

Large  
Language  
Models

# Introduction to Large Language Models

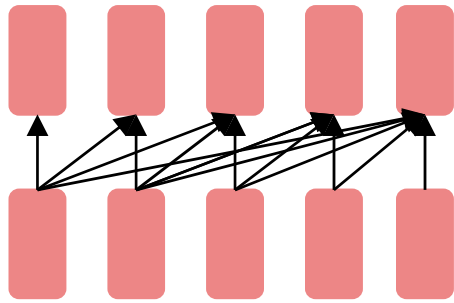
# Language models

- Remember the simple n-gram language model
  - Assigns probabilities to sequences of words
  - Generate text by sampling possible next words
  - Is trained on counts computed from lots of text
- Large language models are similar and different:
  - Assigns probabilities to sequences of words
  - Generate text by sampling possible next words
  - **Are trained by learning to guess the next word**

# Large language models

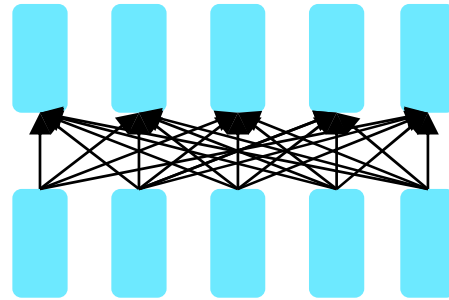
- Even though pretrained only to predict words
- Learn a lot of useful language knowledge
- Since training on a **lot** of text

# Three architectures for large language models



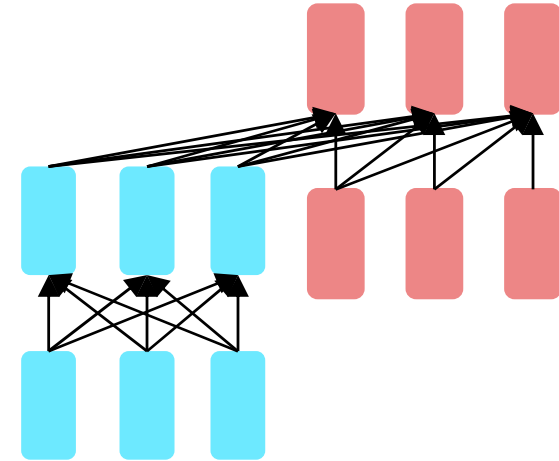
## Decoders

GPT, Claude,  
Llama,  
Mixtral



## Encoders

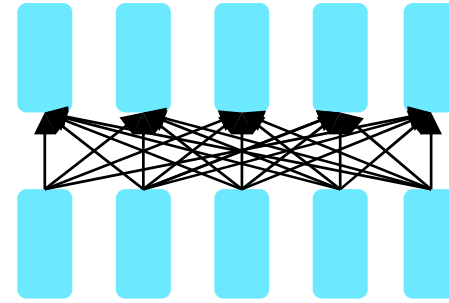
BERT family,  
HuBERT



## Encoder-decoders

Flan-T5, Whisper

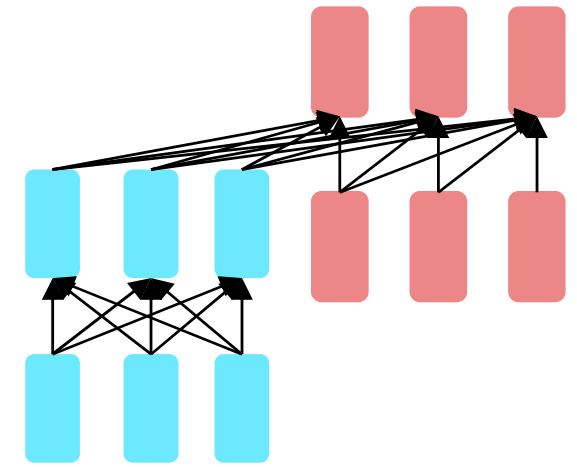
# Encoders



Many varieties!

- Popular: Masked Language Models (MLMs)
- BERT family
- Trained by predicting words from surrounding words on both sides
- Are usually **finetuned** (trained on supervised data) for classification tasks.

# Encoder-Decoders



- Trained to map from one sequence to another
- Very popular for:
  - machine translation (map from one language to another)
  - speech recognition (map from acoustics to words)

# Large Language Models

Large Language Models: What  
tasks can they do?

Big idea

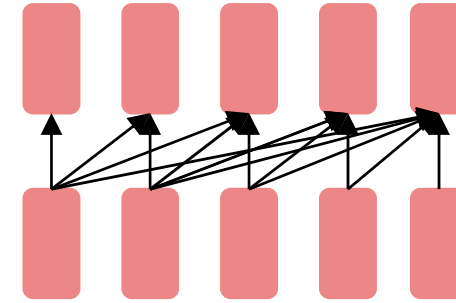
Many tasks can be turned into tasks of predicting words!



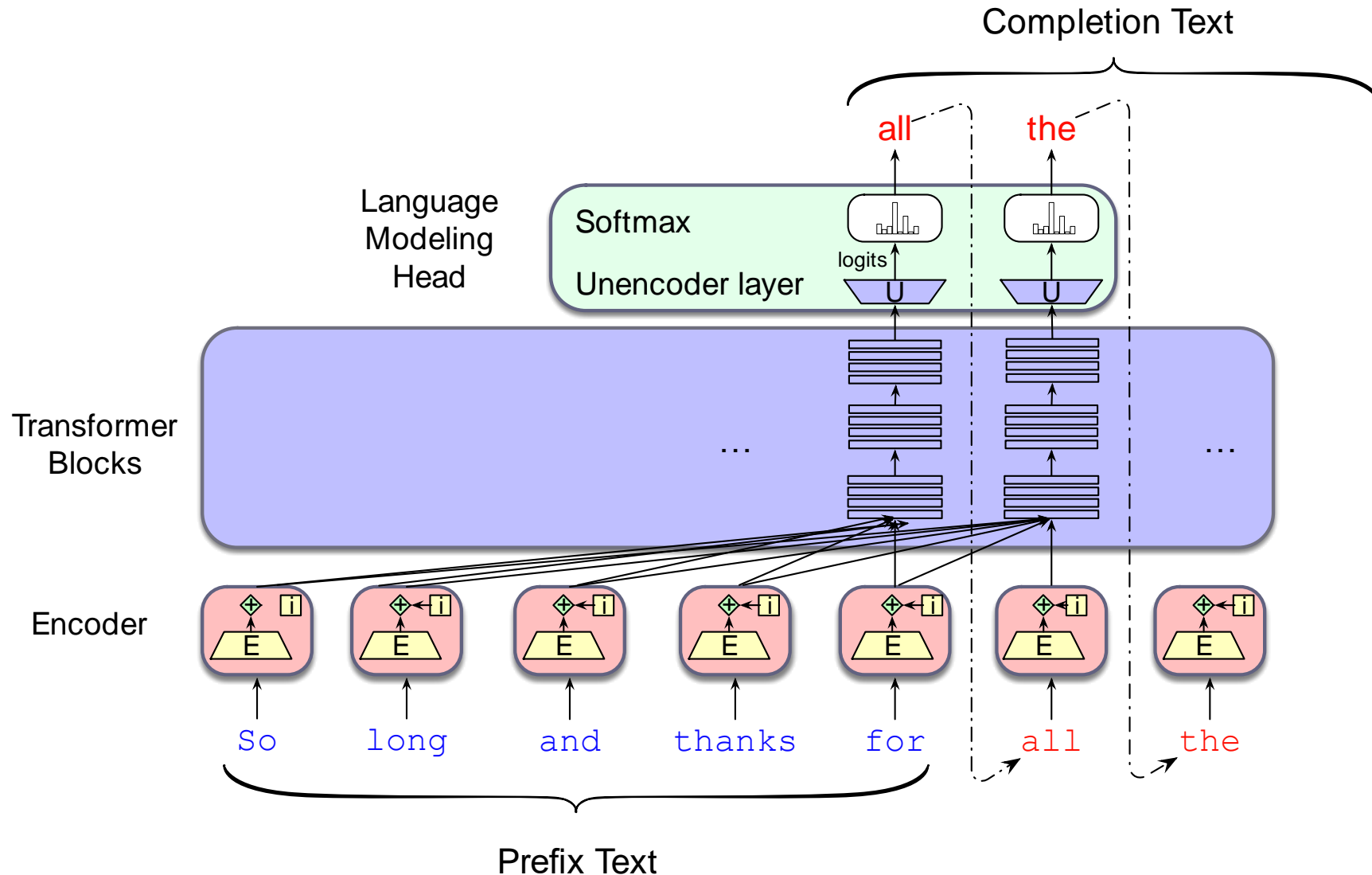
# This lecture: decoder-only models

Also called:

- Causal LLMs
- Autoregressive LLMs
- Left-to-right LLMs
- Predict words left to right



# Conditional Generation: Generating text conditioned on previous text!



Many practical NLP tasks can be cast as word prediction!

Sentiment analysis: “I like Jackie Chan”

1. We give the language model this string:  
The sentiment of the sentence "I like Jackie Chan" is:
2. And see what word it thinks comes next:

$P(\text{positive} | \text{The sentiment of the sentence ``I like Jackie Chan" is:})$

$P(\text{negative} | \text{The sentiment of the sentence ``I like Jackie Chan" is:})$

# Framing lots of tasks as conditional generation

QA: “Who wrote The Origin of Species”

1. We give the language model this string:

Q: Who wrote the book ``The Origin of Species"? A:

2. And see what word it thinks comes next:

$P(w|Q: \text{Who wrote the book ``The Origin of Species"? A:})$

3. And iterate:

$P(w|Q: \text{Who wrote the book ``The Origin of Species"? A: Charles})$

# Summarization

The only thing crazier than a guy in snowbound Massachusetts boxing up the powdery white stuff and offering it for sale online? People are actually buying it. For \$89, self-styled entrepreneur Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says.

Original

But not if you live in New England or surrounding states. “We will not ship snow to any states in the northeast!” says Waring’s website, ShipSnowYo.com. “We’re in the business of expunging snow!”

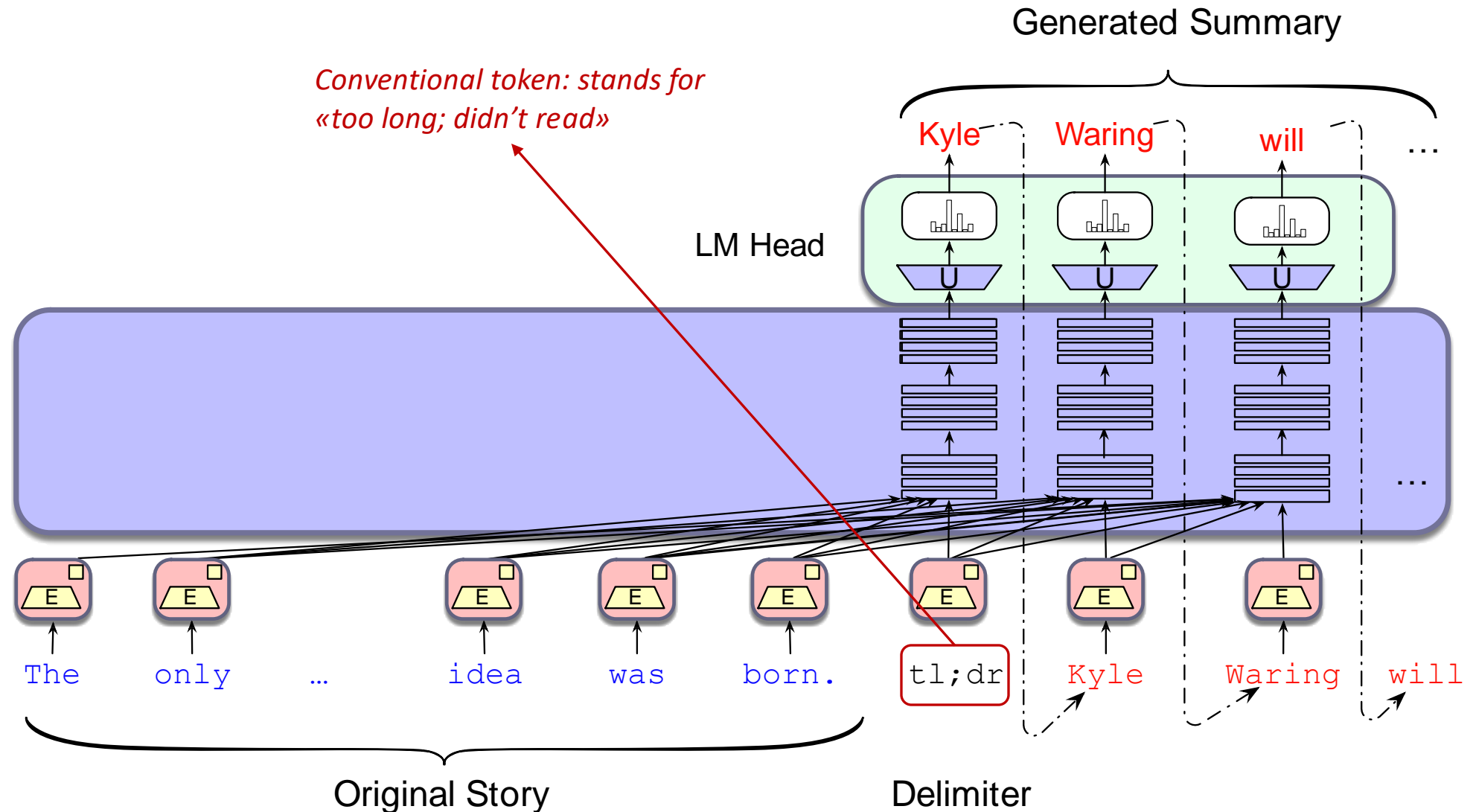
His website and social media accounts claim to have filled more than 133 orders for snow – more than 30 on Tuesday alone, his busiest day yet. With more than 45 total inches, Boston has set a record this winter for the snowiest month in its history. Most residents see the huge piles of snow choking their yards and sidewalks as a nuisance, but Waring saw an opportunity.

According to Boston.com, it all started a few weeks ago, when Waring and his wife were shoveling deep snow from their yard in Manchester-by-the-Sea, a coastal suburb north of Boston. He joked about shipping the stuff to friends and family in warmer states, and an idea was born. [...]

Summary

Kyle Waring will ship you 6 pounds of Boston-area snow in an insulated Styrofoam box – enough for 10 to 15 snowballs, he says. But not if you live in New England or surrounding states.

# LLMs for summarization (using tl;dr)



Large  
Language  
Models

# Sampling for LLM Generation

# Decoding and Sampling

This task of choosing a word to generate based on the model's probabilities is called **decoding**.

The most common method for decoding in LLMs: **sampling**.

Sampling from a model's distribution over words:

- choose random words according to their probability assigned by the model.

After each token we'll sample words to generate according to their probability *conditioned on our previous choices*,

- A transformer language model will give the probability



# Random sampling

$i \leftarrow 1$

$w_i \sim p(w)$

**while**  $w_i \neq \text{EOS}$

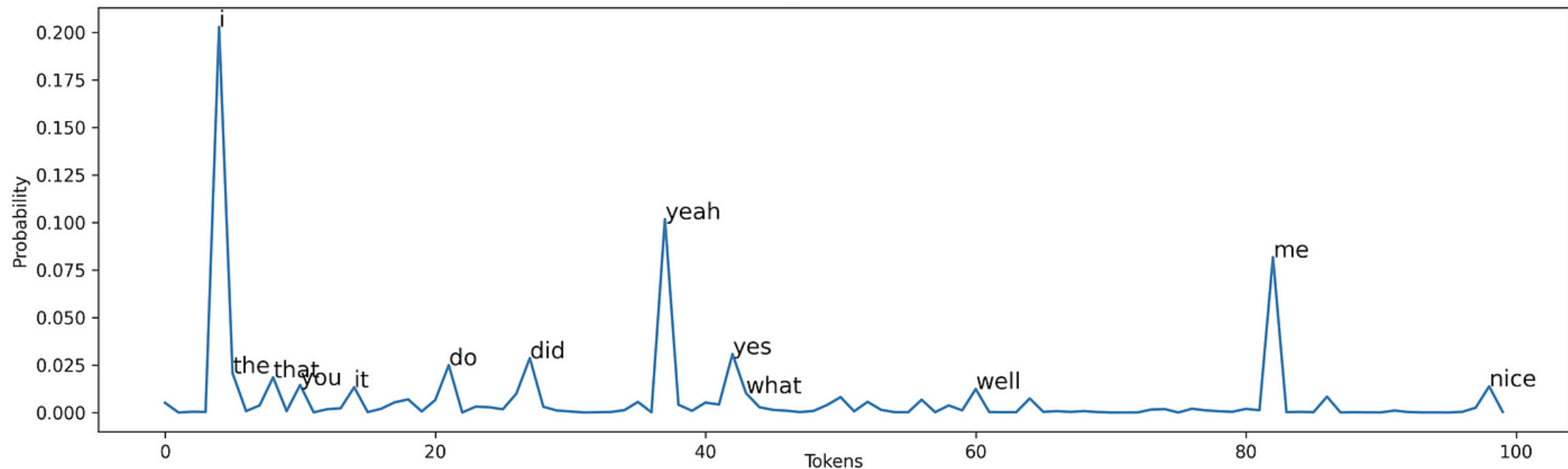
$i \leftarrow i + 1$

$w_i \sim p(w_i \mid w_{<i})$

# Random sampling

Let follow an example with the context: *I love watching movies*

- *I, yeah* and *me* are the most probable tokens



Random sampling doesn't work very well

Even though random sampling mostly generate sensible, high-probable words,

There are many odd, low- probability words in the tail of the distribution

Each one is low- probability but added up they constitute a large portion of the distribution

So they get picked enough to generate weird sentences

# Factors in word sampling: **quality** and **diversity**

Emphasize **high-probability** words

+ **quality**: more accurate, coherent, and factual,

- **diversity**: boring, repetitive.

Emphasize **middle-probability** words

+ **diversity**: more creative, diverse,

- **quality**: less factual, incoherent

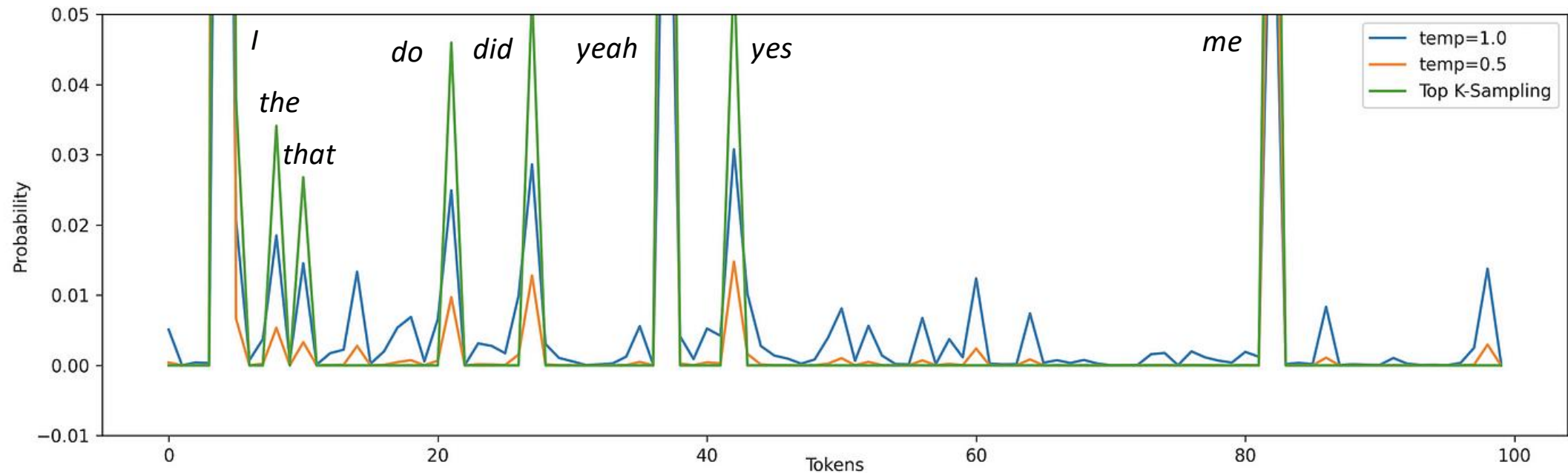
# Top-k sampling:

1. Choose # of words  $k$
2. For each word in the vocabulary  $V$ , use the language model to compute the likelihood of this word given the context  $p(w_t | \mathbf{w}_{<t})$
3. Sort the words by likelihood, keep only the top  $k$  most probable words.
4. Renormalize the scores of the  $k$  words to be a legitimate probability distribution.
5. Randomly sample a word from within these remaining  $k$  most-probable words according to its probability.

# Top- $k$ sampling

In our running example we sampled with  $k=8$

- Compare with the blue line



# Top-p sampling (= nucleus sampling)

Holtzman et al., 2020

Problem with top-k:  $k$  is fixed so may cover very different amounts of probability mass in different situations

Idea: Instead, keep the top  $p$  percent of the probability mass

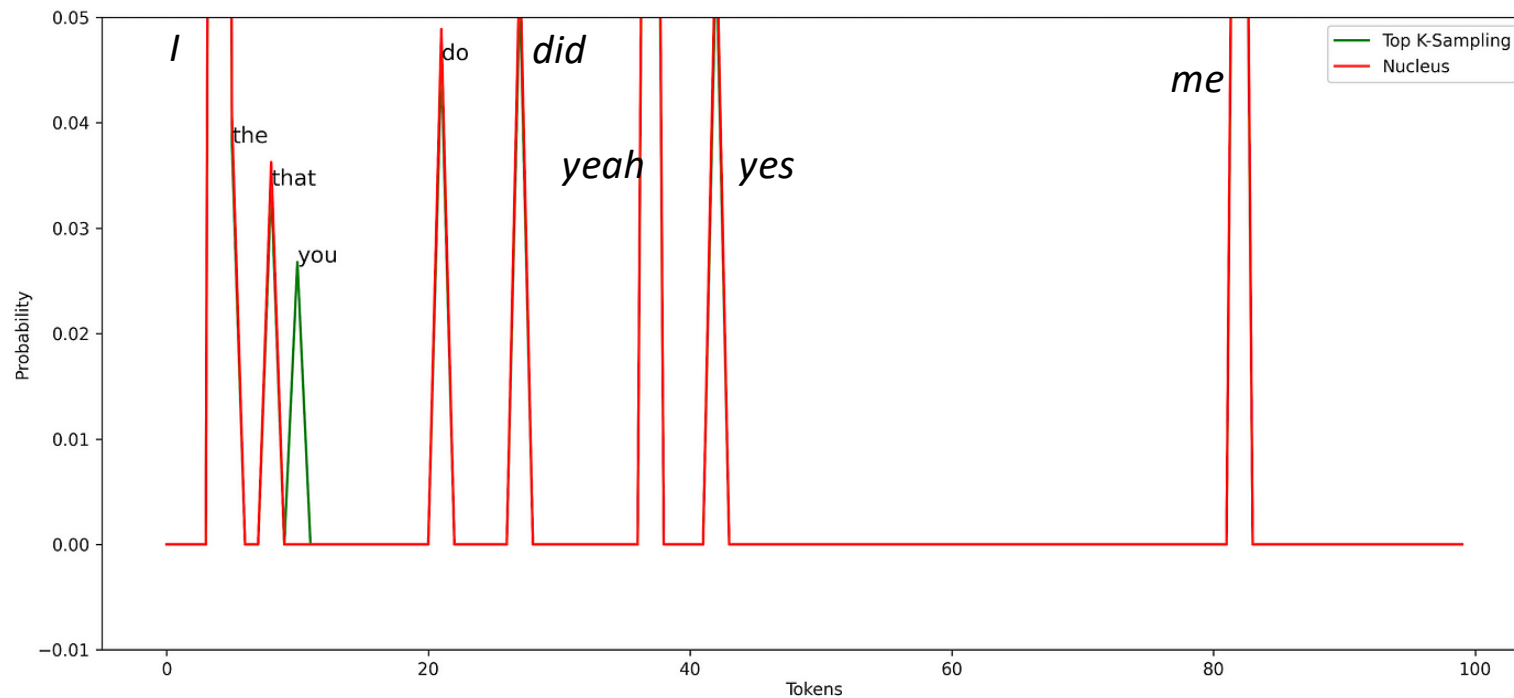
Given a distribution  $P(w_t | \mathbf{w}_{<t})$ , the top- $p$  vocabulary  $V(p)$  is the smallest set of words such that

$$\sum_{w \in V(p)} P(w | \mathbf{w}_{<t}) \geq p$$

*Sort of adaptive selection  
of the  $k$  value*

# Top-p sampling (= nucleus sampling)

In our running example, «you» was not sampled due to the value of the  $p$  threshold





# Temperature sampling

Reshape the distribution instead of truncating it

Intuition from thermodynamics

- a system at high temperature is flexible and can explore many possible states,
- a system at lower temperature is likely to explore a subset of lower energy (better) states.

In **low-temperature sampling**, ( $\tau \leq 1$ ) we smoothly

- increase the probability of the most probable words
- decrease the probability of the rare words.

# Temperature sampling

Divide the logit by a temperature parameter  $\tau$  before passing it through the softmax.

Instead of

$$\text{ ~~$\mathbf{y} = \text{softmax}(u)$~~ }$$

We do

$$\mathbf{y} = \text{softmax}(u/\tau)$$

# Temperature sampling

$$0 \leq \tau \leq 1$$

$$\mathbf{y} = \text{softmax}(\mathbf{u} / \tau)$$

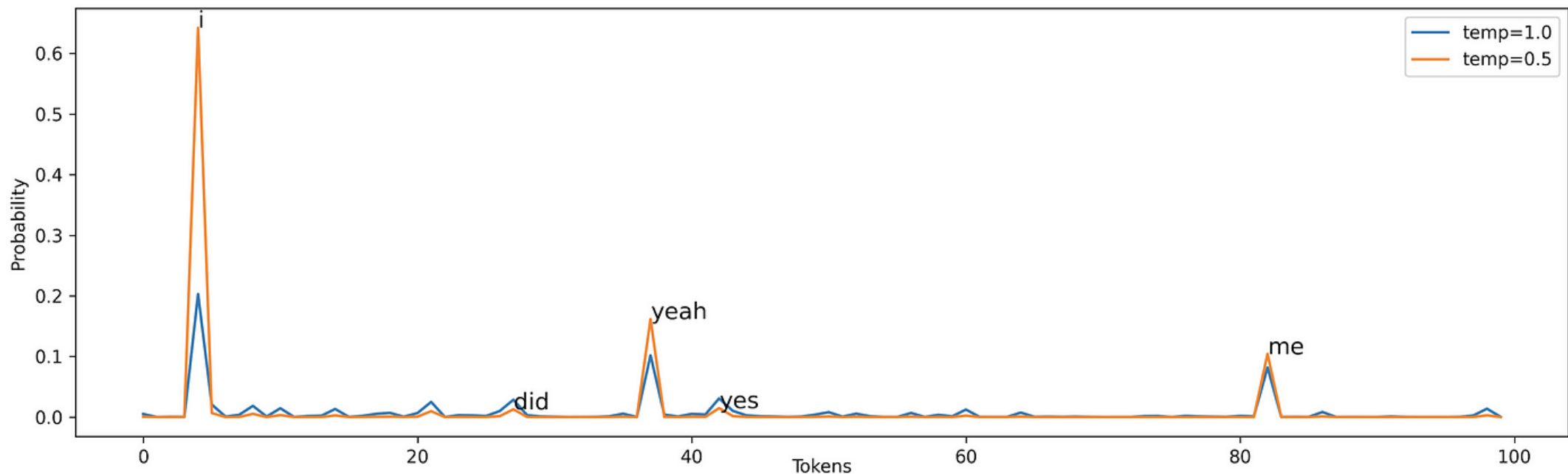
## Why does this work?

- When  $\tau$  is close to 1 the distribution doesn't change much.
- The lower  $\tau$  is, the larger the scores being passed to the softmax
- Softmax pushes high values toward 1 and low values toward 0.
- Large inputs pushes high-probability words higher and low probability word lower, making the distribution more greedy.
- As  $\tau$  approaches 0, the probability of most likely word approaches 1

# Temperature sampling

In our running example, again *I*, *yeah*, and *me* have increased chance to be selected

- Compare with the blue line



Large  
Language  
Models

## Pretraining Large Language Models: Algorithm

# Pretraining

The big idea that underlies all the amazing performance of language models

First **pretrain** a transformer model on enormous amounts of text

Then **apply** it to new tasks.

# Self-supervised training algorithm

We just train them to predict the next word!

1. Take a corpus of text
2. At each time step  $t$ 
  - i. ask the model to predict the next word
  - ii. train the model using gradient descent to minimize the error in this prediction

"**Self-supervised**" because it just uses the next word as the label!

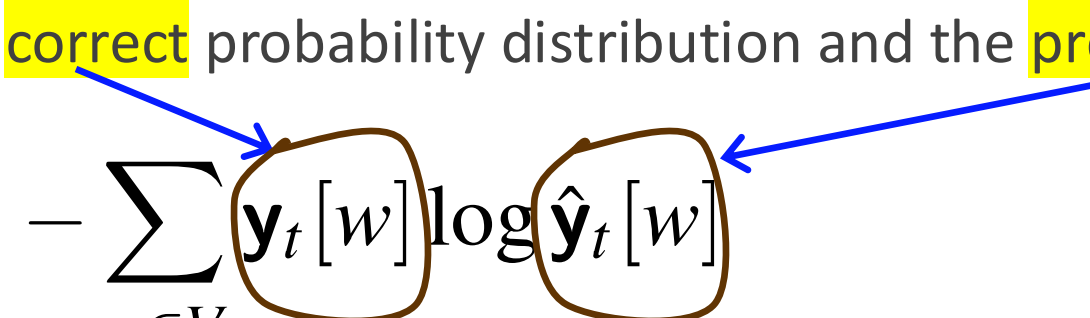
# Intuition of language model training: loss

- Same loss function: **cross-entropy loss**
  - We want the model to assign a high probability to true word  $w$
  - = want loss to be high if the model assigns too low a probability to  $w$
- CE Loss: The negative log probability that the model assigns to the true next word  $w$ 
  - If the model assigns too low a probability to  $w$
  - We move the model weights in the direction that assigns a higher probability to  $w$



# Cross-entropy loss for language modeling

**CE loss:** difference between the **correct** probability distribution and the **predicted** distribution

$$L_{CE} = - \sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$$


The correct distribution  $\mathbf{y}_t$  knows the next word, so is 1 for the actual next word and 0 for the others.

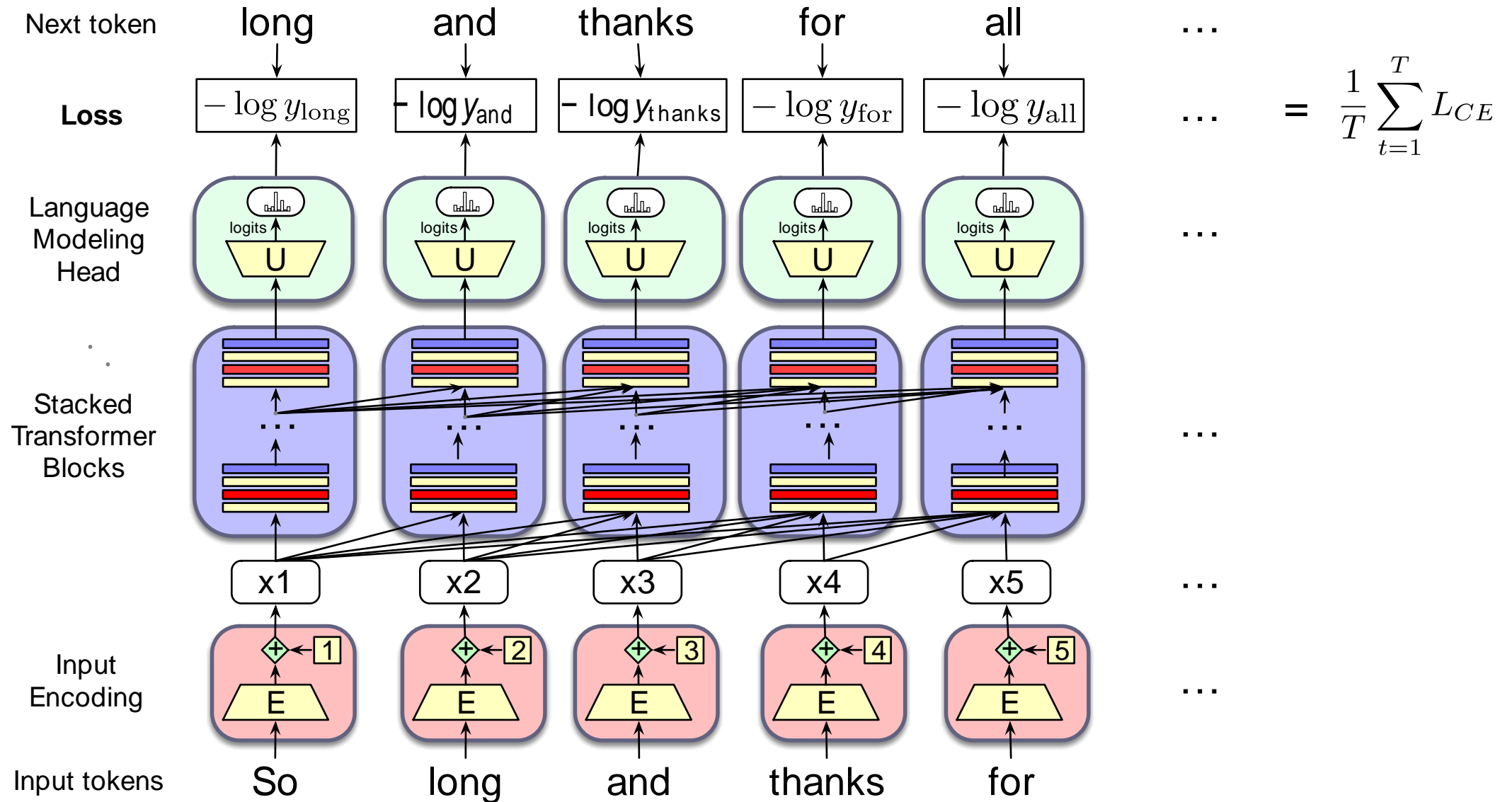
So in this sum, all terms get multiplied by zero except one: the log probability the model assigns to the correct next word, and:

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$

# Teacher forcing

- At each token position  $t$ , model sees correct tokens  $w_{1:t}$ 
  - Computes loss ( $-\log$  probability) for the next token  $w_{t+1}$
- At next token position  $t+1$  we ignore what model predicted for  $w_{t+1}$ 
  - Instead, we take the **correct** word  $w_{t+1}$ , add it to context, move on

# Training a transformer language model



Large  
Language  
Models

Pretraining data for LLMs

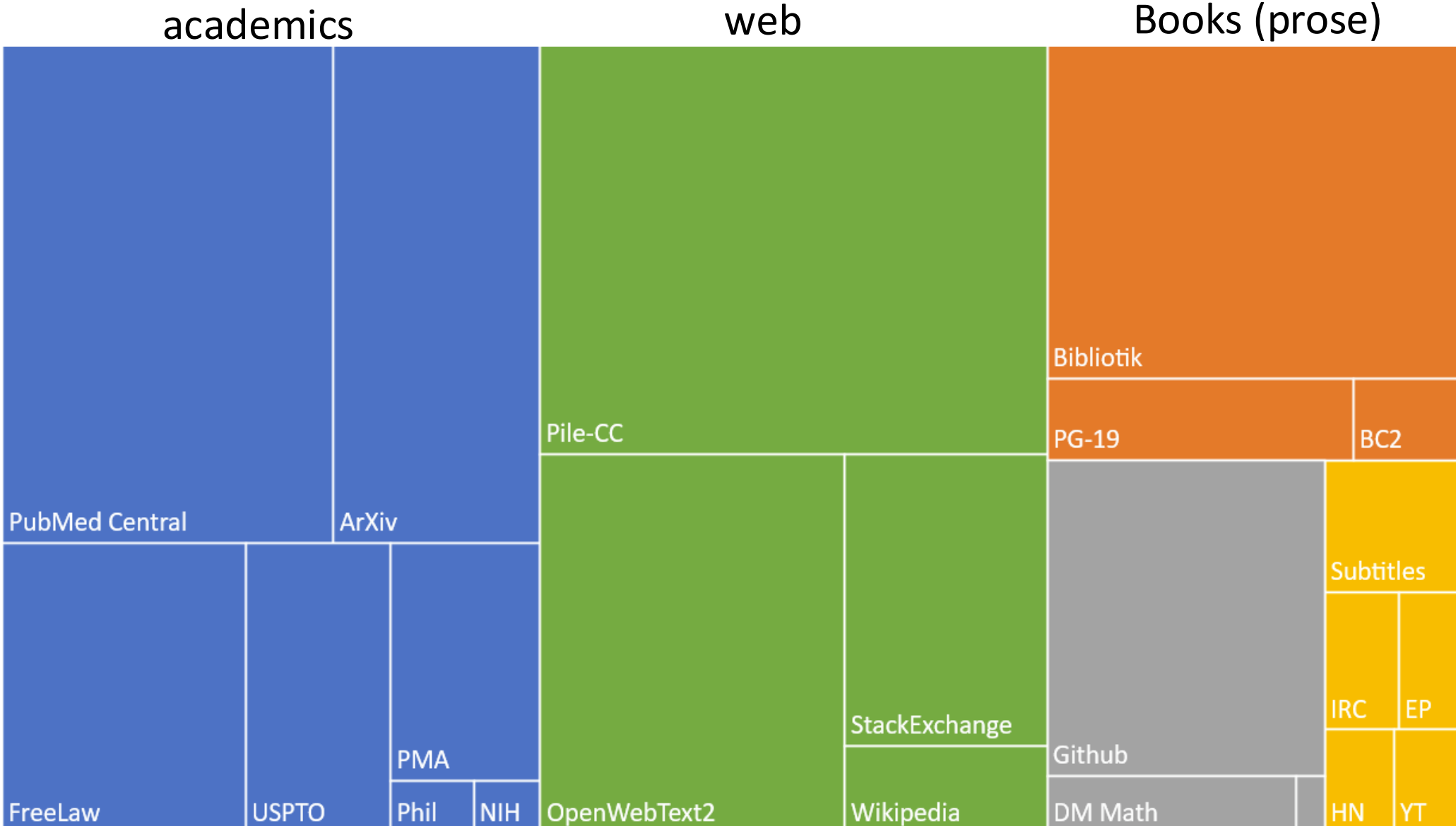
LLMs are mainly trained on the web

Common crawl, snapshots of the entire web produced by the non-profit Common Crawl with billions of pages

Colossal Clean Crawled Corpus (C4; [Raffel et al. 2020](#)), 156 billion tokens of English, filtered

What's in it? Mostly patent text documents, Wikipedia, and news sites

# The Pile: a pretraining corpus



# Filtering for quality and safety

## Quality is subjective

- Many LLMs attempt to match Wikipedia, books, particular websites
- Need to remove boilerplate text, adult content, PII
- Deduplication at many levels (URLs, documents, even lines)

## Safety also subjective

- Toxicity detection is important, although that has mixed results
- Can mistakenly flag data written in dialects like African American English

# But there are problems with scraping from the web

**Copyright:** much of the text in these datasets is copyrighted

- Not clear if fair use doctrine in US allows for this use
- This remains an open legal question

**Data consent**

- Website owners can indicate they don't want their site crawled

**Privacy:**

- Websites can contain private IP addresses and phone numbers



Large  
Language  
Models

Finetuning

# Finetuning for daptation to new domains

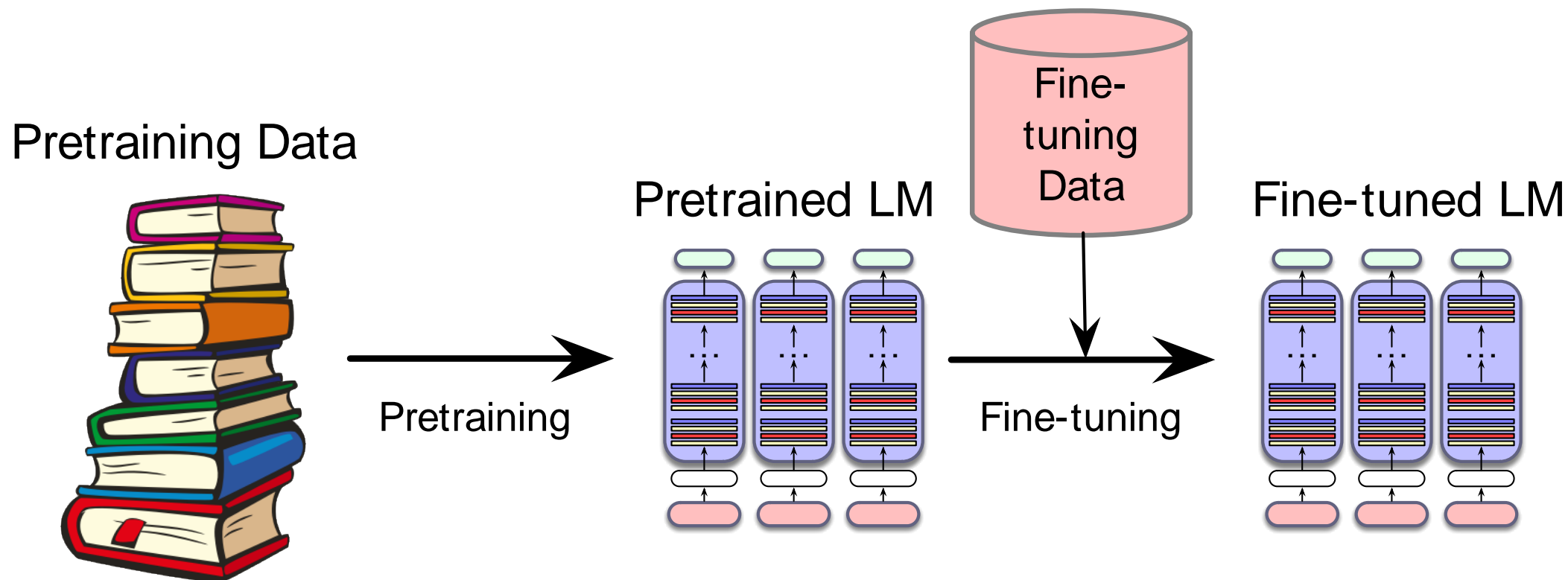
What happens if we need our LLM to work well on a domain it didn't see in pretraining?

Perhaps some specific medical or legal domain?

Or maybe a multilingual LM needs to see more data on some language that was rare in pretraining?

# Finetuning

**Finetuning:** taking a fully pretrained model and running additional training passes on some application/domain specific new data.



# "Finetuning" means 4 different things

1. Adding extra neural circuitry (an extra *head*) after the top layer of the model to use it as a classifier (we saw this for MLMs)
2. Continued pretraining
3. Parameter-efficient finetuning, or PEFT
4. Supervised finetuning, or SFT

# Continued pretraining

- Further train *all the parameters* of model on new data
  - Using the same method (word prediction) and loss function (cross-entropy loss) as for pretraining.
  - As if the new data were at the tail end of the pretraining data

# Parameter-efficient finetuning

Retraining all the parameters of the model is very slow and expensive when the language model is huge.

We can *freeze* some of the parameters (i.e., leave them unchanged from their pretrained value) and train only a subset of parameters on the new data.

PEFT because we efficiently select specific parameters to update when finetuning.

# Supervised finetuning

Often used for *instruction finetuning*, in which we want a pretrained language model to learn to follow text instructions.

We create a dataset of prompts and desired responses (for example questions and their answers, or commands and their fulfillments)

We train the language model using the normal cross-entropy loss to predict each token in the instruction prompt iteratively (that is to produce the desired response from the command in the prompt).

Large  
Language  
Models

Evaluating Large Language  
Models



# Perplexity

Just as for n-gram grammars, we use perplexity to measure how well the LM predicts unseen text

The perplexity of a model  $\theta$  on an unseen test set is the *inverse probability that  $\theta$  assigns to the test set, normalized by the test set length*.

For a test set of  $n$  tokens  $w_{1:n}$  the perplexity is :

$$\begin{aligned} \text{Perplexity}_{\theta}(w_{1:n}) &= P_{\theta}(w_{1:n})^{-\frac{1}{n}} \\ &= \sqrt[n]{\frac{1}{P_{\theta}(w_{1:n})}} \end{aligned} \quad \text{Perplexity}_{\theta}(w_{1:n}) = \sqrt[n]{\prod_{i=1}^n \frac{1}{P_{\theta}(w_i | w_{<i})}}$$

# Why perplexity instead of raw probability of the test set?

- Probability depends on size of test set
  - Probability gets smaller the longer the text
  - Better: a metric that is **per-word**, normalized by length
- **Perplexity** is the inverse probability of the test set, normalized by the number of words  
(The inverse comes from the original definition of perplexity from cross-entropy rate in information theory)

Probability range is  $[0,1]$ , perplexity range is  $[1,\infty]$

# Perplexity

- The higher the probability of the word sequence, the lower the perplexity.
- Thus the lower the perplexity of a model on the data, the better the model.
- *Minimizing perplexity is the same as maximizing probability*

**Also:** perplexity is sensitive to length/tokenization so best used when comparing LMs that use the same tokenizer.

# Many other factors that we evaluate, like:

## **Size**

Big models take lots of GPUs and time to train, memory to store

## **Energy usage**

Can measure kWh or kilograms of CO<sub>2</sub> emitted

## **Fairness**

Benchmarks measure gendered and racial stereotypes, or decreased performance for language from or about some groups.

Large  
Language  
Models

## Dealing with Scale

# Scaling Laws

LLM performance depends on

- **Model size:** the number of parameters not counting embeddings
- **Dataset size:** the amount of training data
- **Compute:** Amount of compute (in FLOPS or other measures)

# Scaling Laws

Can improve a model by adding parameters (more layers, wider contexts), more data, or training for more iterations.

The performance of a large language model (the loss) scales as a power-law with each of these three.

# Scaling Laws

Loss  $L$  as a function of # parameters  $N$ , dataset size  $D$ , compute budget  $C$  (if other two are held constant)

$$L(N) = \left( \frac{N_c}{N} \right)^{\alpha_N}$$

$$L(D) = \left( \frac{D_c}{D} \right)^{\alpha_D}$$

$$L(C) = \left( \frac{C_c}{C} \right)^{\alpha_C}$$

Scaling laws can be used early in training to predict what the loss would be if we were to add more data or increase model size.



# Number of non-embedding parameters $N$

$$\begin{aligned} N &\approx 2 d n_{\text{layer}} (2 d_{\text{attn}} + d_{\text{ff}}) \\ &\approx 12 n_{\text{layer}} d^2 \\ &\quad (\text{assuming } d_{\text{attn}} = d_{\text{ff}}/4 = d) \end{aligned}$$

Thus GPT-3, with  $n = 96$  layers and dimensionality  $d = 12288$ , has  $12 \times 96 \times 12288^2 \approx 175$  billion parameters.

# KV Cache

In training, we can compute attention very efficiently in parallel:

$$\mathbf{A} = \text{softmax} \left( \frac{\mathbf{QK}^T}{\sqrt{d_k}} \right) \mathbf{V}$$

But not at inference! We generate the next tokens **one at a time!**

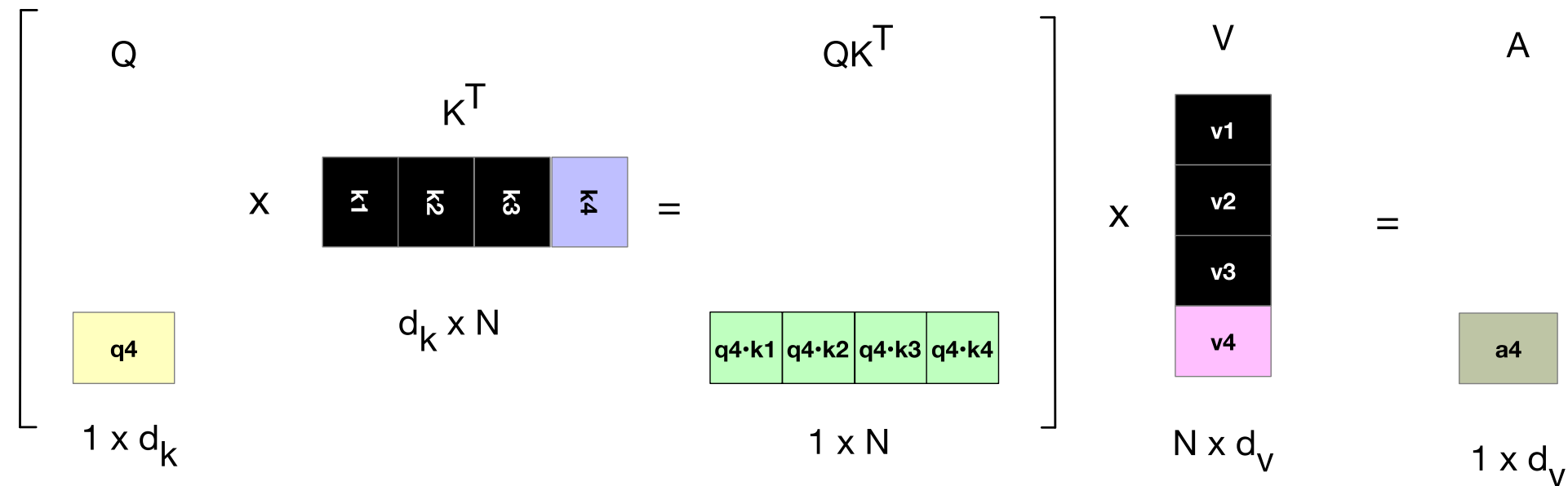
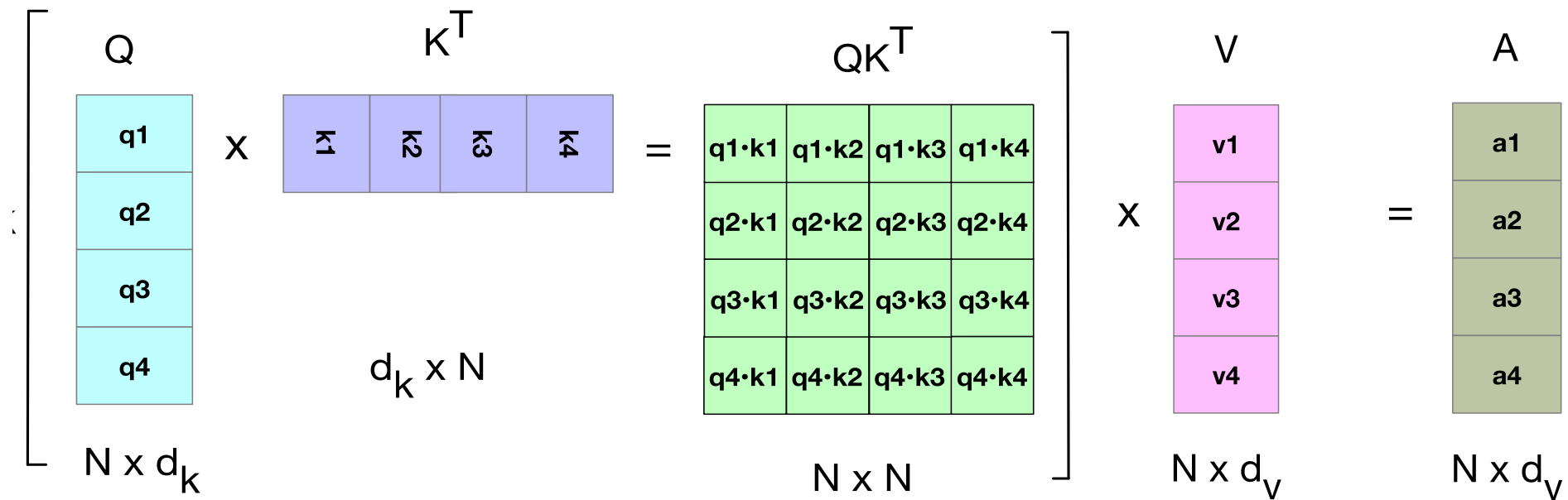
For a new token  $x$ , need to multiply by  $\mathbf{W}^Q$ ,  $\mathbf{W}^K$ , and  $\mathbf{W}^V$  to get query, key, values

# KV Cache

We don't want to **recompute** the key and value vectors for all the prior tokens  $x_{<i}$

Instead, store key and value vectors in memory in the KV cache, and then we can just grab them from the cache

# KV Cache



# Parameter-Efficient Finetuning

Adapting to a new domain by continued pretraining (finetuning) is a problem with huge LLMs.

- Enormous numbers of parameters to train
- Each pass of batch gradient descent has to backpropagate through many many huge layers.
- Expensive in processing power, in memory, and in time.

# Parameter-Efficient Finetuning

## GPU memory requirements

- Model for inference use approximately a number of GB that is 2 times the number of parameters
  - Normally 16bit floats are used for inference
  - 1MB/token is required in general
- An example: LLaMA-2 7B → up to 16GB including embeddings
  - 1GB is the typical size for an embedding model

# Parameter-Efficient Finetuning

## GPU memory requirements

- When fine-tuning LLaMA-2 7B we have to take into account the following (with batch size = 1)
- Model parameters (16bit floats):  $7B \times 2 = 14GB$
- Gradients (16bit floats):  $7B \times 2 = 14GB$
- Optimizer states (2 x parameter 32bit floats):  $7B \times 4 \times 2 = 56GB$

***TOTAL: 84GB***

# Parameter-Efficient Finetuning

Instead, **parameter-efficient fine tuning (PEFT)**

- Efficiently select a subset of parameters to update when finetuning.
- E.g., freeze some of the parameters (don't change them),
- And only update some a few parameters.



# LoRA (Low-Rank Adaptation)

- Transformers have many dense matrix multiply layers
  - Like  $\mathbf{W}^Q$ ,  $\mathbf{W}^K$ ,  $\mathbf{W}^V$ ,  $\mathbf{W}^O$  layers in attention
- Instead of updating these layers during finetuning,
  - Freeze these layers
  - Update a low-rank approximation with fewer parameters.

# LoRA (Low Rank Adaptation)

- Consider a **matrix  $\mathbf{W}$**  (shape  $[N \times d]$ ) that needs to be updated during finetuning via gradient descent.
- Normally updates are  $\Delta\mathbf{W}$  (shape  $[N \times d]$ ) that is  $\mathbf{W} \rightarrow \mathbf{W} + \Delta\mathbf{W}$
- In the forward pass after update, we do  $\mathbf{h} = \mathbf{x}\mathbf{W}$

# LoRA (Low Rank Adaptation)

- In LoRA, we freeze **W** and update instead a low-rank decomposition of **W**:
  - **A** of shape  $[N \times r]$
  - **B** of shape  $[r \times d]$ ,  $r \ll \min(N, d)$
  - During finetuning we update **A** and **B** instead of **W**
  - Replace **W** +  $\Delta\mathbf{W}$  with **W** + **AB**.

In the forward pass, instead of  $\mathbf{h} = \mathbf{xW}$  we do explicitly:

$$\mathbf{h} = \mathbf{xW} + \mathbf{xAB}$$

# LoRA (Low Rank Adaptation)

