At Which Training Stage Does Code Data Help LLMs Reasoning?

Anonymous EMNLP submission

Abstract

Large Language models (LLMs) have exhibited remarkable reasoning capabilities and become the foundation of language technologies. Inspired by the great success of code data in training LLMs, we naturally wonder at which training stage introducing code data can really help LLMs reasoning. To this end, this paper systematically explores the impact of code data on LLMs at different stages. Concretely, we introduce the code data at the pre-training stage, instruction-tuning stage, and both of them, respectively. Then, the reasoning capability of LLMs is comprehensively and fairly evaluated via six reasoning tasks. We critically analyze the experimental results and provide conclusions with insights. First, pre-training LLMs with the mixture of code and text can significantly enhance LLMs' general reasoning capability almost without negative transfer on other tasks. Besides, at the instruction-tuning stage, code data endows LLMs the task-specific reasoning capability. Moreover, the dynamic mixing strategy of code and text data assists LLMs to learn reasoning capability step-by-step during the training process. These insights deepen the understanding of LLMs regarding reasoning ability for their application, such as scientific question answering, legal support, etc. The source code and model parameters are released at the anonymous link: `https://anonymous.4open.science/r/CodeLLM-FD25/`.

1 Introduction

Recently, Large Language Models (LLMs) have achieved impressive generalization performance across various tasks. Significantly, OpenAI developed ChatGPT (OpenAI, 2023a), Google designed PaLM (Chowdhery et al., 2022), Baidu ERNIE built Bot (Baidu, 2023), and Alibaba presented Tongyi Qianwen (Alibaba, 2023). However, these industrial products are regrettably not open-source for commercial reasons. Thanks to the surging open-source projects of LLMs such as LLaMA (Touvron et al., 2023), Alpaca (Taori et al., 2023), and GLM (Du et al., 2022a), the academical researches and industrial products of LLMs mark new milestones.

Two of the key factors to the great success of LLMs are 1) training data and 2) training strategies. First, for the training data, researchers aim to endow LLMs with language capabilities and general knowledge via training models on large-scale data from various domains. For example, LLaMA was trained with 1.4 trillion tokens, which consist of texts (CommonCrawl, C4) and codes (GitHub). These large-scale data with diversity help the model to achieve competitive performance on multiple tasks. Second, for the training strategies, the common pipeline go through two stages: pre-training and instruction-tuning. The pre-training is conduct in self-supervised manner on the massive unlabeled data while instruction-tuning aims to fine-tune models with human-annotated prompts and feedback (Ouyang et al., 2022). Benefiting from the data and training strategies, LLMs gain remarkable skills, such as translation, conversation, examination, legal support, etc. These skills are all based on one of the most important capability, i.e., reasoning capability. So, how can LLMs gain the such strong reasoning capability?

We analyze the reasons from two aspects including the training data and training strategies. First, from the training data aspect, compared with the common textual data, code data is more logical and less ambiguous (refer to case studies in Appendix D). Also, from the experiments, researchers (Liang et al., 2022; Fu and Khot, 2022) verified that models trained on code data have strong reasoning capability. Therefore, code data is essential for model reasoning. Second, for the training strategies, both pre-training and fine-tuning are crucial to the model performance. Pre-training feeds general knowledge to models while fine-tuning feeds domain-specific knowledge to models. To further explore the deep-in reasons of the strong reasoning capability of LLMs, this paper aims to answer an important question: at which stage does code data help LLMs reasoning?

To this end, we conduct comprehensive and fair experiments and provide analyses and conclusion with insights. First, we pre-train LLMs with pure text data and mixture data of code and text, respectively. Subsequently, at the instruction-tuning stage, LLMs is fine-tuned with the pure text data and mixture data of code and text, respectively. After training, to comprehensively measure the model reasoning capability, we evaluate LLMs on six tasks in five logical reasoning, code reasoning, legal reasoning, scientific reasoning, and analogical reasoning. Based on extensive experimental results and analyses, we provide four insights. 1) Pre-training LLMs with the mixture of code and text can significantly enhance LLMs' general reasoning capability almost without negative transfer on other tasks. 2) At the instruction-tuning stage, code data endows LLMs the task-specific reasoning capability. 3) The dynamic mixing strategy of code and text data assists LLMs to learn reasoning capability step-by-step during the training process. These findings deepen the understanding of LLMs regarding reasoning ability for their applications, such as scientific question answering, legal support, etc. The main contributions of this work are summarized as follows.

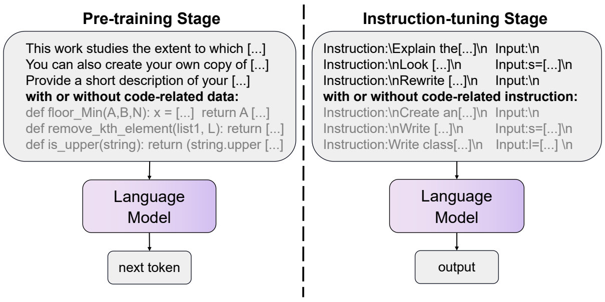
- Research question: this paper raises and aims to answer one essential concern, i.e., at which training stage can codes help LLMs reasoning.

- Analyses and insights: we conduct extensive experiments and provide critical analyses and insights, which deepen the understanding of LLMs regarding reasoning capability.

- Open-source resource[1]: we release the model implementation and the trained model parameters, which contribute to the further researches in LLMs community.

2 Dataset

In this paper, we study two training phases of LLM—-pre-training and instruction-tuning-—on two different datasets (a plain text dataset and a dataset containing code) with the same parameter size and using the same training goals. See Figure 1 for examples of each setup. Specifically, we use the open-sourced PanGu2.6B and PanGu13B of the PanGu-α team (Zeng et al., 2021) as baseline models for text models (trained on 100GB text data and larger text data, respectively), and train Code-PanGu2.6B from scratch on the mixed code data for comparison. We will introduce detailed data in later chapters.



This work studies the extent to which [...]
You can also create your own copy of [...]
Provide a short description of your [...]
with or without code-related data:
def floor_Min(A,B,N): x = [...] return A [...]
def remove_kth_element(list1, L): return [...]
def is_upper(string): return (string.upper [...]

Instruction:\Explain the[...]\n Input:\n
Instruction:\nLook [...]\n Input:s=[...]\n
Instruction:\nRewrite [...]\n Input:\n
with or without code-related instruction:
Instruction:\nCreate an[...]\n Input:\n
Instruction:\nWrite [...]\n Input:s=[...]\n
Instruction:Write class[...]\n Input:l=[...] \n

Figure 1: Examples of pre-training and tuning phase.

2.1 Pre-Training Corpus

The training corpus consists of two parts. To ensure a fair comparison with PanGu2.6B, we collected a large amount of original data from public data sets such as BaiDuQA, CAIL2018, Sogou-CA, and network data sets such as Common Crawl, encyclopedias, news, and e-books according to the PanGu-α team (Zeng et al., 2021). Then we use rule-based data cleaning and model-based data filtering methods to filter to ensure high quality. Finally, we obtain 100GB of text data with the same scale and source as PanGu2.6B by sampling each data source using different ratios. Please refer to Appendix E for a detailed data processing process. To verify the influence of code data on the reasoning ability of the model in the pre-training stage, we used the CodeParrot (Huggingface, 2023) dataset as the second supplementary part. CodeParrot is a public Python dataset from BigQuery, comprising approximately 50GB of code and 5,361,373 files. Figure 2 shows the composition of the 42B token training data.

2.2 Instruction-Tuning Corpus

We collect and construct 500k instruction tuning data to verify the effect of adding code instructions in the instruction tuning stage and convert them into a unified instruction format. The instruction tuning corpus is divided into two parts. The first part is from the natural language open source instruc-

[1] https://anonymous.4open.science/r/CodeLLM-FD25/

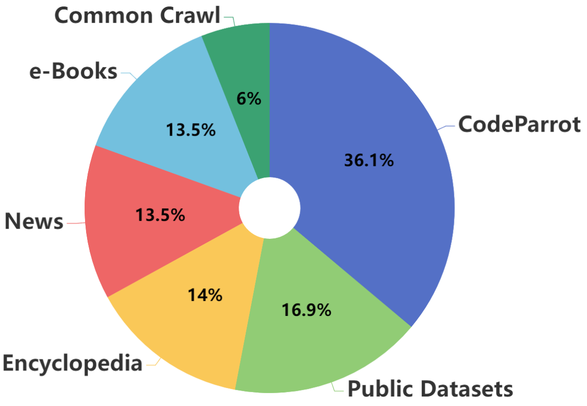
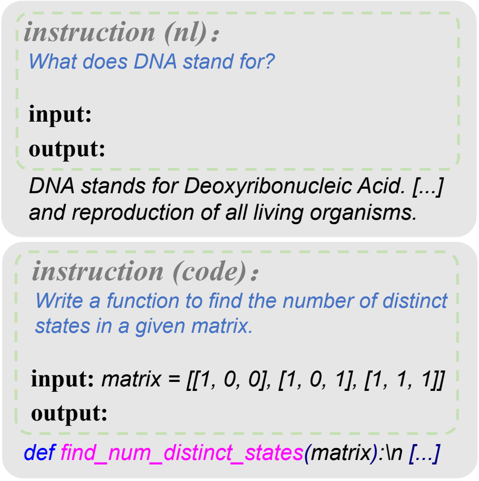


Figure 2: Distribution of the training data.



instruction (nl):
What does DNA stand for?

input:
output:
DNA stands for Deoxyribonucleic Acid. [...] and reproduction of all living organisms.

instruction (code):
Write a function to find the number of distinct states in a given matrix.

input: *matrix = [[1, 0, 0], [1, 0, 1], [1, 1, 1]]*
output:

```
def find_num_distinct_states(matrix):\n [...]
```

Figure 3: Example of the instruction tuning data format.

tion dataset, Alpaca-GPT-4 (Peng et al., 2023) and PromptCLUE (pCLUE team, 2022). Alpaca-GPT-4 is generated by GPT-4, including 52k Chinese and English instruction tuning data. PromptCLUE unifies the differences between different NLP tasks (*e.g.*, reading comprehension, question answering) and converts the original task training set into a unified text-to-text data form, from which we randomly sample 200k data for instruction tuning.

The second part comes from the open-source data CodeAlpaca (Chaudhary, 2023) and our build dataset, with 150k instructions. The CodeAlpaca data contains 20k instruction tuning data generated according to the self-instruct technology, which can be used for instruction tuning of the code generation model. In order to supplement the code-related instruction tuning data, we use the CosQA (Huang et al., 2021) training set and the MBPP (Austin et al., 2021) training set to unify the task format in the way of PromptCLUE and expand the CodeAlpaca data. Figure 3 is an example of the format of instruction tuning data.

3 Model

We conduct experiments on large-scale autoregressive language models by adopting the GPT paradigm(Brown et al., 2020). It iteratively takes all tokens in the corpus as input, predicts the next token, and compares it to the ground truth. Assuming that a sequence $X = x_1, x_2, ..., x_N$ is composed of N tokens, the training objective can be formulated as maximization of the log-likelihood:

$$\mathcal{L} = \sum_{i=1}^{N} \log p(x_n|x_1, ..., x_{n-1}; \Theta) \qquad (1)$$

where $p(x_n|x_1, ..., x_{n-1}; \Theta)$ is the probability of observing the $n - th$ token x_n given the previous context $x_{1:n-1}$, and Θ denotes the model parameters.

3.1 Model Architecture

Similar to recent pre-trained models such as GPT-3 (Brown et al., 2020), LLaMA (Touvron et al., 2023), and PANGU-α (Zeng et al., 2021), we follow a generative pre-training (GPT) architecture for autoregressive language modeling. As shown in Figure 4, the core architecture of the model is a 32-layer transformer decoder. The original GPT model uses a pooler function to obtain the final output. We use an additional query layer on top of the stacked Transformer layers to explicitly induce the expected output with attention to obtain the final embedding.

3.2 Tokenization

For the text-only model, we use the open-source vocabulary of the PanGu2.6B model released by PanGu-α team (Zeng et al., 2021), and the size of the vocabulary is 40,000. For the model training with mixed code, considering that there may be variables, functions, and class names in the code that are often meaningful words, we use the ChatGLM (Du et al., 2022b) vocabulary open-sourced by the THUGLM team to encode text and the code. The vocabulary size is 130,044. In addition, ChatGLM encodes multiple spaces as extra tokens to improve encoding efficiency. Specifically, L spaces are represented by <|extratoken_X|>, where X=8+L. Both vocabularies are BPE-based tokenizers, which use fixed-size vocabularies to handle variable-length characters in open-vocabulary problems.

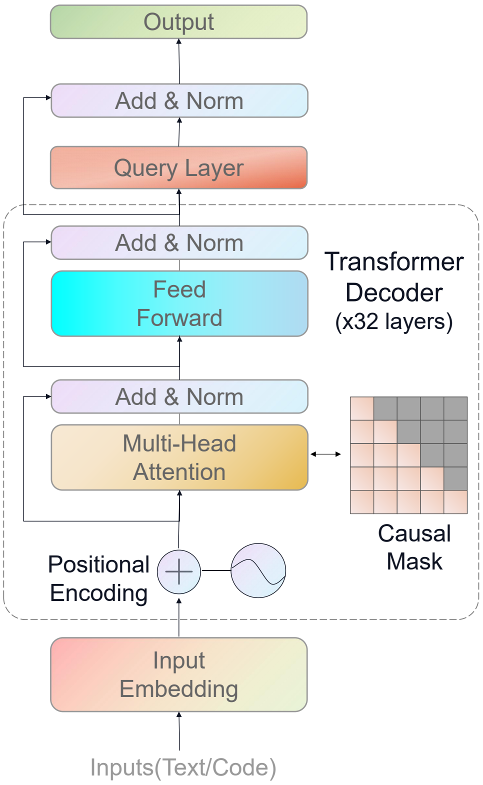


Figure 4: Model architecture. We build a model with 2.6B parameters, consisting of 32-layer left-to-right transformer decoders and a top query layer.

4 Experiments

4.1 Task Description

To measure the reasoning ability of the model, we evaluate it in realistic reason-centric scenarios, including logical reasoning, code reasoning, legal reasoning, scientific reasoning, etc. These reasoning-intensive tasks elucidate the reasoning capabilities of the model through the model's performance in these scenarios. When publicly available, we evaluate the models with the test sets for each task. Otherwise, we use the development sets instead. We describe each task as follows.

Logical Reasoning. Logic is the study of reasoning and argumentation, which focuses on the rules of logic and methods of reasoning in the thinking process. We use the **logic** subject in the **C-Eval** dataset (Huang et al., 2023) to determine whether the model can understand and apply logical rules to make reasonable reasoning.

Code Reasoning. We use **CosQA** (Huang et al., 2021) to test the model performance on the code question-answering task. The dataset includes 604 natural language-code question-answer pairs. Furthermore, we use the **MBPP** dataset (Austin et al., 2021) to test the model code generation ability, containing 427 Python coding questions.

Legal Reasoning. For legal reasoning, we use **JEC-QA** (Zhong et al., 2020), the largest question answering dataset in the legal domain, collected from the National Judicial Examination of China. The examination is a comprehensive evaluation of the professional skills of legal practitioners. Multiple reasoning skills are required to retrieve relevant material and answer legal questions.

Scientific Reasoning. We use the **ScienceQA** dataset (Lu et al., 2022) to evaluate the scientific reasoning ability of the model. The scientific question answering task can diagnose whether the artificial intelligence model has multi-step reasoning ability and interpretability. To answer scientific questions from ScienceQA, a model not only needs to understand multimodal content but also needs to extract external knowledge to arrive at the correct answer.

Analogical Reasoning. We use the **E-KAR** dataset (Chen et al., 2022) to evaluate the model's analogical reasoning ability. It comes from the Civil Service Examination, a comprehensive test of the candidate's critical thinking and problem-solving ability. To solve the analogy reasoning problem, candidates need to understand the relationship among the options, which requires specific reasoning ability and background knowledge, especially common sense and facts, and knowing why a fact is denied.

4.2 Evaluation Details

In evaluation, these tasks are usually divided into two parts, understanding task and generation task. For the understanding task, we follow PanGu2.6B (Zeng et al., 2021), decomposing the task into a perplexity comparison task. We construct a prompt template for each evaluation task and populate the template with instances as input to the model. Table 1 describes the templates for each task, where "/" indicates that the task does not involve templates.

We adopt a perplexity-based approach to solve classification tasks. For each <text, label> pair, input will be automatically generated according to the predesigned prompt in Table 1. The sequences generated by the prompt will be fed into the model, and a perplexity value will be calculated. The label corresponding to the minimum perplexity value will be regarded as the predicted label for this passage.

Task	Dataset	Input&Prompt
Logical	Logic	The answer: $choice, can answer the following questions: $problem
Legal	JEC-QA	The answer: $choice, can answer the following questions: $problem
Scientific	ScienceQA	$lecture\n anwser:$choice can answer the following question:$question
Analogical	E-KAR	The reasoning relationship:$r1, the analogy reasoning relationship:$r2
Code	CosQA	$question? Answered code is correct/wrong: $code
Code	MBPP	$question\n Code:\n

Table 1: The input&prompt template for each task.

Dataset	Task Types	Metrics	PanGu2.6B	PanGu13B	CodePanGu2.6B
Logic	Logical Reasoning	Acc	36.36	**45.45**	40.90
JEC-QA	Legal QA	Acc	27.00	27.00	**28.70**
ScienceQA	Scientific QA	Acc	45.93	45.18	**46.06**
E-KAR	Analogical Reasoning	Acc	32.24	35.52	**36.12**
CosQA	Code QA	Acc	47.01	46.85	**50.50**
MBPP	Code Generation	BLEU	0.52	1.34	**5.06**

Table 2: Results on pre-training stage.

For the generative task, we leverage the properties of autoregressive language models to generate corresponding answers directly from a given input naturally.

4.3 Results

4.3.1 Pre-training Stage

To illustrate the impact of code data in the pre-training phase on the reasoning capabilities of large language models, we compared the performance of the three models in real reasoning-intensive scenarios. Among them, the PanGu2.6B and PanGu13B models (Zeng et al., 2021) are trained on natural language datasets, and the CodePangu2.6B model is trained on mixed data (the dataset mentioned in Chapter 2.1). The models are evaluated in zero-shot manner on downstream tasks. Specifically, we report accuracy on for Logic, JEC-QA, ScienceQA, E-KAR, and CosQA tasks and BLEU score for MBPP task. Figure 2 depicts the results of these tasks. Consistently over these tasks, we observe the following:

- After adding code training, LLM performs better on most reasoning-related tasks, even though most of these tasks are not related to code. This shows that adding code data in the pre-training stage can not only improve the coding-related ability but also improve the general language reasoning ability of the model to a certain extent.

- Even with a larger scale model, *i.e.*, PanGu13B, it is still not as effective as CodePanGu2.6B in these reasoning-centric scenarios. This is similar to the results of HELM (Liang et al., 2022), which suggest that if (a) the computational budget is constrained and (b) the resulting model is applied in the code/reasoning domain, adding code data in the pre-training phase may be more effective than increasing the model parameter size.

In summary, we find that simply adding code data during the pre-training phase can effectively improve the model's general reasoning ability, which might indicate that mixing more code data for training may produce a competitive model to solve tasks that require complex reasoning to complete. This provides a promising prospect for subsequent LLM development.

4.3.2 Instruction-tuning Stage

ChatGPT (OpenAI, 2023a) and GPT-4 (OpenAI, 2023b) successfully use instruction tuning to enable LLMs to follow natural language instructions and complete real-world tasks; this improvement has become standard in open-source LLMs. This is implemented by fine-tuning the model on a wide range of tasks using human-annotated instructions and feedback, by supervised fine-tuning via manually or automatically generated instructions using public benchmarks and datasets, or learning from instruction-following data by developing from state-of-the-art instruction-tuned teacher LLMs.

Dataset	Task Types	Metrics	nl_nl	nl_code	code_code
Logic	Logical Reasoning	Acc	36.36	**40.90**	**40.90**
JEC-QA	Legal QA	Acc	25.20	26.10	**27.10**
ScienceQA	Scientific QA	Acc	**44.45**	43.44	41.90
E-KAR	Analogical Reasoning	Acc	**30.45**	28.66	27.20
CosQA	Code QA	Acc	45.20	48.18	**52.48**
MBPP	Code Generation	BLEU	0.00	5.61	**24.88**

Table 3: Results on Instruction-tuning stage.

To illustrate the impact of code data on the LLMs reasoning ability in the instruction tuning stage, we use the instruction tuning datasets that contain codes and the instruction tuning datasets without codes introduced in chapter 2.2 to fine-tune the PanGu2.6B model (Zeng et al., 2021) and evaluate their performance in reasoning-intensive scenarios. In addition, we also fine-tune the CodePanGu2.6B model using the instruction tuning dataset containing codes to observe the effect of using code data in both pre-training and instruction tuning stages. Figure 3 shows the results of these tasks. Among them, *nl_nl* and *nl_code* represent the fine-tuned model of PanGu2.6B using only text instructions and instructions containing codes, respectively, and *code_code* represents the fine-tuning model of CodePanGu2.6B using instructions containing codes. Consistently over these tasks, we observe the following:

- After fine-tuning with mixed code instruction data, LLM shows different trends in multiple reasoning-related tasks. This indicates that introducing code data in the instruction tuning phase may be less effective than in the pre-training phase. Therefore, it is best to add code data in the pre-training stage to improve the model performance in general reasoning tasks.

- We find that training with code data in both stages can significantly improve code-related tasks, especially code generation tasks. This may be because the code instruction data activates the code reasoning ability of the language model, which suggests that if the LLM needs to complete complex code tasks, the code reasoning ability can be improved by effectively following instructions and generating compliant content.

- Compared with the pre-training stage, the performance of instruction-tuned LLMs on some tasks is degraded, similar to the TÜLU (Wang et al., 2023) results. This may be because the instruction tuning data usually covers a wide range of domains and dialogue content, causing the model to tend to answer questions more comprehensively, resulting in a decline in reasoning ability. We propose that if specific reasoning capabilities are required, they can be augmented by adding domain-specific instruction data during the instruction tuning phase.

In summary, we find that adding code data in the instruction tuning stage is not as effective as the pre-training stage in improving the general reasoning ability of the model. However, we find that code instructions made the model follow natural language instructions and generate correct code, improving the model's code reasoning ability. This also suggests that tuning with relevant data may be helpful when solving specific reasoning tasks.

4.3.3 Other Tasks

We have evaluated the impact of code data on the reasoning ability of LLMs at different training stages in realistic reasoning-centric scenarios. Furthermore, to verify the impact of code data on other comprehension and generation tasks that are less demanding on reasoning, we conduct experiments on other tasks, including two NLI tasks (OCNLI (Hu et al., 2020) and CMNLI (Wang et al., 2018)), requiring the model to identify the relationship between two sentences, either entailment, neutral or contradiction; a free-form multiple-choice Chinese machine reading comprehension dataset (C^3) (Sun et al., 2020) consisting of documents (conversational or more formal mixed-type text) and their associated multiple-choice free-form questions; one reading comprehension task duReader (He et al., 2017), requiring the model to extract a text span from a given paragraph as the correct answer to the question. Refer to Appendix

Dataset	Metrics	w/o.code	w.code
pre-training			
C^3	Acc	54.14	**54.30**
OCNLI	Acc	**41.69**	40.50
CMNLI	Acc	**45.07**	43.49
DuReader	Em/F1	**0.42/15.29**	0.14/8.73
instruction-tuning			
C^3	Acc	**55.07**	54.47
OCNLI	Acc	40.78	**41.19**
CMNLI	Acc	44.82	**45.49**
DuReader	Em/F1	**12.07/34.85**	8.05/25.05

Table 4: Results on Other Tasks.

B for prompt templates and evaluation metrics for different tasks.

Table 4 shows the results of adding code data in the pre-training phase and adding code instructions in the instruction tuning phase (only in this phase). Experimental results show that, in most cases, adding code data at both stages has little negative impact on the performance of other tasks and even produces some benefits. In the DuReader reading comprehension task, part of the performance will be reduced after adding code at different stages. This may be because the model does not thoroughly learn the code and text data, resulting in confusion when the model generates answers to reading comprehension questions. In the future, we will verify and solve it in a larger model and with larger data.

4.3.4 Chain-of-Thought Ability

Compared with the standard prompt technology, Chain-of-Thought (CoT) (Wei et al., 2022) transforms tasks into a continuous chain generation process. This technology enhances the model ability in complex reasoning tasks by providing a language model with a series of related reasoning steps. To evaluate the potential of the model in utilizing chains of thought in solving complex problems, we conduct experiments on two models, PanGu2.6B and CodePanGu2.6B on ScienceQA(CoT) (Lu et al., 2022) and E-KAR(CoT) (Chen et al., 2022) datasets. We incorporate CoT information as a part of the model input with the question and context information. In this way, the model can directly use the reasoning process of the thinking chain for answer generation. The experimental results are shown in Figure 5.

The experimental results show that after the introduction of the Chain-of-Thought, the performance of all models in reasoning problems is significantly improved by making full use of the coherent reasoning process of CoT. The CoT information is used as part of the model input to help the model better understand the problem and generate answers according to the logic of the CoT. Among them, CodePanGu2.6B achieved the best performance, indicating that CodePanGu2.6B can better use CoT information for reasoning. This also suggests that pre-training with mixed-code data may result in a competitive model for tasks that require complex reasoning.

Model	Dataset	w/o.cot	w.cot
PanGu2.6B	ScienceQA	45.93	68.76
CodePanGu2.6B	ScienceQA	46.06	**70.30**
PanGu2.6B	E-KAR	32.24	69.55
CodePanGu2.6B	E-KAR	36.12	**72.84**

Table 5: Results on Chain-of-Thought data.

4.3.5 Exploring Ways to Mix Code and Text Data

Previous experiments have demonstrated that training with mixed code data in the two stages of pre-training and instruction tuning can improve the general and specific reasoning capabilities of LLMs, respectively. Therefore, We naturally wonder how mixing these two types of data can better improve model reasoning ability, which has not been explored in previous studies. Therefore, we design comparative experiments in the instruction tuning stage to verify the impact of different data mixing strategies. The mixed strategy is shown in Table 6. One group is uniform sampling, that is, the proportion of text and code in each group of training data is roughly the same; the other two groups gradually increase or decrease the proportion of code to verify whether step-by-step learning will better activate the reasoning ability of LLMs. The experimental results are shown in table 7.

Phase	Uniform Sampling	Stepwise Increase	Stepwise Decrease
1	5:3	7:3	5:5
2	5:3	7:3	6:4
3	5:3	6:4	7:3
4	5:3	5:5	7:3

Table 6: Mixed strategies of text and code (number of text: number of codes))

The experiment found that the training strategy of using a higher code data ratio in the early stage and gradually reducing the code data ratio in the

Dataset	Uniform Sampling	Stepwise Increase	Stepwise Decrease
Logic	31.82	36.36	**40.90**
JEC-QA	**27.30**	26.70	27.10
ScienceQA	**43.76**	43.19	41.90
E-KAR	**28.66**	28.36	27.20
CosQA	51.65	50.66	**52.48**
MBPP	23.68	23.42	**24.88**

Table 7: Result of different mixed strategies.

later stage achieved the best results in code question answering and code generation tasks, while ensuring the performance of the model in other reasoning tasks. This may be because, due to the strong logic of the code, using more code data in the early stage may help the model activate the code reasoning ability faster. Therefore, if LLMs are expected to have better specific reasoning ability, adopting a stepwise descent strategy can better activate the model potential. In addition, since experiments in the pre-training phase require a lot of resources, we leave the validation of this phase to later work.

5 Related Work

LLM training. Self-attention-based transformer networks have recently brought essential improvements, especially in capturing long-range dependencies (Vaswani et al., 2017; Radford et al., 2018; Dai et al., 2019). LLM is usually based on the transformer architecture. Notable models include BERT (Devlin et al., 2018), GPT-2 (Radford et al., 2019), and T5 (Raffel et al., 2020); after the emergence of GPT-3 (Brown et al., 2020) with 175B parameters, a batch of larger models emerged, including PaLM (Chowdhery et al., 2022), OPT (Zhang et al., 2022), PanGu-α (Zeng et al., 2021), and LLaMA (Touvron et al., 2023), which have achieved remarkable results on various NLP tasks. For LLMs to follow instruction output, instruction tuning plays an important role. This can use human-annotated cues and feedback (Ouyang et al., 2022) or public benchmarks to automatically generate instructions (Wang et al., 2022b; pCLUE team, 2022) to fine-tune models on various tasks. Moreover, Self-Instruct tuning (Wang et al., 2022a; Peng et al., 2023) is a simple and effective way to generate instruction-following data through state-of-the-art teacher LLMs for fine-tuning.

Data Mixtures. Models such as GPT-3 (Brown et al., 2020) and PanGu-α (Zeng et al., 2021) are trained on natural language data from various domains, and models such as LaMDA (Thoppilan et al., 2022) and LLaMA (Touvron et al., 2023) are additionally trained on code data. However, the impact and specific origin of this mixed-code data is unclear. Some researchers have extensively analyzed the performance of current LLM on various tasks, pointing out that code may be the key to improving reasoning ability (Liang et al., 2022; Fu and Khot, 2022). However, the evaluated models have different parameters and data scales, and problems such as unknown training details exist. It is difficult to determine the exact impact of code data on the reasoning ability of LLMs.

6 Conclusion

In this paper, we investigate at which stage introducing code data can help improve the reasoning ability of LLMs. We validate the effect of code data at different stages with the same parameter scale and using the same training objective. We point out that simply adding code data in the pre-training phase can effectively improve the general reasoning ability of the model. Furthermore, we find that adding code instructions in the instruction tuning stage can make the model follow human instructions for output and improve specific code reasoning capabilities. Moreover, we point out that the dynamic mixing strategy of code and text data assists LLMs in learning reasoning capability step-by-step during the training process. We provide a well-designed and tested reference implementation for LLMs training to help researchers and developers better understand and analyze LLMs.

There are many potential directions for further research in this area. On the one hand, one can explore how to mix code training in the pre-training stage to improve the general reasoning ability of LLMs better. On the other hand, this paper explores at which training stage code data helps LLMs reason. Still, code data may only be one of the sources of model reasoning ability, and exploring more sources of reasoning ability can develop LLMs with more general capabilities.

Limitations

In this paper, we conduct an in-depth study on the influence of code data on the reasoning ability of LLM at different stages and summarize them from different perspectives from our own thoughts.

However, our pre-training experiments are all based on the 2.6B model. Due to resource constraints, training larger-scale models for comparative experiments is impractical. Although the conclusions of this paper have certain reference significance, they still cannot guarantee the generalization of larger models. In addition, ChatGPT (OpenAI, 2023a) also uses reinforcement learning to better adapt to the final task and user preferences. Since both the reinforcement learning phase and the instruction tuning phase aim to align the language model, we only performed supervised fine-tuning according to the practices of open-source models such as Stanford Alpaca (Taori et al., 2023) and CodeGeeX (Zheng et al., 2023). The exact impact on the reinforcement learning phase will be discussed in the follow-up work. Besides, since current LLMs are usually multilingual models, we may ignore the impact of code data on the reasoning ability of different language tasks. In order to alleviate this problem, we included Chinese (*e.g.*, JEC-QA (Zhong et al., 2020)) and English (*e.g.*, ScienceQA (Lu et al., 2022)) test tasks in the experiment. The current performance results are consistent, and we will conduct more language tests in the future.

References

Alibaba. 2023. Tongyi qianwen. https://tongyi.aliyun.com/.

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

Baidu. 2023. Ernie bot. https://yiyan.baidu.com/welcome.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Sahil Chaudhary. 2023. Code alpaca: An instruction-following llama model for code generation. https://github.com/sahil280114/codealpaca.

Jiangjie Chen, Rui Xu, Ziquan Fu, Wei Shi, Zhongqiao Li, Xinbo Zhang, Changzhi Sun, Lei Li, Yanghua Xiao, and Hao Zhou. 2022. E-KAR: A benchmark for rationalizing natural language analogical reasoning. In *Findings of the Association for Computational Linguistics: ACL 2022*, pages 3941–3955, Dublin, Ireland. Association for Computational Linguistics.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022a. GLM: general language model pretraining with autoregressive blank infilling. pages 320–335.

Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. 2022b. Glm: General language model pretraining with autoregressive blank infilling. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 320–335.

Hao Fu, Yao; Peng and Tushar Khot. 2022. How does gpt obtain its ability? tracing emergent abilities of language models to their sources. *Yao Fu's Notion*.

Wei He, Kai Liu, Jing Liu, Yajuan Lyu, Shiqi Zhao, Xinyan Xiao, Yuan Liu, Yizhong Wang, Hua Wu, Qiaoqiao She, et al. 2017. Dureader: a chinese machine reading comprehension dataset from real-world applications. *arXiv preprint arXiv:1711.05073*.

Hai Hu, Kyle Richardson, Liang Xu, Lu Li, Sandra Kuebler, and Larry Moss. 2020. Ocnli: Original chinese natural language inference. In *Findings of EMNLP*.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. 2021. CoSQA: 20,000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5690–5700, Online. Association for Computational Linguistics.

Yuzhen Huang, Yuzhuo Bai, Zhihao Zhu, Junlei Zhang, Jinghan Zhang, Tangjun Su, Junteng Liu, Chuancheng Lv, Yikai Zhang, Jiayi Lei, Yao Fu, Maosong Sun, and Junxian He. 2023. C-eval: A multi-level multi-discipline chinese evaluation suite for foundation models. *arXiv preprint arXiv:2305.08322*.

Huggingface. 2023. codeparrot. https://huggingface.co/codeparrot.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110*.

Pan Lu, Swaroop Mishra, Tony Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. 2022. Learn to explain: Multimodal reasoning via thought chains for science question answering. In *The 36th Conference on Neural Information Processing Systems (NeurIPS)*.

OpenAI. 2023a. Chatgpt. https://openai.com/blog/chatgpt.

OpenAI. 2023b. Gpt-4. https://openai.com/gpt-4.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

pCLUE team. 2022. pclue:large-scale prompt-based dataset for multi-task and zero-shot learning in chinese.

Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277*.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551.

Kai Sun, Dian Yu, Dong Yu, and Claire Cardie. 2020. Investigating prior knowledge for challenging chinese machine reading comprehension. *Transactions of the Association for Computational Linguistics*, 8:141–155.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

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. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A Smith, Iz Beltagy, et al. 2023. How far can camels go? exploring the state of instruction tuning on open resources. *arXiv preprint arXiv:2306.04751*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022a. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Anjana Arunkumar, Arjun Ashok, Arut Selvan Dhanasekaran, Atharva Naik, David Stap, et al. 2022b. Benchmarking generalization via in-context instructions on 1,600+ language tasks. *arXiv preprint arXiv:2204.07705*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*.

Wei Zeng, Xiaozhe Ren, Teng Su, Hui Wang, Yi Liao, Zhiwei Wang, Xin Jiang, ZhenZhang Yang, Kaisheng Wang, Xiaoda Zhang, et al. 2021. Pangu-α: Large-scale autoregressive pretrained chinese language models with auto-parallel computation. *arXiv preprint arXiv:2104.12369*.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x.

Haoxi Zhong, Chaojun Xiao, Cunchao Tu, Tianyang Zhang, Zhiyuan Liu, and Maosong Sun. 2020. Jec-qa: A legal-domain question answering dataset. In *Proceedings of AAAI*.

A Experiments Details

Our experiments are developed under the Mindspore framework. In the pre-training stage, we trained CodePanGu2.6B on a cluster of 16 Ascend 910 AI processors, and in the instruction-tuning stage, we tuned models on a cluster of 8 Ascend 910 AI processors. The sequence length for the training data is set to 1024 for all the models. Other detailed configurations can be found in Table 8.

Parameter	Value
Environment Parameter	
Framework	Mindspore v1.7.0
Hardwares	Ascend 910
Mem per GPU	32GB
GPUs per node	8
Model Parameter	
Layers	32
Hidden size	2560
FFN size	10240
Heads	32
Optimization Parameter	
Optimizer	Adam
Initial/final learning rate	1e-4(2e-5)/1e-6
Warm-up step	500
Learning rate scheduler	cosine
Optimizer parameters	$\beta1 = 0.9, \beta2 = 0.95$
Parallelism Parameter	
Data parallel	16(8)
Model parallel	1
pipeline parallel	1

Table 8: Training configurations.(The values in parentheses are instruction-tuning parameters)

B The Template for Other Tasks

We follow Chapter 4.2, conduct experiments on other tasks to verify the impact of code data on other comprehension and generation tasks that are less demanding on reasoning, including C^3 (Sun et al., 2020); two NLI tasks (OCNLI (Hu et al., 2020) and CMNLI (Wang et al., 2018)); one reading comprehension task duReader (He et al., 2017). Table 9 shows the prompt templates for these tasks. The evaluation metrics for duReader, including F1 and exact match(EM), measure the similarity between the predicted and ground-truth text spans. The evaluation metric of other tasks is accuracy.

Task	Input&Prompt
C^3	Question: $question\n Answer:$choice comes from the dialogue: $context
OCNLI	$S1? Yes/Maybe/No, $S2
CMNLI	$S1? Yes/Maybe/No, $S2
duReader	Read document: $Document\n Question:$Question \n Answer:

Table 9: The input&prompt template for other tasks.

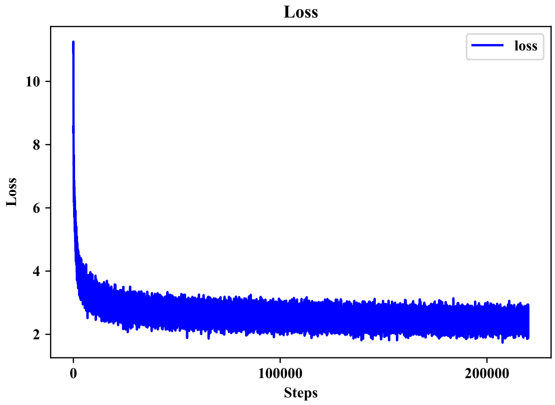


Figure 5: The curves of training loss for CodePanGu2.6B.

C Training Loss

The curves of training loss for the CodePanGu2.6B model are shown in Figure 5. We show that the cross entropy loss decreases steadily during training and the loss of this model converges to around 2.25.

D Case Study

In summary, adding code data in the pre-training stage can effectively improve the general reasoning ability of LLM, and can guide the model to make full use of the coherent reasoning process of the Chain-of-Thought to generate answers. Consistent with GPTRoadMap's point of view (Fu and Khot, 2022), we think this may have something to do with the logic of the code itself. To further explain why the code improves the reasoning ability of the model, we found several sample codes from the dataset and explained each code, as shown in Figure 6.

We found that, regardless of the length of the code dealing with different problems, step-by-step reasoning is required to ensure that the code is generated correctly, similar to the Chain-of-Thought required by other reasoning tasks. This may indicate that the model implicitly learns the thinking

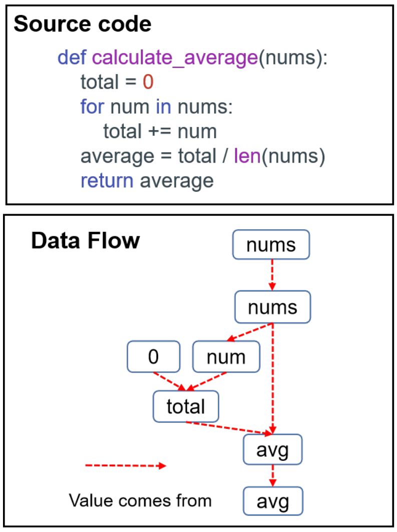
Description	Code	Explanation

Write a funtion to implement quick sort	def quicksort(arr):
	    if len(arr) <= 1:
	        return arr
	    pivot = arr[0]
	    less = [x for x in arr[1:] if x <= pivot]
	    greater = [x for x in arr[1:] if x > pivot]
	    return quicksort(less) +
	            [pivot] + quicksort(greater)
		1.Choose the first element as the pivot
		2.Create a list of elements smaller than or equal to the pivot
		3.Create a list of elements greater than the pivot
		4.Recursively sort the list

Write an online shopping system based on python	class Book:
	    def __init__(self, title, category):
	        self.title = title
	        ...
	    def borrow(self, borrower):
	    ...
	class Library:
	    def __init__(self):
	        self.books = []
	        self.borrow_history = []
	        def borrow_book(self, title,
	                        borrower):
	        ...
		1.Create a book class (Book):
		Attributes:
		title, author, classification, ...
		method:
		borrow, return_book, Display book information (display_info), ...
		2.Create a library class (Library):
		Attributes:
		Book list (books), Borrowing History (borrow_history), ...
		method:
		Add Book (add_book), Remove Book (remove_book), ...

Figure 6: Examples of different codes.

chain ability through the code data, which improves the reasoning ability of the language model. In addition, we analyzed the data flow graph of the *calculate_average* function, as shown in Figure 7. We found many data flow dependence relations in the code data, which are distributed among different code variables. Complex reasoning tasks usually require long dependencies to infer correct conclusions, so the language model may benefit from dependencies such as data and control flow of code data and improve the reasoning ability of the model.


Source code

```
def calculate_average(nums):
    total = 0
    for num in nums:
        total += num
    average = total / len(nums)
    return average
```

Figure 7: Examples of code dependencies.

E Dataset Construction

Cleaning and Filtering. To improve the data quality, we adopt the following rule-based text cleaning strategies over the raw web pages from Common Crawl. Remove the document which contains less than 60% Chinese characters, or less than 150 characters, or only the title of a webpage; Remove the special symbols and duplicated paragraphs in each document; Identify advertisements based on keywords and remove documents that contain advertisements; Convert all traditional Chinese text to simplified Chinese; Identify the navigation bar of the web page and remove it.

Text Deduplication. Although we removed duplicate paragraphs in each document in the previous step, there are still documents with highly overlapping content in different data sources. Therefore, we carry out fuzzy data deduplication over the documents across all our data sources to further remove high-overlap content.

Data Selection. Using the construction process described above, we constructed filtered text corpora from five types of data sources. Based on this corpus, we constructed a training dataset of 100GB text data by sampling each data source according to the ratio of Figure 2 and used this data as the first part of the training set to train CodePanGu2.6B.