The EarlyBIRD Catches the Bug: On Exploiting Early Layers of Encoder Models for More Efficient Code Classification

Anastasiia Grishina
anastasiia@simula.no
Simula Research Laboratory
Oslo, Norway

Max Hort
maxh@simula.no
Simula Research Laboratory
Oslo, Norway

Leon Moonen
leon.moonen@computer.org
Simula Research Laboratory &
BI Norwegian Business School
Oslo, Norway

ABSTRACT

The use of modern Natural Language Processing (NLP) techniques has shown to be beneficial for software engineering tasks, such as vulnerability detection and type inference. However, training deep NLP models requires significant computational resources. This paper explores techniques that aim at achieving the best usage of resources and available information in these models.

We propose a generic approach, EarlyBIRD, to build composite representations of code from the early layers of a pre-trained transformer model. We empirically investigate the viability of this approach on the CodeBERT model by comparing the performance of 12 strategies for creating composite representations with the standard practice of only using the last encoder layer.

Our evaluation on four datasets shows that several early layer combinations yield better performance on defect detection, and some combinations improve multi-class classification. More specifically, we obtain a +2 average improvement of detection accuracy on Devign with only 3 out of 12 layers of CodeBERT and a 3.3x speed-up of fine-tuning. These findings show that early layers can be used to obtain better results using the same resources, as well as to reduce resource usage during fine-tuning and inference.

CCS CONCEPTS

• **Software and its engineering**; • **Computing methodologies** → **Neural networks**; **Natural language processing**; *Information extraction*;

KEYWORDS

sustainability, model optimization, transformer, code classification, vulnerability detection, AI4Code, AI4SE, ML4SE

1 INTRODUCTION

Automation of software engineering (SE) tasks supports developers in creation and maintenance of source code. Recently, deep learning (DL) models have been trained on large open-source code corpora and used to perform code analysis tasks [1–4]. Motivated by the naturalness hypothesis stating that code and natural language share statistical similarities, researchers and tool vendors have started training deep NLP models on code and fine-tuning them on SE tasks [5]. Amongst others, such models have been applied to type inference [6], code clone detection [7], program repair [8–11], and defect prediction [12–15]. In NLP-based approaches, SE tasks are frequently translated to code classification problems. For example, detection of software vulnerabilities is a binary classification problem, bug type inference is a multi-class classification setting, and type inference is a multi-label multi-class classification task in case a type is predicted for each variable in the program.

Most modern NLP models build on the transformer architecture [16]. This architecture uses *attention* mechanism and consists of an *encoder* that converts an input sequence to a *representation* through a series of layers, followed by *decoder* layers that convert this representation to an output sequence. Although effective in terms of learning capabilities, the transformer design results in multi-layer models that need large amounts of data for training from scratch. A well-known disadvantage of these models is the high resource usage that is required for training due to both model and data sizes. While a number of pre-trained models have been published recently, fine-tuning these models for specific tasks still requires additional computational resources [4].

This paper explores techniques that aim at optimizing the use of resources and information available in the models during fine-tuning. In particular, we consider open white-box models, for which the weights from each layer can be extracted. We focus on encoder-only models, as they are commonly used for SE classification tasks, in particular, the transformer-based encoders. The standard practice in encoder models is to obtain the representation of the input sequence from the last layer of the model [17], while information from earlier layers is usually discarded [18]. In detail, the early layers are used only once at the inference stage, to obtain the last-layer representation, but are not used for representing the input directly. To exemplify the amount of discarded information at inference, when fine-tuning a 12-layered encoder, such as CodeBERT [17], for bug detection, 92% of the code embeddings are ignored.[1] However, it has been shown for natural language that early layers of an encoder capture lower-level syntactical features better than later layers [19–22], which can benefit downstream tasks.

Inspired by the line of research that exploits early layers of models, we propose EarlyBIRD,[2] a novel and generic approach for building composite representations from the early layers of a pre-trained encoder model. EarlyBIRD aims to leverage all available

This work is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license.

Pre-print of the paper accepted for publication in:
ESEC/FSE 2023, 3 - 9 December, 2023, San Francisco, USA
© 2023 Copyright held by the owner/author(s).
We plan to post an updated version to arXiv closer to the conference date.

[1] weights from 11 out of 12 layers
[2] Early-layer Based Improvement or Reduction of resources useD

information in existing pre-trained encoder models during fine-tuning to either improve results or achieve competitive results at reduced resource usage during code classification. We empirically evaluate EarlyBIRD on CodeBERT [17], a popular pre-trained encoder model for code, and four benchmark datasets that cover three tasks: defect detection with the Devign and ReVeal datasets [23, 24], bug type inference with the data from Yasunaga et al. [10], and exception type classification [13]. The evaluation compares the *baseline* representation that uses the last encoder layer with results obtained via EarlyBIRD. We both fine-tune the full-size encoder and its pruned version with only several early layers present in the model. The latter scenario analyzes the trade-off between only using a partial model and the performance impact on SE tasks.

Contributions: In this paper, we make the following contributions:
(1) We propose EarlyBIRD, an approach for creating composite representations of code using the early layers of a transformer-based encoder model. The goal is to achieve better code classification performance at equal resource usage or comparable performance at lower resource usage.
(2) We conduct a thorough empirical evaluation of the proposed approach. We show the effect of using composite EarlyBIRD representations while fine-tuning the original-size CodeBERT model on four real-world code classification datasets. We run EarlyBIRD with 10 different random initializations of non-fixed trainable parameters and mark the EarlyBIRD representations that yield statistically significant improvement over the baseline.
(3) We investigate resource usage and performance of pruned models. We analyze the trade-off between removing the later layers of a model and the impact this has on classification performance.

Main findings: With EarlyBIRD, we achieve performance improvements over the baseline code representation with the majority of representations obtained from single early layers on the defect detection task and selected combinations on bug type and exception type classification. Moreover, out of the reduced-size models with pruned later layers, we obtain a +2 average accuracy improvement on Devign with 3.3x speed-up of fine-tuning, as well as +0.4 accuracy improvement with 3.7x speed-up on average for ReVeal.

The remainder of the paper is organized as follows. We present related work in Section 2 and provide background details of the study in Section 3. The methodology is described in Section 4 which is followed by experimental setup in Section 5. We present and discuss results in Section 6 and conclude with Section 7.

2 RELATED WORK

Here, we give an overview of language models for SE tasks and recent encoder models, specifically, as well as different approaches to use early layers of encoder models.

2.1 Transformer Models in Software Engineering

The availability of open source code and increased hardware capabilities popularized training and usage of Deep Learning, including NLP and Large Language Models (LLMs), for SE tasks. To date, deep NLP models have already been applied in at least 18 SE tasks [25]. Pre-trained language models available for fine-tuning on SE tasks largely build on the transformer architecture, sequence-to-sequence models, and the attention mechanism [1, 9, 16]. One widely used benchmark to test different deep learning architectures on SE tasks is CodeXGLUE [4]. The benchmark provides data, source code for model evaluation, and a leader-board with models' performance on different tasks [4].

SE tasks can be translated to input sequence classification and generation of code or text. Examples of generative tasks in SE are code completion, code repair, generation of documentation from code and vice versa, and translation between different programming languages. Such tasks are frequently approached with neural machine translation models. Full transformer models for translation from a programming language (PL) to a natural language (NL) or PL-PL tasks include PLBART [26], PYMT5 [27], TFix [28], CodeT5 [29], Break-It-Fix-It [10]. Alternatively, generative models can include the decoder-only part of the transformer as in GPT-type models. In this case, the decoder both represents the input sequence and transforms it into the output sequence. Decoder-based models for code include, for example, Codex and CodeGPT [1, 4].

In the tasks that require code or documentation representation and their subsequent classification, the encoder-only architectures are used more frequently than in translation tasks. Examples of code classification problems are code clone detection, detection of general bugs, such as the presence of swapped operands, wrong variable names, syntax errors, or security vulnerabilities. A number of encoder models for code applied a widely-used bi-directional encoder, BERT [30], to pre-train it on code, with some modifications of the input. In this way, the CodeBERT [17], GraphCodeBERT [31], CuBERT [24], and PolyglotCodeBERT [32] models were created. In detail, the 12-layer RoBERTa-based CodeBERT model was pre-trained on NL-PL tasks in multiple PLs and utilized only the textual features of code. Note that RoBERTa is a type of BERT model with optimized hyper-parameters and pre-training procedures [33]. Together with the decoder-only CodeGPT model, the encoder-only CodeBERT model was used as a baseline in CodeXGLUE. GraphCodeBERT utilizes both textual and structural properties of code to encode its representations. PolyglotCodeBERT is the approach that improves fine-tuning of the CodeBERT model on a multi-lingual dataset for a target task even if the target task tests only one PL. This paper focuses on the fine-tuning strategies which, by contrast with PolyglotCodeBERT, do not increase the resource usage for fine-tuning. CuBERT is a 24-layer pre-trained transformer-based encoder tested on a number of code classification tasks, including exception type classification. We test the performance of the proposed EarlyBIRD composite representations on defect detection, including the use of one of CodeXGLUE benchmarks, as well as on error and exception type classification tasks. However, the goal of this paper is to achieve improvement over the baseline model when it is fine-tuned with composite code representations. We do not aim to compare results with other models, but rather propose an approach that is applicable to transformer-based encoders for source code and show its performance gains compared to the same model usage without the proposed approach.

2.2 Use of Early Encoder Layers

A number of studies explored different approaches to use information from early layers of DL models for sequence representation, such as probing single layers, pruning and variable learning rates. One way to leverage information from early model layers is to give

different priority to layers while fine-tuning the models [34, 35]. For example, the layer-wise learning rate decay (LLRD) strategy and re-initialization of late encoder layers yielded improvement over the standard fine-tuning of BERT on NLP tasks [36]. The LLRD strategy was initially developed to tune the later encoder layers with larger learning rate. In this way, the later layers can be better adapted to a downstream task under consideration, because the later layers are assumed to learn complex task-specific features of input sequences [34]. Moreover, Peters et al. [37] showed that the performance of fine-tuning improves if the encoder layers are updated during fine-tuning in comparison with training only the classifier on top of fixed (frozen) encoder layers.

Pruning later layers of transformer models is another way to consider only early layers for fine-tuning [38–40]. Sajjad et al. [40] investigated how the performance of transformer models on NLP is affected when reducing their size by pruning layers. They considered six pruning strategies, including dropping from different directions, alternated layer dropping, or dropping layers based on importance, for four pre-trained models: BERT [30], RoBERTa [33], XLNET [41], ALBERT [42]. By pruning model layers, Sajjad et al. were able to reduce the number of parameters to 60% of the initial parameter set while maintaining a high level of performance. While the performance on downstream tasks varies in their study, the lower layers are critical for maintaining performance when fine-tuning for downstream tasks. In other words, dropping earlier layers is detrimental to performance. Overall, pruning layers reduces model size and in turn reduces fine-tuning and inference time. In line with the work of Sajjad et al. [40], we extend our experiments with the pruning of later layers and keeping earlier layers present in the model (see RQ2 in Section 6).

The use of information from single early layers in a number of EarlyBIRD experiments is also inspired by Peters et al. [20]. In their study, Peters et al. present an empirical evidence that language models learn syntax and part-of-speech information on earlier layers of a neural network, while more complex information, such as semantics and co-reference relationships, are captured better by deeper (later) layers. In another study, Karmakar and Robbes probed pre-trained models of code, including CodeBERT, on tasks of understanding syntactic information, structure complexity, code length, and semantic information [18]. While Karmakar and Robbes probed frozen early layers of different models for code in a single strategy, we use 12 different strategies for combining unfrozen early layers during fine-tuning and focus on the tasks of bug detection or bug type classification. The novelty of our study with respect to Karmakar and Robbes is that we combine early layers in addition to extracting each of them, while Karmakar and Robbes extracted early layer representations and used them without composing new representations.

3 ENCODERS FOR CODE CLASSIFICATION

In this section, we present the background on the transformer architecture and different uses of encoder-decoder – or full transformer – architecture, encoder-only, and decoder-only variants. Because our study focuses on encoder-only open-source models available for fine-tuning, the distinction between transformer types is necessary for understanding the methodology afterwards.

In sequence-to-sequence generation scenarios, the transformer model consists of a multi-layer encoder that represents the input sequence and a decoder that generates the output sequence based on the sequence representation from the encoder and the available output generated at previous steps [16]. For source code classification tasks, the transformer is frequently reduced to only its encoder followed by a *classification head*, a component added to the encoder to categorize the representation into different classes. Dropping the decoder for classification is motivated by resource efficiency, because the decoder is conceptually only needed for token generation from the input sequence. During classification of an input, the encoder represents the sequence and passes it to the classification head. Based on this design, a number of pre-trained encoders have been published in recent years, such as BERT and RoBERTa which were pre-trained on natural language, and similar models pre-trained on code, or a combination of code and natural language [30, 33]. The goal of pre-training in the *pre-train and fine-tune* scenario is to capture language patterns in general, so that they can serve as a basis for domain-specific downstream tasks. Pre-trained models can be fine-tuned on different downstream tasks in NLP and SE.

Processing the input sequence for classification consists of several steps: *tokenization, initial embedding, encoding* the sequence with an encoder, and passing the sequence representation through a *classification head*. Tokenization splits the input sequence, adds special tokens, matches the tokens to their ID's in the vocabulary of tokens, and unifies the resulting token length for samples in a dataset. Embedding transforms the one-dimensional token ID to an initial multi-dimensional static vector representation of the token and is usually a part of the pre-trained encoder model. This representation is updated using the attention mechanism of the encoder. Because of attention, the representation of the input is influenced by all tokens in the sequence, so it is contextualized.

CodeBERT is a RoBERTa-based model with 12 encoder layers pre-trained on 6 programming languages (Python, Java, JavaScript, PHP, Ruby, and Go), as well as text-to-code tasks [17]. Pre-training was done on the masked language modeling (MLM) and replaced token detection (RTD) tasks. These tasks respectively train the model to derive what token is masked in MLM, and in RTD predict whether any token in an original sequence is swapped with a different token that should not be in the sequence. CodeBERT outputs a bidirectional encoder representation of the input sequence, which means that the model considers context from pre-pending and subsequent words to represent each token in the input sequence.

A pre-trained model is usually released with a pre-trained tokenizer. The pre-trained tokenizer ensures that token ID's correspond to those processed during pre-training. The tokenizer also adds special tokens, such as a `CLS` token at the start of each input sequence, `PAD` tokens to unify lengths of input sequences, and the `EOS` token to signify the end of the input string and the start of padding sequence [30]. All tokens are transformed by the model in each encoder layer. Out of all tokens, the `CLS` token representation from the last layer, which is updated by all encoder layers, is typically used as a representation for the full sequence.

The standard practice of using the `CLS` token from the last encoder layer is motivated by the pre-training procedure. For example, in MLM, the model predicts the masked token based on the `CLS`

token representation from the 12^{th} layer of BERT and CodeBERT. However, the choice of token to represent the full sequence in fine-tuning can be different. For example, in PLBART [26], a transformer model for code with both an encoder and a decoder, the EOS token is used for representing the input sequence. In this paper, we propose different ways to represent the input sequence and use information from early layers of the model in an effective way.

4 METHODOLOGY

In this paper, the architecture of the code classification model consists of five parts: (1) a tokenizer, (2) an embedding layer, (3) an encoder with several layers, (4) a set of operations to combine sequence representations from encoder layers with EarlyBIRD, and (5) a classification head. The output of each step is used as input into the next step. An overview of the architecture is shown in Figure 1 and described below. The main difference between this architecture and the classification architecture discussed in Section 3 is step (4); the standard architecture only consists of steps (1–3) and (5).

Steps (1)–(3) use a pre-trained tokenizer, embedder, and encoder. EarlyBIRD is formulated in a generic way and can be applied to any encoder, but for our experiments, we fix the CodeBERT model and tokenizer. In step (4), we combine the information from all the layers or some of the early layers of the encoder as opposed to the baseline that uses the last layer of the encoder. Finally, the classification head in step (5) consists of one dropout layer and one linear layer with softmax.

The encoder model represents each token of an input sequence with a vector of size H, also known as hidden size. For each input sequence of length S, and a hidden size H, we obtain a matrix of size $S \times H$ for each of L layers of the base model as shown in Figure 1. For example, the CodeBERT architecture is fixed with 12 encoder layers, i.e., $L = 12$ for that model. All the information available in the encoder for one input sequence is stored in a tensor of size $L \times S \times H$. The EarlyBIRD combinations must produce one vector $\vec{R}$ of size H that represents the input, as shown in Figure 1. Keeping the output code representation of size H is required to provide a fair comparison of EarlyBIRD composite representations with the standard code representation obtained from the last layer. In this way, the dimension of the classification head is the same for all combinations of early layers and has minimal possible influence during fine-tuning.

To combine code representations from early layers, we use maximum pooling, slicing of one layer or one CLS token, and the weighted sum of representations over layers or tokens in an input sequence. We also experiment with different sizes of the model. The combination strategies that use all layers of the pre-trained model are divided into two categories: the strategies that use CLS tokens from the encoder layers; the strategies that use more tokens than just CLS from encoder layers.

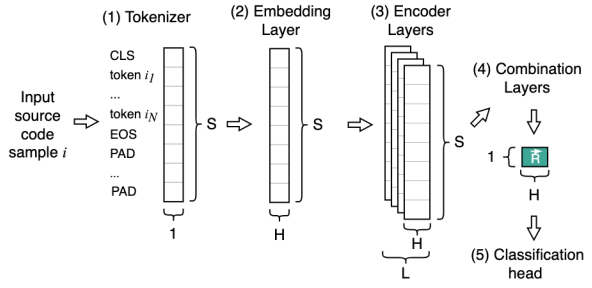


Figure 1: Model architecture for code classification.

The following combinations involve only CLS tokens:

(i) *baseline*: CLS token from the last layer, i.e., layer no. L;
(ii) CLS token from one layer[3] no. l, $l = 1, \ldots (L-1)$;
(iii) max pool over CLS tokens from all layers $\{l \mid l = 1 \ldots L\}$;
(iv) weighted sum over CLS tokens from all layers $\{l \mid l = 1 \ldots L\}$.

The second set of combinations uses representations of all the tokens in tokenized input sequences, including the CLS token:

(v) max pool tokens from one layer no. l, $l = 1 \ldots L$;
(vi) max pool over all layers for each token in the input sequence, max pool over tokens;
(vii) max pool over all layers for each token in the input sequence; weighted sum over tokens;
(viii) max pool over all tokens for each layer no. l, $l = 1 \ldots L$; weighted sum over layers
(ix) weighted sum over tokens from one layer no. l, $l = 1 \ldots L$;
(x) weighted sum over tokens for each one layer no. l, $l = 1 \ldots L$; weighted sum over all layers;
(xi) weighted sum over all layers for each token in the input sequence; weighted sum over all tokens.

Note that weights in the weighted sums are learnable parameters. However, the added number of learnable parameters for fine-tuning constitutes 0.00042%[4] of the number of learnable parameters in the baseline configuration. For this reason, we mention that the models with combinations (ii-x) have the same model size while bearing in mind the overhead of learnable weights in the weighted sums.

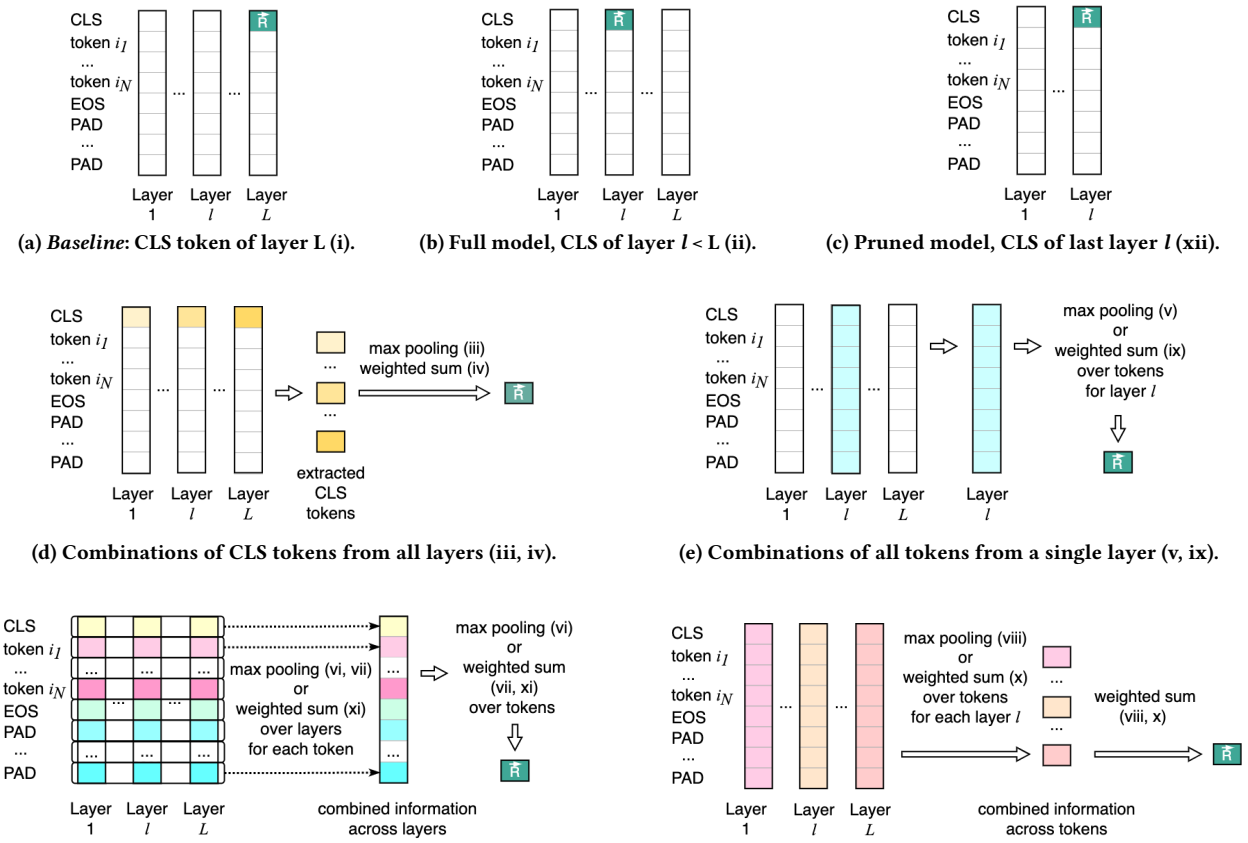
In addition to experiments with token combinations, we also investigate performance of the model with first $l < L$ layers and the baseline token combination, described as follows:

(xii) CLS token from the last layer of the model with $l < L$ encoder layers.

Note that the baseline combination (i) with the usage of the CLS token from layer L corresponds to (ii) and (xii) if $l = L$.

The combinations are presented in Figure 2. Similar combinations are presented close to each other or are combined in the same image if they only have minor differences and share the major parts. For example, in Figure 2c, we illustrate combinations (iii) and (iv), because both of them use CLS tokens from all layers combined using max pooling or weighted sum. The roman numbers which indicate combination types are preserved either in the descriptions below the figures or in the figures themselves, but the order is changed. We mention combination number corresponding to the description in the current section, such as baseline combination (i) in Figure 2a or combination (ii) for CLS token from one early layer in Figure 2b. We highlight what parts of encoder layer outputs are used for each

[3] We use each layer l in this combination separately and mark the set of layers $\{l \mid l = 1 \ldots L\}$; if several layers are used at once.
[4] Weighted sum over tokens adds $S = 512$ learnable weights. Because the weights of the sum are shared across the layers, the maximum number of added weights is $L + H = 524$ out of 124M learnable weights in the base model. Combinations without weighted sums do not add extra learnable parameters to the base model. Weighted sum over layers adds learnable $L = 12$ weights for CodeBERT.



(a) *Baseline*: CLS token of layer L (i). **(b) Full model, CLS of layer l < L (ii).** **(c) Pruned model, CLS of last layer l (xii).**

(d) Combinations of CLS tokens from all layers (iii, iv). **(e) Combinations of all tokens from a single layer (v, ix).**

(f) Combinations of all tokens combined across layers using max pooling or weighted sum (vi, vii, xi). **(g) Combinations of all tokens combined for each single layer first using max pooling or weighted sum (viii, x).**

Figure 2: Combinations of early encoder layers that lead to code representation vector $\vec{R}$ for each tokenized input sequence. The latin numbering in brackets corresponds to the combinations described in Section 4.

combination with color. White cells correspond to the tokens that are not used in early layer combinations. The goal of all combinations is to obtain a vector representation $\vec{R}$ for each input code sample. For example, in Figure 2a, we consider the last layer L and extract only the CLS token marked as $\vec{R}$.

Another remark on the EarlyBIRD combinations concerns the usage of all tokens or only code tokens. Code tokens are those that correspond to tokenized input words or sub-words and are shown in Figure 2 as $\text{token}_{i_1}, ..., \text{token}_{i_N}$ for an input sequence i of size i_N. For each combination that uses more than just a CLS token, i.e., combinations (v-xi), we experiment with code tokens only, as well as with all tokens, including CLS, EOS, and PAD. The motivation to check code tokens exclusively stems from the hypothesis that information in special tokens may introduce noise into results.

5 EXPERIMENTAL SETUP

In this section, we describe the datasets used for empirical evaluation and implementation details of fine-tuning with the proposed EarlyBIRD approach. We investigate binary and multi-task code classification scenarios to explore generalisability of our results.

5.1 Datasets

We fine-tune and test the CodeBERT model using the EarlyBIRD approach on four datasets. The datasets span three tasks: defect detection, error type classification and exception type classification — with 2, 3, and 20 classes, respectively. They also contain data in two programming languages, C++ and Python. In addition, the chosen datasets have similar train subset sizes. In this way, we aim to reduce the effect of the model's exposure to different amounts of training data during fine-tuning. Statistics of the datasets are provided in Table 1. We report the size of the train/validation/test splits. In addition, we compute the average number of tokens in the input sequences upon tokenization with the pre-trained CodeBERT tokenizer. Because the maximum input sequence size for the CodeBERT model is limited to $S = 512$, the number of tokens is indicative of how much information the model gets access to or how much information is cut off, in case of long inputs.

Devign: This dataset contains functions in C/C++ from two open-source projects labelled as vulnerable or non-vulnerable [23]. We reuse the train/validation/test split from the CodeXGLUE Defect

Table 1: Statistics of Fine-Tuning Datasets.

Dataset	# classes	Avg # tokens	# code samples		
			Train	Valid	Test
Devign	2	614	21,854	2,732	2,732
ReVeal	2	512	18,187	2,273	2,274
BIFI	3	119	20,325	2,259	15,055
Exception Type	20	404	18,480	2,088	10,348

detection benchmark.[5] The dataset is balanced: the ratio of non-vulnerable functions is 54%.

ReVeal: Similarly to Devign, ReVeal is a vulnerability detection dataset of C/C++ functions [13]. The dataset is not balanced: it contains 90% non-vulnerable code snippets. Both the Devign and ReVeal datasets contain real-world vulnerable and non-vulnerable functions from open-source projects.

Break-It-Fix-It (BIFI): The dataset contains function-level code snippets in Python with syntax errors [10]. We use the original buggy functions and formulate a task of classifying the code into three classes: Unbalanced Parentheses with 43% of the total number of code examples in BIFI, Indentation Error with 31% code samples, Invalid Syntax containing 26% samples. The provided train/test split is reused, and the validation set is extracted as 10% of train data.

Exception Type: The dataset consists of short functions in Python with an inserted `__HOLE__` token in place of one exception in code.[6] The task is to predict one of 20 masked exception types for each input function and is unbalanced. The dataset was initially created from the ETH Py150 Open corpus[7] as described in the original paper [24]. We reuse the train/validation/test split provided by the authors.

5.2 Implementation

The architecture is based on the CodeBERT[8] tokenizer and encoder model. The model defines the maximum sequence length, hidden size, and has 12 layers, so $S = 512$, $H = 768$, $L = 12$. Hyper-parameters in the experiments are set to $B = 64$, learning rate is $1e$-5, and dropout probability is 0.1. If the tokenized input sample is longer than $S = 512$, we prune the tokens in the end to make the input fit into the model. We run fine-tuning with Adam optimizer and testing for each combination 10 times with different seeds for 10 epochs and report the performance for the best epoch on average over 10 runs. The best epoch is defined by measuring accuracy on a validation set. We use Python 3.7 and Cuda 11.6, and run experiments on one Nvidia Volta A100 GPU.

5.3 Evaluation Metrics

To present the impact of early layer combinations, we compare the accuracy on the test set for all datasets, because it allows us to compare our results with other benchmarks. In addition, we report weighted F1-score denoted as F1 (w) for a detailed analysis of selected combinations to account for class imbalance. To obtain the weighted F1-score, the regular F1-score is calculated for each label and their weighted mean is taken. The weights are equal to the number of samples in a class.

We also report results of the Wilcoxon signed-rank test on the corresponding metrics for the combinations that show improvement over the baseline [43]. The Wilcoxon test is a non-parametric test suitable for the setting in which different model variants are tested on the same test set, because it is a paired test. The Wilcoxon test checks the null hypothesis whether two related paired samples come from the same distribution. We reject the null hypothesis if p-value is less than $\alpha = 0.05$. In case we obtain improvement of a metric over the baseline with an EarlyBIRD combination and the null hypothesis is rejected, we conclude that the combination performs better and the result is statistically significant.

5.4 Research Questions

During our empirical evaluation of composite EarlyBIRD code representations, we address the following research questions:

RQ1. Composite Code Representations with Same Model Size: What is the effect of using combinations (ii-xi) of early layers with the same model size in comparison to the baseline approach of using only the `CLS` token from the last layer, i.e., combination (i), for code representation on model performance in the code classification scenario? The goal is to find out whether any of the EarlyBIRD combination types work consistently better for different datasets and tasks.

RQ2. Pruned Models: What is the effect of reducing the number of pre-trained encoder layers in combinations (xii) on resource usage and model performance on code classification tasks? As opposed to RQ1, in which we consider the combinations that do not reduce the model size, this research question is devoted to investigation of the trade-off between using less resources with reduced-size models and performance variation in terms of classification metrics.

For both research questions, we evaluate the composite representations on binary and multi-task code classification scenarios to explore generalisability of the results obtained for the binary case. We investigate if and what combinations result in better performance, averaged over 10 runs with different seeds. For combinations that improve the baseline on average, we also explore if the results are statistically significant according to the Wilcoxon test.

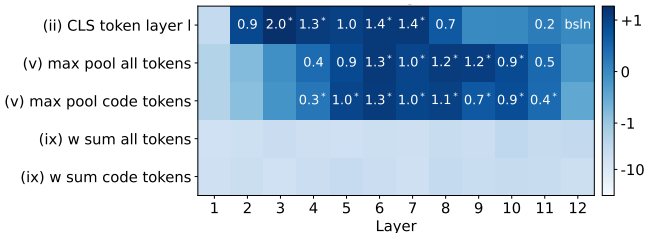
6 RESULTS AND DISCUSSION

6.1 EarlyBIRD with Fixed-Size Models

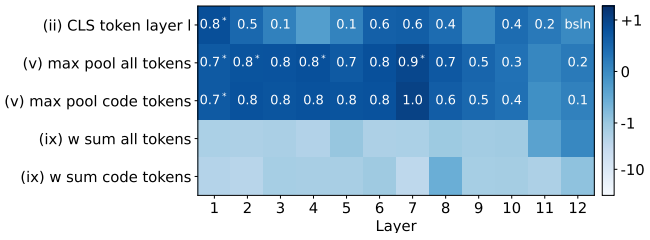
To answer RQ1, we explore one-layer combinations, multi-layer combinations, and estimate statistical significance of the performance improvement.

6.1.1 Combinations of Tokens in Single Selected Early Layers. We show the difference between average accuracy obtained with each combination that uses only one selected early layer in Figure 3. In addition, we note the value of the metric improvement at the crossing of the combination type and the layer number if the combination performs better on average than the baseline. Otherwise, we leave the combinations that have negative or neutral effect colored

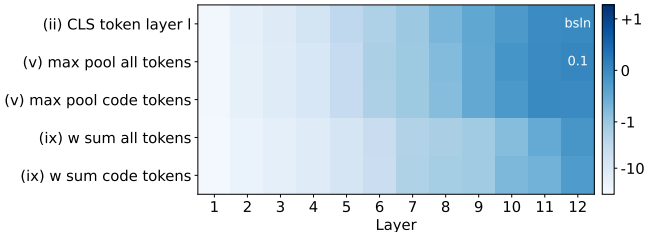
[5] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection
[6] https://github.com/google-research/google-research/tree/master/cubert
[7] https://www.sri.inf.ethz.ch/py150
[8] https://huggingface.co/microsoft/codebert-base



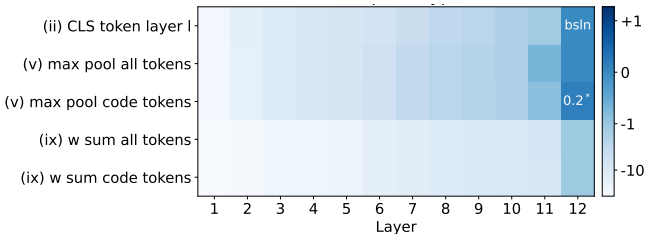
(a) Devign, accuracy.



(b) ReVeal, accuracy.

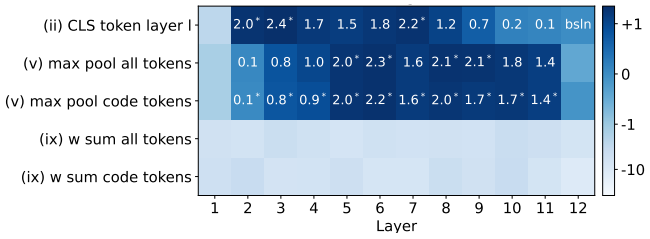


(c) BIFI, accuracy.

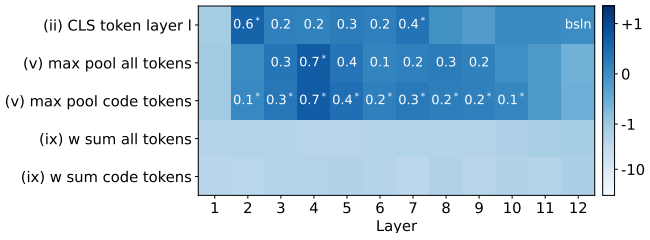


(d) Exception Type, accuracy.

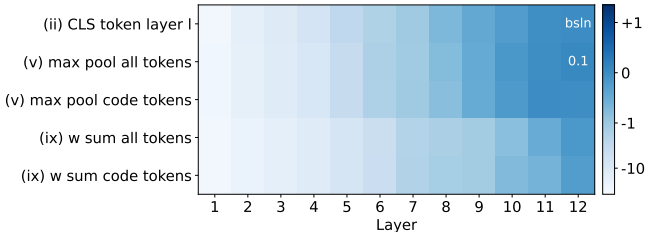
Figure 3: Absolute difference of mean accuracy between EarlyBIRD and baseline (bsln) performance. The marker * indicates statistically significant improvements.



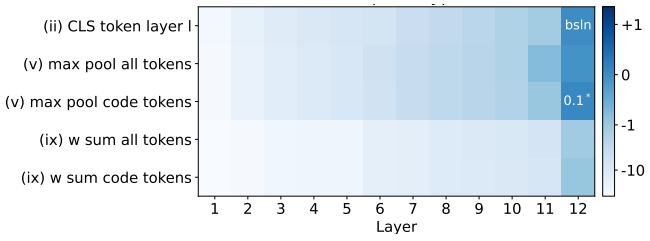
(a) Devign, F1 (w).



(b) ReVeal, F1 (w).



(c) BIFI, F1 (w).



(d) Exception Type, F1 (w).

Figure 4: Absolute difference of mean weighted F1-scores (F1 (w)) between EarlyBIRD and baseline (bsln). The marker * indicates statistically significant improvements.

but not annotated. Statistically significant improvements according to the Wilcoxon test are marked with * next to the improvement details. Combinations that correspond to the baseline are marked with "bsln" and have zero difference, correspondingly. The results for weighted F1-score are similar to accuracy. They are visualized in the same way in Figure 4.

The first rows in Figures 3a and 3b correspond to the combinations (ii) CLS token layer l. With this combination type, average improvement over the baseline is achieved with the majority of early layers. Specifically, we have obtained accuracy improvements ranging from +0.2 to +2.0 for Devign in 8 out of 11 layers, and accuracy improvement from +0.1 to +0.8 for ReVeal in 9 out of 11 layers. The dynamics of the metric change over selected layer numbers is different for Devign and ReVeal. In detail, the average performance of combination (ii) is best with layer 3 on Devign (a +2.0 accuracy improvement) and with layer 1 for ReVeal (a +0.8 accuracy improvement). The best improvement in terms of F1 (w) matches with layer 3 for Devign and is observed at layer 2 for ReVeal as shown in Figure 4.

Max pooling over all available tokens from a selected layer in combination (v) also achieves performance improvement over the baseline, as shown in rows 2 and 3 of Figures 3a, 3b. In general, layers 4–11 yield higher accuracy and layers 2–11 higher F1 (w) with max pooling for Devign than the baseline. For ReVeal, all layers except layer 11 result in better average accuracy and layers 2–10 — higher average F1 (w). Max pooling over all tokens, including

special tokens, achieves the best statistically significant average improvement of accuracy of +0.9 of all combinations for ReVeal.

The weighted sum of all tokens or code tokens exclusively in combination (ix) does not improve the baseline performance. We assume that fine-tuning for 10 epochs is not enough for this type of combination, because the loss at epoch 10 on both training and validation splits is higher for combinations (ix) than for combinations with max pooling. Since the goal of this study is to use the same or less resources for fine-tuning, we have not fine-tuned this combination for more than 10 epochs.

While combinations (ii) and (v) perform better for the majority of layers on the defect detection task, multi-class classification for bug or exception type prediction does not benefit from the combinations to the same extent as the binary task. Only max pooling of tokens of the last encoder layer achieves better performance than the baseline for BIFI (+0.1 accuracy, +0.1 weighted F1-score improvements) and Exception Type (a +0.2 accuracy, +0.1 weighted F1-score improvements) datasets.

The impact of using all tokens or code tokens exclusively depends on the dataset. The difference between performance of single-layer combinations with max pooling of all tokens and only code tokens constitute 0.0-0.1 accuracy or F1 (w). For the multi-class tasks, the average results improve with the use of each later layer in the model. We obtain performance improvement with the max pooling combination (v), while other one-layer combinations do not perform better than the baseline.

The best performing results on Devign and Exception Type classification datasets are statistically significant according to the Wilcoxon test. For ReVeal, the second best result is statistically significant. We have not obtained statistically significant improvements for BIFI. We explain it by the fact that the baseline metric is already high, i.e., 96.7 accuracy. Achieving improvement is usually more challenging when the baseline performs at this level.

In essence, the combinations that involve CLS tokens corresponding to the single layer (ii), as well as the max pooling combinations (v) perform better on average for defect detection datasets Devign and Reveal. However, only the max pooling combination (v) of tokens from the last encoder layer outperforms the baseline on average for multi-class datasets BIFI and Exception Type. The weighted sum of tokens from a selected layer (ix) performs worse than the baseline if fine-tuned for the same number of epochs for all tasks. Multi-class classification tasks require the information from the last layer for better performance in our experiments, while the binary task of defect detection allows us to use early layers and improve the performance over the baseline.

6.1.2 Multi-Layer Combinations. The average performance difference with the baseline of combinations that utilize early layers is shown in Figures 5 and 6. We annotate the difference with the baseline if a combination outperforms the baseline on average and add a star (*) to the number if the improvement is statistically significant.

If we use all information from the available layers, the improvement over the baseline is less than what is observed in Section 6.1.1, where one specific layer has been used. In detail, out of combinations that involve CLS tokens from all early layers, no combination performs better than the baseline for ReVeal, BIFI, or Exception Type

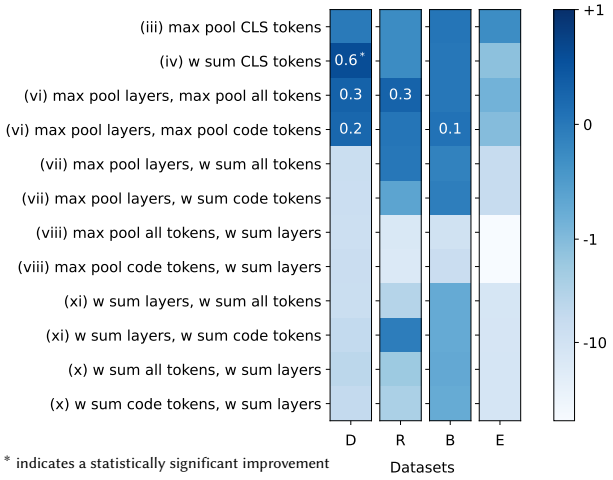


Figure 5: Absolute difference of average accuracy between EarlyBIRD and baseline performance on D (Devign), R (ReVeal), B (BIFI), E (Exception Type).

datasets. However, the best improvement (+0.6 accuracy) out of experiments with all layers is obtained on Devign with the weighted sum of CLS tokens in the combination (iv), which is less than the maximum improvement with the combinations from one selected early layer in Section 6.1.1. The improvement of F1 (w) is shown in Figure 6. We obtained slightly better improvements of F1 (w) for Devign, no F1 (w) improvement for the unbalanced ReVeal dataset. The average F1 (w) difference with the baseline for multi-class tasks are the same as accuracy difference.

If we consider the combinations that involve all tokens, the combination (vi) with two max pooling operations outperforms the baseline for Devign, Reveal, and BIFI with accuracy improvement between +0.1 and +0.3. No combination that involves all layers outperforms the baseline on average for Exception Type dataset. Combinations that involve one max pooling and one weighted sum of all tokens perform worse or neutral in comparison with the baseline. The combinations with only weighted sums perform worse than the baseline on average.

> **Answer to RQ1.** EarlyBIRD achieves statistically significant accuracy and F1-score improvements for defect detection datasets by using single-layer combinations that involve the CLS token or max pooling over all tokens. For bug type and exception type classification, max pooling of the tokens from the last encoder layer has improved the performance. Weighted sum of tokens does not improve performance over the baseline.

6.2 Pruned Models

This section is devoted to the combinations of early layers that are initialized with the first $l < L$ early layers from the pre-trained model and fine-tuned as l-layer models — combinations (xii). We start by comparing the performance of using the CLS token from layer l of the full-size model, i.e., combination (ii), and using the CLS token from layer l of the model that has l layers in total — combination (xii). Figure 7 presents average accuracy obtained with these two combinations depending on the used layer, as well

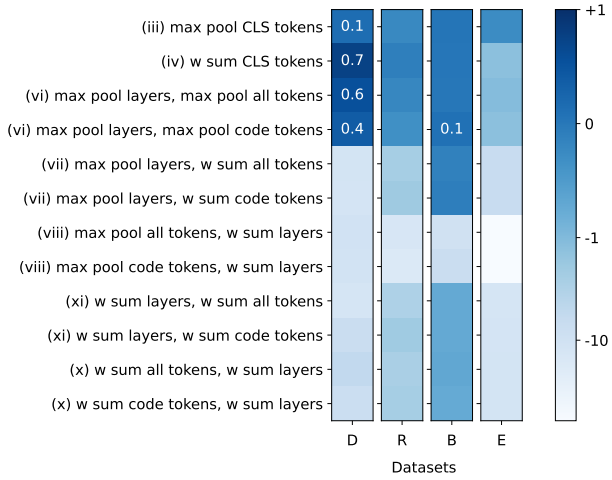


Figure 6: Absolute difference of mean weighted F1-score between combinations and baseline. Datasets are abbreviated to D (Devign), R (ReVeal), B (BIFI), E (Exception Type).

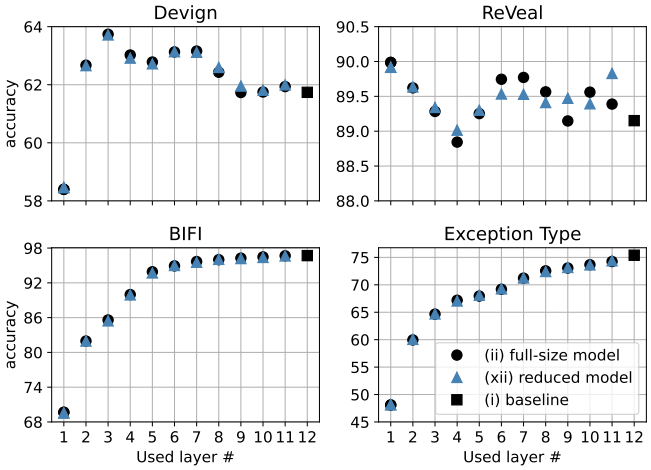


Figure 7: Model performance with a subset of $l < L$ layers (xii) vs. models with all layers (ii); CLS token from layer l.

as the baseline combination of using CLS from the last layer $L = 12$ of CodeBERT. On average, the pruned models with reduced size perform on par with the full-size model for defect detection on the balanced Devign dataset, and for bug type and exception type classification. However, the performance of the two analogical combinations diverges for the unbalanced defect detection dataset ReVeal in layers 4 and 6–11.

Most importantly, the results show that reducing the model size and using the CLS token from the last layer of the reduced model performs on par with the baseline for the defect detection task. The best improvement with the reduced model is achieved with the 3-layer encoder for Devign and the 1-layer encoder for ReVeal. This result shows that it is possible to both reduce resources and improve the model's performance during fine-tuning on the defect detection task with both a balanced and unbalanced dataset.

To explore the trade-off between resource usage and performance degradation for bug type and exception type identification, we show the average speed-up of one fine-tuning epoch and the performance loss compared to the baseline for BIFI and Exception Type datasets in Table 2. We also report the corresponding values for Devign and ReVeal, for which both gains and losses of performance are indicated. The speed up is reported as a scaling factor of the baseline time. The metric difference is shown as gain or loss of the weighted F1-score and accuracy compared to the baseline performance. Statistically significant improvements are reported in bold, while statistically insignificant losses are marked with a star (*). We also underline and discuss selected results that improve the metric values and reduce resource usage.

The majority of combinations (xii) with pruned models outperform the baseline for Devign and ReVeal. Furthermore, models with 2–10 layers show statistically significant improvements of both metrics on Devign, with the 3-layer model achieving +2 accuracy improvement with a 3.3-times average speed-up of fine-tuning with the same hardware and software. Not only does the 3-layer model improve the accuracy over CodeBERT baseline to 63.7, but also outperforms several other models tested on Devign and reported on the CodeXGLUE benchmark [4]. In particular, the 3-layer CodeBERT model outperforms the full-transformer model PLBART [26], and code2vec code representations pre-trained on abstract syntax trees and code tokens in a joint manner [26]. However, our pruned model does not outperform the best performing model reported on CodeXGLUE, CoText, that achieves 66.62 accuracy [44].

Models with 1 and 11 layers achieve statistically significant accuracy improvements for ReVeal. However, the 1-layer model reduces the F1 (w) score. The use of layer 11 does not impact the speed of fine-tuning, while the 1-layer model yields the 3.7x acceleration of the baseline fine-tuning speed. The lack of speed-up with 11-layer model can be explained by the fact that the number of trainable parameters does not decrease linearly with the removal of later layers, since the additional embedding layer and classification head remain unchanged. The 2-layer model results in the best improvement of F1 (w) which is statistically significant. The 2-layer model improves accuracy on ReVeal as well.

For BIFI, we obtain statistically insignificant decrease of F1 (w) and accuracy according to the Wilcoxon test which brings about 1.2x speed-up of the fine-tuning with the 11-layer model. If we decrease the number of layers to 8, the performance on BIFI stays within the (baseline metric−1) limit, but we gain up to 1.7x average speed-up of one-epoch fine-tuning. In case of using models with 1–10 layers, we observe a statistically significant change of distribution and decrease of metric values.

For the unbalanced Exception Type dataset, the performance drops faster and the speed-up is less prominent than for BIFI. The change of mean values of the metrics for all models is statistically significant. In detail, the metrics decrease by -1.0 absolute metric value at 11 layers with 1.1x fine-tuning speed-up and by -1.8 with 10 layers with 1.2x speed-up. We explain the sharper decline of the combinations performance by the lower baseline metric values (75.39 accuracy, 75.30 weighted F1-score) than in the case of BIFI (96.7 accuracy and weighted F1-score). We conclude that for the BIFI dataset with high-performing baseline and 3 classes, the performance loss at removing each layer is less than for the Exception Type classification dataset with lower baseline performance and 20 classes. The resource usage, which is correlated with time spent on

Table 2: Comparison of reduced size models with the baseline. We report metric performance for the baseline and difference with the baseline for reduced models, average time for one-epoch fine-tuning (Time) in mm:ss format, speed up and performance variation obtained with models with *l* layers. Statistically significant improvements are marked in bold, statistically insignificant performance losses are marked with *. Best metric improvement with highest speed-up factors are underlined.

l	Devign				ReVeal				BIFI				Exception Type			
	Time	Speed-up	Acc	F1(w)	Time	Speed-up	Acc	F1(w)	Time	Speed-up	Acc	F1(w)	Time	Speed-up	Acc	F1(w)
12	8:50	1.0x	61.74	60.4	6:57	1.0x	89.15	88.53	8:41	1.0x	96.7	96.7	7:22	1.0x	75.39	75.3
11	8:03	1.1x	+0.3	-0.1*	6:56	1.0x	**+0.6**	+0.3	7:07	1.2x	-0.1*	-0.1*	6:33	1.1x	-1.0	-1.0
10	7:13	1.2x	+0.1	+0.3	6:20	1.1x	+0.2	-0.0*	6:22	1.4x	-0.3	-0.3	6:01	1.2x	-1.8	-1.8
9	6:40	1.3x	+0.3	+0.5	5:53	1.2x	+0.3	-0.1*	5:46	1.5x	-0.5	-0.5	5:29	1.3x	-2.2	-2.3
8	5:52	1.5x	**+0.9**	+1.1	5:15	1.3x	+0.2	-0.1*	5:13	1.7x	-0.6	-0.6	4:55	1.5x	-3.0	-3.0
7	5:23	1.6x	**+1.4**	**+2.2**	4:44	1.5x	+0.3	+0.2	4:43	1.8x	-1.1	-1.1	4:20	1.7x	-4.1	-4.4
6	4:54	1.8x	**+1.4**	**+2.0**	4:25	1.6x	+0.3	+0.1	4:10	2.1x	-1.7	-1.7	3:50	1.9x	-6.1	-6.6
5	4:03	2.2x	**+1.0**	**+1.5**	3:45	1.9x	+0.1	+0.2	3:31	2.5x	-3.0	-3.0	3:15	2.3x	-7.3	-7.7
4	3:22	2.6x	**+1.2**	**+1.6**	3:06	2.2x	-0.2*	+0.2	2:53	3.0x	-6.8	-6.8	2:41	2.7x	-8.3	-9.2
3	2:41	3.3x	**+2.0**	**+2.4**	2:28	2.8x	+0.1	+0.3	2:15	3.8x	-11.3	-11.3	2:10	3.4x	-10.7	-12.0
2	2:00	4.4x	**+1.0**	**+1.9**	1:52	3.7x	+0.4	**+0.6**	1:34	5.5x	-14.7	-14.8	1:34	4.7x	-15.4	-17.1
1	1:18	6.8x	**-3.2**	-2.3	1:13	5.7x	**+0.8**	-1.2	0:58	9.0x	-27.2	-27.3	0:59	7.4x	-27.3	-30.7

tuning, decreases faster for BIFI than for Exception Type. This is partially explained by a larger classification head for the Exception Type dataset, because this dataset has 20 classes as opposed to only 3 classes in BIFI.

> **Answer to RQ2.** We obtain performance improvements over the baseline as well as fine-tuning speed-ups for both defect detection datasets by using the `CLS` token from the last layer of pruned models. For multi-class classification, performance decreases upon pruning each layer from the end of the model. The decrease is sharper for the dataset with 20 exception types than for the task with 3 bug types.

6.3 Threats to Validity

The main threat to external validity is that the results are empirical and may not generalize to all code classification settings, including other programming languages, tasks, and encoder-based models for code. We have tested EarlyBIRD combinations on code in C for defect detection and Python for bug type and exception type classification in this study. The choice of the CodeBERT as the encoder model and its internal structure affects the results. For instance, an encoder model that takes smaller input sequences can perform worse on the same datasets, because larger parts of input code sequences have to be pruned in this case. The external validity can be improved by testing on more datasets and encoder models.

The threats to internal validity concern the dependency of models on initializations of trainable parameters and the choice of methods. Classification head and weighted sums with trainable parameters in our experiments depend on the initialization of the parameters and can lead the model to arrive at different local minima during fine-tuning. To reduce the effect of different random initializations, we have fine-tuned and tested all EarlyBIRD combinations 10 times with different random seeds.

In addition, we used the Wilcoxon test to verify whether the achieved improvements are statistically significant. However, the Wilcoxon test only estimates whether measurements of baseline values and EarlyBIRD combinations are drawn from different distributions. The reported times spent on fine-tuning and corresponding speed-ups have the purpose of illustrating the reduction in resource usage, and will depend on the hardware used. Even when using factors of speed-up for pruned models, there is a chance that these numbers will be different on other hardware configurations.

We implemented the algorithms and statistical procedures in Python, with the help of widely used libraries such as PyTorch, NumPy and SciPy. However, we cannot guarantee the absence of implementation errors which may have affected our evaluation.

7 CONCLUDING REMARKS

In this paper, we have proposed EarlyBIRD, an approach to combine early layers of encoder models for code, and tested different early-layer combinations on the software engineering tasks of defect detection, bug type and exception type classification. Our study is motivated by the hypothesis that early layers contain valuable information that is discarded by the standard practice of representing the code with the `CLS` token from the last encoder layer. EarlyBIRD provides ways to improve the performance of existing models with the same resource utilization, as well as for resource usage reduction while obtaining comparable results to the baseline.
Results: Using EarlyBIRD, we obtain statistically significant improvements over the baseline for the majority of the combinations that involve a single encoder layer on defect detection, and with selected EarlyBIRD combinations on bug type and exception type classification. Max pooling of tokens from selected single layers yields performance improvements for all datasets. Both the classification performance and the average fine-tuning time for one epoch are improved by pruning the pre-trained model to its early layers and using the `CLS` token from the last layer of the pruned model. For defect detection, this results in a +2.0 increase in accuracy and a 3.3x fine-tuning speed-up on Devign, and up to +0.8 accuracy improvement with a 3.7x speed-up on ReVeal. Pruned models do not lead to multi-class classification performance gains, but they

do show a fine-tuning speed-up and the associated decrease in resource consumption.

The results show that pruned models with reduced size either work better or can result in a reduction of resource usage during fine-tuning with different levels of performance variation, which indicates the potential of EarlyBIRD in resource-restricted scenarios of deploying defect detection and but type classification in production environments. For example, EarlyBIRD achieves a 2.1x speed-up for BIFI while reducing accuracy from 96.7 to 95.0.

Future Work: The study can be extended by investigating the generalization to other encoder models, such as CuBERT [24] which uses 24 encoder layers. Moreover, it would be of interest to experiment with other code classification tasks, such as general bug detection and the prediction of vulnerability types. The latter could be investigated using the CWE types from the Common Weakness Enumeration as labeled in the CVEfixes dataset [45].

ACKNOWLEDGMENTS

This work is supported by the Research Council of Norway through the secureIT project (IKTPLUSS #288787). The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

DATA AVAILABILITY

To support open science and allow for replication and verification of our work, all artifacts are made available through Zenodo at the following URL: https://doi.org/10.5281/zenodo.7608802.

REFERENCES

[1] M. Chen et al. *Evaluating Large Language Models Trained on Code*. July 2021. DOI: 10.48550/arXiv.2107.03374. arXiv: 2107.03374.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton. "A Survey of Machine Learning for Big Code and Naturalness." In: *ACM Computing Surveys* 51.4 (July 2018), 81:1–81:37. DOI: 10/gffkhm.

[3] T. Sharma, M. Kechagia, S. Georgiou, R. Tiwari, and F. Sarro. *A Survey on Machine Learning Techniques for Source Code Analysis*. Oct. 2021. DOI: 10.48550/arXiv.2110.09610. arXiv: 2110.09610.

[4] S. Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation." In: *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*. Dec. 2021.

[5] P. Devanbu. "New Initiative: The Naturalness of Software." In: *Proceedings - International Conference on Software Engineering*. Vol. 2. IEEE Computer Society, 2015, pp. 543–546. DOI: 10/ghjj4z.

[6] V. J. Hellendoorn, C. Bird, E. T. Barr, and M. Allamanis. "Deep Learning Type Inference." In: *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, Oct. 2018, pp. 152–162. DOI: 10/gf8npn.

[7] G. Zhao and J. Huang. "DeepSim: Deep Learning Code Functional Similarity." In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Lake Buena Vista FL USA: ACM, Oct. 2018, pp. 141–151. DOI: 10/ghhrrs.

[8] H. Ye, M. Martinez, and M. Monperrus. "Neural Program Repair with Execution-Based Backpropagation." In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. New York, NY, USA: Association for Computing Machinery, July 2022, pp. 1506–1518. DOI: 10/gqrnmm.

[9] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. "SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair." In: *IEEE Transactions on Software Engineering* (2019). DOI: 10/ggssk2.

[10] M. Yasunaga and P. Liang. "Break-It-Fix-It: Unsupervised Learning for Program Repair." In: *International Conference on Machine Learning*. PMLR, 2021, p. 12.

[11] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung. "VulRepair: A T5-based Automated Software Vulnerability Repair." In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 935–947. DOI: 10/grnvb7.

[12] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning." In: *International Conference on Machine Learning and Applications (ICMLA)*. Orlando, FL: IEEE, Dec. 2018, pp. 757–762. DOI: 10/ggssk7.

[13] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. "Deep Learning Based Vulnerability Detection: Are We There Yet?" In: *IEEE Transactions on Software Engineering* 48.9 (Sept. 2022), pp. 3280–3296. DOI: 10/gk52qr.

[14] H. Wei, G. Lin, L. Li, and H. Jia. "A Context-Aware Neural Embedding for Function-Level Vulnerability Detection." In: *Algorithms* 14.11 (Nov. 2021), p. 335. DOI: 10/gq4v3n.

[15] C. Pan, M. Lu, and B. Xu. "An Empirical Study on Software Defect Prediction Using CodeBERT Model." In: *Applied Sciences* 11.11 (May 2021), p. 4793. DOI: 10/gn246h.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. "Attention Is All You Need." In: *International Conference on Neural Information Processing Systems (NeurIPS)*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., 2017, pp. 5998–6008.

[17] Z. Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online, 2020, pp. 1536–1547. DOI: 10/gj58gj.

[18] A. Karmakar and R. Robbes. "What Do Pre-Trained Code Models Know about Code?" In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2021, pp. 1332–1336. DOI: 10/gq2phc.

[19] T. Blevins, O. Levy, and L. Zettlemoyer. "Deep RNNs Encode Soft Hierarchical Syntax." In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Melbourne, Australia: Association for Computational Linguistics, 2018, pp. 14–19. DOI: 10/gmcgnx.

[20] M. Peters, M. Neumann, L. Zettlemoyer, and W.-t. Yih. "Dissecting Contextual Word Embeddings: Architecture and Representation." In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 1499–1509. DOI: 10/ggfw2s.

[21] N. F. Liu, M. Gardner, Y. Belinkov, M. E. Peters, and N. A. Smith. *Linguistic Knowledge and Transferability of Contextual Representations*. Apr. 2019. DOI: 10.48550/arXiv.1903.08855. arXiv: 1903.08855 [cs].

[22] C. Sun, X. Qiu, Y. Xu, and X. Huang. *How to Fine-Tune BERT for Text Classification?* Feb. 2020. DOI: 10.48550/arXiv.1905.05583. arXiv: 1905.05583 [cs].

[23] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu. "Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics

via Graph Neural Networks." In: *International Conference on Neural Information Processing Systems (NeurIPS)*. Vancouver, Canada.: Curran Associates, Inc., 2018, p. 11.

[24] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi. "Learning and Evaluating Contextual Embedding of Source Code." In: *Proceedings of the 37th International Conference on Machine Learning*. PMLR, Nov. 2020, pp. 5110–5121.

[25] C. Niu, C. Li, B. Luo, and V. Ng. *Deep Learning Meets Software Engineering: A Survey on Pre-Trained Models of Source Code*. May 2022. DOI: 10.48550/arXiv.2205.11739. arXiv: 2205.11739 [cs].

[26] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. "Unified Pre-training for Program Understanding and Generation." In: *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Online: Association for Computational Linguistics, 2021, pp. 2655–2668. DOI: 10/gp6kqv.

[27] C. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan. "PyMT5: Multi-Mode Translation of Natural Language and Python Code with Transformers." In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Online: Association for Computational Linguistics, 2020, pp. 9052–9065. DOI: 10/gm3pbm.

[28] B. Berabi, J. He, V. Raychev, and M. Vechev. "TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer." In: *International Conference on Machine Learning*. Vol. 139. Virtual Event: PMLR, 2021, pp. 780–791.

[29] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation." In: *Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, 2021, pp. 8696–8708. DOI: 10/gp6kqt.

[30] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." In: *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL, Vol 1)*. Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: 10/ggbwf6.

[31] D. Guo et al. "GraphCodeBERT: Pre-training Code Representations with Data Flow." In: *International Conference on Learning Representations, ICLR 2021*. Virtual Event, Austria: OpenReview.net, May 2021, pp. 1–18.

[32] T. Ahmed and P. Devanbu. "Multilingual Training for Software Engineering." In: *Proceedings of the 44th International Conference on Software Engineering*. Pittsburgh Pennsylvania: ACM, May 2022, pp. 1443–1455. DOI: 10/gq4xbf.

[33] Y. Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. July 2019. DOI: 10.48550/arXiv.1907.11692. arXiv: 1907.11692 [cs].

[34] J. Howard and S. Ruder. "Universal Language Model Fine-tuning for Text Classification." In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Melbourne, Australia: Association for Computational Linguistics, July 2018, pp. 328–339. DOI: 10/ggsc9c.

[35] C. Sun, X. Qiu, Y. Xu, and X. Huang. "How to Fine-Tune BERT for Text Classification?" In: *Chinese Computational Linguistics*. Ed. by M. Sun, X. Huang, H. Ji, Z. Liu, and Y. Liu. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 194–206. DOI: 10/gg578d.

[36] T. Zhang, F. Wu, A. Katiyar, K. Q. Weinberger, and Y. Artzi. "Revisiting Few-sample BERT Fine-tuning." In: *NeurIPS 2021*. Mar. 2021. arXiv: 2006.05987.

[37] M. E. Peters, S. Ruder, and N. A. Smith. "To Tune or Not to Tune? Adapting Pretrained Representations to Diverse Tasks." In: *Proceedings of the 4th Workshop on Representation Learning for NLP (RepL4NLP-2019)*. Florence, Italy: Association for Computational Linguistics, Aug. 2019, pp. 7–14. DOI: 10/ggvxqn.

[38] A. Fan, E. Grave, and A. Joulin. *Reducing Transformer Depth on Demand with Structured Dropout*. Sept. 2019. DOI: 10.48550/arXiv.1909.11556. arXiv: 1909.11556 [cs, stat].

[39] D. Peer, S. Stabinger, S. Engl, and A. Rodriguez-Sanchez. "Greedy-Layer Pruning: Speeding up Transformer Models for Natural Language Processing." In: *Pattern Recognition Letters* 157 (May 2022), pp. 76–82. DOI: 10/grmn6h. arXiv: 2105.14839 [cs].

[40] H. Sajjad, F. Dalvi, N. Durrani, and P. Nakov. "On the Effect of Dropping Layers of Pre-trained Transformer Models." In: *Computer Speech & Language* 77 (Jan. 2023), p. 101429. DOI: 10/grmn6j. arXiv: 2004.03844 [cs].

[41] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. Jan. 2020. DOI: 10.48550/arXiv.1906.08237. arXiv: 1906.08237.

[42] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. Feb. 2020. DOI: 10.48550/arXiv.1909.11942. arXiv: 1909.11942.

[43] F. Wilcoxon. "Individual Comparisons by Ranking Methods." In: *Breakthroughs in Statistics: Methodology and Distribution*. Ed. by S. Kotz and N. L. Johnson. Springer Series in Statistics. New York, NY: Springer, 1992, pp. 196–202. DOI: 10.1007/978-1-4612-4380-9_16.

[44] L. Phan, H. Tran, D. Le, H. Nguyen, J. Annibal, A. Peltekian, and Y. Ye. "CoTexT: Multi-task Learning with Code-Text Transformer." In: *Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Online: Association for Computational Linguistics, 2021, pp. 40–47. DOI: 10/gmf9t5.

[45] G. Bhandari, A. Naseer, and L. Moonen. "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software." In: *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. ACM, 2021, pp. 30–39. DOI: 10/gpd4hb.