
What a language model is

Key takeaways

1. A language model is a function from a sequence of tokens to one probability per possible next token. Everything it writes is produced one token at a time from that distribution.
2. Tokens are learned sub-word pieces, not characters or words. The vocabulary is built by repeatedly merging the most frequent adjacent pair in a corpus.
3. The text you read is sampled from the distribution, not read off its maximum. The same prompt gives different answers because the decoder rolls dice.

Common misunderstandings of LLMs

Which claim would you defend right now? Choose before the evidence.

“A language model looks up the most likely sentence from the text it was trained on.”

“Large language models process only language.”

“Given the same prompt, a model always produces the same text, because it picks the most likely next word.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Three stages, and two of them change weights

Pretraining and post-training set the weights; a prompt sets only this call

Stage	What changes	What it buys
Pretraining	weights, to predict the next token of text	language, code patterns, broad associations
Post-training	weights, toward demonstrations and preferences	instruction following, preferred behavior
Inference	nothing: the prompt shapes this one call	the text you actually read

This deck says what the trained model computes. Thursday's decks say how training sets its weights, and next week's say what data it is set from.

A language model maps text to probabilities

Input: text, and in current models also image patches or audio frames.

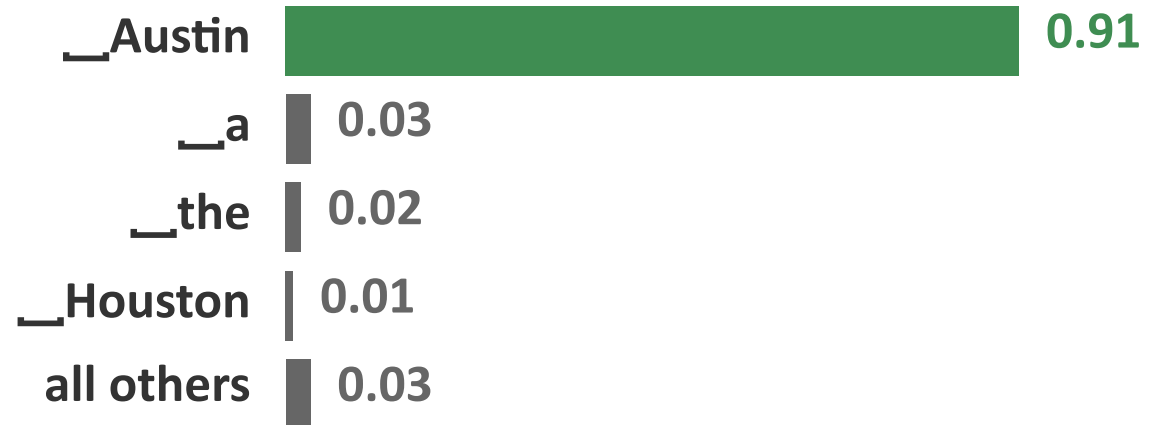
Output: one probability for every possible next token.

“The capital of Texas is”

the prompt, as text

model

one call returns the bars at right



_ marks a space; it is part of the token.

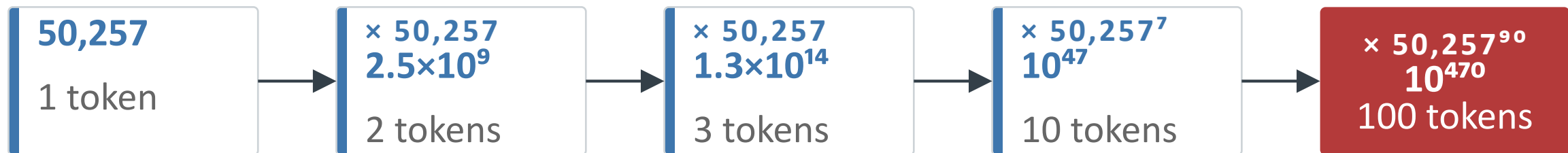
Illustrative values, not a measured model.

The model does not return an answer. It returns a distribution over what could come next.

Probabilities over whole texts explode

No table can hold a probability for every text. Each added token multiplies the number of possible texts by the vocabulary size.

Distinct token sequences with GPT-2's 50,257-token vocabulary:



All counts are approximate: $50,257^n$ texts of n tokens, and $\log_{10} 50,257 \approx 4.70$. A typical agent prompt is thousands of tokens long.

We need a way to factor the problem into pieces a model can compute.

Idea 1: predict one piece at a time

$P(w_1 \dots w_n) = P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_1 w_2) \dots$ is the chain rule. Each factor is one call to the model, and each call sees everything before it.



Generation is a loop around one function

```
tokens = tokenize(prompt)
while not done(tokens):
    probs = model(tokens)           # one call, one distribution
    tokens.append(sample(probs))    # append the draw, call again
```

Autoregressive means the output is fed back as input. That is why the context grows and why generation is slow: one model call per output token.

What should a piece be?

Characters are too small, and words are too many

Characters

About 256 byte values
a tiny vocabulary

Hundreds of steps
for one sentence, one
call each

Little meaning per step
long-range links get
longer

WHAT EVERY CURRENT MODEL USES Sub-word pieces

Tens of thousands
of pieces learned from data

Common words: one piece
rare strings: several

Anything spellable
has a representation

Words

Over a million
in English alone

Every typo is new
and every name, URL,
identifier

An unseen word
has no entry at all

Idea 2: tokenization by byte-pair encoding

Start from bytes and merge the most frequent adjacent pair until the vocabulary is full

```
seq = list(corpus_bytes)
while len(merges) < target:
    pairs = Counter(zip(seq, seq[1:]))
    best = max(pairs, key=pairs.get)
    seq = merge(seq, best); merges.append(best)
```

- `_` marks a space, and it belongs to the piece: `_Austin` and `Austin` are two different tokens.
- An identifier splits on learned pieces: `_in` + `cluding`.

CL100K_BASE, THE GPT-4 TOKENIZER, ON A SENTENCE AND A LINE OF PYTHON: 7, THEN 8 TOKENS

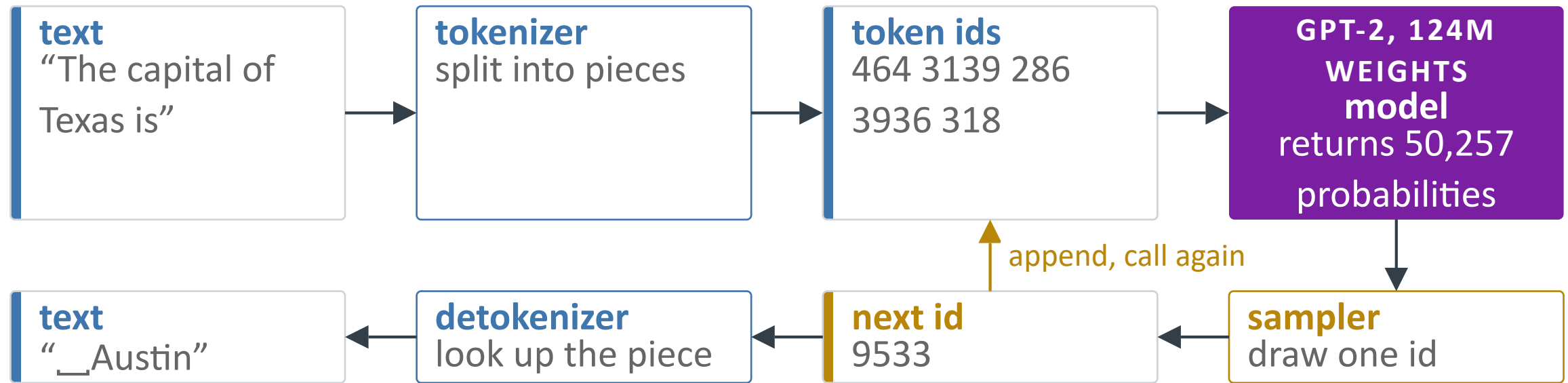
Token ization _is _unbelie vably _useful .

def _compute _total _in cluding _tax (items):

Images and audio are tokenized too, into patch and frame tokens.
“Language” model is a historical name.

So: a language model, precisely

A statistical model over tokens: ids in, one probability per vocabulary entry out, summing to one



The output's length is the vocabulary size: 50,257 for GPT-2, 128,256 for Llama 3.

Instapoll

Instapoll · Review · Nov 5 A

After the prompt “The capital of Texas is”, a model gives the token “ Austin” probability 0.6, “ Houston” probability 0.3, and spreads the rest over other tokens. You call the model 100 times with the default sampling settings. What do you most likely see?

- A. “ Austin” all 100 times, because the model emits its most likely token
- B. “ Austin” about 60 times, “ Houston” about 30, and something else about 10
- C. “ Austin” and “ Houston” about 50 times each, since both are plausible
- D. Whichever of the two comes first in the vocabulary, every time

Producing text: sample, not the maximum

Greedy decoding takes the top token every time. Sampling, the default, draws from the distribution.

GREEDY, 100 CALLS

__Austin (0.6) 100 times of 100
every run identical

SAMPLING, THREE SIMULATED RUNS OF 100

__Austin (0.6) 52, 63, 54 times
__Houston (0.3) 34, 28, 31 times

```
nxt = max(range(V), key=lambda i: probs[i])      # greedy
nxt = random.choices(range(V), weights=probs)[0] # sampling
probs = softmax(logits / T)  # T→0: greedy; T=1: model's own
```

Real decoders also cut off the unlikely tail (top-k, top-p, min-p): tuned heuristics more than theory.

Your agent's answer differs from run to run by design. One run is one sample, not the model's verdict: re-run and cross-check, as in "Models should verify each other."

Demo: a model before and after training

The architecture is a function with 124 million dials, and training is what sets them.

GPT-2 small, prompt “The capital of Texas is”, 25 sampled tokens, same random seed:

Random weights

The capital of Texas is]}wearqueue 436
toursmate flashswer Tick collusion micro Brut
vigorouslynington MyrobyJobrerogotten
meshes Tet \$\$odka advancing transpl...

Trained weights

The capital of Texas is Hearst-Morgan Chase in
a fairytale capital, but has a clear track record
of spinning its \$\$ into advancing its...

Training bought fluent English, not the right answer: sampled freely, this smallest GPT-2 did not say Austin in five tries.

How do we set the dials?

Setting 124 million weights takes three decisions, made in this order.

- 1 Design an architecture with continuous weights**
Thursday: Inside a transformer
- 2 Choose what text to show it**
Next week: Pretraining and post-training
- 3 Pick a learning algorithm that nudges the weights toward that text's statistics**
Thursday: How a model learns

Nothing in the loop writes a rule. The weights move; the program does not.

Key Takeaways

1. A language model maps a sequence of tokens to one probability per possible next token, and generation is that function called once per output token, each call seeing everything before it. The output is fed back as input, which is why context grows and generation is slow.
2. Tokens are learned sub-word pieces: byte-pair encoding merges the most frequent adjacent pair until the vocabulary is full, so common words are one token and rare strings several. Images and audio are cut into patch and frame tokens too, so one model can read them all.
3. Text is sampled from the distribution, not read off its maximum. Greedy decoding is deterministic and repetitive; sampling with a temperature is the default, so one run is one sample. Treat a single agent answer as a draw, not a verdict, and verify it.

Tokens, by hand

10 min

- In pairs, tokenize both strings with the merge table, applying the merges in order. Write the pieces and count them.
- Which string costs more context, and why does the table favor one of them?
- Decision: your agent proposes shortening identifiers across the codebase to cut token spend. Would you accept? Name one cost the token count does not show.
- Compare with a neighbor: did you count the same? Where did your merges differ?

Merge table: apply 1 to 14, in order

1 _ + t → _t	8 _t + otal → _total
2 h + e → he	9 d + e → de
3 a + l → al	10 de + r → der
4 i + n → in	11 o + r → or
5 o + t → ot	12 or + der → order
6 ot + al → otal	13 t + he → the
7 _ + in → _in	14 _ + order → _order

Two identifiers, 22 characters each

```
A  the_total_in_the_order
B  tttl_n_th_ordr_x9q_v2
```

Start from single characters. A merge joins every adjacent pair that matches it exactly, left to right.