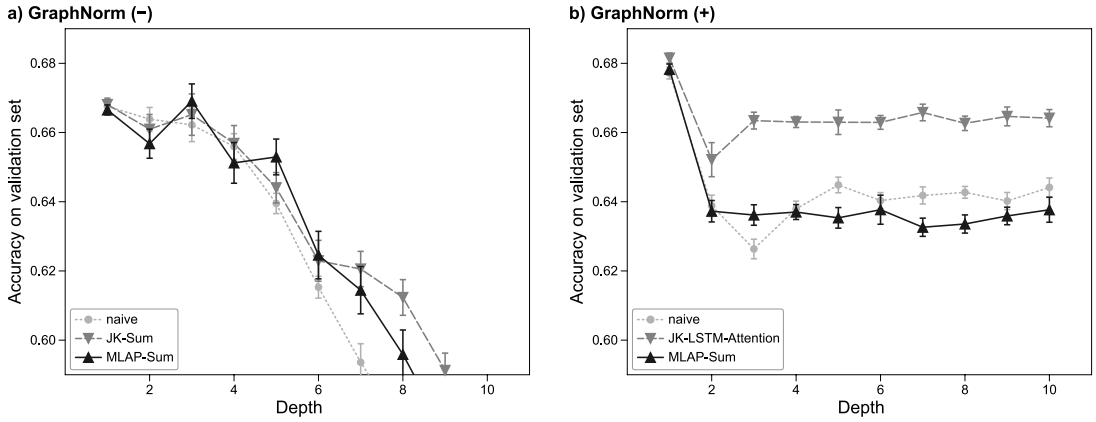

Figure 2.5: The validation performances for the ogbg-ppa dataset. Full results are provided in Appendix Table A.4.

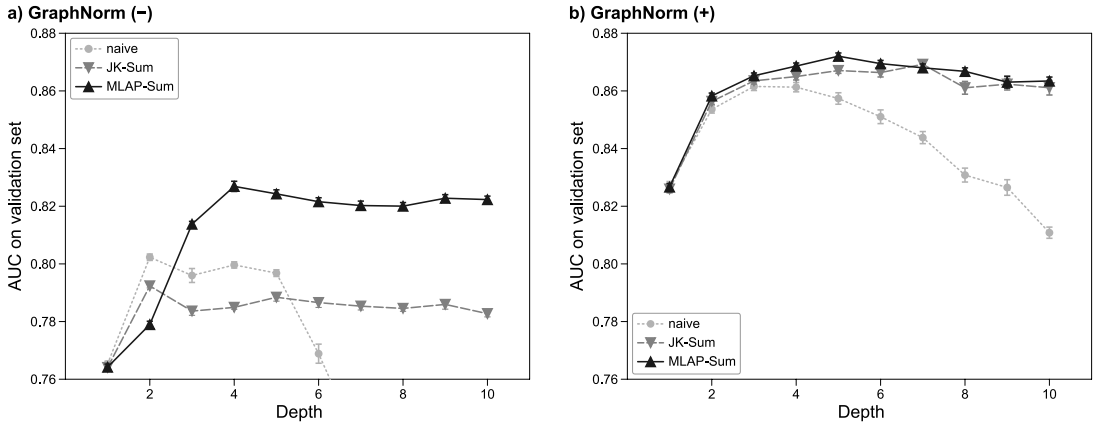

Figure 2.6: The validation performances for the MCF-7 dataset. Full results are provided in Appendix Table A.5.

were considered as moderate to small, according to the classification provided in Cohen [56, Section 3.2].

ogbg-molhiv

The *ogbg-molhiv* column of Table 2.1 summarizes the test performance in the ogbg-molhiv experiments. The best performance was achieved by the 2-layer JK-Sum with GraphNorm model (0.7708±0.0030). The performance of the JK model was better than the best MLAP model (6-layer MLAP-Sum with GraphNorm,

Base	Synthetic Aggregator (L) GN Test perf.	ogbg-molhiv Aggregator (L) GN Test perf.	ogbg-ppa Aggregator (L) GN Test perf.	MCF-7 Aggregator (L) GN Test perf.
naive	- (9) (+) 0.0175 ± 0.0007	- (2) (+) 0.7567 ± 0.0034	- (1) (+) 0.7184 ± 0.0011	- (3) (+) 0.8572 ± 0.0012
JK	LSTM-Att. (10) (+) 0.0163 ± 0.0005	Sum (2) (+) 0.7708 ± 0.0030	LSTM-Att. (1) (+) 0.7198 ± 0.0013	Sum (7) (+) 0.8572 ± 0.0012
MLAP	Sum (9) (+) 0.0150 ± 0.0006	Sum (6) (+) 0.7651 ± 0.0027	Sum (1) (+) 0.7183 ± 0.0012	Sum (5) (+) 0.8634 ± 0.0011

Table 2.1: The test performances (*mean* ± *standard error*) of the selected models. We chose the best combination of the aggregator and the model depth. GN: GraphNorm; LSTM-Att.: LSTM-Attention.

0.7651 ± 0.0027), but the difference was not statistically significant, and the effect size was small (0.153). MLAP performed significantly better than the naive model (2-layer with GraphNorm, 0.7567 ± 0.0034).

ogbg-ppa

The *ogbg-ppa* column of Table 2.1 summarizes the test performance in the ogbg-ppa experiments. The best performance was 0.7198 ± 0.0013 (1-layer JK-LSTM-Attention with GraphNorm). For this dataset, the single-layer model performed the best within each architecture; hence, six out of seven types of models (naive, JK-Sum/-Concat/-MaxPool, and MLAP-Sum/-Weighted) have exactly the same form. Only JK-LSTM-Attention has extra parameters, and thus it is reasonable that it achieved the best performance owing to these parameters, rather than the hierarchical graph representations.

MCF-7

The *MCF-7* column of Table 2.1 summarizes the test performance in the MCF-7 experiments. MLAP-Sum achieved the best test performance (L=5 with GraphNorm, 0.8634 ± 0.0011). These performances are significantly better than the baseline performances with moderate to large effect sizes (0.407–0.418).

Comparison	Synthetic		ogbg-molhiv		ogbg-ppa		MCF-7	
	p	E.S.	p	E.S.	p	E.S.	p	E.S.
MLAP *vs.* naive	*0.004	0.345	*0.039	0.229	0.367	−0.085	$*< 10^{-3}$	0.418
MLAP *vs.* JK	*0.039	0.226	0.120	−0.153	0.192	−0.203	$*< 10^{-3}$	0.407

Table 2.2: The statistical analysis results. We compared the best performance among MLAP models to naive models and JK models. p: p-value of the Mann-Whitney U-test. *: significant difference. E.S.: effect size r. Note that the negative values mean that the naive architecture or JK was better than MLAP.

2.5.2 Analyses on Layer-Wise Representations (RQ2)[4]

Synthetic Dataset

We visualized the learned layer-wise and the aggregated graph representations by a 10-layer MLAP-Sum model with a validation error rate of 0.9056 (Figure 2.7). There were $3 \times 3 = 9$ classes of graphs in the dataset, determined by the combination of the center component type and the peripheral component type (top-right panel in Figure 2.7). The representations in the lower layers were highly discriminative for the *peripheral* types shown by the brightness of the dots. For the *center* types shown by the hue (i.e., red, green, and blue), the higher layers (Layer 6–8) seemed to have slightly better discriminative representation than the lower layers, although it was not as clear as *peripheral* types. The aggregated representations were clearly discriminative for both the center and the peripheral types.

We quantitatively evaluated this observation using layer-wise classifiers for all trained 10-layer models with 30 different random seeds. Figure 2.8a shows the layer-wise classification performance on the training and the test sets. Although the test error rate for each layer-wise representation was not under 0.60, the aggregated representation by MLAP achieved a significantly smaller error rate (0.2093 ± 0.0096; U-test, $p < 10^{-5}$ [Bonferroni corrected]).

In addition to the 9-class classifiers, Figure 2.8b shows the layer-wise classification performance under the *3-class* settings—each classifier was trained to

[4]In these analyses, we used GraphNorm (−) models so that we can avoid the interaction between MLAP and GraphNorm.

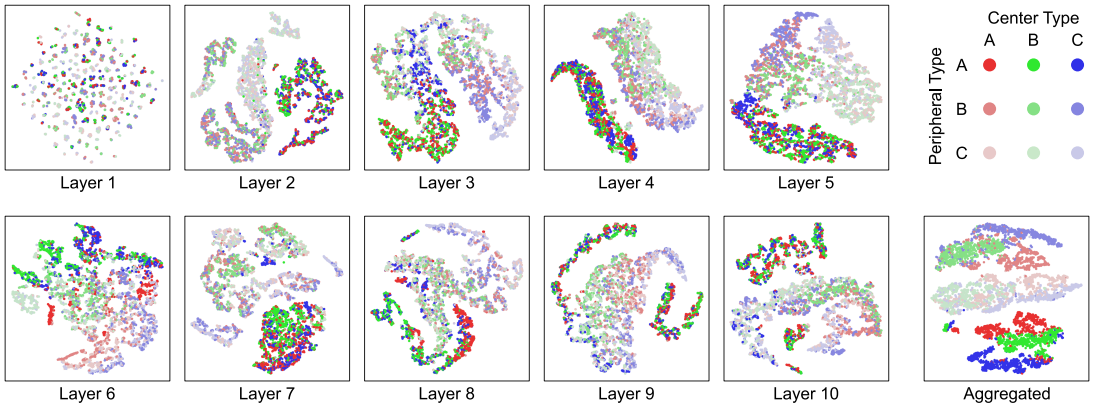

Figure 2.7: The layer-wise graph representations for the graphs in the synthetic dataset. They are visualized in two-dimensional spaces using t-SNE. Dots in each color represent samples in a class.

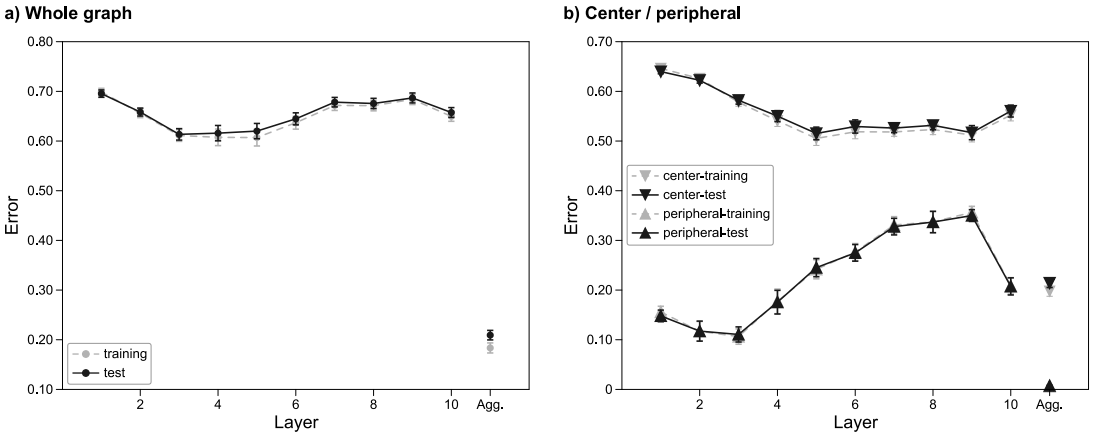

Figure 2.8: The training and test classification performances of the layer-wise representations computed for the graphs in the synthetic dataset. The "Agg." in the horizontal axis indicates the classifier's performance trained with the graph representations after MLAP aggregation. **a)** 9-class. **b)** 3-class (center or peripheral).

predict either the center type or the peripheral type. The results in Figure 2.8b show the discriminability among three *peripheral* types had the peak at Layer 1–3, and aggregating those layer-wise representations resulted in an error rate of almost 0. On the other hand, the discriminability among *center* types was better in higher layers (Layer 5–9), but the layer-wise error rate (best: 0.5152) was higher than that of *peripheral* types. Nonetheless, the aggregated represen-

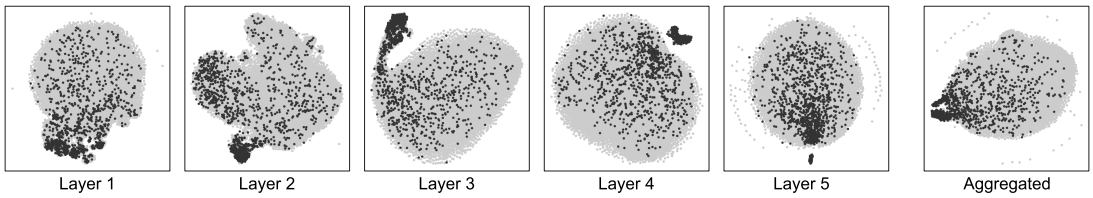

Figure 2.9: The layer-wise graph representations for ogbg-molhiv graphs. They are visualized in two-dimensional spaces using t-SNE. Each gray dot represents a *negative* sample, whereas each black dot represents a *positive* sample.

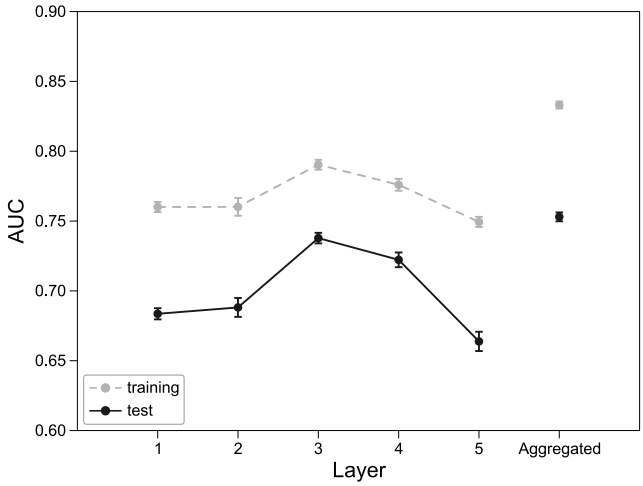

Figure 2.10: The training and test classification performances of the layer-wise representations computed for ogbg-molhiv graphs. The "Aggregated" in the horizontal axis indicates the classifier's performance trained with the graph representations after MLAP aggregation.

tation achieved an error rate of 0.2142, which was much better than any of the layer-wise representations. The 9-class classification performance (Figure 2.8a) had its peak in middle layers (Layer 3–5), which was right in between the two 3-class classifiers. These results are consistent with the qualitative observation in Figure 2.7, which indicates that the graph structures in different level of locality were captured in different MLAP layers.

ogbg-molhiv

In Figure 2.9, we show the layer-wise representations by a 5-layer MLAP-Weighted model trained with the ogbg-molhiv dataset with a validation AUC

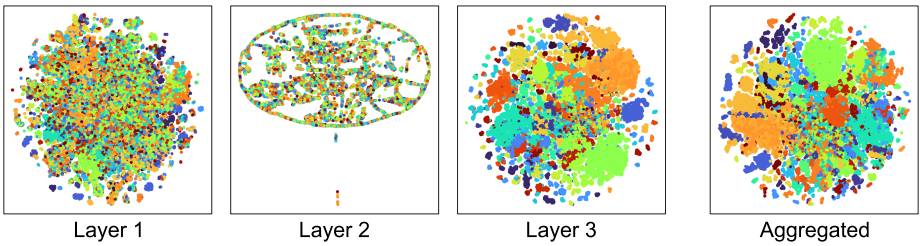

Figure 2.11: The layer-wise graph representations for ogbg-ppa graphs. They are visualized in two-dimensional spaces using t-SNE. Dots in each color represent samples in a class.

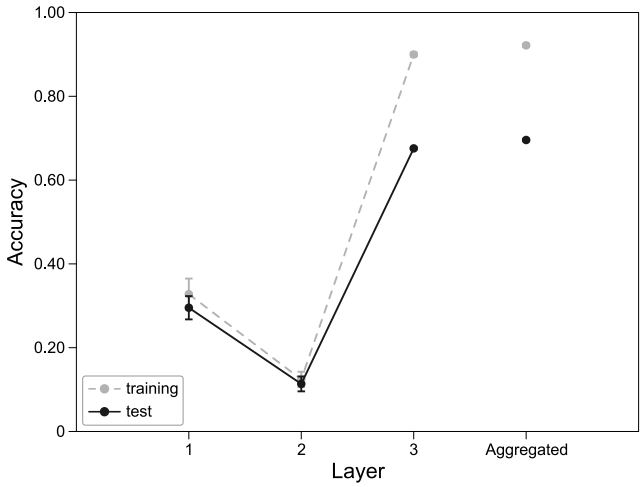

Figure 2.12: The training and test classification performances of the layer-wise representations computed for ogbg-ppa graphs. The "Aggregated" in the horizontal axis indicates the classifier's performance trained with the graph representations after MLAP aggregation.

score of 0.8282. Each gray dot represents a *negative* sample, whereas each black dot represents a *positive* sample. The discriminability between the two classes was slightly better in the lower layers. Aggregating those representations by considering a weighted sum produced a more localized sample distribution than any representations in the intermediate layers.

The analysis using the layer-wise classifiers supported the intuition obtained from the t-SNE visualization. Figure 2.10 shows the training and test AUC scores for each layer-wise classifier. The best test score among the intermediate layers (0.7378 ± 0.0038) was marked at Layer 3, and the score after MLAP aggregation was better than this (0.7530 ± 0.0031). The differences in discriminability between

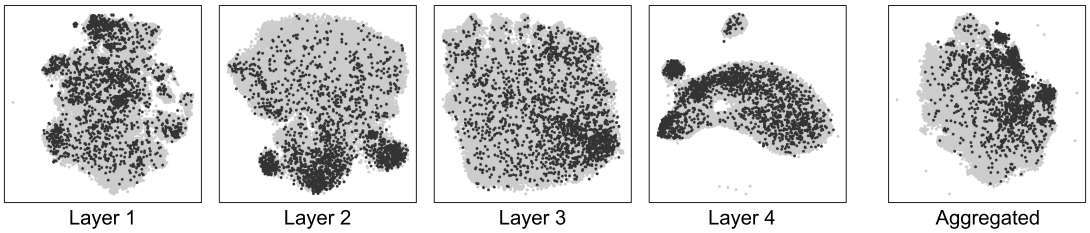

Figure 2.13: The layer-wise graph representations for MCF-7 graphs. They are visualized in two-dimensional spaces using t-SNE. Each gray dot represents a *negative* sample, whereas each black dot represents a *positive* sample.

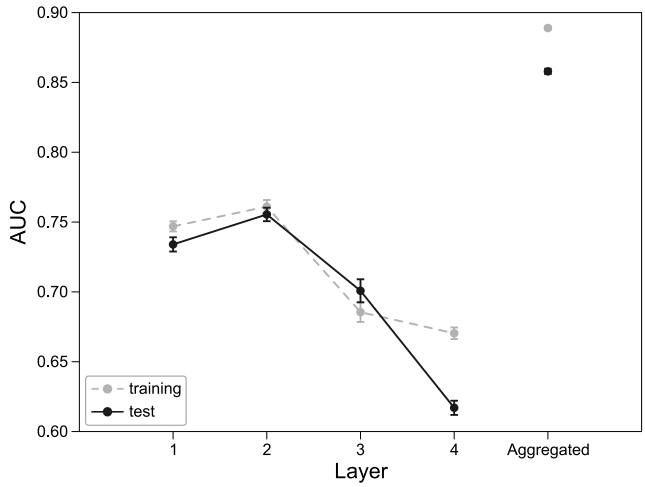

Figure 2.14: The training and test classification performances of the layer-wise representations computed for MCF-7 graphs. The "Aggregated" in the horizontal axis indicates the classifier's performance trained with the graph representations after MLAP aggregation.

each layer-wise representation and the aggregated representation were significant (U-test, $p = 0.012$ between Layer 3 and aggregated, $p < 10^{-5}$ for other layers [Bonferroni corrected]) with moderate to large effect sizes ($r = 0.366$–0.855).

ogbg-ppa

Figure 2.11 shows the t-SNE visualization results of the layer-wise representation by a 3-layer MLAP-Sum model (Accuracy = 0.6854). Layer 3 showed the best discriminative representation, whereas representations in Layer 1 and 2 did not seem clearly discriminative. Furthermore, the discriminability in the MLAP-

aggregated representation seemed at a similar level to Layer 3.

The layer-wise classifier analysis also showed similar results (Figure 2.12). The representations in Layer 3 achieved the best test score (0.6758 ± 0.0029). The score for the aggregated representations was slightly better (0.6952 ± 0.0029), whereas the differences in discriminability between each layer-wise representation and the aggregated representation were significant (U-test, $p = 0.001$ between Layer 3 and aggregated, $p < 10^{-3}$ for other layers [Bonferroni corrected]) with large effect sizes ($r = 0.761$–0.845).

MCF-7

Figure 2.13 shows the layer-wise representation by a 4-layer MLAP-Sum model (AUC = 0.8437). Layer 2 showed the best discriminative representation among layers 1–4, and the discriminability in the aggregated representation was even better.

The layer-wise classifier analysis supported the qualitative results. The test performance of the layer 2 and the aggregated representations were 0.7295 ± 0.0040 and 0.8149 ± 0.0016, respectively, and the latter was significantly better than any layer-wise representations ($p < 10^{-5}$ [Bonferroni corrected], effect size $r = 0.859$).

2.6 Discussion

In this study, we proposed a compositionality-aware GNN architecture called MLAP, which introduces layer-wise attentional graph pooling layers and computes the final graph representation by unifying the layer-wise graph representations. Experiments showed that our MLAP framework, which uses the structural information of graphs from multiple levels of localities, significantly improved the classification performance in two out of four tested datasets, and it showed less inferior performances to baseline methods in other two datasets. The performance of the *naive* architecture tended to degrade as the number of layers increased. This is because the deep *naive* models lost the local structural information through many message passing steps due to oversmoothing, even though the GraphNorm might mediate the effect of oversmoothing. On the other

hand, the difference in performance between *MLAP* and *JK* would be because of the operation order between the graph pooling and the information aggregation from multiple levels of localities. *MLAP* computes the graph representations by $f_{\text{agg}}\left(\text{Pool}^{(l)}(\boldsymbol{h}_n^{(l)})\right)$, whereas *JK* computes them by $\text{Pool}\left(f_{\text{JK}}(\boldsymbol{h}_n^{(l)})\right)$. As *JK* aggregates the node representations from multiple levels of localities *before* the pooling, it may be difficult for the attention mechanism to learn which node to focus on. In other words, structural information in a specific locality might be squashed before the pooling operation. By contrast, the *MLAP* architecture can tune the attention on nodes specifically in each information locality because it preserves the representations in each locality independently. It is also supported by the observation that, for datasets with hierarchical nature, *MLAP* performed better than *JK* even if *JK* has an aggregator with high expressivity such as *LSTM-Attention*[5].

The analyses on the layer-wise graph representations supported our motivation behind MLAP—GNN performance can be improved by aggregating representations in different levels of localities. In the analyses using the synthetic dataset, the discriminability of the representations in the higher layers were worse than that in the lower layers (Figure 2.8). However, using 3-class classifier analyses, we showed that the learned representations had better discriminability of the *peripheral* types in the lower layers, whereas the discriminability of the *center* type was better in higher layers. These results indicated that, even though the apparent classification performances in higher layers were low, those layers indeed had essential information to classify the graphs in the dataset correctly. Aggregating layer-wise representations from multiple steps of message passing has the potential to reflect all the necessary information from various levels of localities in the final graph representation, resulting in performance improvement. The results from real-world dataset experiments were also supportive. For the molecular datasets (ogbg-molhiv and MCF-7), the performance improvement by MLAP would be because of the hierarchical structure of biochemical molecules, whose function is determined by the combination of commonly observed substructures such as carbohydrate chains and amino groups. The MLAP architecture would

[5]In addition, it is provable that MLAP encompasses JK under certain conditions. See A.2 for the proof.

effectively capture such patterns in lower layers and their combinations in higher layers. Similarly, MLAP also performed well for ogbg-ppa datasets when the model did not have GraphNorm. This might be because protein-protein association graphs have fractal characteristics [57], for which aggregating multi-locality features would be beneficial. These results imply that MLAP can use the compositional nature of the graphs—the feature of a whole graph is determined based on the combination of the features of smaller subgraphs.

The aggregation mechanism of the layer-wise representations in JK and MLAP has the advantage of being able to coincide with nearly any other GNN technique. For example, we can apply JK or MLAP for any backbone GNN architecture (e.g., GCN, GIN, GAT). In addition, they can co-exist with the residual connection architectures or normalization techniques. The aggregation mechanism potentially improves the performance of GNN models coordinately with these techniques. Many previous GRL studies have adopted JK architecture in their models and reported performance improvement. In this study, we followed the idea to aggregate layer-wise representations, and we showed that combining the aggregation mechanism with layer-wise attention pooling can further improve the learned graph representation for graph-level classification tasks. Our experimental results validated that MLAP can be used with GraphNorm [45]. The performance of *MLAP + GraphNorm* was significantly better than *naive + GraphNorm* and *JK + GraphNorm* for the synthetic and MCF-7 dataset, although it was comparable to the baselines for the OGB datasets. Comparing these results to those observed in *without*-GraphNorm configuration, the advantage of MLAP over naive and JK was relatively weakened under the existence of GraphNorm. We consider that it is because GraphNorm normalizes the node representation across the entire graph, which might prevent MLAP from learning the representation of the local structures.

Another interesting observation is that MLAP-*Weighted* performed worse than MLAP-*Sum* in some datasets. We speculate that having weight parameters for layers in the aggregation process might induce instability in the training phase. A.3 provides preliminary results supporting this hypothesis. We will continue analyzing the cause of this phenomenon, and it may provide new insights toward further improvements in the MLAP architecture.

2.6.1 Concluding Remarks

In this study, we proposed the MLAP architecture for GNN models, that explicitly makes use of the compositionality in graphs. The results suggested that the proposed architecture was effective to learn graph representations with high discriminability. There are many kinds of real-world networks whose properties are represented in the substructures with multiple levels of localities, and applying MLAP may improve the performances of GRL models.

Chapter 3

MLAP for graph2seq: Utilizing Compositionality in Generating Sequence from Graph

We proposed the MLAP architecture for GNNs and demonstrated how it improves the model performance in graph classification tasks in the previous chapter. In this chapter, we extend the MLAP architecture for generating sequences from graphs (*graph2seq* learning). As a real-world example, we apply it in a source code summarization task and demonstrate that it achieves a SoTA performance.

3.1 Introduction

Graphs are a type of data structure that is capable of encoding a set of rich pairwise information among elements. However, in exchange for the expressivity of a complex graph, it is often difficult for humans to find patterns in such a graph. Therefore, there exists a growing interest in learning to generate a human-interpretable output from a graph. In this study, we focus on generating a sequence of discrete symbols from a graph, known as *graph2seq* tasks. For example, if we could generate a natural question from knowledge graphs, we could use them to develop an artificial intelligent agent that can interact with people [58, 59]. Furthermore, we could solve a path-planning problem as an instance of graph2seq tasks when we regard the input map as a graph [60].

Moreover, traditional sequence to sequence (*seq2seq*) tasks, like neural machine translation (NMT), could benefit from transforming them into graph2seq tasks. Seq2seq studies have achieved significant successes by introducing the encoder-decoder architecture. Pioneering studies applied recurrent neural networks to encode the input sequence [61, 62], and later transformer-based architectures dominated the field [63, 64]. Either way, the input to the models are processed as one-dimensional *sequences*. On the other hand, many kinds of input sequences in seq2seq tasks can actually be represented as *graphs* with enriched information. In many cases, we can derive rule-based conversions from raw input sequences to graphs by considering the context and the compositionality in the sequence—e.g., we can convert a natural language sentence to a graph by analyzing its grammatical structure and dependency relations. Because of such additional information, we can expect that graph2seq models utilizing graph structures in seq2seq tasks have a potential to learn improved representations [65].

Therefore, developing a graph2seq model using GNNs has recently piqued the interest. For example, Marcheggiani and Perez-Beltrachini [66] introduced a simple graph2seq model that combines a GCN encoder and an LSTM decoder to generate a text description from a resource description framework graph or a semantic dependency graph. Xu et al. [67] proposed an attention-based graph2seq model for graph reasoning tasks like bAbI or shortest path search. Chen et al. [68] and Wei et al. [69] applied similar attentional graph2seq techniques in question generation and opinionated text summarization tasks, respectively. Zhu et al. [70] and Cai and Lam [71] proposed graph-transformer-based graph2seq methods, which consider the relationship not only between neighbor nodes but also between distant nodes. However, these methods did not consider the compositionality and hierarchy of the relations among nodes. Chen et al. [68] proposed the Heterogeneous Graph Transformer, that explicitly considers the hierarchy in the input graphs. It splits the original graph into multiple subgraphs and computes the graph representation by aggregating the subgraph representations. It improved the performances of multiple text generation tasks including NMT, owing to its ability to utilize the compositional nature of graphs. However, HetGT only considers single-level hierarchy.

Our assumption in this study is that considering multi-level compositionality

in the graphs is effective to learn good representation for generating sequences. The output sequences of graph2seq tasks often have multi-level compositional nature. For example, words in a sentence form local phrases—e.g., an adjective generally modifies a noun next to the adjective—and then phrases link each other and form longer parts. Here, we speculate that the compositionality in a graph can be associated to the compositionality of the output sequence.

In this study, we apply the MLAP architecture for graph2seq learning, and demonstrate that the compositionality-aware GNN architecture is effective to learn graph representations for generating sequence. As a real-world example, we take up the "extreme source code summarization" task—generating a short natural language summary of a software program snippet [72]. Although program source code is usually written in a programming language (e.g., C, Java, Python) and represented as a sequence of tokens, it can be converted into graphs by syntactic rules of the language (see Section 3.2.2). Also, each function in source code has a name summarizing the behavior of the part of program. Therefore, a task to estimate the function name from a program graph is a good example of graph2seq learning. Representation learning on source code itself has a rich literature (we refer readers to Section 3.2 for related studies). However, many of the methods proposed in those studies do not explicitly consider the compositionality of source code at all, or if they do, they consider only a part of it. Program graphs have both short and long dependencies in them, that is, short dependencies form individual operations and longer ones structure complex algorithms. Therefore, considering compositionality in the graphs can improve the performance. In this study, we train a MLAP model on a source code dataset and show that our proposed method outperforms the SoTA model.

The rest of this chapter is organized as follows: Section 3.2 summarizes related studies on machine learning techniques for source code, Section 3.3 extends the MLAP framework for graph2seq learning, Section 3.4 describes the setup for the source code summarization experiments, and Section 3.5 and Section 2.6 demonstrates and discusses the results.

3.2 Related Works

One of the primary goals of software engineering research is to develop methods and tools that reduce the complexity of the software development process and thus the mental burden on programmers. To achieve this goal, initial studies for artificial program comprehension focused on formal methods based on predicate logic (see D'silva et al. [73] for a review paper and Bérard et al. [74] for an introductory book). Although these formal logic-based techniques achieved remarkable success in program verification and bug finding, the lack of flexibility in those techniques restricted the real-world application.

Rather than formal analyses, researchers started seeking techniques to *learn* representation models of source code. The open-source software (OSS) ecosystem has grown rapidly in recent years , providing software engineering researchers with a plethora of freely-available real-world-oriented source code. Using these source code, machine learning models learn how to embed source code into vector representations and use them to solve software engineering tasks. Hindle et al. [75] facilitated this idea by introducing the *naturalness of source code*, assuming that common styles to write *natural* programs can be modeled as statistical distributions. Subsequently, Allamanis et al. [76] named this research trend *machine learning for big code*.

In this section, we review the recent big code studies in terms of how they compute the code representation. These studies can be classified into two categories: one using natural language processing (NLP) techniques, and the other using GNNs.

3.2.1 Natural Language Processing-Based Methods for Big Code

Many big code studies adopted methods developed for NLP because source code is written as sequences of tokens. For example, Movshovitz-Attias and Cohen [77] used n-gram topic models to predict class-level comments. Corley et al. [78] and Bavishi et al. [79] used token contexts, which comprise neighbor tokens of a token, to learn the token-wise representation. Allamanis et al. [80] also used token

contexts, but they trained method- and class-level representations for predicting their names. Dam et al. [81] used long short-term memory (LSTM) to learn the representation of source code snippets.

In NLP, the attention mechanism introduced by Bahdanau et al. [25] demonstrates a drastic performance improvement, and thus majority of the recent NLP models adopted it. Led by the success in NLP, program comprehension models have applied the attention mechanism as well. Iyer et al. [82] trained an LSTM model with the attention mechanism to generate a natural language summary for an input source code. Jiang et al. [83] also used an attentional recurrent neural network (RNN) model to generate a summary given a set of code changes. Xu et al. [84] proposed a hierarchical attentional RNN model to improve the code representation. They first divided a function to multiple basic blocks[6] and then trained a model composed of a token-level RNN and a block-level RNN. They demonstrated that the hierarchical attention architecture improved the model performance in the function naming task. Rather than using RNN, Allamanis et al. [72] proposed to compute the code representation as an attention-weighted sum of token embeddings. They used a one-dimensional CNN over the token sequence to compute the attention weight in a context-dependent way.

Of late, Vaswani et al. [63] proposed a model called transformer. Rather than using RNNs to process sentence as *sequences*, the transformer model considers a sentence as a *set* of words and computes the representation entirely using the attention mechanism. Based on the transformer, massively pretrained models have dominated the field of NLP. These pretrained models—BERT [64], GPT [85], or XLNet [86]—learn embedded representations of sentences in an unsupervised manner, and then they are fine-tuned against specific tasks. Big code studies instantly adopted these techniques. Kanade et al. [87] applied the BERT to Python code dataset, and demonstrated the improved performance in multiple classification tasks. Feng et al. [88] trained a bimodal BERT model of source code and natural language to generate natural language summaries for given source code. Svyatkovskiy et al. [89] applied GPT-2 [90] on the source code dataset for a code completion task. In addition, Roziere et al. [91] proposed a transformer-based encoder-decoder model to *translate* source code written in one programming

[6] A basic block is a set of sequentially executed statements without jumps.

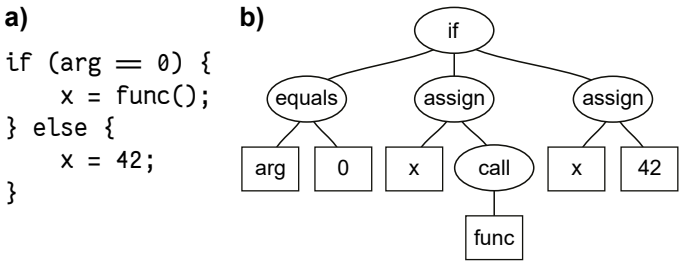

Figure 3.1: An example of an abstract syntax tree. The syntactic properties expressed in the code snippet (a) is equally represented as an AST (b), emphasizing the compositionality of the source code and discarding the minor syntactic details like brackets or semicolons.

language to another, such as C to Java.

These studies have been developed utilizing a rich literature in NLP research. However, as they process source code as sequences of tokens, they find difficulties in capturing the compositionality in source code. As a result, those models, particularly recent transformer-based ones, require hundreds of millions of parameters to solve the tasks. Programming languages have strictly defined syntaxes, unlike natural languages which have ambiguities, and thus we should be able to take advantage of this property.

3.2.2 Using Graph Structures in Programs

Although source code is commonly written as a linear sequence of tokens, a compiler or an interpreter internally converts the source code into graphs representing the context and the compositionality of the source code. For example, an abstract syntax tree (AST) represents the syntactic structure of source code (Figure 3.1). Source code can be hierarchically decomposed into modules, functions, statements, expressions, etc.; an AST represents such compositionality in the source code. Each non-leaf node represents a syntactic element (e.g., a `for` statement or a variable assignment) whereas each leaf node represents an operand (e.g., a variable name or a constant). Another example is a control flow graph (CFG) representing all possible computation paths that can be traversed during program execution. A CFG is a directed acyclic graph wherein each node represents a statement or a block of statements that are executed sequentially without branching. Hence, a CFG represents the context of computation of a program.

Several pioneering studies showed that machine learning models for big code can greatly benefit from such data structures, because the structures allow models to focus on the core semantics of programs [92–94]. Hence, recent big code studies actively explore methods to incorporate the program graphs (e.g., ASTs or CFGs) to improve the embedded representation of code. These studies can be divided into two categories: 1) *linearlize* the graphs to avoid direct handling of the structures in indeterminate forms, or 2) use neural networks designed for graph structures, including but not limited to GNNs.

Graph Linearization

DeFreez et al. [95] proposed FUNC2vec, which used a random walker to linearize pushdown system graphs (similar to CFGs). The random walker samples the valid control paths during execution which we can use for representing the behavior of the program under a specific context. Using the set of control paths, representation of the program is trained by the continuous bag-of-words method proposed for word2vec [96]. Code2vec [97] and code2seq [98] defined the path contexts to linearlize ASTs. A path context is composed of a pair of leaf nodes in an AST and the sequence of non-leaf nodes on the path connecting those two leaves. Code2vec and code2seq first compute the representation for each path context and then the final code representation by aggregating path context representations using attention mechanism.

Methods based on graph linearization are useful because one can build their models upon the rich resources of standard machine learning techniques. However, extracting linearized paths from graphs is often computationally expensive. For example, path contexts used in code2vec and code2seq can be defined for any pair of leaf nodes [97, 98]. Thus, there are N^2 possible paths for an AST with N leaves, which can be an impractically large number for large code snippets. Although Alon et al. [97] claimed that sampling a small proportion of possible paths is enough to train a good model, this kind of sampling may damage the model expressivity, particularly for large code snippets. In addition, the linearization procedure often occludes the compositionality of the structured data.

Neural Network Methods for Graphs

Recent big code studies focused more on processing program graphs *as is* using specially tailored neural network techniques, rather than linearizing them. Peng et al. [99] and Mou et al. [100] introduced tree-based convolutional neural networks (TBCNNs) for ASTs. TBCNNs recursively compute the vector representation of nodes from the leaves of an AST to the root by aggregating the child nodes' representations; hence it can be regarded as a special case of recent message-passing GNN where a graph does not have cycles. Subsequently, Zhang et al. [101] used TBCNN to compute statement-wise representation in a function and learn function representation by aggregating the sequence of statement representations using gated recurrent unit [61].

Rather than building these kinds of tailored neural network models, use of GNNs has become popular (see Section 2.2 for the history of GNN). These approaches can also be seen as graph2seq applications in big code studies. Program comprehension studies using GNNs commonly used *augmented* ASTs, because using GNNs on tree data structure—which is a very sparse graph structure—is inefficient. Allamanis et al. [102] and Fernandes et al. [65] trained gated graph neural network [13] on ASTs augmented with edges connecting consecutive tokens and those representing variable dependency. Based on these studies, Cvitkovic et al. [103] further augmented ASTs with a vocabulary cache, which helps the model to detect the semantic connection between variables. Liu et al. [104] used a retrieval database, which connects the input source code to a set of similar source code, to augment ASTs and applied a novel hybrid graph2seq model.

Using program graphs as the inputs might help the models to capture the compositionality of programs, but they can only do so in implicit manners because they use the graph representations after fixed numbers of message passing steps. Therefore, in this study, we apply MLAP to a source code summarization task and show that explicitly considering the compositionality of program graphs improves the learned representation.

3.3 Methods—MLAP for graph2seq Learning

In this section, we extend the MLAP architecture introduced in Section 2.3 for graph2seq tasks. We evaluate two types of sequence decoder in this study: *Linear* and *LSTM*. The former selects elements in a sequence independently, whereas the latter considers the context through the sequence.

In both the cases, suppose we have computed the layer-wise representations $\boldsymbol{h}_G^{(l)}$ $(l = 1, \ldots, L)$ and the final graph representation $\boldsymbol{h}_G$ in Eq. (2.5) and Eq. (2.7), respectively.

3.3.1 Linear Decoder

The most simple implementation of a decoder uses a linear word classifier solely based on the final graph representation, independently for each position in the decoded sequence. That is, we use five independent classifiers if we want to decode five-word sequences.

The model learns *position-wise* embedded representation matrix for the target vocabulary $\boldsymbol{E}_i^{\text{voc}} \in \mathbb{R}^{N^{\text{voc}} \times d}$ for each position i, where N^{voc} is the size of vocabulary that consists the generated sequence. Here, the words in the vocabulary with ids 0, 1, and 2 have special meanings: the start of a sentence (SOS), an unknown word (UNK), and the end of a sentence (EOS).

The selection probability for a word w from vocabulary in each position is computed as

$$p_{i,w} = \text{softmax}\left(\boldsymbol{h}_G \cdot \boldsymbol{E}_{i,w}^{\text{voc}} + b_{i,w}\right) \quad \forall\, i \in \{1, \ldots, I\}, \tag{3.1}$$

where b_i is a bias parameter and I is the length of decoded sequences. If one or more classifiers select the word EOS for any positions, the words after the first EOS are ignored.

3.3.2 LSTM Decoder

The aforementioned Linear decoder is rather naive, and each word is decoded independent of other words. Thus, it cannot consider the context among decoded

words. Instead, we can use a LSTM-based decoder with an attention mechanism [105] to generate a sequence from those representations.

In this case, the model learns an *position-invariant* embedded representation matrix for the target vocabulary $\boldsymbol{E}^{\text{voc}} \in \mathbb{R}^{N^{\text{voc}} \times d}$.

We first initialize the LSTM state $\boldsymbol{h}_0^{\text{dec}}$ and the memory cell $\boldsymbol{m}_0^{\text{dec}}$ using the aggregated graph representation as

$$\boldsymbol{h}_0^{\text{dec}} = \boldsymbol{h}_G, \tag{3.2}$$

$$\boldsymbol{m}_0^{\text{dec}} = \boldsymbol{h}_G. \tag{3.3}$$

At the step t ($t = 1, \ldots, T$), the state and the memory cell are updated by a standard LSTM [106].

$$\boldsymbol{h}_t^{\text{dec}}, \boldsymbol{m}_t^{\text{dec}} = \text{LSTM}(\boldsymbol{x}_t^{\text{dec}}, \boldsymbol{h}_{t-1}^{\text{dec}}, \boldsymbol{m}_{t-1}^{\text{dec}}), \tag{3.4}$$

$$\boldsymbol{x}_t^{\text{dec}} = \begin{cases} \boldsymbol{E}_{\text{SOS}}^{\text{voc}} & \text{if } t = 0, \\ \boldsymbol{y}_{t-1}^{\text{dec}} & \text{otherwise.} \end{cases} \tag{3.5}$$

Here, $\boldsymbol{y}_t^{\text{dec}}$ is the output of the decoder at t (see following).

To determine the output, the model first compute a context vector using an attention mechanism over the layer-wise and final graph representations. Here, for notation simplicity, we consider $\boldsymbol{h}_G^{(L+1)}$ be $\boldsymbol{h}_G$, and then the computation is as

$$\boldsymbol{c}_t^{\text{dec}} = \sum_{l=1}^{L+1} \text{softmax}\left(\boldsymbol{h}_G^{(l)} \cdot \boldsymbol{h}_t^{\text{dec}}\right) \boldsymbol{h}_G^{(l)}. \tag{3.6}$$

Then, it computes the output as follows.

$$\boldsymbol{y}_t^{\text{dec}} = \tanh\left(\text{LayerNorm}\left(\boldsymbol{W}^{\text{dec}} \cdot [\boldsymbol{c}_t^{\text{dec}}; \boldsymbol{h}_t^{\text{dec}}] + \boldsymbol{b}^{\text{dec}}\right)\right), \tag{3.7}$$

where $\boldsymbol{W}^{\text{dec}}$ and $\boldsymbol{b}^{\text{dec}}$ are trainable parameters. The selection probability for each word in the vocabulary at t is computed as

$$p_{t,w} = \text{softmax}\left(\boldsymbol{y}_t^{\text{dec}} \cdot \boldsymbol{E}_w^{\text{voc}} + b_w\right), \tag{3.8}$$

where b_w is a bias parameter. The LSTM decoder stops when the word EOS is generated or the step t reaches the maximum step T.

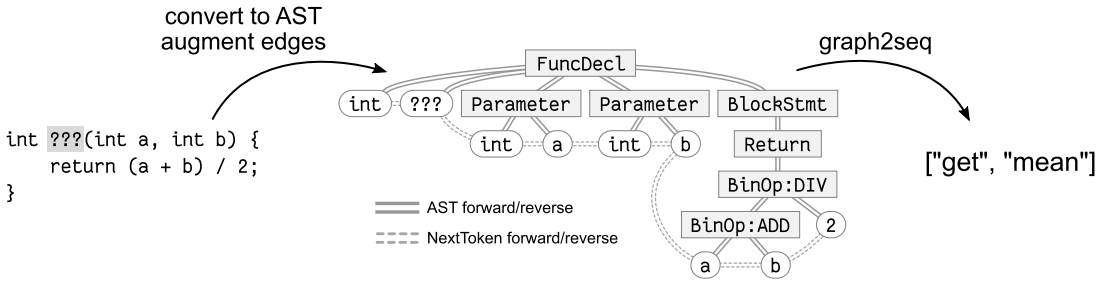

Figure 3.2: A schematic illustration of the extreme code summarization task. A source code snippet with a masked function name, is provided as the input. The snippet is converted into an AST, and then augmented with additional types of edges in preprocess. Models are trained to recover the original function name (`get_mean`).

3.4 Experiments

3.4.1 Task and Dataset

To evaluate the MLAP-graph2seq model, we employed the "extreme source code summarization" task [72], wherein a model predicted the function name for given source code snippet (Figure 3.2). Because a function is commonly named in such a way that the name describes the behavior of the function, recovering the function name from its body would be one of the fundamental tasks that evaluates the models' capability to understand source code semantics.

Here, we used the ogbg-code2 dataset from OGB collection [50]. It contains 452,741 ASTs, each of which contains 125.2 nodes and 124.2 edges on average, extracted from 13,587 OSS projects. The average length of the ground-truth sequence is 2.25. Each node has a 3-dimensional discrete-valued feature containing the depth of node counted from the root of the AST, the type of AST node (e.g., `If` or `Num`), and the node attribute. Here, only the leaf nodes have node attributes, usually the token string associated to the node. Non-leaf nodes have blank attributes.

3.4.2 Preprocess

As introduced in Allamanis et al. [102], we augmented the AST graph with additional types of edges. The original graphs in the ogbg-code2 dataset only have one

type of edges that point from parent nodes to child nodes, and thus models suffer from low efficiency in information diffusion during message passing. Therefore, we added three additional types of edges to the graphs (Figure 3.2):

1. Edges pointing from child nodes to parent nodes (reverse edges of original AST edges).

2. Edges pointing from preceding tokens to following tokens (NextToken edges [102]; only among leaf nodes).

3. Edges pointing from following tokens to preceding tokens (reverse NextToken edges).

3.4.3 Model Configuration

We used MLAP-Weighted architecture throughout the experiment, and evaluated both Linear and LSTM decoders. We set the maximum length of the decoded sequence (I or T) as 5.

We trained a vector embedding for node features as $\boldsymbol{h}_n^{(0)}$ and another vector embedding for edge types as $f_{\text{edge}}^{(l)}$ in Eq. (2.10).

As described in Section 2.4.3, we used GIN as the message passing layer. For each decoder type (Linear or LSTM), we selected the best model configuration among these conditions: number of layers (5 or 6), residual connection (+ or −), and GraphNorm [45] (+ or −). If a model has residual connection, the node presentation before message passing is added to the node representation after dropout, as in

$$\boldsymbol{h}_n^{(l)} = \text{DropOut}\left(\text{ReLU}\left(\text{GraphNorm}\left(\text{GIN}\left(\left\{\boldsymbol{h}_{n'}^{(l-1)}\middle| n' \in \mathcal{N}\right\}\right)\right)\right)\right) + \boldsymbol{h}_n^{(l-1)}.$$

We optimized the model using the Adam optimizer [54]. The models were trained for 50 epochs against a cross-entropy loss computed for each word in the output sequence. The initial learning rate was set to 5×10^{-4} and decayed by $\times 0.2$ after 3 epochs without improvement in the validation F1 score. The batch size was 256.

3.4.4 Performance Evaluation

As proposed by Alon et al. [97], we used an F1 score between predicted words and ground-truth words to evaluate the model performance. Given the unique and order-agnostic sets of predicted words S_p and ground-truth words S_t, we defined the metric as

$$\mathrm{TP} = |S_p \cap S_t|, \qquad \mathrm{FP} = |S_p \setminus S_t|, \qquad \mathrm{FN} = |S_t \setminus S_p|,$$
$$\mathrm{precision} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}, \quad \mathrm{recall} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}, \quad \mathrm{F1} = \frac{2 \times \mathrm{precision} \times \mathrm{recall}}{\mathrm{precision} + \mathrm{recall}}.$$

We calculated an F1 score for each graph and calculated average over the dataset. We trained 10 models with different random seeds, and evaluate the performance by the mean and the unbiased standard deviation.

As baseline models, we compared the performance of our model to the naive GIN architecture without MLAP, code2seq [98], and Graph Multi-head Attention Neural Network (GMAN) [107], which is the SoTA model in the OGB's leaderboard to date[7]. We trained the naive GIN and code2seq models ourselves because there are no available pre-trained models. Note that the definition of the F1 score used in original code2seq code is different from OGB (code2seq first sums up TP, FP, and FN over the dataset and then computes precision, accuracy, and F1), and we replaced the definition with that of OGB. Also, we trained the code2vec models for 100 epochs because of slow convergence. For GMAN, we referred to the performance on the OGB's leaderboard.

3.5 Results

Table 3.1 summarizes the model performance on the ogbg-code2 dataset. We selected the best models according to the validation performance. The MLAP-Weighted model with the Linear decoder achieved the test F1 score of 0.1792 ± 0.0016. It outperformed the current SoTA GNN model (GMAN; 0.1770 ± 0.0012) or non-GNN model (code2seq; 0.1549 ± 0.0009).

[7] `https://ogb.stanford.edu/docs/leader_graphprop/#ogbg-code2`, accessed November 21, 2021.

Decoder	Model	Configuration	#Params	Validation F1	Test F1
Linear	MLAP-Weighted	$L = 6$, Res (+), GN (−)	8.6M	0.1649 ± 0.0014	**0.1792** ± 0.0017
	naive	$L = 6$, Res (+), GN (−)	8.3M	0.1622 ± 0.0018	0.1768 ± 0.0019
LSTM	MLAP-Weighted	$L = 5$, Res (+), GN (+)	5.0M	0.1602 ± 0.0021	0.1762 ± 0.0037
	naive	$L = 6$, Res (+), GN (+)	4.6M	0.1596 ± 0.0024	0.1744 ± 0.0030
GMAN [107]			63.7M	0.1631 ± 0.0090	0.1770 ± 0.0012
code2seq [98]			26.7M	0.1495 ± 0.0010	0.1549 ± 0.0010

Table 3.1: Summary of the model performance (*mean* ± *unbiased standard deviation*) on ogbg-code2. Res (+/−): with or without residual connection; GN (+/−): with or without GraphNorm; #Params: number of trainable parameters.

Comparison	p	E.S.
MLAP-Weighted-Linear *vs.* naive-Linear	*0.007	1.37
MLAP-Weighted-Linear *vs.* GMAN [107]	*0.003	1.53
MLAP-Weighted-Linear *vs.* code2seq [98]	$*< 10^{-5}$	17.7

Table 3.2: The statistical analysis results. We compared the performance of MLAP-Weighted model with the Linear decoder to other models. p: p-value of the t-test. *: significant difference. E.S.: effect size (Cohen's d).

For the tested ogbg-code2 dataset, the LSTM decoder did not perform as good as the Linear decoder. Nevertheless, the MLAP-Weighted model outperformed the naive architecture.

We statistically compared the performance of the MLAP-Weighted model with the Linear decoder to other models (Table 3.2). As a result, it was revealed that our proposed model performed significantly better than the baseline models. Also, the effect size evaluated by Cohen's d [56] demonstrated the performance difference between our model and others were large.

3.6 Discussion

In this study, we extended the MLAP architecture for graph2seq tasks. Our proposed model outperformed the naive architecture and GMAN in the extreme source code summarization task and outperformed the current SoTA model. It indicates that explicitly using the compositional graph information is beneficial in learning to generate sequences from graphs. Here, we emphasize that our best

model using MLAP has only 8.6 million parameters, which is less than a seventh of the current SoTA model. It further supports our hypothesis that utilizing the compositionality helps the model to capture the structural information in graphs.

Code2seq, which is one of the best non-GNN program comprehension model to the best of our knowledge, performed worse than both our model and GMAN, even though code2seq is designed to exploit the domain knowledge in software engineering as much as possible. We could consider the "domain knowledge" used in code2seq or other ML-based program comprehension studies as a set of human experiences in which human programmers mentally construct and interpret the program graph like ASTs. Therefore, it may be enough to use GNNs, which are highly capable of capturing the graph structure, rather than building program comprehension-specific model full of domain knowledge.

Under the current experimental settings, the Linear decoder consistently performed better than the LSTM decoder. We can consider several hypotheses that explain the reason of the results. First, our hyperparameter tuning for the LSTM decoder might not be sufficient. As observed in Table 3.1, the number of parameters of the model using LSTM decoder is smaller than that with Linear decoder. This would limit the expressivity of the LSTM decoder, and we should be able to further improve the performance in this regard (e.g., adopting bidirectional LSTM decoder [108]). Second, the comparably complex LSTM decoder is harder to train with a medium-sized ogbg-code2 dataset. It only contains approximately 4.5×10^5 data points, with each ground-truth sequence having only 2.25 words, which may not be enough for LSTM to capture the context in the output sequences. By contrast, the LSTM decoder should be better than the Linear decoder for tasks that require the models to generate longer sequences.

To further improve the performance of MLAP-graph2seq models, seeking for various decoder implementation would be needed. For example, instead of unidirectional LSTM used in this study, we may use bi-directional LSTM as the decoder [108], which might be beneficial in capturing the context of the output sequences. Furthermore, using transformer-like architecture [63] can be another option.

3.6.1 Concluding Remarks

In this chapter, we propose a graph2seq model using the MLAP architecture. Utilizing the compositionality, our model outperformed the current SoTA model in an extreme source code summarization task with a greatly smaller number of parameters. We will further evaluate the capabilities of our model with wider variety of tasks, including natural language processing.

Chapter 4

Discussion and Perspectives

To conclude the dissertation, let us recall the fundamental questions we raised in Chapter 1:

- *Does a compositionality-aware GNN architecture improve the performances in graph machine learning tasks?*
- *Is it effective for GNNs to explicitly utilize the graph information from various levels of localities to learn more discriminative graph representations?*

The answer to the first question is "*yes*." In this study, we proposed a compositionality-aware GNN architecture, named MLAP, which aggregates graph representations from multiple levels of localities. We demonstrated the benefit from utilizing the compositionality through the experimental evaluation using graph classification tasks and a graph2seq task.

The answer to the second question is also "*yes*." We showed in Chapter 2 that there was information from different scale of localities, represented in different GNN layer, and aggregating them resulted in a more discriminative graph representations. Owing to the aggregation, MLAP can focus on any level in the information compositionality of graphs, from the local information around specific nodes to the global information as the entire graphs.

From the standpoint of parameter optimization, MLAP's skip connection from each layer to the aggregator would serve as a countermeasure to the gradient vanishing or explosion problem [109]. MLAP's skip connections, as well as those used in JK networks [14], send the gradient signal directly from the output layer

to each intermediate layer during back propagation. Therefore, we conjecture that such skip connections would stabilize the gradient signal to each layer and thus the learning process.

It is argued that sequentially stacking multiple attention layer (like transformers) would lose the local signals very rapidly [110]. Hence, existing transformer architectures avoid this problem by adding residual connections around the attention layer to retain local signal. By contrast, our MLAP architecture introduces the attention pooling for each layer *in parallel*. Therefore, the model could focus on node-wise signal in each layer without losing them through the attentional operations. This approach may benefit studies analyzing the behavior of transformer and improves the architecture.

Contrasting the MLAP architecture with the neural mechanism of biological cognitive systems offers an interesting insight. The multi-level attention mechanism introduced in the MLAP architecture can also be seen as an analogy of the attention mechanism in the cognitive system of humans or other primates. Biological cognitive systems, particularly the visual perception mechanism, are hierarchically organized and accompanied by hierarchical attention mechanisms. For example, the ventral visual pathway contributes to the hierarchical computation of object recognition mechanisms [111]. In the ventral visual pathway, the neural information in the area V1 represents the raw visual inputs, and the representations are hierarchically abstracted and composited toward the inferior temporal cortices as the receptive field (i.e., locality) of the information is expanded. DeWeerd et al. [112] found that lesions in the cortical areas V4 and TEO, both of which are components in the ventral pathway, contribute to the attentional processing in receptive fields with different sizes. Brain-inspired neural network architectures can improve the performance or the efficiency of the models, whereas the computational studies on neural networks may contribute to neuroscience research. Hence, neuroscience and artificial neural networks will continue to affect each other and develop in tandem.

4.1 Future Prospects

There are several interesting research directions to expand the proposed methods.

Exploring other aggregator functions than those proposed in this study, i.e., *Sum* and *Weighted*, is needed. For example, it is possible to design an aggregator that models the relationships among layer-wise representations, whereas the proposed aggregators treated the layer-wise representations as independent of each other. Furthermore, one can design an aggregator that only uses the representations in a subset of layers to reduce the computational cost, although the proposed aggregators required the layer-wise representations in all of the GNN layers.

Multi-stage training of the models with MLAP architecture would be beneficial. Rather than training the entire GNN models with MLAP at once as we did in this study, one can first train the GNN backbone without MLAP *and then* fine-tune the model with the MLAP. This kind of multi-stage training would stabilize the learning process, particularly when using the MLAP with an aggregator that has additional trainable parameters, such as the *MLAP-Weighted* architecture.

Our MLAP architecture can be applied to arbitrary deep learning models, not limited to GNNs. For example, CNNs for computer vision would be good candidates. Some CNN studies, such as U-Net [30], have already considered the hierarchy of the information processed in the neural networks. Adopting the hierarchical attention mechanism to such models may improve their performance.

Finally, we are interested in the potential of MLAP in unsupervised machine learning problems. Unsupervised representation learning based on contrastive learning has drawn significant attention [113], which aims to learn representations invariant across different views of an instance. Instead, Wang et al. [114] advocated to learn representations invariant across instances in a category. Owing to the information aggregation from multiple levels of localities, we expect that MLAP could extract representations associated to a certain pattern observed in a category. Thus, combining contrastive learning techniques with MLAP would help a model to improve the acquired representations.

4.2 Concluding Remarks

This dissertation proposed a compositionality-aware GRL technique using MLAP and demonstrated its benefits both in task performance and the acquired representation itself. Existing message-passing-based GNN methods effectively utilized the stationarity and locality in graphs, and this study improved the learned representations in GNN models by utilizing the compositionality—aggregating graph information from multiple levels of localities. Although the utilization of the compositionality in neural network studies is still in its infancy, we believe that analyzing compositionality is a key to building high-performance and interpretable machine learning models, both in GRL and other machine learning domains.

References

[1] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems 29*, pages 3837–3845, 2016.

[2] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst. Geometric Deep Learning: Going beyond Euclidean Data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017.

[3] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, 1999.

[4] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011.

[5] R. I. Kondor and J. Lafferty. Diffusion Kernels on Graphs and Other Discrete Structures. In *Proceedings of the 19th International Conference on Machine Learning*, pages 315–322, 2002.

[6] W. L. Hamilton, R. Ying, and J. Leskovec. Representation Learning on Graphs: Methods and Applications. *IEEE Data Engineering Bullettin*, 40 (3):52–74, 2017.

[7] M. Belkin and P. Niyogi. Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In *Advances in Neural Information Processing Systems 14*, pages 585–591, 2001.

[8] A. Ahmed, N. Shervashidze, S. M. Narayanamurthy, V. Josifovski, and A. J. Smola. Distributed Large-Scale Natural Graph Factorization. In *Proceedings of the 22nd International World Wide Web Conference*, pages 37–48, 2013.

[9] W. L. Hamilton. *Graph Representation Learning*. Morgan and Claypool, 2020.

[10] Z. Zhang, P. Cui, and W. Zhu. Deep Learning on Graphs: A Survey. *arXiv preprint*, arXiv:1812.04202, 2018.

[11] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[12] Q. Li, Z. Han, and X. Wu. Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pages 3538–3545, 2018.

[13] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated Graph Sequence Neural Networks. In *Proceedings of the 4th International Conference on Learning Representations*, 2016.

[14] K. Xu, C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka. Representation Learning on Graphs with Jumping Knowledge Networks. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80, pages 5453–5462, 2018.

[15] M. Gori, G. Monfardini, and F. Scarselli. A New Model for Learning in Graph Domains. In *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine*, 2005.

[16] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20 (1):61–80, 2009.

[17] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral Networks and Locally Connected Networks on Graphs. In *Proceedings of the 2nd International Conference on Learning Representations*, 2014.

[18] T. N. Kipf and M. Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*, 2017.

[19] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In *Advances in Neural Information Processing Systems 28*, pages 2224–2232, 2015.

[20] W. L. Hamilton, Z. Ying, and J. Leskovec. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30*, pages 1024–1034, 2017.

[21] M. Niepert, M. Ahmed, and K. Kutzkov. Learning Convolutional Neural Networks for Graphs. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 2014–2023, 2016.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph Attention Networks. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[23] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How Powerful are Graph Neural Networks? In *Proceedings of the 7th International Conference on Learning Representations*, 2019.

[24] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1263–1272, 2017.

[25] D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[26] O. Vinyals, S. Bengio, and M. Kudlur. Order Matters: Sequence to Sequence for Sets. In *Proceedings of the 4th International Conference on Learning Representations*, 2016.

[27] M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence*, pages 4438–4445, 2018.

[28] Z. Ying, J. You, C. Morris, R. Xiang, , W. L. Hamilton, and J. Leskovec. Hierarchical Graph Representation Learning with Differentiable Pooling. In *Advances in Neural Information Processing Systems 31*, pages 4805–4815, 2018.

[29] H. Gao and S. Ji. Graph U-Nets. In *Proceedings of the 36th International Conference on Machine Learning*, pages 2083–2092, 2019.

[30] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 234–241, 2015.

[31] J. Lee, I. Lee, and J. Kang. Self-Attention Graph Pooling. In *Proceedings of the 36th International Conference on Machine Learning*, pages 3734–3743, 2019.

[32] C. Cangea, P. Velickovic, N. Jovanovic, T. Kipf, and P. Liò. Towards Sparse Hierarchical Graph Classifiers. *arXiv preprint*, arXiv:1811.01287, 2018.

[33] K. Oono and T. Suzuki. Graph Neural Networks Exponentially Lose Expressive Power for Node Classification. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

[34] W. Huang, Y. Rong, T. Xu, F. Sun, and J. Huang. Tackling Over-Smoothing for General Graph Convolutional Networks. *arXiv preprint*, arXiv:2008.09864, 2020.

[35] Y. Min, F. Wenkel, and G. Wolf. Scattering GCN: Overcoming Oversmoothness in Graph Convolutional Networks. In *Advances in Neural Information Processing Systems 33*, pages 14498–14508, 2020.

[36] Y. Rong, W. Huang, T. Xu, and J. Huang. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

[37] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[38] G. Li, M. Müller, A. K. Thabet, and B. Ghanem. DeepGCNs: Can GCNs Go As Deep As CNNs? In *Proceedings of the 2019 IEEE/CVF International Conference on Computer Vision*, pages 9266–9275, 2019.

[39] J. Zhang and L. Meng. GResNet: Graph Residual Network for Reviving Deep GNNs from Suspended Animation. *arXiv preprint*, arXiv:1909.05729, 2019.

[40] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. Simple and Deep Graph Convolutional Networks. In *Proceedings of the 37th International Conference on Machine Learning*, pages 1725–1735, 2020.

[41] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 448–456, 2015.

[42] J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer Normalization. *arXiv preprint*, arXiv:1607.06450, 2016.

[43] L. Zhao and L. Akoglu. PairNorm: Tackling Oversmoothing in GNNs. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

[44] K. Zhou, Y. Dong, K. Wang, W. S. Lee, B. Hooi, H. Xu, and J. Feng. Understanding and Resolving Performance Degradation in Graph Convolutional Networks. *arXiv preprint*, arXiv:2006.07107, 2020.

[45] T. Cai, S. Luo, K. Xu, D. He, T. Liu, and L. Wang. GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training. *arXiv preprint*, arXiv:2009.03294, 2020.

[46] K. Zhou, X. Huang, Y. Li, D. Zha, R. Chen, and X. Hu. Towards Deeper Graph Neural Networks with Differentiable Group Normalization. In *Advances in Neural Information Processing Systems 33*, pages 4917–4928, 2020.

[47] X. Wang, X. He, Y. Cao, M. Liu, and T. Chua. KGAT: Knowledge Graph Attention Network for Recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 950–958, 2019.

[48] E. Ranjan, S. Sanyal, and P. P. Talukdar. ASAP: Adaptive Structure Aware Pooling for Learning Hierarchical Graph Representations. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, pages 5470–5477, 2020.

[49] N. Dehmamy, A. Barabási, and R. Yu. Understanding the Representation Power of Graph Neural Networks in Learning Graph Topology. In *Advances in Neural Information Processing Systems 32*, pages 15387–15397, 2019.

[50] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33*, pages 22118–22133, 2020.

[51] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann. TUDataset: A Collection of Benchmark Datasets for Learning with Graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond*, 2020.

[52] Z. Wu, B. Ramsundar, E. N. Feinberg, J. Gomes, C. Geniesse, A. S. Pappu, K. Leswing, and V. Pande. MoleculeNet: A Benchmark for Molecular Machine Learning. *Chemical Science*, 9(2):513–530, 2018.

[53] D. Szklarczyk, A. L. Gable, D. Lyon, A. Junge, S. Wyder, J. Huerta-Cepas, M. Simonovic, N. T. Doncheva, J. H. Morris, P. Bork, L. J. Jensen, and C. von Mering. STRING v11: Protein-Protein Association Networks with Increased Coverage, Supporting Functional Discovery in Genome-Wide Experimental Datasets. *Nucleic Acids Research*, 47(D1):D607–D613, 2018.

[54] D. P. Kingma and J. L. Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[55] L. van der Maaten and G. Hinton. Visualizing Data Using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605, 2008.

[56] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic press, 1988.

[57] J. S. Kim, K.-I. Goh, G. Salvi, E. Oh, B. Kahng, and D. Kim. Fractality in Complex Networks: Critical and Supercritical Skeletons. *Physical Review E*, 75(1):016110, 2007.

[58] K. Sakaguchi, Y. Arase, and M. Komachi. Discriminative Approach to Fill-in-the-Blank Quiz Generation for Language Learners. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 238–242, 2013.

[59] D. Seyler, M. Yahya, and K. Berberich. Knowledge Questions from Knowledge Graphs. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, pages 11–18, 2017.

[60] Y. Hu and S. X. Yang. A Knowledge Based Genetic Algorithm for Path Planning of a Mobile Robot. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, pages 4350–4355, 2004.

[61] K. Cho, B. van Merrienboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing*, pages 1724–1734, 2014.

[62] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112, 2014.

[63] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems 30*, pages 5998–6008, 2017.

[64] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint*, arXiv:1810.04805, 2018.

[65] P. Fernandes, M. Allamanis, and M. Brockschmidt. Structured Neural Summarization. In *Proceedings of the 7th International Conference on Learning Representations*, 2019.

[66] D. Marcheggiani and L. Perez-Beltrachini. Deep Graph Convolutional Encoders for Structured Data to Text Generation. In *Proceedings of the 11th International Conference on Natural Language Generation*, pages 1–9, 2018.

[67] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin. Graph2seq: Graph to Sequence Learning with Attention-Based Neural Networks. *arXiv preprint*, arXiv:1804.00823, 2018.

[68] Y. Chen, L. Wu, and M. J. Zaki. Toward Subgraph Guided Knowledge Graph Question Generation with Graph Neural Networks. *arXiv preprint*, arXiv:2004.06015, 2020.

[69] P. Wei, J. Zhao, and W. Mao. A Graph-to-Sequence Learning Framework for Summarizing Opinionated Texts. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:1650–1660, 2021.

[70] J. Zhu, J. Li, M. Zhu, L. Qian, M. Zhang, and G. Zhou. Modeling Graph Structure in Transformer for Better AMR-to-Text Generation. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, pages 5459–5468, 2019.

[71] D. Cai and W. Lam. Graph Transformer for Graph-to-Sequence Learning. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence*, pages 7464–7471, 2020.

[72] M. Allamanis, H. Peng, and C. Sutton. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 2091–2100, 2016.

[73] V. D'silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[74] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and P. Schnoebelen. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Science & Business Media, 2013.

[75] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*, pages 837–847, 2012.

[76] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton. A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, 51(4):81:1–81:37, 2018.

[77] D. Movshovitz-Attias and W. Cohen. Natural Language Models for Predicting Programming Comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 35–40, 2013.

[78] C. S. Corley, K. Damevski, and N. A. Kraft. Exploring the Use of Deep Learning for Feature Location. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, pages 556–560, 2015.

[79] R. Bavishi, M. Pradel, and K. Sen. Context2Name: A Deep Learning-Based Approach to Infer Natural Variable Names from Usage Contexts. *arXiv preprint*, arXiv:1809.05193, 2018.

[80] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting Accurate Method and Class Names. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 38–49, 2015.

[81] H. K. Dam, T. Tran, and T. Pham. A Deep Language Model for Software Code. *arXiv preprint*, arXiv:1608.02715, 2016.

[82] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 2073–2083, 2016.

[83] S. Jiang, A. Armaly, and C. McMillan. Automatically Generating Commit Messages from Diffs Using Neural Machine Translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 135–146, 2017.

[84] S. Xu, S. Zhang, W. Wang, X. Cao, C. Guo, and J. Xu. Method Name Suggestion with Hierarchical Attention Networks. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 10–21, 2019.

[85] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving Language Understanding by Generative Pre-Training. Technical report, OpenAI, 2018.

[86] Z. Yang, Z. Dai, Y. Yang, J. G. Carbonell, R. Salakhutdinov, and Q. V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems 32*, pages 5754–5764, 2019.

[87] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. Pre-trained Contextual Embedding of Source Code. *arXiv preprint*, arXiv:2001.00059, 2020.

[88] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of the 2020 Conference*

on Empirical Methods in Natural Language Processing, pages 1536–1547, 2020.

[89] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan. IntelliCode Compose: Code Generation Using Transformer. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1433–1443, 2020.

[90] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language Models Are Unsupervised Multitask Learners. Technical report, OpenAI, 2019.

[91] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample. Unsupervised Translation of Programming Languages. In *Advances in Neural Information Processing Systems 33*, pages 20601–20611, 2020.

[92] C. Omar. Structured Statistical Syntax Tree Prediction. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, pages 113–114, 2013.

[93] C. Maddison and D. Tarlow. Structured Generative Models of Natural Source Code. In *Proceedings of the 31st International Conference on Machine Learning*, pages 649–657, 2014.

[94] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning Natural Coding Conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[95] D. DeFreez, A. V. Thakur, and C. Rubio-González. Path-Based Function Embedding and Its Application to Error-Handling Specification Mining. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 423–433, 2018.

[96] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations*, 2013.

[97] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40:1–40:29, 2019.

[98] U. Alon, O. Levy, and E. Yahav. code2seq: Generating Sequences from Structured Representations of Code. In *Proceedings of the 7th International Conference on Learning Representations*, 2019.

[99] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building Program Vector Representations for Deep Learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management*, pages 547–553, 2015.

[100] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, pages 1287–1293, 2016.

[101] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering*, pages 783–794, 2019.

[102] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to Represent Programs with Graphs. In *Proceedings of the 6th International Conference on Learning Representations*, 2018.

[103] M. Cvitkovic, B. Singh, and A. Anandkumar. Open Vocabulary Learning on Source Code with a Graph-Structured Cache. In *Proceedings of the 36th International Conference on Machine Learning*, pages 1475–1485, 2019.

[104] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu. Retrieval-Augmented Generation for Code Summarization via Hybrid GNN. In *Proceedings of the 9th International Conference on Learning Representations*, 2021.

[105] T. Luong, H. Pham, and C. D. Manning. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference*

on Empirical Methods in Natural Language Processing, pages 1412–1421, 2015.

[106] H. Sak, A. W. Senior, and F. Beaufays. Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition. *arXiv preprint*, arXiv:1402.1128, 2014.

[107] H. Zhang, J. Gu, and P. Shen. GMAN and Bag of Tricks for Graph Classification. `https://github.com/PierreHao/YouGraph/blob/ae8cf5d5bb544f64ee206bcba07ece66d49e00e3/report/GMAN and bag of tricks for graph classification.pdf`, 2021.

[108] K. Al-Sabahi, Z. Zuping, and Y. Kang. Bidirectional Attentional Encoder-Decoder Model and Bidirectional Beam Search for Abstractive Summarization. *arXiv preprint*, arXiv:1809.06662, 2018.

[109] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient Flow in Recurrent Nets: The Difficulty of Learning Long-Term Dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*, pages 237–243. IEEE Press, 2001.

[110] Y. Dong, J. Cordonnier, and A. Loukas. Attention Is Not All You Need: Pure Attention Loses Rank Doubly Exponentially with Depth. In *Proceedings of the 38th International Conference on Machine Learning*, pages 2793–2803, 2021.

[111] D. J. Kravitz, K. S. Saleem, C. I. Baker, L. G. Ungerleider, and M. Mishkin. The Ventral Visual Pathway: An Expanded Neural Framework for the Processing of Object Quality. *Trends in Cognitive Sciences*, 17(1):26–49, 2013.

[112] P. DeWeerd, M. R. Peralta, R. Desimone, and L. G. Ungerleider. Loss of Attentional Stimulus Selection after Extrastriate Cortical Lesions in Macaques. *Nature Neuroscience*, 2(8):753–758, 1999.

[113] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin. Unsupervised Feature Learning via Non-Parametric Instance Discrimination. In *Proceedings of the 2018 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3733–3742, 2018.

[114] F. Wang, H. Liu, D. Guo, and F. Sun. Unsupervised Representation Learning by Invariance Propagation. In *Advances in Neural Information Processing Systems 33*, 2020.

Appendix A

Additional Data and Discussion for Chapter 2

A.1 Full Validation Performances of the Trained Models

In Figures 2.3–2.6, we only plotted a part of validation performances for legibility. Here, we provide a summary of the validation performances in Table A.1 and the full validation performances in Tables A.2–A.5.

GN	Architecture	Synthetic (L)	ogbg-molhiv (L)	ogbg-ppa (L)	MCF-7 (L)
(−)	naive	0.5116 ± 0.0154 (4)	0.8090 ± 0.0014 (4)	0.6676 ± 0.0015 (1)	0.8023 ± 0.0011 (2)
	JK-Sum	0.2347 ± 0.0082 (4)	0.7926 ± 0.0021 (4)	0.6681 ± 0.0018 (1)	0.7924 ± 0.0007 (2)
	JK-Concat.	0.2357 ± 0.0091 (10)	0.7786 ± 0.0019 (4)	0.6666 ± 0.0024 (1)	0.7893 ± 0.0009 (8)
	JK-MaxPool	0.2779 ± 0.0183 (4)	0.7791 ± 0.0019 (2)	0.6668 ± 0.0016 (1)	0.7901 ± 0.0016 (3)
	JK-LSTM-Att.	0.4109 ± 0.0215 (7)	0.7799 ± 0.0016 (8)	0.6667 ± 0.0015 (1)	0.7862 ± 0.0013 (2)
	MLAP-Sum	*0.1930 ± 0.0093 (10)	0.8096 ± 0.0020 (3)	*0.6691 ± 0.0050 (3)	*0.8269 ± 0.0018 (4)
	MLAP-Weighted	0.2836 ± 0.0174 (6)	*0.8129 ± 0.0017 (5)	0.6687 ± 0.0013 (1)	0.8081 ± 0.0013 (3)
(+)	naive	0.0086 ± 0.0003 (9)	0.8156 ± 0.0022 (2)	0.6772 ± 0.0017 (1)	0.8616 ± 0.0014 (3)
	JK-Sum	0.0096 ± 0.0004 (9)	*0.8241 ± 0.0022 (2)	0.6797 ± 0.0024 (1)	0.8692 ± 0.0012 (7)
	JK-Concat.	0.0094 ± 0.0005 (7)	0.8241 ± 0.0026 (2)	0.6700 ± 0.0026 (1)	0.8686 ± 0.0012 (5)
	JK-MaxPool	0.0089 ± 0.0004 (9)	0.8206 ± 0.0034 (3)	0.6760 ± 0.0021 (1)	0.8665 ± 0.0009 (5)
	JK-LSTM-Att.	0.0086 ± 0.0004 (10)	0.8115 ± 0.0018 (2)	*0.6815 ± 0.0015 (1)	0.8616 ± 0.0013 (3)
	MLAP-Sum	*0.0075 ± 0.0004 (9)	0.8221 ± 0.0023 (6)	0.6783 ± 0.0015 (1)	*0.8720 ± 0.0011 (5)
	MLAP-Weighted	0.0100 ± 0.0004 (9)	0.8115 ± 0.0035 (2)	0.6776 ± 0.0018 (1)	0.8634 ± 0.0013 (4)

Table A.1: The summary of the validation performances for model selection. Each cell shows the best performance of an architecture for a dataset in *mean* ± *standard error*. The numbers in the parentheses are the number of layers of the best performing models. We used these validation performances for model selection. GN: GraphNorm; JK-Concat.: JK-Concatenation; JK-LSTM-Att.: JK-LSTM-Attention.

GN	Architecture	Number of Layers				
		1	2	3	4	5
		6	7	8	9	10
(−)	naive	0.7426 ± 0.0008	0.5654 ± 0.0026	0.5382 ± 0.0173	**0.5116** ± 0.0154	0.5186 ± 0.0164
		0.5300 ± 0.015	0.5213 ± 0.0167	0.5346 ± 0.0112	0.5342 ± 0.0140	0.5423 ± 0.0137
	JK-Sum	0.7418 ± 0.0007	0.5074 ± 0.0077	0.2521 ± 0.0083	**0.2347** ± 0.0082	0.2838 ± 0.0184
		0.3278 ± 0.0183	0.4009 ± 0.0111	0.3947 ± 0.0098	0.3864 ± 0.0107	0.3769 ± 0.0084
	JK-Concatenation	0.7233 ± 0.0081	0.5447 ± 0.0092	0.3924 ± 0.0230	0.3329 ± 0.0192	0.3119 ± 0.0216
		0.2936 ± 0.0217	0.3123 ± 0.0285	0.2440 ± 0.0172	0.2436 ± 0.0149	**0.2357** ± 0.0091
	JK-MaxPool	0.7384 ± 0.0042	0.5314 ± 0.0025	0.3304 ± 0.0074	**0.2779** ± 0.0183	0.3240 ± 0.0177
		0.3441 ± 0.0175	0.3678 ± 0.0172	0.3969 ± 0.0132	0.4163 ± 0.0112	0.4229 ± 0.0116
	JK-LSTM-Attention	0.7418 ± 0.0008	0.5869 ± 0.0119	0.4654 ± 0.0202	0.4601 ± 0.0198	0.4418 ± 0.0169
		0.4199 ± 0.0236	**0.4109** ± 0.0215	0.4204 ± 0.0190	0.4126 ± 0.0185	0.4530 ± 0.0148
	MLAP-Sum	0.7423 ± 0.0009	0.5391 ± 0.0055	0.4564 ± 0.0150	0.3453 ± 0.0173	0.2906 ± 0.0145
		0.2250 ± 0.0107	0.2240 ± 0.0110	0.1953 ± 0.0093	0.2016 ± 0.0098	**0.1930** ± 0.0093
	MLAP-Weighted	0.7431 ± 0.0007	0.5539 ± 0.0088	0.4821 ± 0.0181	0.4112 ± 0.0198	0.3536 ± 0.0197
		0.2836 ± 0.0174	0.3029 ± 0.0189	0.2896 ± 0.0143	0.2908 ± 0.0142	0.2861 ± 0.0170
(+)	naive	0.5846 ± 0.0017	0.2763 ± 0.0092	0.0597 ± 0.0023	0.0237 ± 0.0008	0.0148 ± 0.0004
		0.0106 ± 0.0005	0.0093 ± 0.0004	0.0094 ± 0.0005	**0.0086** ± 0.0003	0.0087 ± 0.0005
	JK-Sum	0.5886 ± 0.0019	0.2549 ± 0.0054	0.0576 ± 0.0014	0.0236 ± 0.0007	0.0141 ± 0.0004
		0.0106 ± 0.0004	0.0109 ± 0.0004	0.0099 ± 0.0004	**0.0096** ± 0.0004	0.0097 ± 0.0004
	JK-Concatenation	0.5880 ± 0.0032	0.2634 ± 0.0068	0.0554 ± 0.0011	0.0225 ± 0.0006	0.0172 ± 0.0005
		0.0114 ± 0.0004	**0.0094** ± 0.0005	0.0097 ± 0.0004	0.0100 ± 0.0005	0.0106 ± 0.0004
	JK-MaxPool	0.5880 ± 0.0020	0.3203 ± 0.0025	0.0760 ± 0.0011	0.0250 ± 0.0007	0.0146 ± 0.0005
		0.0110 ± 0.0005	0.0090 ± 0.0004	0.0091 ± 0.0004	**0.0089** ± 0.0004	0.0090 ± 0.0004
	JK-LSTM-Attention	0.5874 ± 0.0019	0.2693 ± 0.0064	0.0615 ± 0.0021	0.0251 ± 0.0010	0.0144 ± 0.0004
		0.0100 ± 0.0005	0.0097 ± 0.0004	0.0090 ± 0.0004	0.0087 ± 0.0005	**0.0086** ± 0.0004
	MLAP-Sum	0.5878 ± 0.0019	0.2669 ± 0.0075	0.0574 ± 0.0018	0.0220 ± 0.0011	0.0153 ± 0.0006
		0.0101 ± 0.0003	0.0091 ± 0.0005	0.0086 ± 0.0004	***0.0075** ± 0.0004	0.0085 ± 0.0004
	MLAP-Weighted	0.5868 ± 0.0018	0.2840 ± 0.0090	0.0696 ± 0.0021	0.0284 ± 0.0006	0.0161 ± 0.0006
		0.0126 ± 0.0005	0.0103 ± 0.0004	0.0104 ± 0.0004	**0.0100** ± 0.0004	0.0100 ± 0.0005

Table A.2: Full validation performances in the synthetic dataset experiments for model selection. **bold**: best performance in an (aggregator, GraphNorm [+/−]) combination; *: the overall best performance.

GN	Architecture	Number of Layers				
		1	2	3	4	5
		6	7	8	9	10
(−)	naive	0.7467 ± 0.0020	0.7731 ± 0.0028	0.8014 ± 0.0014	**0.8090** ± 0.0014	0.8052 ± 0.0013
		0.7909 ± 0.0013	0.7820 ± 0.0013	0.7664 ± 0.0041	0.7599 ± 0.0042	0.7631 ± 0.0022
	JK-Sum	0.7446 ± 0.0022	0.7735 ± 0.0016	0.7885 ± 0.0016	**0.7926** ± 0.0021	0.7775 ± 0.0020
		0.7726 ± 0.0018	0.7753 ± 0.0021	0.7738 ± 0.0017	0.7706 ± 0.0018	0.7720 ± 0.0022
	JK-Concatenation	0.7538 ± 0.0025	0.7677 ± 0.0028	0.7772 ± 0.0019	**0.7786** ± 0.0019	0.7713 ± 0.0021
		0.7732 ± 0.0018	0.7731 ± 0.0019	0.7746 ± 0.0018	0.7693 ± 0.0024	0.7681 ± 0.0023
	JK-MaxPool	0.7436 ± 0.0018	**0.7791** ± 0.0019	0.7723 ± 0.0017	0.7753 ± 0.0030	0.7689 ± 0.0020
		0.7667 ± 0.0025	0.7699 ± 0.0020	0.7655 ± 0.0016	0.7661 ± 0.0021	0.7667 ± 0.0021
	JK-LSTM-Attention	0.7464 ± 0.0022	0.7619 ± 0.0022	0.7661 ± 0.0013	0.7711 ± 0.0017	0.7689 ± 0.0018
		0.7741 ± 0.0017	0.7735 ± 0.0015	**0.7799** ± 0.0016	0.7788 ± 0.0024	0.7738 ± 0.0025
	MLAP-Sum	0.7452 ± 0.0023	0.7787 ± 0.0025	**0.8096** ± 0.0020	0.8002 ± 0.0029	0.8003 ± 0.0024
		0.8019 ± 0.0023	0.8007 ± 0.0025	0.8015 ± 0.0024	0.7971 ± 0.0018	0.8029 ± 0.0025
	MLAP-Weighted	0.7457 ± 0.0024	0.7702 ± 0.0023	0.7987 ± 0.0019	0.8107 ± 0.0020	**0.8129** ± 0.0017
		0.8076 ± 0.0024	0.8019 ± 0.0023	0.7977 ± 0.0026	0.8040 ± 0.0024	0.8027 ± 0.0020
(+)	naive	0.8055 ± 0.0038	**0.8159** ± 0.0022	0.8157 ± 0.0022	0.7958 ± 0.0031	0.7868 ± 0.0038
		0.7852 ± 0.0035	0.7733 ± 0.0037	0.7736 ± 0.0035	0.7634 ± 0.0042	0.7638 ± 0.0042
	JK-Sum	0.8041 ± 0.0026	***0.8241** ± 0.0022	0.8198 ± 0.0024	0.8157 ± 0.0023	0.8144 ± 0.0022
		0.8120 ± 0.0026	0.8096 ± 0.0027	0.8028 ± 0.0026	0.8088 ± 0.0022	0.7994 ± 0.0023
	JK-Concatenation	0.8042 ± 0.0028	**0.8241** ± 0.0026	0.8201 ± 0.0023	0.8212 ± 0.0028	0.8217 ± 0.0025
		0.8135 ± 0.0026	0.8095 ± 0.0031	0.8115 ± 0.0032	0.8099 ± 0.0027	0.8107 ± 0.0031
	JK-MaxPool	0.8039 ± 0.0032	0.8144 ± 0.0030	**0.8206** ± 0.0034	0.8155 ± 0.0030	0.8100 ± 0.0032
		0.8068 ± 0.0027	0.8024 ± 0.0030	0.7979 ± 0.0030	0.7955 ± 0.0031	0.7954 ± 0.0031
	JK-LSTM-Attention	0.8037 ± 0.0030	**0.8115** ± 0.0018	0.8058 ± 0.0028	0.8043 ± 0.0034	0.8045 ± 0.0032
		0.8082 ± 0.0027	0.8098 ± 0.0028	0.7874 ± 0.0040	0.7924 ± 0.0039	0.7791 ± 0.0033
	MLAP-Sum	0.8070 ± 0.0035	0.8121 ± 0.0031	0.8149 ± 0.0027	0.8204 ± 0.0028	0.8190 ± 0.0024
		0.8221 ± 0.0023	0.8178 ± 0.0037	0.8166 ± 0.0019	0.8140 ± 0.0023	0.8082 ± 0.0028
	MLAP-Weighted	0.8027 ± 0.0031	**0.8115** ± 0.0035	0.8065 ± 0.0029	0.8088 ± 0.0025	0.8045 ± 0.0029
		0.8024 ± 0.0037	0.8015 ± 0.0020	0.8036 ± 0.0028	0.7947 ± 0.0037	0.7951 ± 0.0034

Table A.3: Full validation performances in the ogbg-molhiv experiments for model selection. **bold**: best performance in an (aggregator, GraphNorm [+/−]) combination; *: the overall best performance.

GN	Architecture	Number of Layers				
		1	2	3	4	5
		6	7	8	9	10
(−)	naive	**0.6676** ± 0.0015	0.6639 ± 0.0034	0.6622 ± 0.0048	0.6560 ± 0.0037	0.6395 ± 0.0029
		0.6153 ± 0.0032	0.5936 ± 0.0054	0.5746 ± 0.0060	0.5442 ± 0.0099	0.4973 ± 0.0080
	JK-Sum	**0.6681** ± 0.0018	0.6610 ± 0.0042	0.6652 ± 0.0060	0.6569 ± 0.0052	0.6440 ± 0.0044
		0.6229 ± 0.0059	0.6206 ± 0.0051	0.6123 ± 0.0052	0.5913 ± 0.0050	0.5622 ± 0.0090
	JK-Concatenation	**0.6666** ± 0.0024	0.6352 ± 0.0041	0.6636 ± 0.0100	0.6420 ± 0.0057	0.6467 ± 0.0066
		0.6257 ± 0.0053	0.6219 ± 0.0096	0.6152 ± 0.0074	0.6011 ± 0.0064	0.5876 ± 0.0070
	JK-MaxPool	**0.6668** ± 0.0016	0.6076 ± 0.0031	0.6049 ± 0.0076	0.6070 ± 0.0088	0.5962 ± 0.0072
		0.6041 ± 0.0057	0.6023 ± 0.0061	0.5838 ± 0.0055	0.5837 ± 0.0075	0.5292 ± 0.0104
	JK-LSTM-Attention	**0.6667** ± 0.0015	0.6127 ± 0.0019	0.5910 ± 0.0069	0.5888 ± 0.0048	0.5542 ± 0.0116
		0.5020 ± 0.0333	0.4251 ± 0.0452	0.4237 ± 0.0479	0.4499 ± 0.0265	0.3046 ± 0.0305
	MLAP-Sum	0.6665 ± 0.0014	0.6568 ± 0.0042	**0.6691** ± 0.0050	0.6512 ± 0.0059	0.6529 ± 0.0052
		0.6246 ± 0.0069	0.6144 ± 0.0068	0.5959 ± 0.0071	0.5732 ± 0.0106	0.5721 ± 0.0074
	MLAP-Weighted	**0.6687** ± 0.0013	0.6463 ± 0.0034	0.6542 ± 0.0078	0.6358 ± 0.0038	0.6305 ± 0.0047
		0.6192 ± 0.0046	0.6023 ± 0.0079	0.5642 ± 0.0065	0.5583 ± 0.0096	0.5053 ± 0.0155
(+)	naive	**0.6772** ± 0.0017	0.6388 ± 0.0031	0.6263 ± 0.0028	0.6379 ± 0.0022	0.6448 ± 0.0023
		0.6403 ± 0.0023	0.6418 ± 0.0025	0.6427 ± 0.0017	0.6402 ± 0.0025	0.6441 ± 0.0027
	JK-Sum	**0.6797** ± 0.0024	0.6408 ± 0.0025	0.6370 ± 0.0034	0.6414 ± 0.0046	0.6415 ± 0.0045
		0.6291 ± 0.0020	0.6315 ± 0.0039	0.6254 ± 0.0027	0.6295 ± 0.0026	0.6274 ± 0.0032
	JK-Concatenation	**0.6700** ± 0.0026	0.6496 ± 0.0039	0.6537 ± 0.0032	0.6517 ± 0.0040	0.6491 ± 0.0025
		0.6516 ± 0.0036	0.6466 ± 0.0035	0.6465 ± 0.0034	0.6479 ± 0.0032	0.6429 ± 0.0030
	JK-MaxPool	**0.6760** ± 0.0021	0.6475 ± 0.0021	0.6415 ± 0.0029	0.6446 ± 0.0034	0.6349 ± 0.0015
		0.6362 ± 0.0040	0.6313 ± 0.0030	0.6316 ± 0.0039	0.6382 ± 0.0055	0.6353 ± 0.0037
	JK-LSTM-Attention	***0.6815** ± 0.0015	0.6522 ± 0.0049	0.6634 ± 0.0024	0.6630 ± 0.0016	0.6630 ± 0.0035
		0.6629 ± 0.0020	0.6658 ± 0.0024	0.6627 ± 0.0021	0.6647 ± 0.0027	0.6642 ± 0.0025
	MLAP-Sum	**0.6783** ± 0.0015	0.6373 ± 0.0031	0.6362 ± 0.0030	0.6370 ± 0.0022	0.6353 ± 0.0030
		0.6377 ± 0.0042	0.6326 ± 0.0026	0.6336 ± 0.0026	0.6359 ± 0.0025	0.6377 ± 0.0036
	MLAP-Weighted	**0.6776** ± 0.0018	0.6327 ± 0.0029	0.6538 ± 0.0017	0.6550 ± 0.0045	0.6475 ± 0.0036
		0.6561 ± 0.0048	0.6598 ± 0.0048	0.6535 ± 0.0077	0.6443 ± 0.0055	0.6503 ± 0.0045

Table A.4: Full validation performances in the ogbg-ppa experiments for model selection. **bold**: best performance in an (aggregator, GraphNorm [+/−]) combination; *: the overall best performance.

GN	Architecture	Number of Layers				
		1	2	3	4	5
		6	7	8	9	10
(−)	naive	0.7653 ± 0.0010	**0.8023** ± 0.0011	0.7960 ± 0.0024	0.7996 ± 0.0011	0.7968 ± 0.0012
		0.7689 ± 0.0033	0.7361 ± 0.0045	0.7101 ± 0.0045	0.7086 ± 0.0035	0.7097 ± 0.0049
	JK-Sum	0.7641 ± 0.0011	**0.7924** ± 0.0007	0.7836 ± 0.0015	0.7849 ± 0.0007	0.7884 ± 0.0013
		0.7866 ± 0.0017	0.7853 ± 0.0013	0.7846 ± 0.0010	0.7859 ± 0.0016	0.7827 ± 0.0011
	JK-Concatenation	0.7648 ± 0.0010	0.7868 ± 0.0008	0.7797 ± 0.0013	0.7857 ± 0.0007	0.7862 ± 0.0011
		0.7884 ± 0.0016	0.7875 ± 0.0011	**0.7893** ± 0.0009	0.7890 ± 0.0010	0.7881 ± 0.0011
	JK-MaxPool	0.7639 ± 0.0011	0.7860 ± 0.0009	**0.7901** ± 0.0016	0.7811 ± 0.0007	0.7825 ± 0.0010
		0.7814 ± 0.0009	0.7802 ± 0.0009	0.7821 ± 0.0012	0.7808 ± 0.0010	0.7798 ± 0.0011
	JK-LSTM-Attention	0.7651 ± 0.0013	**0.7862** ± 0.0013	0.7794 ± 0.0009	0.7780 ± 0.0012	0.7770 ± 0.0014
		0.7814 ± 0.0014	0.7787 ± 0.0010	0.7784 ± 0.0011	0.7775 ± 0.0015	0.7808 ± 0.0016
	MLAP-Sum	0.7642 ± 0.0011	0.7790 ± 0.0011	0.8138 ± 0.0010	**0.8269** ± 0.0018	0.8243 ± 0.0014
		0.8216 ± 0.0014	0.8202 ± 0.0015	0.8200 ± 0.0012	0.8228 ± 0.0012	0.8223 ± 0.0012
	MLAP-Weighted	0.7640 ± 0.0010	0.7738 ± 0.0012	**0.8081** ± 0.0013	0.8002 ± 0.0012	0.8010 ± 0.0013
		0.8029 ± 0.0015	0.7996 ± 0.0022	0.7985 ± 0.0016	0.8019 ± 0.0016	0.7996 ± 0.0018
(+)	naive	0.8271 ± 0.0015	0.8536 ± 0.0013	**0.8616** ± 0.0014	0.8613 ± 0.0016	0.8574 ± 0.0020
		0.8510 ± 0.0024	0.8438 ± 0.0021	0.8308 ± 0.0024	0.8265 ± 0.0027	0.8108 ± 0.0019
	JK-Sum	0.8259 ± 0.0013	0.8564 ± 0.0010	0.8635 ± 0.0011	0.8650 ± 0.0012	0.8671 ± 0.0010
		0.8664 ± 0.0015	**0.8692** ± 0.0012	0.8611 ± 0.0022	0.8623 ± 0.0020	0.8611 ± 0.0025
	JK-Concatenation	0.8301 ± 0.0016	0.8598 ± 0.0011	0.8664 ± 0.0011	0.8684 ± 0.0011	**0.8686** ± 0.0012
		0.8653 ± 0.0010	0.8613 ± 0.0010	0.8606 ± 0.0015	0.8563 ± 0.0013	0.8518 ± 0.0013
	JK-MaxPool	0.8267 ± 0.0010	0.8558 ± 0.0012	0.8633 ± 0.0013	0.8622 ± 0.0013	**0.8665** ± 0.0009
		0.8629 ± 0.0014	0.8619 ± 0.0012	0.8583 ± 0.0019	0.8541 ± 0.0016	0.8484 ± 0.0019
	JK-LSTM-Attention	0.8257 ± 0.0015	0.8552 ± 0.0010	**0.8616** ± 0.0013	0.8578 ± 0.0014	0.8588 ± 0.0015
		0.8572 ± 0.0019	0.8519 ± 0.0024	0.8469 ± 0.0027	0.8375 ± 0.0027	0.8369 ± 0.0024
	MLAP-Sum	0.8267 ± 0.0012	0.8583 ± 0.0010	0.8653 ± 0.0010	0.8686 ± 0.0011	***0.8720** ± 0.0011
		0.8695 ± 0.0011	0.8680 ± 0.0013	0.8668 ± 0.0012	0.8630 ± 0.0021	0.8634 ± 0.0014
	MLAP-Weighted	0.8270 ± 0.0011	0.8555 ± 0.0013	0.8626 ± 0.0011	**0.8634** ± 0.0013	0.8605 ± 0.0008
		0.8552 ± 0.0013	0.8559 ± 0.0010	0.8497 ± 0.0015	0.8413 ± 0.0018	0.8357 ± 0.0017

Table A.5: Full results in the MCF-7 dataset experiments for model selection. **bold**: best performance in an (aggregator, GraphNorm [+/−]) combination; *: the overall best performance.

A.2 MLAP encompasses JK given same linear aggregator

If both an MLAP model and a JK model have linear aggregators of the same form, we can prove that the MLAP model encompasses the JK model. Here, we consider the relationship between *MLAP-Sum* and *JK-Sum* as an example.

From Eq. (2.19), a *JK-Sum* model computes the graph representation as

$$\boldsymbol{h}_G = \sum_{n \in \mathcal{N}} \mathrm{softmax}\left(f_{\mathrm{gate}}(\boldsymbol{h}_n^{(\mathrm{JK})})\right) \boldsymbol{h}_n^{(\mathrm{JK})}. \tag{A.1}$$

Let $a_n = \mathrm{softmax}(f_{\mathrm{gate}}(\boldsymbol{h}_n^{(\mathrm{JK})}))$ be the attention value for each node n. Then, from Eqs. (2.17) and (2.18),

$$\boldsymbol{h}_G = \sum_{n \in \mathcal{N}} a_n \sum_{l=1}^{L} \boldsymbol{h}_n^{(l)} = \sum_{l=1}^{L} \sum_{n \in \mathcal{N}} a_n \boldsymbol{h}_n^{(l)}. \tag{A.2}$$

By contrast, from Eqs. (2.5) and (2.8), *MLAP-Sum* computes the graph representation as

$$\boldsymbol{h}_G = \sum_{l=1}^{L} \sum_{n \in \mathcal{N}} \mathrm{softmax}\left(f_{\mathrm{gate}}^{(l)}(\boldsymbol{h}_n^{(l)})\right) \boldsymbol{h}_n^{(l)}. \tag{A.3}$$

Therefore, when $\mathrm{softmax}(f_{\mathrm{gate}}^{(l)}(\boldsymbol{h}_n^{(l)}))$ for each l has exactly the same value as a_n, Eq. (A.3) has the same form as Eq. (A.2). This indicates that a *MLAP-Sum* model has greater expressivity than a *JK-Sum* model because $\mathrm{softmax}(f_{\mathrm{gate}}^{(l)}(\boldsymbol{h}_n^{(l)}))$ can take different values for each l.

We can prove that MLAP encompasses JK using other linear aggregators, e.g., *Weighted* or *Concatenation*[8]. Furthermore, we cannot directly apply the same discussion for nonlinear aggregators like LSTM-Attention. Nonetheless, our experiments showed that MLAP models with linear aggregators tended to perform better than JK models with nonlinear aggregators.

[8]We have not evaluated JK-Weighted or MLAP-Concatenation in this study.

A.3 Why does MLAP-Weighted perform worse than MLAP-Sum in some datasets?

In the synthetic dataset and ogbg-ppa, the *MLAP-Weighted* architecture performed worse than *MLAP-Sum*. However, intuitively, establishing balance across layers using the weight parameters sounds reasonable and effective. In this appendix section, we show the results of preliminary analyses on the cause of this phenomenon.

Figure A.1 shows the weight values in the trained 10-layer models with 30 different random seeds for the synthetic dataset, and Figure A.2 shows the weights in 5-layer ogbg-molhiv models with 30 seeds. The weight values for the synthetic dataset, where MLAP-Weighted was inferior to MLAP-Sum, had big variances, and the weight distribution covered the "constant weight" line (dashed horizontal line; an MLAP-Weighted model is virtually equivalent to an MLAP-Sum model if the weight parameters are equal to this value). It is expected that the desirable weight for each layer is not largely different from the constant weight. Therefore, the weight parameters did not modify the model output effectively. However, having such weight parameters with big variance indicates instability during the model training.

By contrast, the weight values for ogbg-molhiv, where MLAP-Weighted performed better than MLAP-Sum, had smaller variances, and the distribution deviated from the constant weight line, particularly in Layers 1 and 5. It is expected that the desirable weight for those layers differed from the constant weight, and the model might adapt to the balance across layers.

This preliminary analyses suggest that, depending on some properties of the datasets, the *MLAP-Weighted* architecture can excel *MLAP-Sum*. We will continue working on the analyses to identify the suitability of each MLAP aggregator to a certain dataset.

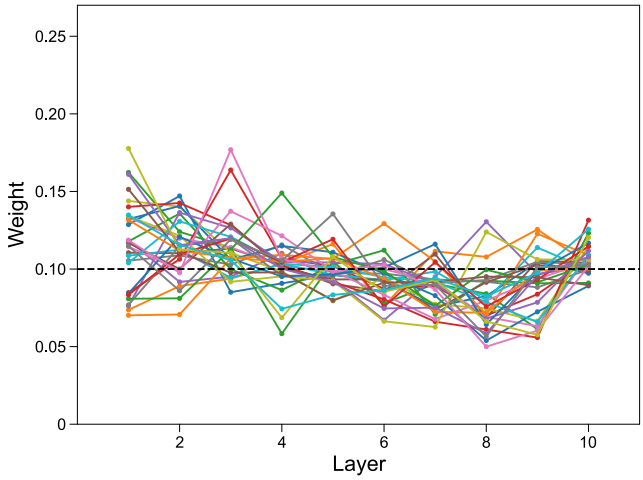

Figure A.1: Weight parameters of 10-layer MLAP-Weighted models for the synthetic dataset. A line shows the weight vector in a model (30 lines in total). The dashed horizontal line shows the weight when all layers contribute to the final graph representation equally.

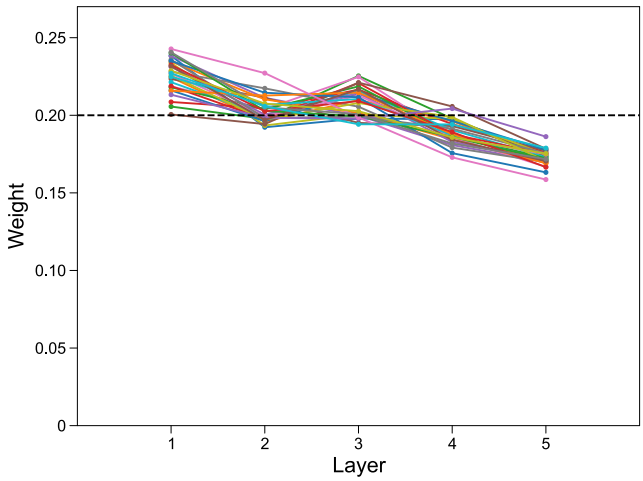

Figure A.2: Weight parameters of 6-layer MLAP-Weighted models for ogbg-molhiv. A line shows the weight vector in a model (30 lines in total). The dashed horizontal line shows the weight when all layers contribute to the final graph representation equally.

Appendix B

Visual Attention Map for Source Code

We were interested in the attentional behavior of the human programmers while developing an MLAP source code summarization model in Chapter 3. We devised a method to quantify the human attention using a machine learning (ML) model with an attention mechanism. In this appendix chapter, we present a technique to visualize the ML attention in a two-dimensional space, called *attention map*, and present the preliminary results comparing the attention map and gaze behavior of a novice human programmer.

B.1 Introduction

Program comprehension is a dominant process in software development and maintenance. Programmers spent 50 to 60 % of their time on program comprehension in a large-scale field study [115]. The study has also shown that the senior programmers spent less time on program comprehension. In other words, the time for program comprehension can be reduced through appropriate experience or education. Such efficient program comprehension might lead to productivity enhancement of the software development process.

The gaze behavior of experts could provide an insight on how to improve the efficiency of program comprehension. We consider that expert programmers can comprehend source code efficiently by directing their gaze, or attention, to

important components in it. In previous studies, researchers conducted gaze measurement experiments with programmers to identify the attended targets. Uwano et al. [116] analyzed individual performance in reviewing source code of computer programs with gaze data. Their result showed that the subjects with high performance were likely to first read the whole lines of the source code from the top to the bottom briefly, and then concentrate their gaze to some specific areas. Crosby et al. [117] conducted gaze experiments to examine how programmers from different experience levels understood source code. Their results showed that the experienced programmers were more likely to focus their gaze on complex statements. However, reflecting the importance of components remains an issue in gaze behavior analysis for source code comprehension.

Apart from program comprehension, visual attention has been modeled to clarify its underlying mechanism. One of the representative studies of visual attention modeling is the saliency map as an indicator of stimulus-driven visual selection [118, 119]. The saliency map has been proposed by mimicking the neural mechanism of the early visual system of humans. In the saliency map theory, visual attention is assumed to be guided by high contrast locations of three elementary features: color, intensity, and orientation. Koide et al. [120] investigated the relationship between art-related expertise and the saliency map. They recorded the gaze behavior of artists and novices during the free viewing of various abstract paintings, and evaluated the consistency between the gaze distribution and the saliency map. The gaze distributions of artists were less consistent with the saliency map than novices. This discrepancy between the experts' gaze behavior and the saliency map, which is a bottom-up attention model, could be explained by the existence of top-down, goal-oriented attention mechanism, which can be modified by experience or education.

This study aims to develop a visual attention map that can identify important components of source code in a top-down, goal-oriented manner. To demonstrate this, we used code2vec, a neural network model for classification of source code with an attention mechanism, to identify important components in source code [97]. We conducted preliminary gaze experiments and a comparison analysis between the gaze behavior of a human subject and the visual attention map generated with our proposed method. Here, we assume the consistency between

the attention map and gaze distribution could be the support for the feasibility of this method to identify important factors.

The rest of this appendix chapter is organized as follows: Section B.2 reviews the mechanism of code2vec program summarization model and introduces our idea on how the attention map on top of code2vec's attention model contributes to analyzing programmers' gaze behavior. Section B.3 explains the procedure to generate the attention map and gaze experiment design using human subjects. Section B.4 summarizes the results of preliminary experiments. Finally, Section B.5 concludes this proposal and provides outlooks.

B.2 Attention Map for Source Code

Code2vec [97] is a machine learning model to learn a vector embedding of source code, called a *code vector*. Code2vec has an attention mechanism to recognize important components in source code for accurate name discrimination. The model has shown good performance in discriminating function names, which concisely represent their functionalities, and the authors showed that the attention mechanism is necessary for achieving good performance.

The left half of Figure B.1 illustrates how code2vec estimates the code vectors and how its attention mechanism defines the importance of source code components. First, the input source code is converted into an abstract syntax tree (AST), which is a tree data structure representing normalized syntactic information. Then, code2vec extracts *path contexts* from the AST. A path context consists of three elements: two terminal nodes (leaves) in the AST and the route connecting those terminal nodes. Code2vec extracts up to 200 path contexts and estimates a path context vector for each, and finally, the code vector is computed as a weighted sum of these path context vector (this step is not depicted in Figure B.1). The *attention* of code2vec defines this weight for the path contexts. A higher attention value means the correspondent path context contains important information for discriminating function name, and hence, the path context vector for such path context greatly affects the final code vector.

In this study, we assume the expertise of a programmer could be represented using the consistency between the code2vec's attention and a programmer's gaze

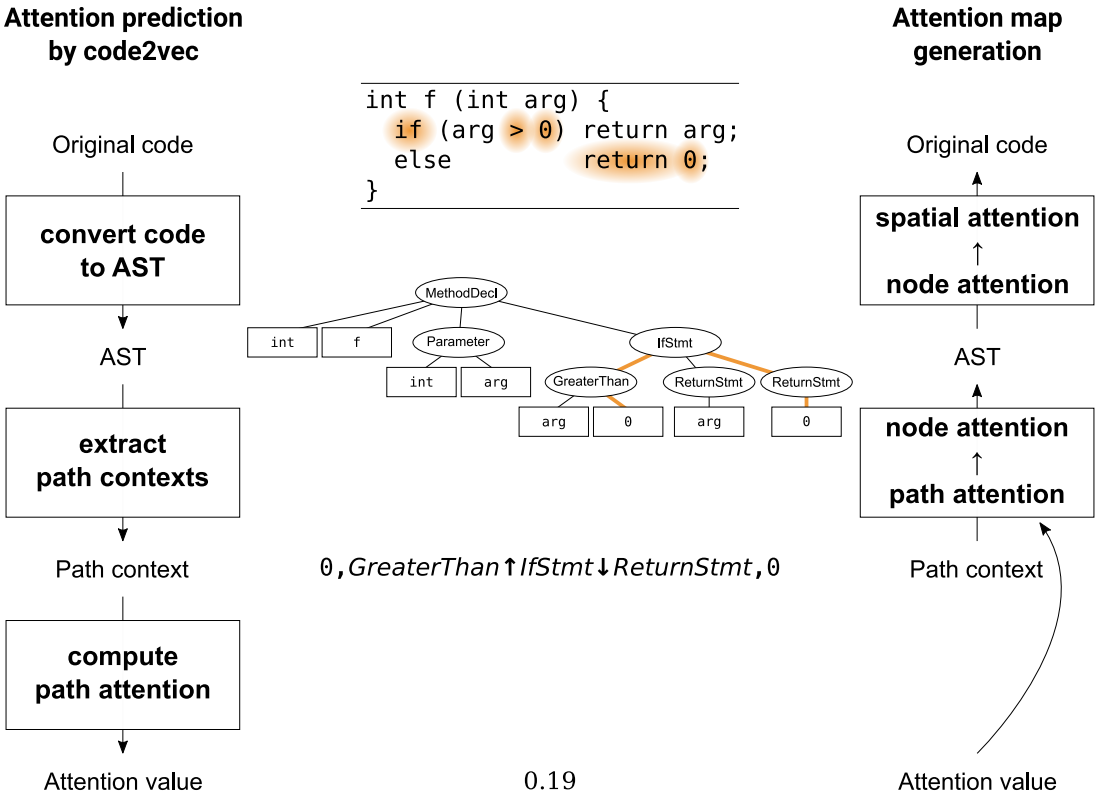

Figure B.1: Attention estimation by code2vec (left) and proposed attention map generation procedure (right). The set of orange edges in AST shows an example of a path, and the path attention value (0.19) is assigned to the tokens in the path (orange shadows in the original code). The attention values for each token are summed up for all paths by repeating this procedure.

focus. Based on this assumption, we evaluate how much a programmer focuses on the source code components that are estimated as important by code2vec's attention mechanism. However, code2vec's attention on linearized AST path contexts is difficult to directly compare with subjects' spatial gaze distribution.

To fill this gap, we propose a method to generate visual *attention map*, which quantifies the importance of each component in the provided source code, using the attention value estimated by code2vec. As code2vec computes the attention for each path context in the given AST of source code, we reconstructed a visual attention map based on the attention value of each path context. The right half of Figure B.1 illustrates the attention map generation procedure. In summary, we first computed attention values of the nodes appearing in the given AST, then

generated a spatial map over a source code image using those node attention values.

First, each path context was decomposed into a list of nodes in the AST. Then, the attention value for each path context was added to the attention value of each node in this list. Repeating this procedure for each path context, the attention distribution over the set of AST nodes was obtained. Then, we converted this node attention distribution into a visual attention map as a mixture of Gaussian functions. Some of the nodes had their correspondent tokens in the source code, like `if` for *IfStmt*, and `>` symbol for *GreaterThan*. For those correspondent tokens, a two-dimensional Gaussian function was allocated to each token such that

1. its center was located at the coordinate of the center of the token in the stimulus image,
2. its maximum height was equal to the attention value defined in the attention distribution over AST nodes, and
3. its variance corresponded to the spatial size of the token in an image.

The attention map was obtained as the summation of these Gaussian functions for all tokens.

B.3 Preliminary Experiments

B.3.1 Acquisition of source code

To test the feasibility of visual attention map generation, we conducted a preliminary experiment using a set of code snippets implementing fundamental algorithms. Based on two popular textbooks about computer algorithms [121, 122], we first selected eleven fundamental algorithms: binary search, linear search, bubble sort, selection sort, insertion sort, greatest common divisor, power, primality testing, run-length encoding, string sort, and substring search. We then collected 1251 Java code snippets implementing the selected algorithms from an open code set provided by AIZU ONLINE JUDGE[9]. In this study, we used a set of 72 code

[9] `http://judge.u-aizu.ac.jp/onlinejudge/`

snippets with minimum deviations of superficial characteristics, i.e., lines of code (LOC) and characters per line (CPL). To further mitigate non-semantic visual variations, the indentation styles of all code snippets were normalized by replacing a tab-space with two white spaces. For keeping algorithmic diversity, the selected code set included six snippets for each algorithm and twelve snippets for linear search. The code set allowed us to examine the feasibility of our proposed method based on a variety of fundamental algorithms.

B.3.2 Gaze Experiment

For the aforementioned spatial attention map, the consistency between the map and the programmer's gaze distribution was quantified. We recorded a programmer's gaze distribution using Tobii Pro TX300 (Tobii Technology, Sweden) during presenting the source code image as visual stimuli (see Section B.3.1). The device has a 23-inch display of full HD resolution (1920 px in width and 1080 px in height). We recorded the subject's gaze points with the sampling rate of 120 Hz. The experimental procedure was controlled by PsychoPy [123].

After the experiment, the stimulus images were clipped into squares 840 px on a side prior to the further analysis to avoid excessively high consistency due to the inclusion of the blank area in the images. Outliers in the recorded gaze data that exceeded these square boundaries were removed. The proportion of removed gaze points against the whole data was less than 0.1 %.

B.3.3 Evaluation

To quantify the consistency, we adopted an evaluation method proposed in [120]. The receiver operating characteristics (ROC) curve was calculated by defining the ground truth as the subject's gaze distribution and the estimation as the binarized attention map computed with code2vec. With a attention threshold, the code2vec attention map C was binarized into C^{bin}:

$$C^{\text{bin}}_{x,y} = \begin{cases} 1 & \text{if } C_{x,y} > \text{threshold} \\ 0 & \text{otherwise,} \end{cases} \tag{B.1}$$

where $C_{x,y}$ and $C^{\text{bin}}_{x,y}$ represent the attention value for pixel (x, y) in C and C^{bin}, respectively. With this binarized attention map, the true positive rate (TPR) and the false positive rate (FPR) for a gaze distribution are calculated as follows.

$$\text{TPR} = \frac{\sum_{x,y} G^{+} \circ C^{\text{bin}}}{\sum_{x,y} G^{+}}, \tag{B.2}$$

$$\text{FPR} = \frac{\sum_{x,y} G^{-} \circ C^{\text{bin}}}{\sum_{x,y} G^{-}}, \tag{B.3}$$

where G^{+} is the gaze distribution which counts the gaze point per pixel, whereas G^{-} is a binary negation of G^{+}, s.t. $G^{-}_{x,y} = 1 \Leftrightarrow G^{+}_{x,y} = 0$ and $G^{-}_{x,y} = 0 \Leftrightarrow G^{+}_{x,y} > 0$. Also, $\circ$ denotes the Hadamard product (pixel-wise product) of two maps or distributions. ROC curve was obtained by computing these TPR and FPR with varying the threshold and plotting those values.

After calculating the ROC curve, the area under the curve (AUC) was obtained. The AUC quantifies the consistency between the subject's gaze distribution and the attention map. A higher AUC value indicates that the subject strongly focused their gaze on important components in source code, and is thus, assumed to represent their expertise in reading source code.

B.4 Results

As a preliminary experiment, we quantified the attention maps for the target source code (see Section B.3.1), and evaluated the consistency of those maps against gaze distributions recorded from a human subject. The rest of this section describes a representative result obtained using a substring search algorithm depicted in Listing B.1.

Figure B.2a shows the estimated spatial attention map for the source code. The attention map was sparse, and there were only a few tokens with high attention value (deep orange color). Table B.1 lists the top 5 of those highly attended tokens. This source code implements a substring search algorithm that counts the number of words until the word "END_OF_TEXT" appears, and these highly attended tokens—especially the tokens `if`, `while`, and `"END_OF_TEXT"`—match our intuitive evaluation of token importance.

```
1  public class Main {
2    public static void main(String[] args) {
3      Scanner in = new Scanner(System.in);
4      String word = in.next();
5      int count = 0;
6
7      word = word.toLowerCase();
8
9      while (true) {
10       String str = in.next();
11       if (str.equals("END_OF_TEXT")) {
12         break;
13       }
14       str = str.toLowerCase();
15       if (str.equals(word)) {
16         count++;
17       }
18     }
19     System.out.println(count);
20   }
21 }
```

Listing B.1: A substring search algorithm written in Java

#	Line number	Token	Attention value
1	11	`if`	1.33
2	11	`"END_OF_TEXT"`	1.31
3	9	`while`	1.17
4	2	`args`	0.89
5	7	`word`$^{(*)}$	0.48

Table B.1: Top 5 tokens with strong attention. *: The one at the left-hand side of the equal symbol.

We recorded the gaze behavior of a research student in the information science division who had a little experience in Java programming (i.e., not an expert programmer). Figure B.2b shows the raw recorded gaze data. Each blue dot represents a gaze point. The subject scanned the entire code region without noticeable focuses.

After recording the gaze distribution, we evaluated the coincidence between the gaze distribution and attention map. The calculated ROC curve is shown in Figure B.2c. The AUC of this ROC curve was 0.85, and this was regarded as moderate consistency.

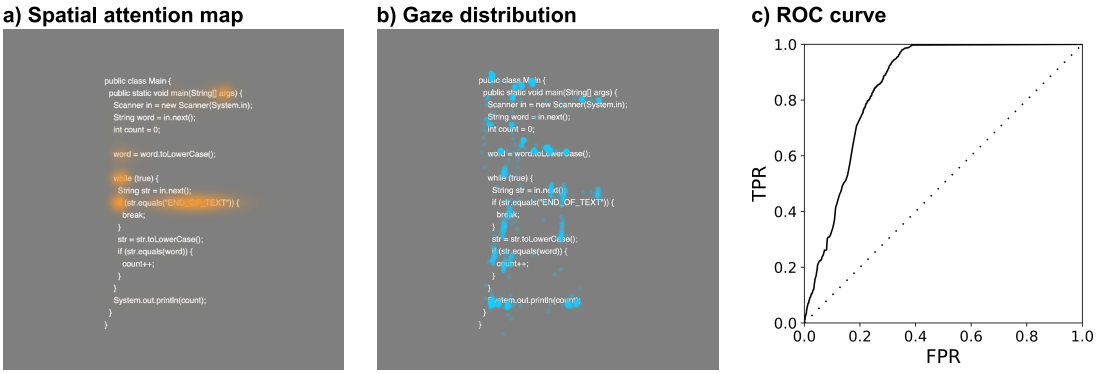

Figure B.2: Preliminary result of a gaze experiment on a novice programmer. **a)** Spatial attention map computed from the code2vec's attention. **b)** Raw gaze distribution recorded for a novice programmer. **c)** ROC curve showing the consistency between the attention map and subject's gaze distribution (AUC = 0.85).

B.5 Discussion

We proposed the visual attention map for source code using code2vec and evaluated the consistency between the attention map and gaze distribution recorded with a subject. The AUC of 0.85 can be regarded as moderate consistency. This result supports the feasibility of our proposed method as the attention model to some extent. The proposed method enables us to evaluate the behavior related to higher cognitive function like program comprehension. This approach may also be applicable to other higher cognitive functions.

The generated visual attention map showed sparser distribution rather than the gaze distribution. Because the subject in this preliminary experiment was not an expert programmer, gaze distribution was not so concentrated. Expert programmers may have sparser gaze distributions as fewer gaze points may lead to a reduction of the time for program comprehension. For the future experiments, we will recruit multiple expert programmers and novices to show the validity of this attention map.

Other attention models, like code2seq [98], can be alternatives to generate different types of attention maps. For example, the seq2seq type attention model like code2seq can model the dynamic transition of attention although this study considered static attention [124]. By providing attended words as the output

of a decoder, it may be possible to model actual dynamic attention. Note that introducing attended words to attention in a model is different from the current study as this study merely evaluated the correspondence of gaze distribution and attention weights of code2vec. In the future, we will develop such models in parallel with the aforementioned experimental study.

References for Appendix B

[115] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2018.

[116] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Eye Tracking Research and Applications Symposium (ETRA)*, pages 133–140, 2006.

[117] M. E. Crosby, J. Scholtz, and S. Wiedenbeck. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *Proceedings of the 14th Workshop of the Psychology of Programming Interest Group*, pages 58–73, 2002.

[118] L. Itti, C. Koch, and E. Niebur. A Model of Saliency-Based Visual Attention for Rapid Scene Analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(11):1254–1259, 1998.

[119] L. Itti and C. Koch. A Saliency-Based Search Mechanism for Overt and Covert Shifts of Visual Attention. *Vision research*, 40(10-12):1489–1506, 2000.

[120] N. Koide, T. Kubo, S. Nishida, T. Shibata, and K. Ikeda. Art Expertise Reduces Influence of Visual Salience on Fixation in Viewing Abstract-Paintings. *PLOS ONE*, 10(2):1–14, 2015.

[121] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[122] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.

[123] J. Peirce, J. R. Gray, S. Simpson, M. MacAskill, R. Höchenberger, H. Sogo, E. Kastman, and J. K. Lindeløv. PsychoPy2: Experiments in Behavior Made Easy. *Behavior Research Methods*, 51(1):195–203, 2019.

[124] Y. Ikutani, N. Koganti, H. Hata, T. Kubo, and K. Matsumoto. Toward Imitating Visual Attention of Experts in Software Development Tasks. In *Proceedings of the 6th International Workshop on Eye Movements in Programming*, pages 33–36, 2019.

Acknowledgements

I would like to thank Prof. Kazushi Ikeda from the bottom of my heart for having me in the Mathematical Informatics Lab. Ikeda-sensei is the greatest mentor I have ever met. The way he sees, considers, and discusses things has had an indispensable impact on my growth as a researcher. I am also deeply grateful to Prof. Takatomi Kubo for his immeasurable support throughout my Ph.D. period. Not only did I learn countless things through intense discussions with Kubo-sensei, they were always intellectually fun. It is one of the highest aims in my life to obtain as wide and deep knowledge on science as he possesses. I would like to express my sincere gratitude to all the thesis committee members—Prof. Hayaru Shouno, Prof. Kenichi Matsumoto, Prof. Junichiro Yoshomoto, Prof. Makoto Fukushima, and Prof. Chie Hieida. I would also like to thank all the labmates for making my daily life in NAIST more enjoyable and exciting in terms of both research and recreation. In particular, discussion with Dr. Nishanth Koganti, Dr. Bryan Lao, Dr. Yoshiharu Ikutani, Dr. Renzo Tan, Ms. Adline Bikeri, Mr. Mario Aburto, Mr. Panyawut Sri-iesaranusorn, and Ms. Kiyoka Ikeda have given me inspiring insights on various topics. Finally, my greatest gratitude goes to my family and my fiancée Midori. I would not have been able to make it through the five-year journey at NAIST without their unlimited understanding and support.

Publication List

The early version of the work in this dissertation was published as listed below.

- T. D. Itoh, T. Kubo, and K. Ikeda. Multi-Level Attention Pooling for Graph Neural Networks: Unifying Graph Representations with Multiple Localities. *Neural Networks*, 145:356–373, 2022.

- T. D. Itoh, T. Kubo, K. Ikeda, Y. Maruno, Y. Ikutani, H. Hata, K. Matsumoto, and K. Ikeda. Towards Generation of Visual Attention Map for Source Code. In *Proceedings of the 2019 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference*, pages 951–954, 2019.

Also, works in related fields were published as follows.

- T. D. Itoh, K. Ishihara, and J. Morimoto. Implicit Contact Dynamics Modeling with Explicit Inertia Matrix Representation for Real-Time Model-Based Control in Physical Environment. *Neural Computation*, 34:1–18, 2021.

- 伊藤健史, 石原弘二, 森本淳. 制御システム. 特開 2021-084188, 2021.

- 計測自動制御学会 (編), 藤原幸一, 久保孝富 (編著), 山川俊貴, 伊藤健史, 中野高志, 吉本潤一郎, 松尾剛行, 藤田卓仙, 桐山瑶子 (著). 次世代医療 AI 生体信号を介した人と AI の融合 (計測・制御セレクションシリーズ). コロナ社, 2021.