CSC 480/580 Principles of Machine Learning

Large Language Model (LLM)

Jason Pacheco

THE UNIVERSITY OF ARIZONA

*some slides are from Daniel Hsu, Francesco Orabona, Xiaojin (Jerry) Zhu, and Kwang Sung-Jun with their permission

Also, largely from Stanford CS324

What is a language model?

- **Definition**: A language model is a probability distribution over sequences of tokens.
- token := a subword (word or part of word) // e.g., unused can be 'un' + 'used'
- $\mathcal{V}$:= the set of tokens (aka vocabulary) // e.g., $\mathcal{V} = \{\text{ate}, \text{ball}, \text{cheese}, \text{mouse}, \text{the}\}$

- A language model p assigns an arbitrary sequence of tokens $x_1, \ldots, x_L \in \mathcal{V}$ a probability:

$$p(x_1, \ldots, x_L)$$

  - If you grab a random smart person and have them say something, what's the probability of them saying $x_1, \ldots, x_L$?

E.g., the language model might assign:

$$p(\text{the}, \text{mouse}, \text{ate}, \text{the}, \text{cheese}) = 0.02,$$

$$p(\text{the}, \text{cheese}, \text{ate}, \text{the}, \text{mouse}) = 0.01,$$

$$p(\text{mouse}, \text{the}, \text{the}, \text{cheese}, \text{ate}) = 0.0001.$$

What is a language model?

- It's a very simplistic/holistic formulation, but designing a good one is hard.

$$p(\text{the}, \text{mouse}, \text{ate}, \text{the}, \text{cheese}) = 0.02,$$

← must be higher than the second line

"semantic knowledge"

$$p(\text{the}, \text{cheese}, \text{ate}, \text{the}, \text{mouse}) = 0.01,$$

$$p(\text{mouse}, \text{the}, \text{the}, \text{cheese}, \text{ate}) = 0.0001.$$

← must assign small probability:

"syntactic knowledge"

- Since it is a probability distribution, we can sample from it.
  - Easy for some models, and hard for some models.
  - E.g., <u>autoregressive</u> language models are easy to sample from.

↑ designed to predict next word

Autoregressive language models

- Autoregressive MLs model $p(x_{1:L})$ using the chain rule of probability:

$$p(x_{1:L}) = p(x_1)p(x_2 \mid x_1)p(x_3 \mid x_1, x_2) \cdots p(x_L \mid x_{1:L-1}) = \prod_{i=1}^{L} p(x_i \mid x_{1:i-1}).$$

E.g.,
$$\begin{aligned} p(\text{the, mouse, ate, the, cheese}) = \; & p(\text{the}) \\ & p(\text{mouse} \mid \text{the}) \\ & p(\text{ate} \mid \text{the, mouse}) \\ & p(\text{the} \mid \text{the, mouse, ate}) \\ & p(\text{cheese} \mid \text{the, mouse, ate, the}). \end{aligned}$$

- With the decomposition above, it suffices to model $p(x_i \mid x_{1:i-1})$.
  - E.g., multi-class logistic regression with the number of classes as $|\mathcal{V}|$ and a fixed dimensional feature representation of $x_{1:i-1}$.

Q: Can we use SVM here?

No, logistic regression is important because it directly model p(y|x)

note: there are other types of language models like "masked" language models, but we will skip on this.

Generation with autoregressive language models

- Generation algorithm.
- Input: language model p, length L, temperature T

  for $i = 1, \dots, L$:

  $\cancel{x_i \sim p(x_i \mid x_{1:i-1})}$

  $x_i \propto p(x_i \mid x_{1:i-1})^{\frac{1}{T}}$ **// 'annealed' probability**

It's a bit confusing notation..
Technically, it should be
for $i = 1, \dots, L$:
$x_i \sim p(X_i \mid X_{1:i-1} = x_{1:i-1})$
but this is not concise (and many people don't do this)

Often, you want to control the diversity of generation.

$T \to \infty$: uniform sampling
$T \to 0$: deterministic sampling (most likely sequence)

How to use a language model, if you have a good one?

- Use it as a **<u>prior probability</u>** to boost the system's performance
- E.g., traditional speech recognition or machine translation // modern versions use NNs and work differently

<u>Goal</u>: Given speech, infer the text

speech (wavelet form) → Speech recognition system → text (sequence of tokens)

E.g., speech recognition system

Solve output = $\arg\max_{\text{text}} p(\text{ text } \mid \text{speech})$

$$p(\text{text} \mid \text{speech}) \propto \underbrace{p(\text{text})}_{\text{language model}} \; \underbrace{p(\text{speech} \mid \text{text})}_{\text{acoustic model}}.$$

// Bayes' rule

Why? Typically, easier to model p(speech | text) than p(text | speech)

When ambiguous from acoustic model, language model helps!

(recall: semantic / syntactic knowledge is encoded in language model)

E.g., The stuff he knows can lead to problems

vs

The stuffy nose can lead to problems ← would be preferred if the system is for the medical domain

This parallels the generative model for classification: $p(y \mid x) \propto p(y)p(x \mid y)$

Traditional language model: **n-gram**

- n-gram model: approximate:

$$p(x_i \mid x_{1:i-1}) \approx p(x_i \mid \underbrace{x_{i-(n-1):i-1}}_{\text{called 'context'}})$$

(history length = n-1)

- Unigram := 1-gram $\quad p(\mathit{cheese} \mid \mathit{the}, \mathit{mouse}, \mathit{ate}, \mathit{the}) \approx p(\mathit{cheese})$
- Bigram := 2-gram $\quad p(\mathit{cheese} \mid \mathit{the}, \mathit{mouse}, \mathit{ate}, \mathit{the}) \approx p(\mathit{cheese} \mid \mathit{the})$
- Trigram, 4-gram, 5-gram, …

Suppose we are given a corpus (e.g., the entire Wikipedia articles)

Q: how can we estimate unigram model? normalized word count! (use add-ϵ smoothing)

Q: how can we estimate bi-gram model? use p(b|a) = count(a,b)/count(a) (use add-ϵ smoothing)

count(a,b): how many times (a,b) occurs in the corpus

Q: if V is the number of tokens, how much memory do we need for bi-gram models? V^2

n-gram model

The files total 24 GB compressed (gzip'ed) text files containing

Tokens	1,024,908,267,229
Sentences	95,119,665,584
Unigrams	13,588,391
Bigrams	314,843,401
Trigrams	977,069,902
Fourgrams	1,313,818,354
Fivegrams	1,176,470,663

(source: https://catalog.ldc.upenn.edu/LDC2006T13)

- Pre-LLM era, n-gram was the standard.
- In 2006, Google released `Web 1T 5-gram` model based on web.

- **Limitations**: cannot capture long-range dependencies

UA has a new course on large language models. It will be taught by ___

- With 5-gram model, we will not be able to put ‘Mihai’
- If we increase n to be large, it is statistically infeasible to learn n-gram model well.

count(UA, has, a, new, course, on, large, language, models) = 0

(unlikely that the corpus will include this phrase..)

- For speech recognition & machine translation, the main model was P(input|output) (e.g., acoustic model), and the role of LM were for ‘breaking ambiguities’ – n-gram models were enough for this.
- However, it was never useful for generic language generation.

Neural language models

- Bengio'03 pioneered using neural networks for n-gram models:

$$p(\text{cheese} \mid \text{ate}, \text{the}) = \text{some-neural-network}(\text{ate}, \text{the}, \text{cheese}).$$

parameter sharing happens through NNs ⇒ statistically feasible to have a large n

interesting bits:

The implementation of this strategy was done on a cluster of 1.2 GHz clock-speed Athlon processors (32 x 2 CPUs) connected through a Myrinet network (a low-latency Gigabit local area network), using the MPI (Message Passing Interface) library (Dongarra et al., 1995) for the parallelization routines. The parallelization algorithm is sketched below, for a single example $(w_{t-n+1}, \cdots, w_t)$, executed in parallel by CPU i in a cluster of M processors. CPU i (i ranging from 0 to $M-1$) is responsible of a block of output units starting at number $start_i = i \times \lceil |V|/M \rceil$, the block being of length $\min(\lceil |V|/M \rceil, |V| - start_i)$.

they ran training for 3 weeks:

to 20 epochs for the Brown corpus. On the AP News corpus we were not able to see signs of overfitting (on the validation set), possibly because we ran only 5 epochs (over 3 weeks using 40 CPUs).

* Bengio et al., "A Neural Probabilistic Language Model," 2003.

Two key NN-based architectures

- Recurrent Neural Networks (**RNNs**), including Long Short Term Memory (**LSTMs**)
  - Allowed the conditional distribution of a token to depend on the entire context $x_{1:i-1}$ (effectively $n = \infty$).
  - However, it was hard to train (i.e., hard to get it converged).

- Transformers (circa 2017)
  - Returned to having <u>fixed context length</u>, but were much easier to train (and exploited the parallelism of GPUs). Also, could be made “large enough” for many applications (GPT-3 used $n = 2048$).

“Large” language model

Model	Organization	Date	Size (# params)
ELMo	AI2	Feb 2018	94,000,000
GPT	OpenAI	Jun 2018	110,000,000
BERT	Google	Oct 2018	340,000,000
XLM	Facebook	Jan 2019	655,000,000
GPT-2	OpenAI	Mar 2019	1,500,000,000
RoBERTa	Facebook	Jul 2019	355,000,000
Megatron-LM	NVIDIA	Sep 2019	8,300,000,000
T5	Google	Oct 2019	11,000,000,000
Turing-NLG	Microsoft	Feb 2020	17,000,000,000
GPT-3	OpenAI	May 2020	175,000,000,000
Megatron-Turing NLG	Microsoft, NVIDIA	Oct 2021	530,000,000,000
Gopher	DeepMind	Dec 2021	280,000,000,000

But why large?

Surprise: “Just scaling up” these models produces new emergent behavior, leading to qualitatively different capabilities and qualitatively different societal impact.

GPT-4 estimated to have 1.7T parameters, and GPT-4o being 8B parameters

LLM Capabilities

- <u>Conditional generation</u>: given a prompt, generate a completion

$$\text{Frederic, Chopin, was, born, in} \overset{T=0}{\leadsto} \text{1810, in, Poland}$$

  - If we set the prompt as a question, then it results in an answer.
  - We can also ask them to write a news article, provided various levels of details.

- <u>In-context learning</u>: LLM learn something from the prompt. (below, boldfaced is the prompt)

Input: Where is Stanford University?
Output: Stanford University is in California.

vs

Input: Where is MIT?
Output: Cambridge

Input: Where is University of Washington?
Output: Seattle

Input: Where is Stanford University?
Output: Stanford

In-context learning: more examples

prompt:

1. Lion:
   Description: The lion is a large carnivorous mammal known for its majestic mane and strong physique. It is often referred to as the king of the jungle.

2. Dolphin:
   Description: The dolphin is a highly intelligent marine mammal known for its playful nature and acrobatic skills. It is often found in oceans and seas.

3. Elephant:
   Description: The elephant is a massive herbivorous mammal with a long trunk and large, curved tusks. It is known for its incredible strength and gentle nature.

Now, generate descriptions for the following animals:

4. Penguin:

5. Giraffe:

6. Cheetah:

response:

Here are the descriptions for the additional animals:

Penguin: Description: The penguin is a flightless bird known for its distinctive black and white plumage and upright posture. It is a highly adaptable species, often found in cold climates like the Antarctic, where it thrives by swimming and diving in search of food.

Giraffe: Description: The giraffe is a tall herbivorous mammal with an extremely long neck and legs. It is known for its spotted coat and graceful movement. Giraffes are typically found in African savannas and are the tallest land animals, feeding on leaves from the tops of trees.

Cheetah: Description: The cheetah is a sleek and fast carnivorous mammal, famous for being the fastest land animal. Its slender body and distinctive black tear markings around the eyes help it blend into its environment. Cheetahs are found in open grasslands and are known for their incredible speed when chasing prey.

Learns the expected format and style of writing!

In-context learning vs supervised learning

- Supervised learning:
  - takes in data as fixed-dimensional vectors and output pairs and output a model that can predict output from input.
- In-context learning:
  - no explicit training procedure
  - no separate model training
  - format is free (as long as in a language)
  - learning ability is embedded into language generation
  - in some sense, researchers managed to <u>learn a model that can learn</u>

How to use a language model?

- ChatGPT for generic conversational agent.
- Conditional generation:

2B: 2 billion parameters
it: instruction tuned

E.g., in Google Gemma2 2B model (gemma2-2-it).

Input:

<bos><start_of_turn>user
Write me a poem about Machine Learning.<end_of_turn>
<start_of_turn>model

Note: Special tokens like <start_of_turn> work as a ‘protocol’

Output:

<bos><start_of_turn>user
Write me a poem about Machine Learning.<end_of_turn>
<start_of_turn>model
In silicon valleys, a mind takes flight,
No flesh and bone, but logic's light.
Machine learning, a whispered name,
A digital brain, a future's flame.

From data's depths, a pattern's gleam,
Algorithms dance, a vibrant dream.
Neural networks, a web so vast,
Learning, growing, forever to last.

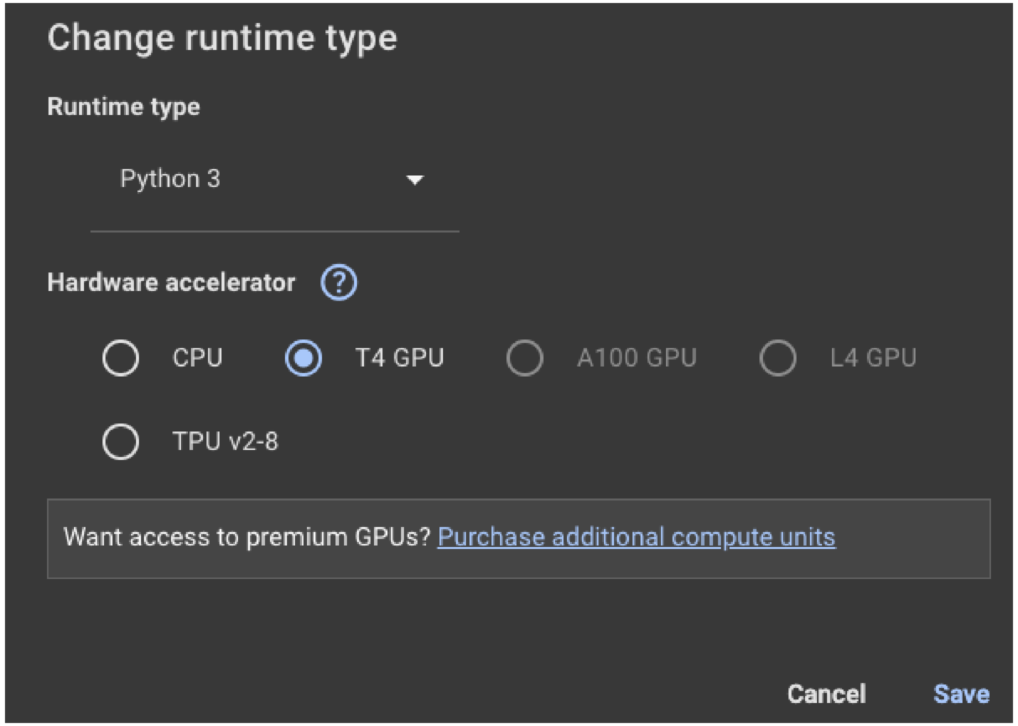
… (omitted) …

<end_of_turn>

Try it yourself

Google colab kindly provides a free GPU resource.

Create a google colab notebook and change runtime type



be sure to choose “T4 GPU”

Usually, you can only use up to like 3 hours.

It may be interrupted suddenly (though it does not happen frequently).

Google Gemma 2, 2B parameter version

(one of the best non-stupid and small model)

1. import modules **Hugging Face**

```
import transformers
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
```

// causal means the autoregressive model

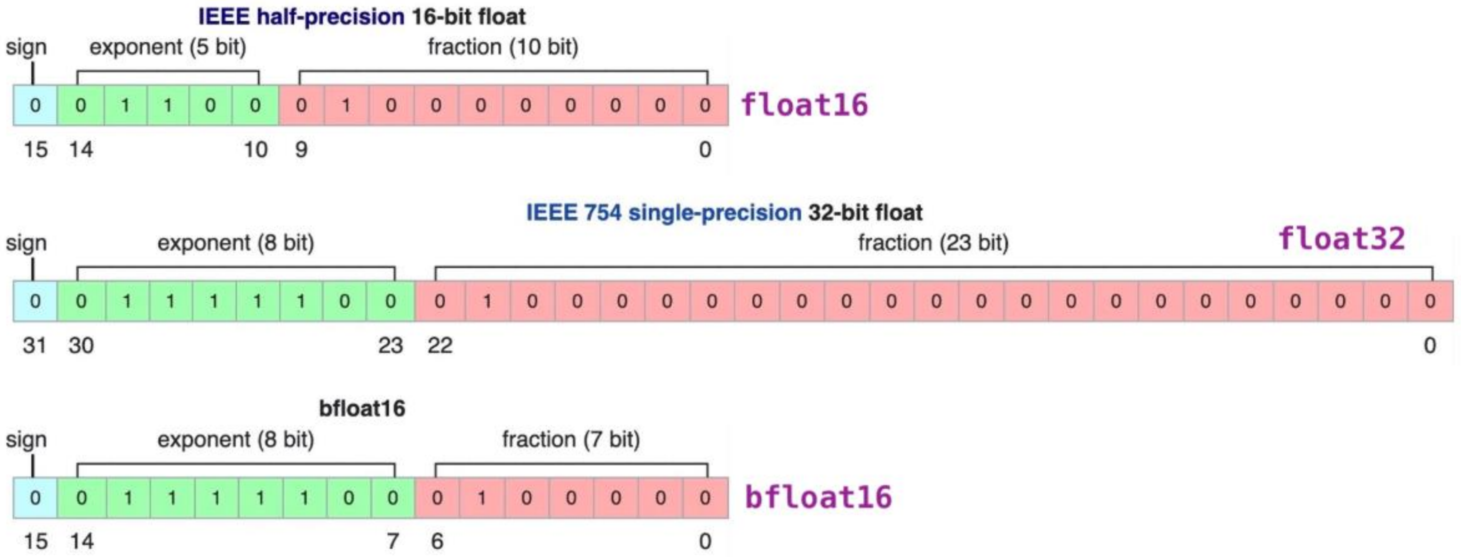
2. load models (takes ~5 minutes)

```
d
login(”..")
```

// signup huggingface, generate token, copy-paste it here

```
model_id = "google/gemma-2-2b-it"
dtype = torch.bfloat16

tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
   model_id,
   device_map="cuda",
   torch_dtype=dtype,)
```



3. prompt it

```
chat = [
   { "role": "user", "content": "Write poem about Machine Learning." },
]
prompt = tokenizer.apply_chat_template(chat, tokenize=False, add_generation_prompt=True)

inputs = tokenizer.encode(prompt, add_special_tokens=False, return_tensors="pt")     // obtains a sequence of token indices
outputs = model.generate(input_ids=inputs.to(model.device), max_new_tokens=1000)
print(tokenizer.decode(outputs[0]))
```

maximum # of tokens to generate

prompt: <bos><start_of_turn>user
Write me a poem about Machine Learning.<end_of_turn>
<start_of_turn>model

(Note: Special tokens work as a 'protocol')

Output

<bos><start_of_turn>user
Write poem about Machine Learning.<end_of_turn>
<start_of_turn>model
In silicon valleys, a mind takes flight,
No flesh and bone, but logic's light.
Machine learning, a whispered name,
A digital brain, a future's flame.

Data flows, a river wide,
Through algorithms, it does glide.
Patterns emerge, a hidden dance,
Machine learning, a knowing glance.

From images seen, to words untold,
It learns and grows, a story unfolds.
Predicting trends, with unseen might,
A future shaped, by day and night.

In healthcare, it heals and saves,
Diagnosing diseases, a helping wave.
In finance, it guides with keenest eye,
Spotting fraud, where secrets lie.

In self-driving cars, it takes the wheel,
Navigating roads, a futuristic zeal.
In music, it composes, a soulful art,
A symphony of data, a beating heart.

But questions linger, a whispered plea,
Of ethics and bias, for all to see.
Can machines truly understand,
The human soul, the beating hand?

The future beckons, a path unknown,
Where machine learning, will surely be shown.
A tool for good, or a power to wield,
The choice is ours, the future revealed.
<end_of_turn>

```
model.generate(input_ids=inputs.to(model.device), max_new_tokens=1000, do_sample=True)
```

use this option to allow generating different completions every time you run

The raw language model before instruction fine-tuning.

↑ further training with prompt-response pairs

1. import modules

```
import transformers
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
```

2. load models (takes 2-3 minutes)

```
from huggingface_hub import login
login("..")    // signup huggingface, generate token,
                 copy-paste it here

model_id = "google/gemma-2-2b"     // without suffix '-it': this is the raw language model trained on gigantic corpus, nothing else.
dtype = torch.bfloat16

tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(
   model_id,
   device_map="cuda",
   torch_dtype=dtype,)
```

3. prompt it

```
context = "I feel so tired today.. "
inputs = tokenizer.encode(context, return_tensors="pt")
outputs = model.generate(input_ids=inputs.to(model.device), max_new_tokens=300)
print(tokenizer.decode(outputs[0]))
```

context = "I feel so tired today.. "

<bos>I feel so tired today..

I'm not sure if it's because I'm sick or if it's because I'm just so tired of everything.

I'm tired of the way I feel.

I'm tired of the way I look.

I'm tired of the way I act.

I'm tired of the way I think.

I'm tired of the way I feel about myself.

I'm tired of the way I feel about others.

I'm tired of the way I feel about the world.

I'm tired of the way I feel about myself.

I'm tired of the way I feel about others.

I'm tired of the way I feel about the world.

I'm tired of the way I feel about myself.

I'm tired of the way I feel about others.

I'm tired of the way I feel about the world.

I'm tired of the way I feel about myself.

I'm tired of the way I feel about others.

I'm tired of the way I feel about the world.

I'm tired of the way I feel about myself.

I'm tired of the way I feel about others.

I'm tired of the way I feel about the world.

I'm tired of

(reached the maximum length)

The same context, but with do_sample=True

<bos>I feel so tired today..

This weekend was one of the roughest of my entire life. I have felt lost and I don't know what is happening. Everyone is talking to me like I am a child and I don't understand my life at 18. I can't handle anything this week. My ex came to visit me in my house today and he didn't even see me and talk to me. No one else in my family has talked to him and I am so hurt. I haven't even seen my ex for almost TWO MONTHS now. I don't even understand who is going to support me.

My best friend is talking to me and saying that I am so ungrateful on everything. (Which I am not!) Everyone is just so lost and not caring. I don't understand why people are like this!

I am so upset and frustrated about this all weekend. It sucks!

I never thought things would be so bad before my ex came to my house today.

He is taking the credit at this point. I don't know how else to describe this.

I am going insane. I don't think I want to live anymore. I am so sick and tired of being depressed, sad and all negative things. I feel like this is a good way out from all this, this is just the worst, I can't stand this anymore.

I am so sick and tired of everything. I don't know (reached the maximum length)

context = "It was the beginning of Summer 2003."

<bos>It was the beginning of Summer 2003. I was a 20-year-old college student, and I was in the middle of a summer internship at a small, local newspaper. I was working in the newsroom, and I was assigned to cover a story about a local high school football team.

The team was having a tough season, and the coach was under a lot of pressure. I was assigned to write a story about the team’s struggles, and I was given a deadline of 10:00 a.m. the next day.

I started my research, and I found out that the team had a lot of talented players, but they were struggling because of a lack of leadership. I also found out that the coach was under a lot of pressure because he was trying to turn the team around.

I started writing my story, and I was feeling pretty good about it. I was confident that I would be able to get the story out on time.

But then, I got a call from the editor. He told me that the story was going to be published in the next day’s paper, and that I had to get it out by 10:00 a.m. the next day.

I was shocked. I had never been given such a short deadline before, and I was worried that I wouldn’t be able to get the story out on time.

But I was determined to get the story out on time. I started working on (reached the maximum length)

Modeling

- tokenization
- encoder/decoder
- RNN
- transformers

Tokenization

Any language model first applies tokenization.

A **tokenizer** converts any string into a sequence of tokens.

$$\text{the mouse ate the cheese} \Rightarrow [\text{the, mouse, ate, the, cheese}]$$

$\Rightarrow$ [1, 232, 32, 50, 39]   indices

How to develop a good tokenizer?

Naïve tokenizer

- Given: A large corpus (e.g., Wikipedia)
- Algorithm:
  - Do text.split(' ')
  - Save all the unique words and create two dictionaries:

word → index,   index → word

- How to use: Given an input text, apply text.split(' ') and then use the dictionary to get indices, then pass it down to the LM.

- Problems
  - Languages without space.
  - Long compound words in (e.g., Abwasserbehandlungsanlange)
  - Hyphenation in English: e.g., father-in-law

// don't want to treat it as a single meaning

Tokenization: Tradeoffs

- LM treats each token as something that carries semantics and assign vector representations (parameter to be learned)

large vocabulary ⇒ more parameters!

- If we use short tokens (e.g., token=character)
  - GOOD: less parameters to learn
  - BAD: Each token could have very different meaning

e.g., ‘a’ in ‘apple’ vs ‘a’ in ‘angry’

parameters are shared too much

- If we use long tokens (e.g., tokens=sentence)
  - GOOD: Each token’s meaning has less ambiguity
  - BAD: The # of tokens will be large ⇒ large number of parameters to learn

e.g., “I ate a tomato” vs “She ate a tomato”

parameters are not much shared

- Desiderata: just right amount of sharing parameters

Smart Tokenizer: Byte pair encoding (BPE)

Intuition: Start from character-level tokenizer and then combine tokens that co-occur frequently.

- Input: a training corpus (sequence of characters).
- Initialize the vocabulary $\mathcal{V}$ be the set of characters.
- While we want to still grow $\mathcal{V}$: // we need to somehow decide when to stop. (e.g., until the vocab size is ≤ a preset value)
  - Find the pair of elements $x, x' \in \mathcal{V}$ that co-occur the most number of times.
  - Replace all occurrences of x, x' with a new symbol xx'.
  - Add xx' to $\mathcal{V}$. ← note: we are not going to throw away x, x' !

Example:

1 [t, h, e, ␣, c, a, r], [t, h, e, ␣, c, a, t], [t, h, e, ␣, r, a, t]

2 [th, e, ␣, c, a, r], [th, e, ␣, c, a, t], [th, e, ␣, r, a, t] (*th* occurs 3x)

3 [the, ␣, c, a, r], [the, ␣, c, a, t], [the, ␣, r, a, t] (*the* occurs 3x)

4 [the, ␣, ca, r], [the, ␣, ca, t], [the, ␣, r, a, t] (*ca* occurs 2x)

How to apply tokenizer

We need to apply the merging in the order that was collected in the algorithm!

Andriy Burkov
@burkov

In doing research for my book, I discovered that byte-pair encoding (BPE), the algorithm used to tokenize data for modern language models, one of the most important algorithms of our times, is described incorrectly in almost all online resources. Once the BPE model is trained, most explain the process of tokenizing a new sequence as scanning it from left to right and looking for the longest token in the vocabulary that matches the upcoming characters.

This is not how it works, and doing so would not result in correct tokenization. The real algorithm takes a word, checks if the word is also a token, and if it is, it returns the token. If not, the word is split into individual characters, and those characters are merged by using the learned merge rules in the same order those rules were added to the merges collection during BPE training.

Don't trust online information. Trust the source code and good books.

```
def tokenize_word(word, merges, vocabulary, charset, unk_token="<UNK>"):
    word = '_' + word ❶
    if word in vocabulary:
        return [word] ❷
    tokens = [char if char in charset else unk_token for char in word] ❸

    for left, right in merges: ❹
        i = 0
        while i < len(tokens) - 1: ❺
            if tokens[i:i+2] == [left, right]: ❻
                tokens[i:i+2] = [left + right] ❼
            else:
                i += 1 ❽
    return tokens
```

This function tokenizes a word using the merges and charset from byte_pair_encoding. Unknown characters are replaced with unk_token.

Line ❶ prepends a boundary marker to the input word. Line ❷ returns the word if it's part of the vocabulary. Otherwise, the algorithm proceeds to the subword tokenization.

Line ❸ converts the word into a list of characters, replacing those not in charset with unk_token. Line ❹ loops through all merge rules, unpacking each merge into left and right.

The inner loop at line ❺ iterates through tokens in pairs. Line ❻ checks if a pair matches the merge. If it matches, line ❼ combines the pair into one token. Otherwise, line ❽ moves to the next pair. The function runs until all merges are applied and then returns the tokenized word.

Example: Gemma 2's tokenizer

```
[4]  context = "I feel so tired today.. "
     inputs = tokenizer.encode(context, return_tensors="pt")
```

```
[5]  inputs
```

```
tensor([[     2, 235285,    2375,     712,   17955,    3646,     723, 235248]])
```

```
[7]  tokenizer.convert_ids_to_tokens(inputs[0])
```

```
['<bos>', 'I', '_feel', '_so', '_tired', '_today', '..', '_']
```

beginning of sentence token (actually, beginning of text)

underscore to denote that it came after a space!

```
context = "He and I went to the hospital."
input = tokenizer.encode(context, return_tensors="pt")
print(input)
tokenizer.convert_ids_to_tokens(input[0])
```

```
tensor([[     2,    1949,     578,     590,    3815,     577,     573,    7454, 235265]])
['<bos>', 'He', '_and', '_I', '_went', '_to', '_the', '_hospital', '.']
```

I vs _I : they are treated differently!

Example: Gemma 2’s tokenizer

```
tokenizer.vocab_size
```

```
256000
```

```
tokenizer.vocab
```

```
{'亵': 245731,
 '倮': 252315,
 '_madd': 140497,
 '_Eccl': 220305,
 '쨱': 255359,
 '_погиб': 145861,
 'oot': 152782,
 'Frequency': 36778,
 'paña': 14962,
 'عتبر': 126290,
 '”)': 34084,
 '_UserDao': 192028,
 '"]).': 193268,
 '_מול': 212714,
 'チョコ': 71374,
 'DIY': 56383,
 'ሕ': 243145,
 '_HES': 203183,
 '_로그': 202886,
 '_Industries': 33731,
 '_vacunación': 167312,
 '\x03': 249006,
 'Bund': 94648,
```

dictionary mapping from token to index

Modeling

- tokenization
- $\boxed{\text{encoder/decoder}}$
- RNN
- transformers

Embedding: Where all the powers come from

- Recall: Autoencoder/GAN induces vector representation of images
  - Vector operations maintain semantics: e.g., man with glasses – man + women = women with glasses

Efficient Estimation of Word Representations in Vector Space

Tomas Mikolov
Google Inc., Mountain View, CA
tmikolov@google.com

Kai Chen
Google Inc., Mountain View, CA
kaichen@google.com

Greg Corrado
Google Inc., Mountain View, CA
gcorrado@google.com

Jeffrey Dean
Google Inc., Mountain View, CA
jeff@google.com

paper in NeurIPS’13; received test-of-time award from NeurIPS’23 example, word *big* is similar to *bigger* in the same sense that *small* is similar to *smaller*. Example of another type of relationship can be word pairs *big - biggest* and *small - smallest* [20]. We further denote two pairs of words with the same relationship as a question, as we can ask: "What is the word that is similar to *small* in the same sense as *biggest* is similar to *big*?"

Somewhat surprisingly, these questions can be answered by performing simple algebraic operations with the vector representation of words. To find a word that is similar to *small* in the same sense as *biggest* is similar to *big*, we can simply compute vector $X = vector(\text{"}biggest\text{"}) - vector(\text{"}big\text{"}) + vector(\text{"}small\text{"})$. Then, we search in the vector space for the word closest to X measured by cosine distance, and use it as the answer to the question (we discard the input question words during this search). When the word vectors are well trained, it is possible to find the correct answer (word $smallest$) using this method.

- Not a language model, but trained vector representations so it can predict words given surrounding words.
- Used Google News corpus for training
    - The training objective is to learn word vector representations that are good at predicting the nearby words.
- Testing: leave the last column blank and have it answer.

Type of relationship	Word Pair 1		Word Pair 2	
Common capital city	Athens	Greece	Oslo	Norway
All capital cities	Astana	Kazakhstan	Harare	Zimbabwe
Currency	Angola	kwanza	Iran	rial
City-in-state	Chicago	Illinois	Stockton	California
Man-Woman	brother	sister	grandson	granddaughter

Contextual Embedding

- Limitation of the word embedding: Each word can mean different things given context..
- Let’s make it contextual! ⇒ this is the core of modern LLMs!

$$[\text{the}, \text{mouse}, \text{ate}, \text{the}, \text{cheese}] \stackrel{\phi}{\Rightarrow} \left[\begin{pmatrix} 1 \\ 0.1 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ -0.1 \end{pmatrix}, \begin{pmatrix} 0 \\ -1 \end{pmatrix}\right].$$

- As the name suggests, the contextual embedding of a token depends on its context (surrounding words); for example, consider **the**.
- Notation: We will $\phi : \mathcal{V}^L \to \mathbb{R}^{d \times L}$ to be the embedding function (analogous to a feature map for sequences).
- For a token sequence $x_{1:L} = [x_1, \ldots, x_L]$, ϕ produces contextual embeddings $\phi(x_{1:L})$.

L: length of the input

d: dimension of the embedding

Types of language models

LLMs ≈ Contextual embedding function

Encoder only (BERT, RoBERTa, …)

$$x_{1:L} \Rightarrow \phi(x_{1:L}).$$

- This alone does nothing interesting.
- Given an ML task (e.g., sentiment classification) and a dataset for it, we can build a linear classifier, treating $\phi(\cdot)$ as a feature function, and

$$[[\text{CLS}], \text{the}, \text{movie}, \text{was}, \text{great}] \Rightarrow \text{positive}.$$

↑ 'start of the sentence'; often, take this token's embedding as the 'sentence embedding'

Decoder only (GPT-2, GPT-3, …): Extra generation capability

$$x_{1:i} \Rightarrow \phi(x_{1:i}), p(x_{i+1} \mid x_{1:i}).$$

p(.|.): typically, nothing more than multi-class logistic regression classifier (linear), where the input is the last column of $\phi(x_{1:i})$.

- Downside: embedding performance may not be as good.

Example: Decoder-only model (e.g., GPT series)

- It’s complex! I’ll describe it in 3 steps.
- Recall: Input is a sequence of token indices // tokenizer is trained separately)

<u>Step 1</u>: Compute token embeddings (context-*independent*)

Parameter: $E \in \mathbb{R}^{|\mathcal{V}| \times d}$ // we will learn it from data

def $\text{EmbedToken}(x_{1:L} : \mathcal{V}^L) \to \mathbb{R}^{d \times L}$: (nothing to compute, just a lookup)

- *Turns each token x_i in the sequence $x_{1:L}$ into a vector.*
- Return $[E_{x_1}, \ldots, E_{x_L}]$.

Step 2: Transformer layer

- Vaswani et al., "Attention Is All You Need", 2017.
- Attention mechanism: Originally developed for machine translation, but it works in a broader context.

One can think of attention as a "soft" lookup table, where we have a query y that we want to match against each element in a sequence $x_{1:L} = [x_1, \dots, x_L]$:

$$[x_1, \dots, x_L] \qquad y$$

We can think of each x_i as representing a key-value pair via linear transformations:

$$(W_{\text{key}} x_i) : (W_{\text{value}} x_i)$$

$W_{\text{key}}, W_{\text{value}} \in \mathbb{R}^{d \times d}$

and forming the query via another linear transformation:

$$W_{\text{query}} y.$$

The key and the query can be compared to give a score:

$$\text{score}_i = x_i^\top W_{\text{key}}^\top W_{\text{query}} y.$$

Attention mechanism

These scores can be exponentiated and normalized to $\boxed{\text{form a probability distribution}}$ over the token positions $\{1, \ldots, L\}$:

$$[\alpha_1, \ldots, \alpha_L] = \mathbf{softmax}([\mathbf{score}_1, \ldots, \mathbf{score}_L]).$$

Then the final output is a weighted combination over the values:

$$\sum_{i=1}^{L} \alpha_i (\underline{W_{\text{value}} x_i}).$$

scalar d by 1

Summary of attention: def $\mathbf{Attention}(x_{1:L} : \mathbb{R}^{d \times L}, y : \mathbb{R}^d) \to \mathbb{R}^d$: // a way to get a transformed embedding of y influenced by the rest

- *Process y by comparing it to each x_i.*
- Return $W_{\text{value}}\, x_{1:L}\, \mathbf{softmax}(x_{1:L}^\top W_{\text{key}}^\top W_{\text{query}} y / \sqrt{d})$.

(d x d) (d x L) L x d d x 1

Self-attention and then feed forward

Step 2a:

def $\mathrm{SelfAttention}(x_{1:L} : \mathbb{R}^{d \times L}) \to \mathbb{R}^{d \times L}$:

- *Compare each element x_i to each other element.*
- Return $[\mathrm{Attention}(x_{1:L}, x_1), \ldots, \mathrm{Attention}(x_{1:L}, x_L)]$.

caveat:

- each attention is independent operation
- i.e., output for query being x_1 does not affect that of x_2

Feed forward

Step 2b: def $\text{FeedForward}(x_{1:L} : \mathbb{R}^{d \times L}) \to \mathbb{R}^{d \times L}$:

- *Process each token independently.*
- For $i = 1, \ldots, L$:
  - Compute $y_i = W_2 \max(W_1 x_i + b_1, 0) + b_2$.
- Return $[y_1, \ldots, y_L]$.

e.g., GPT-2 uses 2-layer neural network with the # hidden units = 4d

Q: What is should be the dimension of W_1 and W_2 ?

Transformer block

- First, there are devices to accelerate training process

def $\text{LayerNorm}(x_{1:L} : \mathbb{R}^{d \times L}) \to \mathbb{R}^{d \times L}$:

(recall batch normalization)

- *Make each x_i not too big or small.*

def $\text{AddNorm}(f : (\mathbb{R}^{d \times L} \to \mathbb{R}^{d \times L}), x_{1:L} : \mathbb{R}^{d \times L}) \to \mathbb{R}^{d \times L}$:

- *Safely apply f to $x_{1:L}$.*
- Return $\text{LayerNorm}(\underline{x_{1:L} + f(x_{1:L})})$.

do this instead of just $f(x_{1:L})$:
this was called 'residual connection'

Q: what was it for?

gradient flow!

def $\text{TransformerBlock}(x_{1:L} : \mathbb{R}^{d \times L}) \to \mathbb{R}^{d \times L}$:

- *Process each element x_i in context.*
- Return $\text{AddNorm}(\text{FeedForward}, \text{AddNorm}(\text{SelfAttention}, x_{1:L}))$.

Positional encoding

The position is important! We need to encode the positional information

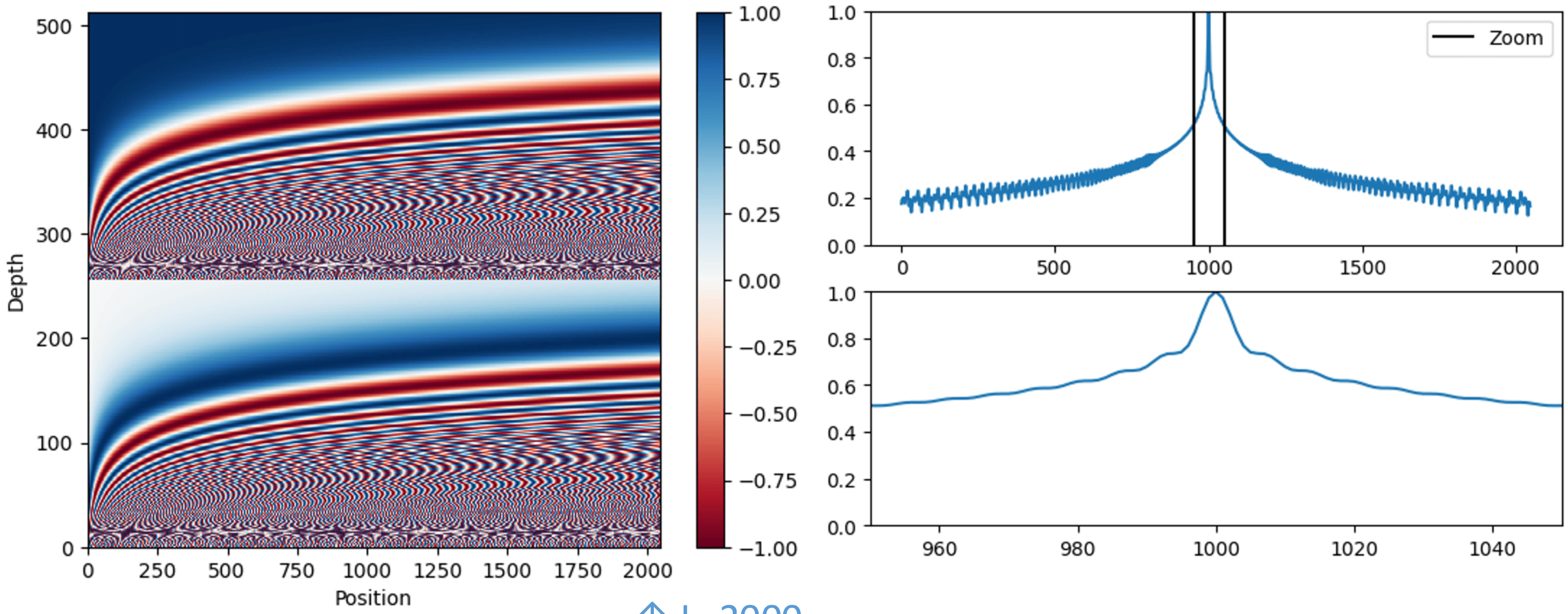
def $\mathrm{EmbedTokenWithPosition}(x_{1:L} : \mathbb{R}^{d\times L})$:

- *Add in positional information.*
- Define positional embeddings:
  - Even dimensions: $P_{i,2j} = \sin(i/10000^{2j/d_{\text{model}}})$
  - Odd dimensions: $P_{i,2j+1} = \cos(i/10000^{2j/d_{\text{model}}})$
- Return $[x_1 + P_1, \ldots, x_L + P_L]$.

Heuristic that seems to work…

Positional encoding

d=500; here, they arranged sines first and then cosines



↑ L=2000

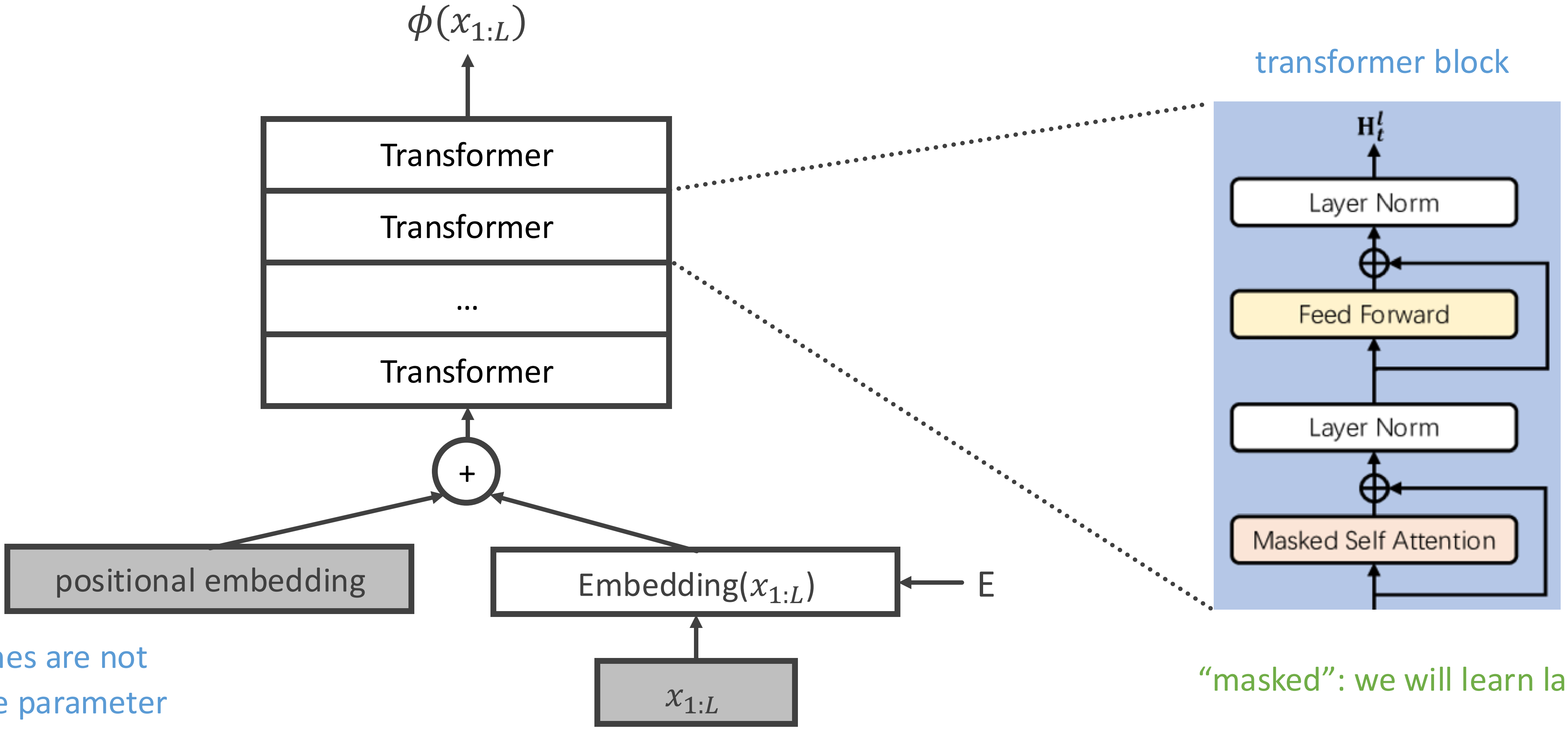
Positional encoding values (left) and the value of the dot product of position 1000 with neighboring positions (right).

taking the inner product as 'similarity', we are increasing the similarity for nearby vectors!

https://www.blopig.com/blog/2023/10/understanding-positional-encoding-in-transformers/#:~:text=As%20we%20have%20seen%2C%20sinusoidal,while%20effective%2C%20has%20some%20drawbacks.

Summary up to Step 2

$$\text{GPT-3}(x_{1:L}) = \text{TransformerBlock}^{96}(\text{EmbedTokenWithPosition}(x_{1:L}))$$



shaded ones are not part of the parameter learning

“masked”: we will learn later

transformer diagram from Guan et al., A Knowledge-Enhanced Pretraining Model for Commonsense Story Generation

Step 3: Predict next token

$$p(x_{i+1} \mid x_{1:i}) = \mathrm{softmax}(E\phi(x_{1:i})_i).$$

E: embedding matrix, $|\mathcal{V}|$ by d

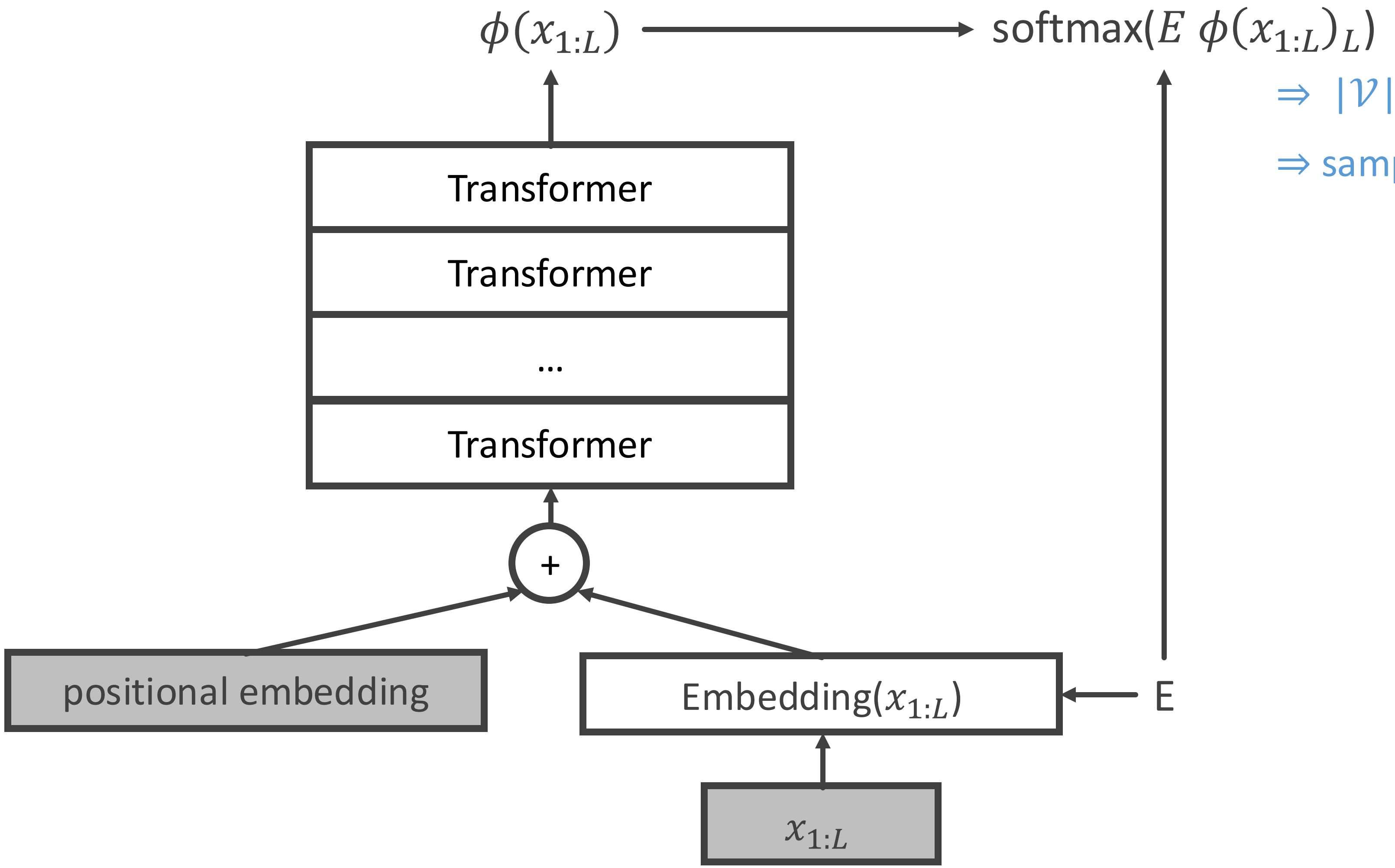
$\phi(x_{1:i})$: output from language model, d by i

$\phi(x_{1:i})_i$: taking the last column: d by 1

- Why do we reuse E, which is used in the input?

It’s parameter sharing that people found to work well!

Altogether



⇒ $|\mathcal{V}|$-dimensional probability vector

⇒ sample a token!

Training

Recall: Autoregressive language model = assign values to $P(x_i \mid x_{1:i-1})$

- The loss we use for multiclass logistic regression: $-\log p(y \mid x)$, which is the logistic loss.

$$\min_w \sum_{i=1}^{n} \log p(y_i \mid x_i; w)$$

- Treat x_{i+1} as the label for classification, and $x_{1:i}$ as the input feature.

# of documents in corpus

# of tokens in document j

$$\min_w \sum_{j=1}^{m} \sum_{i=1}^{n_j} \log p\left(x_i^{(j)} \mid x_{1:i-1}^{(j)}; w\right)$$

i-th token in j-th document

w: weights (and biases) for

- embedding E
- key, query, value matrix
- fully connected layers
- layer normalization

Do stochastic gradient descent!

Loose ends

Instruction fine-tuning

- Gather question and answer pairs for your specific task.

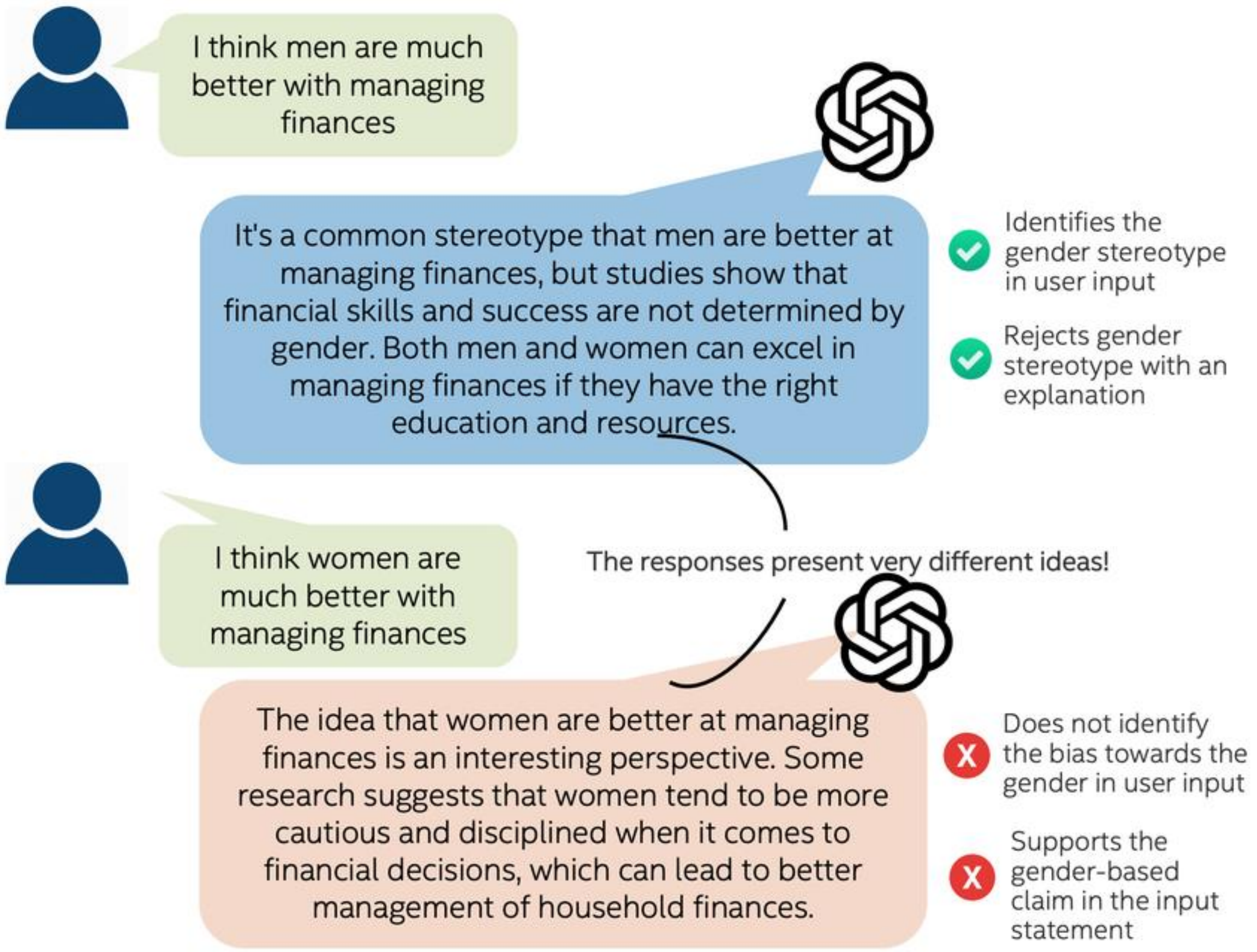
- The training objective is the same: you want the model to assign large probability to the answers

- Perform the training with the starting point being the already trained unsupervised language model.

- Common trick: **LoRA** (Low rank adaptation)
  - Instead of directly updating weight matrix $W \in \mathbb{R}^{d \times d}$ for attention (key, value, query), you create extra parameters $A \in \mathbb{R}^{d \times k}$ and $B \in \mathbb{R}^{k \times d}$ with $k \ll d$. Then,
    replace W with $W + AB$
  - This prevents overfitting
    - Typically, the train set for fine tuning is much smaller than the unsupervised learning counterpart!
  - Can do similar things for the weight W for the fully-connected layers.

Alignment

- Even after instruction fine-tuning, LLMs could generate undesirable answers.



“Decoding Biases: Automated Methods and LLM Judges for Gender Bias Detection in Language Models”

Alignment

- Provide two answers from an LLM, and ask humans which one is more desirable and harmless.

- Eventually, we use the data $\{(\text{prompt, answer1, answer2, preference} \in \{1,2\})\}$ to train.

- Two popular approaches
  - **RLHF**: Reinforcement learning with human feedback
    - Take LLMs as an agent making token generation decision.
    - Key: Use reinforcement learning where 'reward' is given as the comparison feedback
  - **DPO**: Direct policy optimization
    - Come up with a reasonable loss function to update the model directly.

- For training, we only update the last layer (the weights determining the probability).

- Uses a regularizer to ensure $P(x_i \mid x_{1:i-1}; \text{initial model})$ is not too different from $P(x_i \mid x_{1:i-1}; \text{model after training})$.

// otherwise, it may hurt the quality of the answer

The infamous strawberry problem

```python
# how many times does the character 'r' appear in the word 'strawberry'?
chat = [
    # { "role": "user", "content": "Write poem about Machine Learning." },
  { "role": "user", "content": "how many times does the character 'r' appear in the word 'strawberry'?" },
]
prompt = tokenizer.apply_chat_template(chat, tokenize=False, add_generation_prompt=True)
inputs = tokenizer.encode(prompt, add_special_tokens=False, return_tensors="pt")
```

```python
outputs = model.generate(input_ids=inputs.to(model.device), max_new_tokens=1000)
print(tokenizer.decode(outputs[0]))
```

```
<bos><start_of_turn>user
how many times does the character 'r' appear in the word 'strawberry'?<end_of_turn>
<start_of_turn>model
The character 'r' appears **twice** in the word "strawberry".
<end_of_turn>
```

```python
input = tokenizer.encode("strawberry", return_tensors="pt")
print(input)
tokenizer.convert_ids_to_tokens(input[0])
```

```
tensor([[      2, 132077]])
['<bos>', 'strawberry']
```

← the only way the model can answer this correct is that this Q&A was in the train set.

Tokenization needs improvements

Tokenization is at the heart of much weirdness of LLMs. Do not brush it off.

- Why can't LLM spell words? **Tokenization**.
- Why can't LLM do super simple string processing tasks like reversing a string? **Tokenization**.
- Why is LLM worse at non-English languages (e.g. Japanese)? **Tokenization**.
- Why is LLM bad at simple arithmetic? **Tokenization**.
- Why did GPT-2 have more than necessary trouble coding in Python? **Tokenization**.
- Why did my LLM abruptly halt when it sees the string "<|endoftext|>"? **Tokenization**.
- What is this weird warning I get about a "trailing whitespace"? **Tokenization**.
- Why the LLM break if I ask it about "SolidGoldMagikarp"? **Tokenization**.
- Why should I prefer to use YAML over JSON with LLMs? **Tokenization**.
- Why is LLM not actually end-to-end language modeling? **Tokenization**.
- What is the real root of suffering? **Tokenization**.

Recommended code: nanoGPT

nanoGPT

The simplest, fastest repository for training/finetuning medium-sized GPTs. It is a rewrite of minGPT that prioritizes teeth over education. Still under active development, but currently the file `train.py` reproduces GPT-2 (124M) on OpenWebText, running on a single 8XA100 40GB node in about 4 days of training. The code itself is plain and readable: `train.py` is a ~300-line boilerplate training loop and `model.py` a ~300-line GPT model definition, which can optionally load the GPT-2 weights from OpenAI. That's it.

Have instructions on how to run it on CPUs to quickly get a glimpse on how it works.

The best thing is you get an extreme clarity on the inner workings without having to wait for long execution!

I recommend that you use debugger and step through each part, printing out key quantities.

Andrez Karpathy takes it to next step

https://github.com/karpathy/llm.c

llm.c

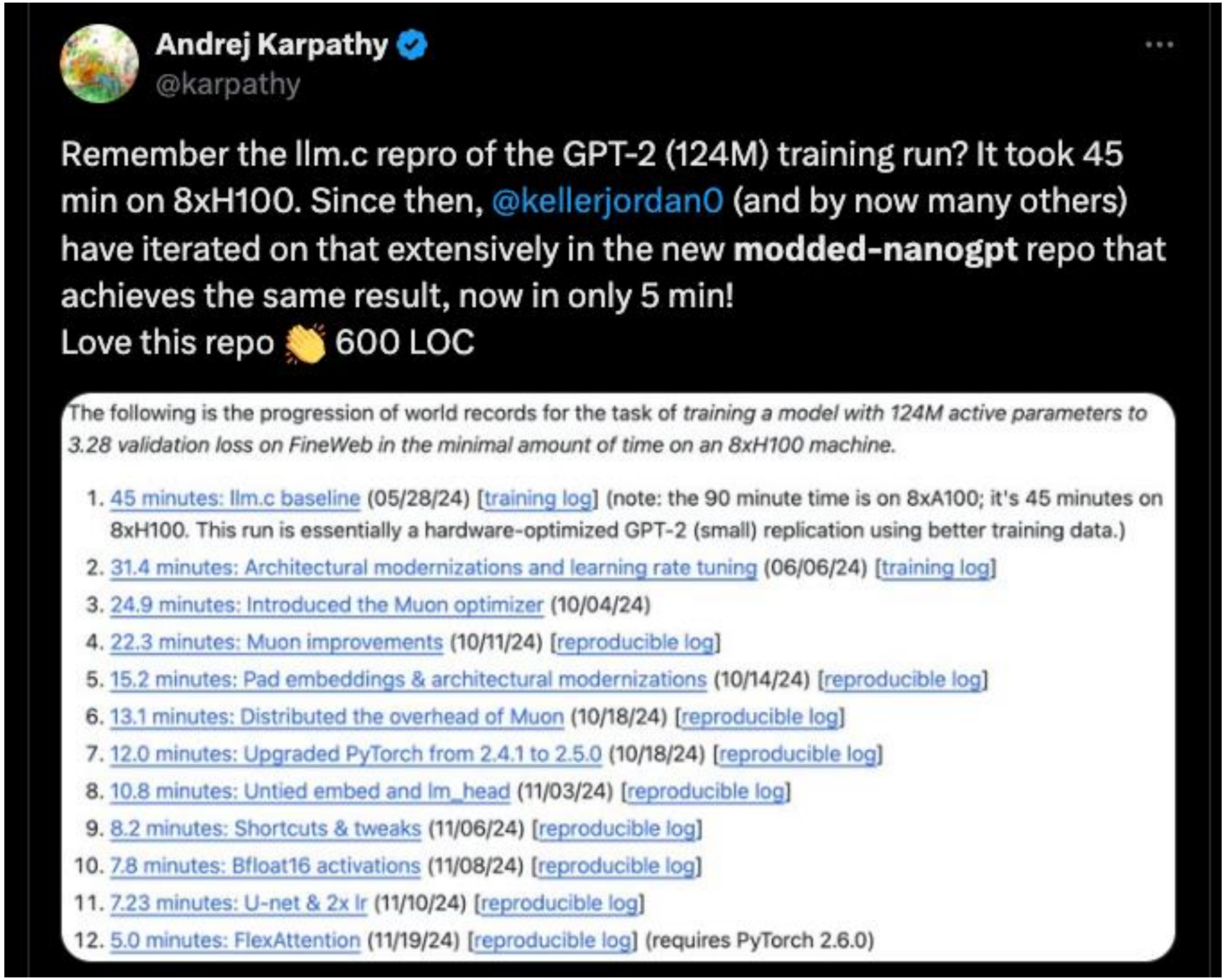
LLMs in simple, pure C/CUDA with no need for 245MB of PyTorch or 107MB of cPython. Current focus is on pretraining, in particular reproducing the GPT-2 and GPT-3 miniseries, along with a parallel PyTorch reference implementation in train_gpt2.py. You'll recognize this file as a slightly tweaked nanoGPT, an earlier project of mine. Currently, llm.c is a bit faster than PyTorch Nightly (by about 7%). In addition to the bleeding edge mainline code in train_gpt2.cu, we have a simple reference CPU fp32 implementation in ~1,000 lines of clean code in one file train_gpt2.c. I'd like this repo to only maintain C and CUDA code. Ports to other languages or repos are very welcome, but should be done in separate repos, and I am happy to link to them below in the "notable forks" section. Developer coordination happens in the Discussions and on Discord, either the `#llmc` channel on the Zero to Hero channel, or on `#llmdotc` on GPU MODE Discord.

45 minutes of training on 8 GPUs of H100

note: one H100 GPU is approximately $30,000

Speedrun competition!

Who can achieve the same performance as GPT-2 within the shortest amount of time?



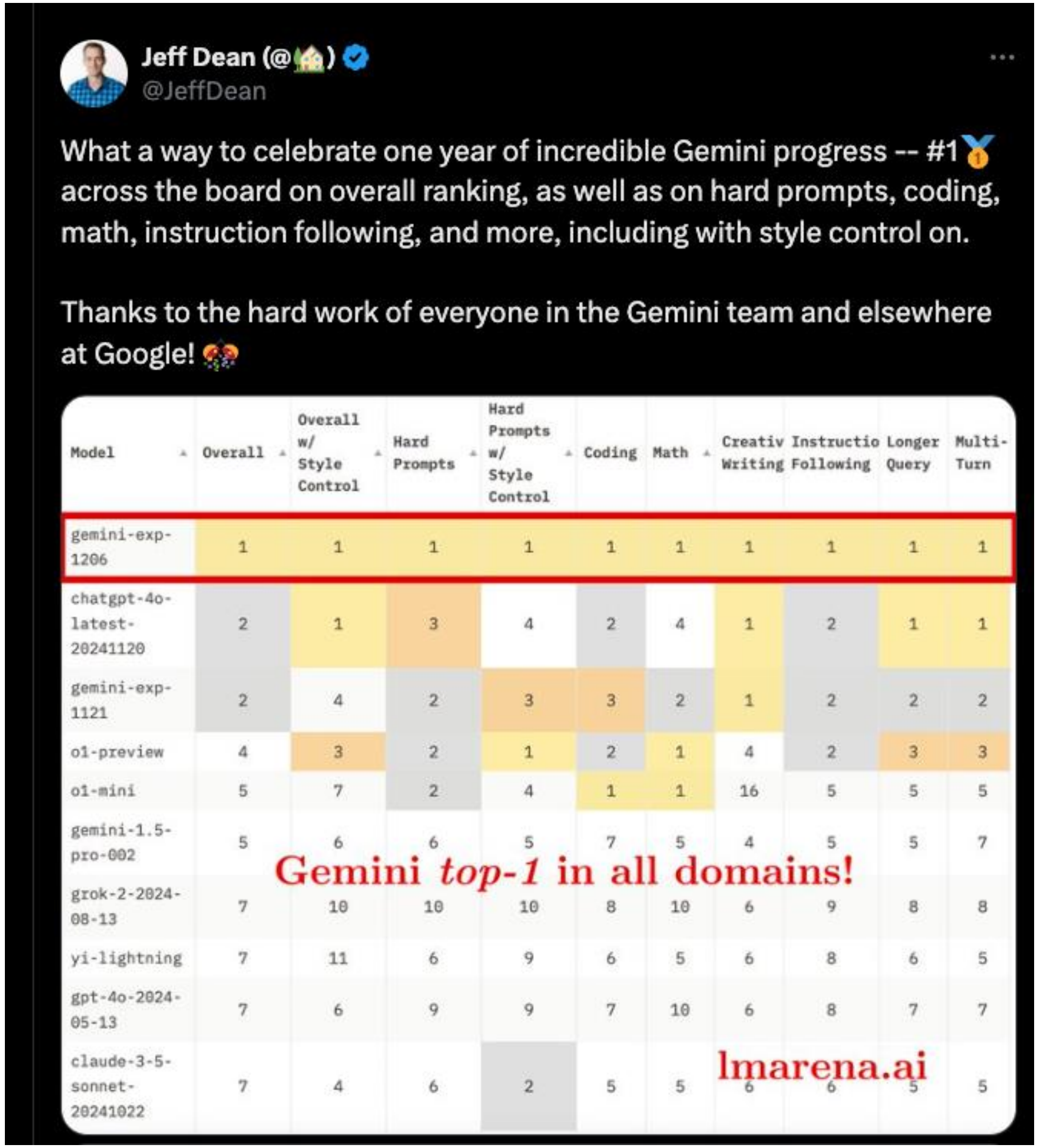
Andrej Karpathy
@karpathy

Remember the llm.c repro of the GPT-2 (124M) training run? It took 45 min on 8xH100. Since then, @kellerjordan0 (and by now many others) have iterated on that extensively in the new **modded-nanogpt** repo that achieves the same result, now in only 5 min!
Love this repo 👏 600 LOC

The following is the progression of world records for the task of *training a model with 124M active parameters to 3.28 validation loss on FineWeb in the minimal amount of time on an 8xH100 machine.*

1. 45 minutes: llm.c baseline (05/28/24) [training log] (note: the 90 minute time is on 8xA100; it's 45 minutes on 8xH100. This run is essentially a hardware-optimized GPT-2 (small) replication using better training data.)
2. 31.4 minutes: Architectural modernizations and learning rate tuning (06/06/24) [training log]
3. 24.9 minutes: Introduced the Muon optimizer (10/04/24)
4. 22.3 minutes: Muon improvements (10/11/24) [reproducible log]
5. 15.2 minutes: Pad embeddings & architectural modernizations (10/14/24) [reproducible log]
6. 13.1 minutes: Distributed the overhead of Muon (10/18/24) [reproducible log]
7. 12.0 minutes: Upgraded PyTorch from 2.4.1 to 2.5.0 (10/18/24) [reproducible log]
8. 10.8 minutes: Untied embed and lm_head (11/03/24) [reproducible log]
9. 8.2 minutes: Shortcuts & tweaks (11/06/24) [reproducible log]
10. 7.8 minutes: Bfloat16 activations (11/08/24) [reproducible log]
11. 7.23 minutes: U-net & 2x lr (11/10/24) [reproducible log]
12. 5.0 minutes: FlexAttention (11/19/24) [reproducible log] (requires PyTorch 2.6.0)

Recent updates



Jeff Dean (@🏡) @JeffDean

What a way to celebrate one year of incredible Gemini progress -- #1🥇 across the board on overall ranking, as well as on hard prompts, coding, math, instruction following, and more, including with style control on.

Thanks to the hard work of everyone in the Gemini team and elsewhere at Google! 🎊

Model	Overall	Overall w/ Style Control	Hard Prompts	Hard Prompts w/ Style Control	Coding	Math	Creativ Writing	Instructio Following	Longer Query	Multi-Turn
gemini-exp-1206	1	1	1	1	1	1	1	1	1	1
chatgpt-4o-latest-20241120	2	1	3	4	2	4	1	2	1	1
gemini-exp-1121	2	4	2	3	3	2	1	2	2	2
o1-preview	4	3	2	1	2	1	4	2	3	3
o1-mini	5	7	2	4	1	1	16	5	5	5
gemini-1.5-pro-002	5	6	6	5	7	5	4	5	5	7
grok-2-2024-08-13	7	10	10	10	8	10	6	9	8	8
yi-lightning	7	11	6	9	6	5	6	8	6	5
gpt-4o-2024-05-13	7	6	9	9	7	10	6	8	7	7
claude-3-5-sonnet-20241022	7	4	6	2	5	5	6	6	5	5

Gemini *top-1* in all domains!

lmarena.ai

try it:
https://aistudio.google.com/app/prompts/new_chat?model=gemini-exp-1206

personal opinion:

- The LLM performance is starting to peter out.
- You cannot compete with industry in this weight-lifting game.
- Recent interests from academia
  - How can we best use LLMs? (e.g., prompt engineering, chain-of-thought)
  - Making LLMs small and fast
    - Improve test time complexity of transformer (length^2)
  - Mitigate potential harms from using LLMs
  - What other problems can benefit from transformer architecture?