Adaptability

Long context and retrieval-augmented code generation

Sean Welleck

Neural Code Generation
Carnegie Mellon University
March 19, 2024

Code generation in 2024

Figure 1: *https://www.youtube.com/watch?v=SSnsmqIj1MI*

Adaptability

Adapt to:

- New code
- New repositories
- New libraries
- New languages
- ...

Adaptability

$$p_\theta(y|x, C)$$

Example contexts C :

- Preceding file contents
- Repository
- New code developed after training
- Documentation, examples, tutorials, etc...
- Discuss: any others?

Adaptability

$$p_\theta(y|x, C)$$

Example contexts C :

- Preceding file contents
  - E.g. HuggingFace modeling_utils.py : $\approx$ 48k tokens
- Repository
  - E.g. `three.js`: $\approx$ 800k tokens

Adaptability

$$p_\theta(y|x, C)$$

Example contexts C :

- Preceding file contents
  - E.g. HuggingFace modeling_utils.py : $\approx$ 48k tokens
- Repository
  - E.g. `three.js`: $\approx$ 800k tokens
- New code developed after training
  - Which data is relevant?
- Documentation, examples, tutorials, etc...
  - Which data is relevant?

Today's lecture

1. Part I: long-context generation
2. Part II: intro to retrieval-augmented generation
    - Case study: retrieving documentation

Today's lecture

1. Part I: long-context generation
2. Part II: intro to retrieval-augmented generation
    - Case study: retrieving documentation

Next week: *Student discussion* on repository-level generation, retrieval augmented generation for code

I. Long-context generation

Long-context generation

- Standard transformers $\rightarrow$ 2048 tokens
- Efficient attention $\rightarrow$ millions of tokens
  - Example recipe: Long-context fine-tuning $\rightarrow$ 100k tokens

Standard transformers

Key problem in standard transformers:

- Memory bottlenecks (e.g., in attention)

Standard transformers | recap

A transformer contains a sequence of ‘transformer blocks’.

- Each block has an **attention** layer and a **feedforward layer**

```python
class Block(nn.Module):
    """ an unassuming Transformer block """

    def __init__(self, config):
        super().__init__()
        self.ln_1 = nn.LayerNorm(config.n_embd)
        self.attn = CausalSelfAttention(config)
        self.ln_2 = nn.LayerNorm(config.n_embd)
        self.mlp = nn.ModuleDict(dict(
            c_fc    = nn.Linear(config.n_embd, 4 * config.n_embd),
            c_proj  = nn.Linear(4 * config.n_embd, config.n_embd),
            act     = NewGELU(),
            dropout = nn.Dropout(config.resid_pdrop),
        ))
        m = self.mlp
        self.mlpf = lambda x: m.dropout(m.c_proj(m.act(m.c_fc(x)))) # MLP forward

    def forward(self, x):
        x = x + self.attn(self.ln_1(x))
        x = x + self.mlpf(self.ln_2(x))
        return x
```

Figure 2: Transformer block from *minGPT*

https://github.com/karpathy/minGPT/blob/master/mingpt/model.py

- Let $Q, K, V \in \mathbb{R}^{s \times d}$, s is sequence length

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

Standard transformers | Problem : quadratic attention

- Let $Q, K, V \in \mathbb{R}^{s \times d}$, s is sequence length

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V$$

For a single query q, write as $\text{Attention}(q, k_1, \ldots, k_s, v_1, \ldots, v_s)$

$$\text{Attention}(q) = \sum_{i=1}^{s} v_i w_i$$

$$\text{where } w_i = \frac{\exp(w_i')}{\sum_{j=1}^{s} \exp(w_j')},$$

$$w_i' = \text{dot}(q, k_i)$$

Intuition: weighted sum of values v, with weights determined by similarity of query and keys

Standard transformers | Problem : quadratic attention

- Let $Q, K, V \in \mathbb{R}^{s \times d}$, s is sequence length

$$\underbrace{\mathrm{Attention}(Q, K, V)}_{\mathbb{R}^{s \times d}} = \underbrace{\mathrm{softmax}\left(\frac{\underbrace{QK^\top}_{\mathbb{R}^{s \times s}}}{\underbrace{\sqrt{d}}_{\mathbb{R}}}\right)}_{\mathbb{R}^{s \times s}} V$$

Standard transformers | Problem : quadratic attention

- Let $Q, K, V \in \mathbb{R}^{s \times d}$, s is sequence length

$$\underbrace{\mathrm{Attention}(Q, K, V)}_{\mathbb{R}^{s \times d}} = \underbrace{\mathrm{softmax}\left(\frac{\overbrace{QK^\top}^{}_{\mathbb{R}^{s \times s}}}{\underbrace{\sqrt{d}}_{\mathbb{R}}}\right)}_{\mathbb{R}^{s \times s}} V$$

$\mathbb{R}^{s \times s}$:

- memory requirement grows quadratically in sequence length s

Accelerators (e.g., CUDA GPU) have *limited memory capacity and bandwidth*:

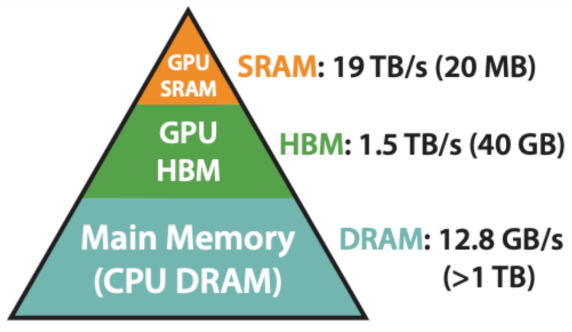

Figure 3: From *FlashAttention* [2]

Standard transformers | Problem : quadratic attention

Accelerators (e.g., CUDA GPU) have *limited memory capacity and bandwidth*:

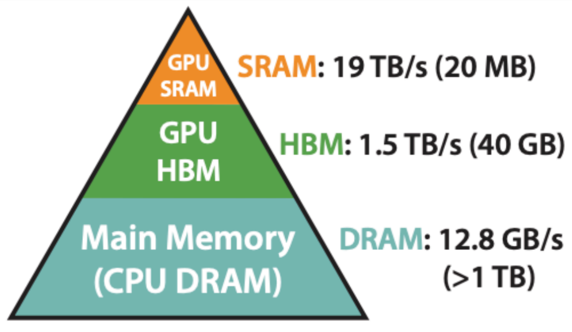

Figure 3: From *FlashAttention* [2]

$\mathbb{R}^{s \times s}$ memory requirement: many slow SRAM $\leftrightarrow$ HBM transfers

Standard transformers | Problem : quadratic attention

Accelerators (e.g., CUDA GPU) have *limited memory capacity and bandwidth*:

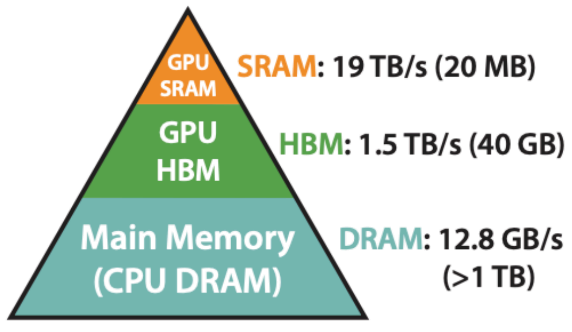

Figure 3: From *FlashAttention* [2]

$\mathbb{R}^{s\times s}$ memory requirement: many slow SRAM $\leftrightarrow$ HBM transfers

- $\therefore$ memory requirement is the practical bottleneck!

Standard transformers | Problem : quadratic attention

Implications:

- Expensive to (pre-)train with a long context length.
  - Example: Code Llama: used 4,096 tokens during pre-training
- Expensive to generate (inference-time)

Standard transformers | Problem : quadratic attention

Implications:

- Expensive to (pre-)train with a long context length.
  - Example: Code Llama: used 4,096 tokens during pre-training
- Expensive to generate (inference-time)

Expensive can mean:

- Slow: bandwidth leads to transfers
- Infeasible: simply run out of memory
  - Example: Inference with batch size 1, hidden size 1024, 100M tokens
    $\implies$ 1000GB memory [8]

Efficient attention

- Standard transformers $\rightarrow$ 2048 tokens
- **Efficient attention $\rightarrow$ millions of tokens**
    - Example: Long-context fine-tuning $\rightarrow$ 100k tokens

Self-attention Does Not Need $O(n^2)$ Memory

A Preprint

Markus N. Rabe and Charles Staats
Google Research
{mrabe,cstaats}@google.com

Memory-efficient attention [Rabe & Staats arXiv 2021] [10]:

- Compute attention in chunks via clever softmax trick
- Avoids full $\mathbb{R}^{s\times s}$ matrix: better memory requirement!

Efficient attention | Memory-efficient attention [Rabe & Staats 2021] [10]

For a single query q, write as $\mathrm{Attention}(q, k_1, \ldots, k_s, v_1, \ldots, v_s)$

$$\mathrm{Attention}(q) = \sum_{i=1}^{s} v_i w_i$$

$$\text{where } w_i = \frac{\exp(w'_i)}{\sum_{j=1}^{s} \exp(w'_j)},$$

$$w'_i = \mathrm{dot}(q, k_i)$$

Efficient attention | Memory-efficient attention [Rabe & Staats 2021] [10]

For a single query q, write as $\mathrm{Attention}(q, k_1, \ldots, k_s, v_1, \ldots, v_s)$

$$\mathrm{Attention}(q) = \sum_{i=1}^{s} v_i w_i$$

$$\text{where } w_i = \frac{\exp(w_i')}{\sum_{j=1}^{s} \exp(w_j')},$$

$$w_i' = \mathrm{dot}(q, k_i)$$

Requires storing s values of w_i', implying $O(s^2)$ for s queries.

Efficient attention | Memory-efficient attention [Rabe & Staats 2021] [10]

For a single query q, write as $\mathrm{Attention}(q, k_1, \ldots, k_s, v_1, \ldots, v_s)$

$$\mathrm{Attention}(q) = \sum_{i=1}^{s} v_i w_i$$

$$\text{where } w_i = \frac{\exp(w_i')}{\sum_{j=1}^{s} \exp(w_j')},$$

$$w_i' = \mathrm{dot}(q, k_i)$$

Requires storing s values of w_i', implying $O(s^2)$ for s queries.

Insight 1:

$$\mathrm{Attention}(q) \equiv \frac{\sum_{i=1}^{s} v_i \exp(w_i')}{\sum_{j=1}^{s} \exp(w_j')}$$

Efficient attention | Memory-efficient attention [Rabe & Staats 2021] [10]

For a single query q, write as $\mathrm{Attention}(q, k_1, \ldots, k_s, v_1, \ldots, v_s)$

$$\mathrm{Attention}(q) = \sum_{i=1}^{s} v_i w_i$$

$$\text{where } w_i = \frac{\exp(w_i')}{\sum_{j=1}^{s} \exp(w_j')},$$

$$w_i' = \mathrm{dot}(q, k_i)$$

Requires storing s values of w_i', implying $O(s^2)$ for s queries.

Insight 1:

$$\mathrm{Attention}(q) \equiv \frac{\sum_{i=1}^{s} v_i \exp(w_i')}{\sum_{j=1}^{s} \exp(w_j')}$$

Insight 2:

- Compute by iteratively adding to a vector $v^* \in \mathbb{R}^d$ and $w^* \in \mathbb{R}$. Less memory!

Efficient attention | Memory-efficient attention [Rabe & Staats 2021] [10]

Specifically, compute softmax normalization online [9]:

For each query q:

For $i = 1, \ldots, s$:[1]

$$w_i' = \mathrm{dot}(q, k_i)$$
$$v^* \leftarrow v^* + v_i \exp(w_i')$$
$$s^* \leftarrow s^* + \exp(w_i')$$

At the end, $\mathrm{Attention}(q) = v^* / s^*$.

[1] In general, segment into "chunks" (aka "blocks" / "tiles"). See paper for full version.

~ 1 million tokens

Sequence length	$n = 2^8$	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Size of inputs and outputs	160KB	640KB	2.5MB	10MB	40MB	160MB	640MB
Memory overhead of standard attention	270KB	4.0MB	64MB	1GB	OOM	OOM	OOM
Memory overhead of memory-eff. attn.	270KB	4.0MB	16MB	17MB	21MB	64MB	256MB
Compute time on TPUv3	0.06ms	0.11ms	0.7ms	11.3ms	177ms	2.82s	45.2s
Relative compute speed	±5%	±5%	-8±2%	-13±2%	-	-	-

Table 2: Memory and time requirements of self-attention during **inference**.

Figure 4: Inference benchmarking

Efficient attention | Memory-efficient attention [Rabe & Staats 2021] [10]

Sequence length	$n = 2^8$	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Size of inputs and outputs	192KB	768KB	2.9MB	12MB	47MB	188MB	750MB
Memory overhead of standard attention	532KB	8.0MB	128MB	2.0GB	OOM	OOM	OOM
Memory overhead of memory-eff. attn.	532KB	8.0MB	41MB	64MB	257MB	1.0GB	4.0GB
Compute time on TPUv3	0.1ms	0.18ms	1.4ms	21ms	336ms	5.3s	85s
Relative compute speed	$\pm 5\%$	$\pm 5\%$	-30$\pm$5%	-35$\pm$5%	-	-	-

Table 3: Memory and time requirements of self-attention during **differentiation**. Note that the slowdown in compute speed is expected due to the use of checkpointing in memory-efficient attention.

Figure 5: Differentiation (proxy for training) benchmarking

Memory-efficient attention | extensions

Memory-efficient attention [Rabe & Staats 2021 [10]]:

- *FlashAttention*: Analogy via new CUDA kernel [Dao et al 2022 [2]]
- *Blockwise Transformers*: also chunk feedforward layer [Liu et al 2023 [7]]
- Ring Attention: Distribute chunks across devices [Liu et al 2023 [8]]

Efficient attention | Ring Attention [Liu, Zaharia, Abbeel arXiv 2023] [8]

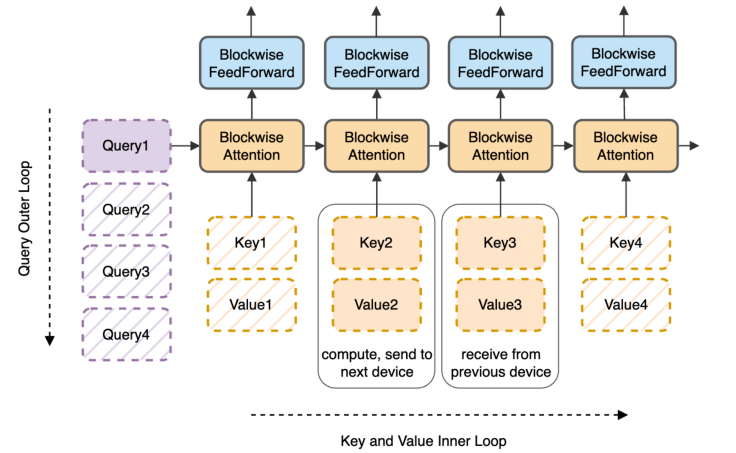

Figure 6: From [8]: each host holds one query block, and key-value blocks traverse through a ring of hosts in a block-by-block fashion.

Figure 7: From [8]. Maximum *training* context size on TPUv4-1024 with vanilla, efficient attention variants, and Ring Attention.

Efficient attention | Ring Attention [Liu, Zaharia, Abbeel arXiv 2023] [8]

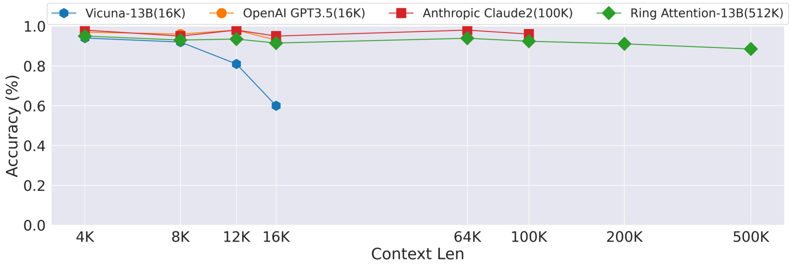

Figure 3: Comparison of different models on the long-range line retrieval task.

Figure 8: Long-range line retrieval task

Long-context generation

- Standard transformers $\rightarrow$ 2048 tokens
- Efficient attention $\rightarrow$ millions of tokens
    - Example adaptation recipe: $\rightarrow$ 100k tokens

Long-context generation

Effective Long-Context Scaling of Foundation Models (Meta 2023)

- Llama / analysis
- + ideas used in CodeLlama (Meta 2023)

Long context fine-tuning of [code-]llama

Pretrain with short context (e.g. 4,096), adapt to long contexts

- Positional embeddings
- Data

Long context fine-tuning | positional embeddings

RoPe embeddings map inputs to a sphere, with a periodic structure

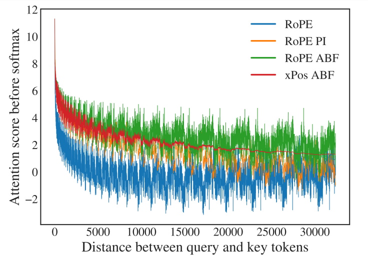

Figure 9: Vanilla RoPE naturally decays as s increases

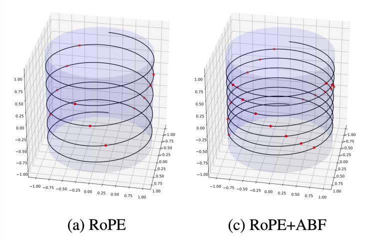

Figure 10: Solution: increase frequency (RoPE ABF)

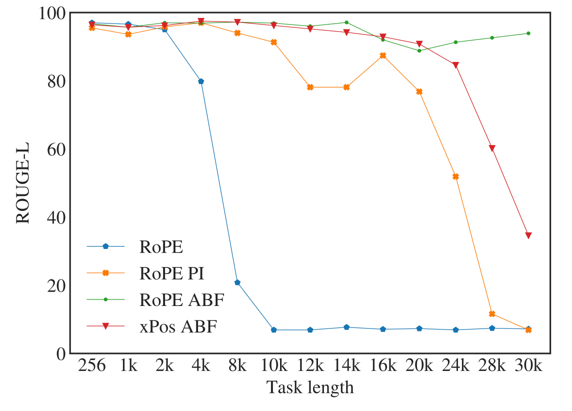

Figure 11: Fixes degradation on diagnostic task

Long context fine-tuning Llama | data

Data:

- Continue pretraining on special mix of 400B tokens
- Finetune on long instruction-tuning data

Long context fine-tuning Llama | data

Special mix: “new long text data” and up-weight long data samples

Continual Pretrain Data	NarrativeQA Δ F1	Qasper Δ F1	Quality Δ EM	QMSum Δ ROUGE-geo
Llama 2 Long data mix	**23.70%**	**43.64%**	**75.5%**	**45.70%**
Llama 2 data mix	18.23%	38.12%	60.3%	44.87%
- remove long text data	19.48%	39.14%	67.1%	36.60%
- upsample existing long text data	22.15%	36.82%	65.0%	42.83%

Table 7: Comparison of different pretraining data mix on long-context tasks. Instead of showing the absolute performance, we report relative improvements over the 7B Llama 2 which has a 4,096-token context window. All models are evaluated with prompts truncated at 16,384 tokens.

Long context fine-tuning Llama | data

Synthetic long + concatenated short instruction data:

Settings	Qasper	NarrativeQA	QuALITY	SummScreenFD	QMSum
Llama 2 Chat baseline	12.2	9.13	56.7	10.5	14.4
Llama 2 Long *finetuned* with:					
"RLHF V5"	22.3	13.2	71.4	14.8	16.9
"RLHF V5" mix pretrain	23.7	16.6	76.2	**15.7**	17.8
"RLHF V5" mix self-inst w/o LM loss	35.7	22.3	59.3	12.2	13.4
"RLHF V5" mix self-inst with LM loss	**38.9**	**23.3**	**77.3**	14.5	**18.5**

Table 9: Comparison of different instruction finetuning data mixes.

Mixed result for CodeLlama:

Synthetic Key Retrieval Task. We prompt the model with a variable number of tokens by concatenating Python solutions from the CodeContest dataset (Li et al., 2022), which results in syntactically valid source code. At a specified relative position within the prompt, we insert the following key, where `<VALUE>` is a two-digit number that is randomly sampled based on the overall number of tokens in the prompt:

```python
def my_function() -> int:
    """Note that this function is used at the end
    """
    return <VALUE>
```

Model	Size	8,000			16,000			24,000		
		0	0.2	0.4	0	0.2	0.4	0	0.2	0.4
Code Llama	7B	100.0	95.3	100.0	54.7	100.0	98.4	3.1	85.9	85.9
Code Llama	13B	100.0	100.0	100.0	100.0	100.0	100.0	100.0	89.1	6.3
Code Llama	34B	76.6	100.0	100.0	95.3	96.9	100.0	81.3	0.0	81.3
Code Llama - Instruct	7B	100.0	97.7	100.0	7.0	96.9	96.1	0.0	62.5	54.7
Code Llama - Instruct	13B	100.0	100.0	100.0	100.0	100.0	93.8	4.7	84.4	100.0
Code Llama - Instruct	34B	92.2	100.0	100.0	68.8	95.3	100.0	46.9	0.0	85.9
gpt-3.5-turbo-16k-0630	-	100.0	100.0	95.3	95.3	90.6	98.4	-	-	-

Context Length / Key Position

Table 17: **Function Key Retrieval Accuracy (%) for Code Llama models**.

Other methods

Long sequence modeling and generation is an active research area:

- Alternative models: state-space models, …
- Proprietary approaches
- …

github.com/gkamradt/LLMTest_NeedleInAHaystack

LLMTest_NeedleInAHaystack Public

The Test

1. Place a random fact or statement (the 'needle') in the middle of a long context window (the 'haystack')
2. Ask the model to retrieve this statement
3. Iterate over various document depths (where the needle is placed) and context lengths to measure performance

Figure 12: *https://github.com/gkamradt/LLMTest_NeedleInAHaystack*

Black box models

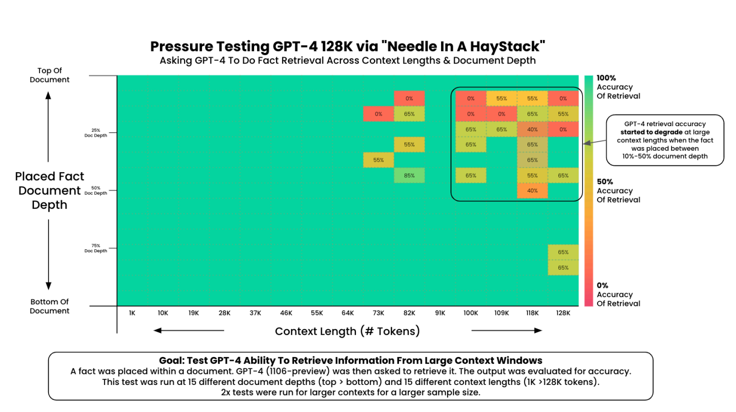

Figure 13: OpenAI's GPT-4-128K (Run 11/8/2023); from *https://github.com/gkamradt/LLMTest_NeedleInAHaystack*

Black box models

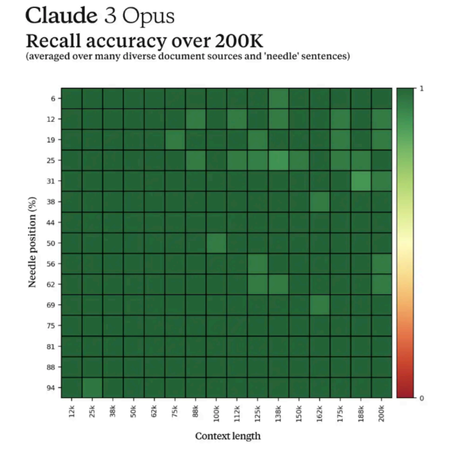

Figure 14: Claude 3 (March 4 2024); from *https://www.anthropic.com/news/claude-3-family*

Black box models

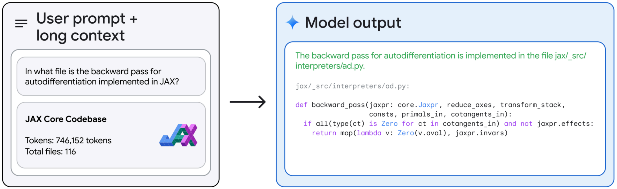

Figure 2 | Given the entire 746,152 token JAX codebase in context, Gemini 1.5 Pro can identify the specific location of a core automatic differentiation method.

Figure 15: Gemini 1.5 (Feb 2024)

Black box models

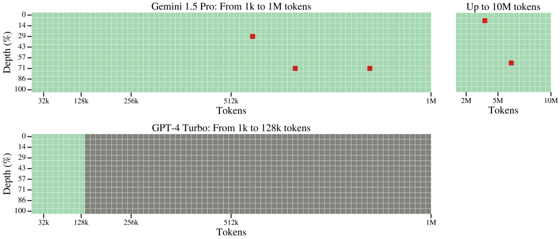

Figure 7 | **Text Haystack.** This figure compares Gemini 1.5 Pro with GPT-4 Turbo for the text needle-in-a-haystack task. Green cells indicate the model successfully retrieved the secret number, gray cells indicate API errors, and red cells indicate that the model response did not contain the secret number. The top row shows results for Gemini 1.5 Pro, from 1k to 1M tokens (top left), and from 1M to 10M tokens (top right). The bottom row shows results on GPT-4 Turbo up to the maximum supported context length of 128k tokens. The results are color-coded to indicate: green for successful retrievals and red for unsuccessful ones.

Figure 16: Gemini 1.5 (Feb 2024)

Black box models

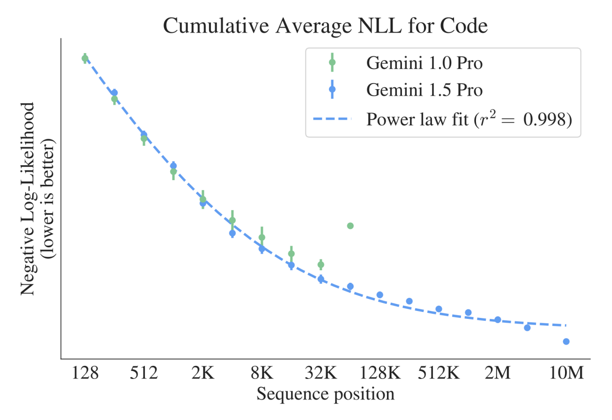

Figure 17: Gemini 1.5 (Feb 2024)

Recap

$$p(y|x, C)$$

- Approach 1: place a long context C in the input of a LM
  - Use techniques to extend the context size of transformers

Observation: how do we select a context?

II. Retrieval-augmented generation

Retrieval-augmented generation

- x: input (e.g. query), y: output (e.g. code)
- $\mathcal{Z}$: data store of items (e.g. code snippets)

$$
\begin{aligned}
p(y|x) &= \sum_{Z_k \subset \mathcal{Z}} p(y|x, Z_k) p(Z_k|x) \\
&\approx \arg\max_{Z_k} p(y|x, Z_k) p(Z_k|x) \\
&\approx p(y|x, \hat{Z}_k) p(\hat{Z}_k|x),
\end{aligned}
$$

Retrieval-augmented generation

- x: input (e.g. query), y: output (e.g. code)
- $\mathcal{Z}$: data store of items (e.g. code snippets)

$$
\begin{aligned}
p(y|x) &= \sum_{Z_k \subset \mathcal{Z}} p(y|x, Z_k)p(Z_k|x) \\
&\approx \arg\max_{Z_k} p(y|x, Z_k)p(Z_k|x) \\
&\approx p(y|x, \hat{Z}_k)p(\hat{Z}_k|x),
\end{aligned}
$$

where

$$\hat{Z}_k = \mathrm{topk}_{z_i \in \mathcal{Z}} s_\phi(x, z_i)$$

is a set of *retrieved* items using scorer $s_\phi(x, z)$.

Retrieval-augmented generation

Concretely:

1. Specify a **datastore** $\mathcal{Z}$ of items to retrieve z
2. **Retriever:** map a query x to ranked list $z_1, z_2, z_3, \ldots$
3. **Generator:** use top-ranked items, $p_\theta(y|x, \{z_k\}_{k=1}^{K})$

Retrieval-augmented generation

Concretely:

1. Specify a **datastore** $\mathcal{Z}$ of items to retrieve z
2. **Retriever:** map a query x to ranked list $z_1, z_2, z_3, \ldots$
3. **Generator:** use top-ranked items, $p_\theta(y|x, \{z_k\}_{k=1}^K)$

Key feature: $\mathcal{Z}$ can have new code, relevant documents, etc...

Example retriever: *BM-25*

- Score text based on term (n-gram) frequencies and length
- Doesn't use a neural model

Retrieval-augmented generation | retrievers

First thing to do is create an instance of the BM25 class, which reads in a corpus of text and does some indexing on it:

```
from rank_bm25 import BM25Okapi

corpus = [
    "Hello there good man!",
    "It is quite windy in London",
    "How is the weather today?"
]

tokenized_corpus = [doc.split(" ") for doc in corpus]

bm25 = BM25Okapi(tokenized_corpus)
# <rank_bm25.BM25Okapi at 0x1047881d0>
```

Now that we've created our document indexes, we can give it queries and see which documents are the most relevant:

```
query = "windy London"
tokenized_query = query.split(" ")

doc_scores = bm25.get_scores(tokenized_query)
# array([0.        , 0.93729472, 0.        ])
```

Instead of getting the document scores, you can also just retrieve the best documents with

```
bm25.get_top_n(tokenized_query, corpus, n=1)
# ['It is quite windy in London']
```

Figure 18: Usage example from https://github.com/dorianbrown/rank_bm25

Example retriever: *dense passage retriever* (DPR) [6]

- Encoder

$$s_\phi(x) \to \mathbb{R}^d$$

- Score

$$s(x, z_i) = s_\phi(x)^\top s_\phi(z_i) \in \mathbb{R}$$

- Retrieval

$$\texttt{top-}K(s(x, z_1), s(x, z_2), \ldots, s(x, z_{|\mathcal{Z}|}))$$

Example retriever: *dense passage retriever* (DPR) [6]

- Encoder

$$s_\phi(x) \to \mathbb{R}^d$$

- Score

$$s(x, z_i) = s_\phi(x)^\top s_\phi(z_i) \in \mathbb{R}$$

- Retrieval

$$\texttt{top-}K(s(x, z_1), s(x, z_2), \ldots, s(x, z_{|\mathcal{Z}|}))$$

In practice, precompute all $s(z_i)'$s and use a fast maximum inner-product search (MIPS) library (e.g., `faiss` (https://github.com/facebookresearch/faiss)).

Example retriever: *dense passage retriever* (DPR)

- Neural encoder $s_\phi(x) \to \mathbb{R}^d$ (e.g., BERT)
- Train with *InfoNCE loss* [11]

$$L(x, z^+, Z^-) = -\log \frac{\exp(s(x, z^+))}{\exp(s(x, z^+)) + \sum_{z^- \in Z^-} \exp(s(x, z^-))}$$

As negatives, use other examples in the current minibatch [4, 3]

Task-aware retrieval with instructions (TART) [1]:

- Single task retriever $\rightarrow$ instruction-tuned retriever
- Condition on a task-relevant instruction: $s_\phi : (x, I) \rightarrow z_e$

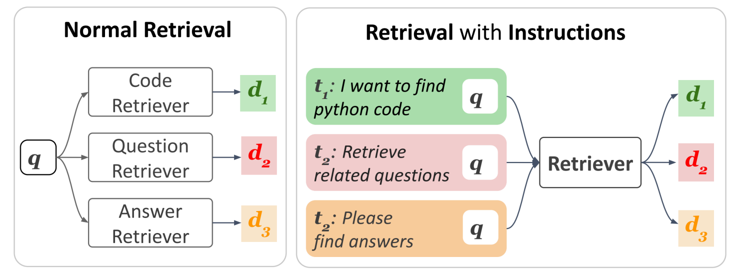

Figure 19: *Task Aware Retrieval with Instructions*, Asai et al. ACL 2023

Retrieval-augmented generation | retrievers

LLM-based retrieval with instructions:

- Adapt LLM (e.g. Mistral 7b) to produce embeddings $z_e \in \mathbb{R}^d$
- Condition on a task-relevant instruction $s_\phi : (x, I) \rightarrow z_e$

Spaces mteb/leaderboard like 2.2k Running App Files Community 84

Massive Text Embedding Benchmark (MTEB) Leaderboard. To submit, refer to the MTEB GitHub repository Refer to the MTEB paper for details on metrics, tasks and models.

Overall | Bitext Mining | Classification | Clustering | Pair Classification | Reranking | Retrieval | STS | Summarization

English | Chinese | French | Polish

Overall MTEB English leaderboard

- **Metric:** Various, refer to task tabs
- **Languages:** English

Rank	Model	Model Size (GB)	Embedding Dimensions	Max Tokens	Average (56 datasets)	Classification Average (12 datasets)	Clustering Average (11 datasets)	Pair Classification Average (3 datasets)
1	SFR-Embedding-Mistral	14.22	4096	32768	67.56	78.33	51.67	88.54
2	voyage-lite-02-instruct		1024	4000	67.13	79.25	52.42	86.87
3	GritLM-7B	14.48	4096	32768	66.76	79.46	50.61	87.16
4	e5-mistral-7b-instruct	14.22	4096	32768	66.63	78.47	50.26	88.34
5	GritLM-8x7B	93.41	4096	32768	65.66	78.53	50.14	84.97

Example: Retrieval with embeddings from **SFR-Embedding-Mistral**

```
from datasets import load_dataset
from sentence_transformers import SentenceTransformer, util

model = SentenceTransformer("Salesforce/SFR-Embedding-Mistral", device="cuda")

ds = load_dataset("openai_humaneval")
datastore = [x['canonical_solution'] for x in ds['test']]
prompts = [x['prompt'] for x in ds['test']]

query = """Instruct: Given a search query, return relevant source code.
Query: code involving Fibonacci numbers"""

embeddings = model.encode([query] + datastore)
scores = util.cos_sim(embeddings[:1], embeddings[1:]) * 100

k = 3

for query_i, scores_i in zip(queries, scores):
    retrieved = [prompts[i] + datastore[i] for i in scores_i.argsort(descending=True)[:k]]
    for item in retrieved:
        print('======== QUERY: ' + query_i)
        print(item)
        print('-----------\n')
```

Retrieval-augmented generation | retrievers

Example: Retrieval with embeddings from SFR-Embedding-Mistral

```
======== QUERY: Instruct: Given a search query, return relevant source code.
Query: code involving Fibonacci numbers


def fib(n: int):
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)

-----------

======== QUERY: Instruct: Given a search query, return relevant source code.
Query: code involving Fibonacci numbers


def prime_fib(n: int):
    """
    prime_fib returns n-th number that is a Fibonacci number and it's also prime.
    >>> prime_fib(1)
    2
    >>> prime_fib(2)
    3
    >>> prime_fib(3)
    5
    >>> prime_fib(4)
    13
    >>> prime_fib(5)
    89
    """
    import math

    def is_prime(p):
        if p < 2:
            return False
        for k in range(2, min(int(math.sqrt(p)) + 1, p - 1)):
            if p % k == 0:
                return False
        return True
    f = [0, 1]
    while True:
```

Case study: retrieving *documentation* for code generation

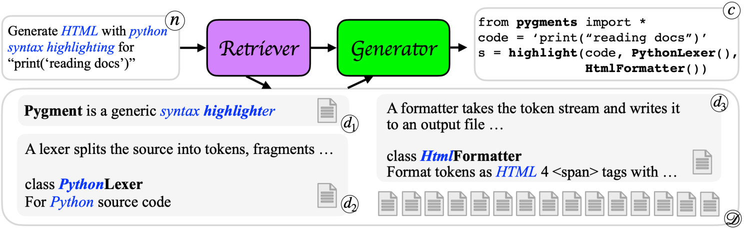

Figure 1: DocPrompting: given an NL intent ⓝ, the retriever retrieves a set of relevant documentation {d_1, d_2, d_3} from a documentation pool $\mathcal{D}$. Then, the generator generates the code ⓒ based on the NL and retrieved docs. DocPrompting allows the model to generalize to previously unseen usages by reading those docs. *Italic blue* highlights the shared tokens between NL and docs; **Bold** shows shared tokens between docs and the code snippet.

Figure 20: *DocPrompting: Generating Code by Retrieving the Docs*, Zhou et al. ICLR 2023

Retrieval-augmented generation | DocPrompting

- Retriever:
    - Dense (CodeT5 encoder, InfoNCE loss + in-batch negatives)
- Generator:
    - Finetune CodeT5 with fusion-in-decoder [5]
        - Attends over k encoded retrieved docs

Retrieval-augmented generation | DocPrompting

- Retriever:
  - Dense (CodeT5 encoder, InfoNCE loss + in-batch negatives)
- Generator:
  - Finetune CodeT5 with fusion-in-decoder [5]
    - Attends over k encoded retrieved docs
- Task:
  - StackOverflow questions $\rightarrow$ Python (CoNaLa dataset)
  - Split data so that test problems have **at least 1 unseen function**
- Datastore:
  - Docs for all Python functions on *`devdocs.io`*

Retrieval-augmented generation for code | DocPrompting

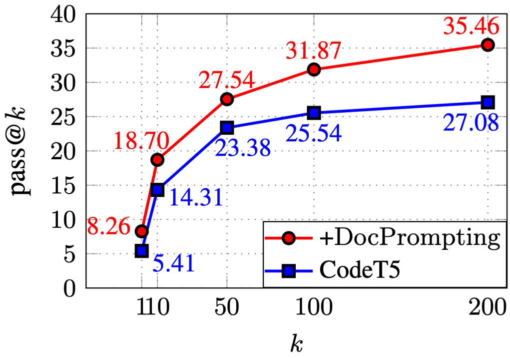

Figure 3: Pass@k of CodeT5 with and without DocPrompting on 100 CoNaLa examples.

Figure 21: Pass@k on CoNaLa

Retrieval-augmented generation for code | DocPrompting

Table 5: Examples of predictions from `CoNaLa`, of the base CodeT5 compared to CodeT5+DocPrompting. Unseen functions are underscored.

NL Intent: **Open image "picture.jpg"**	
Ground truth:	`img = Image.open('picture.jpg') \n Img.show`
CodeT5:	`os.open('picture.jpg', 'r')`
CodeT5+DocPrompting:	`image = Image.open('picture.jpg', 'rb')`
NL Intent: **Exclude column names when writing dataframe 'df' to a csv file 'filename.csv'**	
Ground truth:	`df.to_csv ('filename.csv', header=False)`
CodeT5:	`df.drop(['col1', 'col2'], axis=1, inplace=True)`
CodeT5+DocPrompting:	`df.to_csv('filename.csv', skiprows=1)`

Figure 22: Qualitative example

Retrieval-augmented generation for code | DocPrompting

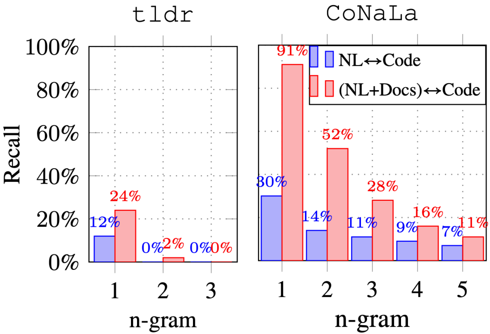

Figure 4: Using documentation significantly increases the n-gram overlap recall between the input and the output, in `tldr` and `CoNaLa`.

Figure 23: Analysis: why does reading docs help generate code (with these models)?

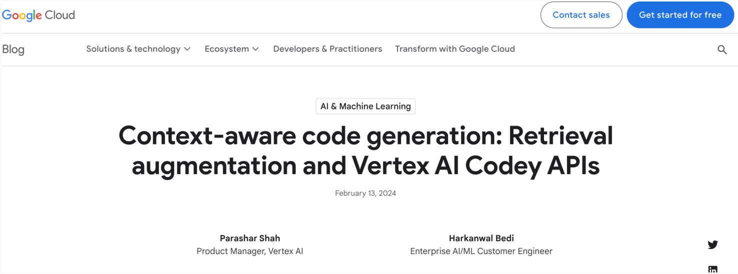

Figure 24: RAG with Blackbox API models (Feb 2024) [a]

[a] https://cloud.google.com/blog/products/ai-machine-learning/context-aware-code-generation-rag-and-vertex-ai-codey-apis

Prompt: "Create python function that takes a prompt and predicts using langchain.llms interface for VertexAI text-bison model"

Output without RAG:

```
[86] response = code_llm.predict(text=user_question, max_output_tokens=2048, temperature=0.1)
     print(response)

     ```python
     import langchain.llms as llms

     def predict(prompt):
       # Load the text-bison model
       model = llms.TextBisonModel()

       # Generate predictions
       predictions = model.predict(prompt, num_return_sequences=3)

       # Return the predictions
       return predictions
     ```
```

Figure 2: Output from the model without any external context

Figure 25: RAG with Blackbox API models (Feb 2024) [a]

[a] https://cloud.google.com/blog/products/ai-machine-learning/context-aware-code-generation-rag-and-vertex-ai-codey-apis

Retrieval-augmented generation for code | Blackbox Models

Output with RAG:

```
[87] results = qa_chain({"query": user_question})
     print(results["result"])
```

````
```python
def predict_with_langchain_llms(prompt):
  """Predicts using langchain.llms interface for VertexAI text-bison model.

  Args:
    prompt: The prompt to predict.

  Returns:
    The prediction.
  """

  # Initialize the VertexAI LLMs client.
  llm = VertexAI(
      model_name="text-bison-32k",
      max_output_tokens=256,
      temperature=0.1,
      top_p=0.8,
      top_k=40,
      verbose=True,
  )

  # Predict the prompt.
  prediction = llm.predict(prompt)

  return prediction

```
````

Figure 26: RAG with Blackbox API models (Feb 2024) [a]

Recap

$$p(y|x, C)$$

- Approach 1: place a long context C in the input of a LM
  - Use techniques to expand the context size of transformers
- Approach 2: retrieve relevant contexts, place in the input of a LM
  - Define a datastore and retriever

A. Asai, T. Schick, P. Lewis, X. Chen, G. Izacard, S. Riedel, H. Hajishirzi, and W.-t. Yih.
Task-aware retrieval with instructions.
In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Findings of the Association for Computational Linguistics: ACL 2023*, pages 3650–3675, Toronto, Canada, July 2023. Association for Computational Linguistics.

T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Re.
Flashattention: Fast and memory-efficient exact attention with IO-awareness.
In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

D. Gillick, S. Kulkarni, L. Lansing, A. Presta, J. Baldridge, E. Ie, and D. Garcia-Olano.
Learning dense representations for entity retrieval, 2019.

M. Henderson, R. Al-Rfou, B. Strope, Y.-H. Sung, L. Lukács, R. Guo, S. Kumar, B. Miklos, and R. Kurzweil.
Efficient natural language response suggestion for smart reply.
ArXiv, abs/1705.00652, 2017.

G. Izacard and E. Grave.
Leveraging passage retrieval with generative models for open domain question answering, 2020.

V. Karpukhin, B. Oguz, S. Min, P. Lewis, L. Wu, S. Edunov, D. Chen, and W.-t. Yih.
Dense passage retrieval for open-domain question answering.
In B. Webber, T. Cohn, Y. He, and Y. Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, Nov. 2020. Association for Computational Linguistics.

H. Liu and P. Abbeel.
Blockwise parallel transformer for large context models, 2023.

H. Liu, M. Zaharia, and P. Abbeel.
Ring attention with blockwise transformers for near-infinite context, 2023.

References iv

M. Milakov and N. Gimelshein.
Online normalizer calculation for softmax.
ArXiv, abs/1805.02867, 2018.

M. N. Rabe and C. Staats.
Self-attention does not need $o(n^2)$*memory*.
2021.

A. van den Oord, Y. Li, and O. Vinyals.
Representation learning with contrastive predictive coding.
ArXiv, abs/1807.03748, 2018.