REASONING DISTILLATION AND STRUCTURAL ALIGNMENT FOR IMPROVED CODE GENERATION

Amir Jalilifard¹, Anderson de Rezende Rocha¹, and Marcos Medeiros Raimundo¹

¹Universidade Estadual de Campinas, Campinas, Brazil, jalilifard@ic.unicamp.br, anderson.rocha@unicamp.br, mrai@unicamp.br

ABSTRACT

Effective code generation with language models hinges on two critical factors: accurately understanding the intent of the prompt and generating code that applies algorithmic reasoning to produce correct solutions capable of passing diverse test cases while adhering to the syntax of the target programming language. Unlike other language tasks, code generation requires more than accurate token prediction; it demands comprehension of solution-level and structural relationships rather than merely generating the most likely tokens. very large language model (VLLM) are capable of generating detailed steps toward the correct solution of complex tasks where reasoning is crucial in solving the problem. Such reasoning capabilities may be absent in smaller language models. Therefore, in this work, we distill the reasoning capabilities of a VLLM into a smaller, more efficient model that is faster and cheaper to deploy. Our approach trains the model to emulate the reasoning and problem-solving abilities of the VLLM by learning to identify correct solution pathways and establishing a structural correspondence between problem definitions and potential solutions through a novel method of structure-aware loss optimization. This enables the model to transcend token-level generation and to deeply grasp the overarching structure of solutions for given problems. Experimental results show that our fine-tuned model, developed through a cheap and simple to implement process, significantly outperforms our baseline model in terms of pass@1, average data flow, and average syntax match metrics across the MBPP, MBPP Plus, and HumanEval benchmarks.

Keywords Large Language Models · Coding · Knowledge Distillation · Chain of Thoughts

1 Introduction

Large language models (LLMs) can generate text, translate from one language to another, and carry out various auto-regressive, sequence-to-sequence, and in-filling tasks, all using a single model architecture. Tasks involving logical problems require a series of steps, one after another, until the final solution arrives. However, tasks that involve a deep understanding of the problem and complex reasoning are still challenging for LLMs. Automatic coding, the task of generating codes given a programming problem, is an example of a task where logical steps should be followed until the final correct solution is reached. Compared to general language-related tasks, coding requires several complex generation abilities such as syntax and exact keyword match, algorithmic thinking, and bottom-up or top-down problem-solving capabilities. Therefore, more sophisticated approaches have been developed for training, inference, and evaluation steps of code generator language models [5, 34, 36]. The recent efforts to enhance code generation by LLMs can be roughly summarized in four major categories: a) training models with increased parameters in order to generate a richer latent space and a larger context window; b) in-context learning; c) code generation through agents interactive feedback; and d) data and instruction optimization. Each approach has advantages and disadvantages, making them appealing in specific conditions.

In-context learning is a subset of prompt engineering methods that aims at constructing a bridging context between the main prompt and the expected correct answer. N-shot prompts add bridging context by elucidating the task to be done with or without examples [7]. One major problem with the n-shot approach is its high sensitivity to the quality, order,

and relevance of the examples [43], their orders, and templates [28, 23]. These complexities become even more evident when the task to be solved has different criteria of correct ordering, relevance, and importance of the given examples.

Therefore, researchers have explored that instead of adding context, they **iteratively generate** context with LLM reasoning. Wei et al. [40] explored the contribution of a series of intermediate reasonings, called Chain-of-Thought (CoT), and concluded that it could significantly boost the complex reasoning performance of LLMs. CoTs can improve the LLM's response due to generating a sequence of intermediate steps that facilitate LLM's reasoning. However, CoT does not explore the solution space, and for more complicated problems, it may not produce the best sequence of steps toward the correct solution. To improve this shortcoming, Wei and colleagues introduced Self-consistency with CoT (CoT-SC), an ensemble of thoughts generated, and then the final response is selected using majority vote [38]. Although CoT and its variants have increased the reasoning abilities of LLMs, they suffer from a lack of self-evaluation and self-improvement while generating thoughts. Tree-of-thoughts applies a self-evaluation and reasoning correction through a depth- or breadth-first-search and explores the solution space more efficiently and achieves a significant performance improvement in non-trivial logical tasks [42]. Both ToT and CoT-SC bring a broader range of solutions that improve the reasoning abilities of LLMs. However, these methods are significantly more expensive and slower than less sophisticated thought-generation methods. Some recent studies proposed slight changes to the original CoT by adding more pseudo-codes like step-by-step reasoning or adapted CoTs, which are more code structure alike [26] [30]. Despite promising results, these methods suffer from all the few-shot prompting issues mentioned above and long prompts that may be less understandable by humans, which makes debugging and interpretability very slow, if not impossible.

Another group of recent efforts has tried to use feedback loops, self-evaluation, auto-correction, and divide-and-conquer strategies to use a group of separate specialized instances of LLMs, called agents, to solve complex reasoning tasks. ToTs, for example, can be generated based on the evaluation and solution selection of two or more agents. Agents have been widely used for code generation. They can give verbal reinforcement and feedback so that the main reasoner corrects its approach toward the solution [35]. These LLM instances can also take several rules similar to real-world software engineering teams to program, document, test, and return code correction and logic improvement feedbacks [18] [10]. Despite phenomenal problem-solving improvements by LLM agents, they are extremely costly and unduly prolonged.

Having a structured, high-quality, and massive data set of programming tasks and their associated codes is fundamental for code generation. Jain et al. [20] investigate data quality by transforming existing programs to be more structured and readable. The proposed data-cleaning pipeline includes renaming variables, modularizing complex code, and inserting natural-language-based plans via LLM-based transformations, leading to performance gains in code generation tasks. Tsai and colleagues [37] evaluated the effect of code quality and diversity versus quantity and demonstrated that a more pruned, high-quality code could reduce the computational cost and lead to better code generation. Gong et al. address the limitations of existing code search datasets by pairing high-quality queries with multiple suitable code matches. The authors collect diverse code candidates and utilize LLMs for automated annotation, filtering, and code generation, resulting in a dataset that improves model performance on code-related tasks [14]. High-quality data is essential for training competent models; however, with new technologies, new versions of programming languages, and new algorithms, the data should be up to date. In such a scenario, other LLMs' effectiveness in pruning low-quality code is unknown, and manual dataset refining is time-consuming and expensive.

Very Large language models have a richer latent space, enabling them to capture deeper semantic and linguistic relationships within sentences, in addition, to feature extended context windows and more nuanced attention mechanisms [39] [44]. As model size increases, these models often demonstrate unexpected capabilities, known as emergent abilities, including enhanced instruction following and step-by-step reasoning [21] [7] [22]. Although recent studies suggest that larger models do not consistently outperform smaller ones [16], it has been shown that code generation performance consistently improves within model families as size increases across various benchmarks [34]. Despite their advantages, VLLMs are costly to train, maintain, and deploy for inference.

In Masked Language Models (MLM) like BERT [9], in different abstraction levels represent from simple similarities between word relations and their relative position in a sentence to more complex semantic and syntactic similarities [3] [33]. In a code MLM, the embeddings from the last abstraction layer might capture the general meaningful coding semantic and synthetic patterns in addition to the algorithmic structure. Magister et al. investigated the possibility of training a student model to learn how to reason [31], yet their study is focused on tasks other than coding. In contrast to different tasks, LLMs can struggle when reasoning to solve coding tasks. This can be even more evident for tasks with higher complexity, non-trivial logical and arithmetic operators[27]. Further, their method is expensive as the training is an entire parameter prone to catastrophic forgetting.

In this paper, our main contributions are as follows:

- Distilling the thought-generation and problem-solving capabilities in very large language models (VLLMs) through parameter-efficient fine-tuning enables a smaller model to learn step-by-step reasoning for accurately solving coding tasks. This means teaching the smaller model to imitate the thinking process of the larger model and to learn how it reason step by step until solving the problem. In practice, it not only enables the smaller model to reason better, but also it might refine the embedding space of the smaller model.
- To instill an understanding of general code structure and problem-solving pathways for programming tasks, we propose a novel loss function that extends beyond basic token generation, incorporating both code structure and general algorithmic design similarities.

The primary objective of this study is to investigate how effectively reasoning distillation enables a smaller model to replicate the implicit chain-of-thought reasoning of a larger model. Additionally, we aim to explore whether a model can be trained to map clusters of similar coding problems — along with their associated reasoning patterns, algorithms, and coding styles — within the embedding space.

2 Method

We first formalize the general Chain-of-Thought(CoT) framework for problem-solving, upon which our approach is constructed. We then prove the learnability of automatically generated context for token-level learning. Afterward, we show how to adapt the token-level loss with the structural characteristics of code.

2.1 Notation

We denote a pre-trained model by p_{θ} . Let $X = \{x_1, x_2, \dots, x_n\}$ represent the sequence of tokens in the task prompt, and $Y = \{y_1, y_2, \dots, y_m\}$ denote the sequence of ground truth response tokens. We define $Z = \{z_1, z_2, \dots, z_{i-1}, z_i\}$ as the sequence of tokens that provide auxiliary context for the model to predict the correct answer; we refer to this auxiliary context as the *bridging* context.

2.2 Reasoning distillation

Given an initial task prompt X, when the mapping between this task and the correct answer Y is non-trivial, in-context prompting methods such as Chain-of-Thought (CoT) and Tree-of-Thought (ToT) propose adding a chain of reasoning steps that bridge the gap between the task and the correct answer by providing informative context Z. Such context can be generated by sampling a sequence of tokens z_i from a pre-trained model, where

$$Z^{(k)} \sim p_{\theta}(z_i^{(k)} \mid X, z_1^{(k)}, z_2^{(k)}, \dots, z_{i-1}^{(k)}),$$

With k being the number of thoughts that can be generated. For ToT and CoT-SC (Self-Consistency), we have k>1. These thoughts can be reevaluated at the end or during the thought generation process to choose the most relevant reasoning steps for a given problem. The sequence of thoughts can be sampled from the same model $[Z^{(k)},Y]\sim p_{\theta}(Z^{(k)},Y\mid X)$ or using independent specialized pre-trained models, followed by a feedback loop (i.e., agents to generate and evaluate in ToT).

We approach the learnability of the sequentially generated contexts within a system of independent pre-trained models, as a general framework for context generation, that collaborate to generate a series of tokens associated with a bridging context and show that such context is learnable. The original CoT and our distillation method is a specific and simpler case of such a system.

Suppose there is a system of Q pre-trained VLLMs $M = \{m_1, m_2, \dots, m_q\}$ which are sampled sequentially to generate a bridging context for a task T. Each model generates tokens auto-regressively by sampling from its probability distribution p_{θ_q} , where θ_q are the parameters of model m_q .

Given an initial prompt, each model m_q samples a sequence of tokens $X_n = \{x_{n1}, x_{n2}, \dots, x_{nL_n}\}$ from a conditional probability distribution:

$$X_n \sim \prod_{i=1}^{L_n} p_{\theta_q}(x_{ni} \mid T, X_{\leq n}, x_{n1}, x_{n2}, \dots, x_{n(i-1)}), \tag{1}$$

Where $X_{\leq n} = \{X_1, X_2, \dots, X_{n-1}\}$ is the sequence of tokens generated by previous models, and L_n is the length of the sequence generated by model m_q .

Using the chain rule, the joint probability distribution over all tokens generated by the models is:

$$p(T, X_1, X_2, \dots, X_K) = p(T) \prod_{n=1}^K \prod_{i=1}^{L_n} p_{\theta_q}(x_{ni} \mid T, X_{< n}, x_{n1}, x_{n2}, \dots, x_{n(i-1)}).$$
 (2)

For a dataset $D = \{(T_i, X_{1i}, X_{2i}, \dots, X_{Ki})\}_{i=1}^N$ of N tasks and their generated contexts, the log-likelihood of the observed data under the joint probability distribution is given by:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \log p(T_i, X_{1i}, X_{2i}, \dots, X_{Ki}),$$
(3)

which can be expanded using the chain rule from Equation (2):

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \left[\log p(T_i) + \sum_{n=1}^{K} \sum_{j=1}^{L_n} \log p_{\theta_q}(x_{nij} \mid T_i, X_{< n, i}, x_{ni1}, x_{ni2}, \dots, x_{ni(j-1)}) \right], \tag{4}$$

where
$$X_{\leq n,i} = \{X_{1i}, X_{2i}, \dots, X_{(n-1)i}\}.$$

Our optimization objective is to maximize the log-likelihood $\mathcal{L}(\theta)$, which is equivalent to minimizing the negative log-likelihood of the generated contexts given their associated tasks. Assuming that the true joint probability distribution is p and we want to approximate it using p_{θ} , the Kullback-Leibler (KL) divergence between these two distributions is:

$$D_{KL}(p \parallel p_{\theta}) = \sum_{x \in \mathcal{X}} p(x) \log \frac{p(x)}{p_{\theta}(x)} = \sum_{x \in \mathcal{X}} p(x) [\log p(x) - \log p_{\theta}(x)], \tag{5}$$

where $\mathcal{X} = \{T, X_1, X_2, \dots, X_K\}.$

This can be expressed as:

$$D_{KL}(p \parallel p_{\theta}) = -\mathbb{E}_p[\log p_{\theta}(x)] + H_p, \tag{6}$$

where $H_p = -\sum_{x \in \mathcal{X}} p(x) \log p(x)$ is the entropy of the true distribution, and $-\mathbb{E}_p[\log p_{\theta}(x)]$ is the expected negative log-likelihood under the true distribution.

Since H_p does not depend on θ , minimizing $D_{KL}(p \parallel p_{\theta})$ is equivalent to minimizing the expected negative log-likelihood $-\mathbb{E}_p[\log p_{\theta}(x)]$. Therefore, by maximizing the log-likelihood $\mathcal{L}(\theta)$, we are effectively minimizing the KL divergence between the true distribution p and our model distribution p_{θ} . The joint probability distribution can be approximated effectively with sufficient data and appropriate model capacity.

Therefore, the bridging context Z is learnable through maximizing the log-likelihood of the joint distribution of tasks and generated contexts under three conditions: a) the transformer model which tries to learn such joint probability distribution should have enough number of parameters; b) the context size should be less than the context window of such a model; and c) there should be enough training examples. It is worth mentioning that despite a theoretical foundation, practical complexities may arise while learning contexts generated by a group of independent specialized language models.

2.3 Learning objective

During the parameter efficient fine-tuning, we minimize the following loss function:

$$\mathcal{L} = \alpha \times \mathcal{L}_{token} + \beta \times \mathcal{L}_s \tag{7}$$

where \mathcal{L}_{token} represents token-level loss presented in Equation 3), calculated over the CoTs generated by a VLLM, the ground truth code, and a set of test cases. The structural loss, \mathcal{L}_s , quantifies the cosine distance between the embeddings of generated code and ground truth code:

$$\mathcal{L}_s = 1 - \frac{E_{gt} \cdot E_{gen}}{|E_{gt}| \times |E_{gen}|} \tag{8}$$

Here, E_{gt} and E_{gen} denote the embeddings of ground truth and generated code, respectively, computed using CodeBERT [12].

The weights α and β are normalized parameters adjusted dynamically based on Curriculum Learning principles [6], with $\beta=1-\alpha$. Early in training, we emphasize token-level accuracy by setting a higher α ; as training progresses, β gradually increases, shifting focus towards structural alignment over exact token matching. To ensure both token and structure losses remain active throughout training without reducing either to zero, we set predefined lower bounds for both α and β . The loss calculation remains active for the CoT, code, and test cases, while attention weights are applied across the entire prompt.

2.4 Data

We used the Taco dataset [25], which consists of a set of 26443 Python coding questions and test cases. The questions are gathered from a variety of programming sources such as CodeForce and Leetcode. Since each question may have several correct answers submitted by different users, each question may have more than one correct solution. We chose questions for which the total number of token for questions, codes, and test cases altogether is less than 2000 tokens. This pre-processing step resulted in a total of 15360 questions and their associated first correct solution. Since each question may have more than one correct answer, we randomly chose 3000 questions from this set and then, from the remaining solutions, chose one for each question randomly. We used these alternative solutions to familiarize the model with different ways of solving the same question. Our final dataset consists of 18360 programming tasks. These questions have three different difficulty levels including easy, medium, and hard programming questions. The questions are generally long (which is challenging for LLMs) and consist of some examples, in addition to the main task description.

3 Experiment setup and preliminary analysis

Studies have shown that relevant contexts such as CoTs can enhance LLMs' commonsense and arithmetic reasoning performance. However, it is unclear how human language CoTs influence code generation.

Before further experimentation, we first evaluated how context quality impacts the quality of generated code in terms of correctness and closeness to a valid solution. To minimize the effect of context size on model responses, we randomly selected 100 questions from the MBPP's [4] training dataset, which includes shorter programming questions than those found in the Taco dataset. We tasked **Llama 3.1 8B** Instruct to create contexts with all previously mentioned elements. Combined with their corresponding questions, these contexts were then provided to Code **Llama 7B** [34], a smaller model from the same family, to generate solution code. To measure how closely the generated code matched the correct solution, we employed a *voting prompt* [42] and asked **GPT-4** and **Claude 3.5** Sunnet to rate both the generated context and code on a scale from 0 to 10, where 0 represents an entirely irrelevant solution and 10 a completely correct solution. Interestingly, the Pearson correlation coefficients for **GPT-4** and Claude **3.5 Sunnet** were 0.65 and 0.60, respectively, indicating a significant correlation between the context's quality and the generated code's quality. Further, the correlation for intermediate scores (4 to 7) is 0.31 and 0.27 for **GPT-4** and **Claude 3.5 Sunnet**, respectively. This correlation score suggests moderate context and reasoning quality may lead to inconsistent code quality. Therefore, generating high-quality CoTs, problem definitions, and test cases is essential for better code generation. Such contexts can be used directly on the prompt or can be used to refine the embedding space of language models to generate better responses.

Therefore, we propose generating rich context, in terms of question understanding and reasoning, with huge language models, and using them to train a smaller language model so that it can learn how to follow a series of correct reasoning steps to generate a valid code given a programming question. To do so, we use parameter efficient fine tuning along with a new loss that helps the model learn both the sequence of correct code tokens to be generated and the general structure of code for group of similar problems using embedding similarity. Our approach is simple to implement and cheap to execute.

3.1 Context generation

We employ Llama 3.1 70B [11] to generate a structured context for each problem, comprising: a) the main intention of the question, b) a sequence of algorithmic steps leading to the correct solution, c) relevant mathematical formulas if

applicable, and d) potential edge cases. These contexts are generated based on the ground truth solutions identified for each question. To maintain objectivity, we provide Llama with the correct solution but instruct it to generate a step-by-step problem-solving sequence without explicitly revealing the correct code or offering direct coding hints. Additionally, we prompt Llama to explain how specific Python built-in features or libraries, as used in the ground truth code, contribute to cleaner and more efficient solutions.

3.2 Fine-tuning

Low-rank adaptation (LoRA) for Large Language Models [17] was employed with ranks of 16, 32, and 64, and corresponding α values of 8, 16, 32, 64, and 128, to train and establish the mapping between programming questions, the problem's main intention, step-by-step reasoning, code, and test cases. The training configuration included a 5×10^{-5} learning rate, 20 warm-up steps, a dropout rate of 0.1, 16-bit floating-point precision, and the Adam optimizer. Each model was evaluated using various benchmarks. The best performance was achieved when the LoRA rank and α were set to 32, although consistent improvements were observed across all configurations. Llama 3.1 8B was used as the base model for fine-tuning.

4 Results and discussion

We evaluated our model across several benchmarks, including MBPP, MBPP Plus, and HumanEval [8], selecting pass@1 as the primary evaluation metric. Pass@1 indicates the probability that a model correctly passes all test cases on its first attempt. Due to the known limitations associated with few-shot prompting, all evaluations were conducted under zero-shot settings. We first compared our model's performance against the base model, followed by comparisons against smaller LLMs with up to 16 billion parameters. Our results are organized into two categories: 1) Context Distillation (CD) alone, and 2) Context Distillation combined with our proposed structure-aware loss. Given that MBPP Plus is a newer variant of MBPP with limited reported results in the literature, we restrict our comparative analysis with smaller LLMs to the HumanEval and MBPP benchmarks.

Model (zero-shot)	MBPP+	MBPP	HumanEval
Llama 3.1 8B (baseline model)	48.2	37.67	21.95
Ours - Only context distillation	56.31	42.85	28.83
Ours - structure-aware context distillation	56.86	42.42	35.86

Table 1: Pass@1 performance on zero-shot setting compared to the baseline model.

Model (zero-shot)	MBPP	HumanEval
CODEGEN-MONO 2.7B [32]	27.31	33.4
CODEGEN-MONO 6.1B [32]	32.48	_
CodeGeeX 13B [45]	22.9	24.4
SantaCoder 1.1B [2]	14.0	35.0
StarCoder 15.5B [24]	33.0	52.0
WizardCoder 15B [29]	51.8	57.3
Phi1 1.3B [15]	55.5	50.6
INCODER 6B [13]	21.30	_
Code-Cushman-001 2.5B [1]	45.90	33.5
Code Llama 7B [34]	25.1	26.1
Llama 3.1 8B [11]	37.67	21.95
Ours - only context distillation	42.85	28.83
Ours - structure-aware context distillation	42.42	35.86

Table 2: Pass@1 performance on the zero-shot setting of small models on MBPP and HumanEval datasets.

As shown in Table 1, our fine-tuned model demonstrates notable improvements across all benchmarks. The distillation of the previously described context contributed to consistent performance gains on each benchmark. Specifically, incorporating approximately 14% structure-aware loss resulted in a significant performance boost on HumanEval, alongside a slight improvement on MBPP Plus, though it led to a marginally reduced pass@1 performance on MBPP. Additionally, we observed a performance increase of 9% (pass@1 of 34%) on MBPP and 4% (pass@1 of 30%) on

HumanEval when using Code Llama 7B as the base model. These results suggest that our model can improve the correct code generation in its first attempt between 4% to 14% depending on the benchmark.

Table 2 further highlights the performance of our model in comparison with other small models. Unlike these alternatives, our fine-tuned model is significantly more cost-effective to train and offers competitive performance. The Phi1 [15] model with 1.3B is the only model smaller than ours that has better performance in both MBPP and HumanEval, yet our model is significantly cheaper due to using a straightforward parameter efficient fine-tuning and without any need to construct text-book quality training datasets and training from scratch. Further, our model is based on a vastly used open-source model in dozens of studies and fine-tuning efforts. Using such a well-known open-source model brings new possibilities like using model merging methods and task vector arithmetic to create new models with multitasking abilities and more fair and unbiased results [19]. WizardCoder [29] uses a complex process to generate a training dataset using Evol-Instruct [41] approach. With 15 billion parameters, its performance exceeds all the other models, yet it is highly expensive to evaluate the generated dataset and train compared to our cheap approach, which costs approximately \$50 for its entire dataset generation and fine-tuning pipeline. The synthetic dataset generation step is not only expensive, but it is also prone to bias due to reliance on GPT3.5 and may lack the diversity required to solve real-world problems. Finally, it is unclear how smaller versions of WizardCoder perform on MBPP and HumanEval benchmarks.

We evaluated our model's performance by comparing its average dataflow and syntax match similarity scores with the ground truth code. Dataflow match metric evaluates how closely the data flow structures of the generated code align with those of the reference code, considering the information passed and processed within each. A higher dataflow match score signifies a stronger similarity in data flow between the generated and reference code. The syntax match similarity evaluates the syntactic similarity of the generated code relative to the reference code. The syntactic match score measures the degree to which the generated code conforms to the syntactic conventions of the reference code. Between these two metrics, the dataflow match is more closely to the code correctness as it measures how the data is modified along the code lines and the logic behind the information flow. Although the syntax match is not highly correlated to code correctness, it can still capture the incomplete code generation and syntax errors which can also lead to an incorrect code generation.

The experimental results outline that our model achieved significantly higher average dataflow match scores across all benchmarks. Incorporating structure-aware loss consistently improved the average dataflow and syntax match scores across all benchmarks, except for a slight decrease in HumanEval (see Table 3).

Table 3: Performance Metrics for average syntax match and dataflow match

Benchmark	Model	Average syntax match	Average dataflow match
	Llama 3.1 8B	0.2183	0.3809
MBPP	Ours - only context distillation	0.2369	0.4489
	Ours - structure-aware context distillation	0.2427	0.4566
	Llama 3.1 8B	0.2738	0.5512
MBPP Plus	Ours - only context distillation	0.3259	0.6441
	Ours - structure-aware context distillation	0.3521	0.6503
	Llama 3.1 8B	0.3033	0.3099
HumanEval	Ours - only context distillation	0.3076	0.3602
	Ours - structure-aware context distillation	0.3019	0.3688

In addition, we carried out complementary analyses to investigate the effect of our fine-tuning on the understanding of questions. To do so, we divide the questions of the MBPP test set into two parts, one with 10% of tokens that was used to ask our model to complete the question given these tokens. The other part was used to calculate the perplexity of our fine-tuned model against the base model. As illustrated in Figure 1 (see the Appendix), with a lower mean perplexity of

19.82 compared to the 21.57 of the base model, it has a lower uncertainty about understanding the general context of the coding questions.

As shown in this example, our model has a more complete understanding of the problem to be solved, and unlike the base model, our model generates a code that considers both increasing and decreasing monotonic arrays (see the Appendix).

We also evaluated the performance of the base model when CoTs generated by the Llama 3.1 70B model are added directly as the MBPP questions' context and passed to the Llama 3.1 8B. The results of our experiment show that the base modllms-hallucinations-attentionel reaches a pass@1 of 64.7% when it uses the CoTs of very large models. In comparison, the Llama 3.1 70B model itself attains a pass@1 of 83%. Our analysis revealed that the Llama 3.1 8B model struggles to understand the intent of questions that feature complex phrasing and lacks the mathematical knowledge required to generate code involving mathematical formulas. While our fine-tuned model improves the general comprehension of a question's main intent, it does not significantly enhance the model's mathematical reasoning capabilities.

5 Conclusion

This work investigates the feasibility of distilling key capabilities of large language models—such as intent recognition, step-by-step reasoning, and handling edge cases—through cost-effective and parameter-efficient fine-tuning. We propose a novel loss function designed to simultaneously capture fine-grained token-level information and broader semantic and structural relationships, facilitating effective mappings between programming tasks and corresponding code solutions. Our experiments indicate consistent performance improvements over the base model across multiple metrics and benchmarks. Specifically, our analysis shows that the fine-tuned model better comprehends programming questions and can more accurately predict or complete questions given partial context compared to the base model. Additionally, we demonstrate that incorporating high-quality chains-of-thought (CoTs), generated by very large language models (VLLMs), significantly enhances small-model code generation performance. Furthermore, our results indicate that fine-tuning enables smaller models to internalize these high-quality contexts to some extent. Despite achieving notable improvements across several benchmarks, the fine-tuned small model still encounters difficulties in fully grasping complex question intent and accurately interpreting mathematical formulations required by certain MBPP tasks. Future research will focus on addressing these challenges and further exploring methods to distill richer context information from VLLMs into smaller models. We are also interested in investigating the possibility of distilling the context generated through a sequential collaboration of a set of agents based on the general framework of context distillation presented in the current work.

References

- [1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint* arXiv:2303.08774, 2023.
- [2] Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv* preprint arXiv:2301.03988, 2023.
- [3] Vito Walter Anelli, Giovanni Maria Biancofiore, Alessandro De Bellis, Tommaso Di Noia, and Eugenio Di Sciascio. Interpretability of bert latent space through knowledge graphs. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 3806–3810, 2022.
- [4] 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.
- [5] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.
- [6] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.
- [7] Tom B Brown. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.
- [8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

- [9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.
- [10] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38, 2024.
- [11] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [12] 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, 2020.
- [13] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv* preprint *arXiv*:2204.05999, 2022.
- [14] Jing Gong, Yanghui Wu, Linxi Liang, Zibin Zheng, and Yanlin Wang. Cosqa+: Enhancing code search dataset with matching code. *arXiv preprint arXiv:2406.11589*, 2024.
- [15] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. arXiv preprint arXiv:2306.11644, 2023.
- [16] Michael Hassid, Tal Remez, Jonas Gehring, Roy Schwartz, and Yossi Adi. The larger the better? improved llm code-generation via budget reallocation. *arXiv preprint arXiv:2404.00725*, 2024.
- [17] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.
- [18] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.
- [19] Gabriel Ilharco, Marco Tulio Ribeiro, Mitchell Wortsman, Suchin Gururangan, Ludwig Schmidt, Hannaneh Hajishirzi, and Ali Farhadi. Editing models with task arithmetic. *arXiv preprint arXiv:2212.04089*, 2022.
- [20] Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators. *arXiv preprint arXiv:2311.14904*, 2023.
- [21] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.
- [22] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.
- [23] Sawan Kumar and Partha Talukdar. Reordering examples helps during priming-based few-shot learning. *arXiv* preprint arXiv:2106.01751, 2021.
- [24] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv* preprint *arXiv*:2305.06161, 2023.
- [25] Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.
- [26] Jia Lio, Ge Li, Yongmin Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 2023.
- [27] Changshu Liu. Can large language models reason about code? arXiv preprint arXiv:2108.07732, 2024.
- [28] Yao Lu, Max Bartolo, Alastair Moore, Sebastian Riedel, and Pontus Stenetorp. Fantastically ordered prompts and where to find them: Overcoming few-shot prompt order sensitivity. *arXiv preprint arXiv:2104.08786*, 2021.
- [29] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. arXiv preprint arXiv:2306.08568, 2023.
- [30] Yingwei Ma, Yue Yu, Shanshan Li, Yu Jiang, Yong Guo, Yuanliang Zhang, Yutao Xie, and Xiangke Liao. Bridging code semantic and llms: Semantic chain-of-thought prompting for code generation. *arXiv* preprint *arXiv*:2310.10698, 2023.
- [31] Lucie Charlotte Magister, Jonathan Mallinson, Jakub Adamek, Eric Malmi, and Aliaksei Severyn. Teaching small language models to reason. *arXiv preprint arXiv:2212.08410*, 2022.

- [32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv* preprint arXiv:2203.13474, 2022.
- [33] Matthew E Peters, Sebastian Ruder, and Noah A Smith. To tune or not to tune? adapting pretrained representations to diverse tasks. *arXiv preprint arXiv:1903.05987*, 2019.
- [34] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.
- [35] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.
- [36] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.
- [37] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. Code less, align more: Efficient llm fine-tuning for code generation with data pruning. *arXiv preprint arXiv:2407.05040*, 2024.
- [38] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- [39] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint* arXiv:2206.07682, 2022.
- [40] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.
- [41] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.
- [42] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. *URL https://arxiv. org/pdf/2305.10601. pdf*, 2023.
- [43] Lui Yoshida. The impact of example selection in few-shot prompting on automated essay scoring using gpt models. In *International Conference on Artificial Intelligence in Education*, pages 61–73. Springer, 2024.
- [44] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2023.
- [45] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.

Appendices

We conducted additional analyses beyond pass@1 to better understand our model's behavior relative to the Llama 3.1 8B baseline. Our model demonstrates a clearer understanding of the questions and shows reduced confusion during token generation.

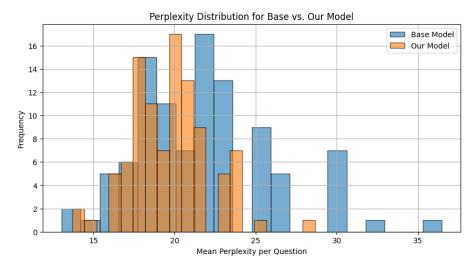


Figure 1: The perplexity of our model in understanding the coding question compared to base model.

```
An example of MBPP's question and both model's answers

Question:

Write a Python function to check whether the given array is monotonic or not.

Base model's code:

def is_monotonic(arr):
    if arr[i] < arr[i - 1]:
        return False

return True

Our model's code:

def is_monotonic(arr):
    return (
        all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1)) or
        all(arr[i] >= arr[i + 1] for i in range(len(arr) - 1))

)
```

Figure 2 shows the perplexity of our fine-tuned model against the base model while generating the code for one of the MBPP's questions. In general our model is less confused during the token generation. For the initial tokens both models have similar perplexities likely due to the fact that the initial tokens of a python functions are always the same when we explicitly ask the model to create a function that solve a coding problem.

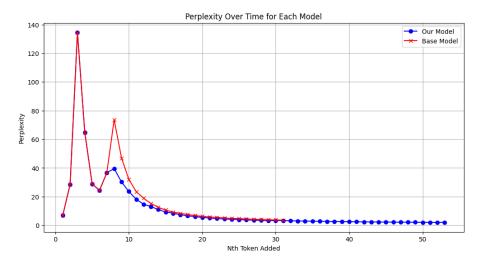


Figure 2: The perplexity of our model while generating the code for a given MBPP question compared to the base model.