CASCADED FAST AND SLOW MODELS FOR EFFICIENT SEMANTIC CODE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

The goal of natural language semantic code search is to retrieve a semantically relevant code snippet from a fixed set of candidates using a natural language query. Existing approaches are neither effective nor efficient enough towards a practical semantic code search system. In this paper, we propose an efficient and accurate semantic code search framework with cascaded fast and slow models, in which a fast transformer encoder model is learned to optimize a scalable index for fast retrieval followed by learning a slow classification-based re-ranking model to improve the performance of the top K results from the fast retrieval. To further reduce the high memory cost of deploying two separate models in practice, we propose to jointly train the fast and slow model based on a single transformer encoder with shared parameters. The proposed cascaded approach is not only efficient and scalable, but also achieves state-of-the-art results with an average mean reciprocal ranking (MRR) score of 0.7795 (across 6 programming languages) as opposed to the previous state-of-the-art result of 0.713 MRR on the CodeSearchNet benchmark.

1 INTRODUCTION

Building tools that enhance software developer productivity has recently garnered a lot of attention in the deep learning research community. Parallel to the progress in natural language processing, pre-trained language models (LM) like CodeBERT (Feng et al., 2020b), CodeGPT (Lu et al., 2021), CodeX (Chen et al., 2021), PLBART (Ahmad et al., 2021) and CodeT5 (Wang et al., 2021b) have now been proposed for understanding and generation tasks involving programming languages.

Recent work on code generation like Chen et al. (2021)'s 12B parameter CodeX and Austin et al. (2021)'s 137B parameter LM use large scale autoregressive language models to demonstrate impressive capabilities of generating multiple lines of code from natural language descriptions, well beyond what previous generation models like GPT-C (Svyatkovskiy et al., 2020) could accomplish. However, this impressive performance is often predicated on being able to draw many samples from the model and machine-check them for correctness. This setup will often not be the case in practice. Code generation models also entail security implications (possibility of producing vulnerable or misaligned code) making their adoption tricky.

Given this current landscape, code retrieval systems can serve as attractive alternatives when building tools to assist developers. With efficient implementations, code search for a single query can typically be much faster for most practical index sizes than generating code with large scale LMs. As opposed to code generation, code retrieval offers the possibility of a much greater control over the quality of the result - the index entries can be verified beforehand. Leveraging additional data post training is easier when working with code search systems as this would simply require extending the index by encoding the new instances. Code search systems can be particularly of value for organizations with internal proprietary code. Indexing source code data internally for search can prevent redundancy and boost programmer productivity. A recent study by Xu et al. (2021) surveys developers to understand the effectiveness of code generation and code retrieval systems. Their results indicate that the two systems serve complementary roles and developers prefer retrieval modules over generation when working with complex functionalities, thus advocating the need for better code search systems.
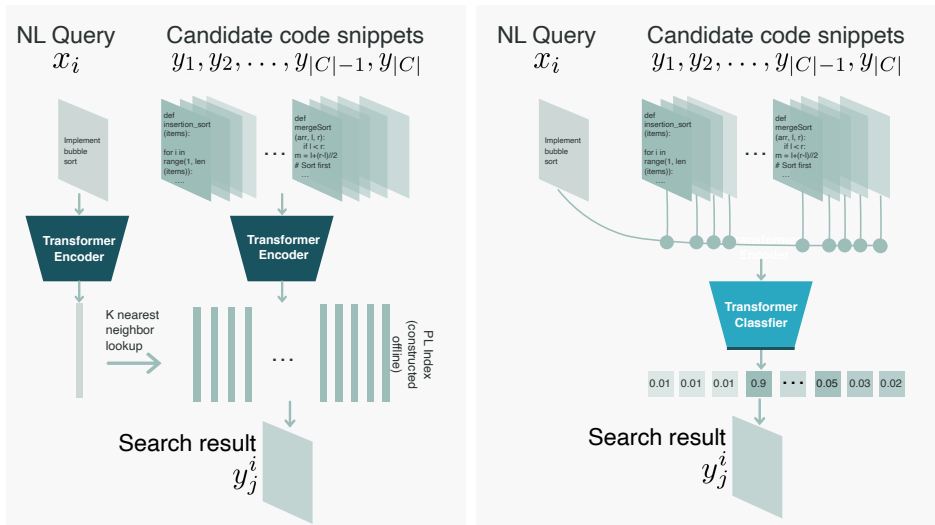
Figure 1: Illustration of the *fast encoder* (left) and *slow classifier* (right) based semantic code search approaches (at inference stage). With the encoder based approach, we independently compute representations of the NL query and candidate code sequences. The code snippet with representation nearest to the query vector is then returned as the search result. With the classifier based approach, we jointly process the query with each code sequence to predict the probability of the code matching the query description. The code sequence corresponding to the highest classifier confidence score is then returned as the search result.

Neural approaches to code search (Sachdev et al., 2018; Guo et al., 2021; Ye et al., 2016; Gu et al., 2018) involve encoding query and code independently into dense representations in the same semantic space. Retrieval is then performed using representational similarity (based on cosine or euclidean distances) of these dense vectors. An orthogonal approach involves encoding the query and the code jointly and training semantic code search systems as binary classifiers that predict whether a code answers a given query (Lu et al., 2021; Huang et al., 2021). With this approach, the model processes the query paired with each candidate code sequence. Intuitively, this approach helps in sharpening the cross information between query and code and is a better alternative for capturing matching relationships between the two modalities (natural language and programming language) than simple similarity metric between the encoder based sequence representations.
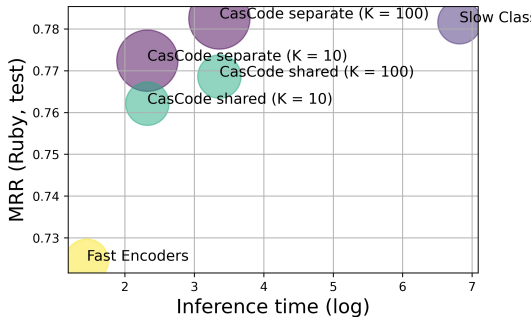

Figure 2: Overview of the speed versus performance trade-off of current code search approaches. Areas of the circles here are proportional to model sizes. With CasCode, we are able to achieve performance comparable to the optimal classifier based approach (top right), while requiring substantially lesser inference time.

While this latter approach can be promising for code retrieval, previous methods have mostly leveraged it for binary classification tasks involving NL-PL sequence pairs. Directly adapting this approach to code search tasks would be impractical due to the large number of candidates to be considered for each query. We depict the complementary nature of these approaches in Figure 1 when using a transformer (Vaswani et al., 2017) encoder based model for retrieval and classification.

In order to leverage the potential of such nuanced classifier models for the task of retrieval, we propose a cascaded scheme (CasCode) where we process a limited number of candidates with the classifier model. This limiting is performed by employing the encoder based approach and picking its top few candidate choices from the retrieval set for processing by the second classifier stage. Our cascaded approach leads to state of the art performance on the CodeSearchNet benchmark with an overall mean reciprocal ranking (MRR) score of 0.7795, substantially surpassing previous results. We propose a variant of the cascaded scheme with shared parameters, where a single transformer model can serve in both the modes - encoding and classification. This shared variant substantially reduces the memory requirements, while offering comparable retrieval performance with an MRR score of 0.7700.

Figure 2 illustrates the trade off involved between inference speed and MRR for different algorithmic choices, where we have the (*fast*) encoder model on one extreme, and the (*slow*) classifier model on the other. With CasCode, we offer performance comparable to the optimal scores attained by the classifier model, while requiring substantially lesser inference time, thus making it computationally feasible. Our codebase will be made publicly available for research purposes.

2 BACKGROUND

Early work on neural approaches to code search include Sachdev et al. (2018) who used unsupervised word embeddings to construct representations for documents (code snippets), followed by Cambronero et al. (2019)'s supervised approach leveraging the pairing of code and queries. Feng et al. (2020b) proposed pre-training BERT-style (Devlin et al., 2019) masked language models with unlabeled (and unpaired) source code and docstrings, and fine-tuning them for text-to-code retrieval task. With this approach, the query representation can be compared during inference against a pre-constructed index of code representations and the nearest instance is returned as the search result. Miech et al. (2021) and Li et al. (2021) have previously proposed similar approaches for text-to-visual retrieval.

Guo et al. (2021) leverage pairs of natural language and source code sequences to train text-to-code retrieval models. They adopt the contrastive learning framework (Chen et al., 2020) to train the retrieval model, where representations of natural language (NL) and programming language (PL) sequences that match in semantics (a positive pair from the bimodal dataset) are pulled together, while representations of negative pairs (randomly paired NL and PL sequences) are pushed apart. The infoNCE loss (a form of contrastive loss function (Gutmann & Hyvärinen, 2010)) used for this approach can be defined as follows:

$$\mathcal{L}_{\text{infoNCE}} = \frac{1}{N} \sum_{i=1}^{N} -\log \frac{\exp(f_\theta(x_i)^T f_\theta(y_i)/\sigma)}{\sum_{j \in B} \exp(f_\theta(x_i)^T f_\theta(y_j)/\sigma)} \tag{1}$$

where $f_\theta(x_i)$ is the dense representation for the NL input x_i, and y_i is the corresponding semantically equivalent PL sequence. N is the number of training examples in the bimodal dataset, σ is a temperature hyper-parameter, and B denotes the current training minibatch.

While the above approach applies for any model architecture, Guo et al. (2021) employ GraphCodeBERT (a structure-aware transformer encoder pre-trained on code) and CodeBERT for f_θ in their experiments. We refer to this approach as the one using *fast encoders* for retrieval. During inference, we are given a set of candidate code snippets $\mathcal{C} = \{y_1, y_2, \ldots y_{|\mathcal{C}|}\}$, which are encoded offline into an index $\{f_\theta(y_j) \ \forall j \in \mathcal{C}\}$. For a test NL query x_i, we then compute $f_\theta(x_i)$ and return the code snippet from $\mathcal{C}$ corresponding to the nearest neighbor (as per some distance metric e.g. cosine similarity) in the index. The rank r_i assigned to the correct code snippet (for the query x_i) from $\mathcal{C}$ is then used to compute the mean reciprocal ranking (MRR) metric $\frac{1}{N_{test}} \sum_{i=1}^{N_{test}} \frac{1}{r_i}$.

During inference, we are only required to perform the forward pass associated with $f_\theta(x_i)$ and the nearest neighbor lookup in the PL index, as the PL index itself can be constructed offline. This makes the approach very suitable for practical scenarios where the number of candidate code snippets $|\mathcal{C}|$ could be very large.

In a related line of work, Lu et al. (2021) propose a benchmark (NL-code-search-WebQuery) where natural language code search is framed as the problem of analysing a query-code pair to predict

whether the code answers the query or not. Huang et al. (2021) release a new dataset with manually written queries (as opposed to docstrings extracted automatically), and propose a similar benchmark based on binary classification of query-code pairs.

3   CASCODE

Although the approach proposed by Guo et al. (2021) is efficient for practical scenarios, the independent encodings of the query and the code make it less effective. We could instead encode the query and the code candidate jointly within a single transformer encoder and perform binary classification. In particular, the model could take as input the concatenation of NL and PL sequences $[x_i; y_j]$ and predict whether the two match in semantics.

The training batches for this binary classification setup can again be constructed using the bimodal dataset (positive pairs denoting semantic matches), and the negative pairs (mismatch) can be constructed artificially.   Given a set of paired NL-PL semantically equivalent sequences $\{x_i, y_i\}_{i=1}^{N}$, the cross-entropy objective function for this training scheme would be:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N} \sum_{i=1, j \neq i}^{N} \log p_\theta(x_i, y_i) + \log(1 - p_\theta(x_i, y_j)) \tag{2}$$

where $p_\theta(x_i, y_j)$ represents the probability that the NL sequence x_i semantically matches the PL sequence y_j, as predicted by the classifier. With a minibatch of positive pairs $\{x_i, y_i\} \ \forall i \in \mathcal{B}$, we can randomly pick y_j ($j \in \mathcal{B}; j \neq i$) from the PL sequences in the minibatch and pair it with x_i to serve as a negative pair. When using a transformer encoder based classifier, the interactions between the NL and PL tokens in the self-attention layers can help in improving the precision of this approach over the previous (independent encoding) one.

During inference, we can pair the NL sequence x_i with each of the y_j from $\mathcal{C}$ and rank the candidates as per the classifier's confidence scores of the pair being a match. This involves $\mathcal{C}$ forward passes (each on a joint NL-PL sequence, thus longer inputs than the previous approach), making this approach infeasible when dealing with large retrieval sets. We refer to this approach as the one using *slow classifiers* for retrieval. Figure 1 provides an illustration of these two different approaches.

We propose unifying the strengths of the two approaches - the speed of the *fast encoders* with the precision of the *slow classifiers*, with a cascaded scheme, called CasCode. Figure 3 shows the overall framework of our approach. Our hybrid strategy combines the strengths of the two approaches in the following manner - the first stage of *fast encoders* provides top-K candidates from the set $\mathcal{C}$ of candidate code snippets. In practice, the size of the retrieval set ($|\mathcal{C}|$) can often be very large, and varies from 4360 to 52660 for the CodeSearchNet datasets we study in our experiments.
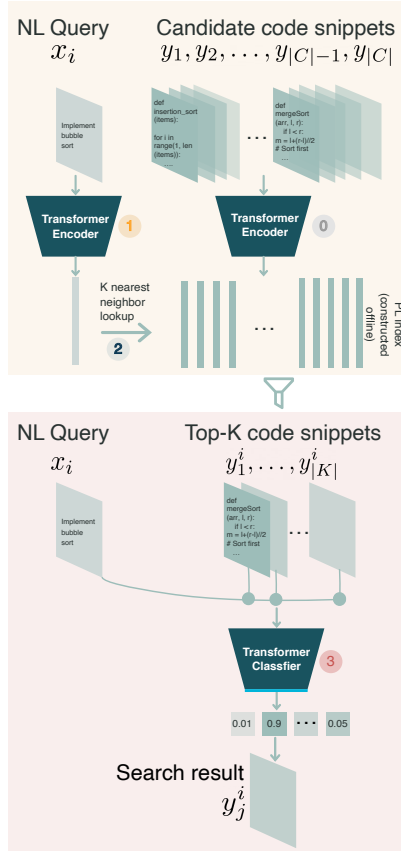

Figure 3: CasCode: Our proposed cascaded scheme for semantic code search. At the top, the transformer encoder independently processes the query x_i and the code snippets in the *fast* retrieval stage.   The top K candidates (based on the nearest neighbor lookup) from this stage are passed on to the second stage, where a transformer classifier jointly processes the query sequence with each of the filtered candidates to predict the probability of their semantics matching. The second stage classifiers are thus accelerated for the code retrieval task by the first stage of encoders.

```
Prompt the user to      def continue_prompt(message = ""):
continue or not             answer = False
                            message = message + """\n"Yes" or "No" to continue: """

                            while answer not in ("Yes", "No"):
                                answer = prompt ( message , eventloop = eventloop ())
                                if answer == "Yes":
                                    break
                                if answer == "No":
                                    break
                            return answer


Sends a message        def message(self, data):
to the framework            logging.info("""Driver sends framework
scheduler.                  message {}""". format(data))
                            return self.driver.sendFrameworkMessage(
                            data)
```

Table 1: Examples of bimodal pairs (natural language/docstring with corresponding code sequence) from CodeSearchNet (Python)

The top K candidates are then passed to the second stage of *slow classifiers* where each of them is paired with the NL input (query) x_i and fed to the model. For a given pair, this second stage classifier will return the probability of the NL and PL components of the input matching in semantics. Using these as confidence scores, the rankings of the K candidates are refined.

The resulting scheme is preferable for $K << |\mathcal{C}|$, as this would add a minor computational overhead on top of what is required by the *fast encoder* based retrieval. The second stage of refinement can then improve retrieval performance provided that the value of K is set such that the recall of the *fast encoder* is reasonably high. K would be a critical hyper-parameter in this scheme, as setting a very low K would lead to high likelihood of missing the correct snippet in the set of inputs passed to the second stage *slow classifier*, while a very high K would make the scheme infeasible for retrieval. As we discuss ahead in Section 4, CasCode with a K as small as 10 already offers significant gains in retrieval performance over the baselines, with marginal gains as we increment K to 100 and beyond.

In order to minimize the memory overhead incurred by the two stage model, we propose to share the weights of the transformer layers of the *fast encoders* and the *slow classifiers*. This can be achieved by training a model with the joint objective of infoNCE ($\mathcal{L}_{\text{infoNCE}}$) and binary cross-entropy ($\mathcal{L}_{\text{CE}}$.) While the number of parameters in this shared variant would be nearly half of the separat (non-shared) case, the computational cost at inference would be the same. Note that we would need some exclusive parameters for the classifier model, specifically the classification head (MLP) on top of the encoder. Thus, in this shared parameter variant of CasCode, the transformer model consuming the three kinds of inputs - NL only and PL only (for the *fast encoder* stage) and NL-PL (for the *slow classifier* stage) is identical except for the MLP layers in the second stage.

4 EXPERIMENTS

4.1 DATASET, BASELINES & METRICS

We use the CodeSearchNet code corpus from Husain et al. (2019) that includes six programming languages - Ruby, Javascript, Go, Python, Java and Php. Our pre-processing and train-val-test splits are identical to the setting from Guo et al. (2021)[1], who filter low-quality queries and expand the retrieval set to make the code search task more challenging and realistic. Table 1 shows 2 examples of bimodal pairs from the resulting dataset and the statistics of the dataset after pre-processing are provided in Table 2.

Our *fast encoder* baseline is based on the CodeBERT model from Feng et al. (2020b) that is pre-trained on programming languages. In order to have a strong baseline, we use a newer CodeBERT checkpoint that is pre-trained (using masked language modeling and replaced token detection tasks)

---

[1]https://github.com/microsoft/CodeBERT/tree/master/GraphCodeBERT

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

Table 2: Data statistics of the filtered CodeSearchNet corpus for Go, Java, Javascript, 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)

for longer, after we found that the CodeBERT checkpoint from Feng et al. (2020b) was not trained till convergence. When starting from our new checkpoint, we find that the CodeBERT baseline, if fine-tuned with a larger batch-size (largest possible that we can fit on 8 A100 GPUs) and for a larger number of epochs, is able to perform substantially better than the results reported before. We report the baselines from Guo et al. (2021) in Table 3 along with the results for our replication of two of these baselines. Previous studies have emphasized this effect - larger batch sizes are known to typically work well when training with the infoNCE loss in a contrastive learning framework, due to more negative samples from the batch (Chen et al., 2020).

We also train GraphCodeBERT, which is proposed by Guo et al. (2021) as a structure aware model pre-trained on programming languages. GraphCodeBERT leverages data flow graphs during pre-training to incorporate structural information into its representations. However, for the code search task, we report (Table 3) that GraphCodeBERT does not offer any significant improvements in performance over CodeBERT, when both variants are trained with a large batch size. For simplicity, we finetune the CodeBERT pre-trained model (architecturally equivalent to RoBERTa-baseLiu et al. (2019) model - 12 layers, 768 dimensional hidden states and 12 attention heads) and refer this as the *fast encoder* baseline for the remainder of our experiments.

MRR score for this CodeBERT baseline is shown in Table 3. For this baseline and the variants we propose, along with MRR, we also report Recall@K for $K = \{1, 2, 5, 8, 10\}$, that indicates the hit rate (ratio of instances where we find the correct output in the top K results). We encourage future work on code search to report these additional metrics, as these are important in evaluating the utility of a retrieval system and is commonly reported in similar work in text based image or video retrieval (Miech et al., 2021; Bain et al., 2021).

Figure 5 shows the Recall@K (K varied over the horizontal axis) for the 6 different programming languages, with the *fast encoder* models, over the validation set. As alluded to in Section 3, for designing the cascaded scheme, we need to pick a K that is large enough to provide reasonably high recall, and small enough for the second stage to be reasonably fast. We pick $K = 10$ and 100 where the recall for all 6 datasets is over 85% and 90% respectively.

4.2 RESULTS WITH CASCODE

We first show results for the ***slow classifier***s, trained using the CodeSearhNet datasets that we mention above. We finetune the CodeBERT pre-trained checkpoint (mentioned above) with a classification head (fully connected layers) for this task. On the validation set, we study the performance of this finetuned classifier for retrieval and report the MRR scores in Figure 4 for different values of K, where K is the number of top candidates passed from the first (*fast encoder*) stage to the second. Interestingly, the retrieval performance of this joint classifier fails to improve beyond certain values of K. For example, increasing K from 10 to 100 only marginally improves the MRR for Ruby, Javascript and Java, while for other languages there is no significant improvement beyond $K = 10$. Further training details for CasCode variants and the *fast encoder* baselines are provided in Appendix A.

Next, we train *fast* and *slow* models with **shared parameters**, denoted by CasCode (shared). The training objective for this model is the average of the binary cross-entropy loss $\mathcal{L}_{\text{CE}}$ and the infoNCE loss $\mathcal{L}_{\text{infoNCE}}$ as described in Section 3. The MRR scores for the baselines and our separate and shared variants are listed in Table 3. With our cascaded approach, we observe

Model/Method	Ruby	Javascript	Go	Python	Java	Php	Overall
NBow	0.162	0.157	0.330	0.161	0.171	0.152	0.189
CNN	0.276	0.224	0.680	0.242	0.263	0.260	0.324
BiRNN	0.213	0.193	0.688	0.290	0.304	0.338	0.338
selfAtt	0.275	0.287	0.723	0.398	0.404	0.426	0.419
As reported by Guo et al. (2021)							
RoBERTa	0.587	0.517	0.850	0.587	0.599	0.560	0.617
RoBERTa (code)	0.628	0.562	0.859	0.610	0.620	0.579	0.643
CodeBERT	0.679	0.620	0.882	0.672	0.676	0.618	0.693
GraphCodeBERT	0.703	0.644	0.897	0.692	0.691	0.649	0.713
As reported by Wang et al. (2021a)							
SYNCOBERT	0.722	0.677	0.913	0.724	0.723	0.678	0.740
Replicated with a larger training batch-size							
CodeBERT	0.7245	0.6794	0.9145	0.7305	0.7317	0.681	0.7436
GraphCodeBERT	0.7253	0.6722	0.9157	0.7288	0.7275	0.6835	0.7422
Ours (K=10)							
CasCode (shared)	0.7621	0.6948	0.9193	0.7529	0.7528	0.7001	0.7637
CasCode (separate)	0.7724	0.7087	0.9258	0.7645	0.7623	0.7028	0.7727
Ours (K=100)							
CasCode (shared)	0.7686	0.6989	0.9232	0.7618	0.7602	0.7074	0.77
CasCode (separate)	**0.7825**	**0.716**	**0.9272**	**0.7704**	**0.7723**	**0.7083**	**0.7795**

Table 3: Mean Reciprocal Ranking (MRR) values of different methods on the codesearch task on 6 Programming Languages from the CodeSearchNet corpus (test set). The first set consists of four finetuning-based baseline methods (NBow: Bag of words, CNN: convolutional neural network, BiRNN: bidirectional recurrent neural network, and multi-head attention), followed by the second set of models that are pre-trained then finetuned for code search (RoBERTa: pre-trained on text by Liu et al. (2019), RoBERTa (code): RoBERTa pre-trained only on code, CodeBERT: pre-trained on code-text pairs by Feng et al. (2020a), GraphCodeBERT: pre-trained using structure-aware tasks by Guo et al. (2021)). SYNCOBERT: pre-trained using syntax-aware tasks by Wang et al. (2021a). In the last four rows, we report the results with the shared and separate variants of our CasCode scheme using the fine-tuned CodeBERT models for K of 10 and 100.
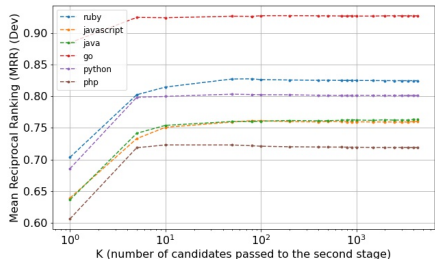

Figure 4: Mean reciprocal ranking (MRR) at different values of K over the validation set of CodeSearchNet Husain et al. (2019) when using a finetuned CodeBERT (*slow*) binary classifier (match or not) for text-code retrieval.
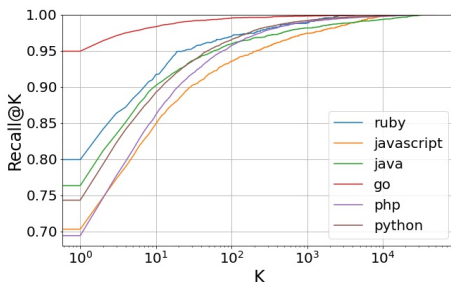

Figure 5: Recall at different values of K over the validation set of CodeSearchNet Husain et al. (2019) when using a finetuned CodeBERT encoder (*fast*) for text-code retrieval.

significant improvements over the *fast encoder* baselines, the overall MRR[2] averaged over the

---

[2]we report MRR on the scale of 0-1, some works (eg. Wang et al. (2021a)) use the scale 0-100

six programming langues for CasCode (separate) is 0.7795, whereas the *fast encoder* baseline (CodeBERT) reaches 0.7422. The improvements with CasCode are noticeably greater over the baseline for Ruby, Javascript, Python and Java. We report modest improvements on the Go dataset, where the *fast encoder* baseline is already quite strong (0.9145 MRR).

The shared variant of CasCode attains an overall MRR score of 0.77, which is comparable to the separate variant performance. This slight difference can be attributed to the limited model capacity in the shared case, as the same set of transformer layers serve in the encoder and classifier models. We also evaluate the MRR scores for the CasCode (shared) model in the *fast encoder* stage where the test set MRR scores were $0.7308, 0.6634, 0.9048, 0.7193, 0.7244, 0.6803$ for Ruby, Javascript, Go, Python, Java and PHP respectively, with the overall MRR being 0.7372. We note in passing, that the cascaded model that was trained in a multi-task manner, gives competitive retrieval performance, even when used only in its first (encoder only) stage.

We also report the Recall@K metric for CasCode separate and CasCode shared variants in Figure 6. For all six programming languages, we observe improvements over the *fast encoder* baseline with our cascaded scheme. Similar to our observation from Table 3, the shared variant of CasCode is slightly worse than the separate one.

Retrieval speed comparison: Having established the improvements in retrieval performance with CasCode, we proceed to analyze the trade-off between inference speed and performance, for the different methods discussed. For each variant, we record the time duration required to process (obtain a relevant code snippet from the retrieval set) 100 natural language queries from the held-out set. We use the Ruby dataset of CodeSearchNet for this analysis, which contains 4360 candidate code snippets for each NL query. We conduct this study on a single Nvidia A100 GPU. Our results are shown in Table 4.

For the *fast encoder* approach (using infoNCE-finetuned CodeBERT), we first incur some computational cost to encode all the candidate code snippets and construct the PL index (6.76 seconds for Ruby's retrieval set). This computation is common to all approaches, except the *slow* (binary, joint) classifier one. Since this computation can be performed offline before the model is deployed to serve user queries, we do not include this cost in our results in Table 4. With the PL index constructed beforehand, we report the time required to encode a user NL query, and perform nearest neighbor lookup on the PL index with the encoding, in the first row of Table 4. This computation is again performed by all the CasCode variants, and thus acts as the lower bound on time taken by CasCode for retrieval. For the analysis to be as close to real world scenarios as possible, we do not batch the queries and encode them one by one. Batching them would require assuming that we have the NL queries beforehand, while we would be receiving them on the fly from users when deployed.

With the *slow classifier* approach, we would pair a given query with each of the 4360 candidates, and thus this would lead to the slowest inference of all the variants. For all variants of CasCode, the inference duration listed in Table 4 includes the time taken by the *fast encoder* based retrieval (first stage). For CasCode's second stage, we can pass the K combinations (query concatenated with each of the top-K candidate from the fast stage) in a batched manner. The shared variant, while requiring half the parameters, incurs the same computational cost when used in the cascaded fashion. We note from Table 4 that at a minor drop in the MRR score, lowering CasCode's K from 100 can lead to almost 3x faster inference for the shared case.

5 CONCLUSION & FUTURE WORK

We propose CasCode, which is a cascaded scheme consisting of transformer encoder and joint binary classifier stages for the task of semantic code search and achieve state of the art performance on the CodeSearchNet benchmark, with significant improvements over previous results. We also propose a shared parameter variant of CasCode, where a single transformer encoder can operate in the two different stages when trained in a multi-task fashion. At almost half the number of parameters, CasCode's shared variant offers comparable performance to the non-shared (separate) variant.

A limitation of our current cascaded scheme is that the computation spent in generating representations in the first stage of *fast encoders* is not leveraged in the second stage. We process raw token level inputs in the second stage. Ideally the representations designed in the first stage should

Model	# params	Inference duration (s)	MRR	# queries/s
Fast encoders (CodeBERT)	125M	4.27	0.7245	23.42
Slow Binary classifiers	125M + 0.5M	914.86	0.7816	0.11
CasCode (separate, K=100)	250M + 0.5M	28.83	0.7825	3.46
CasCode (shared, K=100)	125M + 0.5M	29.56	0.7686	3.38
CasCode (separate, K=10)	250M + 0.5M	10.22	0.7724	9.78
CasCode (shared, K=10)	125M + 0.5M	13.07	0.7621	7.65

Table 4: Inference speed comparison for the different variants studied. The number of parameters corresponding to the classifier head are separated with a + sign in the second column. Inference duration is reported for 100 queries from the Ruby subset of CodeSearchNet, using a single A100 GPU. Constructing the PL index offline requires 6.76 seconds for the Ruby dataset and is not included in the durations listed here. MRR scores are reported on the entire test set. Throughput of the retrieval model (measured in # queries processed per second) is listed in the last column.
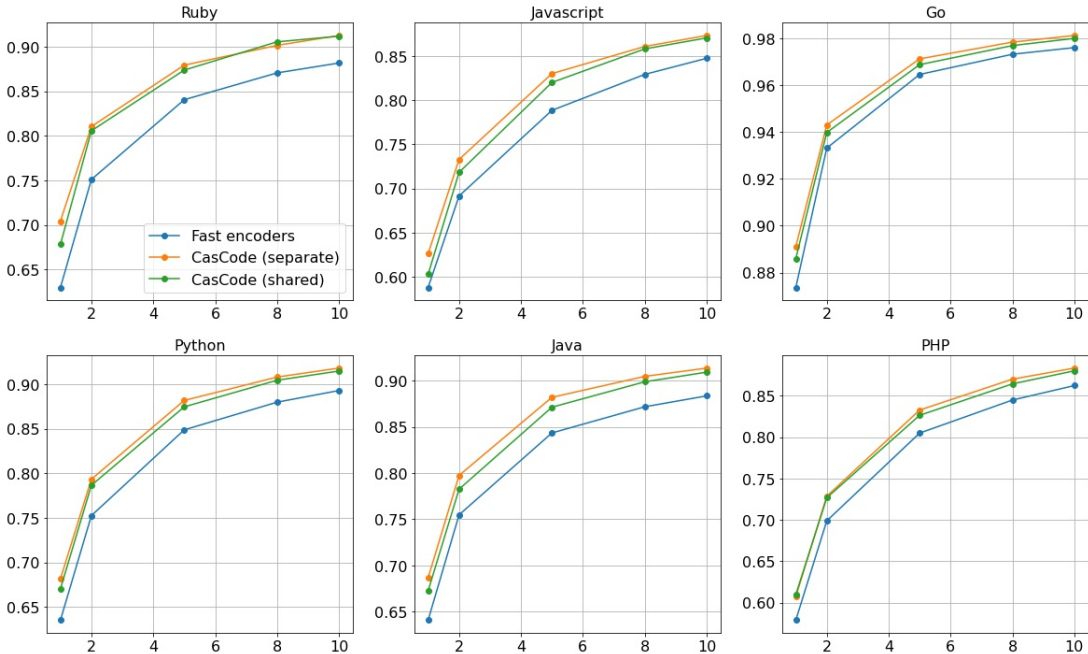

Figure 6: Recall @ K = $\{1, 2, 5, 8, 10\}$ with the *fast encoder* and CasCode (shared and separate) methods on the test set queries of CodeSearchNet dataset.

be useful for the classification stage too (Li et al., 2021). Our initial attempts along this direction did not turn fruitful, and future work could address this aspect. Another limitation warranting further investigation is associated with the training of the shared variant of CasCode. Here, training with the multitask learning framework (joint objective of infoNCE and binary cross entropy) leads to a model that performs slightly worse than the separate variant (individually finetuned models). We tried augmenting the capabilites of this model with solutions like using independent CLS tokens for the three modes the model has to operate in (NL only, PL only, NL-PL concatenation), and adjusting the relative weight of the two losses involved, but could not achieve any improvement over the separate variant.

REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, Online, June 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.naacl-main.211. URL `https://aclanthology.org/2021.naacl-main.211`.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Max Bain, Arsha Nagrani, Gül Varol, and Andrew Zisserman. Frozen in time: A joint video and image encoder for end-to-end retrieval. In *IEEE International Conference on Computer Vision*, 2021.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 964–974, 2019.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pp. 1597–1607. PMLR, 2020.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL `https://aclanthology.org/N19-1423`.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020a. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://www.aclweb.org/anthology/2020.findings-emnlp.139`.

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

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944. IEEE, 2018.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *ICLR 2021*, 2021.

Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 297–304. JMLR Workshop and Conference Proceedings, 2010.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. Cosqa: 20,000+ web queries for code search and question answering. *arXiv preprint arXiv:2105.13239*, 2021.

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

Junnan Li, Ramprasaath R. Selvaraju, Akhilesh Deepak Gotmare, Shafiq Joty, Caiming Xiong, and Steven Hoi. Align before fuse: Vision and language representation learning with momentum distillation. In *NeurIPS*, 2021.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Antoine Miech, Jean-Baptiste Alayrac, Ivan Laptev, Josef Sivic, and Andrew Zisserman. Thinking fast and slow: Efficient text-to-visual retrieval with transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9826–9836, 2021.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 31–41, 2018.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1433–1443, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Xin Wang, Fei Mi Yasheng Wang, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. 2021a.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. 2021b.

Frank F Xu, Bogdan Vasilescu, and Graham Neubig. In-ide code generation from natural language: Promise and challenges. *arXiv preprint arXiv:2101.11149*, 2021.

Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th international conference on software engineering*, pp. 404–415, 2016.

A APPENDIX

Training details: We begin with the baseline implementation of GraphCodeBERT (publicly available) and adapt their codebase to also implement the CodeBERT model. For the cascaded schemes, many of our training design decisions are therefore the same as GraphCodeBERT.

We use 8 A100 GPUs (each with 40 GB RAM) to train our baselines and CasCode variants. During training, we set the batch-size to a value that occupies as much available GPU RAM as possible. This happens to be 576 for the CodeBERT and GraphCodeBERT baseline finetuning with the infoNCE loss (fast encoders). For training the joint NL-PL classifier of CasCode (separate), we use a batch size of 216. For CasCode (shared), we need to further reduce the batch size to 160. All models are trained for 100 epochs.

For all our experiments we use a learning rate of 2e-5. We use the Adam optimizer to update model parameters and perform early stopping on the development set. For the CasCode variants, when performing evaluation on the development set, we use $K = 100$ candidates from the fast encoder stage. Using 8 A100 GPUs, rough training durations for CasCode on the Ruby, Javascript, Go, Python, Java and PHP datasets are 6.5, 8, 33, 41, 15 and 21 hours respectively with separate variant and 13, 17, 38, 42, 55.5, 56 hours with the sharef variant. We typically perform evaluation on the validation set once every epoch, but make it infrequent in some cases to speed up training for larger datasets like Python and PHP. Given the significant amount of computation invested in training these retrieval models, we plan to release these checkpoints to avoid wasteful redundant training and encourage future work on semantic code search.