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.

Pretraining, post-training, and inference

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

Pretraining

Text corpus
Predict the next token

learn from text

Weights change

Language, code patterns, broad associations

Post-training

Demonstrations
and preferences

learn behavior

Weights change

Instruction following, preferred behavior

Inference

Prompt
Shapes this one call

run the model

Weights fixed

The text you actually read

This deck says what the trained model computes. The next lessons 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.

Why predict just one token at a time?

Suppose we gave every possible answer its own output probability. Even short answers would require an impossibly large output.

Output probabilities needed for whole answers, using GPT-2's vocabulary:



For an n-token answer, the output would have $50{,}257^n$ entries.

Predicting one token needs only 50,257 output probabilities per call. The chain rule lets us build a whole answer from these small distributions.

Generating text one token at a time

$P(w_1 \ldots w_n) = P(w_1) \cdot P(w_2 \mid w_1) \cdot P(w_3 \mid w_1\, w_2) \cdots$ 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.

Choosing the token vocabulary

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

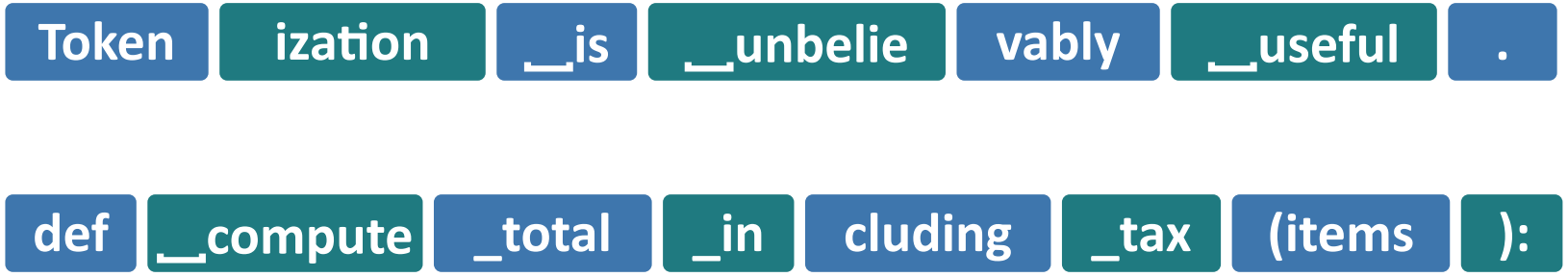
Tokenization by byte-pair encoding

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

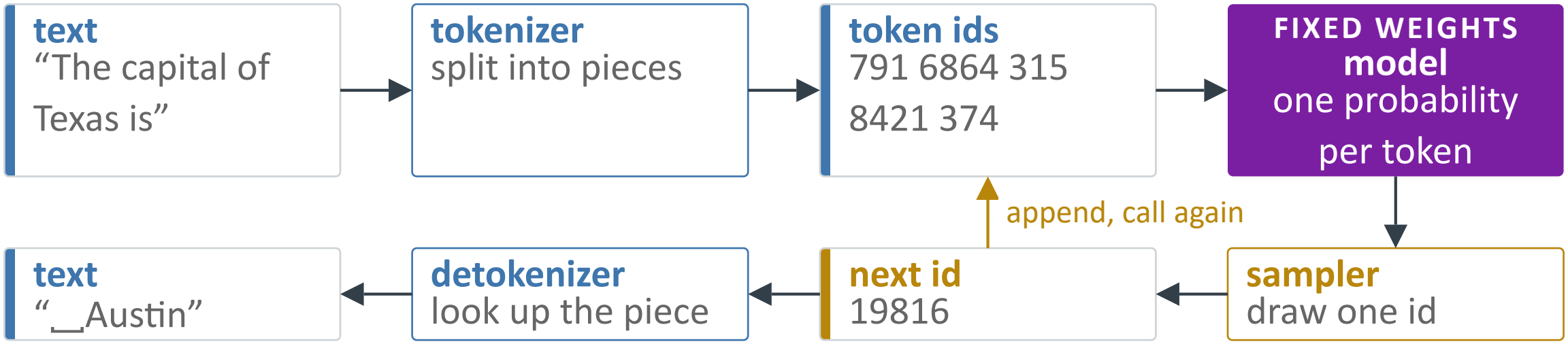


Images and audio are tokenized too, into patch and frame tokens. “Language” model is a historical name.

source: Sennrich, Haddow & Birch, arxiv.org/abs/1508.07909; byte-level BPE from Radford et al. 2019 (GPT-2); splits from github.com/openai/tiktoken

What a language model computes

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



Each call returns one probability per vocabulary token. The sampler chooses one token.

source: Token IDs: OpenAI tiktoken, cl100k_base (as on the preceding slide), github.com/openai/tiktoken

Instapoll

Instapoll · Review · Nov 3 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

Greedy decoding and sampling

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
```

Temperature T reshapes the probabilities before sampling. Lower T favors likely tokens; higher T gives less likely tokens more chance. T = 1 leaves the distribution unchanged.

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."

source: Holtzman et al., The Curious Case of Neural Text Degeneration, arxiv.org/abs/1904.09751 (2020)

Training changes weight matrices and vectors

GPT-2 small, 25 sampled tokens, same seed. Matrix slices are illustrative.

Random weights

```
[[ 0.02, -0.01,  0.03],
 [-0.04,  0.01, -0.02]]
```

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

Trained weights

```
[[ 0.01, -0.08,  0.04],
 [-0.02,  0.06, -0.03]]
```

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.

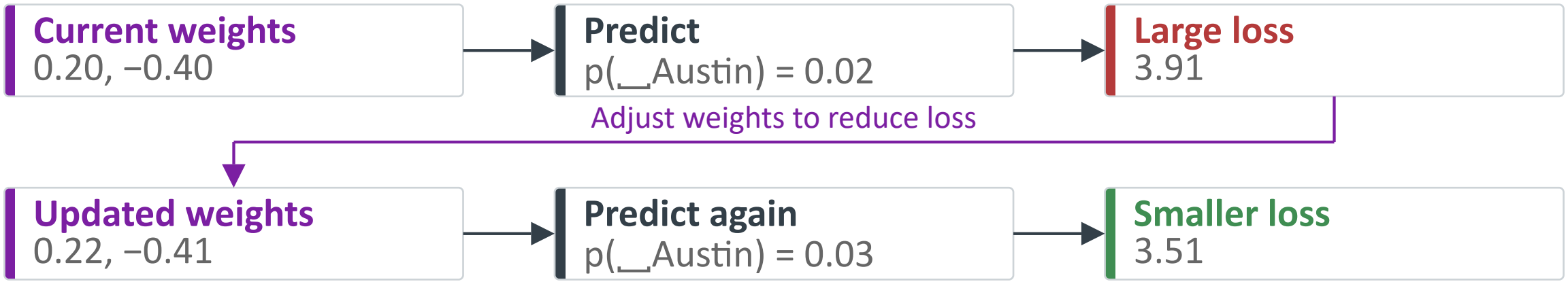
source: Radford et al., Language Models are Unsupervised Multitask Learners (OpenAI, 2019); weights and sampler from Hugging Face transformers (huggingface.co/openai-community/gpt2)

Errors guide weight updates

Prompt: “The capital of Texas is”. The training text gives the target “_Austin”.

Low probability for the target incurs a large penalty, called the loss.

SAME PROMPT AND TARGET · ILLUSTRATIVE VALUES · TWO WEIGHTS SHOWN



Backpropagation computes how each weight affects the loss. Apply small increases or decreases to reduce it, and repeat over many training examples.

source: PyTorch, Optimizing Model Parameters, docs.pytorch.org/tutorials/beginner/basics/optimization_tutorial.html

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.

Match, then discuss

10 min

	Operation
1	Convert text to integer token IDs.
2	Return a probability for each next token.
3	Draw one token from those probabilities.
4	Change weight values from training examples.
5	Make the token distribution sharper or flatter.
6	Always choose the highest-probability token.

	Concept (use each once)
A	Greedy decoding
B	Training
C	Tokenization
D	Temperature
E	Sampling
F	One model call

1. Change the training data

How can you get a trained model to make more mistakes if you can modify its training data? Give one concrete edit. Would it work without retraining?

2. Probability and exact answers

An agent must compute 347 × 29. It can generate the answer or call a calculator. Both the answer and the tool arguments are predicted tokens. What errors can the calculator eliminate, and what errors can remain? Would always choosing the most likely token guarantee correctness?