Efficient Language Model Adaptation using Code Property Graph for Code Comprehension and Generation Tasks

Imam Nur Bani Yusuf[1], **Lingxiao Jiang**[1]

[1]Singapore Management University
imamy.2020@smu.edu.sg, lxjiang.smu.edu.sg

Abstract

Large language models have demonstrated promising performance across various software engineering tasks. Fine-tuning is a common practice to adapt these models for different downstream tasks, but it becomes challenging in resource-constrained environments due to the increased memory requirements associated with the growing number of trainable parameters. Several efficient fine-tuning approaches have been proposed; however, they do not take advantage of the inherent structure, control flow, and dependencies information within source code. We introduce Transducer Tuning, a technique to adapt large models for downstream code tasks using Code Property Graphs (CPGs) from input source code. Our approach introduces a modular component, called Transducer, that enriches the code embeddings from large models with structure, control-flow, and dependencies information from CPGs. Transducer consists of a GVE (Graph Vectorization Engine) and an ABF (Attention-Based Fusion) layers. The GVE extracts the CPGs from input source code and transforms structure, control-flow, and dependencies information from CPGs into graph feature vectors, which are then fused into the code embeddings generated by a large model via the ABF layer. By optimizing the transducers for different downstream tasks, Transducer Tuning enhances embeddings generated by the large model without the need to fine-tune the large model for each specific task. We have evaluated our approach on four downstream tasks: code summarization, assert generation, code repair, and code translation. Our results demonstrate that Transducer Tuning achieves competitive performance compared to traditional fine-tuning strategies while requiring up to 99% fewer trainable parameters. Furthermore, Transducer Tuning remains competitive against state-of-the-art efficient fine-tuning approaches (e.g., LoRA, Prompt-Tuning, Prefix-Tuning), while only leveraging 1.5%-80% of the total trainable parameters used in the baselines. These findings suggest that integrating structure, control-flow, and dependencies information with our proposed Transducer Tuning enables more efficient model adaptation, creating further opportunities for end users to adapt large models to resource-constrained settings easily.

Introduction

Language models have demonstrated promising performance across various software engineering tasks, such as code generation (Zan et al. 2023), code summarization (Zheng et al. 2023; Wan et al. 2024), and code repair (Xia, Wei, and Zhang 2023; Jin et al. 2023). ”Pretrain, then finetune” is a common practice to leverage this model to solve specific downstream tasks (Devlin et al. 2019; Feng et al. 2020; Guo et al. 2021; Ahmad et al. 2021; Wang et al. 2023). These models are initially pretrained by large companies on huge corpora to learn general-purpose code representation through diverse code patterns (Shi et al. 2023; Han et al. 2024). End-users can solve downstream tasks by fine-tuning these pretrained model on downstream data.

Fine-tuning adapts language models to specific tasks by updating their parameters using input-output examples from the target task, typically through gradient descent (Rumelhart, Hinton, and Williams 1986). This process helps the models better align with the data from the downstream task, leading to the performance improvement. However, one challenge in fine-tuning is the increased memory demand that comes with the rising number of trainable parameters in a language model (Cai et al. 2020; Hu et al. 2022). In this work, we focus on downstream code-related tasks, with the goal of developing an efficient fine-tuning technique specifically tailored for these tasks.

Prior studies have proposed various efficient adaptation techniques, such as Adapter-based (Bapna and Firat 2019; Houlsby et al. 2019; Pfeiffer et al. 2020b,a, 2021; Liu et al. 2022a; Hu et al. 2022; Hyeon-Woo, Ye-Bin, and Oh 2022; Yeh et al. 2024; Ponti, Sordoni, and Reddy 2022; Kopiczko, Blankevoort, and Asano 2024) and Prompt-based (Li and Liang 2021; Lester, Al-Rfou, and Constant 2021; Liu et al. 2022b) approaches to address the memory requirement challenge. These approaches introduce trainable parameters that are significantly smaller than the language model and finetune only these parameters. The number of these parameters also affects how well language models adapt to a new task, where typically larger is better (Li and Liang 2021; Lester, Al-Rfou, and Constant 2021).

One drawback of the aforementioned approaches is they do not take advantage of the inherent structure, control flow, and dependencies information within source code. Those information, typically represented using graphs, offers a more accurate depiction of source code than just a simple sequences. By leveraging this information, we believe that language models can potentially achieve a better understanding of source code using fewer trainable parameters during the fine-tuning stage while minimizing performance degradation due to the reduced trainable parameters.

We introduce a new adaptation technique for downstream code tasks called Transducer Tuning. Transducer Tuning leverages modular neural network layers to efficiently adapt language models for these tasks, minimizing the number of trainable parameters while achieving good performance. The additional layers, referred to as Transducer, enhance model inputs with dependency information represented as Code Property Graphs (CPGs) (Yamaguchi et al. 2014), which capture the syntactic structure, control flow, and data dependencies of the input source code.

The Transducer consists of two key components: the Graph Vectorization Engine (GVE) and the Attention-based Fusion (ABF) layers. First, the GVE extracts CPGs from the input source code and converts the dependency information into graph feature vectors. These vectors are then fused into the code embeddings generated by a language model through the ABF layer, enriching the initial code embeddings with dependency information derived from the input source code. Finally, Transducer Tuning optimizes the Transducer for various downstream tasks, enabling the language model to to provide optimal inputs for the language model without the need to fine-tune it.

We have evaluated Transducer Tuning on four downstream tasks: code summarization, assert generation, code repair, and code translation. The results show that Transducer Tuning performs comparably to traditional full fine-tuning methods while requiring up to 99% fewer trainable parameters in all downstream tasks and models. Moreover, Transducer Tuning achieve competitive performance against efficient fine-tuning techniques such as LoRA (Hu et al. 2022), Prompt-Tuning (Lester, Al-Rfou, and Constant 2021), and Prefix-Tuning (Li and Liang 2021), while only leveraging 1.5%-80% of the total trainable parameters used in those baselines.

Our contributions are the following.

- We propose a novel adaptation method called Transducer Tuning to effectively adapt language models for downstream code tasks using CPGs that induces minimum trainable parameters while achieving good performance.
- We evaluate Transducer Tuning on four downstream tasks and show that Transducer Tuning can achieve comparable performance while requiring fewer trainable parameters against all baselines.

Code Property Graphs (CPGs)

Code Property Graphs (CPGs) (Yamaguchi et al. 2014) unify the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). The AST shows the structure of statements and expressions, the CFG outlines the execution order and conditions for code paths, and the PDG captures dependencies between statements, using edges to represent data and control dependencies.

Figure 1 illustrates the Code Property Graph (CPG) for the code presented in Listing 1. In this graph, the `DECL` nodes represent assignments, the `PRED` node represents the

Listing 1: A simple program for the CPG explanation.

```
1 def main():
2     x = a()
3     if x > 10:
4         x = 0
5         b()
```

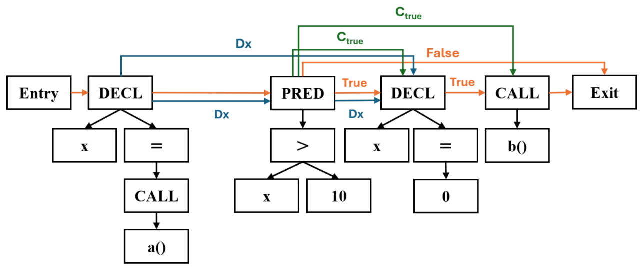

Figure 1: An example of Code Property Graph (CPG). DECL: declaration, PRED: predicate, CALL: function call, Black Edges: syntax tree, Orange Edges: control flow, Blue Edges: data dependency, Green Edges: control dependency.

conditional check, and the `CALL` nodes represent function calls. The black edges correspond to the syntactical structure of the code. The orange edges depict the control flow, illustrating the possible execution paths. Specifically, there are two possible paths: one where the condition `x > 10` in line 3 is true, and another where it is false. The C_{true} edges correspond to control dependencies, indicating that the subsequent assignment `x = 0` in line 4 and function call `b()` in line 5 depend on the condition `x > 10` in line 3 being true. The D_x edges represent data dependencies, showing how the value of `x` is utilized throughout the code.

By jointly taking into account the structure, control flow, dependencies in source code, we believe it can potentially helps language models to achieve a better understanding of source code using fewer trainable parameters during the fine-tuning stage, while minimizing performance degradation due to the reduced trainable parameters.

Transducer Tuning

Transducer's Architecture

Figure 2 shows the high-level architecture of Transducer Tuning. Transducer consists of two components, i.e., GVE (Graph Vectorization Engine) and ABF (Attention-Based Fusion) layer. The GVE generates the feature vector from the CPG extracted from the input source code. ABF layer subsequently fuses the feature vector into the code embedding generated by the language model. We refer to this language model as the backbone model in the subsequent paragraph.

Graph Vectorization Engine (GVE) comprises three subcomponents: Graph Extractor, Graph Vectorizer, and Graph Processing Layer. First, Graph Extractor extracts CPGs from the input source code c, resulting in a node list N and edge list E. Then, Graph Vectorizer transforms each

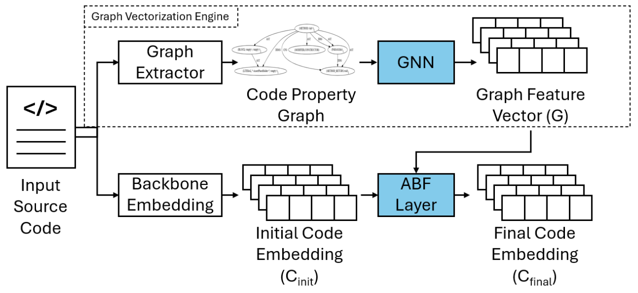

Figure 2: The high-level architecture of Transducer Tuning. The bleu colored components are updated during the fine-tuning stage. The Final Code Embedding is passed to the backbone model, which remains frozen during the fine-tuning stage.

node label $n_i \in N$ into a vector. This component can be viewed as a function F that maps an input x to its vector representation $\mathbf{h}$: $F : x \rightarrow \mathbf{h}$. In practice, the function F can be a pre-trained embedding model, TF-IDF, or binary vector. The output of the Graph Vectorizer is the node vector $\mathbf{H}_{\text{init}}$, where each node vector $\mathbf{h}_{\text{init,i}} \in \mathbf{H}_{\text{init}}$ has dimension d_{init}.

Next, a Graph Processing Layer converts the node vector $\mathbf{H}_{\text{init}}$ into a feature vector $\mathbf{G}$ with dimension d_{g}, which serves as the vector representation of the input CPG. The Feature Generator consists of several sub-layers: Normalization (Ba, Kiros, and Hinton 2016), Down Projection, Feature Generator, Up Projection, and Mean Pooling.

First, the Normalization Layer stabilizes training by ensuring that each node vector $\mathbf{h}_{\text{init,i}} \in \mathbf{H}_{\text{init}}$ has a normalized scale. We use Root Mean Squared (RMS) normalization (Raffel et al. 2020), which scales the input vector based on its root mean square. Next, the Down Projection Layer transforms each normalized node vector $\mathbf{h}_{\text{norm,i}} \in \mathbf{H}_{\text{norm}}$ using learnable weights $\mathbf{W}_{\text{down}}$, producing a down-projected vector $\mathbf{h}_{\text{down,i}} \in \mathbf{H}_{\text{down}}$ with dimension d_{down}, where $d_{\text{down}} < d_{\text{init}}$. This reduction in dimensionality decreases computational complexity and memory usage, while encouraging the model to focus on the most relevant features.

The Feature Generator then takes the down-projected node vectors $\mathbf{H}_{\text{down}}$ as input and produces feature vectors $\mathbf{H}_{\text{feature}}$ using learnable weights $\mathbf{W}_{\text{feature}}$. Finally, the Up Projection Layer transforms each feature vector $\mathbf{h}_{\text{feature,i}} \in \mathbf{H}_{\text{feature}}$ using learnable weights $\mathbf{W}_{\text{up}}$, resulting in an up-projected feature vector $\mathbf{H}_{\text{up}}$ with dimension $d_{\text{up}} > d_{\text{down}}$. This projection increases the model's capacity to represent complex dependencies in the input CPG.

The graph feature vector $\mathbf{G}$ is obtained by applying mean pooling to the node vectors $\mathbf{H}_{\text{up}}$. This vector serves as the final representation of the input CPG and is used to enhance the code embeddings generated by the backbone model via the ABF layer.

Attention-Based Fusion (ABF) Layer takes two inputs, i.e., the graph feature vector $\mathbf{G}$ from Transducer and the input code embedding $\mathbf{C}_{\text{init}}$ generated by the backbone model's embedding layer. The output is a code embedding $\mathbf{C}_{\text{final}}$ that has been enriched using structure, control-flow, and dependencies information encoded in the graph feature vector $\mathbf{G}$. The ABF layer comprises three subcomponents, i.e., Normalization, Attention-Fusion, and Final Projection Layers.

Initially, two Normalization Layer separately normalize the graph feature vector $\mathbf{G}$ and input code embedding $\mathbf{C}_{\text{init}}$ using RMS normalization, resulting in the normalized graph feature vector $\mathbf{G}_{\text{norm}}$ and normalized code embedding $\mathbf{C}_{\text{norm}}$. Then, the ABF Layer fuses these two vectors using the standard attention mechanism as in (Vaswani et al. 2017).

The ABF Layer consists of three trainable weight matrices: $\mathbf{W}_{\text{Q}}$, $\mathbf{W}_{\text{K}}$, and $\mathbf{W}_{\text{V}}$. The matrices $\mathbf{W}_{\text{Q}}$ and $\mathbf{W}_{\text{V}}$ transform the input code embedding $\mathbf{C}_{\text{init}}$ into the query vector $\mathbf{Q}$ and value vector $\mathbf{V}$, respectively, while $\mathbf{W}_{\text{K}}$ converts the graph feature vector $\mathbf{G}$ into the key vector $\mathbf{K}$. Each of these vectors, $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, is two-dimensional with a dimension of d_{abf}. Then, we apply the attention mechanism as in (Vaswani et al. 2017).

First, we compute the dot product between the query vector $\mathbf{Q}$ and key vector $\mathbf{K}$ to obtain alignment scores, determining the focus each token embedding $\mathbf{c}_{\text{init,i}} \in \mathbf{C}_{\text{init}}$ should receive based on the graph feature vector $\mathbf{G}$. These scores are rescaled by $\sqrt{d_{\text{abf}}}$, the dimensionality of $\mathbf{K}$, and passed through a softmax function to generate normalized attention weights. The attention weights are then used to compute a weighted sum of the value vector $\mathbf{V}$.

Next, the Final Projection Layer transforms the output of the attention mechanism into the final code embedding $\mathbf{C}_{\text{final}}$ using the weight matrix $\mathbf{W}_{\text{final}}$. Each vector $\mathbf{c}_{\text{final,i}} \in \mathbf{C}_{\text{final}}$ matches the hidden dimension d_{backbone} of the backbone model. Finally, the code embedding $\mathbf{C}_{\text{final}}$ is passed to the encoder/decoder of the backbone model to generate a prediction.

Usage Scenario

The workflow for using Transducer Tuning involves two stages: training the Transducer and performing inference. During training, the service provider or end-user specifies the backbone model and trains a Transducer using input-output samples from the target downstream task. Only the Transducer's parameters are updated, while the backbone model's parameters remain frozen. In the inference stage, the Transducer enriches the input embeddings generated by the backbone model from unseen downstream data. If a new downstream task arises, a new Transducer can be trained using the same backbone model. The service provider or end-user only needs to deploy the newly trained Transducer, leaving the backbone model and the first Transducer unchanged. Additionally, the Transducer can be easily removed if no longer needed, without affecting the backbone model or other Transducer serving different tasks.

Experimental Setting

Datasets

We evaluate Transducer Tuning on four downstream tasks: code summarization, assert statement generation, code repair, and code translation. For each task, we selected datasets commonly used in recent related studies. We briefly describe each dataset below.

For code summarization, we use the clean Java subset (Shi et al. 2022) of the CodeSearchNet (Husain et al. 2019) containing Java methods paired with Javadoc descriptions. For assert statement generation, we use a dataset from Watson et al. (Watson et al. 2020) comprising test method and assert statement pairs. For code repair, we leverage a dataset released by Tufano et al. (Tufano et al. 2018) consisting of pairs of buggy and fixed code. Finally, for code translation, we use the dataset from CodeXGLUE (et al. 2021) containing parallel code snippets with equivalent functionality in Java and C#. All datasets are curated from open-source GitHub projects.

Dataset Decontamination and Preprocessing

We utilize preprocessed and cleaned datasets that have been divided into training, validation, and testing sets by their respective authors. To ensure data integrity, we first check for potential leakage between splits across all datasets. This process involves two stages: first, we remove exact matches between splits, and then we eliminate near-duplicates using Locality Sensitive Hashing (LSH) and MinHash. Specifically, we tokenize each instance and generate a MinHash signature for each one, which efficiently estimates the Jaccard similarity between instances. We then use LSH to group similar items together and remove those with a similarity score greater than 0.8. As a result, 41% to 53% of instances are retained from the original test split for code-to-code translation tasks, while 98% of instances are maintained for code summarization.

After this, we extract Code Property Graphs (CPGs) for each method in the datasets using Joern.[1] Instances where Joern fails to extract CPGs due to missing dependencies or runtime errors are excluded. Finally, we generate node vectors within the CPGs by converting node labels using the generic embedding model mxbai-embed-large-v1.[2]

In the end, the code summarization dataset contains 82K Java methods for training, 9.1K for validation, and 3.1K for testing. The assert statement generation dataset includes 50K instances for training, 6.3K for validation, and 3.3K for testing. The code repair dataset consists of 52K pairs of buggy and fixed code for training, 6.5K for validation, and 2.8K for testing. The code translation dataset contains 10.3K parallel code snippets for training, 500 instances for validation, and 370 for testing. Detailed statistics are included in the supplemental material.

Transducer Tuning Implementation

We leverage CodeT5+ 220M and CodeT5+ 770M (Wang et al. 2023) as the backbone model because they are the recent models that can be fine-tuned in our local machine. Furthermore, these models are popular with each having more than 10K monthly downloads on HuggingFace as of August 2024. We utilize GATv2 (Brody, Alon, and Yahav 2022) as the Feature Generator. We use 8 as the d_{down} and d_{abf} values.

[1]https://github.com/joernio/joern

[2]https://huggingface.co/mixedbread-ai/mxbai-embed-large-v1

Baselines

We compare Transducer Tuning against several baselines to evaluate its effectiveness. First, we use full fine-tuning. This represents the upper bound, where all backbone parameters are optimized for the task. We also evaluate against without any fine-tuning, which serves as the lower bound as the model relies only on its pre-trained state. We also use a strawman baseline, Linear Adapter, where a linear layer directly transforms the embeddings generated by the backbone model before passing them to the encoder.

Additionally, we compare Transducer Tuning with the widely-used efficient fine-tuning technique, LoRA (Hu et al. 2022). We also include Prefix-Tuning (Li and Liang 2021) and Prompt-Tuning (Lester, Al-Rfou, and Constant 2021) as additional baselines. Each of these has specific hyperparameters that affect the number of trainable parameters. To ensure fair comparisons, we tune these hyperparameters such that these approaches can yield good performance with the least optimal number of trainable parameters.

For LoRA, we tune the rank of the trainable matrices r and consider candidate values $\{4, 8\}$. We inject the low-rank matrices into the query and key vectors of the attention layers within the backbone model. For Prefix-Tuning and Prompt-Tuning, we adjust the lengths of prefixes p and soft-prompts s respectively. The candidate values for both are $\{5, 10, 25, 50\}$. All these candidate values are derived based on ablation studies from their respective papers (Hu et al. 2022; Li and Liang 2021; Lester, Al-Rfou, and Constant 2021).

We find the optimal values by tuning the model using 20% of the training set and evaluating them on the whole validation set. We perform the tuning for each task, each backbone model, and each tuning method. Hence, different datasets and backbone models may have different trainable parameters for each fine-tuning technique. In total, we perform more than 100 tuning trials. We report the selected along with the results in the next section. The detailed hyperparameter tuning results are included in our supplemental material.

Metrics

We evaluate Transducer Tuning against the baselines along two dimensions: efficiency and performance. Our goal is for Transducer Tuning to use fewer trainable parameters while minimizing performance degradation compared to the baselines. To assess efficiency, we compare the number of trainable parameters in Transducer Tuning with those in the baselines, following (Hu et al. 2022; Li and Liang 2021; Lester, Al-Rfou, and Constant 2021). Performance is measured using the smoothed BLEU (Papineni et al. 2002; Lin and Och 2004) for code summarization and CodeBLEU (Ren et al. 2020) for code-to-code translation, following the CodeXGLUE benchmark (et al. 2021). We focus on the relative difference in BLEU and CodeBLEU scores between Transducer Tuning and the baselines. Each experiment is run three times with different random seeds, and we report the average results.

Model	Tuning Method	Summarization	Assert Generation	Code Translation	Code Repair
CodeT5+ 220M	Transducer Tuning	99.84 ± 0.21	82.32 ± 0.30	96.60 ± 1.31	98.10 ± 0.39
	No Finetuning	95.49 ± 0.00	76.85 ± 0.00	94.47 ± 0.00	96.00 ± 0.00
	Full Finetuning	99.91 ± 0.01	83.16 ± 0.01	97.78 ± 0.00	99.87 ± 0.00
	Linear Adapter	98.05 ± 0.88	82.48 ± 0.02	97.70 ± 0.12	99.31 ± 0.71
	LoRA	99.91 ± 0.00	83.17 ± 0.00	97.78 ± 0.00	99.87 ± 0.00
	Prefix-Tuning	99.93 ± 0.01	83.17 ± 0.00	97.78 ± 0.00	99.87 ± 0.00
	Prompt-Tuning	99.91 ± 0.01	83.17 ± 0.00	94.40 ± 0.27	97.46 ± 1.13
CodeT5+ 770M	Transducer Tuning	98.11 ± 1.61	81.16 ± 0.71	94.88 ± 0.08	96.75 ± 3.94
	No Finetuning	87.90 ± 0.00	74.13 ± 0.00	90.10 ± 0.00	95.71 ± 0.00
	Full Finetuning	99.81 ± 0.01	83.16 ± 0.01	97.78 ± 0.00	99.87 ± 0.00
	Linear Adapter	98.24 ± 1.39	81.23 ± 0.88	97.77 ± 0.02	99.82 ± 0.02
	LoRA	99.79 ± 0.02	83.17 ± 0.00	97.78 ± 0.00	99.87 ± 0.00
	Prefix-Tuning	99.85 ± 0.00	83.15 ± 0.02	97.78 ± 0.00	99.87 ± 0.00
	Prompt-Tuning	99.82 ± 0.01	83.17 ± 0.00	90.15 ± 0.01	99.22 ± 0.71

Table 1: Performance comparison across different tuning methods for various tasks. Transducer Tuning yields significant improvement against No Fine-tuning and competitive performance against other tuning baselines. The standard deviations represent the variability across two trials, each conducted with a different random seed.

Model	Transducer	Full Finetuning	LoRA	Linear Adapter	Prefix-Tuning	Prompt-Tuning
CodeT5+ 220M	3.52	4.48	4.48	3.68	4.49	3.03
CodeT5+ 770M	5.76	8.20	8.19	8.20	8.20	6.13

Table 2: Average improvement against No Fine-tuning across different tuning methods. Higher is better.

Results

Table 1 presents the overall performance results on the selected downstream tasks. The improvement of Transducer Tuning over the No Fine-tuning baseline is substantial. Furthermore, Transducer Tuning performs competitively against Full Fine-tuning and other efficient tuning methods. Notably, these performance gains are achieved with the fewest trainable parameters among the tuning methods, as illustrated in Table 3.

Comparison with No Fine-tuning. Table 1 demonstrates that Transducer Tuning can enhance the performance of the backbone model without any fine-tuning. Table achieving an average improvement of 3.52 for CodeT5+ 220M and 5.76 for CodeT5+ 770M. For CodeT5+ 220M, Transducer Tuning boosts performance by 4.35 in summarization, 5.47 in assert generation, 2.13 in code translation, and 2.10 in code repair. The improvements are even more pronounced with CodeT5+ 770M, with increases of 10.21 in summarization, 7.03 in assert generation, 4.78 in code translation, and 1.04 in code repair. Based on Table 3, these gains are achieved with only 30K additional parameters for CodeT5+ 220M, representing just 0.0014% of the total parameters in the backbone model. For CodeT5+ 770M, the improvement requires only 37K additional parameters, which constitutes 0.005% of the total parameters in the backbone model.

Comparison with Other Tuning Baselines. Table 1 demonstrates that Transducer Tuning achieves competitive performance compared to other tuning baselines. Table 2 highlights the improvements of various tuning methods over No Fine-tuning. For CodeT5+ 220M, the largest performance difference is 0.97 between Transducer Tuning and Prefix-Tuning, while for CodeT5+ 770M, the largest difference is 2.43 between Transducer Tuning and Full Finetuning, Linear Adapter, and Prefix-Tuning.

To put these differences into perspective, it is important to consider the efficiency metric. Table 4 shows the ratio of trainable parameters for each tuning method relative to Transducer Tuning, with all values greater than 1. This indicates that every baseline requires more parameters than Transducer Tuning. For instance, Prefix-Tuning uses 18 times more trainable parameters than Transducer Tuning for CodeT5+ 220M. Given this context, a difference of 0.97 can be considered small. Similarly, for CodeT5+ 770M, Prefix-Tuning and Linear Adapter use 18 and 28 times more parameters, respectively, while Full Fine-tuning requires 19K times more. Hence, a difference of 2.43 can also be regarded as small when considering efficiency. Despite these substantial differences in parameter usage, the relatively minor performance gaps for CodeT5+ 770M suggest that Transducer Tuning delivers comparable performance to other tuning baselines while being significantly more efficient.

Besides efficiency, the performance differences between Transducer Tuning and other tuning methods remain small, particularly when considering the significant improvements over the No Fine-tuning baseline. For example, in assert generation with CodeT5+ 770M, the gap between Transducer Tuning and Full Fine-tuning is only 2 points, while Transducer Tuning outperforms the No Fine-tuning baseline by 7.03 points. This trend is consistent across most results in Table 1, underscoring that the performance gaps between Transducer Tuning and other tuning methods are minimal.

In summary, Transducer Tuning efficiently enhances model performance with minimal additional parameters. It consistently outperforms the No Fine-tuning baseline

Model	Tuning Method	Summarization	Assert Generation	Code Translation	Code Repair	Average
CodeT5+ 220M	Transducer Tuning	30,728	30,728	30,728	30,728	30,728
	Full Fine-tuning	222,882,048	222,882,048	222,882,048	222,882,048	222,882,048
	Linear Adapter	589,824	589,824	589,824	589,824	589,824
	LoRA	884,736	442,368	884,736	442,368	663,552
	Prefix-Tuning	184,320	921,600	184,320	921,600	552,960
	Prompt-Tuning	38,400	76,800	38,400	76,800	57,600
CodeT5+ 770M	Transducer Tuning	37,128	37,128	37,128	37,128	37,128
	Full Fine-tuning	737,639,424	737,639,424	737,639,424	737,639,424	737,639,424
	Linear Adapter	1,048,576	1,048,576	1,048,576	1,048,576	1,048,576
	LoRA	2,359,296	1,179,648	2,359,296	1,179,648	1,769,472
	Prefix-Tuning	491,520	491,520	491,520	1,228,800	675,840
	Prompt-Tuning	102,400	102,400	102,400	102,400	102,400

Table 3: Trainable parameter count comparison across different tuning methods. Transducer Tuning leverages the fewest trainable parameters among all tuning methods. LoRA, Prefix-Tuning, and Prompt-Tuning can have different parameter counts for different tasks due to different tuned hyperparameters.

Model	Full Fine-Tuning	LoRA	Linear Adapter	Prefix-Tuning	Prompt-Tuning
CodeT5+ 220M	7253	22	19	18	2
CodeT5+ 770M	19867	48	28	18	3

Table 4: Ratio of trainable parameters in tuning baselines compared to Transducer Tuning. All value larger are larger than 1, indicating that the all baselines uses more parameters than Transducer Tuning. To exemplify, a value of 7253 for Full Fine-tuning means that this approach uses 7253 times more trainable parameters than Transducer Tuning.

across all tasks, demonstrating its effectiveness in optimizing the backbone model. Compared to other tuning baselines, Transducer Tuning delivers competitive results while using significantly fewer trainable parameters.

Discussion

To assess the impact of information from Code Property Graphs (CPGs) and each component in Transducer Tuning on model performance, we trained four variants of Transducer Tuning. The first variant, GVE + ABF, serves as the default setting that uses the GNN for GVE and the ABF layer. The second variant, GVE-only, removes the ABF layer, summing the graph features generated by the GNN directly with the input code embeddings from the large model. The third variant, ABF-only, excludes the GVE, relying solely on the ABF. The fourth variant, Linear, replaces the GNN in GVE with a simple linear layer to produce the graph feature vectors. In this variant, the node vectors are treated as a sequence.

Information from CPGs Enhance Model Performance. Table 5 presents the results from these experiments. Models utilizing GNNs, specifically GAT and GVE-only, to process CPGs and generate graph features consistently outperformed the variants without GNNs, such as ABF-only and Linear. Notably, there is a significant performance drop in code translation and code repair tasks for the ABF-only and Linear variants. Across eight experiments covering four tasks and two models, either the GAT or GVE-only variant achieved the best performance in most cases. These findings clearly indicate that incorporating structure, control-flow, and dependency information through GNNs significantly enhances the model's ability to adapt to a new downstream code task with few trainable parameters.

The ABF Layer: Sometimes Useful, but Not Harmful. The results in Table 5 also suggest that the ABF layer can be either beneficial or neutral, depending on the task. For example, in CodeT5+ 770M, the GVE-only variant performed best in code translation and code repair. However, in other scenarios, such as Summarization and Assert Generation for CodeT5+ 770M, removing the ABF layer leads to poorer performance. This variability suggests that several methods can be used to fuse graph features with code embeddings from large models, and the optimal method may vary depending on the specific task. Further exploration into different fusion strategies is a promising direction for future work.

Related Works

Several techniques have been proposed to adapt pretrained models for downstream tasks, with direct fine-tuning being a common approach. This method updates all model parameters using task-specific data, typically a small set of input-output examples. It has been successfully applied in various software engineering tasks, including code repair (Mastropaolo et al. 2021; Jiang, Lutellier, and Tan 2021; Tian et al. 2020), code generation (Yusuf, Jiang, and Lo 2022; Yusuf, Jamal, and Jiang 2023), code mutant injection (Mastropaolo et al. 2021), code summarization (Wei et al. 2019), assert generation (Watson et al. 2020), and vulnerability detection (Chakraborty et al. 2022; Fu et al. 2022; Thapa et al. 2022). However, as model size increases, so do memory requirements due to the growing number of trainable parameters. To address these challenges, more efficient fine-tuning techniques, such as Adapter-based and Prompt-based methods, have been developed.

Model	Variant	Summarization	Assert Generation	Code Translation	Code Repair
CodeT5+ 220M	GVE + ABF	**99.84** ± 0.21	82.32 ± 0.30	**96.60** ± 1.31	98.10 ± 0.39
	GVE-only	99.31 ± 0.06	**83.14** ± 0.04	96.03 ± 0.00	99.37 ± 0.09
	ABF-only	94.33 ± 7.32	77.07 ± 8.62	92.53 ± 2.65	97.48 ± 3.38
	Linear	99.54 ± 0.09	83.08 ± 0.07	91.76 ± 0.36	**99.45** ± 0.59
CodeT5+ 770M	GVE + ABF	98.11 ± 1.61	81.16 ± 0.71	94.88 ± 0.08	96.75 ± 3.94
	GVE-only	96.23 ± 1.55	78.79 ± 2.51	**97.78** ± 0.00	**99.54** ± 0.12
	ABF-only	98.64 ± 0.52	**83.16** ± 0.01	91.11 ± 0.33	94.33 ± 4.16
	Linear	**99.17** ± 0.42	79.31 ± 4.64	90.40 ± 0.02	93.27 ± 0.88

Table 5: Ablation study results showing the impact of different components in Transducer Tuning on model performance across various tasks. The table compares the default setting (GVE + ABF) with three other variants: GVE-only, ABF-only, and Linear. Results indicate the significance of using information from CPGs and processing it using GNN layer to enhance model performance.

Adapter-based methods (Bapna and Firat 2019; Houlsby et al. 2019; Pfeiffer et al. 2020b,a, 2021; Liu et al. 2022a; Hu et al. 2022; Hyeon-Woo, Ye-Bin, and Oh 2022; Yeh et al. 2024; Ponti, Sordoni, and Reddy 2022; Kopiczko, Blankevoort, and Asano 2024) introduce additional trainable parameters into the backbone model, updating only these parameters during the adaptation stage rather than the entire model. However, determining the optimal placement for these trainable parameters is not trivial and requires an understanding of the model architecture. In contrast, our technique simplifies this process by modifying only the input embeddings, which are then processed through the encoder and/or decoder. This approach does not require end-users to have detailed knowledge of the model architecture. Moreover, as Transducer Tuning is easily applicable to any existing language model because it does not modify the model architecture.

Prompt-based methods (Li and Liang 2021; Lester, Al-Rfou, and Constant 2021; Liu et al. 2022b) append trainable soft-token parameters to the embeddings generated by large models. During training, only these soft-tokens are updated, while the rest of the model's weights remain frozen. This approach is somewhat similar to Transducer Tuning because it involves modifying the input embeddings. However, Transducer Tuning is distinct because it is specifically designed to incorporate information from graph data (e.g., CPGs) into the input embeddings, which is not addressed by the existing prompt-based methods.

Threats To Validity

Internal Validity. The internal validity of our study could be influenced by the choice of hyperparameters. We addressed this by tuning hyperparameters across all tasks using validation split. To further reduce the impact of random variability, we used a fixed random seed for all experiments and repeated each model configuration two times with different random seeds to ensure consistency.

External Validity. The external validity of our study could be impacted by dataset leakage between training, validation, and testing splits. To minimize this risk, we removed exact and near-duplicates using Locality Sensitive Hashing and MinHash. Additionally, since test instances may have been encountered during pretraining, we compared Transducer Tuning's performance against a No Fine-tuning baseline to better quantify its relative improvement.

Construct Validity. The choice of evaluation metrics may affect our study's construct validity. While BLEU and CodeBLEU are widely used (Ahmed et al. 2024; Hu et al. 2020; Yusuf, Jiang, and Lo 2022; Yusuf, Jamal, and Jiang 2023; Dey et al. 2022), they might not fully capture the nuances of our model's performance. However, our main goal is to demonstrate reduced training parameters with minimal performance degradation, which can be effectively shown through relative differences. BLEU and CodeBLEU adequately illustrate these differences, making the absolute performance values less critical.

Conclusion

We introduce Transducer Tuning, a technique for adapting large models to downstream code tasks by leveraging Code Property Graphs (CPGs) extracted from input source code. Our approach incorporates a modular component called Transducer, which enriches the code embeddings from large models with structural, control-flow, and dependency information from CPGs. Transducer consist of two key elements: the Graph Vectorization Engine (GVE) and the Attention-Based Fusion (ABF) layers. The GVE extracts CPGs from the input code and converts structural, control-flow, and dependency information into graph feature vectors. These vectors are then integrated into the code embeddings generated by the large model through the ABF layer. By optimizing the transducers for various downstream tasks, Transducer Tuning enhances the embeddings produced by the large model without requiring fine-tuning for each specific task. Our experimental results show that Transducer Tuning achieves performance comparable to strong baselines while using significantly fewer parameters.

In future work, we plan to explore the selection of code features for adapting language models with Transducer Tuning. While this study focus on CPGs, other graph features may be more effective for certain downstream tasks. Additionally, investigating the transferability of these features across different programming languages, particularly in low-resource scenarios, could yield valuable insights.

References

Ahmad, W. U.; Chakraborty, S.; Ray, B.; and Chang, K. 2021. Unified Pre-training for Program Understanding and Generation. In *NAACL-HLT*, 2655–2668. Association for Computational Linguistics.

Ahmed, T.; Pai, K. S.; Devanbu, P.; and Barr, E. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *ICSE*. ACM.

Ba, L. J.; Kiros, J. R.; and Hinton, G. E. 2016. Layer Normalization. *CoRR*, abs/1607.06450.

Bapna, A.; and Firat, O. 2019. Simple, Scalable Adaptation for Neural Machine Translation. In *EMNLP/IJCNLP (1)*, 1538–1548. Association for Computational Linguistics.

Brody, S.; Alon, U.; and Yahav, E. 2022. How Attentive are Graph Attention Networks? In *ICLR*. OpenReview.net.

Cai, H.; Gan, C.; Zhu, L.; and Han, S. 2020. TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning. In *NeurIPS*.

Chakraborty, S.; Krishna, R.; Ding, Y.; and Ray, B. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Software Eng.*, 48(9): 3280–3296.

Devlin, J.; Chang, M.; Lee, K.; and Toutanova, K. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*, 4171–4186. Association for Computational Linguistics.

Dey, S.; Vinayakarao, V.; Gupta, M.; and Dechu, S. 2022. Evaluating Commit Message Generation: To BLEU Or Not To BLEU? In *ICSE (NIER)*, 31–35. IEEE/ACM.

et al., S. L. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *NeurIPS Datasets and Benchmarks*.

Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; and Zhou, M. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *EMNLP (Findings)*, volume EMNLP 2020 of *Findings of ACL*, 1536–1547. Association for Computational Linguistics.

Fu, M.; Tantithamthavorn, C.; Le, T.; Nguyen, V.; and Phung, D. Q. 2022. VulRepair: a T5-based automated software vulnerability repair. In *ESEC/SIGSOFT FSE*, 935–947. ACM.

Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; Tufano, M.; Deng, S. K.; Clement, C. B.; Drain, D.; Sundaresan, N.; Yin, J.; Jiang, D.; and Zhou, M. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *ICLR*. OpenReview.net.

Han, Z.; Gao, C.; Liu, J.; Zhang, J.; and Zhang, S. Q. 2024. Parameter-Efficient Fine-Tuning for Large Models: A Comprehensive Survey. *CoRR*, abs/2403.14608.

Houlsby, N.; Giurgiu, A.; Jastrzebski, S.; Morrone, B.; de Laroussilhe, Q.; Gesmundo, A.; Attariyan, M.; and Gelly, S. 2019. Parameter-Efficient Transfer Learning for NLP. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, 2790–2799. PMLR.

Hu, E. J.; Shen, Y.; Wallis, P.; Allen-Zhu, Z.; Li, Y.; Wang, S.; Wang, L.; and Chen, W. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*. OpenReview.net.

Hu, X.; Li, G.; Xia, X.; Lo, D.; and Jin, Z. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empir. Softw. Eng.*, 25(3): 2179–2217.

Husain, H.; Wu, H.; Gazit, T.; Allamanis, M.; and Brockschmidt, M. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR*, abs/1909.09436.

Hyeon-Woo, N.; Ye-Bin, M.; and Oh, T. 2022. FedPara: Low-rank Hadamard Product for Communication-Efficient Federated Learning. In *ICLR*. OpenReview.net.

Jiang, N.; Lutellier, T.; and Tan, L. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *ICSE*, 1161–1173. IEEE.

Jin, M.; Shahriar, S.; Tufano, M.; Shi, X.; Lu, S.; Sundaresan, N.; and Svyatkovskiy, A. 2023. InferFix: End-to-End Program Repair with LLMs. In *ESEC/SIGSOFT FSE*, 1646–1656. ACM.

Kopiczko, D. J.; Blankevoort, T.; and Asano, Y. M. 2024. VeRA: Vector-based Random Matrix Adaptation. In *ICLR*. OpenReview.net.

Lester, B.; Al-Rfou, R.; and Constant, N. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *EMNLP (1)*, 3045–3059. Association for Computational Linguistics.

Li, X. L.; and Liang, P. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *ACL/IJCNLP (1)*, 4582–4597. Association for Computational Linguistics.

Lin, C.; and Och, F. J. 2004. ORANGE: a Method for Evaluating Automatic Evaluation Metrics for Machine Translation. In *COLING*.

Liu, H.; Tam, D.; Muqeeth, M.; Mohta, J.; Huang, T.; Bansal, M.; and Raffel, C. 2022a. Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning. In *NeurIPS*.

Liu, X.; Ji, K.; Fu, Y.; Tam, W.; Du, Z.; Yang, Z.; and Tang, J. 2022b. P-Tuning: Prompt Tuning Can Be Comparable to Fine-tuning Across Scales and Tasks. In *ACL (2)*, 61–68. Association for Computational Linguistics.

Mastropaolo, A.; Scalabrino, S.; Cooper, N.; Nader-Palacio, D.; Poshyvanyk, D.; Oliveto, R.; and Bavota, G. 2021. Studying the Usage of Text-To-Text Transfer Transformer to Support Code-Related Tasks. In *ICSE*, 336–347. IEEE.

Papineni, K.; Roukos, S.; Ward, T.; and Zhu, W. 2002. Bleu: a Method for Automatic Evaluation of Machine Translation. In *ACL*, 311–318. ACL.

Pfeiffer, J.; Kamath, A.; Rücklé, A.; Cho, K.; and Gurevych, I. 2021. AdapterFusion: Non-Destructive Task Composition for Transfer Learning. In *EACL*, 487–503. Association for Computational Linguistics.

Pfeiffer, J.; Rücklé, A.; Poth, C.; Kamath, A.; Vulic, I.; Ruder, S.; Cho, K.; and Gurevych, I. 2020a. AdapterHub: A Framework for Adapting Transformers. In *EMNLP (Demos)*, 46–54. Association for Computational Linguistics.

Pfeiffer, J.; Vulic, I.; Gurevych, I.; and Ruder, S. 2020b. MAD-X: An Adapter-Based Framework for Multi-Task Cross-Lingual Transfer. In *EMNLP (1)*, 7654–7673. Association for Computational Linguistics.

Ponti, E. M.; Sordoni, A.; and Reddy, S. 2022. Combining Modular Skills in Multitask Learning. *CoRR*, abs/2202.13914.

Raffel, C.; Shazeer, N.; Roberts, A.; Lee, K.; Narang, S.; Matena, M.; Zhou, Y.; Li, W.; and Liu, P. J. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.*, 21: 140:1–140:67.

Ren, S.; Guo, D.; Lu, S.; Zhou, L.; Liu, S.; Tang, D.; Sundaresan, N.; Zhou, M.; Blanco, A.; and Ma, S. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *CoRR*, abs/2009.10297.

Rumelhart, D. E.; Hinton, G. E.; and Williams, R. J. 1986. Learning representations by back-propagating errors. *nature*, 323(6088): 533–536.

Shi, E.; Wang, Y.; Du, L.; Chen, J.; Han, S.; Zhang, H.; Zhang, D.; and Sun, H. 2022. On the Evaluation of Neural Code Summarization. In *ICSE*, 1597–1608. ACM.

Shi, E.; Wang, Y.; Zhang, H.; Du, L.; Han, S.; Zhang, D.; and Sun, H. 2023. Towards Efficient Fine-Tuning of Pretrained Code Models: An Experimental Study and Beyond. In *ISSTA*, 39–51. ACM.

Thapa, C.; Jang, S. I.; Ahmed, M. E.; Camtepe, S.; Pieprzyk, J.; and Nepal, S. 2022. Transformer-Based Language Models for Software Vulnerability Detection. In *ACSAC*, 481–496. ACM.

Tian, H.; Liu, K.; Kaboré, A. K.; Koyuncu, A.; Li, L.; Klein, J.; and Bissyandé, T. F. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *ASE*, 981–992. IEEE.

Tufano, M.; Watson, C.; Bavota, G.; Penta, M. D.; White, M.; and Poshyvanyk, D. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *ASE*, 832–837. ACM.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L.; and Polosukhin, I. 2017. Attention is All you Need. In *NIPS*, 5998–6008.

Wan, Y.; He, Y.; Bi, Z.; Zhang, J.; Zhang, H.; Sui, Y.; Xu, G.; Jin, H.; and Yu, P. S. 2024. Deep Learning for Code Intelligence: Survey, Benchmark and Toolkit. *CoRR*, abs/2401.00288.

Wang, Y.; Le, H.; Gotmare, A.; Bui, N. D. Q.; Li, J.; and Hoi, S. C. H. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *EMNLP*, 1069–1088. Association for Computational Linguistics.

Watson, C.; Tufano, M.; Moran, K.; Bavota, G.; and Poshyvanyk, D. 2020. On learning meaningful assert statements for unit test cases. In *ICSE*, 1398–1409. ACM.

Wei, B.; Li, G.; Xia, X.; Fu, Z.; and Jin, Z. 2019. Code Generation as a Dual Task of Code Summarization. In *NeurIPS*, 6559–6569.

Xia, C. S.; Wei, Y.; and Zhang, L. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *ICSE*, 1482–1494. IEEE.

Yamaguchi, F.; Golde, N.; Arp, D.; and Rieck, K. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *IEEE Symposium on Security and Privacy*, 590–604. IEEE Computer Society.

Yeh, S.; Hsieh, Y.; Gao, Z.; Yang, B. B. W.; Oh, G.; and Gong, Y. 2024. Navigating Text-To-Image Customization: From LyCORIS Fine-Tuning to Model Evaluation. In *ICLR*. OpenReview.net.

Yusuf, I. N. B.; Jamal, D. B. A.; and Jiang, L. 2023. Automating Arduino Programming: From Hardware Setups to Sample Source Code Generation. In *MSR*, 453–464. IEEE.

Yusuf, I. N. B.; Jiang, L.; and Lo, D. 2022. Accurate generation of trigger-action programs with domain-adapted sequence-to-sequence learning. In *ICPC*, 99–110. ACM.

Zan, D.; Chen, B.; Zhang, F.; Lu, D.; Wu, B.; Guan, B.; Wang, Y.; and Lou, J. 2023. Large Language Models Meet NL2Code: A Survey. In *ACL (1)*, 7443–7464. Association for Computational Linguistics.

Zheng, Z.; Ning, K.; Wang, Y.; Zhang, J.; Zheng, D.; Ye, M.; and Chen, J. 2023. A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends. *CoRR*, abs/2311.10372.

Reproducibility Checklist

This paper:

- Includes a conceptual outline and/or pseudocode description of AI methods introduced **(yes)**
- Clearly delineates statements that are opinions, hypothesis, and speculation from objective facts and results **(yes)**
- Provides well marked pedagogical references for less-familiar readers to gain the background necessary to replicate the paper **(yes)**

Does this paper make theoretical contributions? (no)
If yes, please complete the list below.

- All assumptions and restrictions are stated clearly and formally. (yes/partial/no)
- All novel claims are stated formally (e.g., in theorem statements). (yes/partial/no)
- Proofs of all novel claims are included. (yes/partial/no)
- Proof sketches or intuitions are given for complex and/or novel results. (yes/partial/no)
- Appropriate citations to theoretical tools used are given. (yes/partial/no)
- All theoretical claims are demonstrated empirically to hold. (yes/partial/no/NA)
- All experimental code used to eliminate or disprove claims is included. (yes/no/NA)

Does this paper rely on one or more datasets? (yes)
If yes, please complete the list below.

- A motivation is given for why the experiments are conducted on the selected datasets **(yes)**

- All novel datasets introduced in this paper are included in a data appendix. **(NA)**
- All novel datasets introduced in this paper will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. **(NA)**
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are accompanied by appropriate citations. **(yes)**
- All datasets drawn from the existing literature (potentially including authors' own previously published work) are publicly available. **(yes)**
- All datasets that are not publicly available are described in detail, with an explanation of why publicly available alternatives are not scientifically satisfactory. **(NA)**

Does this paper include computational experiments? (yes)

If yes, please complete the list below.

- Any code required for pre-processing data is included in the appendix. **(yes)**
- All source code required for conducting and analyzing the experiments is included in a code appendix. **(yes)**
- All source code required for conducting and analyzing the experiments will be made publicly available upon publication of the paper with a license that allows free usage for research purposes. **(yes)**
- All source code implementing new methods has comments detailing the implementation, with references to the paper where each step comes from **(partial)**
- If an algorithm depends on randomness, then the method used for setting seeds is described in a way sufficient to allow replication of results. **(yes)**
- This paper specifies the computing infrastructure used for running experiments (hardware and software), including GPU/CPU models; amount of memory; operating system; names and versions of relevant software libraries and frameworks. **(yes)**
- This paper formally describes evaluation metrics used and explains the motivation for choosing these metrics. **(yes)**
- This paper states the number of algorithm runs used to compute each reported result. **(yes)**
- Analysis of experiments goes beyond single-dimensional summaries of performance (e.g., average; median) to include measures of variation, confidence, or other distributional information. **(yes)**
- The significance of any improvement or decrease in performance is judged using appropriate statistical tests (e.g., Wilcoxon signed-rank). **(partial)**
- This paper lists all final (hyper-)parameters used for each model/algorithm in the paper's experiments. **(yes)**
- This paper states the number and range of values tried per (hyper-) parameter during the development of the paper, along with the criterion used for selecting the final parameter setting. **(yes)**