Contrastive Learning of Natural Language and Code Representations for Semantic Code Search

Anonymous ACL submission

Abstract

Retrieving semantically relevant code functions given a natural language (NL) or programming language (PL) query is a task of great practical value towards building productivity enhancing tools for software developers. Recent approaches to solve this task involve leveraging transformer based masked language models that are pre-trained on NL and PL and fine-tuned for code search using a contrastive learning objective. However, these approaches suffer from uninformative in-batch negative samples.

We propose DyHardCode: a contrastive learning framework that leverages hard negative examples, which are mined globally from the entire training corpus to improve the quality of code and natural language representations. We experiment with different hard negative mining strategies, and provide explanations to the effectiveness of our method from the perspectives of optimization and adversarial learning. We show that DyHardCode leads to improvements in multiple code search tasks. Our approach achieves an average (across 6 programming languages) mean reciprocal ranking (MRR) score of 0.750 as opposed to the previous state of the art result of 0.713 MRR on the CodeSearchNet benchmark.¹

1 Introduction

The availability of large scale datasets consisting of human written software in the past decade through platforms like Github has resulted in a fascinating array of tasks that can be performed with programming languages. These tasks are aimed towards improving developer productivity in different ways.

In this work, we focus on the natural language code search task, where the user input is a query in natural language and the system response is expected to be the most relevant piece of code from a large corpus of code snippets. Resources like StackOverflow are immensely useful for programmers with all levels of experience due to the natural language descriptions associated with the code snippets. However, such openly accessible community forums typically exist only for open-sourced libraries and it would be difficult to develop such forums for a new private tool without a large user community. Besides, such platforms may not be exhaustive in covering all the different functionalities of a software development tool.

Automated natural language (NL) to code search can be a promising framework to address these limitations. Recent work by Xu et al. (2021) studies the effectiveness of code generation and code retrieval tools when offered to a set of developers inside the IDE. They report that generation and retrieval modules complement each other in assisting the user. Particularly, they find retrieval modules are preferred over generation ones when the user is implementing more complex functionalities, thus endorsing the need for better code retrieval tools.

Given the significance of code search, we investigate current state-of-the-art approaches, which are primarily based on fine-tuning a pre-trained (on both natural and programming languages) encoder using a contrastive loss. Related work on text re-

¹Code and models are available at <redacted>
trieval (unimodal setup of NL only) (Xiong et al., 2021; Karpukhin et al., 2020) promotes the use of finding similar examples from the training corpus for use as negative candidates in the contrastive learning setup. We follow this line of ideas to design better representation learning schemes for the code search task. Our primary contributions in this direction are as follows:

- We first define the possible aspects of the NL code search problem that distinguish it from the unimodal text retrieval problem and identify limitations of the existing contrastive learning schemes that merely use local in-batch negatives to learn NL-PL representations for the semantic code search task (Section 2).
- We then propose DyHardCode (Figure 1), a contrastive learning framework that leverages global hard negatives, and compare multiple hard negative mining variants for the bimodal setup of NL-PL that lead to better representation learning for the NL-code search task. We further provide explanations to its effectiveness from the perspectives of optimization and adversarial learning (Section 3).
- We achieve state-of-the-art results on the CodeSearchNet code retrieval benchmark (Husain et al., 2019) for six programming languages, and also the AdvTest set of Python (Section 4).

2 Natural Language Code Search

We focus on the problem of returning a relevant code snippet from a given corpus \( C = \{y_1, \ldots, y_{|C|}\} \) for a natural language (NL) query \( x_q \). While there are multiple ways to use a deep learning model for this task, we follow the setup of Guo et al. (2021), where, based on the similarity score between the query embedding \( x_q \) and the candidate embeddings \( y_c \) (from the corpus \( C \)), we obtain a ranking for the candidates. We can then compute the average mean reciprocal ranking (MRR) over queries from the held out test set to evaluate the resulting code search model.

The training dataset to learn such a model for the natural language code search task consists of bimodal pairs \( \{x_i, y_i\} \), where we denote the NL description (docstrings) of the \( i \)-th datapoint by \( x_i \) and its corresponding programming language (PL) code\(^2\) (function or class) by \( y_i \). Given such a bimodal dataset, our goal is to learn good representations such that a vector representing a piece of code is close to the vector representing the docstring description of the code.

One alternative way to solve the code search task is to train a unified encoder that can take as its input the concatenation of the NL query \( x_q \) and a candidate code snippet \( y_c \), and return the probability of \( y_c \) being the correct response for the query (can be formulated as binary classification). Intuitively, such a model could benefit from the interactions between the NL and PL tokens in the self-attention layers of the Transformer (Vaswani et al., 2017). However, it would need \(|C|\) (size of the candidate code corpus) forward passes for each new query during inference, making it hard to scale to real world setups where \(|C|\) is large. In contrast to this, our approach (where we compute the NL and PL embeddings independently and then calculate their similarity) can benefit by processing the candidate PL embeddings from \( C \) offline, and we would only need to process the NL query \( x_q \) during inference.

It is also possible to combine these two approaches, by having a retrieval model that picks the top \( K \) candidates based on the similarity of the query and candidate embeddings (computed independently), followed by a unified encoder that ranks these top \( K \) candidates. While this approach could enjoy the advantage of using a more powerful model that operates on the concatenation \((y_c, x_q)\), it would be slightly slower and inefficient than the approach we have chosen for this work.

Let \( \theta \) denote the model parameters and \( f_\theta(x) \in \mathbb{R}^d \) be the model’s representation for input \( x \) (variable length sequence). While we could have different models for the two modalities, for simplicity, we will assume a single model \( f_\theta(\cdot) \) to obtain representations for both NL and PL inputs. For a model to be a good retriever, we need \( f_\theta(x_i) \) to be close to \( f_\theta(y_i) \in \mathbb{R}^d \) than to \( f_\theta(y_j) \) \( \forall j \neq i \), as per some similarity metric (e.g. cosine distance or \( L_2 \) distance); here \( \{x_i, y_i\} \) is a pair of the NL docstring \( x_i \) describing the code (PL part) \( y_i \). To have a good initialization for \( \theta \), we leverage recent work in code pre-training and make use of the transformer encoder based CodeBERT model (Guo et al., 2021) that is pre-trained on PL and NL using a hybrid objective of replaced token detection and masked language modeling. We also use the GraphCodeBERT model (Guo et al., 2021) that leverages the inherent structure of code by considering the data flow of the source code in its pre-training stage.

\(^2\)we use the term “PL” and “code” interchangeably
be random and a majority of them will be unrelated to the pair \( \{x_i, y_i\} \) under consideration. Xiong et al. (2021) provide theoretical results that identify issues like diminishing gradient norms, large gradient variances, and slow convergence when training retrieval models with such in-batch local negatives (Section 3.2 in Xiong et al. (2021)). With random mini-batches, majority of the negative samples are likely to be uninformative for the learning of useful representations for retrieval. While these analyses were made for text retrieval in the context of tasks like web search and open domain question answering, similar limitations could exist for our NL-PL bimodal setup, which are yet to be explored. One possible approach to overcome these issues is to construct mini-batches such that the examples within a minibatch are similar, but this could require sophisticated and expensive pre-processing, making it less preferable.

Recent work on text retrieval in the unimodal setup (NL only) (Karpukhin et al., 2020) has explored the use of negatives that are similar to the training instance \( x_i \). These informative instances are found by using discrete methods like TF-IDF or BM25 (Robertson and Zaragoza, 2009). Xiong et al. (2021) find that hard negatives can be directly found using the model that is being optimized, without using any sparse retrieval methods. Such improvements to retrieval have not been studied for the multi-modal setting of NL and Code. Our goal is to study the application of similar dense retrieval ideas to the NL-Code search problem and propose an effective solution that can perform better than the naive contrastive learning framework that uses in-batch negatives only. In Section 3, we study and compare the possible ways in which dense text retrieval methods can be adapted for our problem.

3 DyHardCode: Mining Global Hard Negatives Dynamically

To address the limitation of uninformative negatives, we propose to extract similar examples from the training corpus and use them as hard negatives in an online manner during training, while keeping the minibatch construction random. To facilitate this, we construct a FAISS index (Johnson et al., 2017) consisting of the representations of all the training set pairs: \( \{f_\theta(x_i)\}_{i=1}^N \) and \( \{f_\theta(y_i)\}_{i=1}^N \). The resulting objective \( \langle \theta \rangle \) being minimized is:

\[
\sum_{i=1}^{N} - \log \frac{\exp (f_\theta(x_i)^T f_\theta(y_i)/\sigma)}{\sum_{j \in B \cup H(i, K)} \exp (f_\theta(x_i)^T f_\theta(y_j)/\sigma)}
\]
where $H(i, K)$ represents the set of the top-$K$ hardest negatives for the $i$-th training instance $\{x_i, y_i\}$ globally (from the entire training corpus).

In the text retrieval setup (single modality of NL only), one could pick the nearest neighbors directly from a single FAISS index and use them as hard negatives. However, in the NL code search task, we have a number of possible choices as listed in Table 1. These variants differ in the query embedding we use: $f_\theta(x_i)$ or $f_\theta(y_i)$, and in the choice of the index being probed: NL index $\{f_\theta(x_j)\}_{j=1}^{|C|}$ or the PL index $\{f_\theta(y_j)\}_{j=1}^{|C|}$. We name this general framework of leveraging Dynamic Hard negatives for Semantic Code search as DyHardCode (illustrated in Figure 1). We note that the negative examples returned for an input instance are dynamic as the query embedding $f_\theta(x_i)$ or $f_\theta(y_i)$ will change with the model parameters $\theta$ being updated over the training iterations.

We train GraphCodeBERT using these variants on the CodeSearchNet Ruby corpus and report the performance in Table 1. We observe performance gains with all variants that leverage hard negatives over the choice of random negatives, highlighting that the quality along with the quantity of negative examples matter in contrastive representation learning. Given a batch $B$ of bimodal instances $\{x_i, y_i\}$, we obtain $K$ nearest neighbors for each training instance using one of the methods in Table 1. While the $K$ neighbors can serve as hard negatives for a training instance, we also utilize the $B - 1$ in-batch negatives and the $K \times (B - 1)$ neighbors returned for the fellow in-batch examples as candidate negatives. Thus the number of negatives for each instance would be $( (K + 1) \times B ) - 1$. We use this setup for all subsequent experiments that use hard global negatives. We chose the text-code variant for training the retrieval model on the 6 programming languages (Table 3) as it produces the best empirical performance (on the development set). We provide more justification for choosing this variant in Section 3.3.

### 3.1 Gradient Norms and Hard Negatives

Xiong et al. (2021) provide theoretical analysis that establishes the connection between the gradient norms and convergence rate (Section 3 of their paper). Intuitively, their analysis suggests that a negative instance with larger gradient norm is more likely to reduce the non-stochastic training loss, and hence should be sampled more often in the training mini-batch than the ones with diminishing gradients. Thus, a training scheme with larger gradient norms for the negatives would be more effective. Such correlation between larger gradient norms and better training convergence has also been reported for BERT fine-tuning (Mosbach et al., 2020).

In order to better understand the effectiveness of our hard negative mining strategies as per the above mentioned result, we record the training loss and the gradient norms of different layers of the transformer encoder in the GraphCodeBERT model. These are shown in Figure 2. In line with Xiong et al. (2021)’s results, we observe that the uninformative random negatives lead to lower loss and gradient norms, while global negatives maintain a higher gradient norm, which can reason the effectiveness of using hard negatives.

In Table 2, we present an example of a training instance from the Python CodeSearchNet corpus and the hard negatives (top nearest neighbor) obtained from the corpus using different mining variants. These are observed before the first training iteration, so the nearest neighbors are retrieved using the pre-trained GraphCodeBERT model embeddings that has seen no fine-tuning data. While the neighbor retrieved by text-text is semantically closest to the query, the outputs from text-code and code-code also share some structural similarities with the input (try and except blocks) as compared to the randomly picked code snippet which is fairly unrelated.

---

5As it is trivial for the model to classify the true pair from the trivially negative ones
Table 1: Comparison of different ways of designing hard negatives for the $i$-th training instance $\{x_i, y_i\}$. Strategies that leverage hard negatives (text-code, text-text and code-code) lead to better performance than using random uninformative ones (random), with the exception of code-text. We advocate using the text-code negative mining variant which is consistent with our training objective and leads to the best result on the development set. For this particular hard negative mining variant (text-code), we experiment with turning the FAISS index updates off (text-code; no index updates) to observe the importance of frequent index updates. All variants are trained for 10 epochs, with updates performed every epoch (no index updates for random). We use $K = 10$ negatives per training instance for all variants to have a fair comparison.

### 3.2 Asynchronous Index Update

The FAISS index (NL part: $\{f_{\theta_0}(x_i)\}_{i=1}^N$ and PL part $\{f_{\theta_0}(y_i)\}_{i=1}^N$) is constructed at the beginning of the training with the initial model parameters $\theta_0$. As training progresses and the model parameters are updated, the representations stored in the index would get stale. This could lead to poorer quality of neighbors returned for a query and worsen the quality of negatives. To mitigate this, we update the FAISS index with the latest model parameters after every epoch. The construction/updating of the index requires a forward pass over the entire training dataset, this requires a small fraction of the time required in the regular training epochs, and lesser computational resources.

To empirically verify the importance of these updates, we consider a variant of the text-code mining strategy where we do not update the FAISS index, labeled "text-code; no index updates". Table 1 shows the results with this strategy on the Ruby CodeSearchNet corpus. The drop in the MRR score validates the importance of the index updates. We also present the loss and gradient norms for the this particular variant in Figure 2. For most iterations, the training loss corresponding to the variant without index updates is higher than that of the random variant, but lower than the variants with index updates. This suggests that while it is a more challenging and informative setup than random (which happens to be an easy task due to trivially unrelated negatives, and training loss close to 0), other variants with index updates provide a stronger training signal for learning a retrieval model. The gradient norms corresponding to this variant (text-code; no index updates) happen to be lower than the variants with index updates, suggesting the effectiveness of updating the NL and/or PL index in improving the convergence of dense retrieval training.

### 3.3 DyHardCode as Adversarial Learning

DyHardCode can be interpreted as an implicit implementation of an adversarial learning algorithm. Let $\mathcal{L}(\theta, H)$ denote the contrastive loss defined in Equation 2 (we consider the temperature $\sigma$ as part of the parameters $\theta$), our hard negatives $H$ can be considered as adversaries that try to maximize , while we train $\theta$ to minimize :

$$\theta^* = \mathcal{L}(\theta, H^*)$$

where $H^* = \mathcal{L}(\theta, H)$

We optimize $H$ and $\theta$ alternatively for each
4 Experiments

We perform experiments with our DyHardCode on two NL code search tasks. The first one is on the popular CodeSearchNet corpus (4.1), while the second one is on an adversarial test (4.2) to show the robustness of our method.

4.1 Natural Language Code Search

We use the CodeSearchNet code corpus (Husain et al., 2019) to train our retrieval model. The dataset provides bimodal pairs (natural language docstring and corresponding code) in six programming languages – Python, Java, Go, Ruby, Php, Javascript. We replicate the setting of Guo et al. (2021) by filtering low quality queries using handcrafted rules and expanding the size of target set seen during inference from 1000 to the whole corpus to make the setup more realistic.

With the Mean Reciprocal Rank (MRR) as the
To evaluate the robustness of our proposed training scheme, we conduct evaluation on the CodeSearchNet AdvTest dataset from the CodeSearchNet corpus. The function and variable names appearing in the code snippets in the test and development sets of this Python dataset are normalized (func for function names, arg-i for the i-th variable name). This dataset was processed and released by Lu et al. (2016) that consists of 104 programming problems each with 500 solutions in C/C++. The evaluation metric of our codesearch task, the test results of previously proposed methods can be found in Table 3. GraphCodeBERT (Guo et al., 2021) has been pre-trained on code by considering the inherent structure of code (i.e. the data flow graph), instead of simply treating a code snippet as a sequence of tokens. This led to improvements over CodeBERT baselines for the codesearch task and is currently the state-of-the-art on this task. The training (fine-tuning for codesearch) scheme for all the baselines (top 8 rows in Table 3) uses the objective described in Eq. (1) and the test set results are as reported in Guo et al. (2021). During inference, all models compute the inner product of the query embedding and the candidate code embeddings as relevance scores to rank the code snippets in the corpus of the respective programming language.

We note that with both CodeBERT and GraphCodeBERT models, our DyHardCode training (fine-tuning for codesearch) scheme improves performance over the previous work. GraphCodeBERT model with our DyHardCode scheme leads to state of the art results on all six languages and an overall relative gain of 5.1%, demonstrating the effectiveness of using hard negatives.

### 4.3 Extension to Code-Code Search

We extend the idea of leveraging hard negatives in contrastive learning of representations for retrieval to the Code-Code search task. Here, the query $y_q$ and the set of candidates $\{y_i\}_{i=1}^{|C|}$ are both in the PL domain. We use the POJ-104 dataset (Mou et al., 2016) that consists of 104 programming problems each with 500 solutions in C/C++. The evaluation metric of our codesearch task, the test results of previously proposed methods can be found in Table 3. GraphCodeBERT (Guo et al., 2021) has been pre-trained on code by considering the inherent structure of code (i.e. the data flow graph), instead of simply treating a code snippet as a sequence of tokens. This led to improvements over CodeBERT baselines for the codesearch task and is currently the state-of-the-art on this task. The training (fine-tuning for codesearch) scheme for all the baselines (top 8 rows in Table 3) uses the objective described in Eq. (1) and the test set results are as reported in Guo et al. (2021). During inference, all models compute the inner product of the query embedding and the candidate code embeddings as relevance scores to rank the code snippets in the corpus of the respective programming language.

We note that with both CodeBERT and GraphCodeBERT models, our DyHardCode training (fine-tuning for codesearch) scheme improves performance over the previous work. GraphCodeBERT model with our DyHardCode scheme leads to state of the art results on all six languages and an overall relative gain of 5.1%, demonstrating the effectiveness of using hard negatives.

### 4.3 Extension to Code-Code Search

We extend the idea of leveraging hard negatives in contrastive learning of representations for retrieval to the Code-Code search task. Here, the query $y_q$ and the set of candidates $\{y_i\}_{i=1}^{|C|}$ are both in the PL domain. We use the POJ-104 dataset (Mou et al., 2016) that consists of 104 programming problems each with 500 solutions in C/C++. The evaluation metric of our codesearch task, the test results of previously proposed methods can be found in Table 3. GraphCodeBERT (Guo et al., 2021) has been pre-trained on code by considering the inherent structure of code (i.e. the data flow graph), instead of simply treating a code snippet as a sequence of tokens. This led to improvements over CodeBERT baselines for the codesearch task and is currently the state-of-the-art on this task. The training (fine-tuning for codesearch) scheme for all the baselines (top 8 rows in Table 3) uses the objective described in Eq. (1) and the test set results are as reported in Guo et al. (2021). During inference, all models compute the inner product of the query embedding and the candidate code embeddings as relevance scores to rank the code snippets in the corpus of the respective programming language.

We note that with both CodeBERT and GraphCodeBERT models, our DyHardCode training (fine-tuning for codesearch) scheme improves performance over the previous work. GraphCodeBERT model with our DyHardCode scheme leads to state of the art results on all six languages and an overall relative gain of 5.1%, demonstrating the effectiveness of using hard negatives.

### 4.2 CodeSearchNet AdvTest Set Evaluation

To evaluate the robustness of our proposed training scheme, we conduct evaluation on the CodeSearchNet AdvTest dataset from the CodeSearchNet corpus. The function and variable names appearing in the code snippets in the test and development sets of this Python dataset are normalized (func for function names, arg-i for the i-th variable name). This dataset was processed and released by Lu et al. (2016) that consists of 104 programming problems each with 500 solutions in C/C++. The evaluation metric of our codesearch task, the test results of previously proposed methods can be found in Table 3. GraphCodeBERT (Guo et al., 2021) has been pre-trained on code by considering the inherent structure of code (i.e. the data flow graph), instead of simply treating a code snippet as a sequence of tokens. This led to improvements over CodeBERT baselines for the codesearch task and is currently the state-of-the-art on this task. The training (fine-tuning for codesearch) scheme for all the baselines (top 8 rows in Table 3) uses the objective described in Eq. (1) and the test set results are as reported in Guo et al. (2021). During inference, all models compute the inner product of the query embedding and the candidate code embeddings as relevance scores to rank the code snippets in the corpus of the respective programming language.

We note that with both CodeBERT and GraphCodeBERT models, our DyHardCode training (fine-tuning for codesearch) scheme improves performance over the previous work. GraphCodeBERT model with our DyHardCode scheme leads to state of the art results on all six languages and an overall relative gain of 5.1%, demonstrating the effectiveness of using hard negatives.

### 4.3 Extension to Code-Code Search

We extend the idea of leveraging hard negatives in contrastive learning of representations for retrieval to the Code-Code search task. Here, the query $y_q$ and the set of candidates $\{y_i\}_{i=1}^{|C|}$ are both in the PL domain. We use the POJ-104 dataset (Mou et al., 2016) that consists of 104 programming problems each with 500 solutions in C/C++. The evaluation metric of our codesearch task, the test results of previously proposed methods can be found in Table 3. GraphCodeBERT (Guo et al., 2021) has been pre-trained on code by considering the inherent structure of code (i.e. the data flow graph), instead of simply treating a code snippet as a sequence of tokens. This led to improvements over CodeBERT baselines for the codesearch task and is currently the state-of-the-art on this task. The training (fine-tuning for codesearch) scheme for all the baselines (top 8 rows in Table 3) uses the objective described in Eq. (1) and the test set results are as reported in Guo et al. (2021). During inference, all models compute the inner product of the query embedding and the candidate code embeddings as relevance scores to rank the code snippets in the corpus of the respective programming language.

We note that with both CodeBERT and GraphCodeBERT models, our DyHardCode training (fine-tuning for codesearch) scheme improves performance over the previous work. GraphCodeBERT model with our DyHardCode scheme leads to state of the art results on all six languages and an overall relative gain of 5.1%, demonstrating the effectiveness of using hard negatives.
Section 5 Related Work

Advances in deep learning for NLP and the abundance of source code data has accelerated research on several tasks in the PL domain. Code completion systems (Svyatkovskiy et al., 2020), for instance, offer possible completions to incomplete prompts in the source code domain and can aid developers in writing code faster. Similarly, text to code generation (Yin and Neubig, 2018; Yin et al., 2018; Iyer et al., 2018; Xu et al., 2020) systems generate a source code sequence that solves the task described in the input natural language description.

In the NL domain, our work is closely related to dense text retrieval approaches of Xiong et al. (2021) and Karpukhin et al. (2020) in the unimodal setup. They propose the use of additional informative negatives besides the in-batch ones for effective contrastive learning. Jain et al. (2020) propose contrastive learning as a pre-training strategy for general PL tasks like source code summarization and PL sequence classification.

In computer vision research, contrastive learning based frameworks have been studied extensively for image representation learning (He et al., 2020; Oord et al., 2018). Self-supervised contrastive learning enforces two augmented embeddings of the same image to be close while embeddings of different images are pushed apart. SimCLR (Chen et al., 2020) shows that an appropriate temperature can help the model learn from hard negatives. Robinson et al. (2021) explicitly mine hard negative examples to improve representation learning performance. CLIP (Radford et al., 2021) shows that a simple image-text contrastive learning on large-scale datasets learns superior image representations.

### Table 4: Results on the adversarial test set (Lu et al., 2021) of the CodeSearchNet (Python). $K = 10$ for DyHardCode.

<table>
<thead>
<tr>
<th>Model/Method</th>
<th>Test MRR</th>
<th>Train Batchsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoBERTa</td>
<td>0.1833</td>
<td></td>
</tr>
<tr>
<td>CodeBERT</td>
<td>0.2719</td>
<td>32</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>0.3314</td>
<td>128</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>0.3419</td>
<td>384</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>0.3433</td>
<td>512</td>
</tr>
<tr>
<td>DyHardCode</td>
<td>0.3784</td>
<td>64</td>
</tr>
</tbody>
</table>

### Table 5: Results on the code-code search task. POJ-104 test set (Lu et al., 2021). $K = 1$ for DyHardCode.

<table>
<thead>
<tr>
<th>Model/Method</th>
<th>Test MAP</th>
<th>Train Batchsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoBERTa</td>
<td>0.7677</td>
<td></td>
</tr>
<tr>
<td>CodeBERT</td>
<td>0.8267</td>
<td>32</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>0.8882</td>
<td>160</td>
</tr>
<tr>
<td>DyHardCode</td>
<td>0.8910</td>
<td>160</td>
</tr>
</tbody>
</table>

### 6 Conclusion & Future Directions

We propose the use of global hard negatives in the contrastive learning of NL and PL representations for the task of code search. We compare multiple variants of obtaining these global negatives for a training instance, and find that probing the NL index with the query NL embedding is an effective strategy, and further report that this benefits from updating the index being updated with newer model checkpoints saved during training.

Our current method finds hard negatives by a simple nearest neighbor search based on cosine similarity. However, work in cross-lingual embedding learning shows that in high dimensional spaces this nearest neighbor finding approach leads to a detrimental phenomenon known as the hubness problem (Dinu et al., 2015), where a few nodes (embeddings) become hubs (nearest neighbors of many other nodes), whereas some others become anti-hubs (nearest neighbors to none). Since we also operate on bimodal data, this phenomenon could also affect our search. In future, we would like to investigate the Cross-domain Similarity Local Scaling (CSLS) that penalizes the embeddings that are close to many other in the target space to mitigate the hubness problem (Conneau et al., 2018). There also has been significant recent work in unsupervised representation learning of images using the contrastive loss (mitrovic et al., 2021; Grill et al., 2020), ideas from this string of research can also motivate more progress in training better code search models.
References


Jovana Mitrovic, Brian McWilliams, Jacob Walker, Lars Buesing, and Charles Blundell. 2021. Representation learning via invariant causal mechanisms. ICLR.


<table>
<thead>
<tr>
<th></th>
<th>Go</th>
<th>Java</th>
<th>Javascript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training examples</td>
<td>167,288</td>
<td>164,923</td>
<td>58,025</td>
</tr>
<tr>
<td>Dev queries</td>
<td>7,325</td>
<td>5,183</td>
<td>3,885</td>
</tr>
<tr>
<td>Testing queries</td>
<td>8,122</td>
<td>10,955</td>
<td>3,291</td>
</tr>
<tr>
<td>Candidate codes</td>
<td>28,120</td>
<td>40,347</td>
<td>13,981</td>
</tr>
</tbody>
</table>

Table 6: Data statistics of the filtered CodeSearchNet corpus for Go, Java and Javascript programming languages. For each query in the dev and test sets, the answer is retrieved from the set of candidate codes (last row).

<table>
<thead>
<tr>
<th></th>
<th>PHP</th>
<th>Python</th>
<th>Ruby</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training examples</td>
<td>241,241</td>
<td>251,820</td>
<td>24,927</td>
</tr>
<tr>
<td>Dev queries</td>
<td>12,982</td>
<td>13,914</td>
<td>1,400</td>
</tr>
<tr>
<td>Testing queries</td>
<td>14,014</td>
<td>14,918</td>
<td>1,261</td>
</tr>
<tr>
<td>Candidate codes</td>
<td>52,660</td>
<td>43,827</td>
<td>4,360</td>
</tr>
</tbody>
</table>

Table 7: Data statistics of the filtered CodeSearchNet corpus for PHP, Python and Ruby programming languages. For each query in the dev and test sets, the answer is retrieved from the set of candidate codes (last row).

A Experimental details

Computing Infrastructure: All our experiments are conducted using the Nvidia A-100 GPUs via the Google Cloud Platform, each of which has 40 GB of RAM. The maximum number of GPUs we use is 8 for an experiment using PyTorch’s dataparallel package. Training duration for 10 epochs of GraphCodeBERT or CodeBERT for the results in Table 3 for the ruby, javascript, go, python, java, php datasets require (roughly) 2.5, 7, 29.5, 59.5, 28.5, 55 hrs respectively.

To select the hyper-parameter $K$ (number of hard negatives) for a chosen batch-size, we perform 3 training runs of the GraphCodeBERT model with our objective on the ruby dataset and try $K = \{2, 4, 6, 8, 10\}$. The average MRR scores were $\{0.7851, 0.7867, 0.7860, 0.7854, 0.7869\}$, thus we choose $K = 10$ for NL-code search. Given finite GPU memory, the optimal way to balance batch-size with $K$ is not straightforward and performing a grid search on the two will be prohibitively expensive, which is why we did not tune these choices.

The CodeBERT and GraphCodeBERT pre-trained models we use in our experiments both have 125M parameters.

Dataset details: The CodeSearchNet corpus we use in our experiments is pre-processed in the same manner as done by (Guo et al., 2021) and its de-
tailed statistics are mentioned in Table 6. The Python AdvTest set consists of 251,820 training pairs, 9,604 validation set examples and 19,210 test examples. POJ-104 dataset consists of 104 problems each of which has 500 solutions in C/C++ and is divided into a training set of 64 examples, dev set of 16 examples and test set of 24 examples.