Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Contrastive Learning for Programming Languages

Bachelor's Thesis

Justin Studer

jstuder@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Zhao Meng
Prof. Dr. Roger Wattenhofer

November 3, 2021

Acknowledgements

I want to thank Zhao Meng for supporting me in writing this thesis by providing me with fantastic resources and by offering invaluable inputs both in our discussions and in the form of feedback whenever I reported on my progress. Through his knowledge, he was often able to gently nudge me into the right direction so I wouldn't run into problems later on, yet despite that he gave me the freedom to explore many different ideas, and this overall made working on this thesis very enjoyable.

I also want to thank Zihan Zhang from the Australian National University for running many of the ablations for me. He pointed out to me several errors in my implementations which would have gone unnoticed had he not spent his time on assisting me in my experiments.

And finally, I want to thank the team at TIK for providing me with computational resources which enabled me to run all of the experiments in this work.

Abstract

Machine Learning can assist programmers in various ways, such as by retrieving fitting code snippets from a database to serve as inspiration. Previous approaches to this problem relied on bimodal encoders. We propose to use unimodal encoders, which specialize on just one modality and show that this, in combination with other performance enhancing methods, outperforms prior works to achieve State-of-the-Art performance in the CodeSearchNet benchmark. To that end, we make use of a framework called xMoCo and enhance it using a novel approach of incorporating a Barlow loss as a regularization term, which boosts performance across the board.

Contents

Chapter 1

Introduction

In recent years, Machine Learning has made its way into the vocabulary of nearly every person that uses computers. While it is quickly becoming one of the most effective buzzwords for advertising, recent research manages to apply the idea of making computers "learn for themselves" to many new practically useful areas. These areas can be as "simple" as translating between languages or as complicated as making a car safely drive itself [1].

One area of applications for Machine Learning is in assisting programmers with writing code. While platforms such as GitHub or StackOverflow offer a vast amount of knowledge in solving programming problems, the task of finding suitable examples of code to adapt can be a tedious one. Machine Learning can assist the programmer in various ways, such as offering code autocompletion or by retrieving code samples that could be suitable to solve the task at hand.

1.1 Code Search

The task of retrieving suitable code extracts from a large database of code is called "Code Search" (sometimes also called "Code Retrieval"). Here, the programmer would be able to give a retriever program a problem query in natural language (similar to how one interacts with search engines like Google Search) and be presented with a list of code extracts that might solve the query problem. The difficulty in Code Search is to retrieve code that is semantically related to the input query. In general, this requires the retriever program to "understand" both how natural language and programming languages are structured, what their semantics are and how they relate to each other.

Intuitively, code search can be interpreted as a translation task where the input language is English (or any other human language) and the target is some programming language, where the translation must be retrieved from an existing database of code in that programming language.

While the concept of autocompletion is closely related to Code Search, we have to emphasize here that at their core these two programming aids solve

different problems. Autocompletion attempts to synthesize new code by anticipating what the programmer wants to do, while Code Search retrieves code from a fixed set of samples. Since neither of these problems have been fully solved yet, a combination of both would be ideal, as recent research suggests [2], with autocompletion reducing the amount of time spent on boilerplate code and Code Search providing the programmer with inspiration on how to solve problems that autocompletion fails to solve on its own. While a significant practical usefulness of this has not been shown just yet, it is still worth investigating whether we can improve on previous methods.

1.2 Encoders and the latent space approach

One popular method of tackling Code Search is by using so-called encoders. Encoders are programs that take some input data and transform that data into some other representation. As an analogy, take the example of a camera. A camera takes in light intensity data and turns it into a digital representation that is suitable for being stored. If desired, encoders can reduce the dimensionality of the input data to save storage at the cost of reducing the reconstruction quality.

Similar to images, we can also encode text, but this time in the hope of turning it into a representation that makes it possible for the computer to more easily compare it to other texts. This is the idea of the "latent space", which is a scary sounding term that is used in Machine Learning to say that the data was somehow encoded. Encodings in the latent space are typically called "embeddings" and are nothing more than some (usually) high-dimensional vectors.

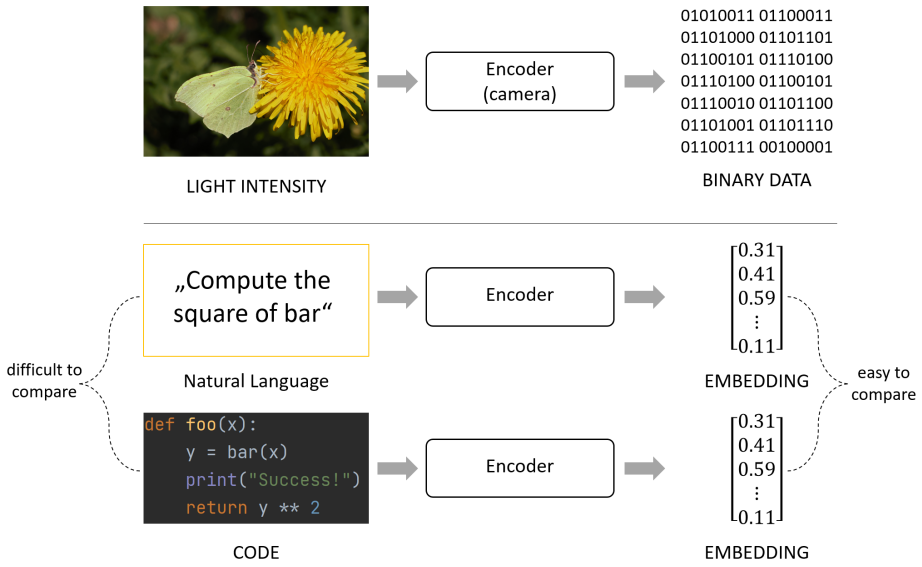

Figure 1.1: The concept of an encoder

Recent methods in Code Search use embeddings to relate natural language and programming languages with each other. Concretely, an encoder is used to transform the natural language and the programming language into the latent space, where it is possible to compare them with each other more easily. This is the basic idea that this thesis revolves around. In the following chapters, we will explain how prior methods apply this idea to make the computer learn the relation between these two types of text data, and how we can improve on them.

1.3 Current methods and our contribution

Current methods in Code Search generally rely on the so-called "Contrastive loss". This loss is explained in more detail in Chapter 3. Intuitively, this loss function attempts to maximize some notion of similarity between embeddings of positive pairs and minimize similarity between embeddings of negative pairs. During inference, we can compute the similarity of an encoded input query and a database of encoded code samples. The code sample with the highest similarity can then be proposed as a solution to the given query.

Recent models such as CodeBERT [3] are encoders that rely on the Self-Attention mechanism and are based on the Transformer architecture [4]. These encoders are pre-trained on publicly available Documentation-Code pairs and then fine-tuned for the Code Search task using the contrastive loss. While they show promising results, they use a unified encoder for both Documentation and Code. We conjecture that this hinders performance and propose to use a different existing framework that is more suitable for multimodal data, called Cross Momentum Contrastive Learning (xMoCo) [5]. xMoCo makes it possible to use separate encoders for Documentation and Code and adds some mechanism to stabilize training and improve scores. Additionally, we use hard negative sampling as proposed in both xMoCo and a different framework called DyHardCode [6] to improve performance. Finally, we combine this framework with a Barlow loss as proposed in Barlow Twins [7] and investigate the effect the simple augmentation method "Easy Data Augmentation" (EDA) [8] can have on performance.

Overall, our contributions consist of the following:

- Apply xMoCo to Code Search to show that solutions to this problem can benefit from separate encoders.
- Combine xMoCo with hard negatives to improve performance in the Code Search task.
- Propose the use of Barlow loss as a regularization term and show its potential by achieve State-of-the-Art performance in the CodeSearchNet benchmark.
- Investigate whether simple Natural Language augmentation can benefit performance in Code Search.

Chapter 2

Related work

2.1 Contrastive learning

Contrastive learning [9] has made possible quick advancements in areas of research such as Computer Vision. There, much effort is put into closing the gap between supervised and self-supervised learning, and frameworks like SimCLR [10] and MoCo [11, 12, 13] have shown just how effective contrastive learning can be. These methods generally rely on augmentation of single-modality data (images) to obtain positive pairs. This is unfortunately unsuitable for many Natural Language tasks, where we often already have positive pairs from different modalities (e.g. different languages).

2.2 xMoCo

Cross Momentum Contrastive Learning (xMoCo) [5] is a framework that proposes to use multiple encoders to better handle multimodal data. It builds on ideas from the Momentum Contrast framwork (MoCo) [11] to circumvent the limitations of relying just on in-batch negatives by building a consistent dictionary of negative samples on-the-fly. Contrastive learning strongly depends on the number and quality of negative samples, so using more samples is one way to improve learning performance.

MoCo proposes to use a slow encoder (or momentum encoder) and queue to use negatives from past iterations without the need of forwarding them through the encoder again. The slow encoder is an additional encoder which is updated not by backpropagation but by computing a weighted average of the weights of the main/fast encoder. The slow encoder is essentially a stabilized representation of the fast encoder and hence produces consistent results over several iterations. Through the slow encoder, one can forward samples and store them in a buffer (a queue) for use as negative samples in later iterations. Because the slow encoder is a stabilized version of the fast encoder, the queue does not suffer from strong fluctuations. After some time, however, these samples will get stale and need

to be replaced, which is why the queue size is limited. A schematic of MoCo is shown in Figure 2.1.

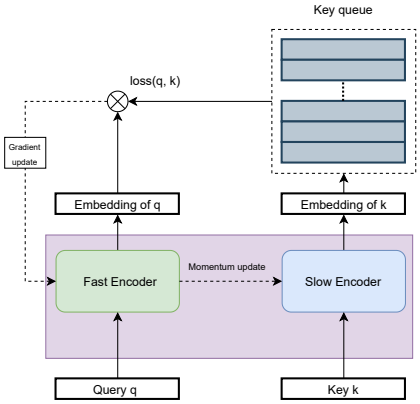

Figure 2.1: Schematics of MoCo

xMoCo builds on top of this by using one fast and slow encoder as well as a queue per modality. This makes it possible to use split encoders for multimodal data with contrastive learning. A schematic of xMoCo is shown in Figure 2.2. Notice that the loss is now the combination of losses obtained by combining the query with the key queue and the key with the query queue.

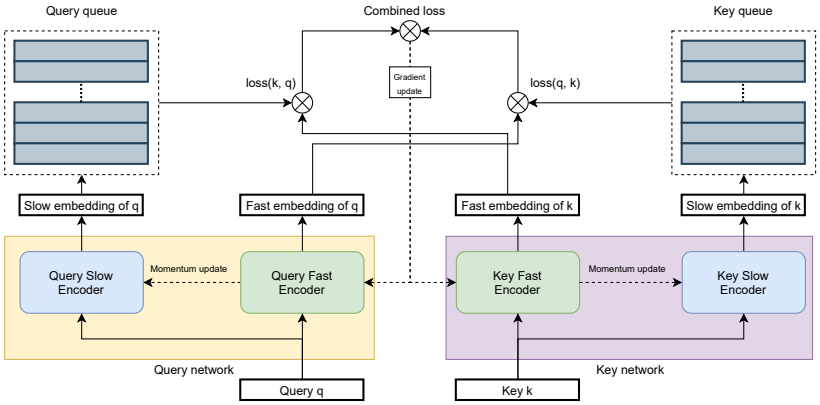

Figure 2.2: Schematics of xMoCo

2.3 DyHardCode

DyHardCode [6] is a framework that takes a different approach to obtaining more (meaningful) negative samples, namely hard negative sampling. It leverages new methods of fast nearest-neighbour search [14] of pre-encoded embeddings to find potential hard negative samples during iterations. These embeddings are

computed before training, and updated after every epoch to reduce staleness. Once found, the corresponding sample will be forwarded through the current encoder to obtain an updated embedding for it.

2.4 CodeBERT, GraphCodeBERT and SynCoBERT

The results of DyHardCode largely rely on models pre-trained on Natural Language and Programming Languages. The first of these models is CodeBERT [3], which was pre-trained for multiple Programming Languages on the CodeSearchNet corpus [15] using Masked Language Modelling (MLM), where the model is trained by making it predict tokens which have been masked out from the input sequence. Additionally, it uses Replaced Token Detection (RTD) to learn from unannotated code. The second one is GraphCodeBERT [16], a model pre-trained only on code that considers data flow for improved performance. The corresponding paper also introduces new pre-training tasks to effectively use this additional information. SynCoBERT is a different model that uses information extracted from the Abstract Syntax Tree (AST) such as which parts of the code correspond to identifiers and where edges in the AST are for pre-training. All of these models can later be fine-tuned on the Code Search task.

2.5 CodeSearchNet

To have a comparable metric for the performance of solutions to the Code Search task, the CodeSearchNet Challenge [15] was created. CodeSearchNet is a dataset consisting of many Documentation-Code pairs from six different Programming Languages obtained through non-forked open-source GitHub repositories. A pre-filtered version thereof [16] is used for fine-tuning and benchmarking of Code Search methods. Several benchmarks for different tasks have been derived from CodeSearchNet and were aggregated in the CodeXGLUE dataset[17], of which the AdvTest benchmark is of particular interest for this work. Said test contains the code samples from CodeSearchNet, but normalized, which gives a better look into how well the model generalizes. While Code Search is the topic of interest for this work, the CodeSearchNet dataset can also be used for related problems such as documentation generation and code autocompletion.

2.6 Barlow Twins

Similar to MoCo, Barlow Twins [7] is a framework originally intended for self-supervised learning in Computer Vision. It does not rely on a contrastive loss but rather uses an entirely different loss we refer to as ”Barlow loss” that depends

on the cross-correlation matrix of samples. In particular, it aims to minimize redundancy in the embedding features and can benefit from larger embedding sizes than contrastive learning.

2.7 Augmentation

Augmentation can be used as a way of regularization and is particularly useful when we otherwise don't have enough data. It alters the data in sometimes humanly imperceivable ways and can make the model more robust to perturbations. Easy Data Augmentation [8] is a simple method of augmenting Natural Language through synonym replacement and other techniques. While not as sophisticated as methods like backtranslation, it is computationaly inexpensive in comparison. As previous methods for Code Search did not use any augmentation, it might be worth investigating whether there is any potential in using such a tool.

Chapter 3

Method

3.1 Contrastive learning

The contrastive learning objective was originally proposed in [9]. For a set of pairs $\{(x_i, y_i)\}_{i=1}^{N}$, one typically [17, 3, 6] formulates it as follows:

$$\min_{\theta} \sum_{i=1}^{N} -\log \frac{\exp(f_\theta(x_i)^T f_\theta(y_i)/\sigma)}{\sum_{j \in B_i} \exp(f_\theta(x_i)^T f_\theta(y_j)/\sigma)} \tag{3.1}$$

where f_θ is a function parameterized by θ, N is the total number of training pairs, B_i is the set of indices of the samples in the batch that contains i and σ is a temperature hyperparameter. Intuitively, this objective aims to maximize the dot products between embeddings of positive pairs (x_i, y_i) and to minimize dot products between negative pairs (x_i, y_j) where $i \neq j$ over the entire training set. Equivalently, we can formulate the following loss function:

$$\mathcal{L}_{\text{CONTRAST}} = \sum_{i \in B} -\log \frac{\exp(f(x_i)^T f(y_i)/\sigma)}{\sum_{j \in B} \exp(f(x_i)^T f(y_j)/\sigma)} \tag{3.2}$$

where B is the current batch. Notice that only in-batch samples are used as negatives, which is why it is desirable to have a large batch size or to employ some other method of obtaining more negatives, as will be explored in the following sections.

This objective is useful because it enables us to have a notion of "similarity" between sample pairs, where a large dot product implies high similarity and a small dot product implies low similarity. The function f is typically chosen to generate a length-normalized high-dimensional vector, which means that the products $f_\theta(x_i)^T f_\theta(y_j)$ form a cosine similarity, because for two vectors a and b it holds that:

$$||a|| = ||b|| = 1 \implies \text{CosineSimilarity}(a, b) = \frac{a^T b}{||a|| \cdot ||b||} = a^T b \tag{3.3}$$

What this in turn means for our purposes is that we minimize angles between positive sample pairs and maximize angles between negative sample pairs. For certain frameworks like xMoCo, the contrastive learning objective has to be altered slightly, as discussed in the next section.

3.2 xMoCo

xMoCo [5] is a framework that extends MoCo [11, 12, 13] to work with multiple encoders. For each modality, xMoCo has a ”fast encoder” and a ”slow encoder”, the latter of which is a moving average of the respective fast encoder. These slow encoders are less prone to fluctuations, which means that their parameters change much slower and the outputs stay roughly the same (consistent) for longer than is the case for the fast encoders. This makes it possible to build a consistent dictionary of embeddings that can be used as negatives in the contrastive learning objective. When using the contrastive learning objective, involving more negatives makes it possible to learn better encoder functions as evidenced by xMoCo [5], MoCo [11, 12, 13] and DyHardCode [6]. While the optimal solution here would be to encode the entire database with the fast encoder in every iteration and to use all samples, this approach would be computationally wasteful. The dictionary is a tradeoff between this optimum and required training time. However, a problem the dictionary has is the potential for staleness of the embeddings which could hurt performance. This is mitigated by limiting the size of the dictionary and by defining it as a queue that replaces the oldest embeddings with new ones after every iteration.

During training, the samples in the current batch are forwarded through both the slow and fast encoder and the produced embeddings length-normalized. The embeddings from the fast encoder serve as positive samples and in-batch negatives like in Formula 3.1. The embeddings from the slow encoder, however, will be put into the queue after the current iteration. A schematic of xMoCo for our modalities is shown in Figure 3.1.

The embeddings that are already present in the queue can be used as negatives by computing the pairwise similarities of the positive samples to them. Computing the pairwise similarity is simple and computationally inexpensive because of the normalization: Simply compute all pairwise dot products of the embeddings to obtain their cosine similarity.

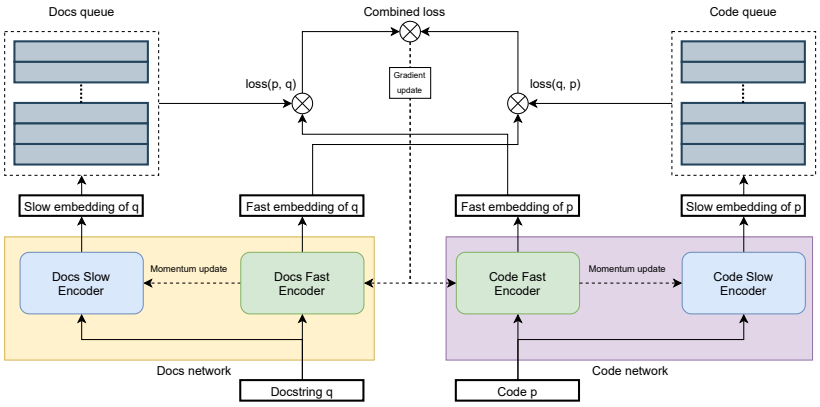

Figure 3.1: Schematic of xMoCo with the "Docstring" and "Code" modalities

With the obtained similarities, we can define two loss functions (one for each modality) that incorporate the additional negatives:

$$\mathcal{L}_{\text{DOCS}} = \sum_{i=1}^{n} -\log \frac{\exp(d(x_i)^T c(y_i)/\sigma)}{\sum_{j \in B} \exp(d(x_i)^T c(y_j)/\sigma) + \sum_{k \in Q_{\text{C}}} \exp(d(x_i)^T k/\sigma)} \tag{3.4}$$

$$\mathcal{L}_{\text{CODE}} = \sum_{i=1}^{n} -\log \frac{\exp(c(y_i)^T d(x_i)/\sigma)}{\sum_{j \in B} \exp(c(y_i)^T d(x_j)/\sigma) + \sum_{k \in Q_{\text{D}}} \exp(c(y_i)^T k/\sigma)} \tag{3.5}$$

where d is the fast encoder for the Documentation strings, c is the fast encoder for the Code strings and n is the batch size. B is the set of in-batch indices and Q_D, Q_C are the embeddings in the queue (Documentation/Code respectively). Notice that in these equations, the queue embeddings stem from the respective other modality and that we take the pre-encoded embeddings from the queue directly instead of forwarding them again. Putting these together, we get our final loss function:

$$\mathcal{L}_{\text{COMBINED}} = \frac{\mathcal{L}_{\text{DOCS}} + \mathcal{L}_{\text{CODE}}}{2} \tag{3.6}$$

It is not strictly necessary to have a factor $\frac{1}{2}$ here, but we chose to use it regardless for monitoring purposes. In the end, this factor will be balanced out by the choice of learning rate. For our purposes, we adapted xMoCo for the Code Search task and combined it with recent pre-trained models called CodeBERT [3] and GraphCodeBERT [16] and fine-tuned them. We also experimented with optional hard negatives and a Barlow loss for regularization. These principles are described in the next sections.

3.3 Hard negatives

A different (though not orthogonal) approach to obtaining more samples for the contrastive learning objective is hard negative mining. In contrast to how the queues in xMoCo work, hard negative samples are not obtained at random from shuffled batches, but are specifically chosen because they are hard for the encoders to separate from the positive embeddings. In some sense, they are more meaningful for learning and can thus improve performance. DyHardCode [6] demonstrates this well by achieving massive improvements over its previous State-of-the-Art [16] by simply incorporating hard negative mining.

Negative mining works by computing the embeddings of the entire dataset before every epoch. During training, we can then obtain the indices of potential hard negatives by running a nearest-neighbour search (in terms of cosine similarity) over these embeddings. Similar to the queue, it is important to update the embeddings periodically to combat staleness. However, for hard negative mining it is sufficient to update all embeddings once per epoch. This is because we only use these to find indices of potential hard negatives, not their actual embeddings. Once found, the corresponding samples are forwarded once again to get updated embeddings.

An issue that can arise here are false negatives. Since we sample the embeddings of the entire dataset, it can happen that a sample's own index is returned as its nearest neighbour. To prevent any negative impact this can have, we can simply mask out embeddings that stand in conflict with the in-batch samples.

To enhance the efficiency of this process, it is also possible to put the newly encoded embeddings in their own queue for use as (not necessarily hard) negatives in later iterations. We can combine hard negatives and xMoCo by adapting the loss functions:

$$\mathcal{L}_{\text{HDOCS}} = \sum_{i=1}^{n} -\log \frac{\exp(d(x_i)^T c(y_i)/\sigma)}{\sum_{j\in B} \exp(d(x_i)^T c(y_j)/\sigma) + \sum_{k\in Q_{\text{C}}} \exp(d(x_i)^T k/\sigma) + \mathcal{H}_{\text{C}} + \mathcal{HQ}_{\text{C}}} \tag{3.7}$$

with the hard negative term

$$\mathcal{H}_{\text{C}} = \sum_{h\in H_C} \exp(d(x_i)^T c_{\text{slow}}(y_h)/\sigma) \tag{3.8}$$

and the hard negative queue term

$$\mathcal{HQ}_{\text{C}} = \sum_{h\in HQ_C} \exp(d(x_i)^T h/\sigma) \tag{3.9}$$

where H_{C} are the indices of the mined Code hard negatives and HQ_{C} are the embeddings in the "hard negative" queue. Note that these embeddings are not

necessarily hard negatives for the samples in the current batch because they were mined for previous batches. Also notice that in Equation 3.8 we use the slow encoder c_{slow} because we found this to produce better scores, as further explained in Chapter 4.3.2. $\mathcal{L}_{\text{HCODE}}$ is defined analogously. These two losses are then added to form $\mathcal{L}_{\text{HCOMBINED}}$ like in Equation 3.6.

3.4 Barlow regularization

An entirely different idea that does not rely on the contrastive learning objective was proposed for self-supervised Visual Representation learning in the paper "Barlow Twins" [7], called the "Barlow loss". This loss function computes the empirical cross-correlation matrix of the samples and determines the loss value based on its difference to an identity matrix. The loss is minimized when all entries on the diagonal are 1 and all off-diagonal elements are 0. The importance of on/off-diagonal can be traded off using an additional hyperparameter. As the authors of the paper describe it, this loss function minimizes the data redundancy of the embedding features.

Concretely, the authors of Barlow Twins define the loss function as follows:

$$\mathcal{L}_{\text{BARLOW}} = \sum_i (1 - \mathcal{C}_{ii})^2 + \lambda \sum_i \sum_{j \neq i} \mathcal{C}_{ij}^2 \tag{3.10}$$

where λ is a positive constant trading off the importance of on-diagonal and off-diagonal terms and $\mathcal{C}$ is the cross-correlation matrix of the current embeddings standardized and computed along the batch dimension.

While not immediately useful for our task of computing similarities, we can incorporate this loss into the xMoCo framework as a regularization term. Like before, we forward all samples through the fast and slow encoder, but for this loss we only use the (normalized) embeddings of the fast encoder. These can optionally be fed through an additional projector to obtain higher-dimensional representations. This was shown to be useful for the Barlow loss for other problems [7], so we also explore this option in this work. Being required to reduce this additional loss, the overall objective becomes harder to minimize, but that could improve generalization performance. We can adjust the impact this term has by adding a weight hyperparameter for it. Incorporating this into the previous framework is straightforward and it also works with hard negatives. A schematic of it is shown in Figure 3.2.

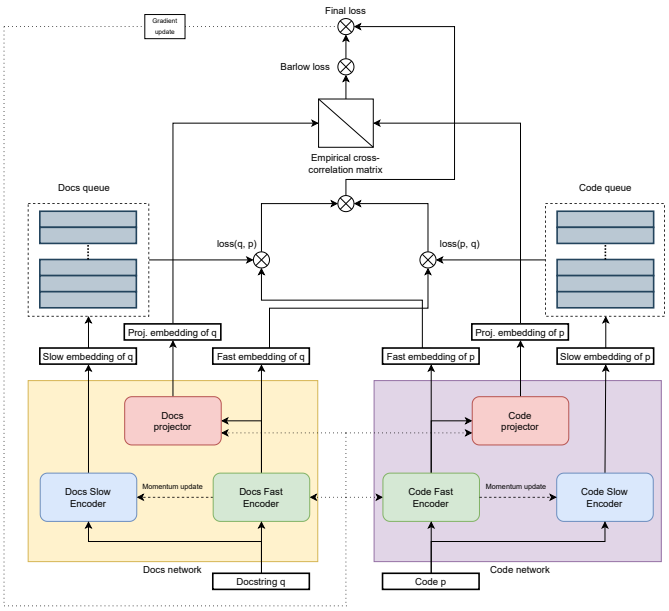

Figure 3.2: Schematic of xMoCo with added Barlow loss

The new loss then looks as follows:

$$\mathcal{L}_{\text{BCOMBINED}} = \mathcal{L}_{\text{COMBINED}} + \beta\mathcal{L}_{\text{BARLOW}} \tag{3.11}$$

where β is the aforementioned weight for the Barlow loss. This can be done analogously with the hard negative loss $\mathcal{L}_{\text{HCOMBINED}}$ to define $\mathcal{L}_{\text{BHCOMBINED}}$.

3.5 Augmentation

A different regularization method useful for many problems is augmentation. In augmentation, the input data is somehow transformed to create perturbations. These perturbations reduce the potential for overfitting and generally make the model more robust. This can be particularly useful when the amount of training data is limited. Previous Code Search solutions did not use any augmentation, which is why we decided to explore its potential for this particular task. For our purposes, we exclusively augmented the Documentation part of the data, as no off-the-shelf solution for augmentation of Code was available.

There are several methods of augmenting Natural Language, such as back-translation, where (usually) Machine Translation is used to translate the input data to a different language and back. Because Machine Translation typically isn't bijective, the output of this process will be an altered version of the original

data. However, this comes with a heavy computational cost. For our purposes, we employed a much simpler method called Easy Data Augmentation (EDA) [8].

EDA leverages a synonym database to augment input data. It has four main components that perturb the input, as described by the paper's authors:

- "Synonym Replacement (SR): Randomly choose n words from the sentence that are not stop words. Replace each of these words with one of its synonyms chosen at random."
- "Random Insertion (RI): Find a random synonym of a random word in the sentence that is not a stop word. Insert that synonym into a random position in the sentence. Do this n times."
- "Random Swap (RS): Randomly choose two words in the sentence and swap their positions. Do this n times."
- "Random Deletion (RD): Randomly remove each word in the sentence with probability p."

The number of selected words n can be adapted based on the length of the input sequence, for example by defining it as $n = \alpha l$, where α is the fraction of words that should be replaced and l is the length of the sequence.

Chapter 4

Experiment details

4.1 Dataset

The datasets we used are based on the CodeSearchNet Challenge dataset [15]. This dataset consists of many bimodal Documentation-Code pairs as well as unannotated code snippets from 6 programming languages. This data was collected from publicly available GitHub repositories. The authors of GraphCodeBERT [16] pre-filtered the bimodal pairs to create a cleaned-up version of CodeSearchNet, which we will use for our experiments for better comparison. Their filtering process consisted of removing comments in the code, removing faulty (unparseable) code, removing samples with less than 3 or more than 256 tokens, removing samples with special tokens such as URLs in their documentations and removing samples where the documentation is not in English. Table 4.1 shows statistics for the individual programming languages for the filtered dataset.

Programming Language	Training samples	Dev samples	Test samples
Go	167,288	7,325	8,122
Java	164,923	5,183	10,955
JavaScript	58,025	3,885	3,291
PHP	241,241	12,982	14,014
Python	251,820	13,914	14,918
Ruby	24,927	1,400	1,261

Table 4.1: Statistics for the different programming language datasets (taken from [16])

An additional dataset we used, called AdvTest [17], which contains samples exclusively from Python, is derived from the CodeSearchNet dataset. It contains $19,210$ test samples which were normalized. This means that function names and variables are replaced with special tokens to make it harder for the retriever to distinguish between codes. In particular, the results on this test set should give a better estimate on the generalization ability of models.

4.2 Evaluation method

For evaluation, we use the Mean Reciprocal Rank (MRR) defined in Equation 4.1. To obtain the rank for every sample in the test set, we give our retriever the Documentation string and compute the position of the correct code snippet in the similarity ranking.

$$MRR = \frac{1}{|T|} \sum_{i=1}^{|T|} \frac{1}{\text{rank}_i} \tag{4.1}$$

where T is the test set and rank_i is the rank of the correct code snippet in the similarity ranking.

We follow the method GraphCodeBERT [16] uses and combine the development and test sets from Table 4.1 to form the full test set T per language for a more realistic score. Additionally, we test our models on the AdvTest dataset.

4.3 Implementation details

We implemented our method in Python and used PyTorch[1] as a foundation for the Machine Learning models. To parallelize computation, we used PyTorch Lightning[2], which simplifies writing device-agnostic implementations. To obtain the pre-trained CodeBERT and GraphCodeBERT models and tokenizers, we used the Huggingface Transformers[3] library. All code is available on GitHub[4].

4.3.1 xMoCo

We implemented the basic modules of xMoCo in PyTorch Lightning. Imporantly, we noticed during initial experimentation that it is crucial to disable any regularization mechanisms in the slow encoders to improve scores (i.e. to put the encoders in evaluation mode). The queues are implemented as PyTorch Lightning register buffers, which are simply tensor matrices which we loop over by advancing a pointer after every update. For multi-GPU training, every GPU processes just a subset of the entire batch. PyTorch Lightning then automatically syncs gradients over all GPUs. To update the queue, we locally gather the embeddings from all GPUs and put them into the GPU-local queue copy to have a consistent queue over all GPU-threads.

[1] https://pytorch.org/
[2] https://www.pytorchlightning.ai/
[3] https://huggingface.co/
[4] https://github.com/jstuder3/nl-pl_moco

The embeddings produced by CodeBERT and GraphCodeBERT have dimension 768. We normalize these embeddings before storing them in the queue or using them for loss computation. Because we use shuffling, it is possible that samples in the queue are actually false negatives. We could optionally mask out these entries by setting their value in the similiarity matrix to -1, however, we found that detecting false negatives is as expensive as doing forward passes and has little impact on score, so we generally did not use this mechanism for the queue.

4.3.2 Hard negatives

Inspired by DyHardCode's approach, we used Facebook AI's Similarity Search (FAISS) library[5] for Python to quickly find nearest neighbours in the sense of cosine similarity. Before every epoch, we push the entire dataset through the slow encoders to generate a FAISS index for each modality. During training, we can then query this FAISS index by feeding it the fast encoder embeddings for which we want to find hard negatives. Based on DyHardCode's findings, we use the documentation embeddings to find code hard negatives for the documentation part and code embeddings to find documentation hard negatives for the code part. The FAISS index will return the indices of potential hard negatives. We then forward these through the slow encoder again to obtain an updated embedding. We use the slow encoder rather than the fast encoder in this step because during initial testing we found that this (surprisingly) benefits the validation score. A schematic of this is shown in Figure 4.1. As mentioned previously, the hard negatives can be put into their separate queue as well (not shown in this figure). Because there are relatively few hard negative samples compared to the size of the queue, it is feasible to filter out false negatives here by manually setting their similarity value to -1.

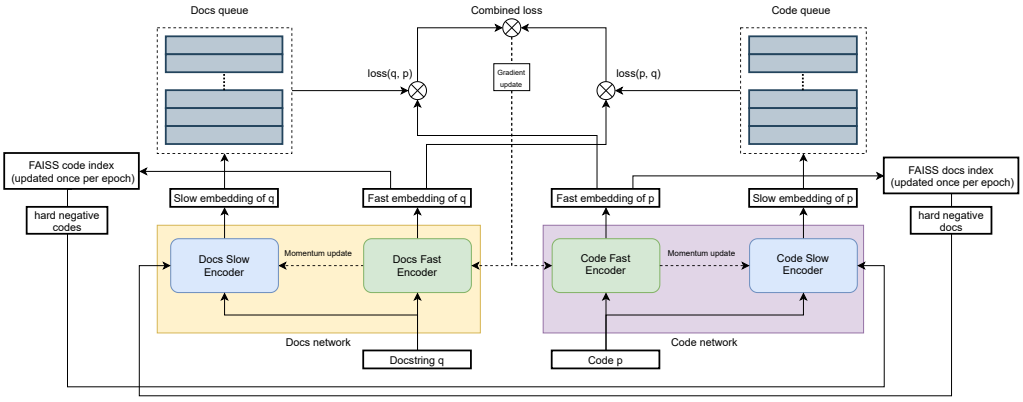

Figure 4.1: Schematic of xMoCo with hard negatives

[5] https://github.com/facebookresearch/faiss

4.3.3 Barlow regularization

Barlow Twins [7] uses tied projectors. However, since we focus on a multimodal problem, we implemented Barlow loss with the option of using either tied or separate projectors per modality for experimentation purposes. During training, the length-normalized embeddings produced by the fast encoder are first fed through these projectors, then standardized along the batch dimension and finally used to compute the cross-correlation matrix. While it would make sense to divide the values of this matrix by the batch size to enforce it to become truly uncorrelated for every sample, we found during initial testing that not dividing it like in the pseudocode provided in [7] actually produces better results. It should be noted, however, that this makes our results depend on the batch size and the number of GPUs used. Because of that, we kept the batch size and number of GPUs constant.

Figure 3.2 shows a schematic of how the Barlow loss can be incorporated. Notice, however, that it is also possible without further modification to use hard negatives in combination with Barlow loss, which this figure does not show.

4.3.4 Augmentation

For augmentation, we directly copied the EDA implementation by the paper's authors[6]. We did not change the default probability parameters of EDA and only changed necessary things about the code to make it work for our purposes.

When augmentation is enabled, the training set will be randomly augmented in one go before every epoch, rather than during the training itself. This is done for simplicity and to reduce the computational impact augmentation would have if we were to augment samples in a lazy manner.

4.4 Experiment hyperparameters and hardware

Unless otherwise noted, we used the hyperparameters listed in Table 4.2. In the following sections, we will refer to this configuration as "base config", with ablations listed in section 5.3. We used early stopping with a patience of 3 epochs on the MRR to reduce wasted computations and used the checkpoint with the best MRR over all epochs to produce the final test results.

The hardware used in all experiments were either 4 GeForce RTX 3090 or 4 Titan RTX GPUs and an AMD EPYC 7742 CPU. Computation times for the smallest dataset (Ruby) ranged from 1.5 hours to 6 hours depending on the number of hard negatives chosen. For the larger datasets (e.g. Python),

[6] https://github.com/jasonwei20/eda_nlp

Hyperparameter	Value
Batch size	32
Learning rate	1e-5
Docs encoder and Code encoder	CodeBERT
xMoCo momentum	0.999
Regular queue size	8192
Hard negative queue size	0
Number of hard negatives	0
Barlow lambda λ	0.005
Barlow weight β	0
Barlow projector dimension	0 (no projector)
EDA augmentation enabled	False

Table 4.2: Base hyperparameters used for the experiments

computation times could be as large as 40 hours until early stopping terminates execution.

Chapter 5

Results

5.1 Test results

We ran the tests on two different configurations. A separate model is trained for every language. If not stated otherwise, the models are configured as in Table 4.2. The first one is xMoCo with hard negatives, where we mine 8 hard negatives per positive sample. Secondly, we tested the combination of 8 hard negatives with Barlow regularization with $\beta = 5e - 5$ and no projector (i.e. we compute the cross-correlation matrix directly on the standardized embeddings). The final scores are shown in Table 5.1 and Table 5.2.

We can see here that xMoCo with hard negatives (with or without Barlow regularization) outperforms the previous State-of-the-Art average, with the Barlow variant improving by 1.9% relatively to DyHardCode with GraphCodeBERT and 10.2% over using just vanilla CodeBERT. xMoCo matches or outperforms previous methods in most languages. The largest difference can be observed in PHP, where the Barlow variant improves over the previous best score produced by SynCoBERT by 5.3% relatively.

Model/Method	**Ruby**	**JavaScript**	**Go**	**Python**	**Java**	**PHP**	**Overall**
CodeBERT	0.679	0.620	0.882	0.672	0.676	0.618	0.693
GraphCodeBERT	0.703	0.644	0.897	0.692	0.691	0.649	0.713
SynCoBERT	0.722	0.677	0.913	0.724	0.723	0.678	0.740
DyHardCode (CB)	0.715	0.666	0.917	0.713	0.729	0.636	0.729
DyHardCode (GCB)	0.740	0.687	0.921	**0.738**	0.738	0.677	0.750
xMoCo w/ h.n.	0.747	0.683	**0.929**	0.723	0.766	0.707	0.759
xMoCo w/ h.n. & barl.	**0.751**	**0.690**	0.924	0.733	**0.772**	**0.715**	**0.764**

Table 5.1: MRR test results on the combined validation/test set for the different subsets of the CodeSearchNet corpus. Results of prior works are taken from [6] and [18].

The results on the AdvTest benchmark show that our models perform slightly worse in this task than previous methods did. Interestingly, the Barlow variant

produced a lower score than the one without Barlow, despite the results from the CodeSearchNet benchmark showing the inverse effect.

Model/Method	AdvTest (Python)
CodeBERT	0.272
GraphCodeBERT	0.352
SynCoBERT	**0.381**
DyHardCode (CB)	0.378
DyHardCode (GCB)	-
xMoCo w/ h.n.	0.367
xMoCo w/ h.n. & barl.	0.361

Table 5.2: MRR test results the AdvTest benchmark. Results of prior works are taken from [6] and [18].

5.2 Visualizations

5.2.1 Latent space visualization

To see how the embeddings before and after training differ, we used a combination of Principal Component Analysis and t-SNE [19] (both implemented in the scikit-learn library[1]) to reduce the 768-dimensional embeddings to 2 dimensions. Figure 5.1 shows the results of a random subset of the embeddings for Ruby, with black lines connecting corresponding pairs. Before training (i.e. using non-finetuned CodeBERT), we observe that Documentation and Code form their own clusters. However, after training corresponding pairs are generally very close together. The method used to make this visualization is further described in Appendix A.

5.2.2 Cross-correlation matrix visualization

In order to get a better grasp on how the Barlow loss affects training, we logged the cross-correlation matrix during validation to observe how it changes over time. Figure 5.2 shows the cross-correlation matrix prior to training and after 3 epochs for a large Barlow weight $\beta = 0.5$ to make the effect more visible. The used projector was an identity function, which means that the produced cross-correlation matrices have shape 768x768, as 768 is the encoder output dimensionality. The observed result is not an identity matrix, but that is expected since Barlow is used as a regularization term and not the main loss. Additionally, our implementation slightly deviates from the exact specification as discussed in section 4.3.3.

[1] https://scikit-learn.org/stable/index.html

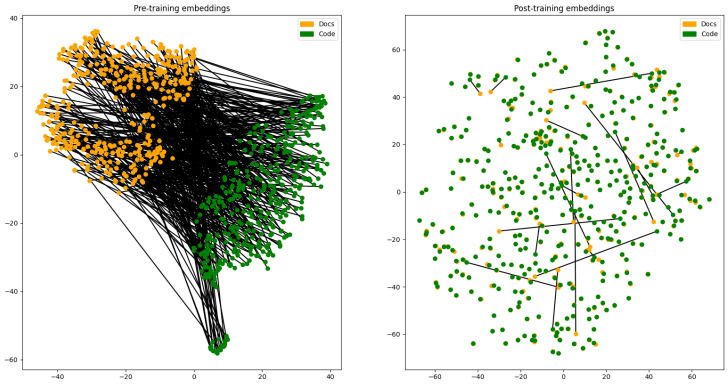

Figure 5.1: Dimensionality reduction for a subset of the Ruby corpus before (left) and after training (right)

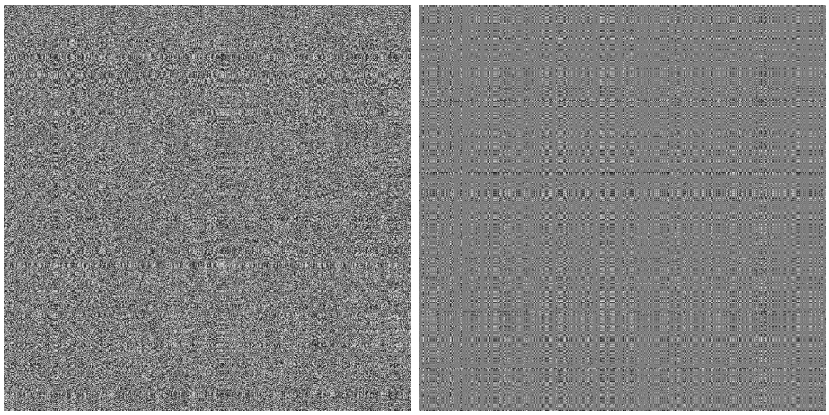

Figure 5.2: Aggregated cross-correlation matrix of the validation set prior to training (left) and after 3 epochs (right)

5.3 Ablations

Because the used methods introduce several new hyperparameters (or transfer prior methods to a new problem), we conducted ablations on the validation set to determine good hyperparameter combinations. These were later evaluated on the test set to form the results listed in section 5.1. The following chapter outlines these ablations. The experiments were all run using the same seed to ensure consistent results. All ablations were run only once and if not otherwise noted use the hyperparameters listed in Table 4.2 (also referred to as "base config"). Experiments with a "(∗)" in the caption were run by Zihan Zhang.

5.3.1 Queue size

Firstly, we conducted experiments using different queue sizes, as shown in Table 5.3. We find that performance increases first and drops off at too large sizes, which is roughly consistent with the findings in [5]. We find that a queue size of 8192 provides the best results.

	Regular queue size				
	0	2048	4096	8192	16384
Base config	0.7692	0.7921	0.7948	**0.7955**	0.7929

Table 5.3: Results for different queue sizes (∗)

5.3.2 Encoder choice

Since xMoCo offers the unique ability of using different models for the different modalities, we experimented with different combinations of CodeBERT and GraphCodeBERT, as shown in Table 5.4. We find that the combination of CodeBERT on both the Docstring and Code encoder provides by far the best result.

	Docs/Code encoder			
	CB/CB	CB/GCB	GCB/CB	GCB/GCB
Base config	**0.7955**	0.6997	0.6839	0.7438

Table 5.4: Results for the different encoder combinations (∗)

5.3.3 Hard negatives

Then, we investigated the effect that hard negatives have on our results, as well as how using a queue for the hard negatives affects scores, as shown in Table 5.5. We find that using the maximum possible number of hard negatives produces the best results and that using a hard negative queue in trade for a smaller regular queue does not provide any benefits over using just the regular queue. In fact, it seems this can in some cases even hurt performance.

5.3.4 Barlow parameters

We then investigated different choices for the Barlow weight β (without a projector) and hard negatives. We first conducted some informal experiments to determine a rough range of interesting values and then tested those. Notably, we found that if the Barlow weight is chosen too large, the MRR will collapse. As

		Regular/hard negative queue size				
		8192/0	6144/2048	4096/4096	2048/6144	0/8192
Hard neg.	0	0.7955	-	-	-	-
	2	0.7995	0.7974	0.7975	0.798	0.7997
	4	0.8008	0.8005	0.8023	0.8008	0.8006
	8	**0.8068**	0.804	0.8011	0.7986	0.7996

Table 5.5: Results for different combinations of number of hard negatives and queue sizes (∗)

seen in Table 5.6, Barlow regularization is fairly unstable, radically going from a good score to collapsing with just a small difference in weight. Nevertheless, it is capable of slightly improving scores over not using it at all.

		Barlow weight						
		1e-4	7e-5	5e-5	3e-5	1e-5	5e-6	0
H. n.	0	<0.1	0.8027	0.8032	0.7987	0.7965	0.7953	0.7955
	8	0.8027	0.8062	**0.8082**	0.8055	0.8054	0.805	0.8068

Table 5.6: Results for different combinations of number of hard negatives and Barlow weights. "<0.1" means that the validation score collapsed

Since we can also use projectors for Barlow, we also experimented with those, as shown in Table 5.7. We can have them either tied (i.e. the same projector is used for Docstrings and Code) or untied. We find that within the range of chosen weights, projectors do not offer any benefit. However, it is possible that with the choice of different weights, one could improve scores. We chose not to investigate further out of time restrictions. Here, too, we can observe that Barlow regularization is fairly unstable, with some scores collapsing below 0.1.

		Projector dimension				
		No projector	2048		4096	
		-	Tied	Untied	Tied	Untied
B. w.	1e-4	0.8027	<0.1	<0.1	<0.1	<0.1
	5e-5	**0.8082**	0.7363	0.7384	<0.1	<0.1
	1e-5	0.8054	0.801	0.7928	0.7581	0.7539

Table 5.7: Results for different combinations of Barlow weights, projector dimension and tying (all use 8 hard negatives); "<0.1" means that the validation score collapsed (∗)

5.3.5 Augmentation

Finally, we investigated whether simple augmentation of the Docstrings using EDA offers any benefits. We found that without Barlow regularization, the score slightly decreased and with Barlow, the results (surprisingly) remained unchanged, as shown in Table 5.8.

		No augmentation	EDA augmentation
B. w.	0	0.8068	0.8064
	5e-5	**0.8082**	**0.8082**

Table 5.8: Results for different combinations of Barlow weights and augmentation (∗)

Chapter 6

Conclusion and future directions

In this work, we transfer xMoCo, a split encoder framework for multimodal tasks, to an application called Code Search. xMoCo is different from previous methods that have been used for this task in that it uses one encoder per modality rather than a single, bimodal encoder. Additionally, it uses queues that hold embeddings from previous iterations. These embeddings can be used as negative samples in the contrastive learning objective. We enhanced xMoCo by incorporating hard negative mining to find particularly impactful samples that can be used as negatives in addition to the samples in the queue, and by introducing the idea of using the Barlow loss as a regularizer.

We are able to outperform previous methods used for Code Search, as indicated by our results for the CodeSearchNet benchmark. This benchmark consists of Documentation-Code pairs from 6 different programming languages. However, we are not able to outperform previous methods at the AdvTest benchmark. This indicates that - despite better scores in the CodeSearchNet benchmark - our method does not generalize better than previous methods.

The results from our ablations show that data augmentation of the Natural Language part of this problem using Easy Data Augmention does not improve scores.

Future directions for research could include transferring the Barlow regularization concept to different tasks, as it is a quite general idea with low computational overhead and has proven to be beneficial for our purposes. Despite EDA not having observably improved scores, it is still possible that data augmentation could be beneficial for real-world applications, where users might not use particularly formal language in their search queries. Data augmentation in the likes of Backtranslation might have a clearer impact on this. We also didn't explore any augmentation for Code due to the lack of availability of off-the-shelf tools. We imagine that using such a method could drastically improve scores for the AdvTest benchmark.

Bibliography

[1] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine *et al.*, "Scalability in perception for autonomous driving: Waymo open dataset," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 2446–2454.

[2] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," *CoRR*, vol. abs/2101.11149, 2021. [Online]. Available: https://arxiv.org/abs/2101.11149

[3] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[5] N. Yang, F. Wei, B. Jiao, D. Jiang, and L. Yang, "xmoco: Cross momentum contrastive learning for open-domain question answering," 2021.

[6] Anonymous, "Contrastive learning of natural language and code representations for semantic code search," 2021. [Online]. Available: https://openreview.net/forum?id=eiAkrltBTh4

[7] J. Zbontar, L. Jing, I. Misra, Y. LeCun, and S. Deny, "Barlow twins: Self-supervised learning via redundancy reduction," *arXiv preprint arXiv:2103.03230*, 2021.

[8] J. Wei and K. Zou, "Eda: Easy data augmentation techniques for boosting performance on text classification tasks," *arXiv preprint arXiv:1901.11196*, 2019.

[9] M. Gutmann and A. Hyvärinen, "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 297–304. [Online]. Available: https://proceedings.mlr.press/v9/gutmann10a.html

[10] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*. PMLR, 2020, pp. 1597–1607.

[11] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[12] X. Chen, H. Fan, R. B. Girshick, and K. He, "Improved baselines with momentum contrastive learning," *CoRR*, vol. abs/2003.04297, 2020. [Online]. Available: https://arxiv.org/abs/2003.04297

[13] X. Chen, S. Xie, and K. He, "An Empirical Study of Training Self-Supervised Vision Transformers," *arXiv e-prints*, p. arXiv:2104.02057, Apr. 2021.

[14] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *CoRR*, vol. abs/1702.08734, 2017. [Online]. Available: http://arxiv.org/abs/1702.08734

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

[16] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[17] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021. [Online]. Available: https://arxiv.org/abs/2102.04664

[18] X. Wang, F. M. Yasheng Wang, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation," 2022.

[19] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne." *Journal of machine learning research*, vol. 9, no. 11, 2008.

Appendix A

Interactive tools

A.1 Interactive code retriever

We wanted to have a more tangible result than just MRR numbers, so we implemented a code retriever that uses pre-trained models generated in this work. It's a simple User Interface implemented in Python's appJar[1] library. The user can enter a query in natural language, choose what programming language the retrieved code should stem from and set which subsets of the CodeSearchNet corpus should be used for retrieval (train/val/test). Additionally, the user can select how many of the top results should be displayed. An image of the interface is shown in Figure A.1 and a couple of example outputs are shown in Figure A.2.

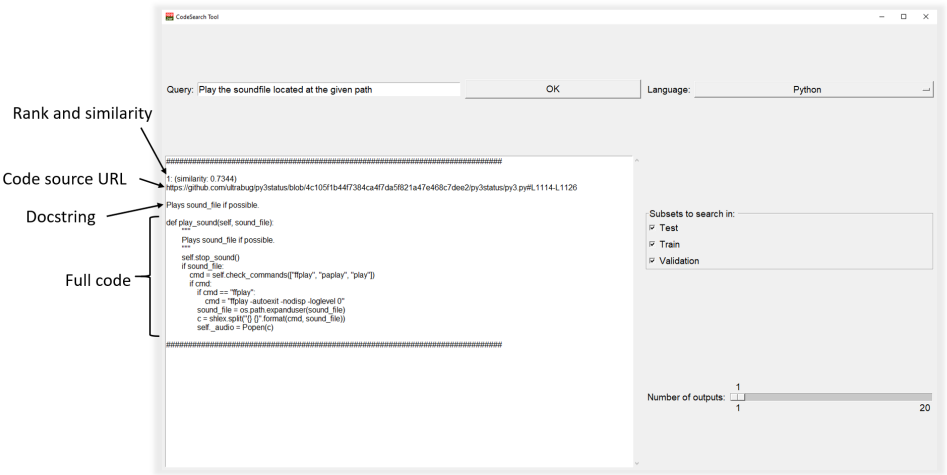

Figure A.1: The code retriever interface with a query and a retrieved code snippet (with annotations)

[1] http://appjar.info/

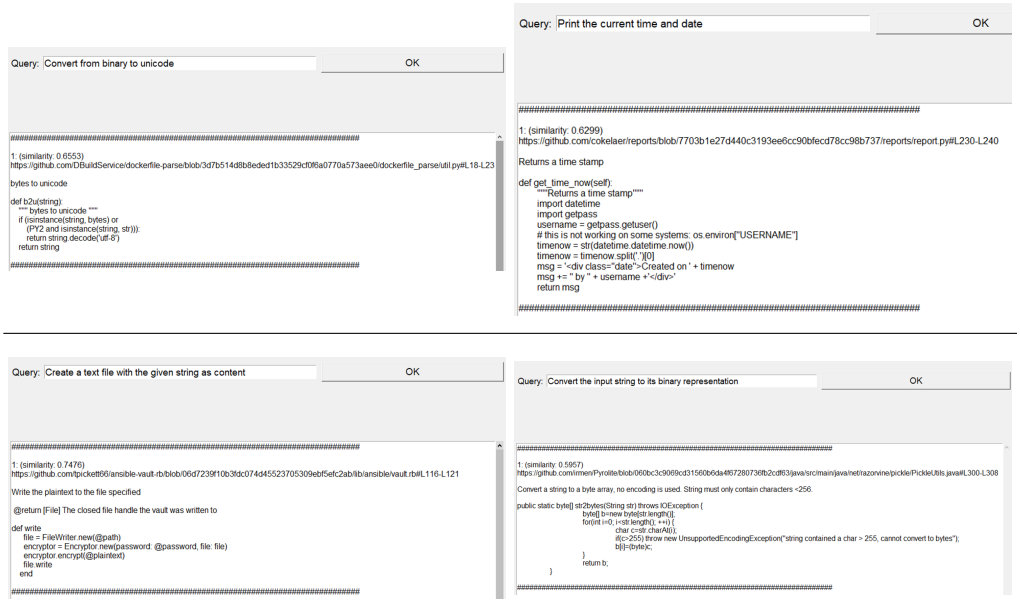

Figure A.2: Some samples of code retrieved for different queries in Python (top row) as well as Ruby and Java (bottom row)

Unfortunately, the tool requires the use of the (fairly large, 4GB each) checkpoints for the xMoCo models. Additionally, the entire code part of the CodeSearchNet corpus needs to be pre-encoded before use. However, once this is done and the model is loaded, code retrieval takes anywhere from 1 to 10 seconds depending on the subset size and hardware used, as only the natural language query has to be encoded and is then compared to the code embeddings. It is required to use a CUDA capable GPU with at least 2 GB of VRAM. For these reasons, the practical usefulness may be limited. A more optimal solution for when speed and expandability is desired would be a server-based code retriever, where queries are sent over the internet to a fast server that can almost instantaneously return search results. What's interesting is that in principle this can be used in an online-manner: If new code samples are added to the database, we can just encode them and add the generated vector to the set of embeddings. The runtime of retrieval will increase linearly in the number of samples in the embedding list, but with a larger dataset the quality of retrieved code snippets might improve.

A.2 Dimensionality reduction

To generate Figure 5.1, we combined two dimensionality reduction techniques. The first one is Principal Component Analysis which analytically finds the most important components of the input data. We used this to reduce the 768-

dimensional embeddings to 50 dimensions. From there, we used t-SNE, a clustering algorithm that iterates over the data and attempts to separate it. Because t-SNE's runtime and quality depend on the number of input dimensions and number of samples used, it is recommended to reduce high-dimensional data to a lower dimension before applying t-SNE for better results, just like we did with PCA. Note that t-SNE is not deterministic and that we randomly sample embeddings instead of using the full corpus, which is why results slightly differ from run to run, as shown in Figure A.3, however, the general clusters remain similar.

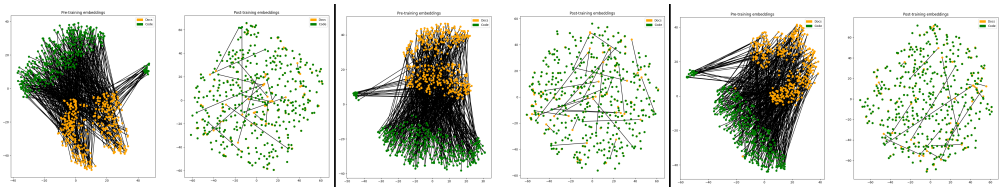

Figure A.3: Three independent runs of the dimensionality reduction method

It is also possible to inspect the data by clicking on individual data points. A pop-up will appear that shows which modality was used to generate the embedding, followed by the actual Docstring/Code. An example of this is shown in Figure A.4. Notice how for the concrete sample shown here, semantically related data (in this case the relation is through the concept of "deletion") is indeed mapped to a similar area, which was the premise of this entire work.

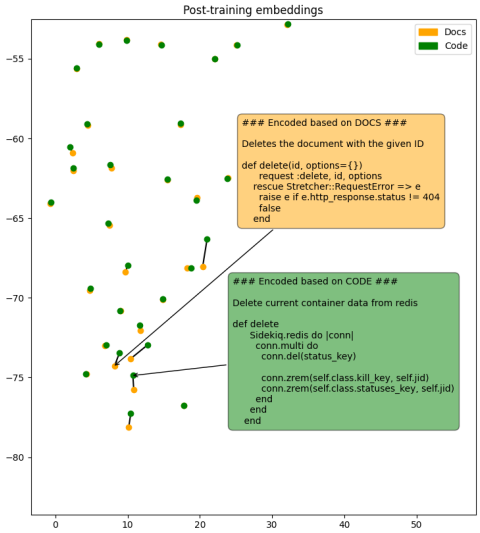

Figure A.4: A concrete example of how similar data is mapped close together