Check for updates

AutoAdapt: On the Application of AutoML for Parameter-Efficient Fine-Tuning of Pre-Trained Code Models

AMAL AKLI, SnT, University of Luxembourg, Luxembourg
MAXIME CORDY, SnT, University of Luxembourg, Luxembourg
MIKE PAPADAKIS, SnT, University of Luxembourg, Luxembourg
YVES LE TRAON, SnT, University of Luxembourg, Luxembourg

Large Language Models (LLMs) have demonstrated their ability to solve tasks across various domains, including software engineering. However, their extensive number of parameters makes full fine-tuning computationally prohibitive. While Parameter-efficient fine-tuning (PEFT) methods, such as adapter fine-tuning, have been proposed to address this issue; yet, they typically employ default configurations that use the same adapter settings across all layers. Concurrently, Automated Machine Learning (AutoML) has demonstrated success in hyperparameter optimization, while Neural Architecture Search (NAS) has proven effective in optimizing neural network architectures. Building on these successes, we introduce AutoAdapt, a novel approach that leverages NAS to automatically discover task-specific, layer-wide adapter configurations, allowing each layer to adopt distinct adapter parameters. AutoAdapt defines a search space tailored for adapter-based fine-tuning and employs an evolutionary algorithm to explore a diverse range of configurations, thereby evaluating the benefits of customizing each layer individually. We evaluate AutoAdapt on well-established software engineering tasks, including vulnerability detection, code clone detection, and code search. Our empirical results demonstrate that AutoAdapt outperforms manually engineered adapter configurations, achieving up to a 5% improvement in F1-score for clone detection and defect detection, and up to a 25% improvement in MRR for code search. Additionally, it surpasses other PEFT techniques, such as Prefix Tuning and LoRA. Furthermore, AutoAdapt is capable of identifying configurations that outperform even full fine-tuning, while training less than 2.5% of the model parameters. A comprehensive analysis reveals that factors such as selective layer adaptation, module selection (e.g., attention versus feed-forward layers), normalization, and dropout significantly influence performance across different tasks. Additionally, our findings suggest the possibility of transferring adapter configurations to similar datasets and tasks, thus simplifying the search for optimal PEFT settings. Our code and data are available for access at: https://github.com/serval-uni-lu/AutoAdapt

CCS Concepts: • **Computing methodologies** → **Genetic algorithms**; **Transfer learning**; **Neural networks**; **Search methodologies**; *Neural networks*; • **Software and its engineering** → *Software defect analysis*; *Software verification and validation*; • **Information systems** → *Information retrieval*.

Additional Key Words and Phrases: PEFT, pre-trained code models, Optimization, AutoML, NAS

1 INTRODUCTION

Large Language Models (LLMs) are increasingly employed across various fields, primarily through the application of transfer learning specializing LLMs to downstream tasks [7]. Typically, pre-trained models are fine-tuned by updating all their parameters on a target dataset for a specific downstream task (full fine-tuning). However,

Authors' Contact Information: Amal AKLI, amal.akli@uni.lu, SnT, University of Luxembourg, Luxembourg; Maxime cordy, maxime.cordy@uni.lu, SnT, University of Luxembourg, Luxembourg; Mike Papadakis, michail.papadakis@uni.lu, SnT, University of Luxembourg, Luxembourg; Yves Le Traon, yves.letraon@uni.lu, SnT, University of Luxembourg, Luxembourg.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/5-ART
https://doi.org/10.1145/3734867

Just Accepted

this process is computationally more expensive as LLMs get larger, and can become prohibitive for models encompassing billions of parameters, or more [4]. Moreover, full fine-tuning leads to individual sets of fine-tuned model parameters for each task, which becomes prohibitively costly in terms of memory when deploying models for multiple tasks, since we need to save all model parameters.

To address this challenge, Parameter-Efficient Fine-Tuning (PEFT) methods have been proposed [13, 17, 18, 24], also called "*Delta Tuning*". These methods often involve freezing the pre-trained model's parameters and introducing a few additional trainable parameters within the transformer layers [39]. One such method introduces modules known as "*Adapters*". They have demonstrated significant success in natural language processing and are increasingly applied to other fields, such as code intelligence tasks. For example, Liu et al. conducted an empirical study by fine-tuning state-of-the-art code models using PEFT methods for both code understanding and code generation tasks. Their findings indicate that PEFT methods can achieve performance comparable to or even better than full fine-tuning of code models with just 0.5% additional parameters [26].

Adapters are typically added to all layers of a pre-trained model, positioned between the dense layer and the normalization layer within the feed-forward modules of the transformer layer [17]. Recent studies [11, 45] have demonstrated that it is not necessary to insert adapters into all layers of large language models (LLMs). Moreover, the size of the adapter significantly influences performance, with larger adapters generally improving model performance. However, this improvement comes at the cost of increased computational overhead, particularly when adapters are inserted into every layer. This highlights the need for further optimization in adapter design choices, raising the question of how to automatically identify the optimal adapter configurations for a given model and task while minimizing the size of the added parameters as much as possible.

Meanwhile, Automated Machine Learning (AutoML), and notably Neural Architecture Search (NAS) methods, are increasingly employed to produce and optimize various types of neural network models, including LLMs. Tornede et al. even proposed that LLM fine-tuning could benefit from NAS techniques to design adapter architectures and optimize their placement. Hu et al. followed this idea and proposed DARTS, a NAS method that searches among a mixture of PEFT methods, whereas Zhou et al. employed Bayesian optimization. These methods can effectively recommend, for Natural Language Processing (NLP) applications, which layers to adapt, which adapter size to insert, and the combination of different PEFT modules, for example, Adapter [17], and Lora [18]. However, they don't take into account the idea of using different configurations for each layer based on the unique information it encodes.

While these advances in automated PEFT for NLP are promising, the application of NAS methods to fine-tune pre-trained code models remains underexplored. Existing approaches often constrain the search space to a single adapter architecture that is uniformly applied across all layers. However, we hypothesize that this uniformity may overlook important nuances. LLM layers are known to capture distinct lexical and semantic information [40], suggesting that each layer may benefit from unique adaptations during fine-tuning. In contrast to natural language, source code follows strict syntactic rules and hierarchical structures, with explicit semantic information tied to its functionality through elements such as variable names and function signatures. These characteristics highlight the need for more tailored, layer-specific adaptations. Failing to account for these layer-specific differences could represent a missed opportunity in leveraging NAS to its full potential for PEFT in code models. This raises the question of whether a NAS method designed with a broader search space that considers distinct configurations for individual layers could further enhance the effectiveness of PEFT methods.

To address this question, we propose and implement an evolutionary NAS method [6] that explores a specifically designed search space for adapter fine-tuning, allowing the discovery of optimal adapter configurations for the different LLM layers.

We evaluate our technique in various models, including CodeBERT [8], GraphCodeBERT [10], and UniXcoder [9], which are based on the RoBERTa architecture [27], as well as CodeT5 [43] and CodeT5+ [42], which are

Just Accepted

built on the T5 architecture [33]. The evaluation is carried out on multiple tasks from CodeXGLUE [1], including vulnerability detection using the Devign dataset [46], clone detection with the BigCloneBench dataset [37], and code search using the AdvTest dataset [2], which comes from the CodeSearchNet dataset [22]. This evaluation enables us to answer our first research question:

RQ1: Can AutoAdapt reach higher effectiveness than manually configured adapter modules?

Findings: Our experiments show that AutoAdapt significantly improves fine-tuning performance. Specifically, we observed up to a 5% improvement in the F1-score for both clone detection and defect detection, and up to a 25% improvement in MRR for code search when compared to the standard configuration. Notably, AutoAdapt even surpasses the performance of full fine-tuning while keeping the proportion of trainable parameters minimal (less than 2.5%). These findings underscore the potential of AutoML to effectively enhance PEFT methods.

Building on the positive outcomes of RQ1, in RQ2 we investigate the key factors influencing these results, analyzing performance variations across different models and tasks to derive meaningful recommendations for further optimization. We ask:

RQ2: Do specific architecture design choices yield an improvement across all datasets and tasks?

Findings: Our analysis shows that performance gains depend on the task and model architecture. Strategic adapter placement at attention and feed-forwad sub-modules, larger adapter sizes, and adding normalization layers benefit specific tasks like clone detection and CodeT5-based models, while adding dropout aids tasks with smaller datasets like Devign, highlighting the need for task- and model-specific adapter designs.

To further reduce the need for task-specific optimizations, we investigate whether an adapter optimized for one task can effectively transfer to a similar task. Our final contribution addresses this question:

RQ3: Can adapter configurations and learned parameters effectively transfer between different code understanding tasks?

Findings: Our findings on transferability suggest that configurations identified by AutoAdapt can be effectively applied or used as a starting point for other datasets with similar tasks. This approach significantly reduces the time required to search for optimal PEFT settings for each dataset.

Our contributions can be summarized as follows:

- We investigate the application of AutoML to optimize adapter design within pre-trained code models. Our results reveal that it is possible to automatically discover adapter configurations that surpass existing, carefully crafted adapters, and even full fine-tuning.
- We show that our AutoAdapt method can discover an adapter configuration using less then 2% of model parameters that surpasses full fine-tuning.
- We demonstrate that different layers of the pre-trained model may require distinct adaptations and that adapting only a few layers can be sufficient to achieve results comparable to full fine-tuning.
- We demonstrate that specific design choices bring improvements across multiple code understanding models and tasks. These results can inspire new research in adapter design through the identification of promising configuration elements.
- We show that the transferability of adapter configurations is effective between similar tasks. This provides good practices for researchers aiming to reuse adapters across different tasks.

2 RELATED WORK

2.1 Pre-trained Code Models

As we transitioned into a period marked by advances in computing, sophisticated machine learning methodologies, and abundant data availability, LLMs have emerged as prominent models in recent years. Typically built

[1] https://github.com/microsoft/CodeXGLUE

[2] https://paperswithcode.com/sota/code-search-on-codexglue-advtest

Just Accepted

on a transformer architecture [39] and trained with vast amounts of data, these models have demonstrated an impressive ability to replicate human linguistic capabilities. Their significant impact has been observed across various domains, with notable contributions seen in fields such as Software Engineering (SE) [16]. In the realm of large language models, including code LLMs, three key architectural types stand out [16] : Encoder-only models capture word relationships and sentence context. Notable examples include CodeBERT [8], GraphCodeBERT [10], RoBERTa [27], and ALBERT [23]. Encoder-decoder models use the encoder to create a hidden space for the decoder to generate output text, as seen in PLBART [2], T5 [34], UniXcoder [9], CodeT5 [43], and codeT5+ [42]. Meanwhile, decoder-only models, like the GPT series (GPT-1, GPT-2, GPT-3, GPT-4) [1, 5, 31, 32], directly produce output text using the decoder module. These large models are utilized for specific downstream tasks by fine-tuning them on the target dataset.

2.2 Parameter-Efficient Fine-Tuning

Pre-trained models are initially trained on extensive datasets across various tasks to acquire general knowledge. Subsequently, they are fine-tuned on a target dataset to adapt and specialize for specific new tasks. Fine-tuning involves initializing the model with parameters learned during the pre-training phase and then making slight adjustments without starting the training process from scratch. Typically, all parameters are updated during full fine-tuning of the model. However, this approach can be computationally intensive due to the growing size of LLMs, which can comprise hundreds of billions of parameters [4].

To address this challenge, Parameter-Efficient Fine-Tuning methods (PEFT) have been proposed [12]. This approach involves training a small set of additional parameters while keeping all the pre-trained parameters frozen. PEFT techniques vary in their approaches to introducing additional parameters and determining their size. Adapter fine-tuning [17] adds modules within the Feed-Forward layer of the transformer model [39], which can be either serial [17] or parallel [14]. Prefix tuning [24] introduces trainable prefix vectors to the keys and values in the multi-head attention of the transformer layer. In contrast, the LoRA technique [18] injects trainable low-rank decomposition matrices into the attention module of the transformer. Other methods employ hybrid approaches, such as MHM [13], which combines prefix tuning with parallel adapters.

Recent studies have shown the efficiency of PEFT techniques in fine-tuning code LLMs [21, 26, 41]. Liu et al. demonstrated that by adjusting just 0.5% additional parameters, PEFT methods can achieve performance comparable to or even better than full fine-tuning in code understanding tasks.

2.2.1 Adapters. Adapter fine-tuning has demonstrated effectiveness, particularly in the context of code LLMs [26]. Adapters are small FFN blocks inserted within the transformer layer, typically consisting of a down-projection matrix $\mathrm{W}_{\mathrm{down}}$, followed by a non-linear activation function $\sigma(.)$, and an up-projection matrix W_{up} . Given a hidden input vector h, the output of the adapter is:

$$Z = \mathrm{W}_{\mathrm{up}}(\sigma(\mathrm{W}_{\mathrm{down}}(h))) + h \tag{1}$$

$$W_{\mathrm{Up}} \in \mathbb{R}^{r \times d} \text{ and } W_{\mathrm{down}} \in \mathbb{R}^{d \times r}$$

Where d is the dimension of the hidden layer of the transformer, and r is the dimension of the adapter.

Typically, this adapter adopts the bottleneck architecture, where the only varied parameter is the size of the bottleneck. It is inserted into all the layers of the pre-trained model at a fixed placement, between the linear layers and normalization layers of feed-forward modules in the transformer layer, as shown in Figure 1.

2.2.2 PEFT frameworks. To facilitate the implementation of PEFT techniques, some frameworks have been developed to simplify their application. OpenDelta [19] is a notable open-source framework [3] that accommodates

[3]https://github.com/thunlp/OpenDelta

Just Accepted

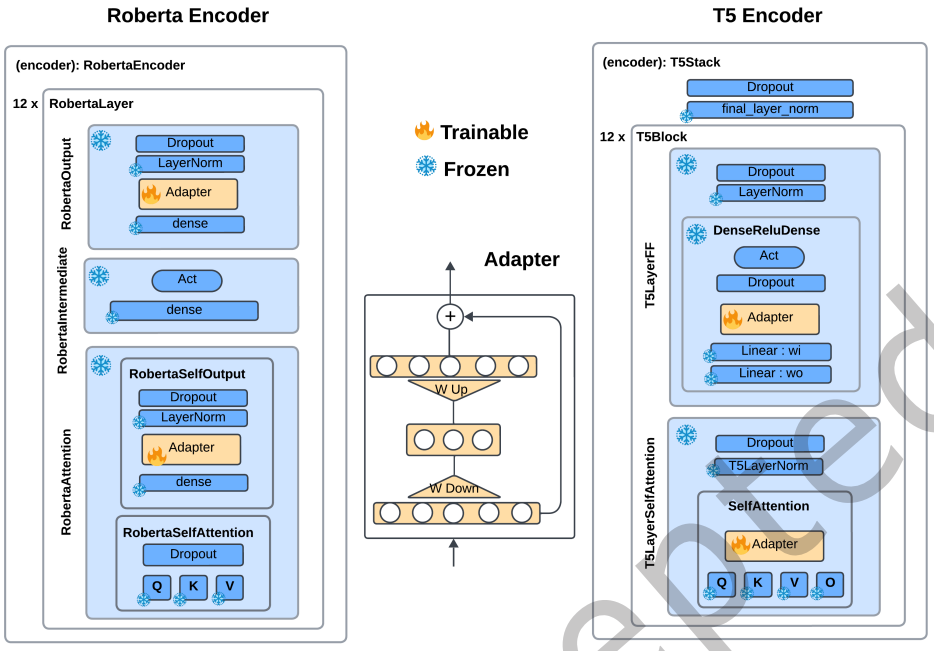


Fig. 1. Architecture of adapters integrated into Roberta and codeT5 encoders for parameter-efficient fine-tuning. The frozen components are shown in blue, while trainable components are highlighted with a flame icon. The adapters, consisting of a bottleneck architecture with learnable parameters (W_up and W_down), are added to the attention and feed-forward layers.

a variety of PEFT methods, including prefix tuning, adapters, LoRA, and other delta tuning strategies, across any chosen pre-trained model. With its extensive configurability, adapting to new models or adjusting parameters for specific techniques becomes more straightforward. Mangrulkar et al. has developed PEFT [29], an integrated package with HuggingFace Transformers [4] aimed at enhancing both model training and inference processes. AdapterHub [30], is a widely recognized framework for transformer adaptation, is built upon the HuggingFace Transformers framework [5]. It facilitates the direct attachment of trained adapters to pre-trained models.

2.3 Enhancing PEFT methods using AutoML techniques

Recent studies have aimed to optimize and enhance the efficiency of PEFT techniques. Guo et al. introduced a framework that automatically selects layers for adaptation by learning gates that activate or deactivate adapters. Their findings suggest that not all layers of LLMs require adaptation, indicating that adapters do not need to be inserted into every layer [11].

Building on the success of Neural Architecture Search (NAS), a subfield of automated machine learning that automates the design of neural networks using optimization methods like reinforcement learning [47], evolutionary optimization [6], and gradient descent [25], researchers have explored its application to optimize PEFT modules. For instance, Hu et al. proposed S3Delta, which adapts Differentiable Architecture Search (DARTS) [25] to automatically search optimal trainable structure [20]. They introduced a unified search space with

[4]https://huggingface.co/blog/peft
[5]https://adapterhub.ml/

Just Accepted

probabilistic gating to enable exploration among a combination of PEFT methods. In line with this approach, Zhou et al. introduced AutoPEFT [45], a framework that employs Bayesian optimization to search for optimal PEFT configurations. This includes optimizing the combination of various PEFT methods, determining the bottleneck dimension for PEFT module, and selecting the appropriate layers for its insertion.

In the same context, Tornede et al. explored the challenges and potential benefits of using AutoML to enhance LLMs. They highlighted the promise of employing Neural Architecture Search for fine-tuning LLMs. Utilizing NAS methods to select and optimize the appropriate layers for adapter insertion, as well as configuring the architecture of PEFT modules, could be valuable for improving downstream performance [38].

Building on this perspective, our study aims to investigate the use of NAS techniques to optimize adapter configurations uniquely for each layer of the pre-trained code model. The next section describes the proposed approach, referred to as *AutoAdapt*.

3 AUTOADAPT

Previous research [40] has demonstrated that distinct semantic and lexical information is stored across different layers of PLMs. In this study, we aim to explore whether each layer of a pre-trained model similarly requires different adaptation by varying adapter configurations and placement in each layer. We aim to explore optimal adapter composition using an evolutionary method within neural architecture search [6]. Evolutionary methods are particularly suited for navigating the discrete and complex search space of adapter configurations, which includes varying layers, parameters, and modules. Unlike gradient-based or reinforcement learning approaches, evolutionary algorithms are computationally efficient and robust when dealing with non-differentiable and highly structured search spaces. Furthermore, our objective includes examining the top returned models from the optimization algorithm. This analysis aims to understand the impact of inserting adapters in various layers and modules and varying their parameters. Through this study, we intend to identify and discuss what seems to be the best choice for adapter fine-tuning design for the code understanding tasks under investigation.

In Figure 2, we present an overview of our adapter optimization method, AutoAdapt, using a refined evolutionary approach. This method involves a search space comprising crucial components for constructing candidate solutions, such as layers, layer sizes, activation functions, insertion modules, and other adapter fine-tuning settings. Subsequently, a constrained population of candidate solutions is derived from this space, and a standard regularized evolutionary algorithm is employed to identify the best individual. In the following sections, we will provide detailed explanations of each component, including the design of the search space, generation of random adapter structures, insertion process into pre-trained models, and optimization steps.

3.1 Encoding scheme

To execute the evolutionary algorithm, it is essential to establish a common structure for the candidate solutions. The candidate solutions we explore within the defined search space can be encoded as follows: each individual is depicted by a vector that matches the number of layers in the pre-trained model. At each index (i) in this vector, corresponding to the i-th layer in the model, there exists a dictionary containing the parameters of the adapter intended for insertion at that layer. These parameters include the insert modules, the sizes of the feed-forward neural network (FFN) linear layers, and the nonlinearity function. To denote a decision of not inserting an adapter at a certain layer, we assign the value 0 at the corresponding index in the vector. We show an example in Figure 3. In this example, we insert two adapters into the first layer of the model. The first adapter is added to the "attention.self" module with a projection size of 32, while the second adapter is inserted into the "output" module with a projection size of 128. Both adapters use the same activation function, "ReLU," includes a dropout layer with a rate of 0.1, and does not incorporate a normalization layer within the adapter. It is worth noting that the OpenDelta framework [19] inserts the adapter module sequentially at the end of the modified module.

Just Accepted

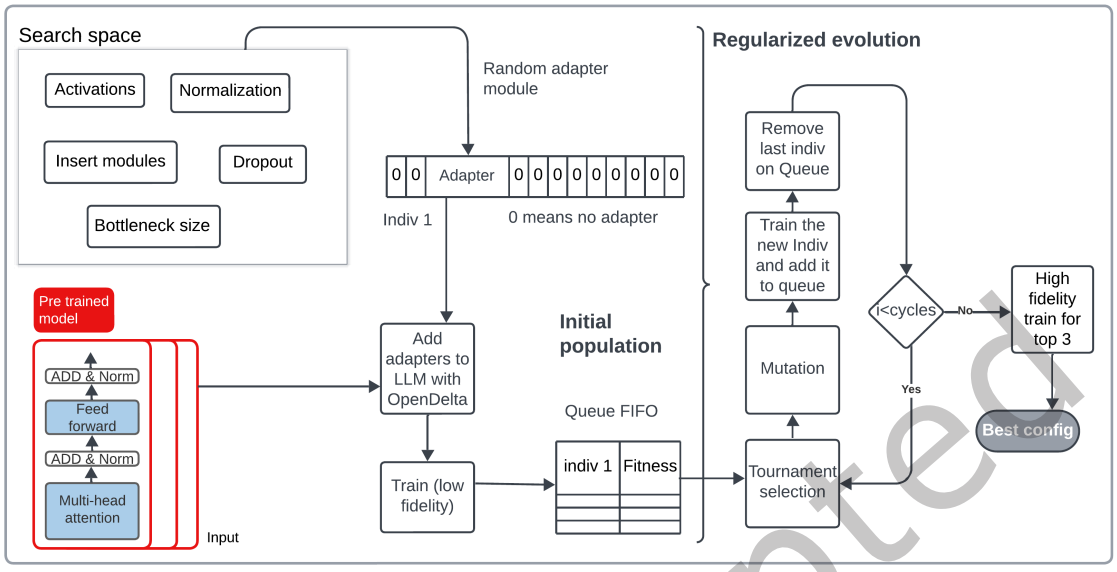


Fig. 2. A general overview of AutoAdapt: random adapters are constructed from the defined search space using the OpenDelta library, added to pre-trained models, and included in the population. The optimization is conducted using regularized evolution to find the optimal adapter structure.

Just Accepted

3.2 Search space

Inspired by the achievements of neural architecture search [47], we embark on a similar path by meticulously crafting a configuration space.

In the baseline adapter design [17], the adapter comprises two linear layers: a down-projection to a smaller dimension, followed by a non-linear activation, and an up-projection back to the original dimension. It is integrated directly after the feedforward layer within the transformer layer. The output of the adapter is subsequently passed through the normalization of the transformer layer. This adapter configuration is either applied uniformly across all layers of the model or selectively to specific layers [11, 45].

In this study, we retain the bottleneck architecture commonly employed in prior works, as adding additional layers introduces more complex transformations to the input, which may not always be beneficial. Moreover, such additions increase the model's parameter count, contradicting the principles of PEFT. However, unlike previous works, we explore the potential inclusion of a dropout layer and a normalization layer within the adapter. Both components, drawn from the original transformer architecture, introduce additional flexibility and hold promise for enhancing the adapter design.

We aim to generate adapters with varying components as defined in the search space illustrated in Figure 3. These variations include the placements where to insert the adapter, projection size, activation functions, the inclusion of normalization layers, and the addition of dropout layers.
Pre-trained model architectures differ significantly between families, particularly in their underlying structures and module naming conventions. As a result, the search space must be adapted accordingly. To maintain

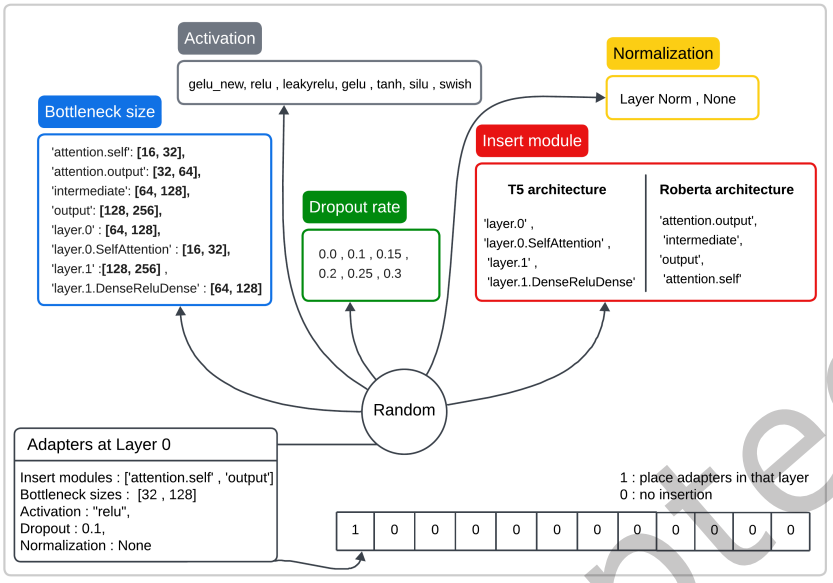


Fig. 3. Overview of the Search Space for AutoAdapt. The search space includes configurations for bottleneck sizes, activation functions, dropout rates, normalization layers, and module insertion points for both T5 and RoBERTa architectures. Each component is selected randomly to explore various adapter designs, using a random binary list whose size equals the number of model layers, to specify adapter placement in specific layers.

consistency and relevance, we have limited our investigation to the RoBERTa [27] and T5 [33] architecture families.

For the insertion locations within the pre-trained model layers, we consider four possible placements, specifically at the end of all main sub-modules in the transformer layer. For each placement, we define a range of possible bottleneck sizes and ensure that each adapter's bottleneck size appropriately aligns with its location. For attention sub-modules, smaller bottleneck sizes (16, 32) often suffice, as attention heads already capture a significant amount of task-specific information. In contrast, for intermediate and output sub-modules, medium or large bottlenecks (128, 256) are more suitable, as these layers handle feature transformations. For activation functions, we consider those commonly employed in pre-trained models as listed in Figure 3. Regarding dropout rates, we explore values ranging from 0 to 0.3, avoiding higher rates as they may cause excessive information loss. Additionally, we evaluate the inclusion or exclusion of a layer normalization step following the non-linear activation function.

3.3 Search method

We aim to generate random adapter configurations and then refine them using a regularized evolutionary algorithm [36], as it can search for efficient and performant models with less search time. This algorithm initiates by constructing an initial population of adapter configurations and evaluates the corresponding delta models (PEFT models) with the defined fitness function. Following this, a selection process identifies the best-performing individuals within the population. Subsequently, mutations are applied to diversify the population, ensuring the exploration of a broader solution space. After mutation, a regularization mechanism is introduced to maintain diversity and prevent premature convergence. The population is then updated with the best individuals, ensuring

Just Accepted

continuous improvement. This iterative process, spanning multiple cycles, facilitates the optimization of adapter configurations by iteratively refining and evolving the population towards superior solutions.

3.3.1 Initialization. The description of an individual Ii (genotype) within the population comprises three essential components of information $(Ii : x1, x2, x3, x4, x5) : x1$ represents the modified modules,$x2$ denotes the bottleneck sizes, $x3$ indicates the non-linearity activation function, $x4$ represents the dropout rate, and $x5$ determines whether a normalization layer is included. We show an example in Figure 3.

The population initialization process involves the generation of random combinations of each parameter previously indicated to form random genotypes. Subsequently, each genotype is mapped to a delta model (model with adapters inserted), which is evaluated by training it on the target dataset.

3.3.2 fitness evaluation . The fitness function is used to evaluate and rank candidate solutions. In our case, it measures both the performance and efficiency of the adapters integrated into the model. The fitness is computed on the validation dataset after training. Our objective is to achieve high-performing models while having less trainable parameters. We define the fitness function as follows:

$$f(x) = \text{Accuracy} + \text{F1-score} - \text{Rate of trainable parameters}$$

We consider both accuracy and F1-score. While accuracy can be misleading in imbalanced datasets or early training stages, the F1-score provides a more robust measure by balancing precision and recall. We subtract the rate of trainable parameters, which is capped at 6% in our evaluation. This reflects our emphasis on the adapter's performance while giving less importance to its size.
The candidate models are initially trained on a training dataset using a low-fidelity approach, involving reduced data and fewer training epochs. The top three performing models are then evaluated at the end of the search with the full dataset and extended training epochs to select the best candidate. This strategy is adopted due to the considerable time required for training large language models on the complete dataset for each candidate. Furthermore, we implement an early stopping method based on the validation loss, halting evaluations predicted to perform poorly on the validation set.

3.3.3 Parent selection. During each generation of the evolutionary algorithm, a parent must be chosen from the current population for genetic operations to produce a child individual. This study employs the tournament selection method [35]. The parent selection process proceeds as follows: A set number of individuals are randomly sampled from the current population to participate in a tournament. These individuals compete based on their fitness, representing the performance of the delta model in our context. The individual demonstrating the highest fitness is designated as the parent and genetic operators are applied to generate the new individual.

3.3.4 Genetic operations. For genetic operations, we employ both mutation and crossover as genetic operators for the chosen parents. To diversify the mutation process, we utilize a binary variable: a value of 0 indicates the swapping of two adapters in the list, while a value of 1 signifies the replacement of a randomly selected adapter configuration with a newly generated one. In the crossover operation, we combine genetic information from two parents to generate a single offspring. We establish a crossover point, and the adapter configurations between these points are exchanged between the parents to create a new solution. Once the new individual is generated, it is incorporated into the population (queue). Our approach, utilizing regularized evolution, introduces the concept of age into the population dynamics. Consequently, the oldest individual is removed from the population rather than the least fit one. This strategy helps maintain diversity within our population in successive iterations.

3.4 Inserting the adapters

After generating a random adapter configuration encoded as explained previously, the corresponding adapter modules will be integrated into the architecture of the pretrained model using the OpenDelta framework [19].

Just Accepted

The original OpenDelta implementation of adapter fine-tuning follows a bottleneck architecture [15], comprising a down projection layer, a non-linearity function, and an up projection layer to revert to the original hidden size of the vector. This structure is necessary as the vector is subsequently passed to the fixed parameters of the next pretrained module, requiring conformity to the existing fixed shapes. However, in our specific case, we aim to incorporate additional parameters, including multiple adapter sizes and configurations related to dropout and normalization. To achieve this, we modify the adapter functions defined in the *AdapterModel* class in the OpenDelta library and introduce additional parameters. Rather than specifying a single bottleneck size, representing the size of the down-projection layer, we now pass multiple sizes along with a dropout rate and a condition for whether to include a LayerNorm in the middle of the adapter. By default, adapters are automatically inserted into all attention and output layers across all layers of the language model. However, for specific requirements involving the insertion of adapters into particular modules within a given language model layer, no modifications to the framework are necessary. Instead, we can leverage the framework's existing parameter *modified_modules* to specify the exact path to the modules where the adapters should be inserted. This requires identifying the precise module name, such as *encoder.layer.x.module*, corresponding to the desired insertion point.

4 EXPERIMENTAL SETUP

4.1 Research questions

RQ1: Can AutoAdapt reach higher effectiveness than manually configured adapter modules? : We aim to explore optimal adapter configurations, including architecture, and placement within pre-trained models, using an evolutionary approach for an automated search. We examine diverse adapter configurations for each layer of the pre-trained code model, contrasting with standard setups using fixed adapter configurations across all layers. We aim to assess whether each layer requires a unique adaptation and if our automated search method can find the optimal choice.

RQ2: Do specific architecture design choices yield improvement for each dataset and task?: We explore various design choices for adapter fine-tuning, by analyzing the final configurations (best-performing setups) from each optimization run conducted on different datasets and models during the evaluation of RQ1, including insertion layers, insertion sub-modules, parameters size, number of adapters, the effect of normalization and dropout. Through assessment, we determine which factors have the greatest impact on the efficiency of adapter fine-tuning. This analysis aims to define the best design choices when employing adapter fine-tuning within pre-trained code models.

RQ3: Can adapter configurations and learned parameters effectively transfer between different code understanding tasks? : Our aim is to investigate the transferability of learned adapter configurations. Initially, we use the top two adapter configurations identified from the Unixocder model when trained on the defect detection dataset to test their generalizability across the other datasets, including *flakiness prediction* and *clone detection*, as well as *code search*.

4.2 Baselines

In this work, we conduct experiments on five pre-trained code models: CodeBERT[6], GraphCodeBERT[7], UniXcoder[8], CodeT5 [9], and codeT5+ [10]. We selected these models because they are based on the Roberta and T5 architectures, which are the primary focus of our code and research efforts.

[6]https://huggingface.co/microsoft/codebert-base
[7]https://huggingface.co/microsoft/graphcodebert-base
[8]https://huggingface.co/microsoft/unixcoder-base
[9]https://huggingface.co/Salesforce/codet5-base
[10]https://huggingface.co/Salesforce/codet5p-220m

Just Accepted

We compare the performance of the searched configurations to the standard full model fine-tuning and the following individual Parameter-Efficient Fine-Tuning (PEFT) module: LoRA [18], Adapter [17], and prefix tuning [24]. We keep the default parameters for these PEFT methods. For the adapter method, which serves as the main baseline for our automatic search approach, we include two configurations: one with the full default parameters of the OpenDelta library (bottleneck size of 24) and another with an increased bottleneck size of 64. This comparison aims to investigate whether simply increasing the adapter size enhances performance. The bottleneck size of 64 was selected to align with the sizes of the adapters identified through the AutoAdapt search process. All methods are trained and tested under identical conditions and datasets.

4.3 Implementation Settings

In line with previous research [26], we establish a learning rate of 1e-4 for PEFT techniques and maintain a rate of 5e-5 for complete fine-tuning. Throughout all fine-tuning trials, we utilize the Adam optimizer with a batch size of 32, maintain a consistent Transformer input window size of 512, and set the seed to 42. We train the models for up to 15 epochs, implementing early stopping based on validation loss. We stop training and assess test performance based on the validation performance.

For our optimization algorithm, we opt for a population size of 50 models, with a sample of 10 models per cycle to maximize diversity and conduct optimization over 30 cycles. The chosen sizes are kept minimal due to the significant time required for training each model. We train individuals within the population for 5 epochs, utilizing 10000 samples from the training data and 2500 samples from validation data as a lower-fidelity estimation. This size was chosen after testing various sample sizes and identifying the one that aligns well with the performance on the full dataset. It represents approximately 50% of our smaller datasets, namely the defect detection and flakiness detection datasets. To ensure fairness, this size is fixed across all other datasets. At the end of the optimization process, the top three configurations are evaluated on the full dataset to select the best.

The training and evaluation methodologies for the listed tasks are derived from the UnixCoder repository [11]. We execute all models on one Linux machine, equipped with an Intel(R) Xeon(R) Silver 4416+ Processor and four NVIDIA L40S GPUs.

4.4 Tasks and datasets

Detailed sample counts for each dataset are provided in Table 1.

4.4.1 Defect Detection: The defect detection task involves analyzing a code snippet to determine whether it might present risks to a software system [44], such as memory leaks or potential Denial of Service (DoS) attacks. We employ the Devign dataset [46] for our experimentation purposes. Devign [12] comprises code functions extracted from two prominent open-source C libraries (FFmpeg and Qemu). Each function in the dataset is meticulously labeled to denote whether it presents a security vulnerability.

4.4.2 Clone Detection: The goal of code clone detection is to identify if two code snippets have similar semantics. We use BigCloneBench [37] as our benchmark to assess code clone detection. This benchmark in CodeXGLUE [13] comprises a range of Java methods, featuring both cloned and unique pairs. For this dataset, we evaluate a subset due to its size, containing over 1,700,000 samples. Due to our limited computational resources, we utilize only a portion, as detailed in Table 1.

[11] https://github.com/microsoft/CodeBERT/tree/master/UniXcoder/downstream-tasks
[12] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection
[13] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Clone-detection-BigCloneBench/dataset

Just Accepted

4.4.3 Code search : Code search is the task of finding the most relevant code snippet from a large codebase based on a natural language query. For this task, we use the AdvTest dataset from CodeXGLUE [14], which is based on the BigBenchClone dataset . AdvTest is more challenging because it includes adversarial examples where function names and variable names in the test set are replaced with special tokens. This makes the task focus more on understanding the meaning of the code rather than relying on specific names.

4.4.4 Flaky tests prediction : Flakiness test prediction involves identifying whether a software test exhibits inconsistent behavior, such as passing or failing under identical conditions. The goal is to detect flaky tests, which are inherently unreliable and can hinder the software development process. In our experiments, we utilize the FlakeFlagger dataset [3], a well-known dataset in this field comprising 21,802 tests.

Table 1. Statistics of datasets

Task	dataset	language	#train	#validate	#test
Vulnerability detection	Devign	C	21854	2732	2732
Clone detection	BigCloneBench	JAVA	90102	41541	41541
Code search	AdvTest	PYTHON	251820	9604	19210
Flakiness prediction	FlakeFlagger	JAVA	13081	4360	4361

4.5 Evaluation metrics

In this study, we use evaluation metrics that have been previously employed in similar research contexts [26]. For a robust evaluation of model performance on the binary classification tasks, we use a comprehensive set of metrics, including Accuracy, Precision, Recall, and F1 score, to assess various aspects of predictive effectiveness. For the code search task, we utilize the Mean Reciprocal Rank (MRR) metric to evaluate retrieval performance. We express the computational costs of fine-tuning based on the number of trainable parameters. Instead of providing absolute values, this count is expressed as a percentage relative to the number of adapter parameters compared to the original model. This metric is referred to as “Trainable rate”.

5 RESULTS

Figure 4 shows the progression of average fitness across optimization cycles for each task. The steady improvement observed across all tasks indicates that the genetic algorithm successfully enhances adapter performance through its optimization process. The differences in improvement rates across tasks suggest that the algorithm effectively adapts to the unique characteristics of each dataset and model. Additional plots in the Appendix detail the evolution of the fitness standard deviation in Figure 9 and the count of unique configurations in Figure 10, further demonstrating the convergence of the algorithm toward optimal solutions. The search process required approximately 0.6 GPU-days per optimization run. Although the models were not identical, their sizes were comparable, and the training dataset remained constant across optimization runs. Consequently, this approach is efficient compared to the manual effort of experimenting with different configurations and training them for evaluation, a process that requires significantly more time.

For the adapter method, which serves as the main baseline for our automatic search approach, we include two configurations: one with the full default parameters of the OpenDelta library (bottleneck size of 24) and another with an increased bottleneck size of 64. This comparison aims to investigate whether simply increasing the

[14]https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/NL-code-search-Adv

Just Accepted

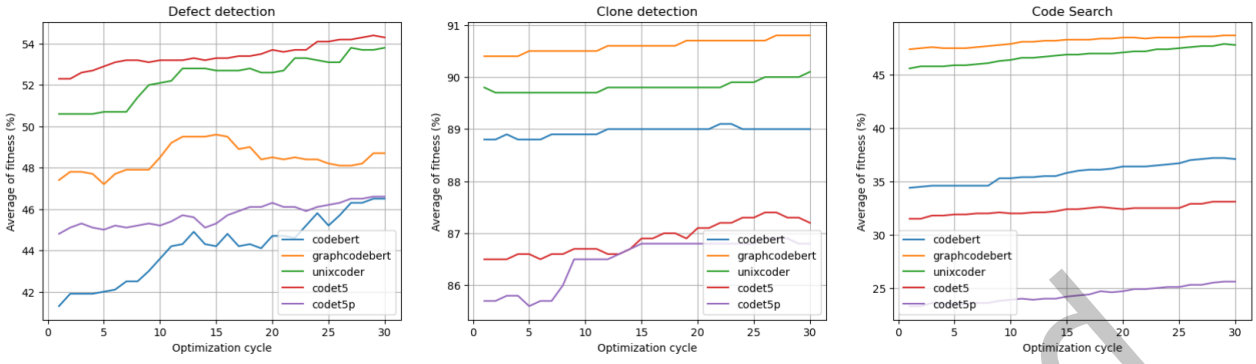


Fig. 4. Evolution of average fitness across optimization cycles for each task. Fitness is calculated as the sum of accuracy and F1-score minus the trainable rate for defect and clone detection, and as MRR minus the trainable rate for code search.

adapter size enhances performance. The bottleneck size of 64 was selected to align with the sizes of the adapters identified through the AutoAdapt search process. For performance comparisons, we consider both accuracy and the F1 score.

5.1 RQ1: Effectiveness of AutoAdaptfor Adapter Fine-Tuning

With this research question, we aim to verify whether various layers of pre-trained models necessitate different adapter configurations, as these layers may contain diverse transferable knowledge, and whether AutoML techniques can be effective in searching for such an optimal solution. We evaluate AutoAdapt on CodeBERT, GraphCodeBERT, UniXcoder, CodeT5, and CodeT5+ models across our evaluation datasets to achieve this. For comparison, we fine-tune the pre-trained models using the baseline PEFT methods mentioned in related work and full fine-tuning.

5.1.1 Defect detection : Table 2 presents the test performance of all fine-tuned models using various methods in the Devign dataset. To further illustrate our search results, Table 8 presents examples of the top configurations for each model discovered by AutoAdapt in the Devign dataset. The results demonstrate that our AutoAdapt method effectively identified an adapter structure that outperformed all other PEFT modules across all the evaluated pre-trained models. Furthermore, it exceeded the full fine-tuning performance of most models, with the exception of CodeBERT's full fine-tuning, where the performance difference remains close to zero.

Among the PEFT modules applied to the Devign dataset, the adapter with a bottleneck size of 64 achieves the best performance, while LoRA delivers comparable results with significantly fewer parameters. In contrast, prefix tuning consistently performs the worst across all cases, despite introducing the highest number of trainable parameters.

Compared to the default adapter (bottleneck size of 24, described in Figure 1), AutoAdapt consistently achieves higher Accuracy and F1-Score across all models. When increasing the adapter size to 64, resulting in a trainable rate between 1.85 and 2.09, AutoAdapt continues to demonstrate superior performance compared to the default adapter configuration with comparable or fewer parameters. We calculate the difference with the best-performing adapter between sizes 24 and 64. The performance improvement in accuracy ranges from +1.13% to +4.46%, while the F1-Score improvement ranges from 0% to +5.77%. These results are achieved with parameter rates ranging from 1.03% to 2.46%. Notably, the smallest adapter configuration identified by AutoAdapt is for the UniXcoder

Just Accepted

Table 2. Fine-tuning results on the Devign dataset for vulnerability detection. 'Params' indicates the size of trainable adapter parameters relative to the total model parameters, with 'r' representing the adapter's bottleneck size. Default methods refer to those defined in the OpenDelta library. The performance difference between the AutoAdapt adapter and the default adapter is calculated using the best-performing default adapter. and All values are expressed as percentages.

Model	Finetune Method	Params %	Accuracy %	F1-Score %	Precision %	Recall %
CodeBERT	Full	100	**63.51**	50.17	67.29	40.00
	Adapter(default)	0.71	59.26	54.37	56.00	52.83
	Adapter($r = 64$)	1.87	60.25	56.24	56.89	55.62
	LoRA(default)	0.23	60.94	49.21	61.11	41.2
	Prefix(default)	7.23	54.25	1.57	66.67	0.8
	AutoAdapt Adapter	1.30	63.18	**57.98**	60.93	55.30
Diff AutoAdapt - default Adapter		-0.57	+2.93	+1.74	+4.04	-0.32
Diff AutoAdapt - full finetune		-98.7	-0.33	+7.81	-6.36	+15.30
graphCodeBERT	Full	100	63.58	55.68	63.13	49.80
	Adapter(default)	0.71	60.25	**59.08**	56.04	62.47
	Adapter($r = 64$)	1.87	60.65	56.91	57.29	56.57
	LoRA(default)	0.23	62.19	55.42	60.45	51.16
	Prefix(default)	7.23	63.03	55.78	61.9	50.76
	AutoAdapt Adapter	2.29	**64.71**	**59.08**	63.22	55.46
Diff AutoAdapt - default Adapter		+1.58	+4.46	0.0	+7.18	-7.01
Diff AutoAdapt - full finetune		-97.71	+1.13	+3.40	+0.09	+5.66
UniXcoder	Full	100	64.57	58.38	63.40	54.10
	Adapter(default)	0.71	60.94	57.91	57.34	58.49
	Adapter($r = 64$)	1.85	62.19	59.41	58.60	60.24
	LoRA(default)	0.23	62.30	54.82	60.98	49.80
	Prefix(default)	7.16	61.97	54.96	60.27	50.52
	AutoAdapt Adapter	1.03	**64.57**	**61.37**	61.47	61.27
Diff AutoAdapt - default Adapter		-0.82	+2.28	+1.96	+2.87	+0.83
Diff AutoAdapt - full finetune		-98.97	0.00	+2.99	-1.93	+7.17
CodeT5	Full	100	63.29	59.83	60.14	59.62
	Adapter(default)	0.80	61.42	56.34	58.67	54.18
	Adapter($r = 64$)	2.09	63.07	56.26	61.69	51.71
	LoRA(default)	0.26	62.52	56.57	60.47	53.15
	Prefix(default)	7.92	59.99	54.44	57.08	52.03
	AutoAdapt Adapter	2.46	**64.20**	**62.03**	60.48	63.67
Diff AutoAdapt - Adapter default		+0.37	+1.13	+5.77	-1.21	+11.96
Diff AutoAdapt - full finetune		-97.51	+0.91	+2.20	+0.43	+4.05
CodeT5+ 220m	Full	100	63.47	58.86	60.97	56.89
	Adapter(default)	0.80	60.65	58.76	56.66	61.04
	Adapter($r = 64$)	2.09	61.05	56.32	58.09	54.66
	LoRA(default)	0.26	60.10	49.26	59.24	42.15
	Prefix(default)	7.92	59.74	47.77	59.11	40.08
	AutoAdapt Adapter	1.95	**63.95**	**59.55**	61.44	57.77
Diff AutoAdapt - Adapter default		-0.14	+2.90	+3.23	+3.35	+3.11
Diff AutoAdapt - full finetune		-98.05	+0.48	+0.96	+0.47	+0.88

Just Accepted

model, where only 1.03% additional parameters resulted in an increase of +2.28% in accuracy and +1.96% in F1-Score compared to the default adapter. Furthermore, this configuration achieved the same accuracy as full fine-tuning for UniXcoder while delivering an additional +2.99% improvement in F1-Score.

PEFT methods are generally expected to achieve performance close to full fine-tuning but are not typically anticipated to surpass it, given the significant difference in trainable parameter rates (e.g., 1% vs. 100%). However, AutoAdapt demonstrates that it is possible to identify adapter configurations that outperform full fine-tuning while using less than 2.5% of trainable parameters. For instance, with the Graphcodebert model, AutoAdapt achieves an accuracy improvement of +1.13% and an F1-Score improvement of +3.4% compared to full fine-tuning, while utilizing only 2.29% of trainable parameters. This highlights the potential of automated search methods to optimize performance efficiently.

These results for the defect detection task indicate that, although increasing the adapter size can enhance performance, the improvements are less efficient compared to those achieved by AutoAdapt, which effectively optimizes adapter configurations with fewer parameters.

5.1.2 Clone detection. Table 3 presents the fine-tuning results of the best configurations identified by AutoAdapt alongside baseline methods. For this task, AutoAdapt successfully discovered adapter configurations that outperform all PEFT baselines across the evaluated models. Notably, AutoAdapt also surpasses full fine-tuning performance for the UniXcoder and CodeT5 models. While full fine-tuning achieves slightly better results for the other models, the performance gap with AutoAdapt is statistically not significant.

Among the baseline PEFT methods, LoRA achieves the best performance in most cases, with an exceptionally low trainable parameter rate ranging from 0.23% to 0.26%. For the CodeT5 and CodeT5+ models, the adapter method outperformed prefix tuning, whereas for CodeBERT, GraphCodeBERT, and UniXcoder, prefix tuning demonstrated better performance than the adapter method.

When comparing the adapter configurations discovered by AutoAdapt to the baseline adapters, AutoAdapt surpasses the baselines with improvements of up to +4.61% in F1-Score while maintaining a trainable parameter rate comparable to or lower than that of an adapter with a size of 64. Accuracy improvements range from +0.24% to +1.39%; however, given that accuracy is already close to 100%, F1-Score is primarily adopted as the key metric for this task. The most significant F1-Score improvements are observed in models based on the RoBERTa architecture (CodeBERT, GraphCodeBERT, and UniXcoder), with gains around +4.5%. In contrast, the improvements for CodeT5 and CodeT5+ are smaller, highlighting the varying impact of AutoAdapt across different model architectures.

Compared to full fine-tuning, AutoAdapt demonstrates superior performance for the UniXcoder and CodeT5 models while reducing the trainable parameters by 98%. For the other models, the difference in F1-Score compared to full fine-tuning is minimal: 0.01 for CodeBERT, 0.21 for GraphCodeBERT, and 1.12 for CodeT5+.

5.1.3 Code search. Table 4 presents the results of fine-tuning PEFT methods and AutoAdapt top adapters on the code search dataset. For this task, the MRR is used as the evaluation metric.

For the code search task, AutoAdapt achieves the best performance among PEFT methods across all models, except for GraphCodeBERT, where LoRA performs the best. AutoAdapt also outperforms full fine-tuning for CodeBERT and UniXcoder while achieving comparable performance for CodeT5 and CodeT5+.

When comparing baseline PEFT methods, LoRA and prefix tuning perform exceptionally well for RoBERTa-based models, even outperforming full fine-tuning when using default parameters. However, for CodeT5 and CodeT5+, prefix tuning exhibits a significant drop in performance, with a 12% decrease in MRR, while LoRA also shows a performance decline of approximately 4%. The default adapter performs the worst across all models. However, with AutoAdapt, we successfully configure the adapter to outperform other methods and even surpass full fine-tuning.

Just Accepted

Table 3. Fine-tuning results on BigCloneBench dataset for Clone Detection. 'Params' indicates the size of trainable adapter parameters relative to the total model parameters, with 'r' representing the adapter's bottleneck size. Default methods refer to those defined in the OpenDelta library. The performance difference between the AutoAdapt adapter and the default adapter is calculated using the best-performing default adapter. All values are expressed as percentages.

Model	Finetune Method	Params %	Accuracy %	F1-score %	Precision %	Recall %
CodeBERT	Full	100	**98.20**	**93.38**	**94.49**	**92.29**
	Adapter(default)	0.71	96.92	89.17	86.10	92.48
	Adapter($r = 64$)	1.87	96.88	89.24	84.63	94.39
	LoRA(default)	0.23	97.95	92.68	90.98	94.44
	Prefix(default)	7.23	97.06	89.86	85.36	94.86
	AutoAdapt Adapter	1.60	98.18	93.37	93.42	93.32
Diff AutoAdapt - default Adapter		-0.27	+1.30	+4.13	+8.79	-1.07
Diff AutoAdapt - full finetune		-98.40	-0.02	-0.01	-1.07	+1.03
GraphCodeBERT	Full	100	**98.35**	**93.94**	**94.79**	**93.11**
	Adapter(default)	0.71	96.60	88.23	84.00	92.92
	Adapter($r = 64$)	1.87	96.92	89.10	86.57	91.78
	LoRA(default)	0.23	97.84	92.38	89.59	95.35
	Prefix(default)	7.23	97.83	92.35	89.39	95.51
	AutoAdapt Adapter	1.96	98.28	93.73	93.59	93.88
Diff AutoAdapt - default Adapter		+0.09	+1.36	+4.63	+7.02	+2.10
Diff AutoAdapt - full finetune		-98.04	-0.07	-0.21	-1.20	+0.77
UniXcoder	Full	100	98.37	94.14	**93.08**	**95.23**
	Adapter(default)	0.71	96.69	88.42	85.21	91.88
	Adapter($r = 64$)	1.85	97.03	89.65	85.92	93.72
	LoRA(default)	0.23	98.21	93.63	91.68	95.65
	Prefix(default)	7.16	98.02	92.97	90.85	95.2
	AutoAdapt Adapter	2.36	**98.42**	**94.26**	93.83	94.69
Diff AutoAdapt - default Adapter		+0.51	+1.39	+4.61	+7.91	+2.97
Diff AutoAdapt - full finetune		-97.64	+0.05	+0.12	+0.75	-0.63
CodeT5	Full	100	98.07	93.08	**91.67**	**94.53**
	Adapter(default)	0.80	97.99	92.79	91.23	94.41
	Adapter($r = 64$)	2.09	97.73	91.88	90.52	93.27
	LoRA(default)	0.26	97.82	92.31	89.55	95.25
	Prefix(default)	7.92	96.55	88.12	83.67	93.06
	AutoAdapt Adapter	1.09	**98.23**	**93.60**	93.04	94.16
Diff AutoAdapt - default Adapter		+0.29	+0.24	+0.52	+1.81	-0.37
Diff AutoAdapt - full finetune		-98.91	+0.16	+0.52	+1.38	-0.37
CodeT5+ 220M	Full	100	**98.33**	**93.95**	**93.31**	**94.60**
	Adapter(default)	0.80	97.50	91.24	87.90	94.85
	Adapter($r = 64$)	2.09	97.62	91.64	88.62	94.86
	LoRA(default)	0.26	97.89	92.57	89.76	95.57
	Prefix(default)	7.92	96.49	88.05	82.57	94.32
	AutoAdapt Adapter	1.90	97.99	92.83	90.92	94.81
Diff AutoAdapt - default Adapter		-0.19	+0.37	+1.19	+2.30	-0.05
Diff AutoAdapt - full finetune		-98.10	-0.23	-1.12	+2.39	+0.21

Just Accepted

The use of AutoAdapt to search for optimized adapter configurations significantly improved performance compared to the default adapter settings. The performance gains range from 8.5% to 25.59%, achieved with the introduction of only 1.37% to 2.13% additional trainable parameters—a rate comparable to that of an adapter with a bottleneck size of 64.

AutoAdapt achieves better performance than full fine-tuning when applied to CodeBERT, GraphCodeBERT, and UniXcoder (RoBERTa-based models). For UniXcoder specifically, it delivers an improvement of 3.73‰. In contrast, the performance difference is smaller for CodeT5+ at only 0.07%, while it reaches 1.91% for CodeT5. These results suggest that PEFT methods for code search are more likely to perform better on RoBERTa-based models compared to T5-based models.

Finding 1: AutoAdapt shows that automatically configuring and optimizing adapter parameters, along with using layer-specific adaptations, can significantly improve performance. It achieves F1-score gains of up to 5.77% for defect detection, 4.63% for clone detection, and 25.37% in MRR for code search. AutoAdapt also outperforms full fine-tuning in most cases while using less than 2.5% of trainable parameters. This demonstrates the potential of AutoML to effectively enhance PEFT methods.

5.2 RQ2: Adapter Design Choices

Given the many parameters explored during the AutoAdapt search and their positive impact on performance, our second research question examines how each parameter influences adapter efficiency, aiming to identify potential recommendations, such as suggesting design choices with a positive impact on model performance. To provide these recommendations, we analyze the final population identified by the evolutionary algorithm, containing the top adapter configurations sorted by fitness. Our analysis focuses on the impact of selective insertion layers, insertion modules, adapter size, normalization, and dropout.

5.2.1 Insert layers : Figure 5 illustrates the placement of adapters across layers for various tasks and models in the final population resulting from the optimization process. Additionally, Figure 6 shows the number of adapters inserted in each layer region of the top 10 models evaluated during the optimization process. The analysis reveals that, for all models, adapters are inserted into only a subset of the layers rather than all layers. Moreover, the insertion pattern often alternates between layers. For CodeT5+, adapters in the top 10 models are predominantly placed in the later layers, whereas CodeT5 exhibits more dense and frequent insertions across layers.

However, no universal rule can be established regarding which specific layers are best for adapter insertion, as the selection appears to depend entirely on the specific combination of model and task. Figures 5 and 6 highlight this variation, emphasizing the importance of automated tools like AutoAdapt to optimize the choice of layers for adapter insertion.

In general, the best-performing configurations do not insert adapters in all layers of the model, aligning with findings from prior research [11]. Notably, the insertion pattern tends to be alternating, where adapters are placed in some layers while skipped in others. This reinforces the importance of automated optimization methods, as different setups benefit from distinct insertion patterns.

5.2.2 Insert modules . We aim to investigate which modules are most suitable for adapter insertion, depending on the layer regions (earliest, middle, or latest). Figure 6 presents the frequency of adapter insertions across different modules and layer regions for the top 10 adapter configurations identified by AutoAdapt for the best-performing models (GraphCodeBERT, UniXcoder, and CodeT5+), as determined in RQ1. Additional frequency matrices for other models are provided in Figure 8.

Just Accepted

Table 4. Fine-tuning Methods and MRR Results Across Models. 'Params' indicates the size of trainable adapter parameters relative to the total model parameters, with 'r' representing the adapter's bottleneck size. Default methods refer to those defined in the OpenDelta library. The performance difference between the AutoAdapt adapter and the default adapter is calculated using the best-performing default adapter. All values are expressed as percentages.

Model	Finetune Method	Params%	MRR %
CodeBERT	Full	100	31.79
	Adapter (default)	0.71	9.86
	Adapter ($r = 64$)	1.87	10.86
	LoRA (default)	0.23	31.3
	Prefix (default)	7.23	32.78
	AutoAdapt Adapter	2.13	**32.85**
Diff AutoAdapt - default Adapter		+0.26	+21.99
Diff with full finetune		-97.87	+1.06
GraphCodeBERT	Full	100	37.53
	Adapter (default)	0.71	11.46
	Adapter ($r = 64$)	1.87	15.67
	LoRA (default)	0.23	**38.44**
	Prefix (default)	7.23	38.17
	AutoAdapt Adapter	1.37	38.26
Diff AutoAdapt - default Adapter		-0.50	+22.59
Diff with full finetune		-98.63	+0.73
UniXcoder	Full	100	36.85
	Adapter (default)	0.71	14.73
	Adapter ($r = 64$)	1.85	15.21
	LoRA (default)	0.23	39.32
	Prefix (default)	7.16	39.8
	AutoAdapt Adapter	1.88	**40.58**
Diff AutoAdapt - default Adapter		+0.03	+25.37
Diff with full finetune		-98.12	+3.73
CodeT5	Full	100	**33.99**
	Adapter (default)	0.80	13.25
	Adapter ($r = 64$)	2.09	13.62
	LoRA (default)	0.26	29.86
	Prefix (default)	7.92	21.16
	AutoAdapt Adapter	1.88	31.99
Diff AutoAdapt - default Adapter		-0.21	+18.37
Diff with full finetune		-98.12	-1.91
CodeT5+ 220M	Full	100	**33.76**
	Adapter (default)	0.80	25.19
	Adapter ($r = 64$)	2.09	24.71
	LoRA (default)	0.26	28.29
	Prefix (default)	7.92	21.15
	AutoAdapt Adapter	1.61	33.69
Diff AutoAdapt - default Adapter		+0.81	+8.50
Diff with full finetune		-98.39	-0.07

Just Accepted

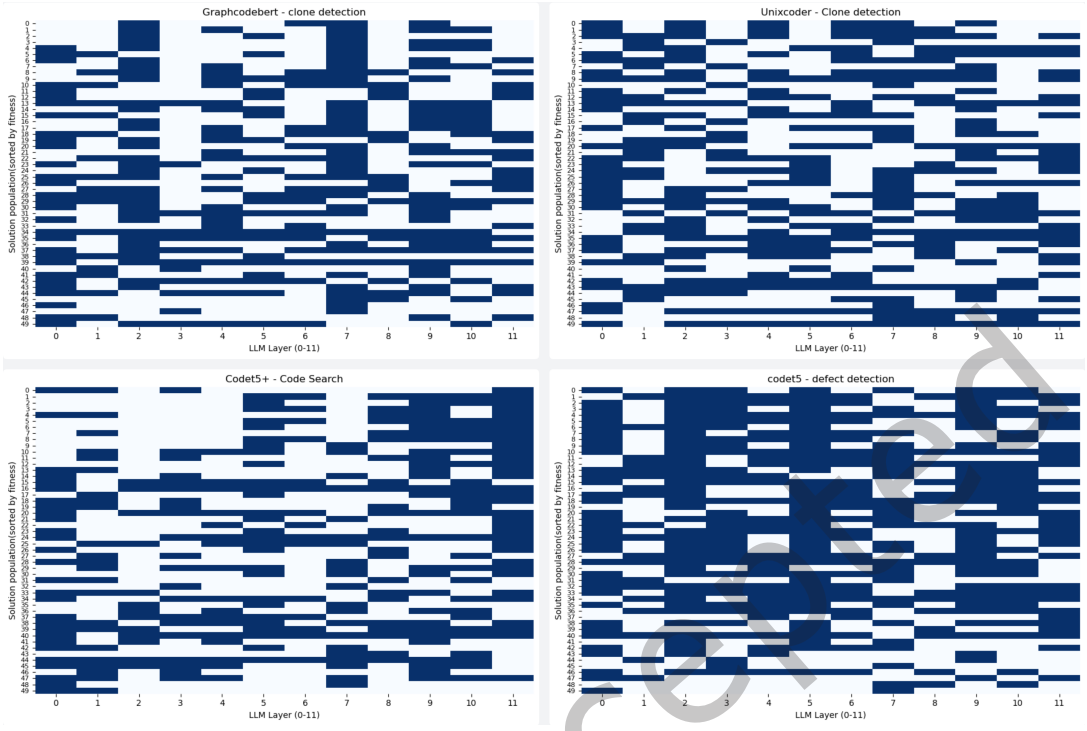


Fig. 5. Illustration of adapter insertion across layers in the final population of the optimization process (50 adapter configurations). Rows represent individual solutions sorted by fitness from top (highest fitness) to bottom (lowest fitness). Columns correspond to LLM layers (0–11). Colored regions indicate layers where adapters were inserted, while uncolored regions denote no adapter insertion.

Just Accepted

The results indicate that *earliest layers* (closer to the input) predominantly benefit from adapter insertion in the attention sub-modules. These layers are crucial for capturing token-level interactions and contextual relationships, which aligns with their functional role in early-stage encoding.

For the *middle layers*, the distribution of adapter insertions is more evenly spread across modules, with no single module showing a dominant preference. This suggests that in this region, both `attention` (self and output) and feed-forward modules are equally critical, as middle layers balance feature transformation and representation learning.

In the *final layers*, the output and feed-forward modules become significantly more active for adapter insertion. These deeper layers are responsible for task-specific representations and heavily rely on feed-forward operations, emphasizing the importance of these modules in refining task-relevant features.

Overall, the findings suggest that *for earlier layers*, it is more effective to focus on `attention.self`, `attention.output`, and `layer.0.SelfAttention`, as these modules handle token interactions and relationships. For *final layers*, priority should be given to the `intermediate`, `output`, and `layer.1.DenseReluDense` modules, where feed-forward computations dominate and task-specific refinements are performed.

5.2.3 Key parameters . In Figure 7, we present the correlation matrices illustrating the relationship between adapter performance (MRR for code search and F1 score for others) and several key adapter hyperparameters

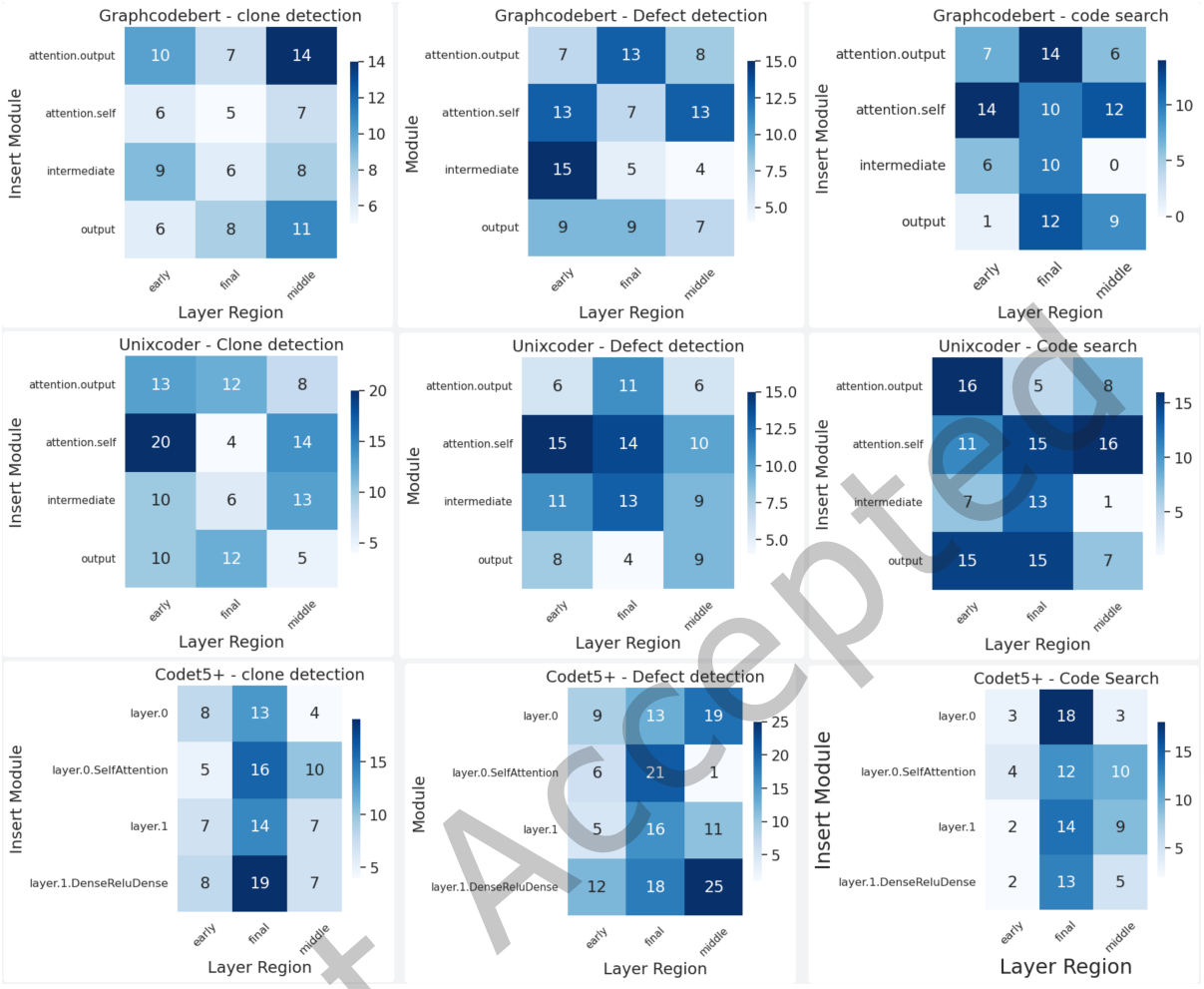


Fig. 6. Frequency matrix showing the distribution of adapter insertions across modules and layer regions in the top 10 candidate models trained during the AutoAdapt search process. Rows represent different modules where adapters could be inserted, and columns correspond to layer regions: Early (layers 1–4), Middle (layers 5–8), and Final (layers 9–12). The cell values indicate the number of insertions in each module-layer region combination.

Just Accepted

from the final populations obtained via AutoAdapt. Specifically, these final populations include the top 50 adapter configurations, each with its respective performance. The values range from -1 (strong negative correlation) to 1 (strong positive correlation), with 0 indicating no linear correlation. The analyzed parameters include the adapter size (measured as the proportion of trainable parameters relative to the model), the normalization fraction (the proportion of adapters incorporating a normalization layer), and the mean dropout (average dropout rate applied within the adapter configurations).

- Adapter size: As illustrated in Figure7, the correlation between performance and adapter parameter size is negative for most models (ranging from -0.46 to -0.15), with the notable exception of CodeT5, which exhibits a strong positive correlation (+0.70).

A task-specific analysis shows that clone detection presents a positive correlation, except for CodeT5+, where the correlation is near zero (-0.03). Suggesting that this task may benefit from increased parameter capacity. Analyzing by model architecture reveals that CodeT5 consistently maintains a strong positive correlation across all tasks (ranging from 0.34 to 0.70), especially for defect detection and code search, tasks that typically require richer representations. For CodeT5+, the correlation is near zero or occasionally positive, suggesting that T5-based models often benefit from additional parameters but may not scale as consistently as CodeT5. In contrast, other model architectures maintain a negative correlation, indicating that simply increasing adapter size or adding more adapters does not uniformly translate into performance gains. These observations underscore that the effectiveness of larger adapters depends heavily on both the model architecture and the task; while T5-based models (particularly CodeT5) may leverage additional parameters effectively, certain tasks like clone detection also appear to benefit from increased parameter capacity.

- Adding Normalization layers: To investigate the potential benefits of incorporating normalization layers within the adapter module, we conducted both task-based and model-based analyses. Our results show that, for the clone detection task, there is a positive correlation between model performance and the proportion of normalization layers when using CodeBERT, GraphCodeBERT, and CodeT5 models. In contrast, the other tasks do not exhibit a consistent trend. When we examine performance by model architecture, CodeT5 consistently shows a positive correlation between its performance and the proportion of normalization layers across all tasks (from +0.26 to +0.29). These findings suggest that while normalization layers can particularly be advantageous for clone detection, their impact may vary depending on the specific task and model used.
- Dropout rate: We incorporate a dropout layer into our AutoAdapt adapters and investigate its utility across different tasks and models. As shown in Figure 7, the correlation between performance and dropout rate is positive for the defect detection task in the cases of GraphCodeBERT, UnixCoder, and CodeT5 +, while it is negative for the others, likely reflecting the smaller size of its dataset and a higher risk of overfitting. Moreover, for CodeBERT applied to the code search task, the correlation also appears positive. However, in other scenarios, the correlation tends to be close to zero or negative. Overall, these results suggest that dropout can be beneficial for tasks with higher overfitting potential, but may not consistently improve performance across all tasks or model architectures.

Finding 2: Our analysis shows that certain design choices do yield performance gains, but they depend heavily on both the task and the model architecture. Strategic adapter placement—targeting attention modules in the earlier layers and feed-forward modules in the later layers, while limiting insertion to select layers — enhances fine-tuning efficiency. Larger adapter sizes and normalization layers benefit clone detection and CodeT5-based models, while dropout is particularly useful for smaller datasets like Devign. These findings emphasize the importance of tailoring adapter designs to each task and model.

Just Accepted

5.3 RQ3: Transferability of Adapters

With this research question, we aim to explore whether optimized adapter configurations for one dataset can be effectively reused for another dataset with a similar task, as well as to a dataset with a different task. To achieve this, we use the top two adapter configurations identified in the Devign dataset when training the UnixCoder model and apply them to FlakeFlagger, BigCloneBench, and AdvTest datasets. In the following, we mean by task similarity as having similar characteristics and overall goals (e.g., classifying source code).

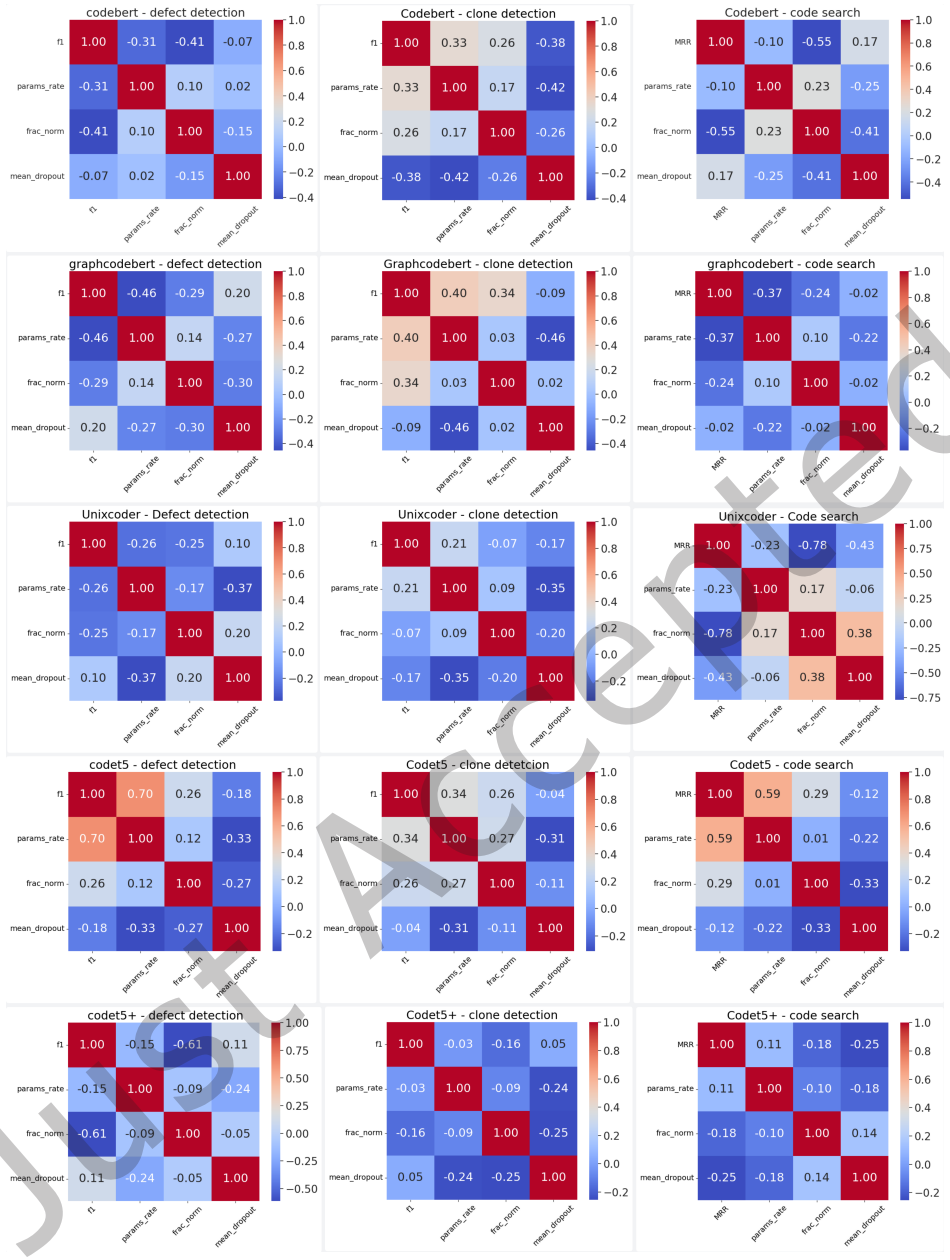


Fig. 7. Correlation heatmaps for the final population of adapter configurations for each model-task combination explored during the AtoAdapt search process. The heatmaps display Pearson correlation coefficients between fitness, parameter rate (`params_rate`), fraction of normalized adapters (`frac_norm`), mean dropout rate (`mean_dropout`), and the number of adapters (`num_adapters`). Color intensity indicates the strength and direction of correlations (with -1 representing a strong negative correlation, 1 a strong positive correlation, and 0 no linear correlation), and annotated values highlighting key relationships that influence model performance across defect detection, code search, and clone detection tasks.

Just Accepted

We investigate whether adapter configurations transfer well between defect detection and test flakiness detection. Thus, we insert the best adapter configurations for Unixcoder found with Devign and fine-tune it on the FlakeFlagger dataset for Flaky tests prediction. Table 5 presents the results, comparing these transferred configurations with standard default adapters (using bottleneck sizes of 24 and 64) and full fine-tuning. Notably, the transferred adapter configurations perform well on the FlakeFlagger dataset. For instance, Config 1, which uses only 1.03% of the total parameters, achieves an F1-score of 68.21%, outperforming the default adapter with a bottleneck size of 64 (F1-score: 65.76%, using 1.87% parameters). Config 2, ranked second for defect detection, uses 3.5% of the parameters and achieves the highest F1-score of 71.43%, surpassing full fine-tuning by 2.12%. These findings highlight the effectiveness of transferring adapter configurations from defect detection to flakiness detection, due to the similarity of the tasks—both being binary classification problems focused on identifying bugs in code. However, the optimal ranking of configurations can vary between datasets and tasks. While evaluating the top configurations consistently provides high performance, some datasets may benefit from configurations with larger parameter budgets.

Table 5. Fine-tuning results on the FlakeFlagger dataset for flakiness detection using adapter configurations searched on the Devign dataset (Adapter_Defect 1 and 2). The trainable rate represents the proportion of adapter parameters to total model parameters, with 'r' indicating the adapter bottleneck size. Default methods refer to those in the OpenDelta library. All values are percentages.

Fine-tune Method	Trainable Rate %	Accuracy %	F1-score %	Precision %	Recall %
Full	100	97.87	69.31	74.47	64.81
Adapter(default, $r = 64$)	0.71	97.5	60.07	73.87	50.62
Adapter(default , $r = 64$)	1.87	97.68	65.76	72.93	59.88
Adapter_Defect 1	1.03	97.8	68.21	73.57	63.58
Adapter_Defect 2	3.5	**97.98**	**71.43**	**75.34**	**67.9**

We also investigate the transferability of adapter configurations from defect detection to clone detection, two similar binary classification tasks with different objectives. Clone detection involves comparing the similarity of two code snippets, whereas defect detection focuses on identifying bugs in code. Table 6 presents the results of fine-tuning the top configurations identified on the Devign dataset on the clone detection dataset.
The first configuration, which uses only 1.03% of the trainable parameters, achieves an F1-score of 93.58—significantly outperforming the default adapters (limited to 89.65) but slightly below the performance of full fine-tuning (94.14). Notably, the second configuration, utilizing 3.5% of the parameters, surpasses full fine-tuning with an F1-score of 94.23. This performance is equivalent to that achieved by the adapter configuration optimized specifically for clone detection, as shown in Table 3. These results demonstrate the potential for transferring adapter configurations across similar tasks while maintaining competitive performance.

Finally, we evaluate the transferability of adapter configurations from defect detection to code search, two fundamentally different tasks. Table 7 presents the results. The first configuration, with a trainable rate of only 1.03%, surpasses the baseline adapters but performs slightly below full fine-tuning, achieving an MRR of 35.75 compared to 36.85 for full fine-tuning. However, the second configuration, using 3.5% of the parameters, achieves an MRR of 39.79, outperforming full fine-tuning by 2.94‰. While slightly below the performance of the configuration optimized specifically for code search (MRR: 40.58%), these results demonstrate the potential for transferring adapter configurations across tasks, even when the tasks are as different as defect detection and code search.

Just Accepted

Table 6. Fine-tuning results on the BigCloneBench dataset for clone detection using adapter configurations (Adapter_Defect 1 and 2) identified from the Devign dataset. The trainable rate denotes the percentage of adapter parameters relative to the total model parameters, with 'r' specifying the adapter bottleneck size. Default methods correspond to those provided by the OpenDelta library. All results are reported as percentages.

Fine-tune Method	Trainable Rate %	Accuracy %	F1-score %	Precision %	Recall %
Full	100	98.37	94.14	93.08	95.23
Adapter(default, $r = 24$)	0.71	96.69	88.42	85.21	91.88
Adapter(default , $r = 64$)	1.85	97.03	89.65	85.92	93.72
Adapter_Defect 1	1.03	98.2	93.58	91.54	95.72
Adapter_Defect 2	3.5	**98.39**	**94.23**	**93.05**	**95.44**

Table 7. Fine-tuning results on the AdvTest dataset for code search using adapter configurations (Adapter_Defect 1 and 2) from the Devign dataset. The trainable rate indicates the proportion of adapter parameters relative to the total model, with 'r' representing the bottleneck size. Default methods are from the OpenDelta library, and all results are expressed as percentages.

Fine-tune Method	Trainable Rate %	MRR %
Full	100	36.85
Adapter(default , $r = 24$)	0.71	14.73
Adapter(default , $r = 64$)	1.85	15.21
Adapter_Defect 1	1.03	35.75
Adapter_Defect 2	3.5	**39.79**

Finding 3: Adapter configurations found by Autoadapt that are effective for the evaluated task can transfer well to another similar task, often achieving performance comparable to task-specific optimizations. For tasks with different objectives, they remain competitive, but may not surpass configurations optimized directly for the target task. The best configuration for one task may not be optimal for another, but suitable options can often be found among the top-ranked ones, depending on the parameters budget.

6 DISCUSSION

In this section, we consider several factors that significantly impact the results, including the search space, PEFT module, and optimization method.

6.1 Search space

This study utilizes evolutionary NAS techniques to optimize adapter fine-tuning configurations for pre-trained model code models. Each layer within these adapters is configured individually to cater to specific tasks, aiming to assess whether custom adapters enhance performance during the fine-tuning process, considering that each layer encodes distinct knowledge.

The search space is defined by the random selection of various elements, including the insertion layer, insertion of sub-modules, size of the inserted bottleneck, activation function, inclusion of normalization layers, and dropout rate. This extensive combination of parameters results in a large and complex search space. To manage this complexity, we constrain the bottleneck size based on the type of sub-module: smaller sizes (e.g., 32, 64)

Just Accepted

are allocated to attention modules due to the substantial processing already performed within the attention mechanism, while larger sizes (e.g., 128, 256) are designated for feed-forward modules. Each sub-module type is limited to two size options to maintain computational efficiency. Additionally, we specify a set of dropout rates to be considered during the search process, including values such as 0.1, 0.15, 0.2, 0.25, and 0.3. These values were chosen based on common practices in the literature and preliminary experiments. However, this limitation may constrain the exploration of potentially advantageous configurations. Future research could aim to expand the range of possible choices by enhancing the search method's ability to evaluate candidate solutions more efficiently with the newest NAS methods designed to be more efficient.

Additionally, our architecture incorporates a skip connection after each adapter, where the adapter's output is added to the original output. Future studies could investigate the Learned Gating method, which introduces a gating parameter that blends the adapter output with the original hidden states at varying rates. Furthermore, experimenting with skip connections between different adapters at various layers could be valuable. Recent research indicates that injecting additional trainable modules to connect unfrozen modules in parameter-efficient fine-tuning can improve gradient flow, thus significantly enhancing convergence speed and performance [28].

6.2 PEFT method

In our study, we employed the Adapter module as the primary PEFT method optimized by AutoAdapt. While previous research suggested that adapter-based methods are among the most effective PEFT approaches for software engineering tasks, our evaluations revealed that the LoRA module outperforms both the Adapter and Prefix methods. Notably, LoRA introduces a minimal number of additional parameters, increasing the trainable parameters by less than 0.3%. Additionally, our analysis addressing RQ2 indicates that the most active sub-modules in the optimal configurations are the attention sub-modules, underscoring their significance in parameter-efficient fine-tuning. Consequently, adopting the LoRA method as a foundation for future optimizations could enhance efficiency and achieve superior performance compared to the other methods.

6.3 Optimization method

Exploring the extensive search space requires a comprehensive and resource-intensive search process. In our experiments, we utilized a population size of 50 pre-trained models and conducted 30 optimization cycles, resulting in a total runtime of approximately 0.6 GPU days for each combination of task and model. However, the limited population size constrained the optimization algorithm from consistently identifying optimal solutions with lower parameter counts.

This limitation was primarily due to the high computational costs associated with fine-tuning each pre-trained model using various candidate configurations. Despite employing low-fidelity evaluations to mitigate these costs, the evaluation process still demanded significant time investment. Consequently, we were compelled to maintain a small population size to manage computational resources effectively. Scaling up the population size would have necessitated substantially longer computational durations given the evaluation methods used.

Future work could explore the integration of surrogate models to assess the performance of candidate solutions without the need for extensive training. Surrogate modeling is a common approach in neural architecture search studies to reduce computational overhead. However, developing an effective surrogate model would require training it on a diverse set of adapter configurations and their corresponding performance metrics. This approach may incur costs comparable to directly training multiple models, particularly when dealing with large language models. Another technique worth investigating is transferring or sharing the trained parameters between similar adapter configurations during the optimization process. Parameter transfer could leverage previously learned representations, potentially reducing the computational burden and accelerating the search process by minimizing the need to train each candidate configuration from scratch.

Just Accepted

7 THREATS TO VALIDITY

External Validity : In this study, due to limited resources, our experimentation was constrained to a specific set of tasks, datasets, and pre-trained models, potentially introducing biases into the results. To mitigate this, we selected widely-used models, and well-known datasets in the field of source code understanding. We ensured experiment consistency and validity by maintaining consistent random seeds across models, along with repeating experiments using different seeds and calculating average results. In future research, we aim to expand our exploration to include a wider range of models and tasks in code generation.

Internal Validity : In our experiments with the BigCloneBench dataset, we opted to evaluate only a subset of the dataset due to its extensive size, comprising over 1,700,000 samples. Conducting experiments on the entire dataset would have been resource-intensive. Consequently, some results may differ from those reported in other papers. However, the subset of data used for evaluation remains sufficiently large for fine-tuning purposes, as fine-tuning typically does not require extensive datasets. Furthermore, to ensure the integrity of our experiments, we independently reproduced all results and did not rely on findings from other papers. Thus, our comparisons remain fair. Another potential threat to validity lies in the parameters utilized in the optimization algorithm, such as the sizes of the linear layers. We intentionally selected smaller sizes for the down projection to avoid introducing an excessive number of new parameters, thereby maintaining the efficiency of parameter fine-tuning.

8 CONCLUSION

In this study, we introduce AutoAdapt, a tailored approach designed to enhance the effectiveness and efficiency of Parameter-Efficient Fine-Tuning methods, particularly focusing on adapter fine-tuning, by automating the exploration of custom adapter configurations and architectures using an evolutionary neural architecture search method. We conducted evaluations in the realm of pre-trained code models, specifically targeting vulnerability detection, clone detection, and code search tasks using established benchmarks.

Our findings show that AutoAdapt significantly outperforms manually engineered adapter configurations and other PEFT techniques, such as Prefix Tuning and LoRA. While using around 2% of the model parameters, AutoAdapt was able to identify configurations that exceed the full fine-tuning performance of the evaluated models and tasks, highlighting its superior efficiency and effectiveness. A comprehensive analysis of our results revealed that several factors play critical roles in enhancing performance across different tasks. These include selective layer adaptation, emphasizing attention modules in earlier layers and feed-forward modules in later layers, incorporating normalization within adapters for tasks such as clone detection, and applying dropout for tasks with smaller datasets like defect detection. Additionally, certain model architectures, such as T5-based architectures, may require larger adapter parameter sizes; however, simply increasing the adapter size does not necessarily lead to performance improvements. These insights highlight the importance of customized adaptations tailored to both the model architecture and task-specific requirements. They also underscore the necessity of automated tools, such as AutoAdapt, to efficiently address these specifications and optimize configurations. When it comes to the transferability of adapter parameters for computational efficiency, it appears that AutoAdapt configurations optimized for one source code dataset could potentially perform well on other datasets with similar tasks. This potential could help conserve computational resources by minimizing the necessity to experiment with a wide range of settings for each dataset.

PEFT methods simplify the use of LLMs by transferring their knowledge to downstream tasks with minimal adjustments. Moving forward, we plan to explore avenues to enhance the performance and efficiency of PEFT methods for code models while keeping costs low.

Just Accepted

9 ACKNOWLEDGMENTS

This work is supported by the Luxembourg National Research Fund (FNR) through the CORE project under Grant C23_IS_18182513_MiCE.

REFERENCES

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333* (2021).

[3] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: predicting flakiness without rerunning tests. In *Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings* (Virtual Event, Spain) *(ICSE '21)*. IEEE Press, 187. https://doi.org/10.1109/ICSE-Companion52605.2021.00081

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Ronghong Cai and Jianping Luo. 2021. Multi-Task Learning for Multi-Objective Evolutionary Neural Architecture Search. In *2021 IEEE Congress on Evolutionary Computation (CEC)*. 1680–1687. https://doi.org/10.1109/CEC45853.2021.9504721

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/V1/N19-1423

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[9] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[11] Junliang Guo, Zhirui Zhang, Linli Xu, Boxing Chen, and Enhong Chen. 2021. Adaptive Adapters: An Efficient Way to Incorporate BERT Into Neural Machine Translation. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 29 (2021), 1740–1751. https://doi.org/10.1109/TASLP.2021.3076863

[12] Zeyu Han, Chao Gao, Jinyang Liu, Sai Qian Zhang, et al. 2024. Parameter-efficient fine-tuning for large models: A comprehensive survey. *arXiv preprint arXiv:2403.14608* (2024).

[13] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2021. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366* (2021).

[14] Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2021. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366* (2021).

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] Xinying Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John C. Grundy, and Haoyu Wang. 2023. Large Language Models for Software Engineering: A Systematic Literature Review. *ArXiv* abs/2308.10620 (2023). https://api.semanticscholar.org/CorpusID:261048648

[17] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International conference on machine learning*. PMLR, 2790–2799.

[18] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).

[19] Shengding Hu, Ning Ding, Weilin Zhao, Xingtai Lv, Zhen Zhang, Zhiyuan Liu, and Maosong Sun. 2023. OpenDelta: A Plug-and-play Library for Parameter-efficient Adaptation of Pre-trained Models. *arXiv preprint arXiv:2307.03084* (2023).

[20] Shengding Hu, Zhen Zhang, Ning Ding, Yadao Wang, Yasheng Wang, Zhiyuan Liu, and Maosong Sun. 2022. Sparse structure search for delta tuning. *Advances in Neural Information Processing Systems* 35 (2022), 9853–9865.

Just Accepted

[21] Kai Huang, Xiangxin Meng, Jian Zhang, Yang Liu, Wenjie Wang, Shuhao Li, and Yuqing Zhang. 2023. An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1162–1174. https://doi.org/10.1109/ASE56229.2023.00181

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[23] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942* (2019).

[24] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).

[25] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[26] J. Liu, C. Sha, and X. Peng. 2023. An Empirical Study of Parameter-Efficient Fine-Tuning Methods for Pre-Trained Code Models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 397–408. https://doi.org/10.1109/ASE56229.2023.00125

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[28] Xinyu Ma, Jiafeng Guo, Ruqing Zhang, Yixing Fan, and Xueqi Cheng. 2022. Scattered or Connected? An Optimized Parameter-efficient Tuning Approach for Information Retrieval. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management* (Atlanta, GA, USA) *(CIKM '22)*. Association for Computing Machinery, New York, NY, USA, 1471–1480. https://doi.org/10.1145/3511808.3557445

[29] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.

[30] Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterHub: A Framework for Adapting Transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020): Systems Demonstrations*. Association for Computational Linguistics, Online, 46–54. https://www.aclweb.org/anthology/2020.emnlp-demos.7

[31] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[32] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[33] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.* 21, 1, Article 140 (Jan. 2020), 67 pages.

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

[35] Noraini Mohd Razali, John Geraghty, et al. 2011. Genetic algorithm performance with different selection strategies in solving TSP. In *Proceedings of the world congress on engineering*, Vol. 2. International Association of Engineers Hong Kong, China, 1–6.

[36] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, Vol. 33. 4780–4789.

[37] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480. https://doi.org/10.1109/ICSME.2014.77

[38] Alexander Tornede, Difan Deng, Theresa Eimer, Joseph Giovanelli, Aditya Mohan, Tim Ruhkopf, Sarah Segel, Daphne Theodorakopoulos, Tanja Tornede, Henning Wachsmuth, et al. 2023. Automl in the age of large language models: Current challenges, future opportunities and risks. *arXiv preprint arXiv:2306.08107* (2023).

[39] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[40] Ivan Vulić, Edoardo Maria Ponti, Robert Litschko, Goran Glavaš, and Anna Korhonen. 2020. Probing Pretrained Language Models for Lexical Semantics. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 7222–7240. https://doi.org/10.18653/v1/2020.emnlp-main.586

[41] Deze Wang, Boxing Chen, Shanshan Li, Wei Luo, Shaoliang Peng, Wei Dong, and Xiangke Liao. 2023. One Adapter for All Programming Languages? Adapter Tuning for Code Search and Summarization. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 5–16. https://doi.org/10.1109/ICSE48619.2023.00013

Just Accepted

[42] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D.Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. *arXiv preprint* (2023).
[43] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
[44] Ruoting Wu, Yu xin Zhang, Qibiao Peng, Liang Chen, and Zibin Zheng. 2022. A Survey of Deep Learning Models for Structural Code Understanding. *ArXiv* abs/2205.01293 (2022). https://api.semanticscholar.org/CorpusID:248506125
[45] Han Zhou, Xingchen Wan, Ivan Vulić, and Anna Korhonen. 2023. Autopeft: Automatic configuration search for parameter-efficient fine-tuning. *arXiv preprint arXiv:2301.12132* (2023).
[46] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).
[47] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

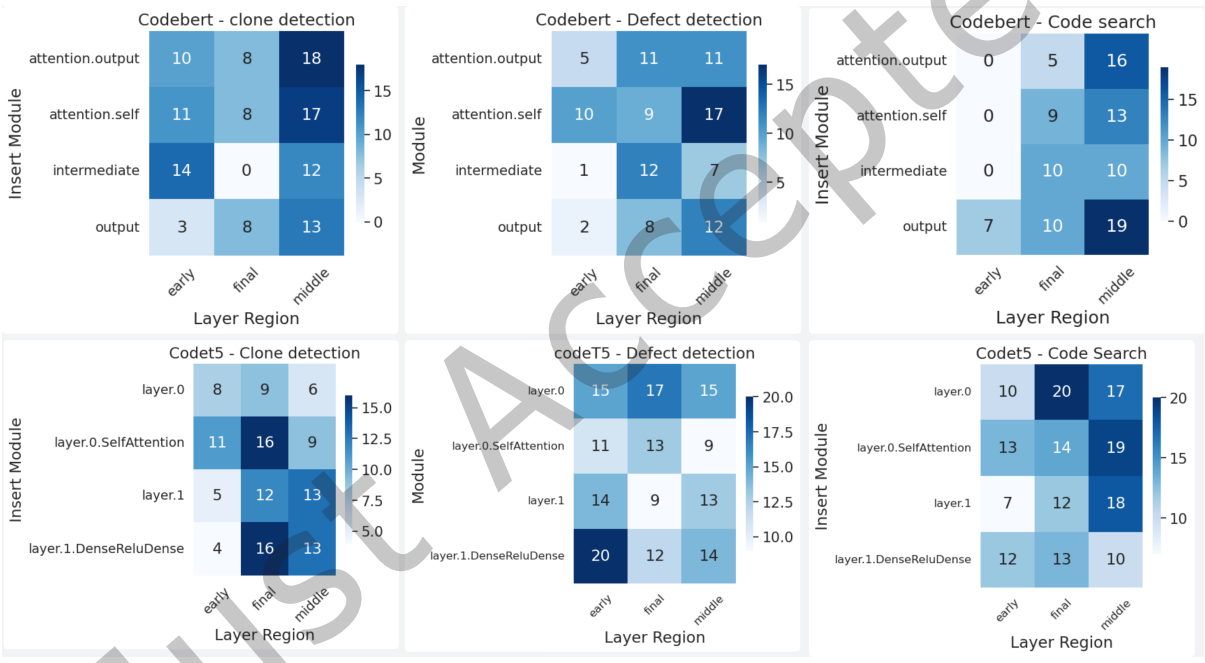


Fig. 8. Frequency matrix showing the distribution of adapter insertions across modules and layer regions in the top 10 candidate models trained during the AutoAdapt search process. Rows represent different modules where adapters could be inserted, and columns correspond to layer regions: Early (layers 1–4), Middle (layers 5–8), and Final (layers 9–12). The cell values indicate the number of insertions in each module-layer region combination.

Just Accepted

Table 8. Top configurations discovered by AutoAdapt for each model on the Devign dataset for defect detection. "Layer" indicates the model layer where the adapter is inserted; "Insert Modules" specifies the sub-modules (grouped by the insert layer) where adapters are added; "Size" gives the corresponding bottleneck sizes; and "Function", "Dropout", and "Norm" are the configuration parameters.

Model	Layer	Insert Modules	Size	Function	Dropout	Norm
UniXcoder	1	attention.self intermediate output	16 64 128	gelu	0.2	LayerNorm
	4	intermediate attention.self	64 32	swish	0.3	LayerNorm
	7	attention.self output	32 128	silu	0.25	LayerNorm
CodeBERT	8	attention.self intermediate attention.output	16 128 32	swish	0.0	None
	9	intermediate	128	tanh	0.0	LayerNorm
GraphCodeBERT	2	attention.self intermediate output	32 128 128	relu	0.0	LayerNorm
	3	intermediate	64	gelu_new	0.15	None
	4	intermediate	64	swish	0.25	None
	5	intermediate attention.output attention.self	64 32 32	swish	0.1	None
	7	attention.self output	16 128	leakyrelu	0.3	None
	10	attention.output	64	swish	0.25	LayerNorm
	12	attention.self attention.output output	16 32 128	leakyrelu	0.1	LayerNorm
CodeT5	2	Layer.0.SelfAttention	16	silu	0.1	None
	3	Layer.1.DenseReluDense	128	gelu_new	0.25	None
	4	Layer.1.DenseReluDense Layer.0	128 64 *128*	tanh	0.3	LayerNorm
	5	layer.1 layer.0 layer.0.SelfAttention	128 64 16	tanh	0.1	None
	6	layer.1	256	leakyrelu	0.2	None
	7	layer.1.DenseReluDense layer.0.SelfAttention	64 16	leakyrelu	0.3	None
	9	layer.0 layer.1.DenseReluDense layer.0.SelfAttention	128 128 32	gelu_new	0.3	None
	10	layer.0.SelfAttention layer.0 layer.1.DenseReluDense	32 128 64	tanh	0.0	None
	11	layer.1.DenseReluDense	128	gelu_new	0.0	layerNorm
	12	layer.0 layer.1.DenseReluDense layer.0.SelfAttention	64 64 16	swish	0.3	layerNorm

Just Accepted

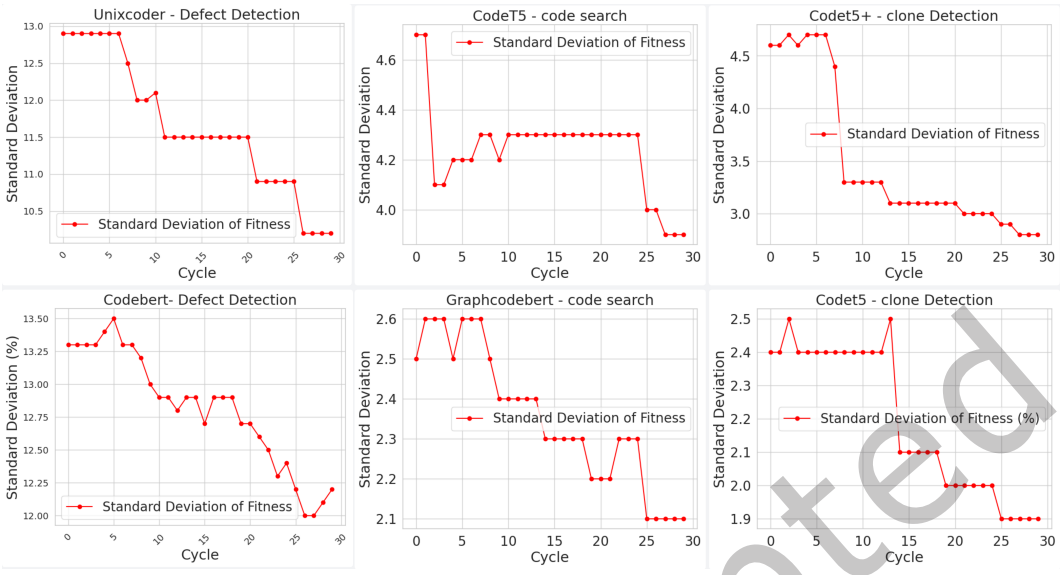


Fig. 9. Evolution of fitness standard deviation over optimization cycles for selected model-task combinations. Values are expressed as percentages. A reduction in standard deviation indicates convergence towards candidate solutions with improved performance.

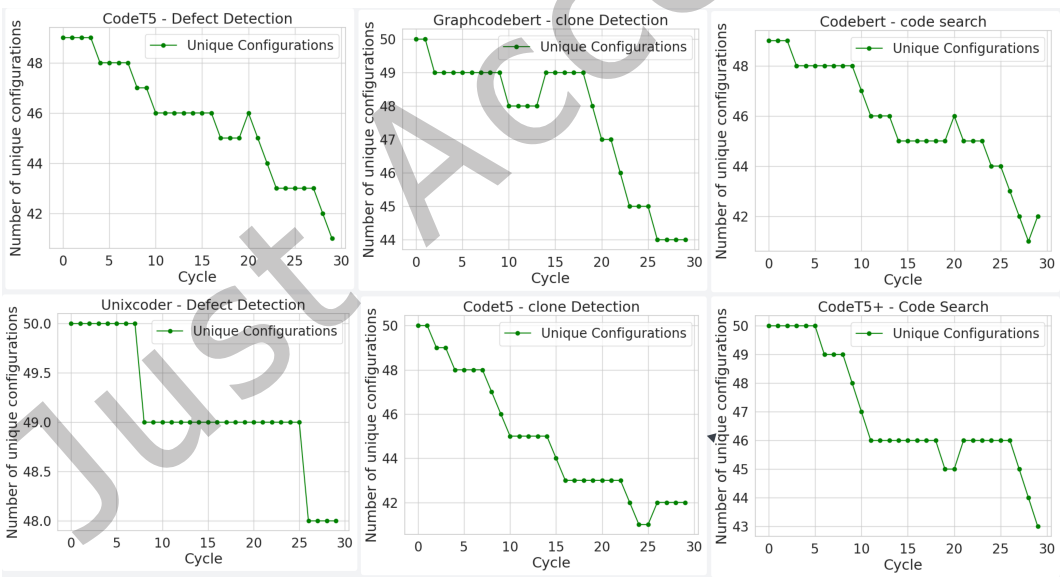


Fig. 10. Count of unique configurations within a population of 50 over optimization cycles. A decrease in this count indicates reduced diversity, as high-performing individuals dominate and yield a more homogeneous set of solutions.

Received 9 September 2024; revised 17 February 2025; accepted 19 April 2025

Just Accepted