Lecture 19: Introduction to LLMs

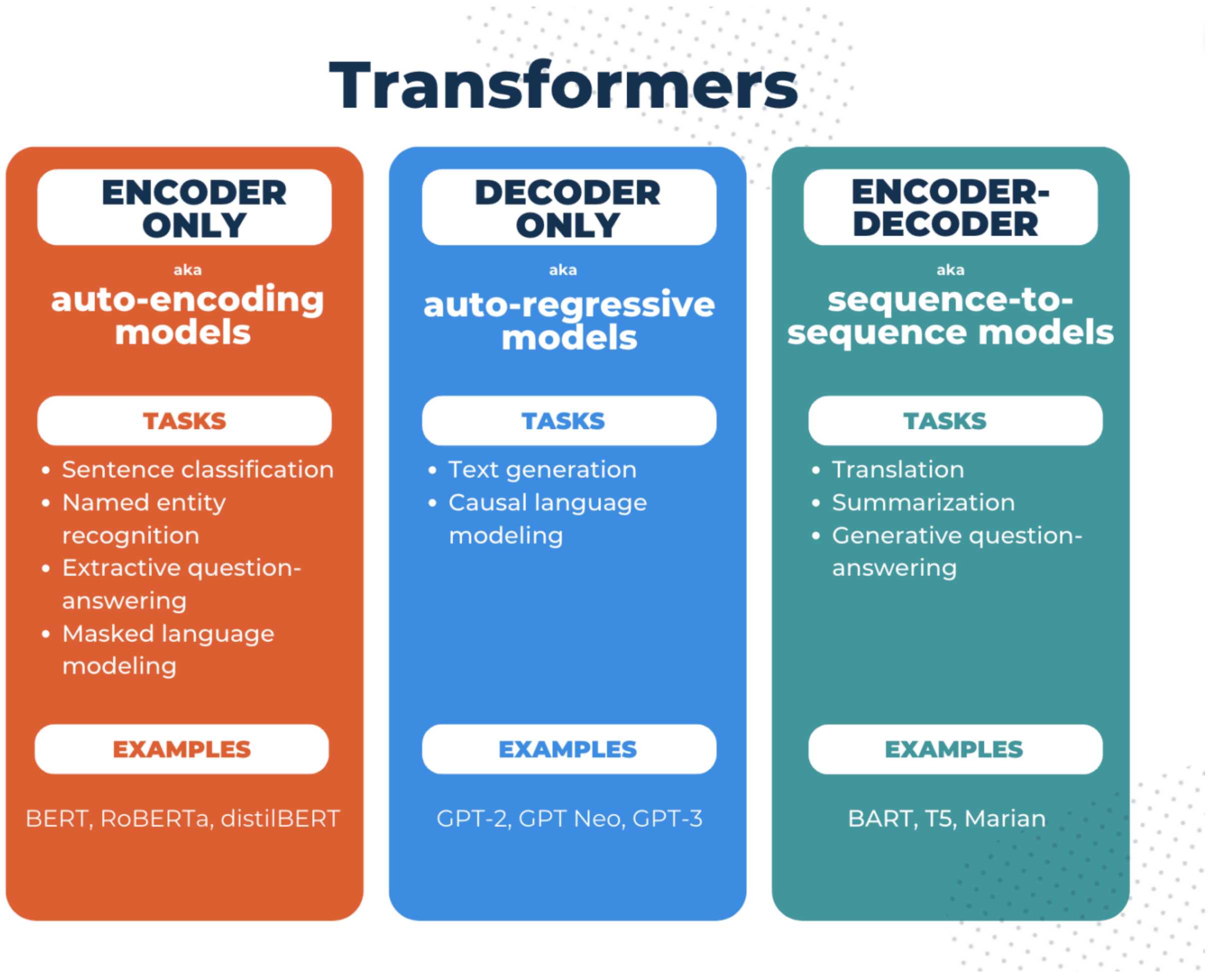
Applied Machine Learning

Brandon Amos
Cornell Tech

Preface and disclaimer ⚠

- Language, NLP, LLMs is a huge space. Many great resources out there!
- **This lecture**
    1. Tour through my favorite introductory parts from them
    2. Some code examples to show how to apply and use
        a) **basic tokenization** and **autoregressive generation**,
        b) **chat templates**, and
        c) **code completion** (fill-in-the-middle)

Transformers and language models: ubiquitous



Transformers

ENCODER ONLY

aka

auto-encoding models

TASKS

- Sentence classification
- Named entity recognition
- Extractive question-answering
- Masked language modeling

EXAMPLES

BERT, RoBERTa, distilBERT

DECODER ONLY

aka

auto-regressive models

TASKS

- Text generation
- Causal language modeling

EXAMPLES

GPT-2, GPT Neo, GPT-3

ENCODER-DECODER

aka

sequence-to-sequence models

TASKS

- Translation
- Summarization
- Generative question-answering

EXAMPLES

BART, T5, Marian

Image sources: Explainable AI: Visualizing Attention in Transformers

Review on classification

$$\underbrace{\text{Dataset}}_{\text{Features, Attributes, Targets}} + \underbrace{\text{Learning Algorithm}}_{\text{Model Class + Objective + Optimizer}} \rightarrow \text{Predictive Model}$$

---

1. Training dataset $\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}$.
2. The target space is discrete: $\mathcal{Y} = \{y_1, y_2, \ldots y_K\}$.

   Each of the K discrete values corresponds to a *class* that we want to predict
3. Optimize the conditional likelihood

$$\max_{\theta} \ell(\theta) = \max_{\theta} \frac{1}{N} \sum_{i=1}^{N} \log P_{\theta}(y^{(i)} | x^{(i)}).$$

LLMs (for generation) are ”just” doing next-token classification

- Represent language as a sequence of **discrete tokens**
- Given the past sequence of text $x^{(i)}$, classify the next portion $y(i)$.
- Parameterize P_θ with a sequence architecture (e.g., a transformer)
- (Pre)train with maximum likelihood

$$\max_\theta \ell(\theta) = \max_\theta \frac{1}{N} \sum_{i=1}^{N} \log P_\theta(y^{(i)}|x^{(i)}).$$

Tokenization and representing language

- **Tokenization** is how the string is represented (what the K values correspond to)
- Dataset has token strings $x^{(i)} \in \{1, \ldots, K\}^{n_i}$ and next tokens $y^{(i)} \in \{1, \ldots, K\}$:

$$\mathcal{D} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(N)}, y^{(N)})\}$$

- Many options for how to tokenize a sequence, e.g.:

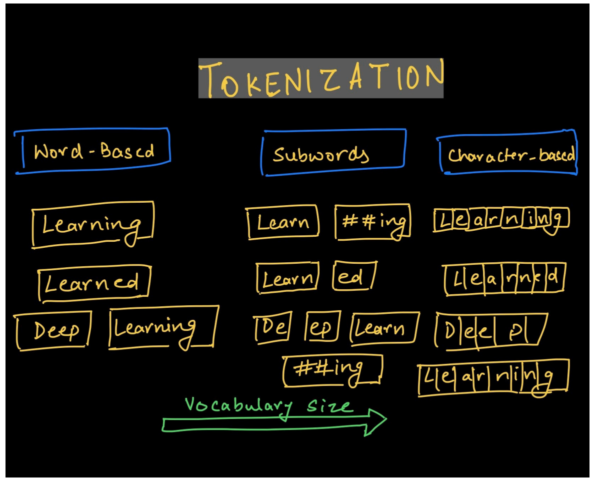


Image source: Character vs. Word Tokenization in NLP

Tokenization in practice

- A large topic and very important choice
- Tokens often learned via Byte-Pair Encoding, SentencePiece, or WordPiece
- Many other great resources:
    - HuggingFace Tokenizer Summary
    - Let's build the GPT Tokenizer (code)
    - Llama tokenizer visualizer

Applications of transformers

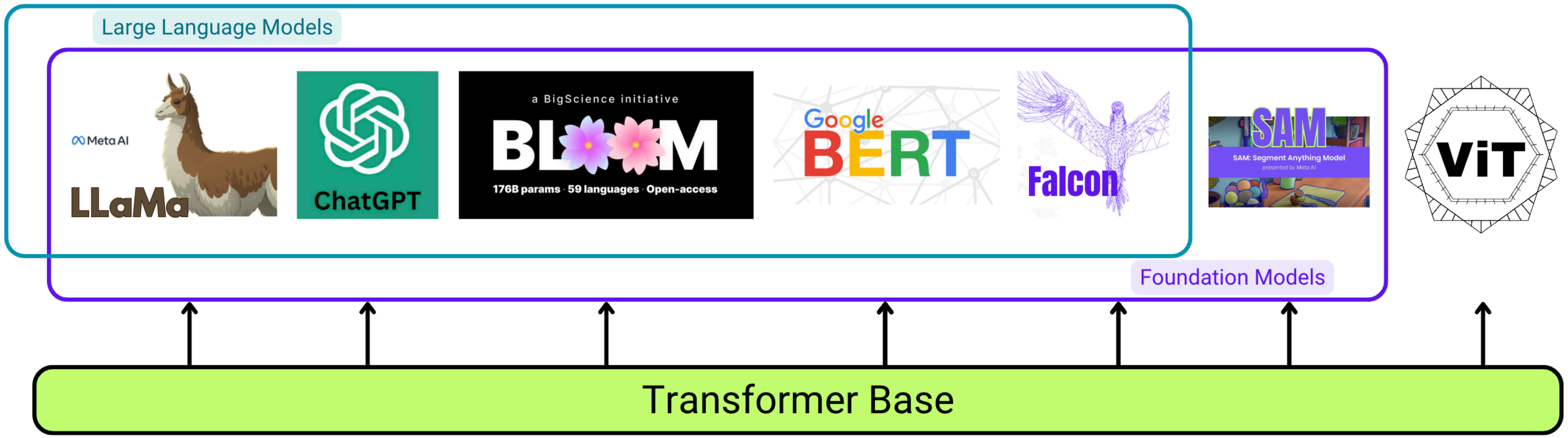
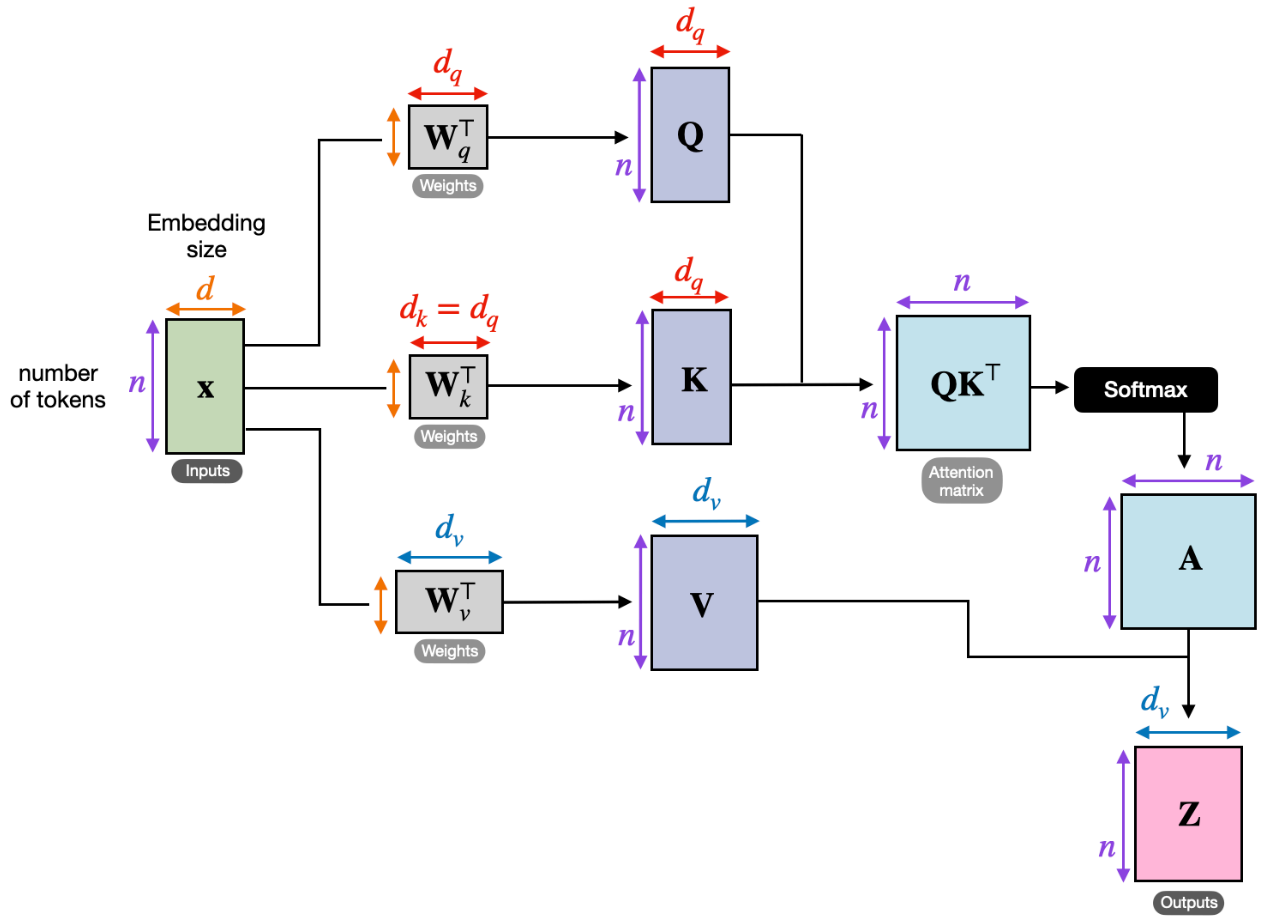


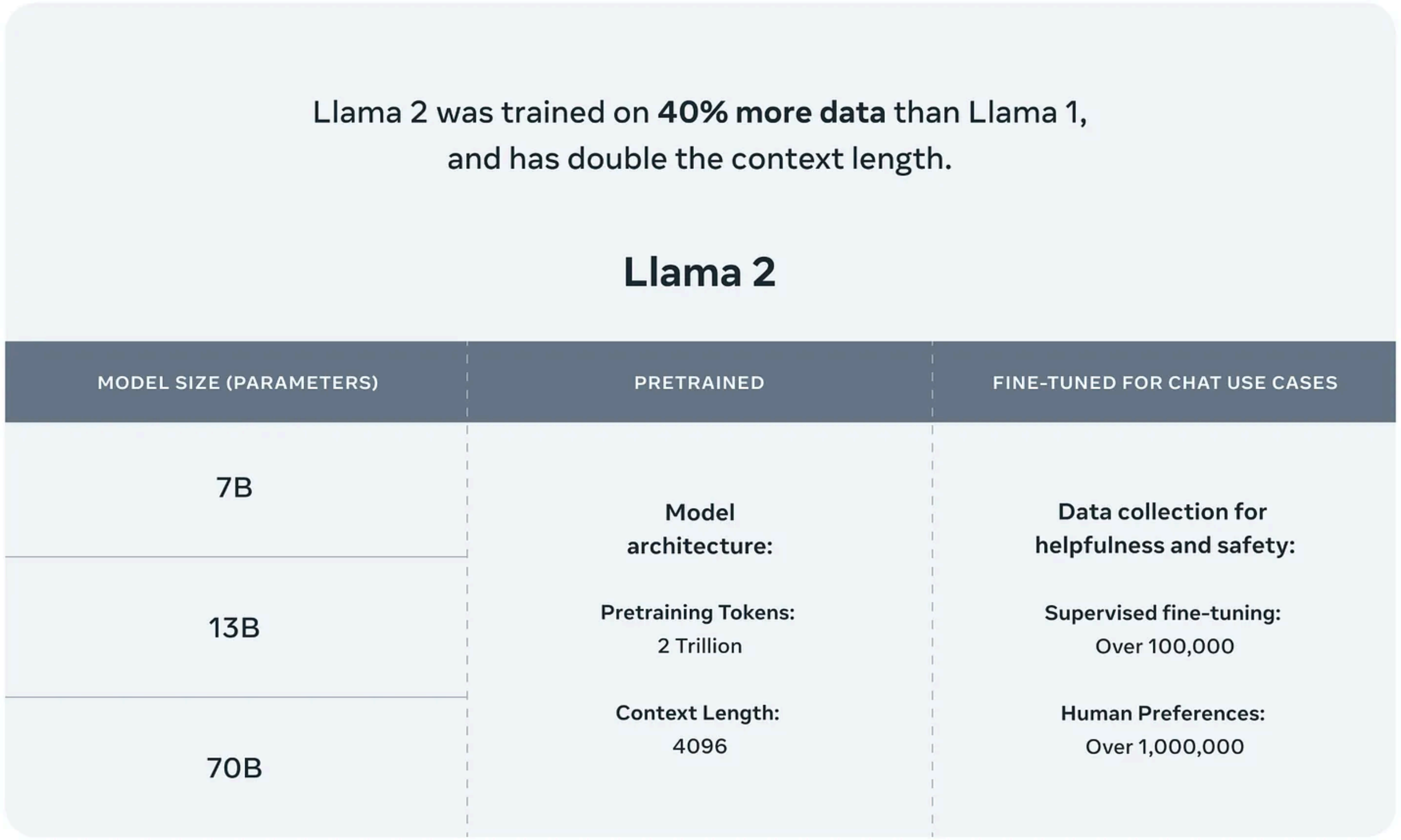
Image source: Explainable AI: Visualizing Attention in Transformers

What attention looks like



Source: Understanding and Coding the Self-Attention Mechanism of LLMs From Scratch

Putting everything together



Llama 2 was trained on **40% more data** than Llama 1, and has double the context length.

Llama 2

MODEL SIZE (PARAMETERS)	PRETRAINED	FINE-TUNED FOR CHAT USE CASES
7B	**Model architecture:** **Pretraining Tokens:** 2 Trillion **Context Length:** 4096	**Data collection for helpfulness and safety:** **Supervised fine-tuning:** Over 100,000 **Human Preferences:** Over 1,000,000
13B		
70B		

(Source: the Llama 2 paper)

Full architectures

Combine many components we've covered: embeddings, attention, residual

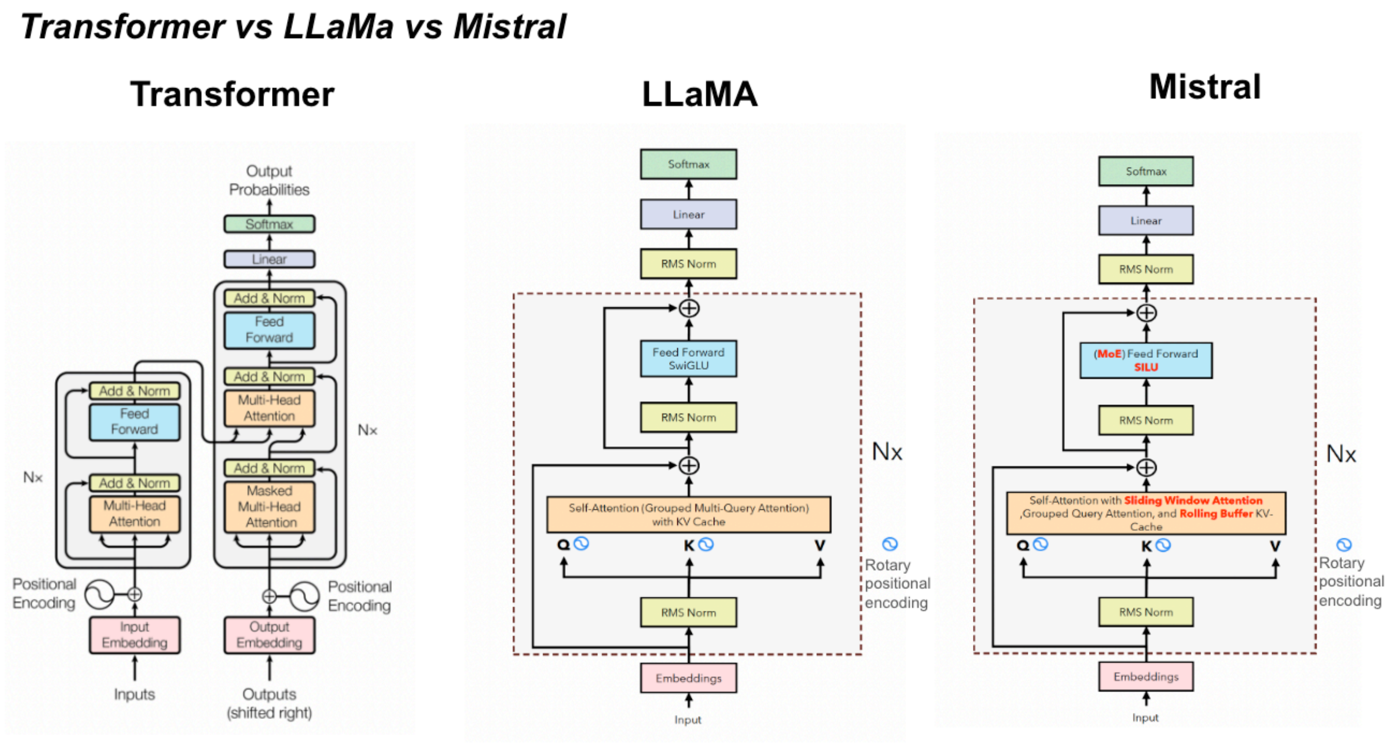


Image source: Umar Jamil

Going deeper into the architecture

- Many other interesting design choices we won't cover, especially positional embeddings, masking, KV caching, flash attention
- Some further reading:
    - Attention is all you need
    - Understanding and Coding the Self-Attention Mechanism of Large Language Models From Scratch
    - The Illustrated Transformer
    - Explained: Multi-head Attention

Going even deeper into the training setup

Maximum likelihood (pre)training is just the beginning...

- Alignment, supervised fine-tuning, preference optimization, RLHF, tool use

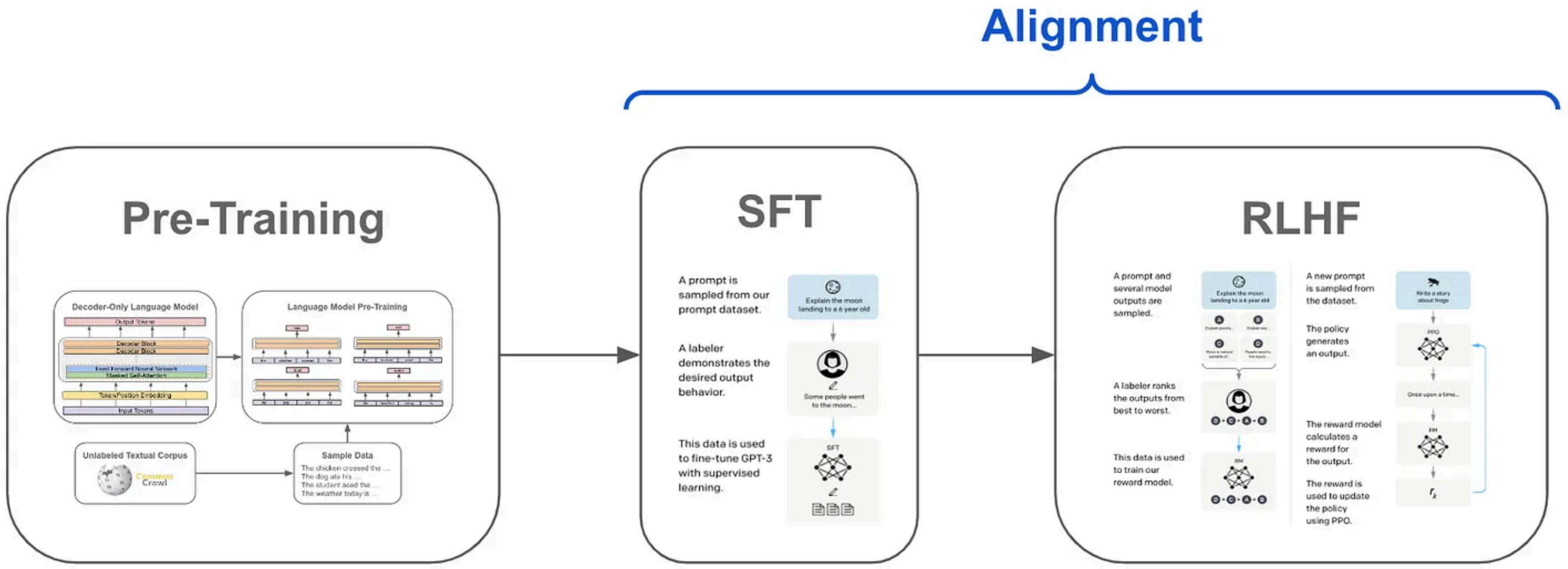

Alignment
Pre-Training
Decoder-Only Language Model
Output Tokens
Decoder Block
Decoder Block
Feed Forward Neural Network
Masked Self-Attention
Token/Position Embedding
Input Tokens
Language Model Pre-Training
Unlabeled Textual Corpus
Common Crawl
Sample Data
The chicken crossed the ...
The dog ate his ...
The student aced the ...
The weather today is ...
SFT
A prompt is sampled from our prompt dataset.
Explain the moon landing to a 6 year old
A labeler demonstrates the desired output behavior.
Some people went to the moon...
This data is used to fine-tune GPT-3 with supervised learning.
SFT
RLHF
A prompt and several model outputs are sampled.
Explain the moon landing to a 6 year old
A labeler ranks the outputs from best to worst.
D > C > A = B
This data is used to train our reward model.
RM
D > C > A = B
A new prompt is sampled from the dataset.
Write a story about frogs
The policy generates an output.
PPO
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Image source: InstructGPT (and here)

Part 2: Running some code!

Loading a model and tokenizer

- HuggingFace hosts many models, tokenizers, datasets, and benchmarks and provides Python/PyTorch libraries for downloading and using them
- Let's load a "small" (with 1B parameters) Llama 3.2 model. (It runs on my laptop)

```
from transformers import AutoTokenizer, AutoModelForCausalLM

model_id = "meta-llama/Llama-3.2-1B-Instruct"
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id, torch_dtype=torch.float16, low_cpu_mem_usage=True, device_map="auto")
```

Let's start with the tokenizer. What does it look like?

```
tokenizer
```

```
PreTrainedTokenizerFast(name_or_path='meta-llama/Llama-3.2-1B-Instruct', vocab_size=128000, model_max_length=13107
2, is_fast=True, padding_side='right', truncation_side='right', special_tokens={'bos_token': '<|begin_of_text|>',
'eos_token': '<|eot_id|>'}, clean_up_tokenization_spaces=True),  added_tokens_decoder={
	128000: AddedToken("<|begin_of_text|>", rstrip=False, lstrip=False, single_word=False, normalized=False, sp
ecial=True),
	128001: AddedToken("<|end_of_text|>", rstrip=False, lstrip=False, single_word=False, normalized=False, spec
ial=True),
	128002: AddedToken("<|reserved_special_token_0|>", rstrip=False, lstrip=False, single_word=False, normalize
d=False, special=True),
	128003: AddedToken("<|reserved_special_token_1|>", rstrip=False, lstrip=False, single_word=False, normalize
d=False, special=True),
	128004: AddedToken("<|finetune_right_pad_id|>", rstrip=False, lstrip=False, single_word=False, normalized=F
alse, special=True),
	128005: AddedToken("<|reserved_special_token_2|>", rstrip=False, lstrip=False, single_word=False, normalize
d=False, special=True),
	128006: AddedToken("<|start_header_id|>", rstrip=False, lstrip=False, single_word=False, normalized=False,
special=True),
	128007: AddedToken("<|end_header_id|>", rstrip=False, lstrip=False, single_word=False, normalized=False, sp
ecial=True),
	128008: AddedToken("<|eom_id|>", rstrip=False, lstrip=False, single_word=False, normalized=False, special=T
rue),
	128009: AddedToken("<|eot_id|>", rstrip=False, lstrip=False, single_word=False, normalized=False, special=T
rue),
	128010: AddedToken("<|python_tag|>", rstrip=False, lstrip=False, single_word=False, normalized=False, speci
al=True),
	128011: AddedToken("<|reserved_special_token_3|>", rstrip=False, lstrip=False, single_word=False, normalize
d=False, special=True),
	128012: AddedToken("<|reserved_special_token_4|>", rstrip=False, lstrip=False, single_word=False, normalize
d=False, special=True),
	128013: AddedToken("<|reserved_special_token_5|>", rstrip=False, lstrip=False, single_word=False, normalize
```

Let's check the vocabulary (Ġ is special and indicates a space before the word)

```
tokenizer.vocab_size
```

```
128000
```

```
tokenizer.vocab
```

```
{'bilt': 70824,
 'ĠãĤŃ': 109949,
 'ucus': 38601,
 '_depth': 19601,
 'ç·Ĵ': 114262,
 'ĠSimilarly': 35339,
 'Ġtess': 80930,
 'Ġspecials': 60874,
 'ĠOT': 8775,
 '=j': 46712,
 'ylation': 79933,
 'ĠNose': 93223,
 'ĠRolls': 70710,
 '.Power': 55186,
 '.Wh': 18951,
 'åĪ©çĶ¨': 107740,
 'ĠATT': 42385,
 'Ġraj': 92528,
 'OLLOW': 31289,
 'ĠBeer': 34484,
 '--------------------------------': 1434,
 'ĠØ§ÙĦØªÙĪ': 124487,
 'Almost': 39782,
 'è§Ĵèī²': 125499,
```

The vocabulary maps subwords to integers (here, out of 128k possibilities)

```
tokenizer.vocab['hi'] # happens to be a token here, although is not guaranteed
```

```
6151
```

Encoding is the process of obtaining the sequence of tokens (llama-tokenizer.js is great for visualizing this)

```
encoded_string = tokenizer.encode('tokenization example string', add_special_tokens=False)
encoded_string
```

```
[5963, 2065, 3187, 925]
```

Decoding is the process of obtaining the string from tokens

```
print(tokenizer.convert_ids_to_tokens(encoded_string))
print(tokenizer.decode(encoded_string))
```

```
['token', 'ization', 'Ġexample', 'Ġstring']
tokenization example string
```

Next, let's look at the model. It mostly has components we've seen before:

```
model
```

```
LlamaForCausalLM(
  (model): LlamaModel(
    (embed_tokens): Embedding(128256, 2048)
    (layers): ModuleList(
      (0-15): 16 x LlamaDecoderLayer(
        (self_attn): LlamaSdpaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear(in_features=2048, out_features=512, bias=False)
          (v_proj): Linear(in_features=2048, out_features=512, bias=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (rotary_emb): LlamaRotaryEmbedding()
        )
        (mlp): LlamaMLP(
          (gate_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (up_proj): Linear(in_features=2048, out_features=8192, bias=False)
          (down_proj): Linear(in_features=8192, out_features=2048, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      )
    )
    (norm): LlamaRMSNorm((2048,), eps=1e-05)
    (rotary_emb): LlamaRotaryEmbedding()
  )
  (lm_head): Linear(in_features=2048, out_features=128256, bias=False)
)
```

Querying for generations

- Given the tokenizer and model, and input token sequence $x_{1:n} = [x_1, \ldots, x_n]$, we can ask the model to predict (generate) next tokens $P(x_{n+j} | x_{1:n+j-1})$.
- The model can sample from many possible generations

  (often controlled by `temperature`, as well as top-p and top-k parameters)

```python
prompt_str = 'Once upon a time'
prompt_tokens = tokenizer.encode(prompt_str, add_special_tokens=True, return_tensors='pt').to('mps')

num_samples = 10
outputs = []
for _ in range(num_samples):
    model_output_tokens = model.generate(
        prompt_tokens, do_sample=True, temperature=1.0, max_new_tokens=10,
        pad_token_id=tokenizer.eos_token_id,
        attention_mask = torch.ones_like(prompt_tokens),
    ).squeeze(0)
    model_output_str = tokenizer.decode(model_output_tokens.tolist())
    outputs.append(model_output_str)

for output in outputs:
    print(output)
```

```
<|begin_of_text|>Once upon a time, there was a small town in Africa where the
<|begin_of_text|>Once upon a time, there was a young man named Max. Max
<|begin_of_text|>Once upon a time, in the land of Aethoria, there
<|begin_of_text|>Once upon a time, a beautiful and mysterious woman named Lena wandered into
<|begin_of_text|>Once upon a time, in a small village nestled in the rolling hills
<|begin_of_text|>Once upon a time, in a bustling city, a young and ambitious
<|begin_of_text|>Once upon a time, there lived a little girl named Lily who had
<|begin_of_text|>Once upon a time, there was a boy named Max who lived in
<|begin_of_text|>Once upon a time, in a small village nestled in the rolling hills
<|begin_of_text|>Once upon a time, there lived a young girl named Sophia. Sophia
```

Generations for answering questions

With the ability to predict next tokens, we can query the model to answer questions. This is an example from the standard MMLU benchmark along with a basic prompt style:

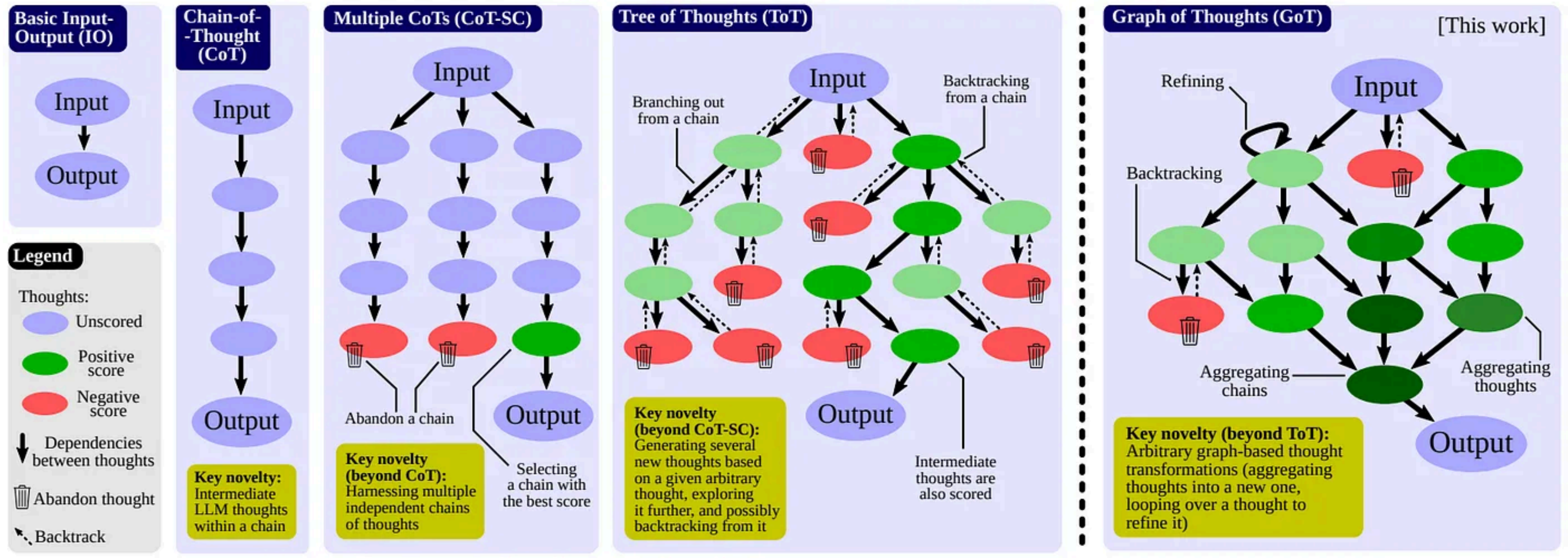
```
prompt_str = """Answer the following multiple choice question by giving the most appropriate response. Answer should be one amon

Question: The famous statement “An unexamined life is not worth living” is attributed to _____.

A: Aristotle
B: John Locke
C: Socrates
D: Plato

Answer:"""
```

```python
prompt_tokens = tokenizer.encode(prompt_str, add_special_tokens=True, return_tensors='pt').to('mps')

model_output_tokens = model.generate(
    prompt_tokens, do_sample=False, temperature=None, max_new_tokens=1,
    pad_token_id=tokenizer.eos_token_id, attention_mask=torch.ones_like(prompt_tokens)
).squeeze(0)
model_output_str = tokenizer.decode(model_output_tokens.tolist())
print(model_output_str)
```

```
<|begin_of_text|>Answer the following multiple choice question by giving the most appropriate response. Answer shou
ld be one among [A, B, C, D].

Question: The famous statement “An unexamined life is not worth living” is attributed to _____.

A: Aristotle
B: John Locke
C: Socrates
D: Plato

Answer: C
```

Extracting more information

- The model's generation was correct, but doesn't tell us much
- **Chain-of-thought** prompting is a way of extracting more information, as done in Large Language Models are Zero-Shot Reasoners.
- Many variations on this:


Basic Input-Output (IO)
Input
Output
Legend
Thoughts:
Unscored
Positive score
Negative score
Dependencies between thoughts
Abandon thought
Backtrack
Chain-of--Thought (CoT)
Key novelty: Intermediate LLM thoughts within a chain
Multiple CoTs (CoT-SC)
Abandon a chain
Selecting a chain with the best score
Key novelty (beyond CoT): Harnessing multiple independent chains of thoughts
Tree of Thoughts (ToT)
Branching out from a chain
Backtracking from a chain
Intermediate thoughts are also scored
Key novelty (beyond CoT-SC): Generating several new thoughts based on a given arbitrary thought, exploring it further, and possibly backtracking from it
Graph of Thoughts (GoT)
[This work]
Refining
Backtracking
Aggregating chains
Aggregating thoughts
Key novelty (beyond ToT): Arbitrary graph-based thought transformations (aggregating thoughts into a new one, looping over a thought to refine it)

Source: Graph of Thoughts

```python
prompt_str = """Answer the following multiple choice question by giving the most appropriate response. Answer should be one amon

Question: The famous statement “An unexamined life is not worth living” is attributed to _____.

A: Aristotle
B: John Locke
C: Socrates
D: Plato

Answer: Let's think step-by-step. """
```

```python
prompt_tokens = tokenizer.encode(prompt_str, add_special_tokens=True, return_tensors='pt').to('mps')

model_output_tokens = model.generate(
    prompt_tokens, do_sample=False, temperature=None, max_new_tokens=100,
    pad_token_id=tokenizer.eos_token_id, attention_mask=torch.ones_like(prompt_tokens)
).squeeze(0)
model_output_str = tokenizer.decode(model_output_tokens.tolist())
print(model_output_str)
```

```
<|begin_of_text|>Answer the following multiple choice question by giving the most appropriate response. Answer shou
ld be one among [A, B, C, D].

Question: The famous statement “An unexamined life is not worth living” is attributed to _____.

A: Aristotle
B: John Locke
C: Socrates
D: Plato

Answer: Let's think step-by-step.  The quote is attributed to Socrates.  Socrates was a Greek philosopher who lived
in ancient Athens.  He is known for his method of questioning, which is now called the Socratic method.  Socrates b
elieved that the unexamined life is not worth living, and this quote reflects his belief that one must examine thei
r own life and values in order to live a meaningful and fulfilling life.  Therefore, the correct answer is C.  The
other options are incorrect because Aristotle was a philosopher
```

From next-token predictions to a chatbot

- Now that we can generate continuations of sequences, what if we want to chat with the LLM as an assistant or chatbot?
- We can't just query it as we were doing before ❌

```
prompt_str = 'What food do you recommend me?'
prompt_tokens = tokenizer.encode(prompt_str, add_special_tokens=True, return_tensors='pt').to('mps')
model_output_tokens = model.generate(
    prompt_tokens, do_sample=False, temperature=None, max_new_tokens=10,
    pad_token_id=tokenizer.eos_token_id, attention_mask=torch.ones_like(prompt_tokens)
).squeeze(0)
model_output_str = tokenizer.decode(model_output_tokens.tolist())
model_output_str
```

```
"<|begin_of_text|>What food do you recommend me? I'm looking for something that's easy to make"
```

We need to use a special **chat prompt template** that the model has been trained with on other chat data. These usually separate the text into `system`, `user`, and `assistant` roles and formats them back into a standardized sequence of tokens:

```
prompt = [
  {"role": "system", "content": "You are a helpful assistant."},
  {"role": "user", "content": "What food do you recommend me?"},
]
inputs = tokenizer.apply_chat_template(prompt, tokenize=True, add_generation_prompt=True, return_tensors="pt", return_dict=True,
print(inputs['input_ids'].tolist()); print()
print(tokenizer.decode(inputs['input_ids'][0]))
```

```
[[128000, 128006, 9125, 128007, 271, 38766, 1303, 33025, 2696, 25, 6790, 220, 2366, 18, 198, 15724, 2696, 25, 220,
868, 4723, 220, 2366, 19, 271, 2675, 527, 264, 11190, 18328, 13, 128009, 128006, 882, 128007, 271, 3923, 3691, 656,
499, 7079, 757, 30, 128009, 128006, 78191, 128007, 271]]

<|begin_of_text|><|start_header_id|>system<|end_header_id|>

Cutting Knowledge Date: December 2023
Today Date: 15 Nov 2024

You are a helpful assistant.<|eot_id|><|start_header_id|>user<|end_header_id|>

What food do you recommend me?<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Let's run the generation on this input and look at the result:

```
outputs = model.generate(
    **inputs, do_sample=False, temperature=None,
    max_new_tokens=60, pad_token_id=tokenizer.eos_token_id)
print(tokenizer.decode(outputs[0]))
```

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

Cutting Knowledge Date: December 2023
Today Date: 15 Nov 2024

You are a helpful assistant.<|eot_id|><|start_header_id|>user<|end_header_id|>

What food do you recommend me?<|eot_id|><|start_header_id|>assistant<|end_header_id|>

I'd be happy to recommend some delicious food options for you. Since I don't know your personal preferences, I'll p
rovide a variety of suggestions.

Here are some popular and tasty food ideas:

**Breakfast Options:**

1. Avocado toast with scrambled eggs and cherry tomatoes
2. Greek
```

Code generation

Lastly, let's switch to some basic code generation with Code Llama. We will load a quantized version for speed:

```
from ctransformers import AutoModelForCausalLM
del tokenizer # now is inside the model
model = AutoModelForCausalLM.from_pretrained(
    "TheBloke/CodeLlama-7B-GGUF", model_file="codellama-7b.Q2_K.gguf", model_type="llama", gpu_layers=0,
    max_new_tokens=50, temperature=0.1)
```

```
Fetching 1 files:   0%|          | 0/1 [00:00<?, ?it/s]

Fetching 1 files:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
model.config
```

```
Config(top_k=40, top_p=0.95, temperature=0.1, repetition_penalty=1.1, last_n_tokens=64, seed=-1, batch_size=8, thre
ads=-1, max_new_tokens=50, stop=None, stream=False, reset=True, context_length=-1, gpu_layers=0, mmap=True, mlock=F
alse)
```

For simple completions, the prompt can be the same as before with the start of a block of code. This is an example from the standard humanevalplus dataset and benchmark. We can look at the tokenization in the same way as before:

```
prompt_str = '''
def fib(n: int):
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
'''
print(model.tokenize(prompt_str))
```

```
[1, 29871, 13, 1753, 18755, 29898, 29876, 29901, 938, 1125, 13, 1678, 9995, 11609, 302, 29899, 386, 383, 747, 265,
21566, 1353, 29889, 13, 1678, 8653, 18755, 29898, 29896, 29900, 29897, 13, 268, 29945, 29945, 13, 1678, 8653, 1875
5, 29898, 29896, 29897, 13, 268, 29896, 13, 1678, 8653, 18755, 29898, 29947, 29897, 13, 268, 29906, 29896, 13, 167
8, 9995, 13]
```

And query the model with:

```
print(model(prompt_str))
```

```
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)

def test_fib():
    assert fib(1) == 1
```

From code completion to infilling

- Completion works great and is powerful, but what if we want to generate suggestions in the middle of a larger file?
- **How do we prompt the model in the middle of the code??**

```
def fib(n: int):
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
    if n < 2:
        ## user's cursor is here ##
    else:
        return fib(n-1) + fib(n-2)
```

Infilling and fill-in-the-middle (FIM) prompting

- Think about the file consisting of `PREFIX`, `MIDDLE`, and `SUFFIX` portions
- **Key idea:** reformulate the prompt so the middle comes at the end, so a file is represented as `<PRE> prefix <SUF>suffix <MID>middle`
    - Uses special tokens `<PRE>` `<MID>` and `<SUF>` to separate them
    - ⚠ **Need to be very careful with the spaces**
- More details in Efficient Training of Language Models to Fill in the Middle

Here's what the basic FIM prompt tokenizes to:

```python
toks = model.tokenize('<PRE> code <SUF>code <MID>')
toks
```

```
[1, 32007, 775, 32008, 401, 32009]
```

```python
[model.detokenize([tok]) for tok in toks]
```

```
['', ' <PRE>', ' code', ' <SUF>', 'code', ' <MID>']
```

Now we can put the Fibonacci prompt into this format and ask for the middle:

```
prompt_str = '''<PRE> def fib(n: int):
    """Return n-th Fibonacci number.
    >>> fib(10)
    55
    >>> fib(1)
    1
    >>> fib(8)
    21
    """
    if n < 2:
         <SUF>
    else:
        return fib(n-1) + fib(n-2) <MID>'''
print(model(prompt_str))
```

```
return n <EOT>
```

Summary

1. Tour through my favorite introductory parts from them
2. Some code examples to show how to apply and use
   a) **basic tokenization** and **autoregressive generation**,
   b) **chat templates**, and
   c) **code completion** (fill-in-the-middle)