# Improving Code Search with Multi-Modal Momentum Contrastive Learning

Zejian Shi[1], Yun Xiong[1 2], Yao Zhang[1], Zhijie Jiang[3*], Jinjing Zhao[4], Lei Wang[3], Shanshan Li[3]
[1]Shanghai Key Laboratory of Data Science, School of Computer Science, Fudan University, China
[2]Peng Cheng Laboratory, Shenzhen, China
[3]School of Computer, National University of Defense Technology, China
[4]National Key Laboratory of Science and Technology on Information System Security, China
{zjshi20,yunx,yaozhang}@fudan.edu.cn, {jiangzhijie,leiwang,shanshanli}@nudt.edu.cn, zhjj0420@126.com

***Abstract*—Contrastive learning has recently been applied to enhancing the BERT-based pre-trained models for code search. However, the existing end-to-end training mechanism cannot sufficiently utilize the pre-trained models due to the limitations on the number and variety of negative samples. In this paper, we propose MoCoCS, a multi-modal momentum contrastive learning method for code search, to improve the representations of query and code by constructing large-scale multi-modal negative samples. MoCoCS increases the number and the variety of negative samples through two optimizations: integrating multi-batch negative samples and constructing multi-modal negative samples. We first build momentum contrasts for query and code, which enables the construction of large-scale negative samples out of a mini-batch. Then, to incorporate multi-modal code information, we build multi-modal momentum contrasts by encoding the abstract syntax tree and the data flow graph with a momentum encoder. Experiments on CodeSearchNet with six programming languages demonstrate that our method can further improve the effectiveness of pre-trained models for code search.**

***Index Terms*—code search, contrastive learning, multi-modal momentum contrast, pre-trained model**

## I. Introduction

Code search aims to retrieve matching code snippets from a codebase with natural language queries, which has become a hot research topic in program comprehension. Recent research on code search mainly focuses on deep learning-based methods [1]–[3], which can be divided into two categories: supervised learning-based methods and self-supervised learning-based methods. To bridge the semantic gap between natural language queries and code snippets, supervised learning-based methods [4]–[9] train a neural model from scratch, which respectively encodes the query and code into a joint embedding space.

To increase the parameters and expressiveness of neural models, self-supervised learning-based methods [10]–[12] adopt a pre-training and fine-tuning paradigm [13]–[15] to train much larger neural models, which commonly refers to the pre-trained models (PTMs). The pre-trained models first learn general-purpose representations from large amounts of unlabeled data by designing pre-training objectives. Then the pre-trained models can be fine-tuned to downstream tasks by using task-oriented labeled data.

*Corresponding author

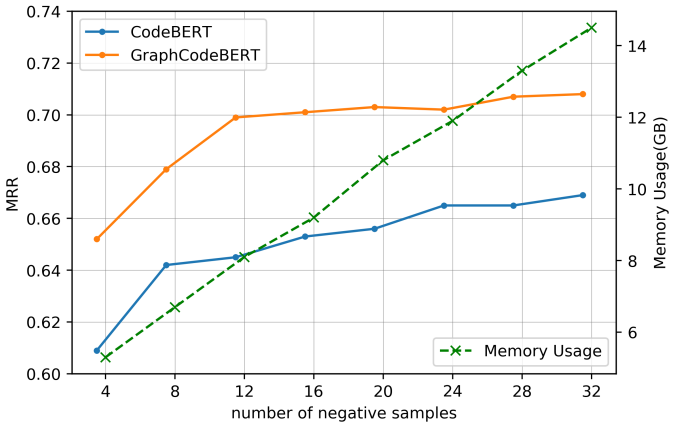


Fig. 1. MRR scores and memory usage of CodeBERT [10] and GraphCodeBERT [11] on the CodeSearchNet Ruby corpus. The memory usage of CodeBERT and GraphCodeBERT is the same. The negative samples are constructed by mismatching query-code pairs in the current batch. The number is for a single query or code.

Recently, several BERT-based pre-trained models [16]–[18] for programming languages have achieved promising results on code search by using the end-to-end training mechanism related to the contrastive loss [19]. With the contrastive loss, the distance between positive query-code pairs is narrowed in the joint embedding space and the distance between negative query-code pairs is enlarged in the joint embedding space [17].

However, we observe that the existing end-to-end training mechanism cannot sufficiently utilize the capabilities of the pre-trained models due to the limitations on the number and variety of negative samples. As shown in Figure 1, as the number of negative samples increases, the Mean Reciprocal Rank (MRR) scores of CodeBERT [10] and GraphCodeBERT [11] are constantly improved. More negative samples mean a larger batch size, which leads to a linear increase in memory usage. The existing end-to-end training mechanism restricts the selection of negative samples within a mini-batch, which makes it difficult to increase the number of negative samples with the fixed batch size [20]. And the batch size also cannot be increased easily beyond the limit of physical memory. Although the MRR score still has a trend to be further improved, the negative samples cannot continue to increase to a sufficiently large number.

Besides, the existing end-to-end training mechanism for contrastive learning lacks the variety of negative samples [21]. In the code search task, the common practice is to treat mismatching query-code pairs as negative samples, which only considers code information in a single modality for constructing negative samples. The code snippet contains rich semantic and structural information, such as the abstract syntax tree (AST) and the data flow graph (DFG), which can be used to help understand the code semantics. To improve the code understanding ability of pre-trained models, incorporating multi-modal code information is a crucial process.

To address the above limitations, we propose MoCoCS, a multi-modal momentum contrastive learning method for code search, to improve the representations of query and code by constructing large-scale multi-modal negative samples. To expand the scale of negative samples, we introduce Momentum Contrast [22], [23] into code search, which enables the construction of large-scale negative samples out of a mini-batch. Specifically, given a batch of code snippets and natural language queries, we use a momentum encoder to encode the query and the code respectively. In addition, we initialize memory banks as queues for storing query and code embeddings encoded by the momentum encoder. In the fine-tuning stage, all embeddings in the queues are used to construct negative samples. Thus, we can obtain large-scale negative samples containing multiple batches without increasing the batch size. Compared with the linear increase in memory usage caused by increasing the batch size, the momentum encoder and queue only require little extra memory. With our method, the pre-trained models can break the limitations of mini-batch and memory, and continue to increase the number of negative samples.

Moreover, to incorporate multi-modal code information, we use the momentum encoder to encode the abstract syntax tree and the data flow graph respectively and build multi-modal momentum contrasts for query, code, abstract syntax tree, and data flow graph. We additionally initialize an AST queue and a DFG queue for storing AST embeddings and DFG embeddings encoded by the momentum encoder. In the fine-tuning stage, all AST and DFG embeddings in the queues are also used to construct negative samples. With such large-scale multi-modal negative samples, the pre-trained models can be enhanced to learn better representations of query and code.

To evaluate the effectiveness of MoCoCS, we conduct comprehensive experiments on the six programming languages of CodeSearchNet [24]. We used three different pre-trained models as the initial models of our method, CodeBERT, GraphCodeBERT, and UniXcoder [12]. The pre-trained CodeBERT fine-tuned with our method improves the MRR scores by 4.6% on average. The pre-trained GraphCodeBERT fine-tuned with our method improves the MRR scores by 4.9% on average. The pre-trained UniXcoder fine-tuned with our method improves the MRR scores by 1.7% on average.

To sum up, the main contributions of this paper are as follows:

- We propose MoCoCS, a multi-modal momentum contrastive learning method for code search, to improve the representations of query and code by constructing large-scale multi-modal negative samples.
- We build multi-modal momentum contrasts for query, code, abstract syntax tree, and data flow graph. By incorporating multi-modal code information through contrastive loss, the pre-trained models are enhanced to learn better representations of query and code.
- We conduct comprehensive experiments on the six programming languages of CodeSearchNet to evaluate our MoCoCS. Experimental results demonstrate that our method can be applied to enhancing pre-trained models for code search in different programming languages.

## II. Background

### A. Self-supervised Code Representation

Code representation refers to representing the source code as dense embeddings in the embedding space, which is a crucial step in code understanding. Different from learning task-specific code representations directly from task-specific data, self-supervised methods [25]–[27] learn general-purpose representations from large amounts of unlabeled data by designing pre-training objectives. Masked Language Modeling (MLM) is the most commonly used pre-training objective to learn contextual code representations. The contextual embeddings are calculated based on the context of code tokens with an attention mechanism. For a certain code token, different contexts will produce different embeddings, which can incorporate better contextual semantics.

### B. Self-supervised Code Search

Self-supervised methods on code search also adopt the pre-training and fine-tuning paradigm, which have achieved promising results to learn query and code representations by using BERT-based pre-trained models [10]–[12]. As shown in Figure 2, we can divide the whole process of the code search task into three stages: the pre-training stage, the fine-tuning stage, and the inference stage. In the pre-training stage, the pre-trained model learns the bimodal representations of query and code by designing pre-training objectives. The most commonly used pre-training objective is also the Masked Language Modeling (MLM), which concatenates the code snippet with the natural language query and randomly masks some tokens with a special token $[MASK]$. The BERT-based pre-trained model is trained to predict the correct values of masked tokens to learn contextual representations of query and code tokens.

In the fine-tuning stage, the parameters of pre-trained models are fine-tuned to better fit the code search task. Given a batch of ($q_i$, $c_i$) pairs, we can encode the query and code into the embedding space with the pre-trained model. With the contrastive loss, the distance between positive query-code pairs is narrowed in the joint embedding space and the distance

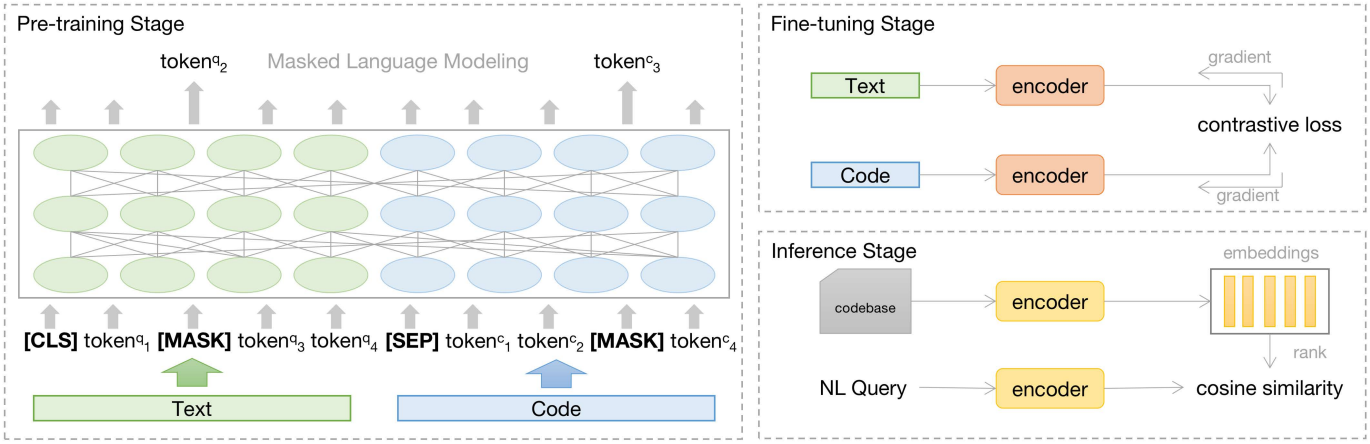


Fig. 2. Three stages of self-supervised code search. In the **pre-training stage**, the pre-trained model learns general-purpose representations of query and code by designing pre-training objectives. In the **fine-tuning stage**, the pre-trained model is fine-tuned to better fit the code search task. In the **inference stage**, the pre-trained model is used to rank candidate code snippets by calculating cosine similarity scores.

between negative query-code pairs is enlarged in the joint embedding space. The contrastive loss is calculated as follows:

$$-log\frac{exp(\boldsymbol{q_i^T}\cdot\boldsymbol{c_i}/t)}{exp(\boldsymbol{q_i^T}\cdot\boldsymbol{c_i}/t)+\sum_{c_i^-}exp(\boldsymbol{q_i^T}\cdot\boldsymbol{c_i^-}/t)}, \tag{1}$$

where $(\boldsymbol{q_i}, \boldsymbol{c_i})$ are the embeddings of $(q_i, c_i)$, $t$ is the temperature coefficient and $\boldsymbol{c_i^-}$ is the code embedding unpaired with $\boldsymbol{q_i}$ in the current batch.

In the inference stage, the pre-trained model is used to rank candidate code snippets by calculating cosine similarity scores. Given a natural language query, we use the pre-trained model to encode the query and all code snippets in the codebase. According to the similarity scores between query and code embeddings, the candidate code snippets can be ranked. The cosine similarity score is calculated as:

$$sim(\boldsymbol{q}_i, \boldsymbol{c}_i) = \frac{\boldsymbol{q_i}^T\boldsymbol{c_i}}{\|\boldsymbol{q_i}\|\,\|\boldsymbol{c_i}\|}. \tag{2}$$

### C. Momentum Contrastive Learning

Contrastive learning [28]–[32] has drawn great attention in both the CV and NLP communities. The most commonly used training mechanism for contrastive learning is the end-to-end mechanism [33], [34], which updates the model parameters through back-propagation. The end-to-end mechanism can only construct negative samples within a mini-batch, which is limited by the physical memory size. Another training mechanism for contrastive learning is the memory bank mechanism [35]. The memory bank mechanism initializes a memory space for storing a large-scale of negative samples. However, the negative samples in the memory bank are constructed from different batches, resulting in an inconsistency problem. To solve the inconsistency problem of large-scale negative samples, Momentum Contrast uses momentum contrastive learning to obtain consistent large-scale negative samples.

The core of momentum contrastive learning is the momentum-updated encoder and the queue for storing negative samples. Generally, there are two types of encoders for momentum contrastive learning, a momentum-updated encoder, and a gradient-updated encoder. The parameters of the gradient-updated encoder are updated by contrastive loss. The parameters of the momentum-updated encoder are updated through the parameters of the gradient-updated encoder. The calculation is as follows:

$$\theta_m = m\theta_m + (1-m)\theta_g, \tag{3}$$

where $m \in [0, 1)$ is the momentum coefficient, $\theta_m$ is the parameter of the momentum-updated encoder and $\theta_g$ is the parameter of the gradient-updated encoder.

The momentum coefficient $m$ determines the speed of updating the parameters of the momentum encoder. To ensure the consistency of negative samples from different batches, the parameters of the momentum encoder need to be updated at a slow speed. Therefore, the momentum coefficient $m$ is set to a relatively large value. In addition, each batch of embeddings encoded by the momentum encoder is stored in the queue by a first-in-first-out strategy. During the training stage, all embeddings in the queue are used to construct negative samples.

## III. Method

In this section, we propose MoCoCS, a novel method to improve code search with multi-modal momentum contrastive learning. We first introduce an overview of our method. Then, the implementation of MoCoCS will be elaborated.

### A. An Overview of MoCoCS

Figure 3 is an overall framework of our proposed MoCoCS. We will introduce MoCoCS in four parts. (a) Query and code embeddings. This part is the process of encoding natural language queries and code snippets into query and code embeddings. (b) AST and DFG embeddings. This part is the process of encoding abstract syntax tree and data flow graph into AST and DFG embeddings. (c) Multi-modal momentum contrastive learning. This part is designed to build multi-modal momentum contrasts for query, code, abstract syntax tree, and

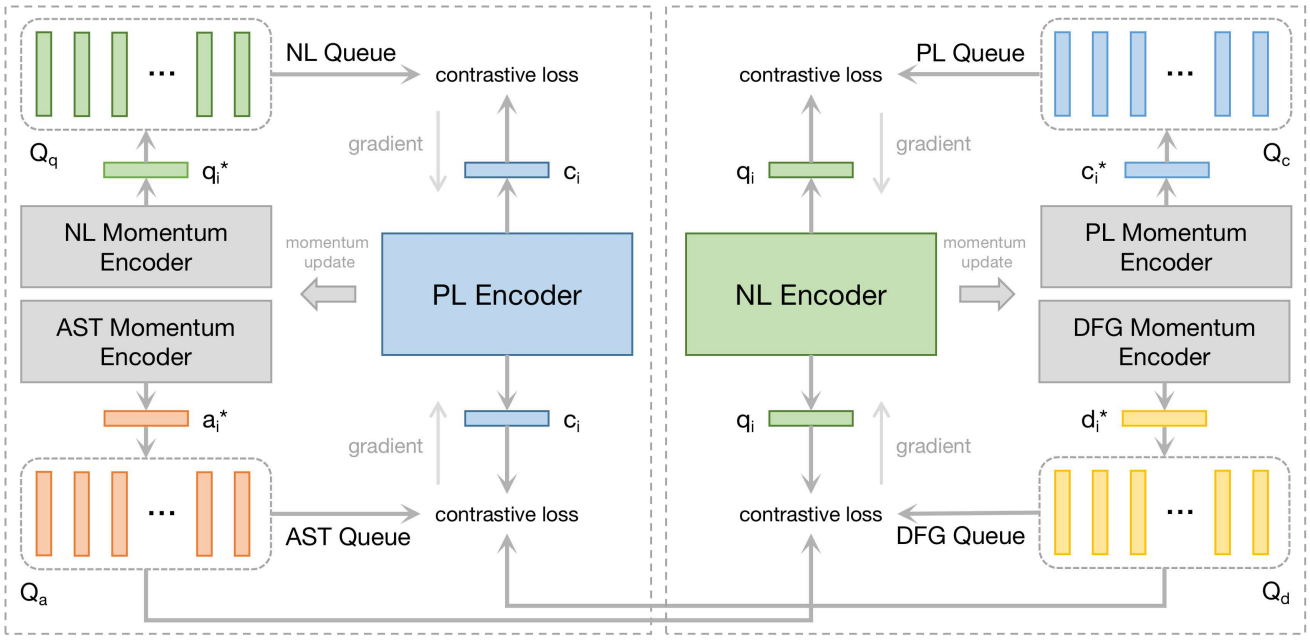


Fig. 3. The framework of our MoCoCS. All encoders adopt transformer-based architectures. **PL Encoder** and **NL Encoder** share the same parameters, which are used to obtain code embeddings and query embeddings respectively. **PL Queue** stores the code embeddings encoded by **PL Momentum Encoder**. **NL Queue** stores the query embeddings encoded by **NL Momentum Encoder**. **AST Queue** stores the ast embeddings encoded by **AST Momentum Encoder**. **DFG Queue** stores the dfg embeddings encoded by **DFG Momentum Encoder**. All the momentum encoders share the same parameters.

data flow graph. (d) Model fine-tuning. This part introduces the fine-tuning stage of pre-trained models for better fitting the downstream code search task.

### *B. Query and Code Embeddings*

Given a batch of $N$ query-code pairs $(q_i, c_i)$, we can use the BERT-based pre-trained model to encode natural language queries and code snippets into the embedding space, such as CodeBERT, GraphCodeBERT. After encoding, we can obtain the query and code embeddings:

$$\begin{aligned} \boldsymbol{q_i} &= Encoder(q_i), \\ \boldsymbol{c_i} &= Encoder(c_i), \end{aligned} \tag{4}$$

where $i \in [0, N)$, $\boldsymbol{q_i}$ and $\boldsymbol{c_i}$ are embeddings of $q_i$ and $c_i$, $Encoder$ refers to the pre-trained model used to encode $q_i$ and $c_i$.

To be specific, the encoding process is the same as RoBERTa [36] and consists of three steps: tokenization, converting tokens to word embeddings, and encoding with Transformer blocks [37]. Tokenization is to tokenize text data into tokens:

$$\begin{aligned} token_1^q, ..., token_n^q &= tokenize(q_i), \\ token_1^c, ..., token_n^c &= tokenize(c_i), \end{aligned} \tag{5}$$

where $tokenize$ is the operator of tokenization, $token_n^q$ and $token_n^q$ refer to the subwords in the vocabulary, obtained by Byte-Pair Encoding (BPE) [38]. BPE is a subword-level representation that allows handling large vocabularies common in natural language corpora.

For each token sequence $tokens^q = [token_1^q, ..., token_n^q]$ and $tokens^c = [token_1^c, ..., token_n^c]$, we concatenate a $[CLS]$ token at the front and a $[SEP]$ token at the end. Through a word embedding matrix, each token can be mapped into a word embedding:

$$\begin{aligned} \boldsymbol{E_q} &= map(tokens^q), \\ \boldsymbol{E_c} &= map(tokens^c), \end{aligned} \tag{6}$$

where $map$ is the function that maps the token to a embedding $\boldsymbol{e_i}$ in the word embedding matrix, $\boldsymbol{E_q} = [\boldsymbol{e_1^q}, ..., \boldsymbol{e_n^q}]$ and $\boldsymbol{E_c} = [\boldsymbol{e_1^c}, ..., \boldsymbol{e_n^c}]$. The word embedding matrix is the non-contextual representation of words, which are learnable parameters.

To obtain contextual embeddings for queries and code, the most common practice is to encode word embeddings with Transformer encoder blocks:

$$\begin{aligned} \boldsymbol{q_i} &= transformer\_block(\boldsymbol{E_q} + \boldsymbol{P_q}), \\ \boldsymbol{c_i} &= transformer\_block(\boldsymbol{E_c} + \boldsymbol{P_c}), \end{aligned} \tag{7}$$

where $transformer\_block$ contains a multi-head attention layer and a feed-forward layer, $\boldsymbol{P_q}$ and $\boldsymbol{P_c}$ denote position embeddings, which are learnable parameters.

To build momentum contrastive contrasts, we also need to encode code and queries with the momentum encoder $Encoder^*$:

$$\begin{aligned} \boldsymbol{q_i^*} &= Encoder^*(q_i), \\ \boldsymbol{c_i^*} &= Encoder^*(c_i), \end{aligned} \tag{8}$$

where $\boldsymbol{q_i^*}$ and $\boldsymbol{c_i^*}$ are embeddings of $q_i$ and $c_i$, which are encoded by the momentum encoder. After the operation of the current batch is completed, the $\boldsymbol{q_i^*}$ and $\boldsymbol{c_i^*}$ of the current batch will not be discarded, but stored in the randomly initialized queues $Q_q$ and $Q_c$.

### C. AST and DFG Embeddings

Abstract syntax tree [39] is a tree-structured representation of source code syntax. Each node on the tree represents a specific syntax structure in the source code. Data flow [11] is a graph that represents the dependency relation between variables in the source code. Each node on the graph represents where the value of each variable comes from. Both AST and DFG contain the structural information of source code, which can better help understand the semantics of source code. Given an executable code snippet, we can parse it into an abstract syntax tree, and further convert the abstract syntax tree into a data flow graph. We use the compile tool to analyze the lexical and syntax structure of code, and parse the source code into AST. After obtaining the AST of the code, the DFG can be built through the relation between variables.

To encode AST and DFG with the momentum encoder, we need to convert tree-structured and graph-structured data into sequence data. For the AST, we use preorder traversal to obtain the sequence data of AST. To obtain the sequence data of DFG, we directly traverse the DFG according to the order of variables in the code. Given a batch of $N$ pairs $(q_i, c_i)$, we can obtain a batch of $N$ AST-DFG pairs $(a_i, d_i)$ by parsing and transforming the source code. After encoding, we can obtain AST and DFG embeddings:

$$\begin{aligned} \boldsymbol{a_i^*} &= Encoder^*(a_i), \\ \boldsymbol{d_i^*} &= Encoder^*(d_i), \end{aligned} \tag{9}$$

where $\boldsymbol{a_i}$ and $\boldsymbol{d_i}$ are embeddings of $a_i$ and $d_i$. The specific encoding process is the same as query and code embeddings. All the momentum encoders $Encoder^*$ share the same parameters. After the operation of the current batch is completed, the $\boldsymbol{a_i^*}$ and $\boldsymbol{d_i^*}$ of the current batch will also not be discarded, but stored in the randomly initialized queue $Q_a$ and $Q_d$.

### D. Multi-Modal Momentum Contrastive learning

To incorporate multi-modal code information, we build multi-modal momentum contrasts for query, code, AST, and DFG. Specifically, we construct positive samples by combining code embeddings with the corresponding query, AST, and DFG embeddings encoded by the momentum encoder or combining query embeddings with the corresponding code, AST, and DFG embeddings encoded by the momentum encoder. We construct negative samples by combining code embeddings with all query embeddings in the NL queue, all AST embeddings in the AST queue, and all DFG embeddings in the DFG queue. We also construct negative samples by combining query embeddings with all code embeddings in the PL queue, all AST embeddings in the AST queue, and all DFG embeddings in the DFG queue.

As shown in Figure 3, we divide MoCoCS into two parts. One is to build contrasts for the query, and the other one is to build contrasts for the code. For each part, we construct positive and negative data pairs for multi-modal momentum contrastive learning. Thus, we can divide all data pairs into four categories: (a) Positive samples for the query, consist of $(\boldsymbol{q_i}, \boldsymbol{c_i^*})$, $(\boldsymbol{q_i}, \boldsymbol{a_i^*})$ and $(\boldsymbol{q_i}, \boldsymbol{d_i^*})$. (b) Negative samples for the query, consist of $(\boldsymbol{q_i}, \boldsymbol{c_j^*})$,$(\boldsymbol{q_i}, \boldsymbol{a_j^*})$, and $(\boldsymbol{q_i}, \boldsymbol{d_j^*})$. (c) Positive samples for the code, consist of $(\boldsymbol{c_i}, \boldsymbol{q_i^*})$, $(\boldsymbol{c_i}, \boldsymbol{a_i^*})$ and $(\boldsymbol{c_i}, \boldsymbol{d_i^*})$. (d) Negative samples for the code, consist of $(\boldsymbol{c_i}, \boldsymbol{q_j^*})$,$(\boldsymbol{c_i}, \boldsymbol{a_j^*})$, and $(\boldsymbol{c_i}, \boldsymbol{d_j^*})$. $\boldsymbol{q_i}$ and $\boldsymbol{c_i}$ are obtained by encoding the current batch query-code pairs with the gradient-update encoder. $\boldsymbol{q_i^*}$ and $\boldsymbol{c_i^*}$ are obtained by encoding the current batch of query-code pairs with the momentum-update encoder. $\boldsymbol{q_j^*}$, $\boldsymbol{c_j^*}$, $\boldsymbol{a_j^*}$ and $\boldsymbol{d_j^*}$ are obtained from the queues. The multi-modal momentum contrastive learning losses for $q_i$ and $c_i$ are defined as follows:

$$\begin{aligned} L(q_i) = &-log\frac{exp(\boldsymbol{q_i^T} \cdot \boldsymbol{c_i^*}/t)}{\sum_{j=0}^{M} exp(\boldsymbol{q_i^T} \cdot \boldsymbol{c_j^*}/t)} - \lambda \cdot log\frac{exp(\boldsymbol{q_i^T} \cdot \boldsymbol{a_i^*}/t)}{\sum_{j=0}^{M} exp(\boldsymbol{q_i^T} \cdot \boldsymbol{a_j^*}/t)} \\ &- \mu \cdot log\frac{exp(\boldsymbol{q_i^T} \cdot \boldsymbol{d_i^*}/t)}{\sum_{j=0}^{M} exp(\boldsymbol{q_i^T} \cdot \boldsymbol{d_j^*}/t)}, \end{aligned} \tag{10}$$

$$\begin{aligned} L(c_i) = &-log\frac{exp(\boldsymbol{c_i^T} \cdot \boldsymbol{q_i^*}/t)}{\sum_{j=0}^{M} exp(\boldsymbol{c_i^T} \cdot \boldsymbol{q_j^*}/t)} - \lambda \cdot log\frac{exp(\boldsymbol{c_i^T} \cdot \boldsymbol{a_i^*}/t)}{\sum_{j=0}^{M} exp(\boldsymbol{c_i^T} \cdot \boldsymbol{a_j^*}/t)} \\ &- \mu \cdot log\frac{exp(\boldsymbol{c_i^T} \cdot \boldsymbol{d_i^*}/t)}{\sum_{j=0}^{M} exp(\boldsymbol{c_i^T} \cdot \boldsymbol{d_j^*}/t)}, \end{aligned} \tag{11}$$

where $i \in [0, N)$, $j \in [0, M)$, N is the batch size, M is the queue size, and $t$ is the temperature coefficient. $\lambda$ and $\mu$ are the weight coefficients.

The final multi-modal momentum contrastive learning loss is calculated as:

$$L = \frac{1}{N} \sum_{i=1}^{N} L(q_i) + L(c_i). \tag{12}$$

**Algorithm 1:** Multi-Modal Momentum Contrast

**Data:** A batch of query sequence $q$, code sequence $c$, AST sequence $a$ and DFG sequence $d$. A queue of query embedding $\boldsymbol{Q_a}$, code embedding $\boldsymbol{Q_c}$, AST embedding $\boldsymbol{Q_a}$ and DFG embedding $\boldsymbol{Q_d}$.

Initialize the gradient-update encoder $E$ and the momentum-update encoder $E^*$.
**for** *mini-batch in DataLoader* **do**
  Get a batch of query embedding $\boldsymbol{q}$, code embedding $\boldsymbol{c}$ using $E$ to encode $q$, $c$.
  Get a batch of query embedding $\boldsymbol{q^*}$, code embedding $\boldsymbol{c^*}$, AST embedding $\boldsymbol{a^*}$, DFG embedding $\boldsymbol{d^*}$ using $E^*$ to encode $q$, $c$, $a$, $d$.
  Construct positive samples for query: $(\boldsymbol{q}, \boldsymbol{c^*})$, $(\boldsymbol{q}, \boldsymbol{a^*})$, $(\boldsymbol{q}, \boldsymbol{d^*})$.
  Construct positive samples for code: $(\boldsymbol{c}, \boldsymbol{q^*})$, $(\boldsymbol{c}, \boldsymbol{a^*})$, $(\boldsymbol{c}, \boldsymbol{d^*})$.
  Construct negative samples for query:$(\boldsymbol{q}, \boldsymbol{Q_c})$, $(\boldsymbol{q}, \boldsymbol{Q_a})$, $(\boldsymbol{q}, \boldsymbol{Q_d})$.
  Construct negative samples for code: $(\boldsymbol{c}, \boldsymbol{Q_q})$, $(\boldsymbol{c}, \boldsymbol{Q_a})$, $(\boldsymbol{c}, \boldsymbol{Q_d})$.
  Calculate our contrastive loss $L$.
  $\boldsymbol{Q_q}$, $\boldsymbol{Q_c}$, $\boldsymbol{Q_a}$, $\boldsymbol{Q_d}$ dequeue earliest batch and enqueue current batch $\boldsymbol{q^*}$, $\boldsymbol{c^*}$, $\boldsymbol{a^*}$, $\boldsymbol{d^*}$.
  Update $E$ by the gradient.
  Update $E^*$ by the momentum.
**end**

After the contrastive loss $L$ is calculated, the current batch of $q_i^*$, $c_i^*$, $a_i^*$, $d_i^*$ are stored in the corresponding queues, and the earliest batch of $q_i^*$, $c_i^*$, $a_i^*$, $d_i^*$ in the queues are discarded.

### E. Model Fine-tuning

In the fine-tuning stage, the gradient can be obtained by calculating our multi-modal contrastive loss $L$. The parameter $\theta$ of $Encoder$ is updated by the gradient. The parameter $\theta^*$ of $Encoder^*$ is momentum-updated by:

$$\theta^* = m\theta^* + (1-m)\theta, \tag{13}$$

where $m \in [0, 1)$ is the momentum coefficient. The momentum coefficient $m$ is set to a relatively large number to ensure that the momentum encoder is updated slowly. Here the momentum coefficient $m$ is set to $0.999$.

## IV. Evaluation

To evaluate our proposed method, we investigate the following research questions:

- **RQ1 (Effectiveness).** How effectiveness is our proposed MoCoCS?
- **RQ2 (Applicability).** Is our method applicable to different pre-trained models and different programming languages?
- **RQ3 (Ablation).** Does each module of our method improve the effectiveness of pre-trained models in the downstream code search task?
- **RQ4 (Parameter Sensitivity).** How do temperature coefficient, momentum coefficient, and queue size affect the results?
- **RQ5 (Memory Usage).** How does our method perform in terms of physical memory usage?

### A. Dataset

We choose a large-scale benchmark dataset CodeSearchNet [24] for our experiments. CodeSearchNet consists of six programming languages: Ruby, Javascript, Java, Go, PHP, and Python. Each language contains four parts: training set, validation set, test set, and codebase. The training set contains query-code pairs that are used to train the model for the code search task. The validation and test sets contain natural language queries that are used to retrieve matching code from the codebase. Different from the original setting of CodeSearchNet, we follow the settings of Guo et al. [11]. The codebase has been extended to include the code snippets in the validation and test sets. And the dataset is processed through the following operations. (a) Remove comments in the code. (b) Remove the instances whose code can not be parsed into an abstract syntax tree. (c) Remove the instances whose document tokens number is shorter than 3 or larger than 256. (d) Remove the instances whose document contains special tokens, such as "https:". (d) Remove the instances whose document is not written in English. The final statistics are shown in Table I.

TABLE I
STATISTICS OF CODESEARCHNET CORPUS

| Language | Training | Validation | Test | Candidate Code |
|---|---|---|---|---|
| Ruby | 24,927 | 1,400 | 1,261 | 4,360 |
| JavaScript | 58,025 | 3,885 | 3,291 | 13,981 |
| Java | 164,923 | 5,183 | 10,955 | 40,347 |
| Go | 167,288 | 7,325 | 8,122 | 28,120 |
| PHP | 241,241 | 12,982 | 14,014 | 52,660 |
| Python | 251,820 | 13,914 | 14,918 | 43,827 |

### B. Models for Comparison

We compare MoCoCS with several baselines:

- **NBow** is a bag-of-words model trained on both natural and programming language corpus.
- **CNN** is a 1D convolutional neural network [40] to embed input sequence of tokens.
- **BiRNN** is a bidirectional RNN model based on GRU cell [41] to summarize the input sequence.
- **SelfAtt** is a multi-head attention [37] model to compute representations of each token in the input sequence.

In addition, we also compare MoCoCS with several recent pre-trained models:

- **RoBERTa** proposed by Liu et al. [36] is a robustly optimized BERT pre-trained model. It is pre-trained with two objectives, Masked Language Model (MLM) and Next Sentence Prediction (NSP).
- **RoBERTa(code)** is structurally the same as RoBERTa. It is pre-trained on code corpus from CodeSearchNet.
- **CodeBERT** proposed by Feng et al. [10] is a bimodal pre-trained model for natural and programming language. It adopts two pre-training objectives, Mask Language Modeling (MLM) and Replaced Token Detection (RTD).
- **GraphCodeBERT** proposed by Guo et al. [11] is an extension of CodeBERT, integrating the data flow of code. It adds two new pre-training objectives, Edge Prediction (EP) and Node Alignment (NA).
- **SynCoBERT** proposed by Wang et al. [21] uses contrastive pre-training objectives and constructs positive samples with multiple views of code.
- **UniXcoder** proposed by Guo et al. [12] is a unified cross-modal pre-trained model for programming language. It utilizes mask attention matrices with prefix adapters to control the behavior of the model and leverages cross-modal contents like AST and code comment to enhance code representation.

We conduct our experiments on an NVIDIA Tesla V100s GPU with 32 GB memory. We fine-tune all the pre-trained models with a learning rate of 2e-5 and a batch size of 32. The number of epochs for fine-tuning is set to 10. The temperature coefficient $t$, momentum coefficient $m$, and queue size are set to 0.05, 0.999, and 8192 respectively. To boost the development of the community, we release MoCoCS at https://github.com/FDUDSDE/ICPC2023MoCoCS.

### C. Evaluation Metrics

To evaluate the effectiveness of our proposed MoCoCS, we adopt two widely used metrics for our experiments: Recall@k

(R@k) and Mean Reciprocal Rank (MRR) [42]. The Recall@k measures the percentage of queries that have more than one correct result in the top $k$ ranked results. The Recall@k is calculated as follows:

$$Recall@k = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\delta(FRank_q \leq k), \tag{14}$$

where $Q$ is a set of queries, $\delta(\cdot)$ is a function that returns 1 if the input is true and 0 otherwise. $FRank_q$ [43] is the rank of the first hit result in the result list of a query $q$.

The Mean Reciprocal Rank measures the average performance of code search models. The MRR is calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{FRank_q}, \tag{15}$$

where $|Q|$ is the size of the query set, $FRank_q$ is the rank of the first hit result in the result list of a query $q$.

## V. Results

In this section, the experimental results of the five research questions are presented. And each research question is answered by analyzing and summarizing the experimental results.

### A. RQ1: Effectiveness

Table II shows the effectiveness of the model fine-tuned with our method. We use a relatively advanced pre-trained model, UniXcoder, as the initial model for our method. $MoCoCS$ represents the model initialized by UniXcoder and fine-tuned with our MoCoCS. Overall, $MoCoCS$ achieves an average MRR score of 0.757 on the six programming languages of CodeSearchNet, which is higher than baseline models and recent pre-trained models. The negative sample is essential in contrastive learning, which can help pre-trained models learn well-distributed features. Our method expands the scale of negative samples by combining negative samples from multiple batches and multiple modalities. Therefore, $MoCoCS$ can achieve better performance by constructing large-scale multimodal negative samples.

TABLE II
MRR scores of $MoCoCS$ and baseline models

| Model | Ruby | JavaScript | Go | Python | Java | PHP | Average |
|---|---|---|---|---|---|---|---|
| NBow | 0.162 | 0.157 | 0.330 | 0.161 | 0.171 | 0.152 | 0.189 |
| CNN | 0.276 | 0.224 | 0.680 | 0.242 | 0.263 | 0.260 | 0.324 |
| BiRNN | 0.213 | 0.193 | 0.688 | 0.290 | 0.304 | 0.338 | 0.338 |
| SelfAtt | 0.275 | 0.287 | 0.723 | 0.398 | 0.404 | 0.426 | 0.419 |
| RoBERTa | 0.587 | 0.517 | 0.850 | 0.587 | 0.599 | 0.560 | 0.617 |
| RoBERTa(code) | 0.628 | 0.562 | 0.859 | 0.610 | 0.620 | 0.579 | 0.643 |
| CodeBERT | 0.679 | 0.620 | 0.882 | 0.672 | 0.676 | 0.628 | 0.693 |
| GraphCodeBERT | 0.703 | 0.644 | 0.897 | 0.692 | 0.691 | 0.649 | 0.713 |
| SynCoBERT | 0.722 | 0.677 | 0.913 | 0.724 | 0.723 | 0.678 | 0.740 |
| UniXcoder | 0.740 | 0.684 | 0.915 | 0.720 | 0.726 | 0.676 | 0.744 |
| $MoCoCS$ | **0.762** | **0.697** | **0.920** | **0.734** | **0.739** | **0.689** | **0.757** |

### B. RQ2: Applicability

Table III shows the applicability of our method to different pre-trained models and different programming languages. Similarly, $MoCoCS\_C$ represents the model initialized by CodeBERT and fine-tuned with our MoCoCS. $MoCoCS\_G$ represents the model initialized by GraphCodeBERT and fine-tuned with our MoCoCS. Overall, $MoCoCS\_C$ achieves an average MRR score of 0.725 on the six programming languages of CodeSearchNet, which is higher than the initial CodeBERT. $MoCoCS\_G$ achieves an average MRR score of 0.748 on the six programming languages of CodeSearchNet, which is higher than the initial GraphCodeBERT. The results show that each pre-trained model with our method improves the performance in the six programming languages. Under the dual-encoder framework, the strategy of increasing negative samples is applicable to different pre-trained models and different programming languages. Therefore, our method can be applied to different pre-trained models and different programming languages.

TABLE III
MRR scores of $MoCoCS$, $MoCoCS\_C$, $MoCoCS\_G$ and their corresponding initial models

| Model | Ruby | Javascript | Go | Python | Java | PHP | Average |
|---|---|---|---|---|---|---|---|
| UniXcoder | 0.740 | 0.684 | 0.915 | 0.720 | 0.726 | 0.676 | 0.744 |
| $MoCoCS$ | **0.762** | **0.697** | **0.920** | **0.734** | **0.739** | **0.689** | **0.757** |
| CodeBERT | 0.679 | 0.620 | 0.882 | 0.672 | 0.676 | 0.628 | 0.693 |
| $MoCoCS\_C$ | **0.698** | **0.655** | **0.907** | **0.707** | **0.715** | **0.666** | **0.725** |
| GraphCodeBERT | 0.703 | 0.644 | 0.897 | 0.692 | 0.691 | 0.649 | 0.713 |
| $MoCoCS\_G$ | **0.745** | **0.687** | **0.914** | **0.724** | **0.733** | **0.683** | **0.748** |

We also evaluate our method with R@k metrics. Our $MoCoCS$ achieves the R@1, R@5, R@10 scores of 0.667, 0.869, 0.907 in the Ruby language and 0.639, 0.848, 0.897 in the Python language, which performs better than the initial UniXcoder. $MoCoCS\_C$ achieves the R@1, R@5, R@10 scores of 0.594, 0.822, 0.877 in the Ruby language and 0.610, 0.825, 0.875 in the Python language, which performs better than the initial CodeBERT. $MoCoCS\_G$ achieves the R@1, R@5, R@10 scores of 0.653, 0.855, 0.899 in the Ruby language and 0.630, 0.841, 0.889 in the Python language, which performs better than the initial GraphCodeBERT. Under the R@k metrics, our method can still improve the effectiveness of pre-trained models.

TABLE IV
R@k scores of $MoCoCS$, $MoCoCS\_C$, $MoCoCS\_G$ and their initial models in the Ruby and Python languages

| Model | Ruby | | | Python | | |
|---|---|---|---|---|---|---|
| | R@1 | R@5 | R@10 | R@1 | R@5 | R@10 |
| UniXcoder | 0.642 | 0.845 | 0.899 | 0.613 | 0.835 | 0.888 |
| $MoCoCS$ | **0.667** | **0.869** | **0.907** | **0.639** | **0.848** | **0.897** |
| CodeBERT | 0.547 | 0.803 | 0.855 | 0.554 | 0.783 | 0.844 |
| $MoCoCS\_C$ | **0.594** | **0.822** | **0.877** | **0.610** | **0.825** | **0.875** |
| GraphCodeBERT | 0.606 | 0.830 | 0.876 | 0.584 | 0.809 | 0.864 |
| $MoCoCS\_G$ | **0.653** | **0.855** | **0.899** | **0.630** | **0.841** | **0.889** |

### C. RQ3: Ablation

Table V shows the results of ablation experiments. We conduct ablation experiments for our $MoCoCS$ with three settings: (a) Removing DFG. (b) Further removing AST. (c) Further removing momentum contrastive learning. Every time we remove a module, the MRR score drops to some extent. The results show that each module of our method improves the performance of pre-trained models. When we remove all three modules, our $MoCoCS$ degenerates into the initial pre-trained model UniXcoder. But the MRR scores of removing all three modules are mostly lower than the MRR scores of the UniXcoder in $RQ1$. This is because the two results are obtained under different batch sizes. Here we use batch size 32 to conduct our experiments, and the original MRR scores of UniXcoder are obtained under the setting of batch size 64. It shows that the batch size does affect the performance of pre-trained models. At the same time, it also demonstrates the effectiveness of our method, we use a smaller batch size to achieve better results.

TABLE V
MRR SCORES OF ABLATION EXPERIMENTS

| Model | Ruby | Javascript | Go | Python | Java | PHP |
|---|---|---|---|---|---|---|
| $MoCoCS$ | **0.7617** | **0.6968** | **0.9197** | **0.7335** | **0.7385** | **0.6887** |
| - DFG | 0.7616 | 0.6960 | 0.9192 | 0.7332 | 0.7382 | 0.6885 |
| - AST | 0.7571 | 0.6948 | 0.9190 | 0.7327 | 0.7379 | 0.6886 |
| - Momentum CL | 0.7420 | 0.6430 | 0.9114 | 0.7129 | 0.7137 | 0.6629 |

### D. RQ4: Parameter Sensitivity

To explore the impact of temperature coefficient $t$, momentum coefficient $m$, and queue size on the effectiveness of pre-trained models, we conduct parameter sensitivity experiments of $MoCoCS$ on the Ruby language. The temperature coefficient $t$ is used to adjust the ability of pre-trained models to distinguish negative samples of different difficulties. The lower the temperature $t$ is set, the further the hard negative samples are pushed in the embedding space. We consider that negative samples with higher similarity scores are harder negative samples. As shown in Figure 4, as the temperature $t$ decreases, the model's ability to distinguish hard negative samples is enhanced, and the MRR score is also improved. But when the temperature $t$ continues to decrease to a certain extent, the MRR score begins to drop. This is because hard negative samples can also be potentially positive samples. Excessively pushing the hard negative samples away may lead to a decline in results. We can also obtain a similar conclusion on the R@k metrics. Experiments on the temperature coefficient show that 0.05 is a better setting of $t$.

The momentum coefficient $m$ is used to adjust how fast the parameters of the momentum encoder are updated. The larger the momentum coefficient $m$ is set, the slower the parameters of the momentum encoder are updated. As shown in Figure 5, as the momentum coefficient $m$ increases, both the MRR and R@k scores are improving. If we use a small momentum coefficient, the parameters of the momentum encoder will mostly come from the gradient-updated encoder. The momentum encoder will be updated quickly in each batch. For each batch, the embeddings encoded by the momentum encoder are stored in the queues. The quickly updated encoder will lead to an inconsistency problem in the negative samples of different batches. Inconsistent negative samples may cause the model to learn to distinguish samples from different batches, rather than distinguishing samples from their own semantic features. In order to avoid the problem of inconsistent negative samples, the momentum coefficient $m$ needs to be set to a relatively large value. But if the value of the momentum coefficient is too large, the parameters of the momentum encoder will be hardly updated. Experiments on the momentum coefficient show that 0.999 is a large enough set of $m$.

The queue size refers to the number of embeddings that the queue can store. The larger the queue size, the more negative samples can be constructed. As shown in Figure 6, as the queue size increases, both the MRR and R@k scores are improving. This is because more negative samples can provide richer information and help the pre-trained model learn more essential features. Although the rising curve of the MRR and R@k scores is relatively flat by increasing the number of negative samples, the improvement brought by large-scale negative samples is still considerable. When the number of negative samples increases to a certain extent, the improvement in model effectiveness is not obvious. Experiments show that 8192 is a large enough set of the queue size.

### E. RQ5: Memory Usage

To prove that our method requires only little memory to increase the number of negative samples, we conduct experiments on the memory usage of our $MoCoCS$ compared with UniXcoder. As shown in Figure 7, when $MoCoCS$ and UniXcoder use the same batch size 32, $MoCoCS$ takes more memory for initializing the queue and the momentum encoder. As the number of negative samples increases, the memory usage of UniXcoder increases linearly. But the memory usage of $MoCoCS$ remains at a stable value. This is because the existing end-to-end training mechanism increases the number of negative samples by increasing the batch size. Our $MoCoCS$ increases the number of negative samples by increasing the queue size, which only requires little extra memory. The experimental results show that our method can construct large-scale negative samples with limited memory usage.

### F. Case Study

We also conduct case studies on the test set of the CodeSearchNet Ruby corpus. We first explore the impact of our MoCoCS on the retrieval results of UniXcoder. As shown in Figure 8, after fine-tuning with our method, our MoCoCS boosts the ranking of the target code snippets for 17% of the test samples and decreases the ranking of the target code snippets for 12% of the test samples. For the remaining 71% of the test samples, the ranking of the target code snippets keeps unchanged. Overall, our method can improve the effectiveness of pre-trained models in the code search task.

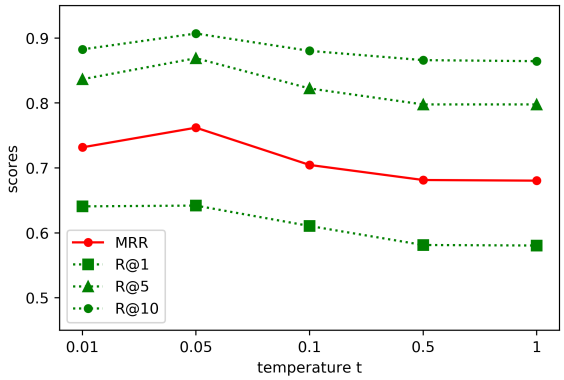


Fig. 4. MRR and R@k scores with different temperature t

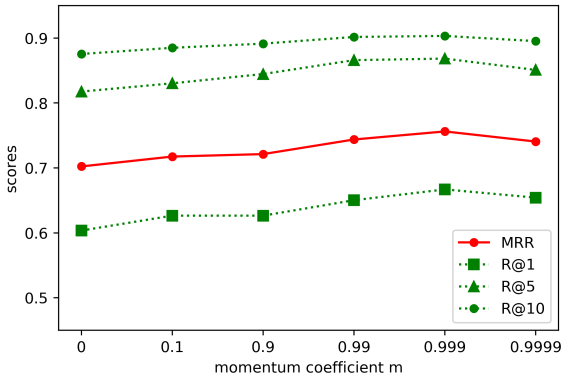


Fig. 5. MRR and R@k scores with different momentum coefficient m

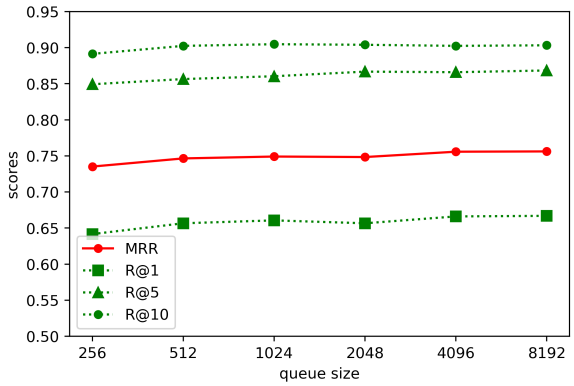


Fig. 6. MRR and R@k scores with different queue size

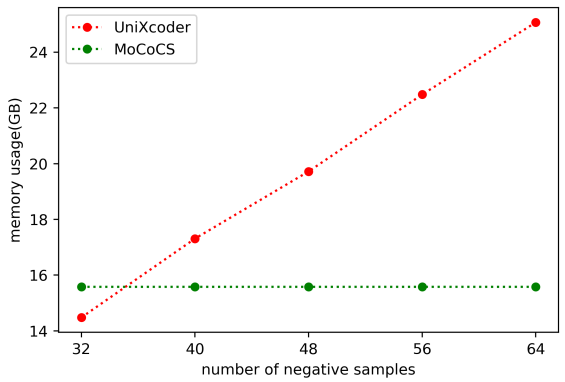


Fig. 7. Memory usage with different numbers of negative samples

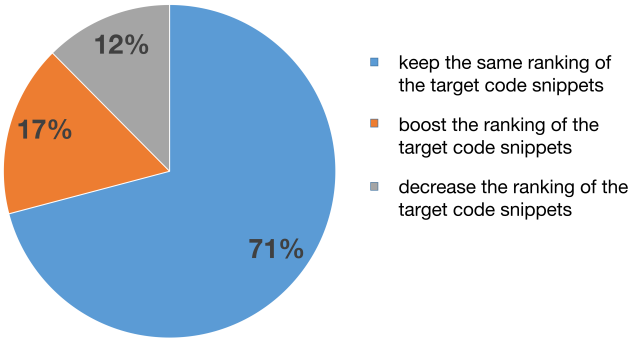


Fig. 8. The impact of our MoCoCS on the retrieval results of UniXcoder

It can be seen that our MoCoCS improves the ranking results for some cases and worsens the ranking results for some other cases. Through observation, we analyze that the possible influencing factor is the degree of correlation between natural language queries and code identifiers. If a matching query and code pair have a low correlation between queries and code identifiers, paying much attention to the code identifier may lead to incorrect ranking results. For example, function names often describe what the function does to some extent. As shown in Figure 9, UniXcoder prefers to retrieve results that function names are more relevant to the queries. And our MoCoCS pays more attention to the functionality of code snippets and achieves better performance than UniXcoder. But in the cases of Figure 10, the matching query and code pairs have a higher correlation between queries and code identifiers. Our MoCoCS does not pay enough attention to the correlation and performs worse than UniXcoder. We can also see the impact of multi-modal negative samples on the pre-trained models through these cases. For example in Query1 of Figure 10, the query is to "Delete the given file". And our MoCoCS returns the "remove_file" function, not the "delete" function. Though it is not the correct answer, it also shows that large-scale multi-modal negative samples can help the pre-trained models obtain better semantic code representations.

## VI. Threats to Validity

Our proposed MoCoCS may suffer from two threats to validity. The first one is the setting of batch size. When the batch size can be set to a large enough value, the effectiveness of our method may be diminished. Because the model already has enough negative samples. The second one is the setting of weight coefficients in the contrastive loss. In the test stage, we do not use the AST and DFG information. In the training stage, applying large weight coefficients to AST and DFG samples in contrastive loss will lead to worse results. Therefore, we need to set the weight coefficient in contrastive loss to a small value.

**Query1:** Create an incremental install path cache for the software step
**(correct) First retrieval of *MoCoCS:***
```
def incremental
  log.internal(log_key) { "Performing incremental cache" }
  create_cache_path
  remove_git_dirs
  git_cmd("add -A -f")
  begin
    git_cmd(%Q{commit -q -m "Backup of #{tag}"})
  rescue CommandFailed => e
    raise unless e.message.include?("nothing to commit")
  end
  git_cmd(%Q{tag -f "#{tag}"})
end
```
**(incorrect) First retrieval of UniXcoder*:***
```
def create_cache_path
  if File.directory?(cache_path)
    false
  else
    create_directory(File.dirname(cache_path))
    git_cmd("init -q")
    git_cmd("config --local user.name \"Omnibus Git Cache\"")
    git_cmd("config --local user.email \"omnibus@localhost\"")
    true
  end
end
```
**Query2:** Run commands in the context of a tab
**(correct) First retrieval of *MoCoCS:***
```
def tab(*args, &block)
  tabs = @_context[:tabs]
  key  = "tab#{tabs.keys.size}"
  return (tabs[key] = { :commands => args }) unless block_given?
  context = (tabs[key] = {:commands => []})
  options = args.extract_options!
  options[:name] = args.first unless args.empty?
  context[:options] = options
  run_context context, &block
  @_context = @_windows[@_windows.keys.last]
end
```
**(incorrect) First retrieval of UniXcoder*:***
```
def run(*commands)
  context = case
              when @_context.is_a?(Hash) && @_context[:tabs]
                @_context[:tabs]['default'][:commands]
              when @_context.is_a?(Hash)
                @_context[:commands]
              else
                @_context
              end
  context << commands.map { |c| c =~ /&$/ ? "(#{c})" : c }.join(" && ")
end
```

Fig. 9. Examples that our MoCoCS performs better than UniXcoder

**Query1:** Delete the given file or directory on the system. This method uses the equivalent of +rm -rf+, so you may pass in a specific file or a glob of files
**(incorrect) First retrieval of *MoCoCS:***
```
def remove_file(*paths)
  path = File.join(*paths)
  log.debug(log_key) { "Removing file `#{path}'" }
  FileUtils.rm_f(path)
  path
end
```
**(correct) First retrieval of UniXcoder*:***
```
def delete(path, options = {})
  build_commands << BuildCommand.new("delete `#{path}'") do
    Dir.chdir(software.project_dir) do
      FileSyncer.glob(path).each do |file|
        FileUtils.rm_rf(file, options)
      end
    end
  end
end
```
**Query2:** Pad message out with spaces
**(incorrect) First retrieval of *MoCoCS:***
```
def zero_pad(n, message)
  len = message.bytesize
  if len == n
    message
  elsif len > n
    raise LengthError, "String too long for zero-padding to #{n} bytes"
  else
    message + zeros(n - len)
  end
end
```
**(correct) First retrieval of UniXcoder*:***
```
def padout(message)
  message_length = self.class.display_columns(message)
  if @last_render_width > message_length
    remaining_width = @last_render_width - message_length
    message += ' ' * remaining_width
  end
  message
end
```

Fig. 10. Examples that UniXcoder performs better than our MoCoCS

## VII. Related Work

### A. NL-to-Code Search

The NL-to-code search aims to retrieve code snippets using natural language queries. Early research on NL-to-code search mainly leverages information retrieval techniques [44]–[48], lacking the semantic understanding between natural language queries and code snippets. To bridge the semantic gap between natural language queries and code snippets, supervised learning-based methods [3]–[6] encode the query and code into a joint embedding space and rank candidate code snippets by embedding similarity scores. Recently, self-supervised learning-based methods [10]–[12] train a BERT-based pre-trained model for code search by adopting the pre-training and fine-tuning paradigm. In addition, contrastive learning has also become a hot technique for enhancing pre-trained models for code search [17], [21], [49]–[52]. CoCoSoDa proposed by Shi et al. [51], also uses momentum contrastive learning to construct large-scale negative samples. The difference with our method is that CoCoSoDa uses soft data augmentation to construct positive and negative samples, while our MoCoCS considers the multi-modal code information of AST and DFG.

### B. Code-to-Code Search

The code-to-code search aims to retrieve similar code snippets using user input code snippets [53], which is also an important form of code search. For example, FaCoY proposed by Kim et al. [54], implements a code-to-code search engine to find code snippets that may be semantically similar to user input code. Senatus proposed by Silavong et al. [55], implements a code-to-code recommendation engine to recommend relevant, diverse, and concise code snippets that usefully extend the code currently being written by a developer in their IDE. Recently, cross-language code-to-code search techniques [56] have also become a research direction worthy of concern.

## VIII. Conclusion

In this paper, we proposed MoCoCS, a multi-modal momentum contrastive learning method for code search, to improve the representations of query and code by constructing large-scale multi-modal negative samples. To improve the effectiveness of pre-trained models, MoCoCS enables the construction of large-scale negative samples out of a mini-batch, which only requires little extra memory. To incorporate multi-modal code information, we use the momentum encoder to encode abstract syntax tree and data flow graph and build multi-modal momentum contrasts for query, code, abstract syntax tree, and data flow graph. Experiments on the six programming languages of CodeSearchNet demonstrate that our method can further improve the effectiveness of pre-trained models for code search.


## Acknowledgment

This work is supported by CNKLSTISS, the National Natural Science Foundation of China Projects No. U1936213, and the Major Key Project of PCL (PCL2021A06).

## REFERENCES


[1] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(FSE/ESEC), 2019, pp. 964-974.

[2] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on source code: a neural code search," In Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, 2018, pp. 31-41.

[3] X. Gu, H. Zhang, and S. Kim, "Deep code search," In 2018 IEEE/ACM 40th International Conference on Software Engineering(ICSE), 2018, pp. 933-944.

[4] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal Attention Network Learning for Semantic Source Code Retrieval," In 2019 34th IEEE/ACM International Conference on Automated Software Engineering(ASE), 2019, pp. 13-25.

[5] C. Zeng, Y. Yu, S. Li, X. Xia, Z. Wang, M. Geng, et al. "degraphcs: Embedding variable-based flow graph for neural code search," ACM Transactions on Software Engineering and Methodology, 2021.

[6] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," In Proceedings of the 28th International Conference on Program Comprehension(ICPC), 2020, pp. 196-207.

[7] Y. Cheng, and L. Kuang, "CSRS: Code Search with Relevance Matching and Semantic Matching," In Proceedings of the 30th International Conference on Program Comprehension(ICPC), 2022, pp. 533-542.

[8] W. Li, H. Qin, S. Yan, B. Shen, and Y. Chen, "Learning code-query interaction for enhancing code searches," In 2020 IEEE International Conference on Software Maintenance and Evolution, 2020, pp. 115-126.

[9] J. Gu, Z. Chen, and M. Monperrus, "Multimodal Representation for Neural Code Search," In 2021 IEEE International Conference on Software Maintenance and Evolution, 2021, pp. 483-494.

[10] Z. Feng, D. Guo., D. Tang, N. Duan, X. Feng, M. Gong, et al. "CodeBERT: A Pre-trained Model for Programming and Natural Languages," In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, 2020, pp. 1536-1547.

[11] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, L. I. U. Shujie, et al. "GraphCodeBERT: Pre-training Code Representations with Data Flow," In International Conference on Learning Representations, 2020.

[12] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "UniXcoder: Unified Cross-Modal Pre-training for Code Representation," In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022, pp. 7212-7225.

[13] A. Radford, and K. Narasimhan, "Improving Language Understanding by Generative Pre-Training," 2018.

[14] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI blog, 2019.

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

[16] H. Li, C. Miao, C. Leung, Y. Huang, Y. Huang, H. Zhang,et al. "Exploring Representation-Level Augmentation for Code Search," arXiv preprint arXiv:2210.12285, 2022.

[17] X. Li, Y. Gong, Y. Shen, X. Qiu, H. Zhang, B. Yao, et al. "CodeRetriever: Large-scale Contrastive Pre-training for Code Search," arXiv preprint arXiv:2201.10866, 2022.

[18] X. Wang, Y. Wang, Y. Wan, J. Wang, P. Zhou, L. Li, et al. "CODE-MVP: Learning to Represent Source Code from Multiple Views with Contrastive Pre-Training," arXiv preprint arXiv:2205.02029.

[19] A. V. D. Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," arXiv preprint arXiv:1807.03748, 2018.

[20] S. Liu, H. Fan, S. Qian, Y. Chen,W. Ding, and Z. Wang, "Hit: Hierarchical transformer with momentum contrast for video-text retrieval," In Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 11915-11925.

[21] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, et al. "SynCoBERT: Syntax-Guided Multi-Modal Contrastive Pre-Training for Code Representation," arXiv preprint arXiv:2108.04556, 2021.

[22] K. He, H. Fan, Y. Wu, S. Xie, and R.B. Girshick, "Momentum Contrast for Unsupervised Visual Representation Learning," 2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 9726-9735.

[23] X. Chen, H. Fan, R. Girshick, and K. He, "Improved baselines with momentum contrastive learning," arXiv preprint arXiv:2003.04297, 2020.

[24] H. Husain, H. H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," arXiv preprint arXiv:1909.09436, 2019.

[25] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," In International Conference on Machine Learning, 2020, pp. 5110-5121.

[26] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. E. Gonzalez, and I. Stoica, "Contrastive code representation learning," In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021.

[27] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, "Towards Learning (Dis)-Similarity of Source Code from Program Contrasts," In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, 2022, pp. 6300-6312.

[28] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, et al. "Supervised contrastive learning," Advances in Neural Information Processing Systems, 2020, pp. 18661-18673.

[29] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," In International conference on machine learning, 2020, pp. 1597-1607.

[30] T. Chen, S. Kornblith, K. Swersky, M. Norouzi, and G. E. Hinton, "Big self-supervised models are strong semi-supervised learners," Advances in neural information processing systems, 2020, pp. 22243-22255.

[31] T. Gao, X. Yao, and D Chen, "SimCSE: Simple Contrastive Learning of Sentence Embeddings," In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021.

[32] Y. Yan, R. Li, S. Wang, F. Zhang, W. Wu, and Xu, W. "ConSERT: A Contrastive Framework for Self-Supervised Sentence Representation Transfer," In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, 2021, pp. 5065-5075.

[33] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," In 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2016, pp. 1735-1742.

[34] R. D. Hjelm, A. Fedorov, S. Lavoie-Marchildon, K. Grewal, P. Bachman, A.Trischler, and Y. Bengio, "Learning deep representations by mutual information estimation and maximization," In International Conference on Learning Representations, 2018.

[35] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, "Unsupervised feature learning via non-parametric instance discrimination," In Proceedings of the IEEE conference on computer vision and pattern recognition, 2018, pp. 3733-3742.

[36] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, et al. "Roberta: A robustly optimized bert pretraining approach," arXiv preprint arXiv:1907.11692, 2019.

[37] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, et al. "Attention is all you need. Advances in neural information processing systems," 2017.

[38] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," arXiv preprint arXiv:1508.07909, 2015.

[39] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A Novel Neural Source Code Representation Based on Abstract Syntax Tree," 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 783-794.

[40] Y. Kim, "Convolutional Neural Networks for Sentence Classification," Conference on Empirical Methods in Natural Language Processing, 2014.

[41] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches," In Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, 2014, pp. 103-111.

[42] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 689-699.

[43] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis," In 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), 2016, pp. 357-367.

[44] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," In Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 111-120.

[45] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," IEEE Transactions on Software Engineering, 2011, pp. 1069-1087.

[46] F. Lv, H. Zhang, J. G. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model," In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2015, pp. 260-270.

[47] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for java using free-form queries," In International Conference on Fundamental Approaches to Software Engineering, 2009, pp. 385-400.

[48] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for JavaScript frameworks," In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 690-701.

[49] X. Li, D. Guo, Y. Gong, Y. Lin, Y. Shen, X. Qiu, et al. "Soft-Labeled Contrastive Pre-training for Function-level Code Representation," arXiv preprint arXiv:2210.09597, 2022.

[50] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," In Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, 2021, pp. 511-521.

[51] E. Shi, W. Gub, Y. Wang, L. Du, H. Zhang, S. Han, et al. "CoCoSoDa: Effective Contrastive Learning for Code Search," In Proceedings of the 45th International Conference on Software Engineering, 2023.

[52] Z. Shi, Y. Xiong, X. Zhang, Y. Zhang, S. Li, and Y. Zhu, "Cross-Modal Contrastive Learning for Code Search," In 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022, pp. 94-105.

[53] Y. Fujiwara, N. Yoshida, E. Choi, and K. Inoue, "Code-to-code search based on deep neural network and code mutation," In 2019 IEEE 13th International Workshop on Software Clones, 2019, pp. 1-7.

[54] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "FaCoY: a code-to-code search engine," In Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 946-957.

[55] F. Silavong, S. Moran, A. Georgiadis, R. Saphal, and R. Otter, "Senatus: a fast and accurate code-to-code recommendation engine," In Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 511-523.

[56] G. Mathew, and K. T. Stolee, "Cross-language code search using static and dynamic analyses," In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 205-217.