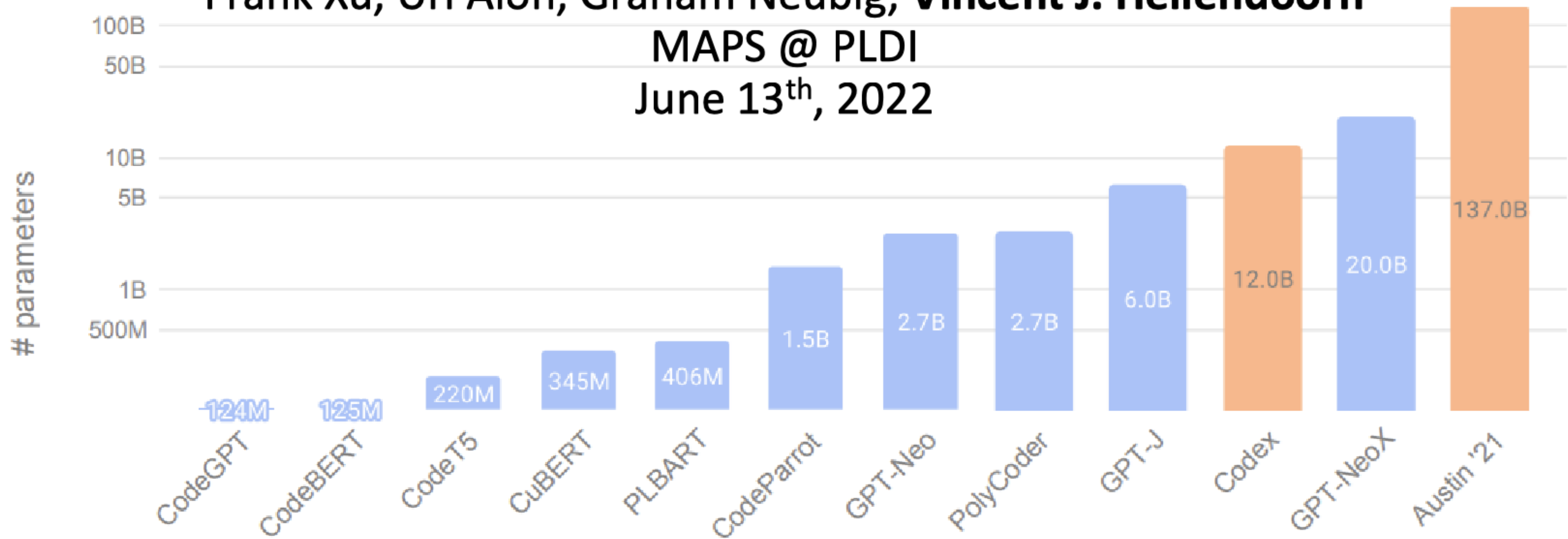
A Systematic Survey of Large Language Models of Source Code

Frank Xu, Uri Alon, Graham Neubig, **Vincent J. Hellendoorn**

MAPS @ PLDI

June 13th, 2022



Outline

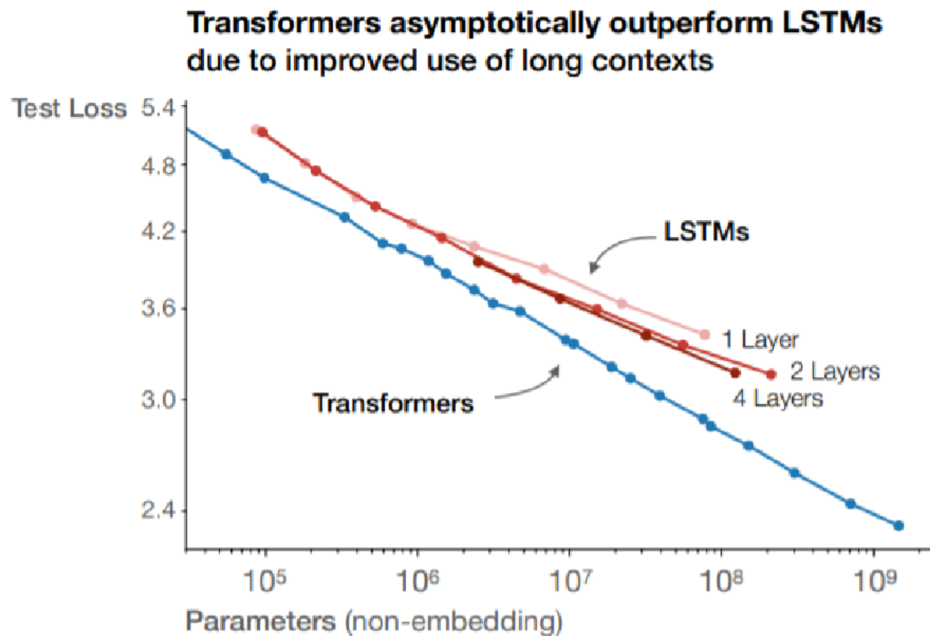
Background

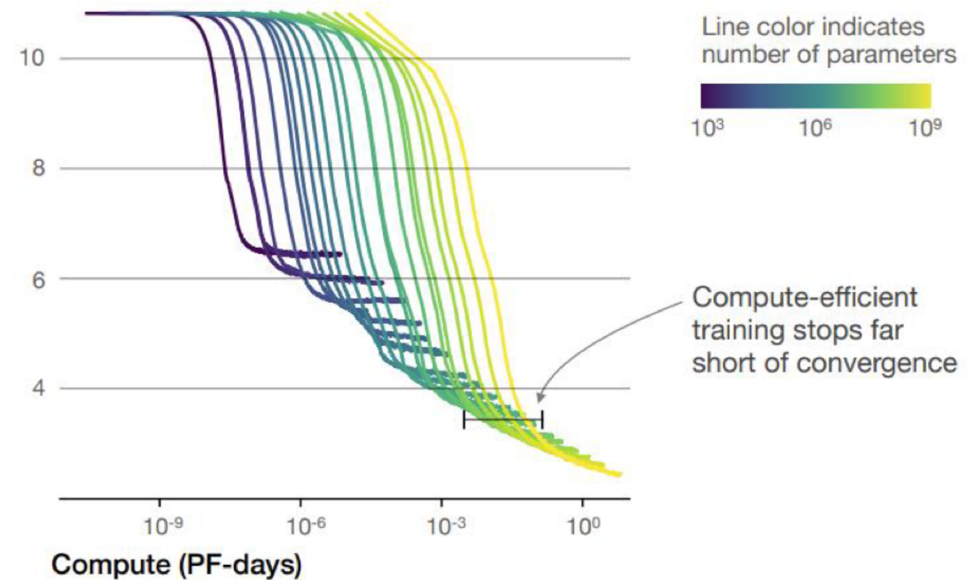
State of the Field

Challenges

Transformers

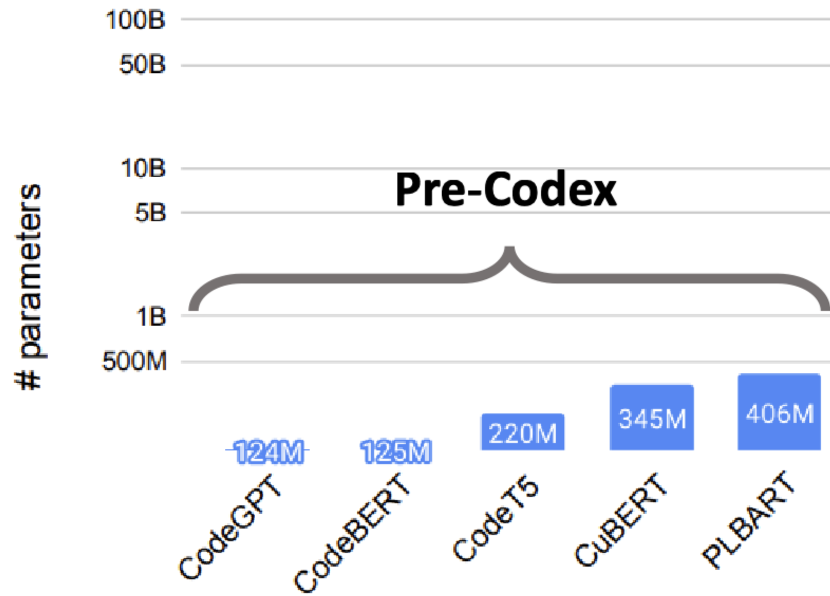
Allow for unprecedented *scaling*





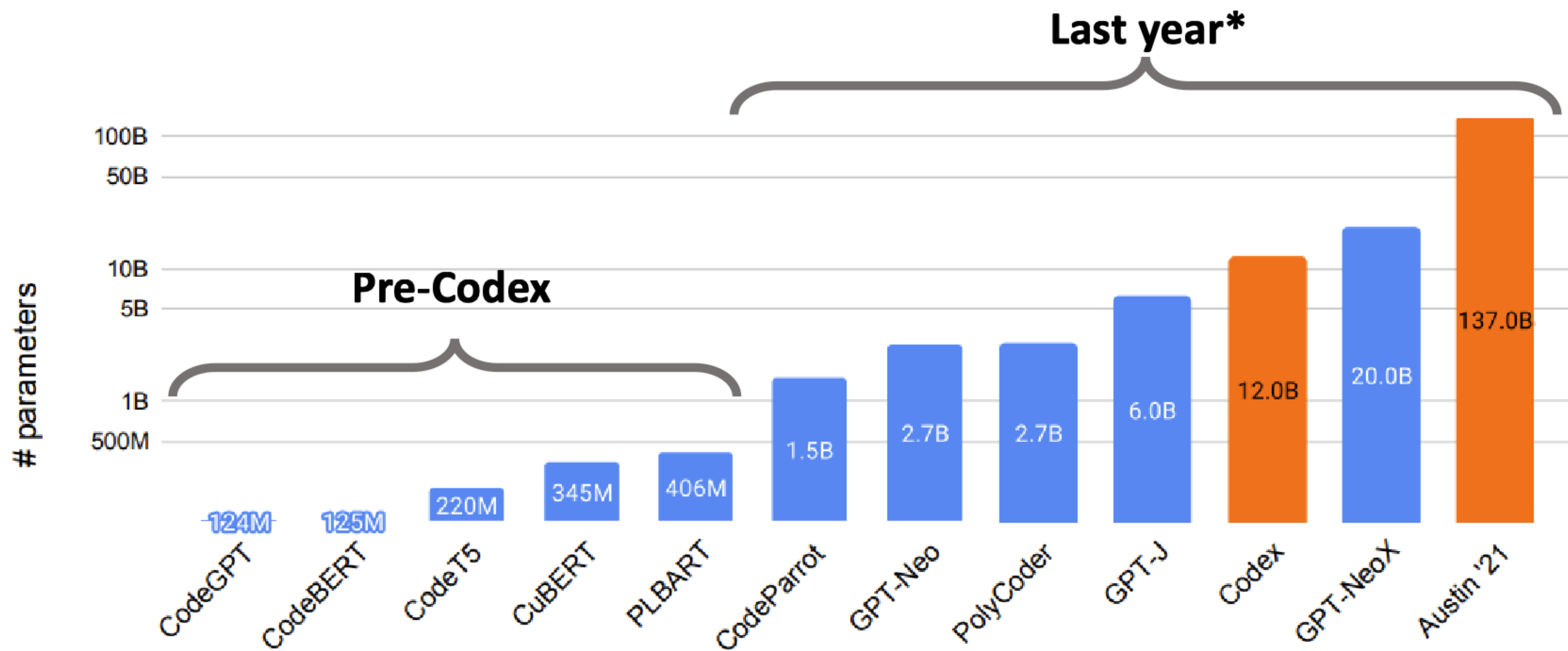
Scaling Laws for Neural Language Models. Kaplan *et al.*, 2020. https://arxiv.org/pdf/2001.08361.pdf

Software: We Scale Too



https://arxiv.org/pdf/2202.13169.pdf

Software: We Scale Too



Note: orange is closed-source

https://arxiv.org/pdf/2202.13169.pdf -- *As of February 2022; missing newer models including CodeGen (16B), PaLM (535B)

Why We’re Here

GitHub Copilot (June 2021)

- Closed-source
- Limited details

```
1  package main
2
3  type CategorySummary struct {
4      Title        string
5      Tasks       int
6      AvgValue    float64
7  }
8
9  func createTables(db *sql.DB) {
10     db.Exec("CREATE TABLE tasks (id INTEGER PRIMARY KEY, title TEXT, value INTEGER, category TEXT
11 }
12
13 func createCategorySummaries(db *sql.D
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
```

https://copilot.github.com/

Outline

Background

State of the Field

Challenges

What Makes a Good LLM for Code?

1. Data
   - Volume
   - Preprocessing
2. Model Size
   - Parameters
3. Initialization
   - NL pretraining
   - Joint code + NL training
4. Training
   - Code tokens seen
   - Language effects
   - Batch size & misc.

A SYSTEMATIC EVALUATION OF LARGE LANGUAGE MODELS OF CODE

Frank F. Xu, Uri Alon, Graham Neubig, Vincent J. Hellendoorn
School of Computer Science
Carnegie Mellon University
{fangzhex,ualon,gneubig}@cs.cmu.edu, vhellendoorn@cmu.edu

ABSTRACT

Large language models (LMs) of code have recently shown tremendous promise in completing code and synthesizing code from natural language descriptions. However, the current state-of-the-art code LMs (e.g., Codex (Chen et al., 2021)) are not publicly available, leaving many questions about their model and data design decisions. We aim to fill in some of these blanks through a systematic evaluation of the largest existing models: Codex, GPT-J, GPT-Neo, GPT-NeoX-20B, and CodeParrot, across various programming languages. Although Codex itself is not open-source, we find that existing open-source models do achieve close results in some programming languages, although targeted mainly for natural language modeling. We further identify an important missing piece in the form of a large open-source model trained exclusively on a multi-lingual corpus of code. We release a new model, PolyCoder, with 2.7B parameters based on the GPT-2 architecture, that was trained on 249GB of code across 12 programming languages on a single machine. In the C programming language, *PolyCoder outperforms all models including Codex*. Our trained models are open-source and publicly available at `https://github.com/VHellendoorn/Code-LMs`, which enables future research and application in this area.

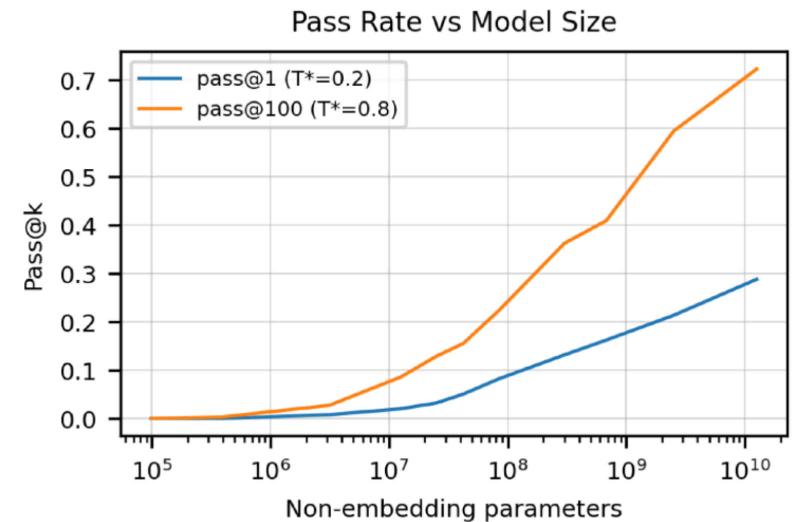
Codex

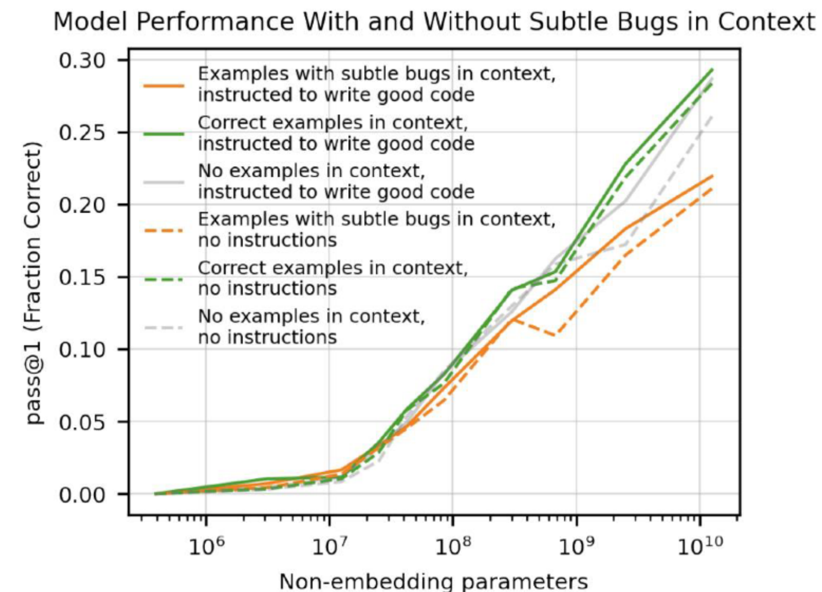
The first many-billion parameter LM for code

- Initialized from GPT-3
- Fine-tuned on 159GB of Python
  - Introduced HumanEval: a benchmark of NL → Python Code problems with tests

Some Findings:

- Strong, log-linear **scaling** after ~ 50M params
- Prompting matters, even non-functional aspects





https://arxiv.org/pdf/2107.03374.pdf

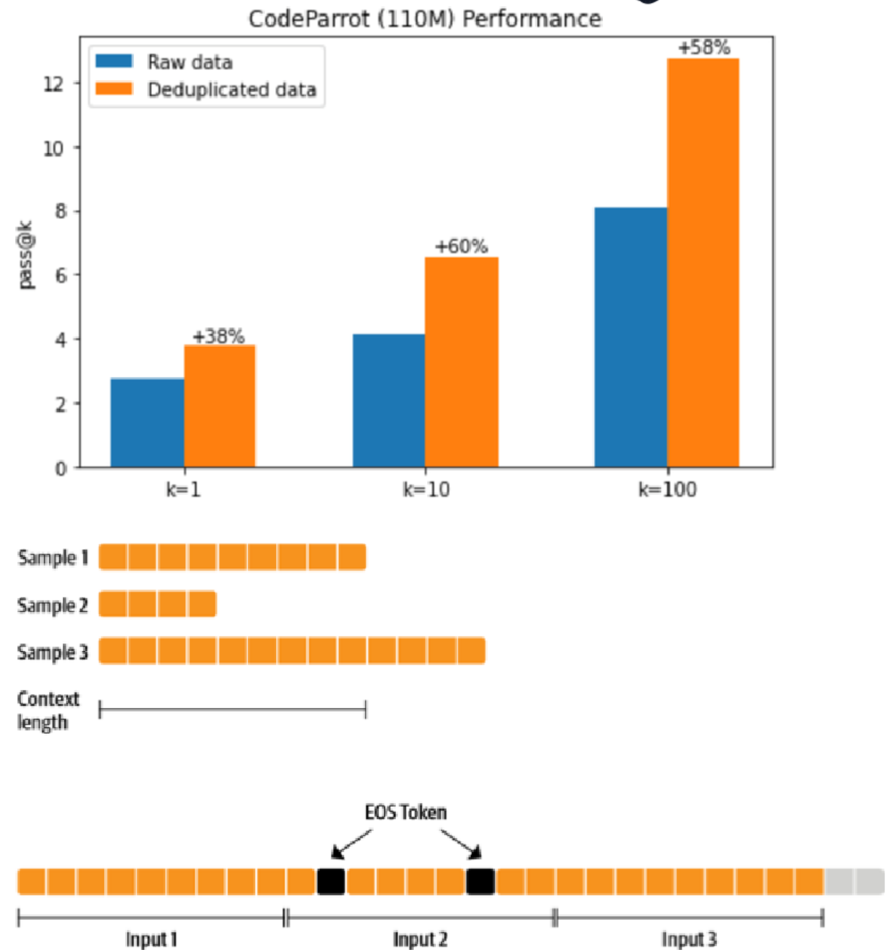
CodeParrot

The first OSS entry

- 1.5B parameters
- 26B Python tokens from BigQuery

Some Findings:

- Deduplication is key
- Uses windows of 1,024 tokens
    - Large inputs are expensive with attention
    - Codex goes up to 4K

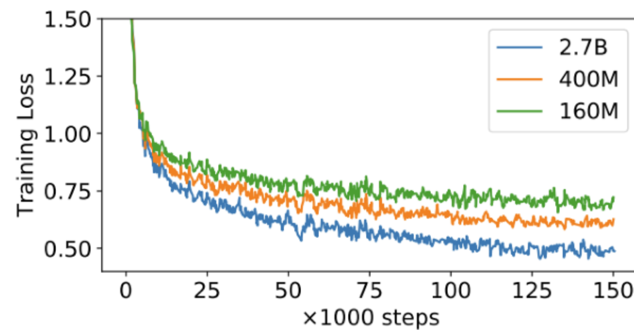


https://huggingface.co/blog/codeparrot

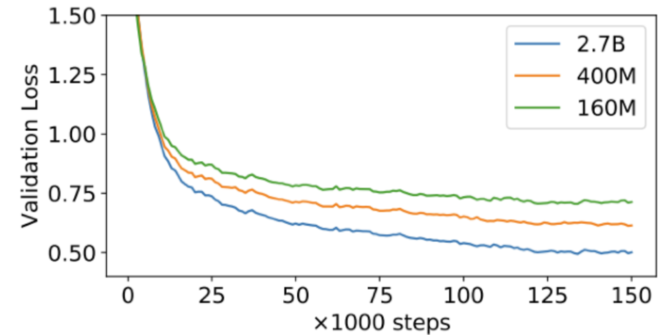
PolyCoder

Our entry from CMU

- 2.7B parameters
- Trained on 12 languages
  - First OSS multi-lingual LLM



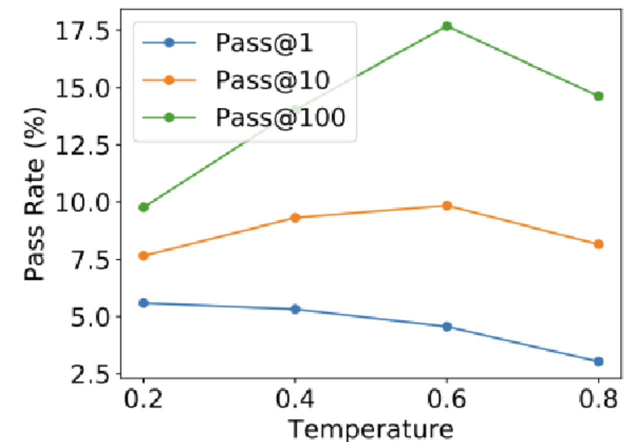
(a) Training



(b) Validation

Some Findings:

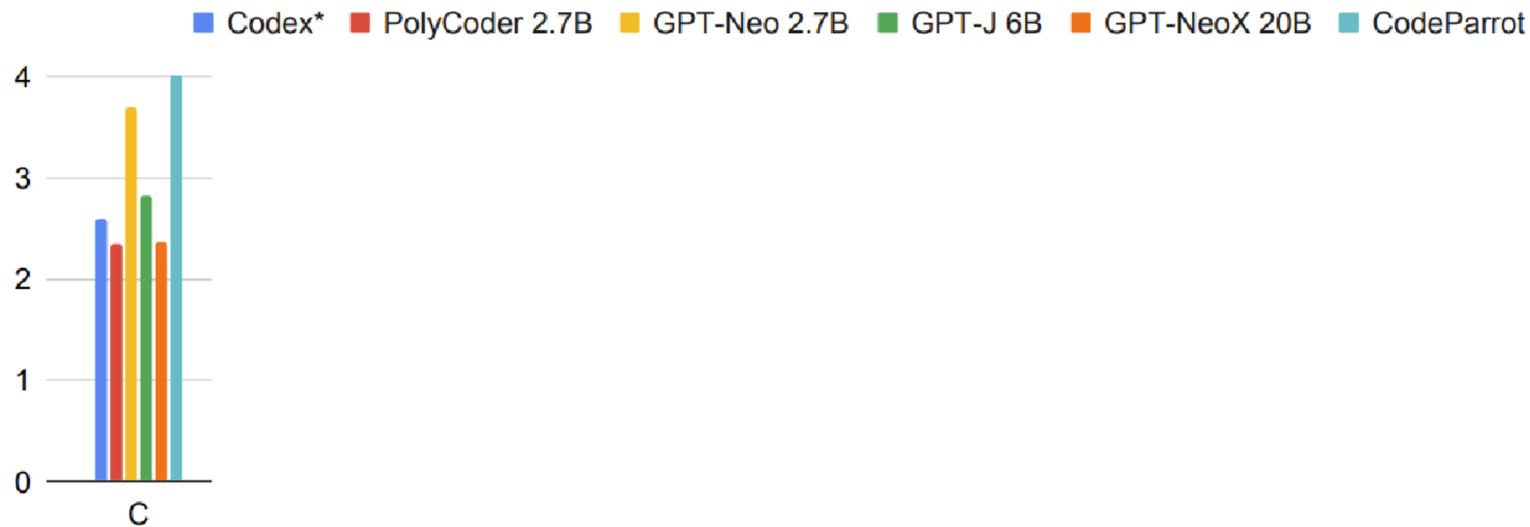
- Edge of single-node/"lab-machine" scale training
  - Ca. 45 days on 8 * RTX 8000 48GB
- Further insights into sampling *temperature*
  - Among others



https://arxiv.org/pdf/2202.13169.pdf

A Systematic Evaluation of Large Language Models of Code

- The good news: PolyCoder outperforms Codex on C

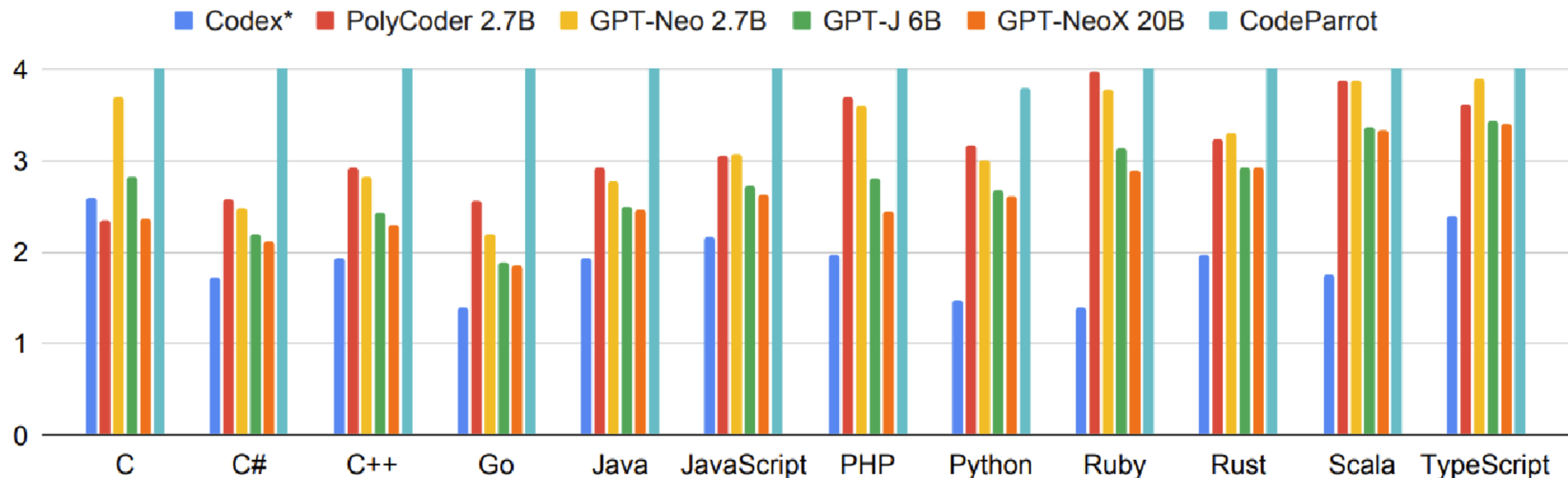


* Since the exact training set of Codex is unknown, it may include files from these test sets rendering Codex's results overly-optimistic.

https://arxiv.org/pdf/2202.13169.pdf – **NOTE:** CodeParrot Python score is incorrect, should be ca. 2.9

A Systematic Evaluation of Large Language Models of Code

- The good news: PolyCoder outperforms Codex on C
- The bad news: most LMs, even some trained on less code, are better on others



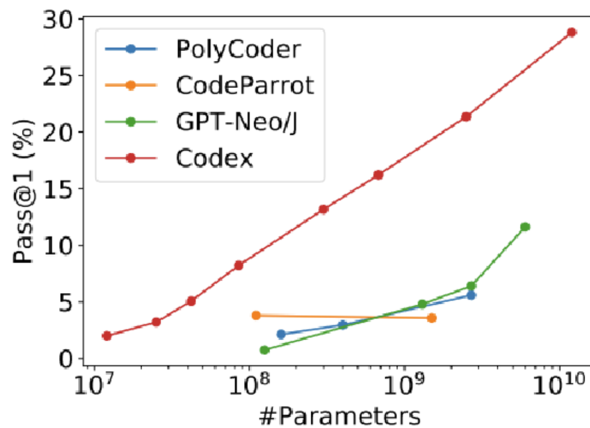
* Since the exact training set of Codex is unknown, it may include files from these test sets rendering Codex's results overly-optimistic.

https://arxiv.org/pdf/2202.13169.pdf – **NOTE:** CodeParrot Python score is incorrect, should be ca. 2.9

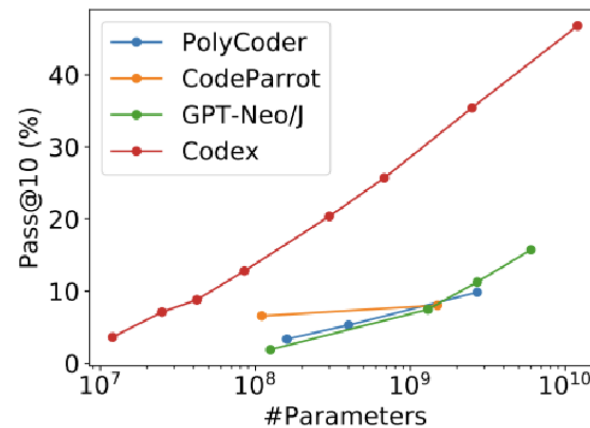
A Systematic Evaluation of Large Language Models of Code

Goal: understand what makes Codex work

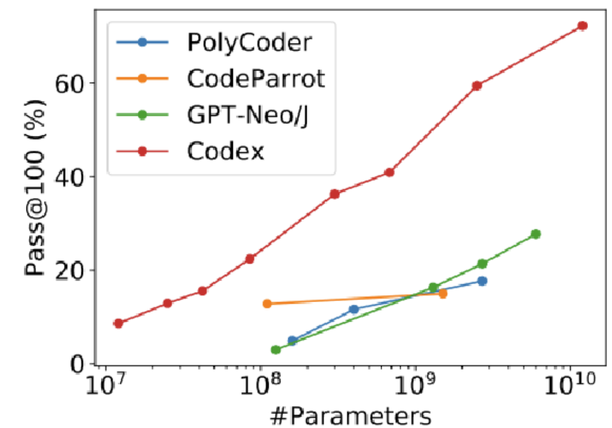
- It seems *unreasonably* effective



(a) Pass@1



(b) Pass@10



(c) Pass@100

https://arxiv.org/pdf/2202.13169.pdf

A Systematic Evaluation of Large Language Models of Code

Goal: understand what makes Codex work

- It seems *unreasonably* effective
- What gives? It does more data preprocessing, but CodeParrot does the same

	PolyCoder	CodeParrot	Codex
Dedup	Exact	Exact	Unclear, mentions “unique”
Filtering	Files $>$ 1 MB, $<$ 100 tokens	Files $>$ 1MB, max line length $>$ 1000, mean line length $>$ 100, fraction of alphanumeric characters $<$ 0.25, containing the word ”auto-generated” or similar in the first 5 lines	Files $>$ 1MB, max line length $>$ 1000, mean line length $>$ 100, auto-generated (details unclear), contained small percentage of alphanumeric characters (details unclear)
Tokenization	Trained GPT-2 tokenizer on a random 5% subset (all languages)	Trained GPT-2 tokenizer on train split	GPT-3 tokenizer, add multi-whitespace tokens to reduce redundant whitespace tokens

https://arxiv.org/pdf/2202.13169.pdf

A Systematic Evaluation of Large Language Models of Code

Goal: understand what makes Codex work

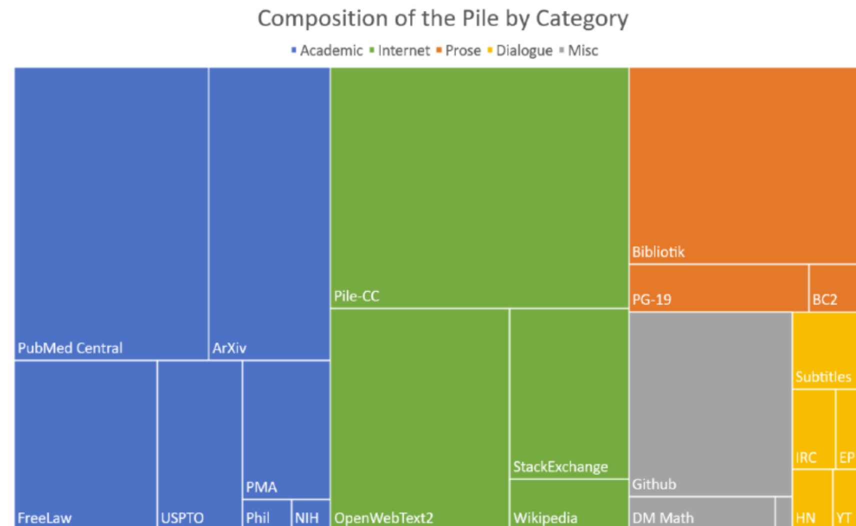
- It seems *unreasonably* effective
- What then? Candidate explanations:

	PolyCoder (2.7B)	CodeParrot (1.5B)	Codex (12B)	
Model Initialization NL Knowledge	From scratch Learned from comments in the code	From scratch Learned from comments in the code	Initialized from GPT-3 Natural language knowledge from GPT-3	Initialization
Learning Rate	1.6e-4	2.0e-4	1e-4	
Optimizer	AdamW	AdamW	AdamW	
Adam betas	0.9, 0.999	0.9, 0.999	0.9, 0.95	
Adam eps	1e-8	1e-8	1e-8	
Weight Decay	-	0.1	0.1	
Warmup Steps	1600	750	175	
Learning Rate Decay	Cosine	Cosine	Cosine	
Batch Size (#tokens) Training Steps Context Window	262K 150K steps, 39B tokens 2048	524K 50K steps, 26B tokens 1024	2M 100B tokens 4096	Training

https://arxiv.org/pdf/2202.13169.pdf

Pre-Training: Let's Talk GPT-x

- Various open source LLMs trained on The Pile
  - Large web-crawl including GitHub (ca. 10%) & StackOverflow
  - Mainly of interest: GPT-J, GPT-Neo, GPT-NeoX
    - Up to 20B parameters (NeoX)



https://arxiv.org/pdf/2101.00027.pdf

Let’s Talk GPT-x

- Trained far longer, but on similar #code tokens
- Around 100M parameters, CodeParrot is decidedly better, followed by PolyCoder

Model	Pass@1	Pass@10	Pass@100	Tokens Trained	Code Tokens	Python Tokens
PolyCoder (160M)	2.13%	3.35%	4.88%	39B	39B	2.5B
PolyCoder (400M)	2.96%	5.29%	11.59%	39B	39B	2.5B
PolyCoder (2.7B)	5.59%	9.84%	17.68%	39B	39B	2.5B
CodeParrot (110M)	3.80%	6.57%	12.78%	26B	26B	26B
CodeParrot (1.5B)	3.58%	8.03%	14.96%	26B	26B	26B
GPT-Neo (125M)	0.75%	1.88%	2.97%	300B	22.8B	3.1B
GPT-Neo (1.3B)	4.79%	7.47%	16.30%	380B	28.8B	3.9B
GPT-Neo (2.7B)	6.41%	11.27%	21.37%	420B	31.9B	4.3B
GPT-J (6B)	11.62%	15.74%	27.74%	402B	30.5B	4.1B
Codex (300M)	13.17%	20.37%	36.27%	100B*	100B*	100B*
Codex (2.5B)	21.36%	35.42%	59.50%	100B*	100B*	100B*
Codex (12B)	28.81%	46.81%	72.31%	100B*	100B*	100B*

https://arxiv.org/pdf/2202.13169.pdf

Let's Talk GPT-x

- Trained far longer, but on similar #code tokens
- But in 1-3B range, Neo is *clearly better*

Model	Pass@1	Pass@10	Pass@100	Tokens Trained	Code Tokens	Python Tokens
PolyCoder (160M)	2.13%	3.35%	4.88%	39B	39B	2.5B
PolyCoder (400M)	2.96%	5.29%	11.59%	39B	39B	2.5B
PolyCoder (2.7B)	5.59%	9.84%	17.68%	39B	39B	2.5B
CodeParrot (110M)	3.80%	6.57%	12.78%	26B	26B	26B
CodeParrot (1.5B)	3.58%	8.03%	14.96%	26B	26B	26B
GPT-Neo (125M)	0.75%	1.88%	2.97%	300B	22.8B	3.1B
GPT-Neo (1.3B)	4.79%	7.47%	16.30%	380B	28.8B	3.9B
GPT-Neo (2.7B)	6.41%	11.27%	21.37%	420B	31.9B	4.3B
GPT-J (6B)	11.62%	15.74%	27.74%	402B	30.5B	4.1B
Codex (300M)	13.17%	20.37%	36.27%	100B*	100B*	100B*
Codex (2.5B)	21.36%	35.42%	59.50%	100B*	100B*	100B*
Codex (12B)	28.81%	46.81%	72.31%	100B*	100B*	100B*

https://arxiv.org/pdf/2202.13169.pdf – NeoX 20B is even better, has been benchmarked here https://arxiv.org/pdf/2204.05999.pdf

Let's Talk GPT-x

- Trained far longer, but on similar #code tokens
- But in 1-3B range, Neo is *clearly better*
- Past 1B parameters, CodeParrot & PolyCoder are <u>seriously underfitting</u>
  - We trained 2.7B parameters with ~40B tokens (seen) – 400B would have been better
- What is the best pretraining/initialization signal?
  - Let's ask the future

CodeGen

A 3-tier training regime

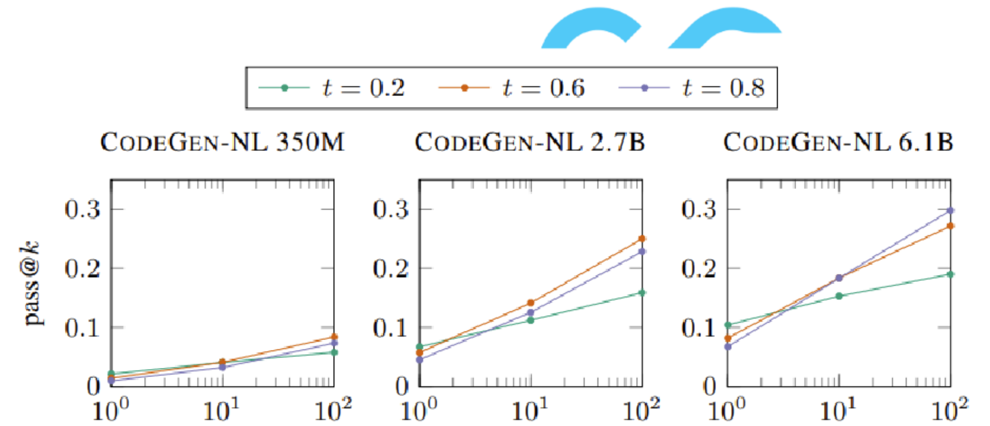
1. Initialize on The Pile
2. Calibrate on 6 languages from BigQuery GitHub
3. Fine-tune on Python-only

https://arxiv.org/pdf/2203.13474.pdf

CodeGen

Key observations:

- NL Scaling is decent, but capped

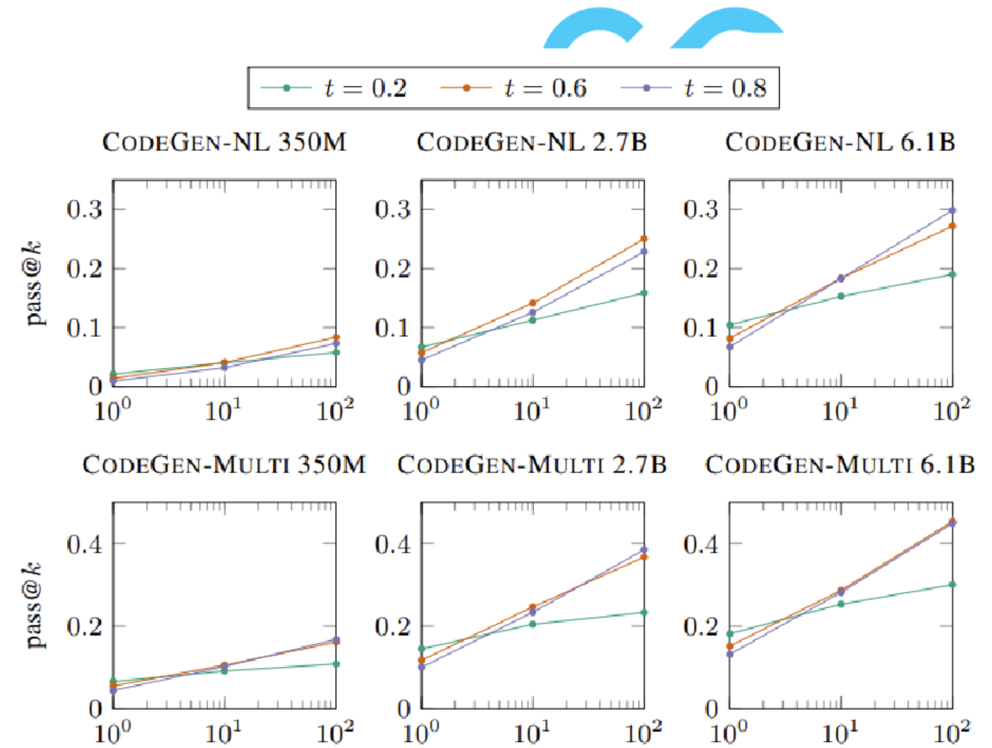


https://arxiv.org/pdf/2203.13474.pdf

CodeGen

Key observations:

- NL Scaling is decent, but capped
- Multi-lingual training helps modestly
  - (note change in y-range)



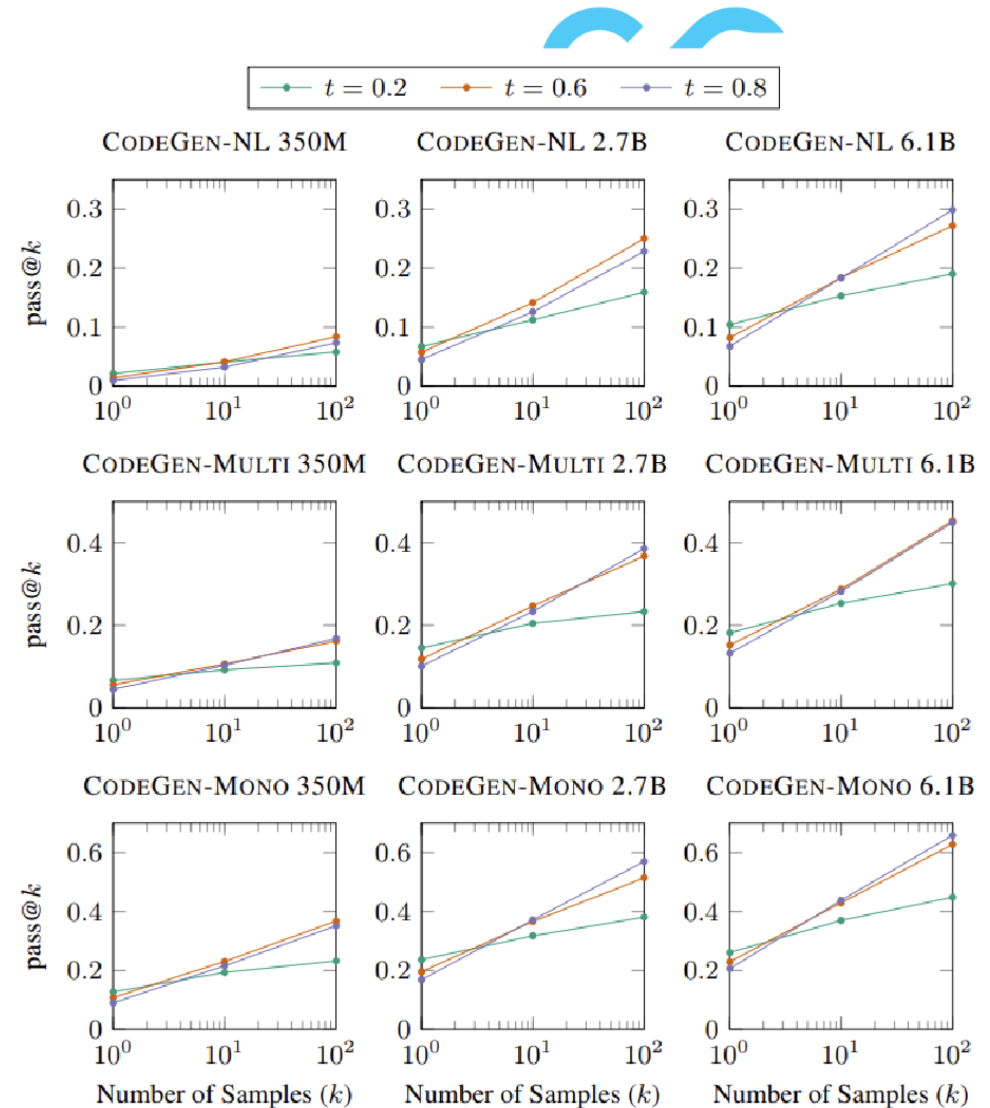
https://arxiv.org/pdf/2203.13474.pdf

CodeGen

Key observations:

- NL Scaling is decent, but capped
- Multi-lingual training helps modestly
  - (note change in y-range)
- Monolingual fine-tuning is crucial
  - First to match Codex

- Is “Multi” before “Mono” necessary?
  - Unclear, Codex suggests not

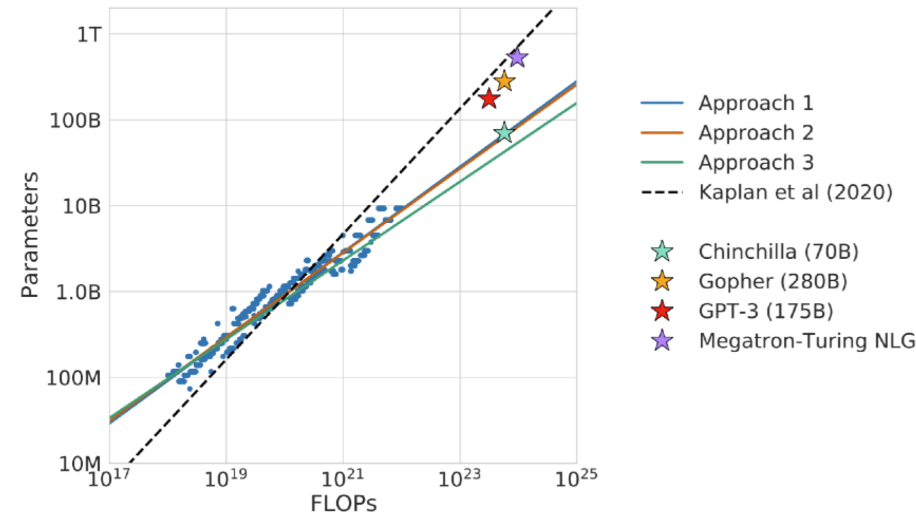


https://arxiv.org/pdf/2203.13474.pdf

How to Match Codex

- Data
  - Several 100B tokens required
    - Rarely available for a single programming language; NL initialization works well
  - Language-specific data, 50B tokens or more, is needed for fine-tuning
- Model
  - Performance increases log-linearly with parameters
  - 2B to 6B parameters is a sweet-spot (for now)
    - Fast to train, performance just 10%-25% shy of Codex
    - But harder/more complex tasks need far more (see PaLM)
- Initialization
  - Pre-training on NL seems helpful, but still unclear if essential
  - Language-specific fine-tuning seems important
    - But better architectures might address this

Open Research Questions



- **Fundamentally:** Better Scaling Laws for Code
  - Chinchilla suggests smaller models, more data
  - If same for code, PolyCoder was near-optimal*
    - The trick is finding that much mono-lingual data
- Context window: 4,096 vs. 2,048
  - AFAIK, only Codex uses the former
  - Code files are large – it should help
  - But, 4K is expensive, all-but necessitates sparse/dense attention
- Tokenization: PolyCoder vocabulary is code-specific, Codex & others aren't
  - Codex's vocab seems to be GPT-3 + sequences of 1 – 24 spaces.
  - Does it matter? This work suggests some code-specific tokenization might help: https://openreview.net/pdf?id=rd-G1nO-Jbq
    - But note: no results on LLMs.

https://arxiv.org/pdf/2203.15556.pdf -- We used 1.4e^{21} FLOPs; Chinchilla suggests using that budget to train ~3-4B parameters and ~75B tokens

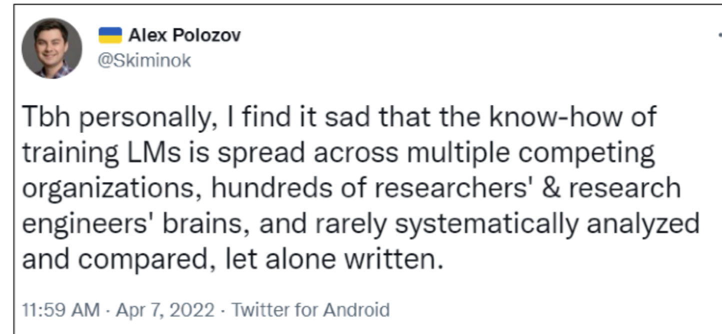
Outline

Background

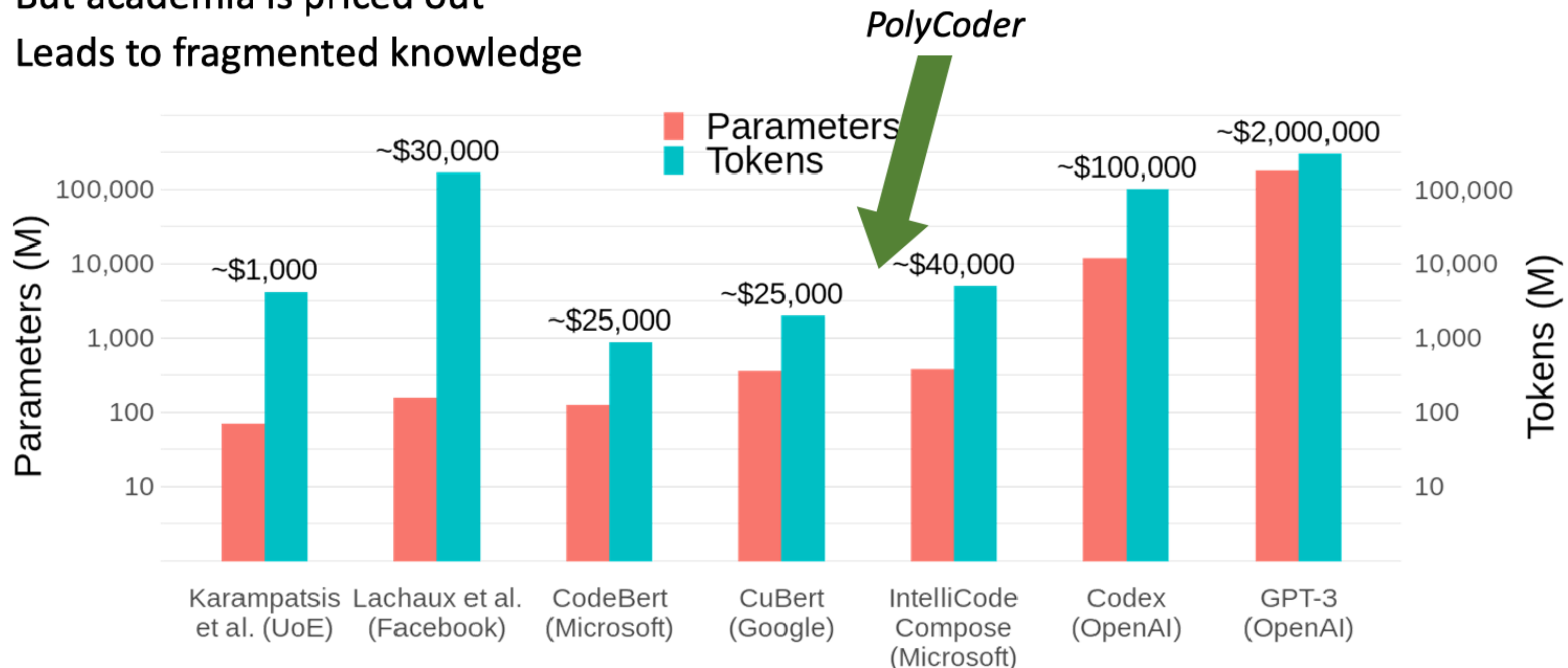
State of the Field

Challenges

The Cost of Scaling



- Huge interest in OSS models
  - But academia is priced out
  - Leads to fragmented knowledge



https://cacm.acm.org/magazines/2022/1/257443-the-growing-cost-of-deep-learning-for-source-code/fulltext
Costs based on approximate PetaFlop seconds at $3/h per V100 GPU – Codex is likely an underestimate
https://twitter.com/Skiminok/status/1512097828373377026

Forgetting Natural Language

Fine-tuning *solely* on code is powerful

- See PALM, CodeGen, Codex
- But, models tend to forget their NL training days
  - That's not ideal for communicating with humans

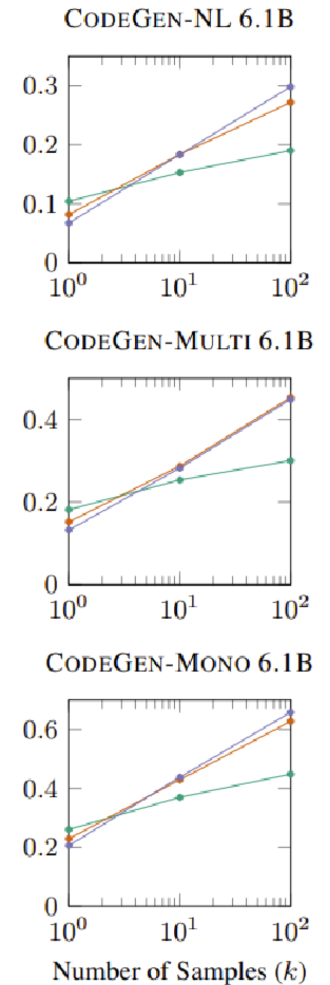
CodeGen-NL 6.1B:

```
Q: How to sum items in a list in Python?
I have a list of items and I want to sum them. How can I do this?

A:
You can use the sum function:
>>> l = [1,2,3,4,5]
>>> sum(l)
15
```

CodeGen-Multi 6.1B:

```
Q: Tensorflow session is not defined.
  """
  if not tf.contrib.eager.in_eager_mode():
    raise RuntimeError('tf.contrib.eager.in_eager_mode() must be True.')
  return tf.contrib.eager.get_default_session()
```



Reliability

- LLMs don't know to generate semantically correct code
  - We just hope they do based on seeing enough data – spoilers: they don't
  - In fact, poor prompts make them more likely to generate vulnerable code
  - Not just a matter of data volume: models associate prompts with good/bad examples seen
- That creates opportunities for prompt engineering
  - E.g., Jigsaw, page 26 of PaLM
  - Unclear if that is future-proof
- What is the alternative?
  - Not sure! Tests are nice, but rarely available – should models write those too?
  - Bringing static analysis in the loop may help (also Jigsaw) – but how to check *any* code?
  - Nothing definitive yet

Questions?

Thanks to my CMU collaborators: Frank Xu, Uri Alon, Graham Neubig!

Models available at: https://github.com/VHellendoorn/Code-LMs/