Trends and Opportunities in Large Language Models of Source Code

Vincent J. Hellendoorn
Carnegie Mellon University
May 3rd, 2022
Why We’re Here

GitHub Copilot (June 2021)

• Closed-source
• Limited details
Outline

Intro to (Foundation) Language Models

State of the Field
Trends, findings, questions

Opportunities

Challenges
Language Modeling

Language is largely “left-to-right”

I am going to the ___ movies grocery store meeting

Recurrent Neural Networks (RNNs) capture this naturally

http://colah.github.io/posts/2015-08-Understanding-LSTMs/
Language Modeling

RNNs condense all history into a single state
... which is provably problematic

Attention: Learn to Ask

https://distill.pub/2016/augmented-rnns/
Transformers

Do we still need RNNs?
- Attention is powerful & highly parallelizable
- Using just attention is possible, but takes quite a few ingredients.
Transformers

Allow for unprecedented scaling
• A key property of foundation models

Transformers Are Good Foundation Models

1. Strong, consistent scaling with compute

Transformers Are Good Foundation Models

1. Strong, consistent scaling with compute
2. Powerful initialization from (generic) pretraining

Transformers Are Good Foundation Models

1. Strong, consistent scaling with compute
2. Powerful initialization from (generic) pretraining
3. Emergent capabilities at large scale

Software: We Scale Too

Note: orange is closed-source

Outline

Intro to (Foundation) Language Models

State of the Field
Trends, findings, questions

Opportunities

Challenges
Models: a Bird’s Eye View

**Trained entirely on Code:**
- CodeParrot (Misc., 2021)
- PolyCoder (CMU, 2022)
- InCoder (FAIR, 2022)

**Trained mostly on NL:**
- GPT-Neo/J/NeoX (Misc + EleutherAI, 2021/2)
- PALM (Google, 2022)
- Austin et al. (Google, 2021)

**A bit of both:**
- Codex (2021, OpenAI)
- CodeGen (2022, Salesforce)

I’ll discuss best-practice based on all of these
What Makes a Good LLM for Code?

1. Data
   • Volume
   • Preprocessing

2. Model Size
   • Parameters

3. Initialization
   • NL pretraining

4. Training
   • Code tokens seen
   • Language effects
   • Batch size & misc.
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
CodeParrot

The first OSS entry
• 1.5B parameters
• 26B Python tokens from BigQuery

Some Findings:
• Ca. 70% duplication – deduplication is key
• Code files can be very long
  • Segment into windows of 1,024
  • This is common in NL training too

https://huggingface.co/blog/codeparrot
PolyCoder

Our entry from CMU
• 2.7B parameters
• Trained on 12 languages

Some Findings:
• Edge of single-node/“lab-machine” scale training
  • Ca. 45 days on 8 * RTX 8000 48GB
• Further insights into sampling temperature

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 likely 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 likely 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

Goal: understand what makes Codex work

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

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

*https://arxiv.org/pdf/2202.13169.pdf*
Goal: understand what makes Codex work

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

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

- Large batches yield lower loss
  - 2M+ tokens per batch is now common
- But, greatly increases GPU needs
  - At 2.7B params, a 48GB GPU can fit ca. $2^{15}$ tokens
  - We can simulate larger batches with "gradient accumulation", but that is very slow

Pre-Training: Let’s Talk GPT-x

• Various open source LLMs exist
  • Mainly of interest: GPT-J, GPT-Neo, GPT-NeoX
  • Trained with/by EleutherAI
  • Up to 20B parameters (NeoX)

• Trained on The Pile
  • Large web-crawl including GitHub (ca. 10%) & StackOverflow
  • “Third” option, besides code-only or NL first, then Code

Composition of the Pile by Category

Let’s Talk GPT-x

- Trained far longer, but on similar #code tokens

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

Let’s Talk GPT-x

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Pass@1</th>
<th>Pass@10</th>
<th>Pass@100</th>
<th>Tokens Trained</th>
<th>Code Tokens</th>
<th>Python Tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyCoder (160M)</td>
<td>2.13%</td>
<td>3.35%</td>
<td>4.88%</td>
<td>39B</td>
<td>39B</td>
<td>2.5B</td>
</tr>
<tr>
<td>PolyCoder (400M)</td>
<td>2.96%</td>
<td>5.29%</td>
<td>11.59%</td>
<td>39B</td>
<td>39B</td>
<td>2.5B</td>
</tr>
<tr>
<td>PolyCoder (2.7B)</td>
<td>5.59%</td>
<td>9.84%</td>
<td>17.68%</td>
<td>39B</td>
<td>39B</td>
<td>2.5B</td>
</tr>
<tr>
<td>CodeParrot (110M)</td>
<td>3.80%</td>
<td>6.57%</td>
<td>12.78%</td>
<td>26B</td>
<td>26B</td>
<td>26B</td>
</tr>
<tr>
<td>CodeParrot (1.5B)</td>
<td>3.58%</td>
<td>8.03%</td>
<td>14.96%</td>
<td>26B</td>
<td>26B</td>
<td>26B</td>
</tr>
<tr>
<td>GPT-Neo (125M)</td>
<td>0.75%</td>
<td>1.88%</td>
<td>2.97%</td>
<td>300B</td>
<td>22.8B</td>
<td>3.1B</td>
</tr>
<tr>
<td>GPT-Neo (1.3B)</td>
<td>4.19%</td>
<td>7.41%</td>
<td>16.30%</td>
<td>380B</td>
<td>28.8B</td>
<td>3.9B</td>
</tr>
<tr>
<td>GPT-Neo (2.7B)</td>
<td>6.41%</td>
<td>11.27%</td>
<td>21.37%</td>
<td>420B</td>
<td>31.9B</td>
<td>4.3B</td>
</tr>
<tr>
<td>GPT-J (6B)</td>
<td>11.62%</td>
<td>15.74%</td>
<td>27.74%</td>
<td>402B</td>
<td>30.5B</td>
<td>4.1B</td>
</tr>
<tr>
<td>Codex (300M)</td>
<td>13.17%</td>
<td>20.37%</td>
<td>36.27%</td>
<td>100B*</td>
<td>100B*</td>
<td>100B*</td>
</tr>
<tr>
<td>Codex (2.5B)</td>
<td>21.36%</td>
<td>35.42%</td>
<td>59.50%</td>
<td>100B*</td>
<td>100B*</td>
<td>100B*</td>
</tr>
<tr>
<td>Codex (12B)</td>
<td>28.81%</td>
<td>46.81%</td>
<td>72.31%</td>
<td>100B*</td>
<td>100B*</td>
<td>100B*</td>
</tr>
</tbody>
</table>

Let’s Talk GPT-x

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

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

Let’s Talk GPT-x

• Trained far longer, but on similar #code tokens
• But in 1-3B range, Neo is *clearly better*
• CodeParrot saw the most Python tokens – evidently important at small scale
  • But at 1B+ parameter scale, total training data volume matters, a lot
    • Neo saw 10-15x as many tokens
• CodeParrot & PolyCoder are **seriously underfitting** for their size
  • We trained 2.7B parameters with ~40B tokens (seen); 400B would have been better
    • Unrealistic on a single node
  • What is the best pretraining_INITIALIZATION signal?
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

CodeGen

Key observations:
• NL Scaling is decent, but capped
  • Helpful temperature observations

CodeGen

Key observations:

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

CodeGen

Key observations:
• NL Scaling is decent, but capped
  • Helpful temperature observations
• 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

How to Match Codex

• Data
  • Several 100B tokens required
    • Rarely available for a single programming language; NL initialization works well
    • Language-specific fine-tuning (50GB or more) is key

• Model
  • Performance increases log-linearly with parameters
  • 2B to 6B parameters is a sweet-spot (for now)
    • Low memory footprint enables large batch sizes; performance just 10%-25% shy of Codex
    • Fairly good latency, but needs work

• Resources (for 2.7B parameters)
  • Memory: 2.5TB+ of RAM, for 2M tokens per batch without gradient accumulation
  • Compute: ca. 200 PetaFLOP/s Days ≈ 3 weeks on 64 A100s (at 45% throughput)
  • Both scale linearly with model size; 12B parameters needs 4-5x as much
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](https://openreview.net/pdf?id=rd-G1nO-Jbq)
    - But note: no results on LLMs.

[https://arxiv.org/pdf/2203.15556.pdf](https://arxiv.org/pdf/2203.15556.pdf) -- We used 1.4e21 FLOPs; Chinchilla suggests using that budget to train ~3-4B parameters and ~75B tokens
Outline

Intro to (Foundation) Language Models

State of the Field
Trends, findings, questions

Opportunities

Challenges
What’s Next?

• Breaking free from left-to-right
  • FAIR’s InCoder, Codex edit mode
  • Iteratively refining generations

• New Scaling Frontiers
  • Google’s PaLM

• New Tasks
  • Repair, type prediction, translation
InCoder

- Causal Masking
  - I.e., decoder-only
  - Drop 1+ random spans
  - Infill using placeholders
- Train on Python + S.O.
- Up to 6.7B params

InCoder

- Causal Masking
  - I.e., decoder-only
  - Drop 1+ random spans
  - Infill using placeholders
- Train on Python + S.O.
- Up to 6.7B params

Enables tons of tasks
- Variable naming
- Type inference
- Completion
- Repair

Training

Original Document

```python
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file.""
    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
```

Masked Document

```python
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file.""
    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
```

Zero-shot Inference

Type Inference

```python
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file.""
    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
```

Variable Name Prediction

```python
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file.""
    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
```
InCoder

• Based on Causal Masking
  • Powerful idea! Suffix context is very helpful
  • Probably worth exploring masking strategies beyond Poisson-random on tokens

Codex can do this too

- Not many details
  - Can train like this with encoder/decoder setup (see also (Code)T5)

```python
def get_files(path: str, size: int):
    def prune(dirp, files):
        for file in files:
            file = os.path.join(dirp, file)
            if os.path.getsize(file) > size:
                yield file
        for (dirp, _, files) in os.walk(path):
            yield from prune(dirp, files)
```

Infilling Task

<table>
<thead>
<tr>
<th>Input</th>
<th>She ate [blank] for [blank].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>She ate leftover pasta for lunch.</td>
</tr>
</tbody>
</table>

Our Infilling Framework

- Data: She ate leftover pasta for lunch.
- Input: She ate [blank] for [blank].
- Target: leftover pasta [answer] lunch [answer]

Train

- Data [sep] Target

Infilling

- Input [sep] Target
- Output

Iterating (CodeGen, Austin et al.)

- Who gets everything right the first time?
  - Iteration is surprisingly feasible!
  - Never explicitly trained for, just concatenate history
    - Is that good/bad? Who knows!

PaLM(-Coder) – 535B parameters

**Data:** 780B tokens Google internal web crawl

**Code:** 39B tokens from 24 languages, 39B tokens
  - Mostly Java, HTML, JS
  - Followed by odd fine-tuning regime, mostly Python

**Some Findings:**
  - Even plain PaLM matchers/outperforms Codex 12B
    - Despite training on just 2.7B Python tokens
    - Hard to account for model scale, of course – need new laws
  - Also good at: repair, translation
  - Large models are very likely to memorize code

Outline

Intro to (Foundation) Language Models

State of the Field
Trends, findings, questions

Opportunities

Challenges
The Cost of Scaling

- Academia is priced out
  - Leads to fragmented knowledge

Costs based on approximate PetaFlop seconds at $3/h per V100 GPU

https://twitter.com/Skiminok/status/1512097828373377026 — and just to be clear, I think very highly of Alex

Tbh personally, I find it sad that the know-how of training LMs is spread across multiple competing organizations, hundreds of researchers' & research engineers' brains, and rarely systematically analyzed and compared, let alone written.
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

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:
```python
>>> l = [1,2,3,4,5]
>>> sum(l)
15
```

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
  • … which seems awfully palliative to me

• 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
  • Nothing definitive yet
Questions?

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