Extracting the Essence and Discarding the Dross: Enhancing Code Generation with Contrastive Execution Feedback

Xuanyu Zhang
Du Xiaoman Financial
Beijing, China
zhangxuanyu@duxiaoman.com

Qing Yang
Du Xiaoman Financial
Beijing, China
yangqing@duxiaoman.com

Abstract

Recent advancements have integrated the execution process and feedback into the training of large language models for code generation, demonstrating enhanced model performance. However, current methods amalgamate erroneous code with feedback and the final correct code as target sentences, inadvertently increasing the probability of generating both correct and incorrect code during inference. While multiple iterations of feedback can eventually yield the correct answer, this iterative process is cumbersome and time-consuming for users who prefer immediate accurate results. To address this challenge, we propose ConCoder, a contrastive learning-based code generation model with execution feedback. This approach enables the model to efficiently produce accurate code from the outset while rectifying and optimizing the incorrect code. Furthermore, our training emphasizes learning from the causes of errors, allowing the model to understand and avoid mistakes. Through extensive experiments, ConCoder demonstrates significant improvements in generating accurate code and understanding error correction, paving the way for more reliable code generation models.

1 Introduction

Code generation has long been a central challenge in natural language processing, with significant progress seen over the past decades. The recent works (Roziere et al., 2023; Li et al., 2023; Lozhkov et al., 2024; Guo et al., 2024; Bai et al., 2023) in leveraging large language models (LLMs) (Achiam et al., 2023) pre-trained on extensive code corpora have revolutionized this field, showcasing remarkable advancements in understanding intents and generating functional code.

Code differs from traditional text in that it can be executed and its correctness can be judged concurrently. Leveraging this unique characteristic, some research efforts have proposed utilizing code execution and feedback to enhance the capabilities of models. For example, OpenCodeInterpreter (Zheng et al., 2024) combines code execution and human feedback, harnessing compiler diagnostics to swiftly identify and rectify errors, while leveraging human insights to fine-tune and refine the code generation process. This innovative approach enables the model to deliver solutions that are technically robust and precisely tailored to user requirements, thus significantly enhancing its overall performance.

However, the methodology of integrating execution feedback into model training presents notable challenges. Typically, the training process involves appending erroneous code with feedback and the eventual correct code, which inadvertently results in the model learning not only the correct solutions but also the mistakes encountered along the way. This phenomenon increases the model's propensity to generate incorrect code during the inference phase, despite ultimately arriving at the correct answer after multiple iterations of feedback and refinement. This iterative process, while effective, is cumbersome and time-consuming, which is a significant drawback for users who desire immediate and accurate solutions.

The motivation for our research stems from the need to enhance the efficiency of code generation models by ensuring they learn to produce correct code without being distracted by the erroneous outputs encountered during training. Our proposed model, ConCoder, leverages contrastive learning principles (Hadsell et al., 2006; Chen et al., 2020) to address these challenges. The core idea behind ConCoder is to increase the separation between erroneous and correct code while simultaneously decreasing the distance between correct code snippets expressed in different ways. This objective ensures that the model not only learns to generate correct code expediently but also internalizes the reasons behind errors, thereby improving its capability to avoid similar mistakes in the future. This also avoids the problem that the probability of the model generating correct and incorrect codes simultaneously increases during the reasoning process after the problematic code is concatenated with the feedback and the final correct code as the target sentence for learning. In our experimental evaluation, ConCoder demonstrated significant improvements over previous models in terms of code generation. The model's ability to provide correct solutions at an earlier stage, without the need for iterative feedback loops, marks a substantial advancement in the field of automated code generation. This efficiency not only saves time for users but also simplifies the overall process, making automated code generation more practical and user-friendly.

In general, our paper has the following contributions:

- We are the first to introduce contrastive learning methods into code LLMs, improving the model's code generation capabilities by increasing the distance between correct and erroneous code answers, while reducing the distance between correct code answers.
- Different from traditional contrastive learning, by emphasizing the feedback analysis of error codes, the model can understand both the results and the reasons behind them.
- Experiments on four code generation datasets, HumanEval, HumanEval+, MBPP, MBPP+, shows the effectiveness of our model. Comparing with previous SOTA, OpenCodeInterpreter, ConCoder achieves 3%+ average relative improvement on these datasets.

2 Methodology

In this section, we detail the methods employed in ConCoder. Our approach is built upon the principles of contrastive learning, enabling the model to differentiate erroneous code from correct code and enhance its capacity to generate accurate solutions. Below, we systematically describe the definition, training objectives, and the specific mechanisms used in ConCoder.

Consider a dialogue-based interaction where the model is tasked with generating code snippets. In such interactions, both erroneous and correct code snippets are generated, each of which provides valuable information to the model for improving its performance. Let $\hat{h}_1$ (blue box on the left in Figure 1) represent an erroneous code snippet generated in the first-turn dialogue interaction, and h_1 (pink box on the left in Figure 1) represent the correct code snippet from the same dialogue. Similarly, let $\hat{h}_2$ (blue box on the right in Figure 1) and h_2 (pink box on the right in Figure 1) denote the erroneous and correct code snippets from another dialogue interaction for the same problem.

Our approach leverages contrastive learning principles to refine the learning process, effectively discriminating between erroneous and correct code generations based on execution feedback. The core objective is to minimize the interference of erroneous code during model training, ensuring the model predominantly learns to generate the correct code from the outset. To achieve this, we consider both contrastive learning loss and cross-entropy loss simultaneously.

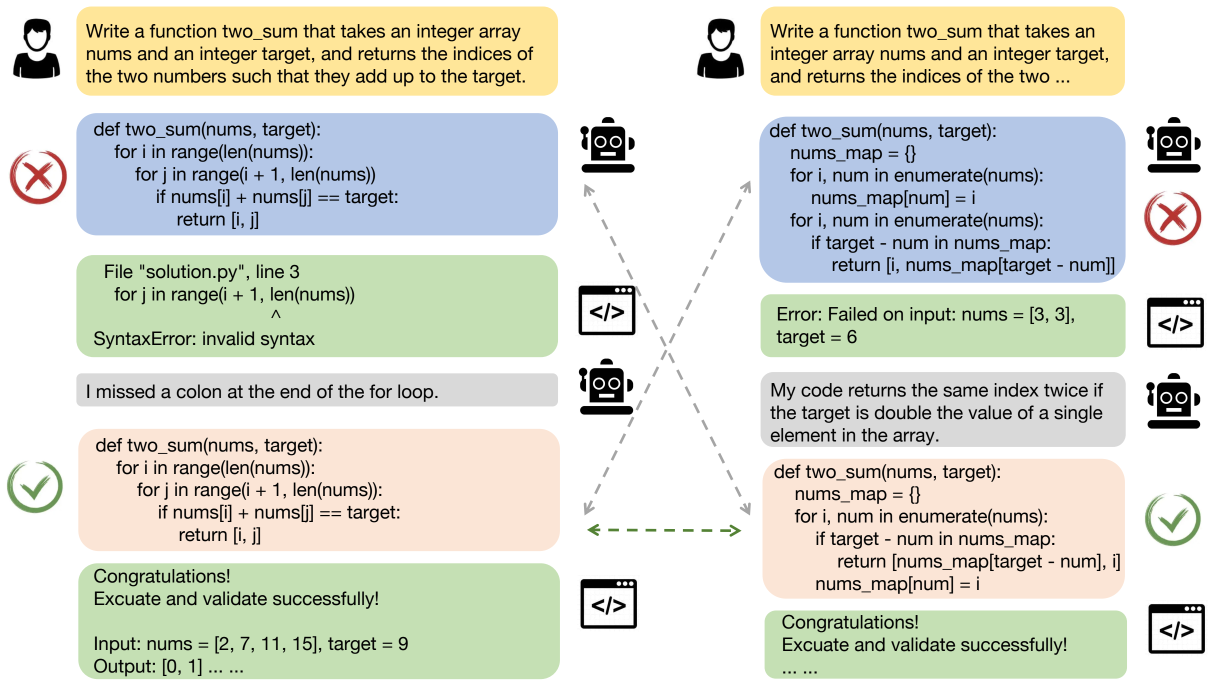
The contrastive learning loss, specifically the InfoNCE loss, focuses on capturing the relationships between correct and incorrect code snippets. The goal is to bring the correct code snippets (h_1 and h_2) closer together in the representation space, while pushing apart the erroneous snippets ($\hat{h}_1$ and $\hat{h}_2$) from the correct ones. These representations are the results after mean pooling of the last-layer hidden states corresponding to the code. This process is formulated as follows:

$$\mathcal{L}_{\text{InfoNCE}} = - \log \left(\frac{e^{sim(h_1,h_2)}}{e^{sim(h_1,h_2)} + e^{sim(h_1,\hat{h}_2)} + e^{sim(\hat{h}_1,h_2)}} \right) \quad (1)$$

where $sim(a, b)$ denotes the cosine similarity between representations a and b. Its expression form is used in various recommendation and retrieval tasks (Zhang et al., 2021, 2022a; Zhang and Yang, 2021a) in addition to being applied in contrastive learning.

To ensure that the model not only learns from the contrastive relationships but also improves the capabilities of code sequence generation and error correction and optimization, we incorporate the cross-entropy loss $\mathcal{L}_{\text{CE}}$ from the dialogue interactions. This process ensures that the model learns to recognize why certain code snippets are incorrect and how to improve upon them. The cross-entropy loss plays a crucial role in helping the model adapt and optimize its output in response to the dialogue context, further enhancing its overall performance.

The final loss function used to train the model is a weighted sum of the InfoNCE loss and the



Write a function two_sum that takes an integer array nums and an integer target, and returns the indices of the two numbers such that they add up to the target.

```
def two_sum(nums, target):
    for i in range(len(nums)):
        for j in range(i + 1, len(nums))
            if nums[i] + nums[j] == target:
                return [i, j]
```

```
File "solution.py", line 3
    for j in range(i + 1, len(nums))
                                    ^
SyntaxError: invalid syntax
```

I missed a colon at the end of the for loop.

```
def two_sum(nums, target):
    for i in range(len(nums)):
        for j in range(i + 1, len(nums)):
            if nums[i] + nums[j] == target:
                return [i, j]
```

```
Congratulations!
Excuate and validate successfully!

Input: nums = [2, 7, 11, 15], target = 9
Output: [0, 1] ... ...
```

Write a function two_sum that takes an integer array nums and an integer target, and returns the indices of the two ...

```
def two_sum(nums, target):
    nums_map = {}
    for i, num in enumerate(nums):
        nums_map[num] = i
    for i, num in enumerate(nums):
        if target - num in nums_map:
            return [i, nums_map[target - num]]
```

```
Error: Failed on input: nums = [3, 3],
target = 6
```

My code returns the same index twice if the target is double the value of a single element in the array.

```
def two_sum(nums, target):
    nums_map = {}
    for i, num in enumerate(nums):
        if target - num in nums_map:
            return [nums_map[target - num], i]
        nums_map[num] = i
```

```
Congratulations!
Excuate and validate successfully!
... ...
```

Figure 1: The training process of contrastive learning.

cross-entropy loss. By combining the above losses, the model can learn from the underlying causes of errors and the associated feedback, while also refining its ability to clearly differentiate between correct and incorrect code snippets. This distinction mitigates the potential drawback of the cross-entropy loss, where optimizing for both correct and incorrect code generation probabilities simultaneously could diminish the model's focus on producing accurate solutions from the outset. The combined loss function is expressed as:

$$\mathcal{L} = \alpha\mathcal{L}_{\text{InfoNCE}} + \mathcal{L}_{\text{CE}} \tag{2}$$

where α is a hyperparameter that controls the balance between the two loss components. By adjusting α, we can regulate the relative contribution of the contrastive learning objective and the cross-entropy loss, enabling us to optimize the model's performance based on specific training goals.

3 Experiment

3.1 Dataset and Implementation

During the training phase, we construct a large-scale dataset consisting of 200k code examples with execution feedback using GPT-4. It leverages GPT-4's advanced language and reasoning capabilities to simulate a variety of realistic programming tasks and generate corresponding execution feedback. And this dataset is generated based on the open-source code dataset available in OpenCodeInterpreter (Zheng et al., 2024), which aggregates multiple datasets from Code Alpaca (Chaudhary, 2023), Magicoder (Wei et al., 2023), WizardCoder (Luo et al., 2023) and so on. Each code sample is equipped with detailed feedback, highlighting the execution outcomes and potential improvements or corrections for the code. By curating a diverse range of examples, we ensure that the dataset encompasses a broad spectrum of programming concepts and scenarios.

For the prediction and evaluation phase, we select four widely used datasets in code evaluation: HumanEval, HumanEval+, MBPP, and MBPP+. Specifically, HumanEval+ and MBPP+ are expanded versions of the original HumanEval and MBPP datasets, created by EvalPlus (Liu et al., 2024), which have been enlarged by several dozen times to provide a more comprehensive testbed. We adopt a single-turn configuration, meaning that for each problem, our model generates only one code answer without multiple iterations or external feedback. The evaluation is conducted using the pass@1 metric.

In our experiments, we employ NVIDIA A800 GPU for training and evaluation. The hyper-parameters of models are decided by original base model, including Code LLaMA and DeepSeek Coder. The batch size of the model is set to 256.

Model	Size	Type	HumanEval (+)	MBPP (+)	Average (+)
		API			
GPT-4 Turbo (Achiam et al., 2023)	-	-	85.4 (81.7)	83.0 (70.7)	84.2 (76.2)
GPT-3.5 Turbo	-	-	72.6 (65.9)	81.7 (69.4)	77.2 (67.7)
Gemini Pro (Saeidnia, 2023)	-	-	63.4 (55.5)	72.9 (57.9)	68.2 (56.7)
		Open Source Models			
StarCoder (Li et al., 2023)	7B	Base	24.4 (20.7)	33.1 (28.8)	28.8 (24.8)
CodeT5+ (Wang et al., 2023b)	6B	Base	29.3 (23.8)	51.9 (40.9)	40.6 (32.4)
CodeGen-Mono (Nijkamp et al., 2022)	6B	Base	29.3 (25.6)	49.9 (42.1)	39.6 (33.9)
Mistral (Jiang et al., 2023)	7B	Base	28.7 (23.2)	50.1 (40.9)	39.4 (32.1)
OpenChat (Wang et al., 2023a)	7B	Instruct	72.0 (67.1)	62.7 (52.9)	67.4 (60.0)
CodeLlama-Python (Roziere et al., 2023)	7B	Base	37.8 (34.1)	57.6 (45.4)	47.7 (39.8)
WizardCoder-CL (Luo et al., 2023)	7B	Instruct	48.2 (40.9)	56.6 (47.1)	52.4 (44.0)
Magicoder-CL (Wei et al., 2023)	7B	Instruct	60.4 (55.5)	64.2 (52.6)	62.3 (54.1)
Magicoders-S-CL (Wei et al., 2023)	7B	Instruct	70.7 (66.5)	68.4 (56.6)	69.6 (61.6)
OpenCodeInterpreter (Zheng et al., 2024)	7B	Instruct	72.6 (67.7)	66.4 (55.4)	69.5 (61.6)
ConCoder (Our Model)	7B	Instruct	**73.2 (68.3)**	**70.9 (59.8)**	**72.1 (64.1)**
DeepseekCoder (Guo et al., 2024)	6.7B	Base	47.6 (39.6)	70.2 (56.6)	58.9 (48.1)
DeepseekCoder-Instruct	6.7B	Instruct	73.8 (70.1)	73.2 (63.4)	73.5 (66.8)
Magicoder-DS (Wei et al., 2023)	6.7B	Instruct	66.5 (60.4)	75.4 (61.9)	71.0 (61.2)
Magicoder-S-DS (Wei et al., 2023)	6.7B	Instruct	76.8 (70.7)	75.7 (64.4)	76.3 (67.6)
OpenCodeInterpreter (Zheng et al., 2024)	6.7B	Instruct	76.2 (72.0)	73.9 (63.7)	75.1 (67.9)
ConCoder (Our Model)	6.7B	Instruct	**79.3 (72.0)**	**81.7 (69.0)**	**80.5 (70.5)**
- contrastive loss	6.7B	Instruct	77.4 (72.0)	79.6 (68.8)	78.5 (70.4)
- a negative pair in contrastive loss	6.7B	Instruct	79.1 (72.7)	80.5 (68.3)	79.8 (70.5)

Table 1: Pass@1 of different code LLMs on HumanEval (+), MBPP (+) and their average (+).

And the max length of code sequence is set to 4096. The AdamW (Loshchilov and Hutter, 2018) is used as our optimizer with 4e-5 learning rate. α is set to 0.35 according to the performance of our model.

3.2 Result and Analysis

As illustrated in Table 1, we conduct experiments on code LLMs of various base models, including CodeLlama-7B (Roziere et al., 2023) and DeepseekCoder-6.7B (Guo et al., 2024). Other results are obtained from OpenCodeInterpreter (Zheng et al., 2024). Compared with the previous best feedback-based model, OpenCodeInterpreter, our model, ConCoder, has achieved an average relative improvement of approximately 3% to 7%. In particular, for the model based on DeepseekCoder, our method has increased the Pass@1 from 73.9% to 81.7% on MBPP. The improvement is significant, indicating that our approach is more effective in utilizing feedback to generate accurate and relevant code. Besides, the consistent performance enhancements across diverse architectures suggest that ConCoder is not limited to a specific model or dataset. Instead, it can be seamlessly integrated into a variety of pre-trained code LLMs. This adaptability makes ConCoder a highly versatile tool, capable of enhancing code generation across a broad range of scenarios and data distributions.

As further evidenced by the results at the bottom of Table 1, our ablation study sheds light on the pivotal role played by the contrastive learning component in the success of ConCoder. When the contrastive mechanism was entirely removed, the model's performance suffered a marked decline on both the HumanEval and MBPP datasets. This outcome underscores the importance of contrastive learning in guiding the model to better understand feedback. When only one negative pair is removed from the contrastive loss, the impact is less pronounced on HumanEval but considerably larger on MBPP. Intuitively, MBPP encompasses a broader range of programming tasks and samples, which particularly benefits from the enriched representation learning facilitated by multiple negative pairs.

4 Related Work

4.1 Pre-training with Code

Code plays a critical role in the pre-training of large language models (Zhang and Yang, 2023c, 2025), significantly contributing to their reasoning and problem-solving capabilities. By incorporating code into the training corpus, LLMs can better handle complex logical structures and sequential reasoning tasks. For this reason, many general-purpose LLMs, such as LaMDA (Thoppilan et al., 2022), LLaMA (Touvron et al., 2023a,b) and PaLM (Chowdhery et al., 2023), include a proportion of code data in their pre-training datasets, ranging from 5% to 13% of their training corpus.

In addition to general-purpose LLMs, there has been a surge of specialized LLMs designed explicitly for code-related tasks. These models are typically pre-trained on vast amounts of code data, allowing them to achieve high performance in code generation, debugging, and understanding. Notable examples include CodeGeeX (Zheng et al., 2023), CodeT5+ (Wang et al., 2023b), StarCoder (Li et al., 2023), CodeLLaMa (Roziere et al., 2023) and DeepSeek-Coder (Guo et al., 2024; Zhu et al., 2024). These models rely heavily on massive pre-training datasets consisting primarily of code, showcasing the importance of domain-specific corpora in addressing programming tasks.

4.2 Fine-tuning with Code

Despite the success of specialized LLMs for code, pre-training such models requires extensive computational resources and a vast amount of data, which may not always be feasible. Furthermore, while pre-trained models are capable of generating syntactically correct code, they often lack fine-grained instruction-following and interactive capabilities essential for real-world applications. To address these challenges, fine-tuning approaches have emerged as an effective alternative. Fine-tuning leverages pre-trained general-purpose models by adapting them to specific domains (Zhang and Yang, 2023b; Zhang et al., 2023a) or tasks (Zhang, 2019, 2020; Zhang and Wang, 2020; Zhang and Yang, 2021b; Zhang et al., 2022b, 2023b; Zhang and Yang, 2023a; Zhang et al., 2023b) with smaller and task-specific datasets. In the field of code, WizardCoder (Luo et al., 2023) introduced Evol-Instruct, a framework that evolves existing instruction data into more complex and diverse datasets, enhancing the model's ability to handle intricate instructions. Similarly, MagiCoder (Wei et al., 2023) proposed the OSS-Instruct method, which leverages open-source code snippets to generate diverse, realistic, and controllable coding instruction datasets. These datasets have proven instrumental in significantly improving the performance of various LLMs through instruction tuning.

4.3 Fine-tuning with Code Feedback

The executable nature of code has led to the development of numerous feedback-driven methods for improving code generation through iteration. Unlike traditional text generation tasks, code outputs can be executed to provide concrete feedback in the form of runtime results or error messages. This has motivated researchers to explore iterative approaches, where the feedback from previous iterations is used to refine and improve subsequent generations. For example, OpenCodeInterpreter (Zheng et al., 2024) amalgamates execution and human feedback to effectuate dynamic code refinement. It deploys compiler diagnostics for the rectification of errors and capitalizes on human insights to refine code generation.

Although OpenCodeInterpreter (Zheng et al., 2024) successfully integrates code execution feedback into the code generation process by utilizing compiler diagnostics and user feedback for dynamic optimization, its training approach combines erroneous and the final correct code into a single target sequence. That is to say, this method allows the model to iteratively refine and correct its outputs, but it also increases the likelihood of generating incorrect code during the initial stages of inference. However, the contrastive learning method we use can solve this problem well, allowing the model to extract its essence and discard its dross.

5 Conclusion

In this paper, we introduce ConCoder, a novel contrastive learning-based model for code generation that addresses the limitation where the model tends to simultaneously increase the probability of generating both correct and incorrect code. Unlike previous models that relied solely on cross-entropy loss, ConCoder enhances the separation between correct and incorrect code representations, ensuring that the model generates accurate code from the outset and reduces reliance on iterative feedback. Extensive experiments on diverse datasets showcased the superior performance of ConCoder.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. *GitHub repository*.

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

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Raia Hadsell, Sumit Chopra, and Yann LeCun. 2006. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Ilya Loshchilov and Frank Hutter. 2018. Fixing weight decay regularization in adam.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Hamid Reza Saeidnia. 2023. Welcome to the gemini era: Google deepmind and the information industry. *Library Hi Tech News*, (ahead-of-print).

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv preprint arXiv:2201.08239*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023a. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Guan Wang, Sijie Cheng, Xianyuan Zhan, Xiangang Li, Sen Song, and Yang Liu. 2023a. Openchat: Advancing open-source language models with mixed-quality data. *arXiv preprint arXiv:2309.11235*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Xuanyu Zhang. 2019. MC^2: Multi-perspective convolutional cube for conversational machine reading comprehension. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 6185–6190, Florence, Italy. Association for Computational Linguistics.

Xuanyu Zhang. 2020. Cfgnn: Cross flow graph neural networks for question answering on complex tables. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05):9596–9603.

Xuanyu Zhang, Bingbing Li, and Qing Yang. 2023a. Cgce: A chinese generative chat evaluation benchmark for general and financial domains. *arXiv preprint arXiv:2305.14471*.

Xuanyu Zhang, Zhepeng Lv, and Qing Yang. 2023b. Adaptive attention for sparse-based long-sequence transformer. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 8602–8610, Toronto, Canada. Association for Computational Linguistics.

Xuanyu Zhang and Zhichun Wang. 2020. Rception: Wide and deep interaction networks for machine reading comprehension (student abstract). *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(10):13987–13988.

Xuanyu Zhang and Qing Yang. 2021a. Dml: Dynamic multi-granularity learning for bert-based document reranking. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM '21, page 3642–3646, New York, NY, USA. Association for Computing Machinery.

Xuanyu Zhang and Qing Yang. 2021b. Position-augmented transformers with entity-aligned mesh for textvqa. In *Proceedings of the 29th ACM International Conference on Multimedia*, MM '21, page 2519–2528, New York, NY, USA. Association for Computing Machinery.

Xuanyu Zhang and Qing Yang. 2023a. Generating extractive answers: Gated recurrent memory reader for conversational question answering. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 7699–7704, Singapore. Association for Computational Linguistics.

Xuanyu Zhang and Qing Yang. 2023b. Self-qa: Unsupervised knowledge guided language model alignment. *arXiv preprint arXiv:2305.11952*.

Xuanyu Zhang and Qing Yang. 2023c. Xuanyuan 2.0: A large chinese financial chat model with hundreds of billions parameters. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, CIKM '23, page 4435–4439, New York, NY, USA. Association for Computing Machinery.

Xuanyu Zhang and Qing Yang. 2025. Finmoe: A moe-based large chinese financial language model. In *Proceedings of the Ninth Workshop on Financial Technology and Natural Language Processing*, pages 42–45, Abu Dhabi, UAE. Association for Computational Linguistics.

Xuanyu Zhang, Qing Yang, and Dongliang Xu. 2021. Combining explicit entity graph with implicit text information for news recommendation. WWW '21, page 412–416, New York, NY, USA. Association for Computing Machinery.

Xuanyu Zhang, Qing Yang, and Dongliang Xu. 2022a. Deepvt: Deep view-temporal interaction network for news recommendation. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, CIKM '22, page 2640–2650, New York, NY, USA. Association for Computing Machinery.

Xuanyu Zhang, Qing Yang, and Dongliang Xu. 2022b. TranS: Transition-based knowledge graph embedding with synthetic relation representation. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 1202–1208, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.