One Model to Train them All: Hierarchical Self-Distillation for Enhanced Layer Representations in Embeddings

Anonymous ACL submission

Abstract

Deploying language models often requires handling model size vs. performance trade-offs to satisfy downstream latency constraints while preserving the model's usefulness. Model distillation is commonly employed to reduce model size while maintaining acceptable performance. However, distillation can be inefficient since it involves multiple training steps. In this work, we introduce MODULARSTARENCODER, a modular multi-exit encoder with 1B parameters, useful for multiple tasks within the scope of code retrieval. MODULARSTARENCODER is trained with a novel self-distillation mechanism that significantly improves lower-layer representations—allowing different portions of the model to be used while still maintaining a good trade-off in terms of performance. Our architecture focuses on enhancing text-to-code and code-to-code search by systematically capturing syntactic and semantic structures across multiple levels of representation. Specific encoder layers are targeted as exit heads, allowing higher layers to guide earlier layers during training. This self-distillation effect improves intermediate representations, increasing retrieval recall at no extra training cost. In addition to the multi-exit scheme, our approach integrates a repository-level contextual loss that maximally utilizes the training context window, further enhancing the learned representations. We also release a new dataset constructed via code translation, seamlessly expanding traditional text-to-code benchmarks with code-to-code pairs across diverse programming languages. Experimental results highlight the benefits of self-distillation through multi-exit supervision.

1 Introduction

Large language models (LLMs) have significantly impacted the field of natural language processing, demonstrating remarkable performance across various applications (Niu et al., 2023). However, the amount of computation required to operate state-of-the-art models poses significant challenges for the large-scale deployment of these models.

To mitigate these challenges, the research community has explored several model strategies to reduce the operational cost of LLMs without sacrificing their effectiveness. A prominent technique in model compression is quantization (Jacob et al., 2017; Lin et al., 2023; Egiazarian et al., 2024), which involves the reduction of numerical precision in the model's parameters. Quantization effectively decreases memory requirements and enhances inference speed, facilitating the deployment of large language models in resource-constrained environments. Concurrently, knowledge distillation has emerged as a powerful technique whereby a smaller "student" model is trained to emulate the behavior of a larger "teacher" model, as evidenced by works such as DISTILBERT (Sanh et al., 2019) and TINYBERT (Jiao et al., 2019). Additionally, pruning methods selectively eliminate less influential weights or neurons, further reducing model complexity and aiming to preserve performance (Han et al., 2015).

Recent efforts have increasingly focused on developing efficient architectures requiring fewer parameters. Model families such as LLaMA (Dubey et al., 2024), Qwen (Hui et al., 2024), Mistral (Jiang et al., 2023), and SmolLM (Allal et al., 2025) exemplify a paradigm shift towards smaller, more accessible architectures. These model families are deployed at various resolutions—ranging from lightweight variants optimized for heavily resource-constrained environments to larger versions that retain competitive performance.

In parallel, advancements in dynamic inference strategies have introduced mechanisms that further optimize computational efficiency. Techniques like multi-exit networks enable early predictions at intermediate layers, reducing unnecessary computations. For instance, early-exit architectures

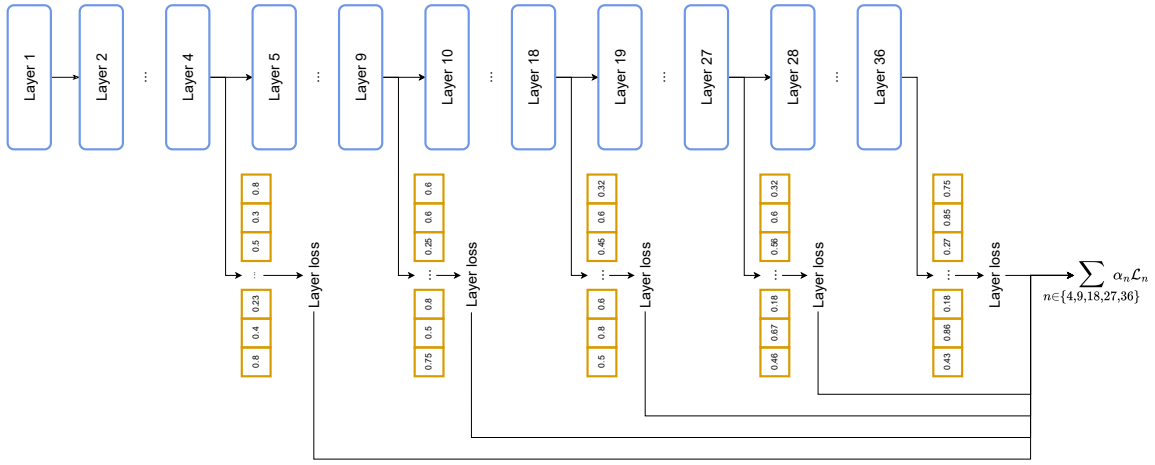


Figure 1: Overview of our multi-exit self-distillation encoder, shown here with exit heads at selected layers (e.g., Layers 4, 9, 18, 27, and 36). Each exit head predicts an output embedding and adds a "layer loss," contribution weighted by a coefficient α_i, summed into the overall objective $\mathcal{L}$.

such as BranchyNet (Teerapittayanon et al., 2017) dynamically balance computation and accuracy by allowing predictions before full model execution. Similarly, Matryoshka representation learning (Kusupati et al., 2022) extends this idea to embeddings, introducing a loss function that yields multi-granular representations. This approach allows downstream tasks to adjust computational complexity by pruning embedding dimensionality, further contributing to efficient model deployment.

Building on these principles, we propose MODULARSTARENCODER, a modular multi-exit encoder architecture that integrates a novel intra-model self-distillation mechanism. In our design, specific intermediate layers are supervised by both the primary task loss and auxiliary distillation losses on specific exit heads, encouraging lower layers to learn better representations by mimicking the outputs of higher layers. We apply a shared embedding head comprising a masked language modeling head and an in-context classification head across a chosen subset of layers. We then fine-tuned the model with different projection heads for each exit point. We reached state-of-the-art results on multiple retrieval tasks (such as code-to-code and text-to-code), fine-tuning *one* single modular model that can be sliced depending on the end-user computational constraints.

Our contributions are as follows:

- We introduce a self-distillation framework that enables training multiple model resolutions within a unified layer stack, reducing redundancy and improving scalability. We believe this approach can significantly affect LLM training pipelines that depend on multiple model distillations.
- We train and plan to release MODULARSTARENCODER, which consists of a pre-trained and fine-tuned encoder: The former is a modular pre-trained encoder with up to 1 billion parameters and four exit points, allowing users to perform multiple exit fine-tuning depending on downstream tasks. The latter is a fine-tuned encoder for various retrieval tasks, enabling the end user to select the model size that meets their memory and computational constraints.
- We plan to release SYNTHCODE2CODE2NL a new dataset constructed via code translation, expanding popular text-to-code datasets across diverse programming languages with code-to-code pairs. SYNTHCODE2CODE2NL comprises 1.071.367 triplets of natural language-code-code.

2 Methodology

2.1 Dataset

In the pre-training phase, we leveraged The Stack V2 dataset (Lozhkov et al., 2024), a large open-source code repository. Since many samples exceed typical context lengths, we split any tokenized sample that surpassed 1536 tokens — 75% of our 2048 token limit — while preserving each repository's original structure. This strategy ensures that our model sees manageable code segments.

Table 1: SYNTHCODE2CODE2NL details: Average character count and sample size per language for the CodeSearchNet dataset and the synthesized portion obtained through translation.

Language	CSN samples	CSN avg. char	Synth. samples	Synth. avg. char
English	1.071.367	180	-	-
PHP	280.706	514	116.967	579
Python	274.454	474	117.374	518
Go	234.089	350	124.125	541
Java	282.118	505	116.098	707
C++	-	-	141.956	938
Ruby	-	-	158.494	456
C	-	-	136.365	1029
JavaScript	-	-	159.988	557

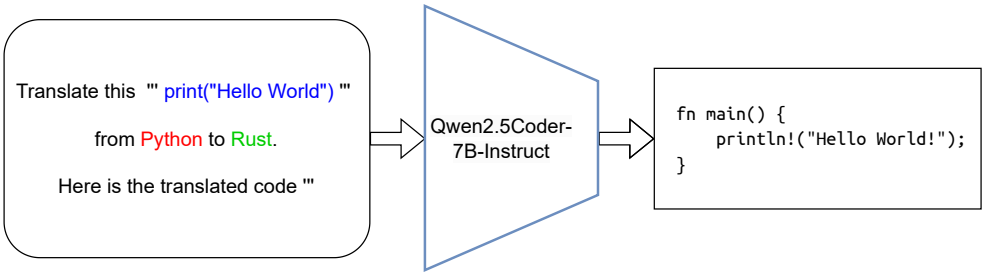


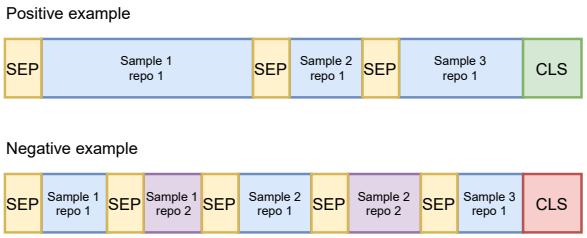
Figure 2: Prompt provided to Qwen2.5-Coder-7B-Instruct for translating a given code snippet (`print("Hello World")` in the example) from a source programming language (`Python`) to a target one (`Rust`).

We created SYNTHCODE2CODE2NL, a dataset for fine-tuning that supports text-to-code and code-to-code search. Using the popular CODESEARCHNET (Husain et al., 2019) as a seed dataset and selecting popular programming languages (Python, Java, Go, and PHP), we augmented it by transpiling available code snippets onto other languages.

During the preprocessing phase of SYNTHCODE2CODE2NL, we deduplicated the dataset using both the CodeSearchNet code column and the synthesized code column. After a manual inspection, we discovered that both columns contained code snippets that differed only in identifiers or function arguments, as several tasks were semantically identical but paraphrased with different parameter requirements (e.g., two identical paraphrased tasks were asking for opening a socket on a different port). During the data near deduplication phase, we relied on Locality Sensitive Hashing (LSH) with a Jaccard similarity threshold of 0.7 and 256 permutations, analyzing character-level 5-grams.

To generate semantically similar code snippets for code-to-code search, we translated each snippet into a different language randomly sampled from Go, Ruby, Python, Java, C++, PHP, C, JavaScript. We prompted the QWEN2.5-CODER-7B-INSTRUCT model with the source code, the name of the source language, and the name of the target language (see fig. 2). During code translation, we choose the token with the highest probability as output (greedy search) to prevent semantic discrepancies.

This process yielded pairs of code snippets in distinct languages tied to the same natural language description. As a result, every sample in the fine-tuning dataset includes a natural language description and two code snippets from distinct languages. SYNTHCODE2CODE2NL contains 1.071.367 samples where in the first code column we directly processed code snippets from CodeSearchNet, including Python, Java, PHP, and Go. The third column, artificially synthesized via code translation, includes Go, Ruby, JavaScript, Python, C++, PHP, C, and Java code snippets. Motivated by preliminary experiments indicating that near-deduplication on the code columns improved model performance, we then near-deduplicated the code columns of the datasets. Table 1 shows the average number of characters per language in SYNTHCODE2CODE2NL, we emphasize that synthesized data is significantly longer than human-written code. Appendix A provides examples of code translation.




```
def build_examples(repo_1, repo_2):
    is_negative = random(0, 1)
    input = empty_string

    while not empty(repo_1):
        input += <sep_token>

        # Positive case
        pos_sample = sample_random_snippet(repo_1)
        if len(input) + len(pos_sample) > context_length - 1:
            break
        input += pos_sample

        if is_negative and not empty(repo_2):
            # Negative (snippets from different repos)
            input += <sep_token>
            neg_sample = sample_random_snippet(repo_2)
            if len(input) + len(neg_sample) > context_length - 1:
                break
            input += neg_sample
        input += <cls_token>

    return input
```

Figure 3: On the left side the illustration of the in-context loss framework, where samples from different repositories are concatenated. Positive examples share the same repository context, while negative examples come from different repositories. On the right side, in-context loss framework pseudocode.

2.2 Architecture

We built MODULARSTARENCODER on top of STARCODER-2 (Lozhkov et al., 2024), reducing its size from 15B to 1B parameters in `bfloat16`. Our architecture consists of 36 hidden layers, each with 16 attention heads and 4 key-value heads, using Grouped Query Attention (GQA) (Ainslie et al., 2023). The model employs Rotary Positional Encoding (RoPE) (Su et al., 2021) with a base period $\theta = 10^{-6}$ and features a hidden dimensionality of 1024 with an intermediate size of 12,288.

To enhance efficiency, we replaced the causal self-attention layers with bidirectional self-attention, similar to recent work that "modernized" BERT (Warner et al., 2024). Unlike STARCODER-2, which uses sliding window attention, we opted for full attention to ensure greater modularity, avoiding the receptive field constraints of sliding window mechanisms (Zhu et al., 2021). Additionally, we extended the maximum input length to 2048 tokens, accommodating longer code snippets compared to prior code encoders such as STARENCODER (Li et al., 2023).

Finally, our implementation integrates FLASHATTENTION V2 (Dao, 2023) for faster inference. Table 2 summarizes the architectural details.

2.3 Pre-training

We pre-trained MODULARSTARENCODER with a batch size of 3.99M tokens for 245,000 training steps, processing 1T tokens. The pre-training and fine-tuning were conducted on 512 NVIDIA Ampere (64GB) GPUs using the Leonardo supercomputer (Turisini et al., 2023), requiring 450,000 GPU working hours.

To enable both token-level and snippet-level embeddings after pre-training, we employed a multi-objective pre-training strategy that combined two losses, as detailed in section 2.3.1 and section 2.3.2. The pre-training was performed on THESTACKV2, whose context length analysis revealed an average of ≈ 630 tokens per code snippet. As described in section 2.3.1, we concatenated multiple snippets to facilitate our multi-loss methodology, allowing our in-context classification loss to expand the average context window to $\approx$ 1300 tokens, reaching the maximum context length 20

We optimized the model using the AdamW optimizer with β_1 set to 0.9, β_2 to 0.95, ϵ to 1e-6, and a weight decay of 1e-1. The learning rate was initialized at 6.24e-4 and scheduled using a multi-step learning rate decay (Bi et al., 2024) with 4,000 warmup steps. The learning rate was reduced at 120,000, 185,000, 220,000, 230,000, and 240,000 training steps, applying a decay factor of 0.36, and from step 185,000 onward, further reduced by factors of 0.1, 0.031, 0.01, and 0.001. Table 2 summarizes the hyperparameters for architecture, pre-training, and fine-tuning.

2.3.1 Masked Language Modeling and In Context Classification

The training objectives of BERT (Feng et al., 2020), specifically Masked Language Modeling (MLM) and Next Sentence Prediction (NSP), have become a de facto standard. However, The NSP loss constrains the context window length to the sentence length, leading to too many padding tokens and redundant computation(Zeng et al., 2022), and has been shown to not yield significant benefits after fine-tuning (Warner et al., 2024; Aroca-Ouellette

Table 2: Hyperparameters for Architecture, Pre-training, and Fine-tuning

Architecture	
Hyperparameter	**Value**
Model size	1B parameters
Precision	bfloat16
Hidden layers	36
Attention heads	16
Hidden dimensionality	1024
Positional encoding	RoPE ($\theta = 10^{-6}$)
Context length	2048
Attention mechanism	Grouped-Query Attention
Attention pattern	Bi-directional
Pre-training	
Batch size	3.99M tokens
Pretraining steps	245.000
Pretraining Tokens	1T
Loss function	MLM + In-Context loss
Multi-layer loss	yes
Optimizer	AdamW
Weight decay	1e-1
Initial learning rate	6.24e-4
Learning rate schedule	Multi-step
Warmup steps	4000
Fine-tuning	
Dataset size	635.404 samples
Fine-tuning steps	20.000
Loss function	CLIP loss
Multi-layer loss	yes
Batch size	2048
Learning rate	1.0e-5
Temperature parameter	10.0
Hardware (Pre-training + fine-tuning)	
GPUs	512 NVIDIA Ampere (64GB)
Overall Training hours	450.000

and Rudzicz, 2020). Given that the average number of tokens per data sample in Stack v2 is 630, a large context window of 2048 results in substantial padding, making long-context training inefficient. While Wang et al. (2023) demonstrated the advantages of training LLMs with multiple objectives, we revisited the NSP loss and introduced an in-context classification (ICC) objective. We hypothesize that predicting whether multiple code snippets belong to the same context (in our case, the same repository) can enhance semantic search performance while allowing efficient concatenation of multiple code fragments. Our final training objective is the summation of two losses: (1) MLM loss and (2) ICC loss: $\mathcal{L} = \mathcal{L}_{MLM} + \mathcal{L}_{ICC}$.

In $\mathcal{L}_{MLM}$, a certain percentage of tokens are randomly masked and predicted using a classification head. Following Zhang et al. (2024), we adopt a 15% masking rate with the standard 80-10-10 token replacement strategy (Feng et al., 2020). The secondary objective, $\mathcal{L}_{ICC}$, determines whether randomly concatenated inputs (separated by a $< SEP >$ token) originate from the same repository (see fig. 3). Each concatenated sample has a 50% probability of containing source code from different repositories. This approach increases input density—reducing padding by expanding the average input length from 630 to 1300 tokens—and potentially enhances cross-language understanding. Since repositories are inherently modular and often contain files written in multiple languages, learning from repository-level context may improve inter-language generalization.

2.3.2 Multi-layer Loss

To achieve layer-wise modularity in transformer architectures, we apply the previously introduced loss (section 2.3.1) across a selected set of layers, sharing classification heads (masked language modeling and in-context classification) while incorporating a positional embedding of the layer index. The total loss is computed as the sum of individual layer losses, weighted by a factor α to prioritize deeper layers: $\mathcal{L} = \sum_{i \in \iota} \mathcal{L}_i \cdot \alpha$ where $\alpha = i/|I|$ and $I = \{1, \ldots, 36\}$ represents all layers, and the selected subset $\iota = \{4, 9, 18, 27, 36\}$ defines the layers where the loss is applied. The selected subset was chosen to enable four model variants equally spaced in depth (9, 18, 27, 36) along with an additional "tiny" version (4) to see the model performance in a lower number of parameters set. This approach allows for flexible model deployment, enabling adaptive layer pruning while maintaining performance trade-offs.

2.4 Fine-tuning

Following (Su et al., 2023), we fine-tune a single model for both text-to-code and code-to-code retrieval using instruction prompting. The optimization objective combines CLIP loss (Radford et al., 2021) with a multi-layer loss (details in 2.3.2).

To enhance representation learning, we replace the standard single-head projection with five distinct projection heads, applied at different exit points of the pre-trained model (layers 4, 9, 18, 27, and 36). We used a batch of 2048 elements, ensuring that text-to-code and code-to-code were equally distributed across the batch.

We performed data augmentation by randomly replacing frequently occurring words (appearing more than twice and having at least three characters) with random strings. We applied the augmentation exclusively to code snippets in 30% of cases,

Table 3: Performance of different models on text-to-code with CodeSearchNet using **codeXGLUE**. We reported the results presented in codet5plus, unixcoder and modernBERT (Wang et al., 2023; Guo et al., 2022; Warner et al., 2024).

	CodeSearchNet							
Model	Ruby	JS	Go	Python	Java	PHP	avg. MRR	avg. NDCG
MODULARSTARENCODER	74.1	**74.0**	82.5	**92.5**	**78.7**	**84.5**	**81.0**	**84.2**
Codet5+ 770M	78.0	71.3	92.7	75.8	76.2	70.1	77.4	-
OpenAI text-embedding-3-large	84.7	85.3	95.9	99.8	90.1	95.6	91.9	93.3
Unixcoder	74.0	68.4	91.5	72.0	72.6	67.6	74.4	-
ModernBERT-large	-	-	-	-	-	-	-	59.5

leaving natural language descriptions unchanged. After conducting a grid search, we selected $1.0e-5$ as the learning rate, maintained throughout the fine-tuning process, and set the temperature parameter at 10.0.

2.5 Evaluation

We evaluated MODULARSTARENCODER on both text-to-code and code-to-code retrieval tasks using the CODEXGLUE benchmark (Lu et al., 2021). For text-to-code retrieval, we employed the CODESEARCHNET dataset, where the goal is to retrieve the most relevant code snippet given a natural language query. Specifically, the query corresponds to a documentation comment, and the model is tasked with ranking the correct code snippet among 999 distractor snippets (Husain et al., 2019). This setup assesses the model's ability to learn meaningful cross-modal representations between code and natural language.

For code-to-code retrieval, we relied on two datasets from CODEXGLUE: the Code Translation (CT) benchmark and POJ-104. The Code Translation dataset consists of semantically equivalent code snippets in different programming languages, and we framed the task as cross-language code retrieval rather than translation. In this setting, given a Java code snippet as a query, the model retrieves the corresponding C# implementation, testing its capability to capture cross-lingual semantic similarities between functionally equivalent programs.

In contrast, the POJ-104 dataset contains C/C++ programs that solve the same problem but with different implementations, making it a suitable benchmark for assessing retrieval of semantically similar code across diverse implementations. This setup evaluates the model's capacity to generalize across structural variations while preserving semantic equivalence.

Table 4: Performance of different models on Code Translation (CT) and POJ104 for code-to-code search with codeXGLUE dataset.

	CT	**POJ104**
Model	MRR	mAP
MODULARSTARENCODER	**98.9**	56.5
Codet5+ 110M-embedding	98.4	24.5
OpenAI text-embedding-3-large	98.8	**82.9**
Unixcoder	97.6	41.0
ModernBERT-large	93.1	27.3

3 Results and Discussion

3.1 Benchmarks

Table 3 presents the results for CodeSearchNet (t2c) task in terms of Mean Reciprocal Rank (MRR) for each single language, average NDCG and average MRR. Results for Unixcoder, ModernBERT, and CodeT5+ are reported from the original papers (Guo et al., 2022; Warner et al., 2024; Wang et al., 2023). On CODESEARCHNET, MODULARSTARENCODER achieves an MRR of 81.0, outperforming CODET5+ (Wang et al., 2023) (770M), UNIXCODER (Guo et al., 2022), and MODERNBERT-LARGE (Warner et al., 2024). Although MODERNBERT-LARGE reports only average NDCG results (59.5). The only encoder that surpasses MODULARSTARENCODER is OpenAI's text-embedding-3-large.

Table 4 presents results from both POJ104 and CT dataset reported respectively in MRR for code translation and mean average precision for POJ104. For Code Translation (CT), we report MRR on Java-to-C# retrieval. MODULARSTARENCODER reaches the best performance among the tests. We decided to replicate the experiments in a zero-shot setting for code-to-code tasks, as our model does not integrate POJ104 and code translation datasets in the training set.

Referring to Table 4, on the POJ104 dataset in

Table 5: Performance comparison of MODULARSTARENCODER model layers and baseline fine-tuned models on the CodeSearchNet benchmark. The table displays the overall retrieval performance measured by Mean Reciprocal Rank (MRR). We refer to the model fine-tuned with multiple exit points simultaneously as *self-distilled*; the others are fine-tuned singularly for each exit point.

		CodeSearchNet						
Model	Size	Ruby	Javascript	Go	Python	Java	PHP	avg. MRR
Layer-4	110M	59.5	61.3	72.1	86.2	68.2	75.5	70.5
Layer-4 (self-distilled)		**62.2**	**64.7**	**74.8**	**88.1**	**71.4**	**78.0**	**73.2**
Layer-9	250M	64.9	65.7	74.3	87.3	72.0	78.8	73.8
Layer-9 (self-distilled)		**67.6**	**69.4**	**78.9**	**90.2**	**75.5**	**82.3**	**77.3**
Layer-18	500M	73.8	73.5	82.4	92.1	78.4	84.0	80.7
Layer-18 (self-distilled)		**74.1**	**74.0**	**82.5**	**92.5**	**78.7**	**84.5**	**81.0**
Layer-27	750M	72.3	71.8	80.8	90.8	76.9	82.3	79.1
Layer-27 (self-distilled)		**73.2**	**73.3**	**81.7**	**92.1**	**77.8**	**83.8**	**80.3**
Layer-36	1B	72.3	**72.9**	**80.7**	**91.5**	**77.1**	**82.9**	79.5
Layer-36 (self-distilled)		**73.5**	72.6	80.5	91.4	76.9	82.7	**79.6**

zero-shot, MODULARSTARENCODER achieves an mAP of 0.57, which is state-of-art between open-sourced models, even though it is significantly behind OpenAI text-embedding-3-large. However, direct comparison with OpenAI text-embedding-3-large remains challenging because it is closed-source, and details such as model size, training methodology, or potential data contamination are undisclosed.

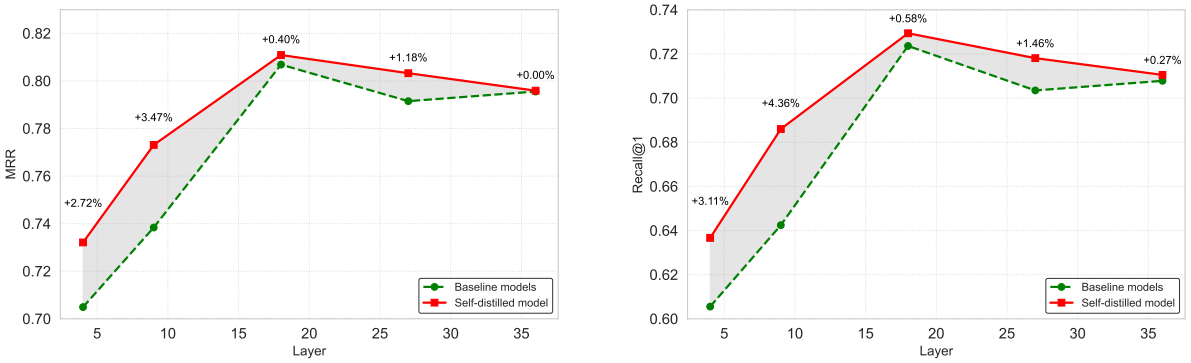
3.2 Ablation Study

We conducted an ablation study by fine-tuning singularly each exit point (also starting from MODULARSTARENCODER, pre-trained) and pruning the subsequent layers (e.g., for the baseline on layer 18, we retain only the first 18 layers and fine-tune the model using just one projection head on that layer). The multi-exit model consistently outperforms the single-exit baseline, indicating that *lower-level layers benefit from training signals propagated from deeper layers*. This behavior is highlighted in Table 5, where our model, indicated by *self-distilled*, outperforms all the single exit baselines consistently. This finding underscores a promising new direction in self-distillation for large-scale code and text models, enabling high performance even in more compact configurations. Moreover, Figure 4 illustrates that MODULARSTARENCODER maintains robust performance from layers 18 to 36, allowing users to scale down the network to match their memory, computational, or latency constraints while preserving strong retrieval accuracy.

4 Related work

Since the introduction of ELMo (Peters et al., 2018), deep contextual information has enhanced generating embeddings for textual retrieval or classification, reaching state-of-the-art results in several tasks. BERT (Feng et al., 2020) followed those findings, adapting the Transformer architecture (Vaswani et al., 2017) to enable a bi-directional representation with two different training objecting, namely the masked language modeling and the next sentence prediction losses. (Lan et al., 2019; Liu et al., 2019) adapted the BERT architecture to obtain an enhanced pre-trained model by removing or modifying the NSP, focusing on pre-training data or hyperparameters optimization. More recently, modernBERT (Warner et al., 2024) tied the gap between modern decoders (Jiang et al., 2023; Hui et al., 2024; Dubey et al., 2024; Touvron et al., 2023; Lozhkov et al., 2024) advancements that rely upon models with an increased number of parameters, trained upon more tokens, and being capable of handling longer contextual information.

In code representation, large language models must be adapted by training them on a curated corpus focused on software and by leveraging code's syntactic and semantic structures, which differ significantly from natural language. Feng et al. (2020) adapted the BERT architecture to produce semantically meaningful embeddings for source code, resulting in codeBERT. This was accomplished by including more source code in the training set and focusing on a training loss that can leverage bimodal (natural language and code) contextual in-



(a) MRR (b) Recall@1

Figure 4: Performance Comparison Across Layers: The graph illustrates the MRR and the Recall@1 for different layers, comparing baseline models and a self-distilled model.

formation (Clark et al., 2020). GraphCodeBERT enhanced codeBERT (Feng et al., 2020) representations by incorporating data flow graphs, capturing dependencies between variables and operations, and improving tasks like code summarization and clone detection. UniXcoder (Guo et al., 2022) extended this by introducing a unified encoder-decoder framework, integrating abstract syntax trees (ASTs) and data flow information. Wang et al. (2023) expanded these findings with codet5plus, stressing how multiple losses that leverage code semantics impact the model pertaining. The work incorporated text-code contrastive learning, text-code Matching, and text-code causal LM for better code understanding and generation.

When trying to achieve better performance, research has shifted toward models with a high number of parameters. While this trend appears effective from a performance perspective, end users may face computational or memory limitations as LLMs vary from millions to billions of parameters. Sanh et al. (2019) pioneered the introduction of knowledge distillation, using a "teacher" model that guides a smaller model to emulate its behavior. This methodology has been widely adopted and improved upon recently (DeepSeek-AI et al., 2025; Hui et al., 2024), becoming a standard for obtaining high-performing smaller LLMs.

Our work differs from previous work by adapting a modern architecture (Lozhkov et al., 2024) to a code encoder-only based model and introducing a novel 'self-distillation' mechanism. We replace the next sentence prediction loss with an in-context classification focused on the repository level and expand the context to 2048 tokens. Our novel self-distillation mechanism improves low-level layers, resulting in a modular transformer architecture without additional teacher models or further data for distillation.

5 Conclusion

In this work, we introduced MODULARSTARENCODER, a modular multi-exit encoder architecture designed to improve efficiency and scalability in code retrieval tasks. By integrating an intra-model self-distillation mechanism, our approach enables multiple resolution models to be trained within a unified layer stack, reducing redundancy while maintaining high retrieval performance. Our evaluation on CODEXGLUE demonstrates that MODULARSTARENCODER achieves state-of-the-art results among open-source models, outperforming prior baselines across text-to-code and code-to-code retrieval tasks. Ablations further highlighted the benefits of self-distillation, showing that lower layers gain representational strength from deeper layers, leading to superior performance compared to single-exit models.

Beyond performance gains, MODULARSTARENCODER offers practical benefits by providing multiple exit points, allowing users to balance computational efficiency and accuracy based on resource constraints. The results suggest that self-distillation provides a promising direction for efficient large-scale encoders, reducing deployment costs without sacrificing effectiveness.

Finally, we plan to release in open-access our SYNTHCODE2CODE2NL and both pre-trained and fine-tuned MODULARSTARENCODER models.

Limitations

Due to our dependence on multiple GPUs, we encountered significant computational constraints. Parameter grid searches with smaller and embryonic models were the only ways to extrapolate the best hyperparameter setup. The best hyperparameters for smaller models can differ from those for larger ones; thus, we faced a limitation in finding an optimal training setup. Ablating both the in-context classification and the multi-layer loss in a real scenario was impossible as we depended on smaller models to understand their performances. Therefore, computational resources pose a significant constraint in this work, and we want to emphasize how this factor undermines the possibility of replicating the experiments.

Here, we highlight potential threats to the validity of the research process, focusing on both external and internal factors.

External validity When synthesizing the SYNTHCODE2CODE2NL code, we rely on code translation; we understand that synthesized data adheres to stylistic writing patterns distinct from those of humans. We tested the model's performance on standard benchmarks. However, the impact of utilizing code snippets as synthetic data in training large language models for generalization over human text-to-code and code-to-code search is still not fully understood.

Internal validity The ablation study focused on fine-tuning the model with and without multi-layer loss. However, this comparison does not account for how the model behaves when starting from a model not pre-trained on multi-layer loss. Although our experiments present promising results, further inspection is necessary to better understand this phenomenon.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. *arXiv e-prints*, arXiv:2305.13245.

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, et al. 2025. Smollm2: When smol goes big–data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*.

Stéphane Aroca-Ouellette and Frank Rudzicz. 2020. On losses for modern language models. *arXiv preprint arXiv:2010.01694*.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, Alex X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. 2024. Deepseek LLM: scaling open-source language models with longtermism. *CoRR*, abs/2401.02954.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: pre-training text encoders as discriminators rather than generators. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Tri Dao. 2023. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. *arXiv e-prints*, arXiv:2307.08691.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng

Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *Preprint*, arXiv:2501.12948.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. 2024. The llama 3 herd of models. *CoRR*, abs/2407.21783.

Vage Egiazarian, Andrei Panferov, Denis Kuznedelev, Elias Frantar, Artem Babenko, and Dan Alistarh. 2024. Extreme Compression of Large Language Models via Additive Quantization. *arXiv e-prints*, arXiv:2401.06118.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *arXiv e-prints*, arXiv:2002.08155.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 7212–7225. Association for Computational Linguistics.

Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. *arXiv e-prints*, arXiv:1506.02626.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. 2024. Qwen2.5-coder technical report. *CoRR*, abs/2409.12186.

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

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *arXiv e-prints*, arXiv:1712.05877.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. *arXiv e-prints*, arXiv:2310.06825.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2019.

TinyBERT: Distilling BERT for Natural Language Understanding. *arXiv e-prints*, arXiv:1909.10351.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. 2022. Matryoshka representation learning. In *Advances in Neural Information Processing Systems*, volume 35, pages 30233–30249. Curran Associates, Inc.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2019. ALBERT: A Lite BERT for Self-supervised Learning of Language Representations. *arXiv e-prints*, arXiv:1909.11942.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. StarCoder: may the source be with you! *arXiv e-prints*, arXiv:2305.06161.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2023. AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration. *arXiv e-prints*, arXiv:2306.00978.

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

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv e-prints*, arXiv:2402.19173.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An empirical comparison of pre-trained models of source code. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2136–2148. IEEE.

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana. Association for Computational Linguistics.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. 2021. Learning transferable visual models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv e-prints*, arXiv:1910.01108.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. 2023. One embedder, any task: Instruction-finetuned text embeddings. In *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 1102–1121. Association for Computational Linguistics.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2021. RoFormer: Enhanced Transformer with Rotary Position Embedding. *arXiv e-prints*, arXiv:2104.09864.

Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2017. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks. *arXiv e-prints*, arXiv:1709.01686.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton-Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurélien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288.

Matteo Turisini, Giorgio Amati, and Mirko Cestari. 2023. LEONARDO: A pan-european pre-exascale supercomputer for HPC and AI applications. *CoRR*, abs/2307.16885.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *arXiv e-prints*, arXiv:1706.03762.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 1069–1088. Association for Computational Linguistics.

Benjamin Warner, Antoine Chaffin, Benjamin Clavié, Orion Weller, Oskar Hallström, Said Taghadouini, Alexis Gallagher, Raja Biswas, Faisal Ladhak, Tom Aarsen, Nathan Cooper, Griffin Adams, Jeremy Howard, and Iacopo Poli. 2024. Smarter, better, faster, longer: A modern bidirectional encoder for fast, memory efficient, and long context finetuning and inference. *CoRR*, abs/2412.13663.

Jinle Zeng, Min Li, Zhihua Wu, Jiaqi Liu, Yuang Liu, Dianhai Yu, and Yanjun Ma. 2022. Boosting distributed training performance of the unpadded BERT model. *CoRR*, abs/2208.08124.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. 2024. Code Representation Learning At Scale. *arXiv e-prints*, arXiv:2402.01935.

Chen Zhu, Wei Ping, Chaowei Xiao, Mohammad Shoeybi, Tom Goldstein, Anima Anandkumar, and Bryan Catanzaro. 2021. Long-short transformer: Efficient transformers for language and vision. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 17723–17736.

A Synthetic dataset

SYNTHCODE2CODE2NL is a fine-tuning dataset designed for text-to-code and code-to-code search, built by augmenting CODESEARCHNET (Husain et al., 2019) with transpiled code snippets across multiple languages (Python, Java, Go, PHP, Ruby, C++, C, JavaScript). The dataset underwent a preprocessing phase, including deduplication based on the original and synthesized code columns. Near-deduplication was performed using Locality Sensitive Hashing (LSH) with a Jaccard similarity threshold of 0.7 over character-level 5-grams to remove semantically identical snippets differing only in identifiers or function arguments.

For code-to-code search, we translated each snippet into a randomly sampled target language using the QWEN2.5-CODER-7B-INSTRUCT model with greedy search to ensure consistency. Each dataset entry consists of a natural language description and two code snippets in different languages. SYNTHCODE2CODE2NL contains 1,071,367 samples, with original code from CODESEARCHNET (Python, Java, PHP, Go) and translated code (Go, Ruby, JavaScript, Python, C++, PHP, C, Java). In Figure 5, In Figure 6 and Figure 7 some examples of code translation are shown.

GO

```go
// readNextChunk creates an io.LimitedReader for the next chunk of data,
// and returns io.EOF if the last chunk has been read.
func readNextChunk(r wire.SyncScanner) (io.Reader, error) {
    status, err := r.ReadStatus("read-chunk")
    if err != nil {
        if wire.IsAdbServerErrorMatching(err, readFileNotFoundPredicate) {
            return nil, errors.Errorf(errors.FileNoExistError, "no such file or directory")
        }
        return nil, err
    }

    switch status {
    case wire.StatusSyncData:
        return r.ReadBytes()
    case wire.StatusSyncDone:
        return nil, io.EOF
    default:
        return nil, errors.Errorf(
            errors.AssertionError,
            "expected chunk id '%s' or '%s', but got '%s'",
            wire.StatusSyncData, wire.StatusSyncDone, []byte(status),
        )
    }
}
```

Ruby

```ruby
def read_next_chunk(r)
  status, err = r.read_status("read-chunk")
  if err
    if AdbServerErrorMatching(err, readFileNotFoundPredicate)
      return nil, FileNoExistError.new("no such file or directory")
    end
    return nil, err
  end

  case status
  when wire.StatusSyncData
    return r.read_bytes
  when wire.StatusSyncDone
    return nil, EOFError.new
  else
    return nil, AssertionError.new(
      "expected chunk id '#{wire.StatusSyncData}' or '#{wire.StatusSyncDone}', but got '#{status}'"
    )
  end
end
```

Figure 5: Example of a code snippet translation from *Go* to *Ruby*.

JAVA

```java
public void get(URI uri, Credentials credentials) throws IOException {
        get(uri, null, credentials);
}
```

C

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char *username;
    char *password;
} Credentials;

typedef struct {
    char *scheme;
    char *host;
    int port;
    char *path;
} URI;

void get(URI *uri, Credentials *credentials) {
    // Implementation of the get function
    printf("Fetching resource from %s://%s:%d%s\n", uri->scheme, uri->host, uri->port, uri->path);
    if (credentials != NULL) {
        printf("Using credentials: %s:%s\n", credentials->username, credentials->password);
    }
}

int main() {
    URI uri = {"http", "example.com", 80, "/index.html"};
    Credentials credentials = {"user", "pass"};
    get(&uri, &credentials);
    return 0;
}
```

Figure 6: Example of a code snippet translation from *Java* to *C*.

Python

```
def toString(self):
        result = []
        k, v = self.optimalRepr()
        longest = reduce(lambda x, y: x if x > len(y) else len(y), k, 0)
        for ind in range(len(k)):
            result.append("%s : %s" % (k[ind].ljust(longest), v[ind]))
        return "\n".join(result)
```

PHP

```
public function toString() {

    /**

     * Return a printable view of the dictionary

     */

    $result = [];

    list($k, $v) = $this->optimalRepr();

    $longest = array_reduce($k, function($x, $y) {

        return $x > strlen($y) ? $x : strlen($y);

    }, 0);

    for ($ind = 0; $ind < count($k); $ind++) {

        $result[] = sprintf("%s : %s", ltrim($k[$ind], ' '), str_pad($v[$ind], $longest, ' ', STR_PAD_LEFT));

    }

    return implode("\n", $result);

}
```

Figure 7: Example of a code snippet translation from *Python* to *PHP*.