Self-Supervised Learning for Code Retrieval and Summarization through Semantic-Preserving Program Transformations

Nghi D. Q. Bui
Singapore Management University
dqnbui.2016@phdcs.smu.edu.sg

Yijun Yu
The Open University, UK
y.yu@open.ac.uk

Lingxiao Jiang
Singapore Management University
lxjiang@smu.edu.sg

Abstract

Code retrieval and summarization are useful tasks for developers, but it is also challenging to build indices or summaries of code that capture both syntactic and semantic essential information of the code. To build a decent model on source code, one needs to collect a large amount of data from code hosting platforms, such as Github, Bitbucket, etc., label them and train it from a scratch for each task individually. Such an approach has two limitations: (1) training a new model for every new task is time-consuming; and (2) tremendous human effort is required to label the data for individual downstream tasks. To address these limitations, we are proposing Corder, a self-supervised contrastive learning framework that trains code representation models on unlabeled data. The pre-trained model from Corder can be used in two ways: (1) it can produce vector representation of code and can be applied to code retrieval tasks that does not have labelled data; (2) it can be used in a fine-tuning process for tasks that might still require label data such as code summarization. The key innovation is that we train the source code model by asking it to recognize similar and dissimilar code snippets through a *contrastive learning paradigm*. We use a set of semantic-preserving transformation operators to generate code snippets that are syntactically diverse but semantically equivalent. The contrastive learning objective, at the same time, maximizes the agreement between different views of the same snippets and minimizes the agreement between transformed views of different snippets. Through extensive experiments, we have shown that our Corder pretext task substantially outperform the other baselines for code-to-code retrieval, text-to-code retrieval and code-to-text summarization tasks.

ACM Reference Format:
Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Learning for Code Retrieval and Summarization through Semantic-Preserving Program Transformations. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

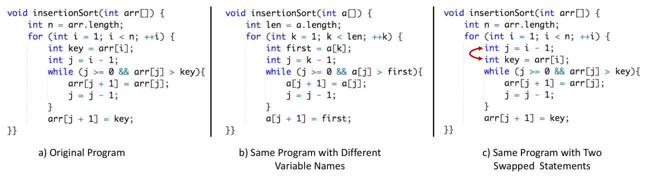
1 Introduction

Almost every part of human life today depends on reliable software, including health-care, logistics, education, etc., where software engineering plays a significant role. Understanding and reuse billions of lines of code that is available from the Web could speed up the software development process.

Towards this goal, deep learning models for code have been found useful in many software engineering tasks, such as predicting bugs [36, 53, 58], translating programs [9, 19], classifying program functionality [12, 43], searching code [18, 30, 47], generating comments from code [4, 22, 52], etc. They are useful because these tasks can be seen as `code retrieval` where code could be either the documents to be found or the query to search for.

For the neural networks to learn a good representation model of source code, the key idea is to exhibit patterns of semantically equivalent and non-equivalent code snippets in large quantity, which impose a major challenge in spending tremendous human effort to collect and label the code snippets. To overcome this challenge, one can depend on heuristic methods to label the code snippets automatically, such as using test cases to compare programs [40]. The downside of such a heuristic approach is extra costs associated with code execution, which may not always be trusted. Another way is to collect source code freely available on code hosting platforms such as Github, and extract the snippets that share similar comments [22], method name [4], or code documentation and treat such snippets as semantically equivalent [23]. The drawback to this strategy is that it can add a lot of noises because not all code snippets of identical comments, method names, or documentations are indeed semantic equivalents. For example, Kang et al. [28] shows that the pre-trained Code2vec [4] model does not perform well for other code modeling tasks when it was trained specifically for the method-name prediction task. Jiang et al. [27] performs further analysis to show the reason that methods with similar names are not necessarily semantically equivalent, which explains the poor transferred learning results of Kang et al. [28] on Code2vec since the model is forced to learn incorrect patterns of code.

To address these limitations, we developed Corder, a self-supervised constrative representation learning framework for source code that trains the network to identify semantically equivalent code snippets from a large set of transformed code-base. Essentially, the main goal of Corder is to invent a pretext task that enables the learning of the neural network to overcome the problems caused by imprecise heuristics to identify semantically equivalent programs. The pretext task that we are implementing in our case is the *instance discrimination* task to which some of the recent work relate [8]. The neural network is asked to discriminate against instances of code snippets



```
void insertionSort(int arr[]) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;
        while (j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
}}
```

a) Original Program

```
void insertionSort(int a[]) {
    int len = a.length;
    for (int k = 1; k < len; ++k) {
        int first = a[k];
        int j = k - 1;
        while (j >= 0 && a[j] > first){
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = first;
}}
```

b) Same Program with Different Variable Names

```
void insertionSort(int arr[]) {
    int n = arr.length;
    for (int i = 1; i < n; ++i) {
        int j = i - 1;
        int key = arr[i];
        while (j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
}}
```

c) Same Program with Two Swapped Statements

Figure 1: An Example of Semantically Equivalent Programs

that are semantically equivalent or dissimilar. In this way, the model is taught to learn which snippets are similar and which are not. In the end, the model is established with a knowledge of how different instances of the source code snippet look like, i.e. the semantically equivalent code snippet should be close to the vector space and the non-semantically equivalent code snippet should be further apart.

Our idea is to leverage program transformation techniques to transform a code snippet into different versions of itself. Although these transformed programs might be syntactically different from the original snippet, they are semantically equivalent. Figure 1 shows an example of the transformed programs: Figure 1a shows a code snippet, then Figure 1b shows a semantically equivalent snippet to Figure 1a, with the variable names changed. The snippet in Figure 1c is another transformed version of Figure 1a, with two independent statements that have been swapped. The goal is to teach the neural network that these snippets are similar and should be close in the vector space.

Corder uses the *contrastive learning* methods that have been used as a method in a self-supervised learning settings. The objective of contrastive learning is to simultaneously maximize the agreement between the differently transformed snippets of the same original snippet and minimize the agreement between the transformed snippets of other different snippets. Updating the parameters of a neural network using this contrastive learning objective causes the representations of semantically equivalent snippets to 'attract' each other, while representations of non-corresponding views to 'repel' each other.

Once the model has been trained on such a pretext task with the contrastive learning objective [1], it can be used in two ways. First, a neural network encoder (which is a part of the end-to-end learning process) has been trained and can be used to produce the representations of any source code. The vector representations of source code can be useful in many ways of code retrieval. Secondly, the pre-trained model can be fine-tuned with a small amount of labeled data to achieve good performance for other tasks, such as code summarization. In this work, we consider three tasks that can leverage such a pre-trained model, namely: code-to-code retrieval, text-to-code retrieval, and code summarization (which is code-to-text). We trained different Corder instances (e.g instances with different encoders) on large-scale Java datasets.

To summarize, our major contributions are as follows:

- First, we explore a novel perspective of learning source code model from unlabeled data. Unlike existing work that uses imprecise heuristics, we adapted program transformation techniques to generate precise semantically-equivalent code snippets. To the best of our knowledge, we are the first to use the program transformation technique for this use case.
- Second, to accomplish our goal, we developed Corder, a self-supervised constrastive learning framework to identify semantically-equivalent code snippets that are generated from the program transformation operators. We call this task as the Corder pretext task.
- Last, we conducted extensive experiments to demonstrate that our proposed pretext task is better than the other pretext tasks to learn the source code model. We demonstrate the usage of our pre-trained model in two use cases: (1) we use the pre-trained models to produce vector representations of code and apply such representations in the code-to-code retrieval task. The results show that any neural network encoder trained on the Corder pretext task outperforms the same encoders trained on other pretext tasks with a significant margin. Moreover, our technique outperforms the baseline that was designed specifically for code-to-code retrieval, such as FaCoy [30] significantly; (2) we use the pre-trained models in a fine-tuning process for supervised code modeling tasks, such as text-to-code retrieval and code summarization. The results show that our pre-trained models on the Corder pretext task perform better than training the code models from scratch and the other pretext task, by a large margin.

2 Related Work

Self-Supervised Learning has made tremendous strides in the field of visual learning [16, 17, 29, 34, 39, 57], and for quite some time in the field of natural language processing [13, 31, 35, 41]. Such techniques allow for neural network training without the need for human labels. Typically a self-supervised learning technique reformulates an unsupervised learning problem as one that is supervised by *generating virtual labels automatically from existing (unlabeled) data*. *Contrastive learning* has emerged as a new paradigm unifying many past approaches to self-supervised learning by formulating the supervised learning problem as the task to compare similar and dissimilar items, such as Siamese Neural Networks [7], triple loss [48], contrastive predictive coding [44]. Contrastive learning methods specifically minimize a distance between similar data (positives) representations and maximize the distance between dissimilar data (negatives).

Deep Learning Models of Code : There has been a huge interest in applying deep learning techniques for software engineering tasks such as program functionality classification [42, 56], bug localization [20, 45], function name prediction [15], code clone detection [56], program refactoring [22], program translation [9], and code synthesis [6]. Allamanis et al. [2] extend ASTs to graphs by adding a variety of code dependencies as edges among tree nodes, intended to represent code semantics, and apply Gated Graph Neural Networks (GGNN) [37] to learn the graphs; Code2vec [4], Code2seq [3], and ASTNN [56] are designed based on splitting ASTs into smaller ones, either as a bag of path-contexts or as flattened subtrees representing individual statements. They use various kinds of Recurrent Neural Network (RNN) to learn such code representations. Surveys on code embeddings [10, 25] present evidence to show that there is a strong need to alleviate the requirement of labeled data for code modeling and encourage the community to invest more effort into the methods on learning source code with unlabeled data. Unfortunately, there

[1]We call this as Corder pretext task.

Algorithm 1 Corder's learning algorithm

```
1: input: batch size $N$, encoder $f$, set of transformation operators $\mathcal{T}$.
2: for sampled minibatch $\{p_k\}_{k=1}^{N}$ do
3:     for all $k \in \{1, \ldots, N\}$ do
4:         draw two transformation operators $t \sim \mathcal{T}$, $t' \sim \mathcal{T}$
5:         # the first transformation
6:         $\tilde{p}_i = t(p_k)$
7:         $\boldsymbol{v}_i = f(\tilde{p}_i)$
8:         # the second transformation
9:         $\tilde{p}_j = t'(p_k)$
10:        $\boldsymbol{v}_j = f(\tilde{p}_j)$
11:    end for
12:    for all $i \in \{1, \ldots, 2N\}$ and $j \in \{1, \ldots, 2N\}$ do
13:        $s_{i,j} = z_i^\top z_j / (\|z_i\| \|z_j\|)$ # pairwise similarity
14:
15:    end for
16:    define $\ell(i,j)$ as $\ell(i,j) = -\log \frac{\exp(s_{i,j})}{\sum_{k=1}^{2N} 1_{k \neq i} \exp(s_{i,k})}$
17:
18:    $\mathcal{L} = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)]$
19:    update networks $f$ to minimize $\mathcal{L}$
20: end for
21: return encoder network $f(\cdot)$
```

is little effort that invests to design the source code model with unlabeled data: Yasunaga and Liang [54] presents a self-supervised learning paradigm for program repair, but it is designed specifically for program repair only. There are methods, such as [14, 24] that perform pretraining source code data on natural language model (BERT, RNN, LSTM), but they simply train the code tokens similar to the way pretrained language models on text do, so they miss a lot of information about syntactical and semantic features of code that could have been extracted from program analysis.

3 Approach

3.1 Approach Overview

Figure 2 presents an overview of our approach. Overall, this framework comprises the following three major components.

- **A program transformation module** that transforms a given code snippet p resulting in two transformed programs of the code snippets, denoted $\tilde{p}_i$ and $\tilde{p}_j$.
- **A neural network encoder** $f(\cdot)$ that can receive an intermediate representation of a code snippet (such as AST) and map it into a vector representation. In this case, it should map $\tilde{p}_i$ and $\tilde{p}_j$ into two code vectors v_i and v_j, respectively.
- **A contrastive loss function** is defined for the contrastive learning task. Given a set $\tilde{p}_k$ including a positive pair of examples p_i and $\tilde{p}_j$, the *contrastive prediction task* aims to identify $\tilde{p}_j$ in $\{\tilde{p}_k\}_{k \neq i}$ for a given p_i.

3.2 Approach Details

With the above components, here we describe the Corder training process in the following three steps. Also, a summarization of the proposed algorithm is depicted in Algorithm 1.

- A mini-batch of N samples is randomly selected from a large set of code snippets. Each code snippet p in N is applied with two different randomly selected transformation operators, resulting in 2N transformed code snippets.

$$\tilde{p}_i = t(p), \quad \tilde{p}_j = t'(p), \quad t, t' \sim \mathcal{T} \tag{1}$$

  where p is the original code snippet, $\tilde{p}_i$ and $\tilde{p}_j$ are transformed code snippets by applying two transformation operators t and t' into p, respectively. t and t' are randomly chosen from a set of available operators $\mathcal{T}$.
- Each of the transformed snippet $\tilde{p}_i$ and $\tilde{p}_j$ will be fed into the same encoder $f(\cdot)$ to get the embedding representations.

$$v_i = f(\tilde{p}_i), \quad v_j = f(\tilde{p}_j) \tag{2}$$

- We use the Noise Contrastive Estimate (NCE) loss function [8] to compute the loss. Let $\text{sim}(\boldsymbol{u}, \boldsymbol{v}) = \frac{\boldsymbol{u}^\top \boldsymbol{v}}{\|\boldsymbol{u}\| \|\boldsymbol{v}\|}$ denote the dot product between ℓ_2 normalized $\boldsymbol{u}$ and $\boldsymbol{v}$ (i.e. cosine similarity). Then the loss function for a pair of representations (v_i, v_j) is defined as

$$\ell(i,j) = -log \frac{exp(sim(v_i, v_j))}{\sum_{k=1}^{2N} \mathbf{1}_{k \neq i} \exp(\text{sim}(v_i, v_k))} \tag{3}$$

  where $\mathbf{1}_{k \neq a} \in \{0, 1\}$ is an indicator function evaluating to 1 iff $k \neq i$. Noted that for a given positive pair, the other $2(N-1)$ transformed code snippets are treated as negative samples. We calculate the loss for the same pair a second time as well where the positions of the samples are interchanged.
  The final loss is computed across all pairs in a mini-batch can be written as:

$$L = \frac{1}{2N} \sum_{k=1}^{N} [\ell(2k-1, 2k) + \ell(2k, 2k-1)] \tag{4}$$

3.2.1 Program Transformation Operators The key idea to enable the neural network encoder to learn a set of diverse code features without the need for labeled data is that we can generate multiple versions of a program without changing its semantics. To do so, we apply a set of semantic-preserving program transformation operators to generate such different variants. Although there are many methods for transforming the code [46], we mainly apply three transformations in this work, which are variable renaming, adding dead code (unused statements), and permute statement to reflect different ways to change the structure of the AST. The more sophisticated a change is, in principle, the better the neural network encoder can learn. We will evaluate how the change can effect the performance of the neural network in Section 6.1

- **Variable Renaming (VN)** is a refactoring method that renames a variable in code, where the new name of the variable is taken randomly from a set of variable vocabulary in the training set. Noted that each time this operator is applied to the same program, the variable names are renamed differently. This operator does not change the structure of the AST representation of the code, it only changes the textual information, which is a feature of a node in the AST.
- **Unused Statement (US)** is to insert dead code fragments, such as unused statement(s) to a randomly selected basic block in the code. We traverse the AST to identify the blocks and randomly select one block to insert predefined dead code fragments into it. This operator will add more nodes to the AST. It should be noted that to diversify the transformed program, we prepare a large set of unused statement(s). When the operator is applied, random

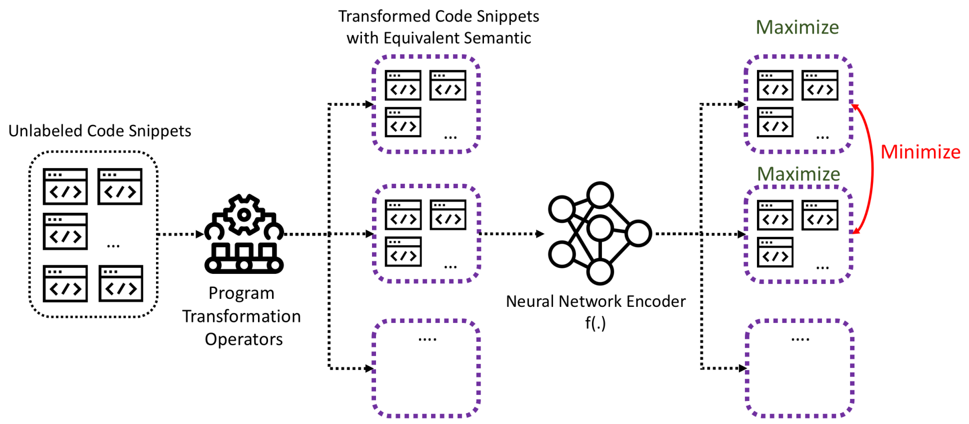

Unlabeled Code Snippets
Program Transformation Operators
Transformed Code Snippets with Equivalent Semantic
Neural Network Encoder f(.)
Maximize
Maximize
Minimize

Figure 2: Overview of Corder pretext task. Unlabeled code snippets from a large codebase go through a program transformation module. Snippets in the purple dashed box are transformed snippets from the same original snippet. The goal is to maximize the similarity of the snippets in the same group (purple dashed box) and minimize the similarity of snippets across different groups

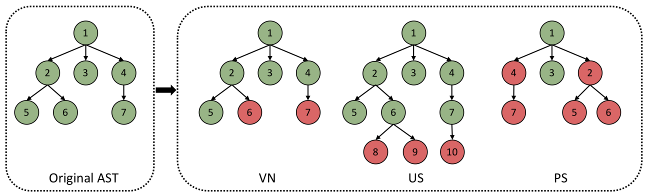


Figure 3: Example of how the AST structure is changed with different transformation operators

statements in the set is selected to added into the code block, i.e., a transformed snippet is different each time we apply the same operator.

- **Permute Statement (PS)** is to swap two statements that have no dependency on each other in a basic block in the code. We traverse the AST and analyze the data dependency to extract all of the possible pairs of swap-able statements. If a program only contains one such pair, it will generate the same output every time we apply the operator, otherwise, the output will be different.
- **Loop Exchange (LX)** replaces for loops with while loops or vice versa. We traverse the AST to identify the node the defines the for loop (or the while loop) then replace one with another with modifications on the initialization, the condition, and the afterthought.
- **Switch to If (SF)** replaces a switch statement in the method with its equivalent if statement. We traverse the AST to identify a switch statement, then extract the subtree of each case statement of the switch and assign it to a new if statement.

Each of the transformation operators above is designed to change the structure representation of the source code differently. For example, with Variable Renaming, we want the NN to understand that even the change in textual information does not affect the semantic meaning of the source code, inspired by a recent finding of Zhang et al. [55]. It is suggested that the source code model should be equipped with adversarial examples of token changes to make the model become more robust. With Unused Statement, we want the NN still to learn how to catch the similarity between two similar programs even though the number of nodes in the tree structure has increased. With Permute Statement, the operator does not add nodes into the AST but it will change the position of the subtrees in the AST, we want the NN to be able to detect the two similar trees even if the positions of the subtrees have changed. Figure 3 illustrates how the AST structure changes with the corresponding transformation operator.

3.2.2 *Neural Network Encoder for Source Code*

The neural network can also be called as an *encoder*, written as a function $f(\cdot)$. The encoder receives the intermediate representation (IR) of code and maps it into a code vector embedding $\vec{v}$ (usually a combination of various kinds of code elements), then $\vec{v}$ can be fed into the next layer(s) of a learning system and trained for an objective function of the specific task of the learning system. The choice of the encoder depends mostly on the task and we will rely on previous work to choose suitable encoders for a particular task, which will be presented in Section 5.

4 Use Cases

We present three use cases to make good use of the pre-trained Corder models, which are code-to-code retrieval, text-to-code retrieval, and code summarization.

.

4.1 Using the Pre-trained Encoders to Produce Code Vectors for Unsupervised Downstream Task

The first way to use pre-trained encoders from our Corder pretext task is to use such encoders to produce the vector representations of code. Then the representations can be applicable for a downstream task, such as code-to-code retrieval.

4.1.1 *Code-to-Code Retrieval*

Code-to-code search is useful for developers to find other code in a large codebase that is similar

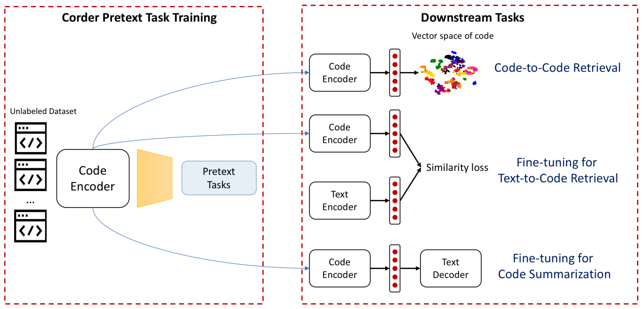


Figure 4: Process on how Corder pre-trained model can be applied in different downstream tasks

to a given code query. Most of the work that is designed for code-to-code retrieval, such as Facoy [30], Krugle [1] is based on the simple text mining approach or traditional code clone detection method. These techniques required tremendous effort of handcraft feature engineering to extract good features of code. In our case, we adapt pre-trained source code encoders from the Corder pretext task to map any code snippet into a vector representation, then we perform the retrieval task based on the vectors (see Figure 4, Code-to-Code Retrieval). Assume that we have a large codebase of snippets, we used the pre-trained encoders to map the whole codebase into representations. Then for a given code snippet as a query, we map such query into vector representation too. Then, one can find the top-k nearest neighbors of such query in the vector space, using cosine similarity as the distance metric, and finally can retrieve the list of candidate snippets. These snippets are supposed to be semantical equivalent to the query.

4.2 Fine-Tuning the Encoders for Supervised Learning Downstream Tasks

A paradigm to make good use of a large amount of unlabeled data is self-supervised pre-training followed by a supervised fine-tuning [8, 21], which reuses parts (or all) of a trained neural network on a certain task and continue to train it or simply using the embedding output for other tasks. Such fine-tuning processes usually have the benefits of (1) speeding up the training as one does not need to train the model from randomly initialized weights and (2) improving the generalizability of the downstream model even when there are only small datasets with labels.

As shown in Figure 4, the encoder serves as a pre-trained model, in which the weights resulted from the Corder pretext task are transferred to initialize the model of the downstream supervised learning tasks.

4.2.1 Text-to-Code Retrieval This task is to, given a natural language as the query, the objective is to find the most semantically related code snippets from a collection of codes [18, 23]. Note that this is different from the *code-to-code* retrieval problem, in which the query is a code snippet. The deep learning framework used in the literature for this task is to construct a bilateral neural network structure, which consists of two encoders, one is a *natural language encoder* (such as BERT, RNN, LSTM) to encode text into text embedding, the other is a *source code encoder* to encode an immediate source code representation into the code embedding [18, 23]. Then, from text embedding and code embedding, a mapping function is used to push the text and the code to be similar to the vector space, called a shared embedding between the code and the text. In the retrieval process, the text description is given, and we use its embedding to retrieve all the embeddings of the code snippets that are closest to the text embedding. In the fine-tuning process, the source code encoder that has been pre-trained on the Corder pretext task will be used to initialize for the source code encoder, the parameters of the text encoder will be initialized randomly.

4.2.2 Code Summarization The purpose of this task is to predict a concise text description of the functionality of the method given its source code [5]. Such descriptions typically appear as documentation of methods (e.g. "docstrings" in Python or "JavaDocs" in Java). This task can be modeled as a translation task where the aim is to translate a source code snippet into a sequence of text. As such, the encoder-decoder model, such as seq2seq [49] is usually used in the literature for this task. In our case, the encoder can be any code modeling technique, such as TBCNN [42], Code2vec [4], LSTM or Transformer. In the fine-tuning process, the source code encoder that has been pre-trained on the Corder pretext task will be used to initialize for the source code encoder.

5 Empirical Evaluation

5.1 Settings

5.1.1 Data Preparation As presented, we will perform the evaluation on three tasks, namely, code-to-code search, text-to-code search, and code summarization. We used the JavaSmall and JavaMed datasets that have been widely used recently for code modeling tasks [3, 4]. JavaSmall is a dataset of 11 relatively large Java projects from GitHub, which contains about 700k examples. JavaMed is a dataset of 1000 top-starred Java projects from GitHub which contains about 4M examples.

Then, we parse all the snippets into ASTs using SrcML [11]. We also perform the transformation on all of the ASTs to get the transformed ASTs based on the transformation operators described in Section 3.2.1, having the ASTs as well as the transformed ASTs. It should be noted that SrcML is a universal AST system, which means that it uses the same AST representations for multiple languages (Java, C#, C++, C). This enables the model training on each of the languages once and they can be used in other languages. Another thing to note is that these two datasets are not the ones used for evaluation purposes, they are only for the purpose of training the Corder pretext task on different encoders. We will describe the evaluation datasets used for each of the tasks separately in each of the subsections.

5.1.2 Encoders We choose a few well-known AST-based code modeling techniques as the encoder $f(\cdot)$, which are Code2vec [4], TBCNN [42], We also include two token-based techniques by treating source code simply as sequences of tokens and using a neural machine translation (NMT) baseline, i.e. a 2-layer Bi-LSTM, and the Transformer [51]. A common setting used among all these techniques is that they all utilize both node type and token information to initialize a node in ASTs.

We set both the dimensionality of type embeddings and text embeddings to 128. Note that we try our best to make the baselines as strong as possible by choosing the hyper-parameters above as the "optimal settings" according to their papers or code. The following presents specific settings for each of the baselines.

- Code2vec [4][2] and Code2seq [3][3]: since Code2seq is a follow-up work of Code2vec (only different in the decoder layer to predict the sequence), we follow the settings in Code2seq to set the size of each LSTM encoders for ASTs to 128 and the size of LSTM decoder to 320. We also set the number of paths sampled for each AST to 200 as suggested, since increasing this parameter does not improve the performance.
- TBCNN [42][4] uses a tree-based convolutional layer with three weight matrices serving as model parameters to accumulate children's information to the parent, each will have the shape of 128 x 128. We also set the number of convolutional steps to 8.
- Transformer [51]: We choose to set the number of layers to 5 and the attention dimension size to 128.
- 2-layer Bi-LSTM: We followed the strategy from Alon et al. [3]: we set the token embedding size to 128, the size of the hidden unit in the encoder to 128, and the default hyperparameters of OpenNMT [32].

5.1.3 Research Questions

We want to answer 2 research questions through the evaluations: (1) Is the code vectors provided by the pre-trained model (with any encoders) useful in the space searching task (code-to-code search)? and (2) Can the pre-trained models be used for a fine-tuning process to improve the performance of the models without training the models from scratch?

5.2 Using Pre-trained Encoders to Produce Code Representations for Code-to-Code Retrieval

5.2.1 Datasets, Metrics, and Baselines

Given a code snippet as the input, the task aims to find the most semantically related code from a collection of candidate codes. The datasets we used to evaluate for this task are:

- OJ dataset [42] contains 52000 C programs with 104 classes, which results in 500 programs per class. Since the dataset is for C++, we translate the whole dataset with the C++ to Java Converter [5] to make the language of the evaluation dataset aligned with the pretrained models for Java (see Section 5.1). Then we use the data that has been translated to Java for evaluation.
- BigCloneBench (BCB) dataset [50] contains 25,000 Java projects, cover 10 functionalities and including 6,000,000 true clone pairs and 260,000 false clone pairs. This dataset has been widely used for code clone detection task.

The OJ and BigCloneBench datasets have been widely used for the code clone detection task. The code clone detection task is to detect semantically duplicated (or similar) code snippets in a large codebase. Thus these datasets are also suitable for the code-to-code retrieval task, with the aims to find the most semantically related codes given the code snippet as the query.

[2]https://github.com/tech-srl/code2vec
[3]https://github.com/tech-srl/code2seq
[4]https://github.com/crestonbunch/tbcnn/
[5]https://www.tangiblesoftwaresolutions.com/product_details/cplusplus_to_java_converter_details.html

We randomly select 50 programs per class as the query, so that the total number of queries is 5200 for 104 classes. For each of the queries, we want to retrieve all of the semantically similar code snippets, which are the programs in the same class of the query. With OJ, each query can have multiple relevant results, so that we use Mean Average Precision (MAP) as the metric to evaluate for the code-to-code search on the OJ dataset. Mean average precision for a set of queries is the mean of the average precision scores for each query, which can be calculated as

$$MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q} \tag{5}$$

where Q is the number of queries in the set and $AveP(q)$ is the average precision for a given query q.

For the BCB dataset, since the size of the dataset is large, we reduce the size by randomly select 50,000 samples clone pairs and 50,000 samples none clone pairs and evaluate within these pairs. Then within the clone pairs, we again randomly select 5000 pairs and pick one code snippet of a pair as the query. Let's denote a clone pair as $p = (c_1, c_2)$, we pick c_1 as the query. For each of the query c_1, we want to retrieve the c_2, which is the snippet that is semantically identical to the query. With BCB, the assumption is that each query has only one relevant result so that we use Mean Reciprocal Rank (MRR) as the metric to evaluate for the task. Mean Reciprocal Rank is the average of the reciprocal ranks of results of a set of queries Q. The reciprocal rank of a query is the inverse of the rank of the first hit result. The higher the MRR value, the better the code search performance. MRR can be calculated as follows:

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} \frac{1}{rank_i} \tag{6}$$

Noted that for both datasets, we limited the number of return results to 10.

We use these baselines for the code-to-code retrieval task:

- Word2vec: the representation of the code snippet can be computed by simply calculate the average of the representations of all of the token in the snippet
- Doc2vec: we use Gensim [6] to train the Doc2vec model on the JavaMed dataset and use the method provided by Gensim to infer the representation for a code snippet
- ElasticSearch: we treat the code snippet as a sequence and use the text tokenizer provided by ElasticSearch to index the code token and use ElasticSearch as a fuzzy text search baseline.
- Facoy [30] is a search engine that is designed specifically for code-to-code search.

Besides the baselines above, we also want to see if our Corder pretext task performs better than the other pretext task for the same encoder. Among the encoders, the Transformer [51] can be pre-trained with other pretext tasks, such as the masked language modeling, where a model uses the context words surrounding a [MASK] token to try to predict what the [MASK] word should be. Code2vec [4] is also applicable for another pretext task, which is the method name prediction (MNP). The path encoder in Code2vec can encode the method body of a code snippet, then use the representation of the

[6]https://github.com/RaRe-Technologies/gensim

Table 1: Results of code-to-code search. For BigCloneBench (BCB), the metric is MAP. For OJ, the metric is MRR

Model	Pre-training	Performance	
		BCB(MRR)	OJ(MAP)
ElasticSearch	-	0.131	0.235
Word2vec	-	0.255	0.234
Doc2vec	-	0.318	0.415
FaCoy	-	0.587	0.585
Code2vec	MNP-JavaMed	0.453	0.517
	Corder-JavaMed	0.633	0.698
TBCNN	Corder-JavaMed	0.832	0.931
Bi-LSTM	Corder-JavaMed	0.612	0.681
Transformer	Masked LM-JavaMed	0.634	0.719
	Corder-JavaMed	0.825	0.841

method body to predict the method name. With this, the Code2vec model can be pre-trained with MNP as a pretext task. The path encoder of the Code2vec for the method name prediction task can be reused to produce representation for any code snippet. For such reasons, we include 2 additional baselines, which are a pre-trained Transformer on the masked language model on the JavaMed dataset, and a pre-trained Code2vec on MNP on the JavaMed dataset.

5.2.2 Results Table 1 shows the results of the code-to-code retrieval task. The column "Pre-training" with different options, such as "Corder-JavaMed", "MNP-JavaMed", means that an encoder is used with a different pretext task. As one can see, ElasticSearch, an information retrieval approach, performs worst among the baselines. Word2vec and Doc2vec perform better but the results are still not so good. Code2vec and Bi-LSTM, when pre-training with the Corder process on the JavaMed, can perform better than FaCoy, a method designed specifically for code-to-code retrieval. Code2vec, when pre-training with the method name prediction (MNP) pretext task, performs much worse than the pre-training with the Corder pretext task. Transformer, when pre-training with the masked language model (Masked-LM) pretext task, performs much worse than the pre-training with the Corder pretext task. This shows that our proposed pretext task performs better than the other pretext tasks to train the representation of the source code.

5.3 Fine-tuning Pre-trained Encoders for Text-to-code Retrieval

5.3.1 Datasets, Metrics, and Baselines Given a natural language as input, the task aims to find the most semantically related code from a collection of candidate codes. We use the dataset released by DeepCS [18], which consists of approximately 16 million preprocessed Java methods and their corresponding docstrings.

For the metrics, we use Precision at k (Precision@k) and Mean Reciprocal Rank to evaluate this task. Precision@k measures the percentage of relevant results in the top k returned results for each query. In our evaluations, it is calculated as follows:

$$Precision@k = \frac{\#relevant\ results\ in\ the\ top\ k\ results}{k} \quad (7)$$

Precision@k is important because developers often inspect multiple results of different usages to learn from. A better code search engine should allow developers to inspect less noisy results. The

Table 2: Results of text-to-code search

Model	Pre-training	P@1	P@5	P@10	MRR
NBow	-	0.394	0.581	0.603	0.384
Code2vec	-	0.406	0.529	0.564	0.395
	MNP-JavaSmall	0.415	0.538	0.572	0.409
	MNP-JavaMed	0.435	0.546	0.583	0.420
	Corder-JavaSmall	0.512	0.578	0.610	0.446
	Corder-JavaMed	**0.549**	**0.608**	**0.625**	**0.50**
TBCNN	-	0.506	0.581	0.632	0.551
	Corder-JavaSmall	0.541	0.620	0.658	0.658
	Corder-JavaMed	**0.610**	**0.686**	**0.710**	**0.682**
Bi-LSTM	-	0.469	0.540	0.702	0.630
	Corder-JavaSmall	0.532	0.581	0.723	0.619
	Corder-JavaMed	**0.567**	**0.639**	**0.768**	**0.661**
Transformer	-	0.514	0.653	0.793	0.651
	Masked LM-JavaSmall	0.539	0.698	0.845	0.687
	Masked LM-JavaMed	0.569	0.601	0.845	0.687
	Corder-JavaSmall	0.620	0.698	0.845	0.687
	Corder-JavaMed	**0.642**	**0.756**	**0.881**	**0.728**

higher the metric values, the better the code search performance. We evaluate and Precision@k when the value of k is 1, 5, and 10. These values reflect the typical sizes of results that users would inspect.

We choose to use the three methods presented in CodeSearchNet [23] for the text-to-code retrieval models, which are: neural bag-of-words, 2-layer BiLSTM, and Transformer. We also include Tree-based CNN (TBCNN) [42] and Code2vec [4] which are AST-based encoders that receive the AST representation of the code snippets as the input. We perform evaluations under 2 settings: (1) train from scratch and (2) fine-tune with a pre-trained model. In the second setting, each of the encoders will be pre-trained through the Corder pretext task, then the pre-trained encoder will be used for the fine-tuning process. We include the pre-trained Code2vec model from the method name prediction (MNP) task to demonstrate that our Corder pretext task is better for a fine-tuning process than the MNP task.

5.3.2 Results Table 2 shows the performance of text-to-code search task. The column "Pre-training" with different options, such as "-", "MNP-JavaSmall", "MNP-JavaMed", "Corder-JavaSmall", and "Corder-JavaMed", means that an encoder is used with different settings. "-" means that there is no pretext task applied for the model. "JavaSmall" means that the encoder is pre-trained with the pretext task on the JavaSmall dataset, same for "JavaMed". There are 3 observations : (1) Corder pre-training task on any of the model improves the performance significantly; (2) pre-training on a larger dataset improves the results with a higher margin than pre-training on a smaller dataset; and (3) Corder pretext task for Code2vec performs better than the MNP task to fine-tune the model for text-to-code retrieval.

5.4 Fine-tuning Pre-trained Encoders for Code Summarization

5.4.1 Dataset, Metric, and Baselines For this task, we consider predicting a full natural language sentence given a short code snippet. We also use the Java dataset provided by DeepCS [18], which consists of approximately 16 million preprocessed Java methods and

Table 3: Results of code summarization

Model	Pre-training	BLEU
MOSES	-	11.57
IR	-	14.56
Code2seq	-	24.56
	JavaSmall	25.79
	JavaMed	**27.52**
Bi-LSTM	-	19.67
	JavaSmall	21.56
	JavaMed	**22.86**
Transformer	-	22.45
	JavaSmall	23.78
	JavaMed	**25.98**

Table 4: Results on Analysis on the Impact of Different Transformation Operators on the text-to-code retrieval (TTC) and code summarization (CS)

Models	Ops	Tasks	
		FT-TTC (MRR)	FT-CS (BLEU)
TBCNN	-	0.551	-
	VR	0.562	-
	US	0.603	-
	PS	0.591	-
	All	0.682	-
Bi-LSTM	-	0.587	19.67
	VR	0.591	19.89
	US	0.652	20.90
	PS	0.610	20.67
	All	0.661	22.86
Transformer	-	0.651	22.45
	VR	0.668	22.96
	US	0.692	24.25
	PS	0.631	23.57
	All	0.728	25.98

their corresponding docstrings. The target sequence length in this task is about 12.3 on average. Since this dataset consists of parallel corpus of code snippets and docstrings, it is suitable for either the text-to-code retrieval task or the code summarization task.

To measure the prediction performance, we follow [4] to use the BLEU score as the metric. For the baselines, we present results compared to 2-layer bidirectional LSTMs, Transformer and Code2seq [3], a state-of-the-art model for code summarization task. We provide a fair comparison by splitting tokens into subtokens and replacing UNK during inference. We also include numbers from the baselines used by Iyer et al. [26], such as MOSES [33] and an IR-based approach that use Levenshtein distance to retrieve the description.

5.4.2 Results Table 3 shows the performance of Corder pretraining on the code summarization task. As seen, pretraining the model can improve the BLEU score with a significant margin for any of the encoder.

6 Analysis and Ablation Study

In this section, we perform some analysis and ablation studies to measure how different design choices can affect the performance of Corder.

6.1 Impact of Different Transformation Operators

We perform an ablation study to measure how each transformation operator affects the performance of particular code learning tasks. This means that in our Corder training algorithm, when drawing the transformation operators to transform the code snippet, the set of available operators $\mathcal{T}$ only contain one single operator. We choose Variable Renaming, Unused Statement and Permute Statement as the operator to evaluate in this analysis We train Corder with three encoders: TBCNN, Bi-LSTM, and Transform with similar settings in the Evaluation Section, but we only use one operator at a time. Then we perform the fine-tuning process on the text-to-code retrieval and code summarization task, also with similar to the Evaluation Section. Table 4 shows that the Unused Statement operator consistently among the operator that perform the best for most of the tasks. An explanation for this is that since the Unused Statement is the operator that modify the code structure extensively, which make the neural network can learn the features better. With all of the operators applied, the performance of the fine-tuning process is the best for both tasks, this means that the changes we made on the source code have a big impact on what the neural network learned.

6.2 Embedding Visualization for Clusters of Code Snippets

We visualize the code vectors to help understand and explain why the vectors produced by Corder pre-training are better than the vectors produced by other We choose Code2vec as the encoder for this analysis since Code2vec has been adapted in two different pretext tasks: (1) Corder pretext task (Corder-Code2vec); and (2) method name prediction task [4] (Code2vec). The goal is to show that our Corder pretext task performs better than the method name prediction as a pretext task to train the source code model. We use the code snippets in OJ dataset [42] that has been used for the code-to-code retrieval task. We choose the embeddings of the first 6 classes of the OJ dataset then we use T-SNE [38] to reduce the dimensionality of the vectors into two-dimensional space and visualize. As shown in Figure 5, the vectors produced by Corder-Code2vec group similar code snippets into the same cluster with much clearer boundaries. This means that our instance discrimination task is a better pretext task than the method name prediction task in Alon et al. [4] for the same Code2vec encoder.

7 Conclusion

We have proposed Corder, a self-supervised learning approach that can leverage large scale unlabeled data of source code. Corder works by training the network over a contrastive learning objective to compare similar and dissimilar code snippets that are generated from a set of program transformation operators. The snippets produced by such operators are syntactically diverse but semantically equivalent.

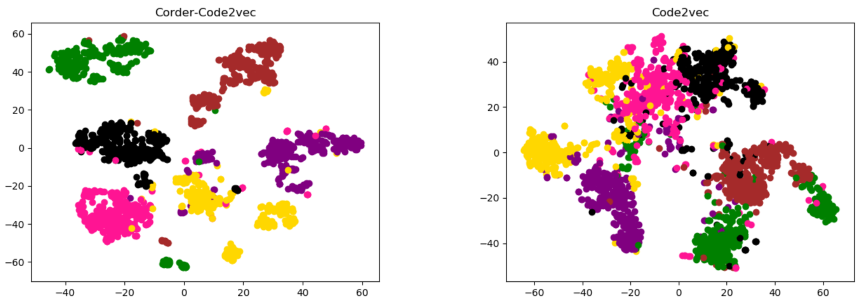


Figure 5: Visualization of the vector representations of the code snippets from 6 classes in the OJ Dataset produced by Corder-Code2vec and Code2vec

The goal of the contrastive learning methods is to minimize a distance between the representations of similar snippets (positives) and maximize the distance between dissimilar snippets (negatives). We adapted Corder into 3 tasks: code-to-code retrieval, fine-tuning for text-to-code retrieval, fine-tuning for code summarization and find that Corder pre-training significantly outperform those models not using contrastive learning, and the other baselines on these tasks.

References

[1] [n.d.]. Krugle Code Search, howpublished = https://krugle.com/, note = Accessed: 2020-09-30.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR*.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *ICLR*.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. In *POPL*. 40:1–40:29.

[5] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing, IJCNLP 2017, Taipei, Taiwan, November 27 - December 1, 2017, Volume 2: Short Papers*, Greg Kondrak and Taro Watanabe (Eds.). Asian Federation of Natural Language Processing, 314–319. https://www.aclweb.org/anthology/I17-2053/

[6] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *7th ICLR*.

[7] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1994. Signature verification using a" siamese" time delay neural network. In *Advances in neural information processing systems*. 737–744.

[8] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A simple framework for contrastive learning of visual representations. *arXiv preprint arXiv:2002.05709* (2020).

[9] Xinyun Chen, Chang Liu, and Dawn Song. 2018. Tree-to-tree neural networks for program translation. In *NeurIPS*. 2547–2557.

[10] Zimin Chen and Martin Monperrus. 2019. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061* (2019).

[11] Michael L Collard, Michael John Decker, and Jonathan I Maletic. 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *ICSM*. 516–519.

[12] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-scale malware classification using random projections and neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing*. 3422–3426.

[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[15] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Structured Neural Summarization. In *7th ICLR*.

[16] Basura Fernando, Hakan Bilen, Efstratios Gavves, and Stephen Gould. 2017. Self-supervised video representation learning with odd-one-out networks. 3636–3645.

[17] Spyros Gidaris, Praveer Singh, and Nikos Komodakis. 2018. Unsupervised representation learning by predicting image rotations. *arXiv preprint arXiv:1803.07728* (2018).

[18] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *40th ICSE*. 933–944.

[19] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2017. DeepAM: Migrate APIs with Multi-modal Sequence to Sequence Learning. In *IJCAI* (Melbourne, Australia). 3675–3681.

[20] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Neural Attribution for Semantic Bug-Localization in Student Programs. In *NeurIPS*. 11861–11871.

[21] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural computation* 18, 7 (2006), 1527–1554.

[22] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. 200–210.

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

[24] Yasir Hussain, Zhiqiu Huang, Yu Zhou, and Senzhang Wang. 2020. Deep transfer learning for source code modeling. *International Journal of Software Engineering and Knowledge Engineering* 30, 05 (2020), 649–668.

[25] Bill Ingram. 2018. *A Comparative Study of Various Code Embeddings in Software Semantic Matching*. https://github.com/waingram/code-embeddings.

[26] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[27] Lin Jiang, Hui Liu, and He Jiang. 2019. Machine learning based recommendation of method names: how far are we. In *34th ASE*. 602–614.

[28] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. 2019. Assessing the generalizability of code2vec token embeddings. In *34th ASE*. 1–12.

[29] Dahun Kim, Donghyeon Cho, and In So Kweon. 2019. Self-supervised video representation learning with space-time cubic puzzles. In *AAAI*, Vol. 33. 8545–8552.

[30] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. 946–957.

[31] Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Skip-thought vectors. In *NeurIPS*. 3294–3302.

[32] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. 2017. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810* (2017).

[33] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*. Association for Computational Linguistics, 177–180.

[34] Bruno Korbar, Du Tran, and Lorenzo Torresani. 2018. Cooperative learning of audio and video models from self-supervised synchronization. In *NeurIPS*. 7763–7774.

[35] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. 1188–1196.

[36] Jian Li, Pinjia He, Jieming Zhu, and Michael R Lyu. 2017. Software defect prediction via convolutional neural network. In *IEEE QRS*. 318–328.

[37] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated Graph Sequence Neural Networks. In *ICLR*.

[38] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, Nov (2008), 2579–2605.

[39] Aravindh Mahendran, James Thewlis, and Andrea Vedaldi. 2018. Cross pixel optical-flow similarity for self-supervised learning. In *Asian Conference on Computer Vision*. 99–116.

[40] Henry Massalin. 1987. Superoptimizer - A Look at the Smallest Program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II), Palo Alto, California, USA, October 5-8, 1987*, Randy H. Katz and Martin Freeman (Eds.). ACM Press, 122–126. https://dl.acm.org/citation.cfm?id=36194

[41] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *NeurIPS*. 3111–3119.

[42] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *AAAI*.

[43] R. Nix and J. Zhang. 2017. Classification of Android apps and malware using deep neural networks. In *International Joint Conference on Neural Networks*. 1871–1878.

[44] Aaron van den Oord, Yazhe Li, and Oriol Vinyals. 2018. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748* (2018).

[45] Michael Pradel and Koushik Sen. 2018. DeepBugs: A learning approach to name-based bug detection. *ACM on Programming Languages* 2, OOPSLA (2018), 147.

[46] Md Rabin, Rafiqul Islam, and Mohammad Amin Alipour. 2020. Evaluation of Generalizability of Neural Program Analyzers under Semantic-Preserving Transformations. *arXiv preprint arXiv:2004.07313* (2020).

[47] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on Source Code: A Neural Code Search. In *2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Philadelphia, PA, USA). 31–41. https://doi.org/10.1145/3211346.3211353

[48] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 815–823.

[49] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[50] Jeffrey Svajlenko and Chanchal K Roy. 2015. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 131–140.

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[52] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *33rd ASE* (Montpellier, France). New York, NY, USA, 397–407. https://doi.org/10.1145/3238147.3238206

[53] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction. In *IEEE QRS*. 17–26.

[54] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. *arXiv preprint arXiv:2005.10636* (2020).

[55] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *34th AAAI*.

[56] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *41st ICSE*. 783–794.

[57] Richard Zhang, Phillip Isola, and Alexei A Efros. 2016. Colorful image colorization. In *European conference on computer vision*. 649–666.

[58] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *NeurIPS*. 10197–10207.