

CPSC 488/588, Fall 2023
AI Foundation Models

Lecture 25:
Foundation Models for Code and Math

Ansong Ni
Yale University

Dec 5, 2023

Why Build Foundation Models for Code and Math

- Quick Poll
 - GitHub Copilot
 - OpenAI ChatGPT



Why Build Foundation Models for Code and Math

- How to automatically write programs is one of the ***oldest*** and ***hardest*** problems in AI and CS:

This process of constructing instruction tables should be very fascinating. There need be no real danger of it ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself.

— Alan Turing (1945)

Programming
Languages

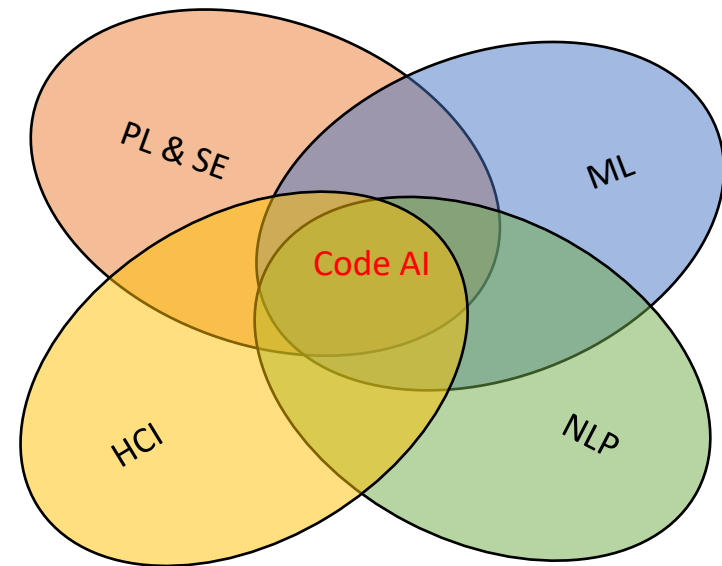
D. Gries
Editor

**Toward Auto-
matic Program
Synthesis**

Zohar Manna
Stanford University,* Stanford, California
and
Richard J. Waldinger
Stanford Research Institute,†
Menlo Park, California

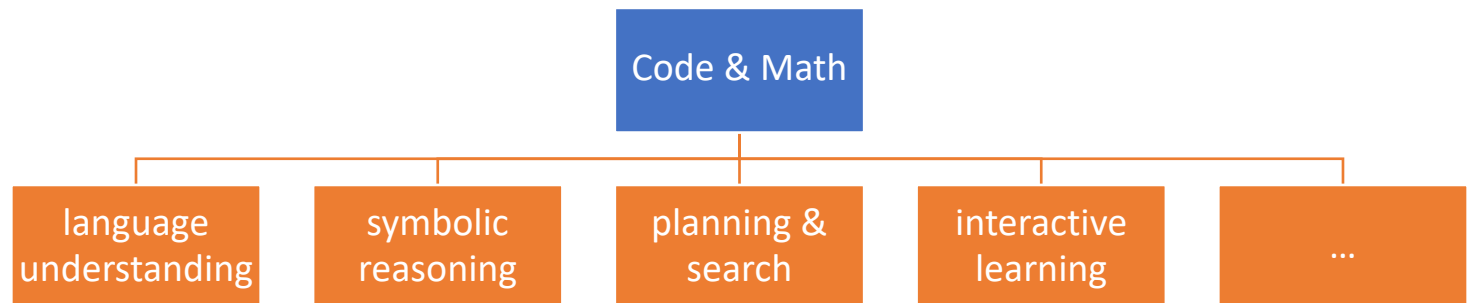
Why Build Foundation Models for Code and Math

- They relate to several important areas in CS
 - Programming Languages (PL)
 - Software Engineering (SE)
 - Machine Learning (ML)
 - Natural Language Processing (NLP)
 - Human-Computer Interaction (HCI)
 - ...



Why Build Foundation Models for Code and Math

- Coding and doing math are great **testbeds for *intelligence***:
 - language understanding
 - symbolic reasoning
 - planning & search
 - interactive learning
 - ...



Why Build Foundation Models for Code and Math

- They empower many real-world applications:

The collage illustrates four real-world applications of foundation models:

- FlashFill:** A screenshot of a Microsoft Excel spreadsheet showing a list of names. A green box highlights a pattern, and a text overlay says "Type a few examples to show a clear pattern".
- Virtual Assistants:** A smartphone screen displaying a chat interface with a virtual assistant. The assistant responds to user prompts like "Let me know when you want to know more — I'll be here." and "Send me Esquire Wisdom daily".
- Robotics Control:** A diagram showing a "Large Language Model" connected to "Policy Code" and "Perception APIs/Control APIs". The policy code includes functions like `detect_objects`, `is_empty`, and `stack_objects`. Below the code is a photo of a robotic arm stacking blocks.
- Database Query and Visualization:** A screenshot of a Tableau interface showing a data visualization of "Wine Sales" by country and price. The interface includes a search bar and a list of dimensions and measures.

Images from: <https://developers.googleblog.com/2018/03/new-creative-ways-to-build-with-actions.html>; <https://support.microsoft.com/en-us/office/save-time-with-flash-fill-9159216a-75a0-4c11-82e6-8eca29cb3b89>; <https://github.com/features/copilot>; <https://code-as-policies.github.io/>; <https://www.tableau.com/blog/ask-data-simplifying-analytics-natural-language-98655>

Before we start...

Preliminaries

- Assume basic knowledge on terms in NLP and related to LLMs
 - E.g., BERT, GPT, prompting, autoregressive, retrieval, etc
- Mixing of terms
 - Foundation Models \approx LM \approx LLM
 - Code LM/LLM: Language models that have seen code during training
- Code and math LMs
 - They are deeply connected as
 - Both are formal languages;
 - Both require symbolic reasoning
 - This lecture mostly focuses on code LMs but many methods apply for math LMs as well

Outline

- A brief history of code and math LLMs
- Data collection, filtering and tokenization
- Training of code LLMs
 - Decoder-only models and code infilling
 - Encoder-only models;
 - Encoder-decoder models;
 - Reinforcement Learning
- Post-training methods for code LLMs
 - Neuro-symbolic approaches
 - Prompting methods for code
 - Retrieval-augmented generation for code

A Brief History of LMs for Code & Math

Key Events (2020-2021)

- Feb 2020: CodeBERT [1]

- *First attempt -- 16 months after original BERT paper*
- *125M parameters*



- May 2020: GPT-3 [2] paper is out

- *People find that GPT-3 has some coding abilities*
- *Though it is not specifically trained on code*



- Jun 2021: GitHub Copilot is released

- *Revolutionary performance*
- *Multi-line, whole function completion for the first time*



- Jul 2021: Codex [3] paper is out

- *First 10B+ model trained specifically for code*
- *Hero behind GitHub Copilot*

[1] Feng et al. (2020), "CodeBERT: A Pre-Trained Model for Programming and Natural Languages."

[2] Brown et al. (2020), "Language Models are Few-Shot Learners."

[3] Chen et al. (2021), "Evaluating Large Language Models Trained on Code."

Key Events (2022)



- **Feb 2022: AlphaCode [1] paper is out**
 - *Claims 54.3% rankings in competitions with human participants*
 - *Up to 41B, model not released nor publicly accessible*



- **Mar 2022: CodeGen [2] is released**
 - *Open-source 10B+ code LM*
 - *Later found that the model is severely under-trained (later CodeGen2)*



- **Apr 2022: PaLM [3] paper is out**
 - *PaLM-Coder is a 540B code model*
 - *The models are also severely under-trained (later PaLM-2)*



- **Jun 2022: Minerva [4] paper is out**
 - *Finetuned PaLM for math reasoning, up to 540B*



- **Nov 2022: The Stack [5] data is released**
 - *3TB of permissively licensed code data*
 - *Foundational data work for many code LMs in the future*

[1] Li et al. (2022), "Competition-Level Code Generation with AlphaCode."

[2] Nijkamp et al. (2022), "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis."

[3] Chowdhery et al. (2022), "PaLM: Scaling Language Modeling with Pathways."

[4] Lewkowycz et al. (2022), "Solving Quantitative Reasoning Problems with Language Models."

[5] Kocetkov et al. (2022), "The Stack: 3 TB of permissively licensed source code."

Key Events (2023)



- Feb 2023: LLaMA [1] is released
 - *Trained with more data (1T tokens)*
 - *Not as large but more performant than larger models*



- Mar 2023: GPT-4 [2] technical report is out
 - *State-of-the-art in every aspect, coding included*



- May 2023: StarCoder [3] is released
 - *SoTA in open-source, matches Codex-12B in performance*
 - *Trained on the Stack*



- Aug 2023: CodeLLaMA [4] is released
 - *Shortly after the release of LLaMA 2 in Jul 2023*
 - *Continued training of LLaMA 2 on code*
- Oct 2023: Llemma [5] is released
 - *Open math model trained on Proof-Pile-2 by continued training of CodeLLaMA*

[1] Touvron et al. (2023), “LLaMA: Open and Efficient Foundation Language Models.”

[2] OpenAI. (2022), “GPT-4 Technical Report.”

[3] BigCode. (2022), “StarCoder: May the source be with you!”

[4] Rozière et al. (2023), “Code Llama: Open Foundation Models for Code.”

[5] Azerbayev et al. (2023), “Llemma: An Open Language Model For Mathematics.”

Data Collection, Filtering and Tokenization

Code Data Collection and Filtering

- **Data Sources:**
 - Mostly GitHub and similar platforms;
- **Quality Filtering** (take [1] as an example):
 - GitHub stars ≥ 5
 - $1\% \leq \text{Comment-to-code ratio} \leq 80\%$
- **License:**
 - Only permissive licensed open-source repo may be used;
 - E.g., MIT, Apache 2.0

Deduplication and De-contamination

- **Deduplication:**

- Remove (near-)duplicated files from the training data;
- **Why:** repeated training data can significantly hurt the performance [1]

- **Decontamination:**

- Remove the files that contain solutions to benchmarks used for evaluation;
- **Why:** better measure generalization ability of trained LMs

- **Methods:**

- Exact match
- Near-deduplication

Model	Dataset Deduplication Method
InCoder Fried et al. (2022)	Exact Match (alphanumeric token sequence)
CodeGen (Nijkamp et al., 2022)	Exact Match (sha-256)
AlphaCode (Li et al., 2022)	Exact Match (non-whitespace text)
PolyCoder (Xu et al., 2022a)	Exact Match (hash)
PaLM Coder (Chowdhery et al., 2022)	Near-deduplication (Levenshtein distance)
CodeParrot (Tunstall et al., 2022)	Near-deduplication (MinHash)
Codex (Chen et al., 2021)	Exact Match ("unique python files")

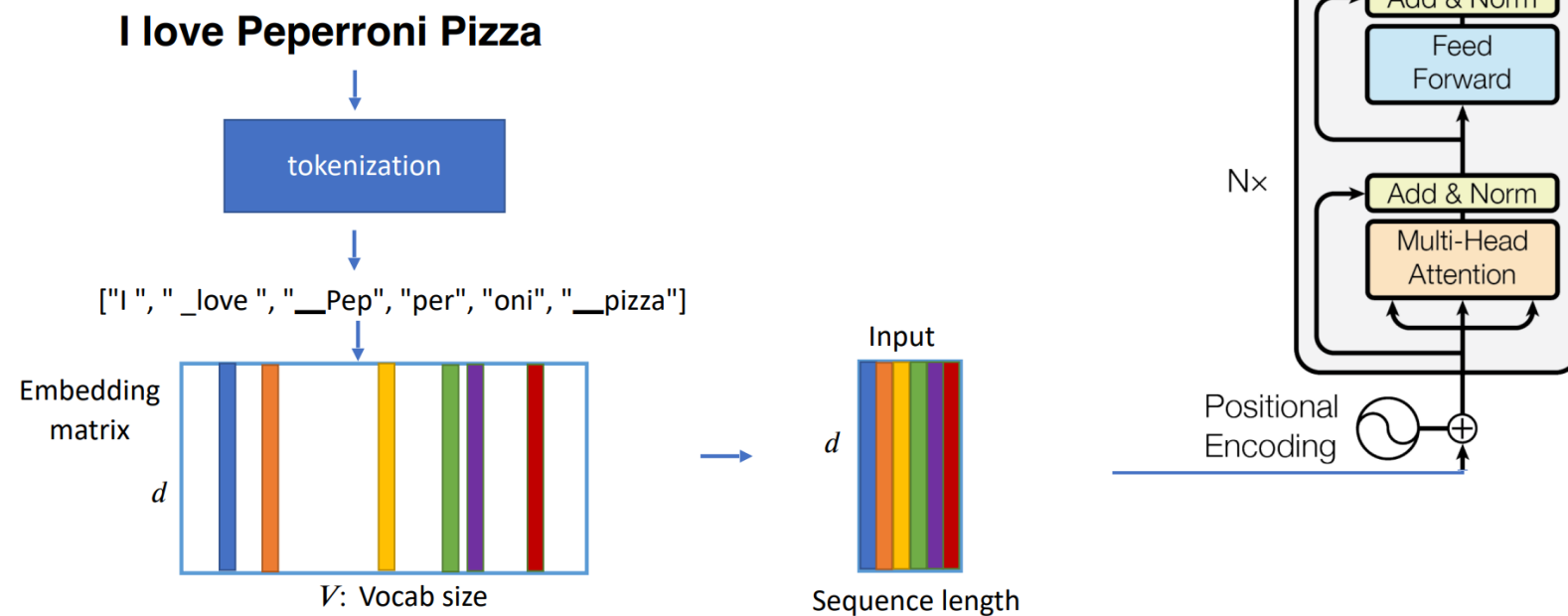
Table 4: Various deduplication methods adopted for different model training data.

[1] Hernandez et al. (2023), “Scaling laws and interpretability of learning from repeated data.”

[2] Ben Allal et al. (2023), “SantaCoder: Don’t Reach for the Stars!”

Tokenization for Code LM (1)

- Recall from Lecture 3



- Tokenization is a ***big deal*** for coding task

Tokenization for Code LM (2)

- Tokenization is a ***big deal*** for coding task
- Code looks very similar but also very different than natural language:
 - **Similar:** semantic meaning of variable/function/class names
 - E.g., "is_correct", "AttentionLayer", "compute_perplexity"
 - **Different:** Whitespace characters, punctuation, indentations
 - E.g., "df.shape[1]", "def f(x):\n\tif x>0:\n\t\treturn x\n\telse:\n\t\treturn x+1"
- Trade-off between:
 - Vocabulary size
 - # tokens needed to encode the same sequence
 - Generalization ability for different tasks

Tokenization for Code LM (3)

- Trade-off between:
 - Vocabulary size
 - # tokens needed to encode the same sequence
 - Generalization ability for different tasks → downstream performance

Lev.	Description	Example
0	Whitespaces in the middle of tokens are prohibited and each punctuation char is treated as a separate token (except '_')	[<code>'for', 'i', 'in', 'range', '(', 'df', '.', 'shape', '[', '1', ']', ')', ':', 'NEW_LINE', 'INDENT', 'print', '(', 'i', ')', 'NEW_LINE', 'print', '(', 'df', '.', 'columns', '[', 'i', ']', ')'</code>]
1	Similar to Level 0, but tokens consisting of several punctuation chars are allowed	[<code>'for', 'i', 'in', 'range', '(', 'df', '.', 'shape', '[', '1', '])', ':', 'NEW_LINE INDENT', 'print', '(', 'i', ') NEW_LINE', 'print', '(', 'df', '.', 'columns', '[', 'i', '])'</code>]
2	Similar to Level 1, but dots are allowed in tokens	[<code>'for', 'i', 'in', 'range', '(', 'df', '.shape', '[', '1', '])', ':', 'NEW_LINE INDENT', 'print', '(', 'i', ') NEW_LINE', 'print', '(', 'df', '.columns', '[', 'i', '])'</code>]
3	Whitespaces and single punctuation chars allowed in tokens, except NEW_LINE	[<code>'for i in range', '(', 'df', '. shape [1', '])', ':', 'NEW_LINE INDENT', 'print', '(', 'i', ') NEW_LINE', 'print', '(', 'df', '. column', 's [i', '])'</code>]
4	Composite tokens of arbitrary complexity are allowed	[<code>'for i in range', '(', 'df', '. shape', '[1]', ')', ':', 'NEW_LINE', 'INDENT print', '(', 'i', ')', 'NEW_LINE print', '(', 'df', '. columns', '[i]'</code>]

Training of Code LLMs

Decoder-only (GPT) Models

- Model architecture and pretraining objectives:
 - Mostly follow those of general-purpose LLMs, e.g., Codex follows the GPT-3
- Multi-stage training:
 - Some models are based off a general-purpose LM
 - E.g., [1] CodeGen-NL→CodeGen-Multi→CodeGen-Mono
 - E.g., [2] LLaMA→CodeLLaMA

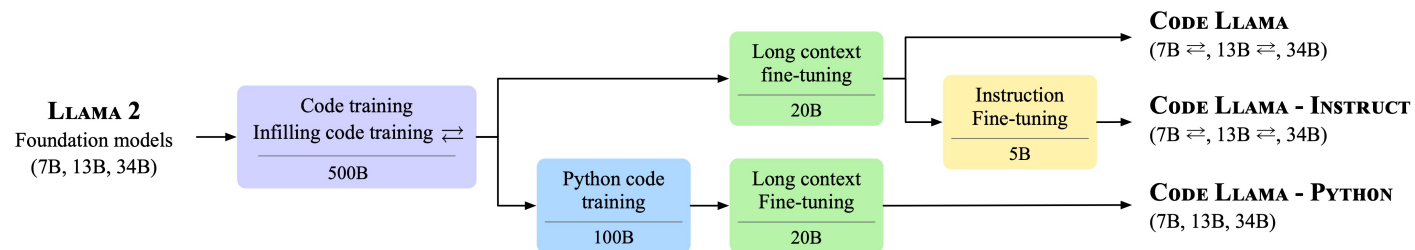


Figure 2: **The Code Llama specialization pipeline.** The different stages of fine-tuning annotated with the number of tokens seen during training. Infilling-capable models are marked with the ⇔ symbol.

[1] Nijkamp et al. (2023), “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.”

[2] Rozière et al. (2023), “Code Llama: Open Foundation Models for Code.”

Code Infilling: Fill in the middle

- Infilling task:
 - <prefix>, <suffix> → <middle>
- Trained via data augmentation [1]:
 - Preprocessing:
 - Special tokens <IF>
 - <prefix>, <middle>, <suffix>
 - <prefix>, <IF>, <suffix>, <IF>, <middle>
 - Mixing with original data
 - Training with normal autoregressive objectives

Docstring Generation

```
def count_words(filename: str) -> Dict[str, int]:  
    """  
    Counts the number of occurrences of each word in the given file.  
  
    :param filename: The name of the file to count.  
    :return: A dictionary mapping words to the number of occurrences.  
    """  
    with open(filename, 'r') as f:  
        word_counts = {}  
        for line in f:  
            for word in line.split():  
                if word in word_counts:  
                    word_counts[word] += 1  
                else:  
                    word_counts[word] = 1  
    return word_counts
```

A use case of infilling [2]

[1] Bavarian et al. (2022), "Efficient Training of Language Models to Fill in the Middle."

[2] Fried et al. (2022), "InCoder: A Generative Model for Code Infilling and Synthesis."

Encoder (BERT) Models for Code (1)

- Aka *code representation learning*
- Code is *multi-modal* and it's usually *automatic* to obtain other modalities
- Other modalities of code may better capture the semantics of code

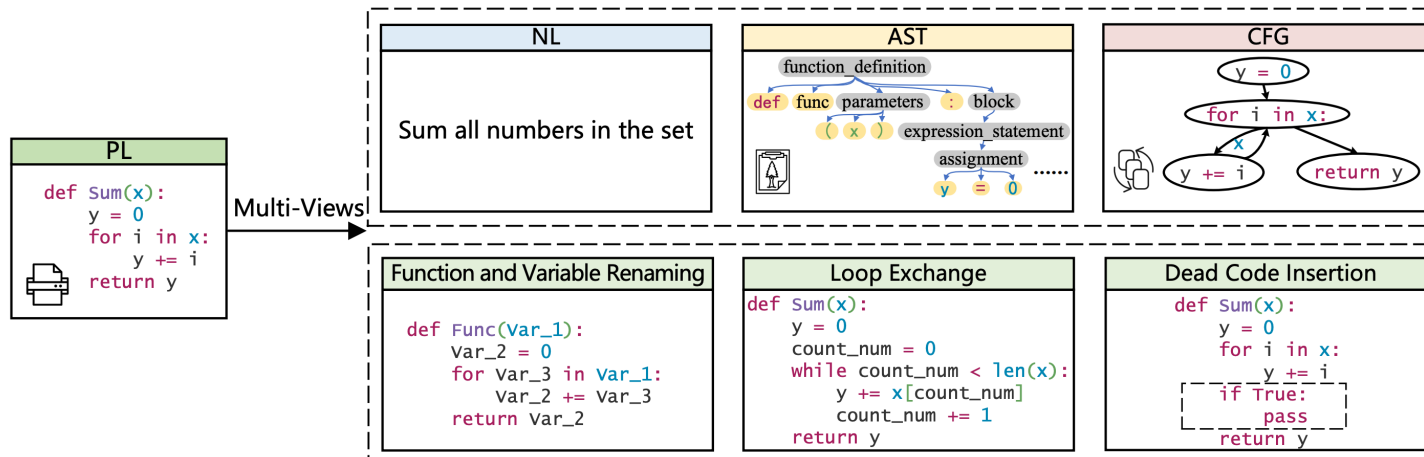
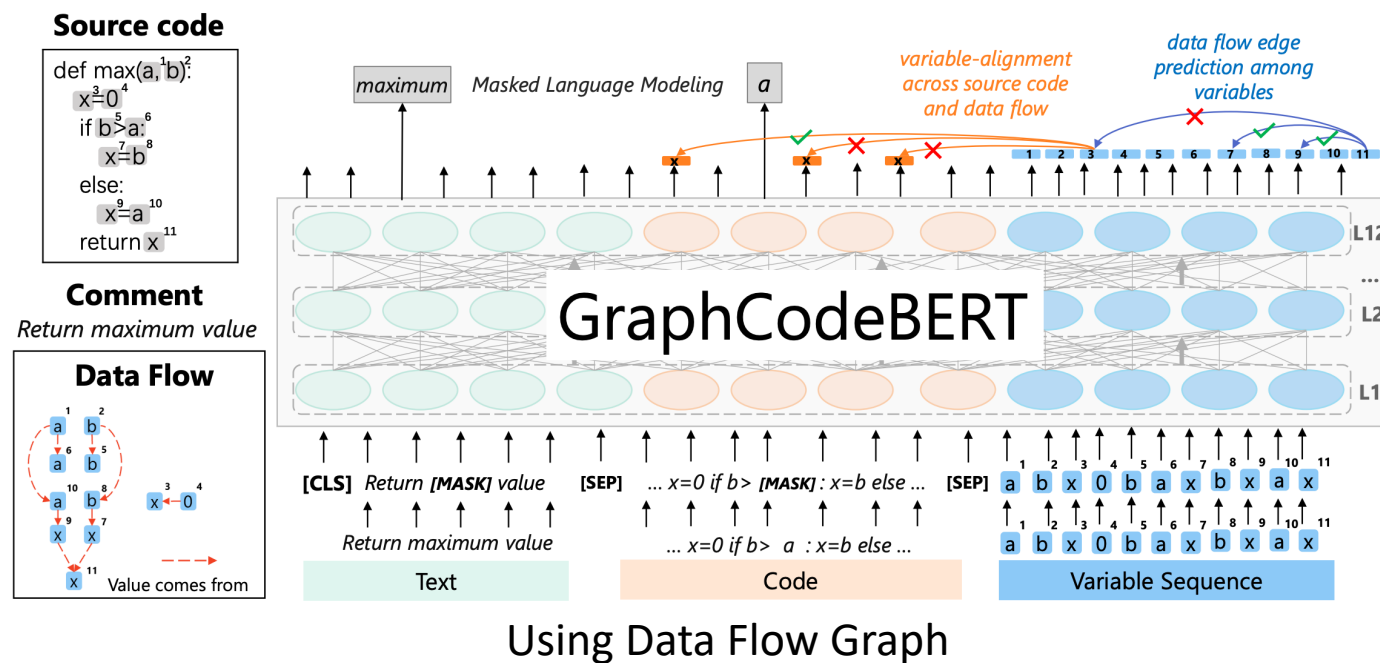


Figure 2: Multiple views of source code.

Encoder (BERT) Models for Code (2)

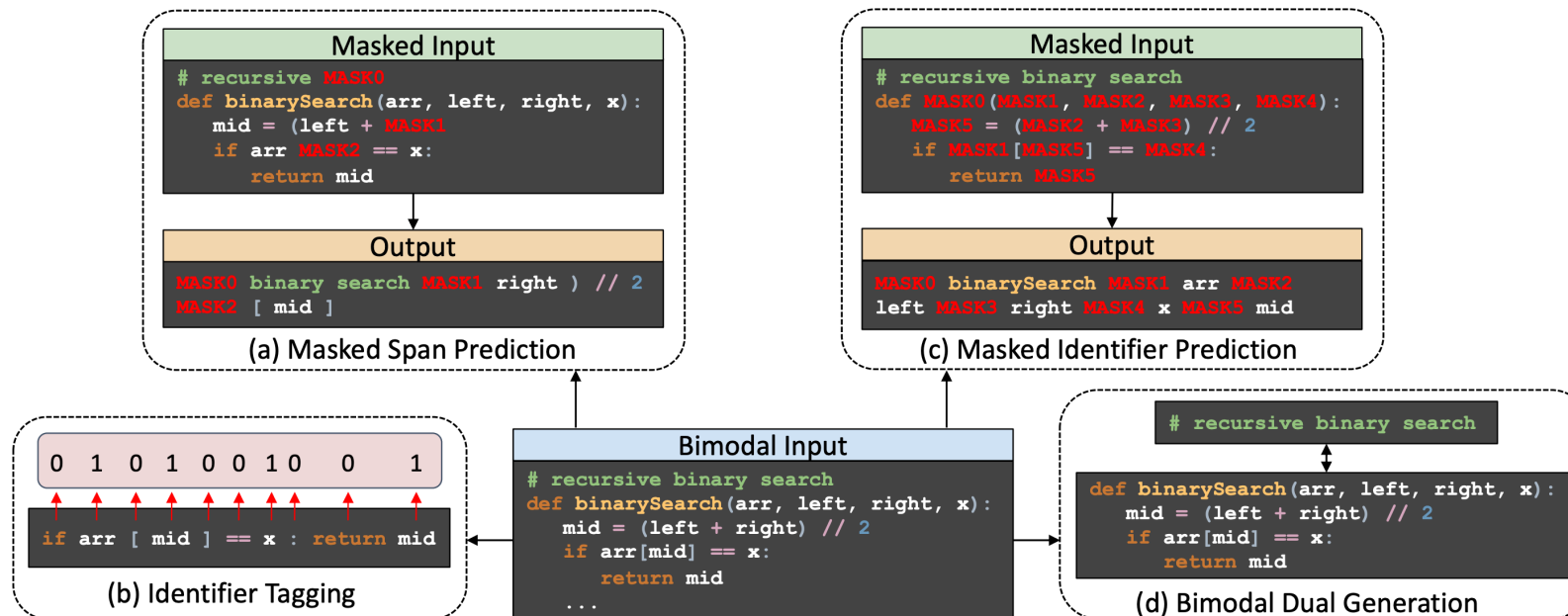
- Code is *multi-modal*
 - Natural language;
 - Surface form;
 - Control flow graph;
 - Abstract-syntax-tree (AST);
 - Data flow graph;
 - Dependency graph;
 - Compiled machine code;
 - ...



- **General idea:** *jointly encode* other modalities with surface form

Encoder-Decoder (BART/T5) Models for Code

- A mixture of **classification** and **generation** tasks for code are typically used during pretraining
 - Researchers get very creative in proposing new pretraining tasks
- E.g., CodeT5 [1]



[1] Wang et al. (2021), "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation."

Reinforcement Learning (1)

- Code generation is a natural task to apply RL as we can automatically obtain *feedback from computers*:

- Pass/fail a parser;
- Pass/fail compilation;
- With/without runtime error;
- Pass/fail test cases

$$r(W^s) = \begin{cases} -1.0 & , \text{ if } W^s \text{ cannot be compiled (i.e. compile error)} \\ -0.6 & , \text{ if } W^s \text{ cannot be executed with unit tests (i.e. runtime error)} \\ -0.3 & , \text{ if } W^s \text{ failed any unit test} \\ +1.0 & , \text{ if } W^s \text{ passed all unit tests} \end{cases}$$

Rewards used for CodeRL

- Examples:
 - CodeRL [1] (offline actor-critic)
 - RLTF [2] (online w/ feedback from compiler)

[1] Le et al. (2021), “CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning.”

[2] Liu et al. (2023), “RLTF: Reinforcement Learning from Unit Test Feedback.”

Reinforcement Learning (2)

- Benefits of using RL:
 - Not limit to learning from a single solution from the dataset;
 - Release the dependency for annotated solutions;
 - Able to directly incorporate preferences;
- Limitations:
 - Insufficient test cases may lead to false positives [1]
 - Rewards are typically sparse and underspecified [2];
 - Especially if we start with a weaker model

[1] Smith et al. (2015), *"Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair."*

[2] Agarwal et al. (2019), *"Learning to Generalize from Sparse and Underspecified Rewards."*

Post-Training Methods for Code LLMs

Neuro-Symbolic Approaches (1): Incorporating Code Execution

- In addition to providing RL learning signal at training time
- **Execution information** can also help improve models **at test time**
- Methods:
 - Sampling + filtering (codex [1])
 - Sampling solutions then filter out those fail to pass a small subset of test cases

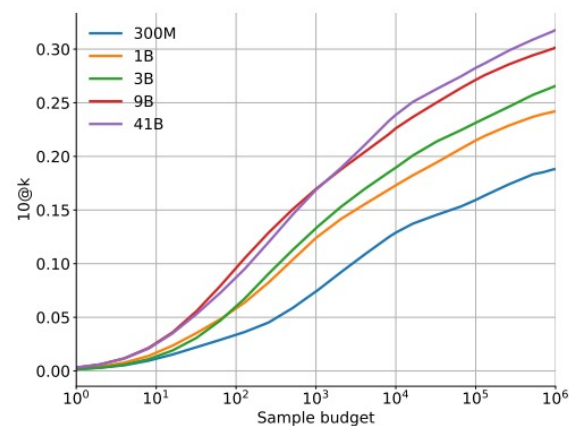
	INTRODUCTORY	INTERVIEW	COMPETITION
GPT-NEO 2.7B RAW PASS@1	3.90%	0.57%	0.00%
GPT-NEO 2.7B RAW PASS@5	5.50%	0.80%	0.00%
1-SHOT CODEX RAW PASS@1	4.14% (4.33%)	0.14% (0.30%)	0.02% (0.03%)
1-SHOT CODEX RAW PASS@5	9.65% (10.05%)	0.51% (1.02%)	0.09% (0.16%)
1-SHOT CODEX RAW PASS@100	20.20% (21.57%)	2.04% (3.99%)	1.05% (1.73%)
1-SHOT CODEX RAW PASS@1000	25.02% (27.77%)	3.70% (7.94%)	3.23% (5.85%)
1-SHOT CODEX FILTERED PASS@1	22.78% (25.10%)	2.64% (5.78%)	3.04% (5.25%)
1-SHOT CODEX FILTERED PASS@5	24.52% (27.15%)	3.23% (7.13%)	3.08% (5.53%)

Codex-12B on APPs. Filtered Pass@k is significantly better

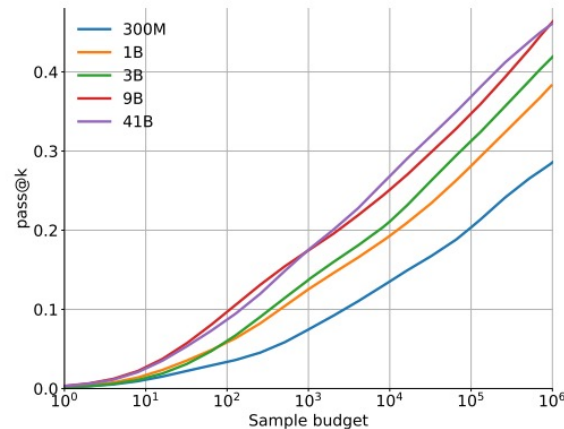
[1] Chen et al. (2021), “Evaluating Large Language Models Trained on Code.”

Neuro-Symbolic Approaches (1): Incorporating Code Execution

- Methods:
 - Sampling + filtering (codex [1])
 - Sampling + filtering + clustering (AlphaCode [2])
 - Sampling lots of diversified program candidates (i.e., up to 1M)
 - Filtering using open test cases
 - Diversify the picked candidates by clustering and selecting from different clusters



(a) 10 attempts per problem



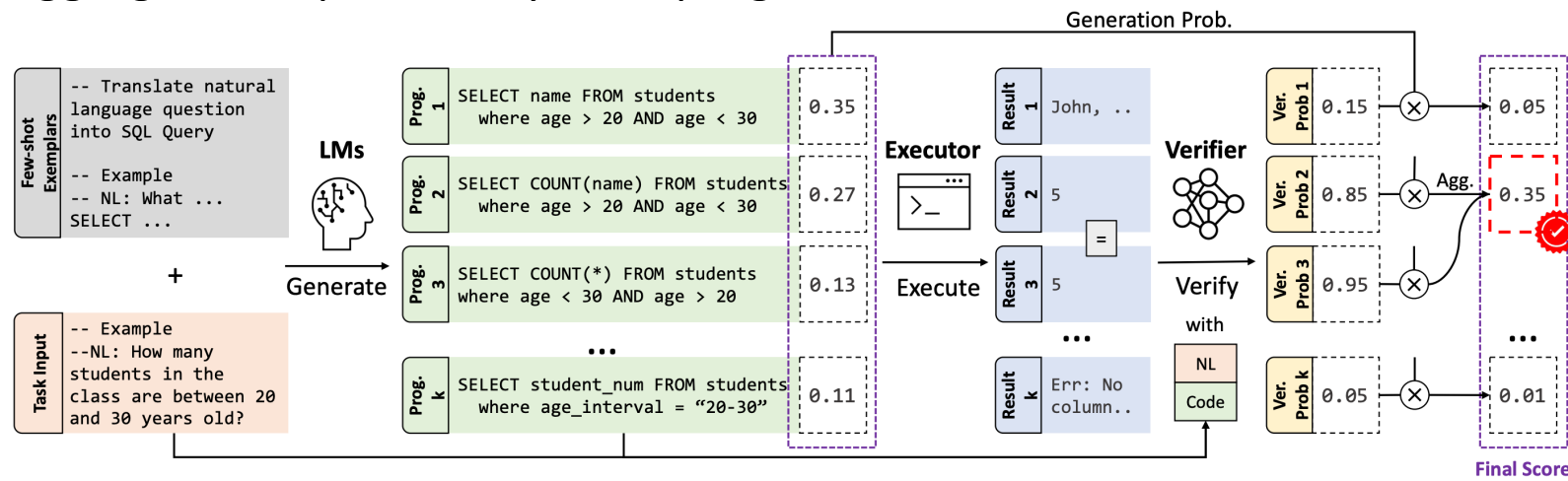
(b) Unlimited attempts per problem

[1] Chen et al. (2021), "Evaluating Large Language Models Trained on Code."

[2] Li et al. (2022), "Competition-Level Code Generation with AlphaCode."

Neuro-Symbolic Approaches (1): Incorporating Code Execution

- Methods:
 - Sampling + filtering (codex [1])
 - Sampling + filtering + clustering (AlphaCode [2])
 - Sampling + verification + voting (LEVER [3])
 - Train a verifier to verify the program with its execution results
 - Aggregate the probability from programs that reach the same execution results



[1] Chen et al. (2021), "Evaluating Large Language Models Trained on Code."

[2] Li et al. (2022), "Competition-Level Code Generation with AlphaCode."

[3] Ni et al. (2023), "LEVER: Learning to Verify Language-to-Code Generation using Execution."

Neuro-Symbolic Approaches (2): Constraint Decoding

- How does code completion work before LLMs?
 - Remember: programs are in *formal languages*, which means that they are regulated by **strict grammar**;
 - Completion Engine (CE): tells you the valid next tokens w/ static analysis 🙋
 - Sounds a lot like a language model, right?
 - But it is a *symbolic* process
- Combining LM with CE [1]:
 - Filter out next token from the LM that are not approved by CE
 - Best of both worlds!

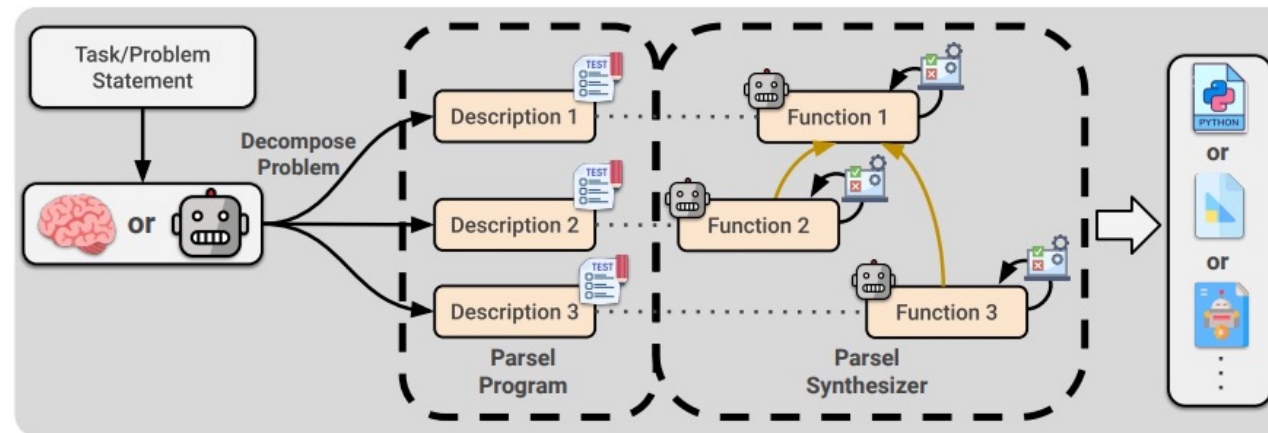
```
1 import numpy as np
2
3 def test():
4     📡 a = np.
```

- [x] abs
- [x] absolute
- [x] add
- 📦 add_docstring
- 📦 add_newdoc
- 📦 add_newdoc_ufunc
- 📦 all
- 📦 allclose

[1] Poesia et al. (2022), “Synchronesh: Reliable code generation from pre-trained language models.”

Neuro-Symbolic Approaches (3): Planning and Search

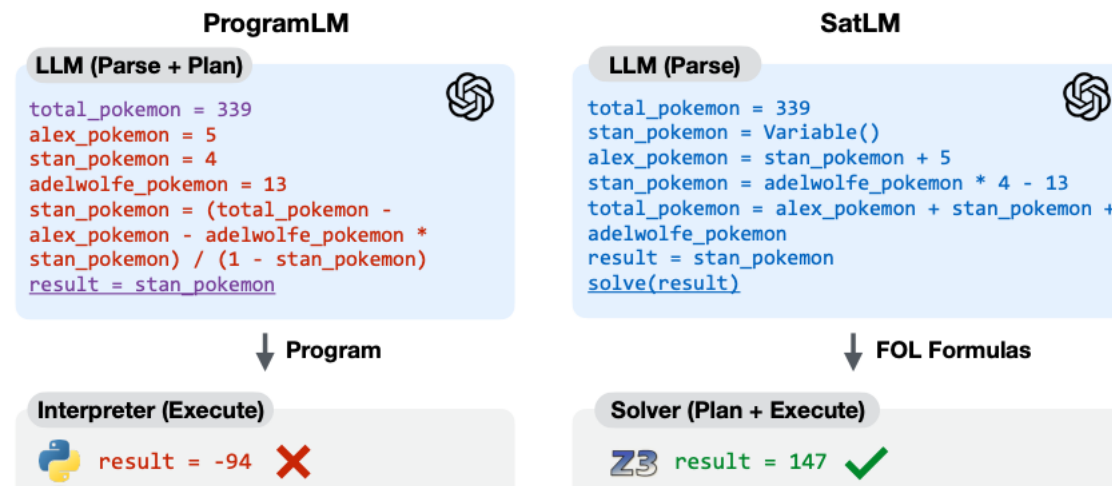
- Programs are **compositional** by design
 - Human programmers typically decompose the problem into smaller parts and write functions to solve each of them → **Planning + Implementation**
 - Given the components (e.g., individual functions), we can use a *solver* to find out if they are sufficient in completing the task → **Search**
- Example 1: **Parsel** [1]



[1] Zelikman et al. (2022), “Parsel : Algorithmic Reasoning with Language Models by Composing Decomposition.”

Neuro-Symbolic Approaches (3): Planning and Search

- Programs are **compositional** by design
 - Human programmers typically decompose the problem into smaller parts and write functions to solve each of them → **Planning + Implementation**
 - Given the components (e.g., individual functions), we can use a *solver* to find out if they are sufficient in completing the task → **Search**
- Example 2: **SatLM** [1]



[1] Xi et al. (2023), "SATLM: Satisfiability-Aided Language Models Using Declarative Prompting."

Prompting Methods using Code for LLMs

- Chain-of-thought (CoT) prompting [1]
 - Explicitly write the reasoning process as **natural language**
- Program-of-thought (PoT) prompting [2] and Program-aided LM (PAL) [3]
 - Explicitly write the reasoning process as a **program**
 - Use *program execution* to obtain the final answer
- Works well with math and other symbolic reasoning tasks
- Also closely related to *tool-use* of LLMs

Program-aided Language models (this work)

Input

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 tennis balls.

tennis_balls = 5

2 cans of 3 tennis balls each is

bought_balls = 2 * 3

tennis balls. The answer is

answer = tennis_balls + bought_balls

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves of bread did they have left?

Model Output

A: The bakers started with 200 loaves

loaves_baked = 200

They sold 93 in the morning and 39 in the afternoon

loaves_sold_morning = 93

loaves_sold_afternoon = 39

The grocery store returned 6 loaves.

loaves_returned = 6

The answer is

answer = loaves_baked - loaves_sold_morning
- loaves_sold_afternoon + loaves_returned

>>> print(answer)

74



[1] Wei et al. (2022), "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models."

[2] Chen et al. (2022), "Program of Thoughts Prompting: Disentangling Computation from Reasoning for Numerical Reasoning Tasks."

[3] Gao et al. (2022), "PAL: Program-aided Language Models."

Retrieval Augmented Generation for Code

- Retrieval-augmented generation (RAG)
 - Retrieves relevant pieces of information from some knowledge base and include them in the prompt
- When programmers code, we look at:
 - Current file (e.g., defined variables, function, classes)
 - Documentation of external libraries ← “DocPrompting” [1]
 - Definitions of imported functions and classes ← “Repo-level Prompt Generator” [2]
 - Github, StackOverflow, geeksforgeeks... ← “REDCODER” [3]
- We should give such information to the LLMs as well!

[1] Zhou et al. (2022), “*DocPrompting: Generating Code by Retrieving the Docs.*”

[2] Shrivastava et al. (2023), “*Repository-Level Prompt Generation for Large Language Models of Code.*”

[3] Parvez et al. (2021), “*Retrieval Augmented Code Generation and Summarization.*”

Summary

- A brief history of code and math LLMs
- Data collection, filtering and tokenization
- Training of code LLMs
 - Decoder-only models and code infilling
 - Encoder-only models;
 - Encoder-decoder models;
 - Reinforcement Learning
- Post-training methods for code LLMs
 - Neuro-symbolic approaches
 - Prompting methods for code
 - Retrieval-augmented generation for code

More Topics I hoped to cover...

- **Interdisciplinary applications**
 - Code as Policies: Language Model Programs for Embodied Control (2023)
 - Large Language Models for Compiler Optimization (2023)
- **Self-Improvement with code LLMs**
 - STaR: Bootstrapping Reasoning With Reasoning (2022)
 - CodeT: Code Generation with Generated Tests (2022)
 - Teaching Large Language Models to Self-Debug (2023)
 - DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines (2023)
- **More ways to learn a code LLM**
 - Show Your Work: Scratchpads for Intermediate Computation with Language Models (2021)
 - Learning Math Reasoning from Self-Sampled Correct and Partially-Correct Solutions (2022)

Hope you enjoyed the lecture!

Questions?