Transformers

Introduction to Transformers

LLMs are built out of transformers

Transformer: a specific kind of network architecture, like a fancier feedforward network, but based on attention

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

A very approximate timeline

1990 Static Word Embeddings

2003 Neural Language Model

2008 Multi-Task Learning

2015 Attention

2017 Transformer

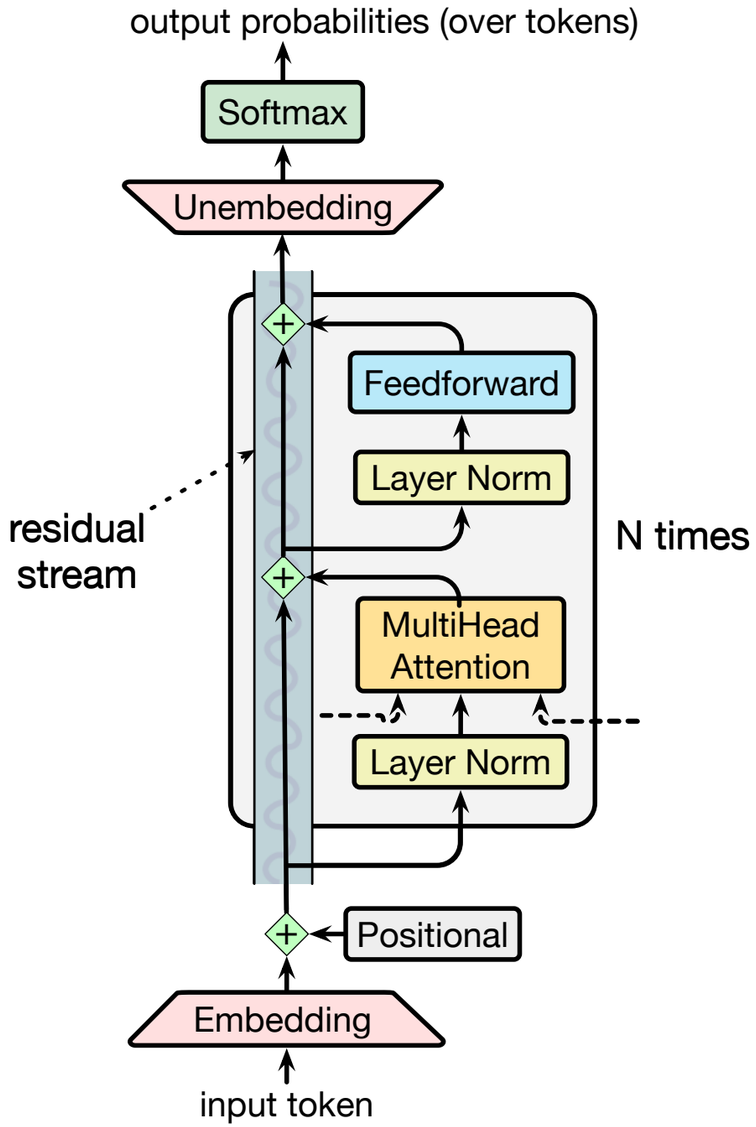
2018 Contextual Word Embeddings and Pretraining

2019 Prompting

2019 RLHF Alignment

2021 Instruction Tuning

The transformer



Transformers

Attention

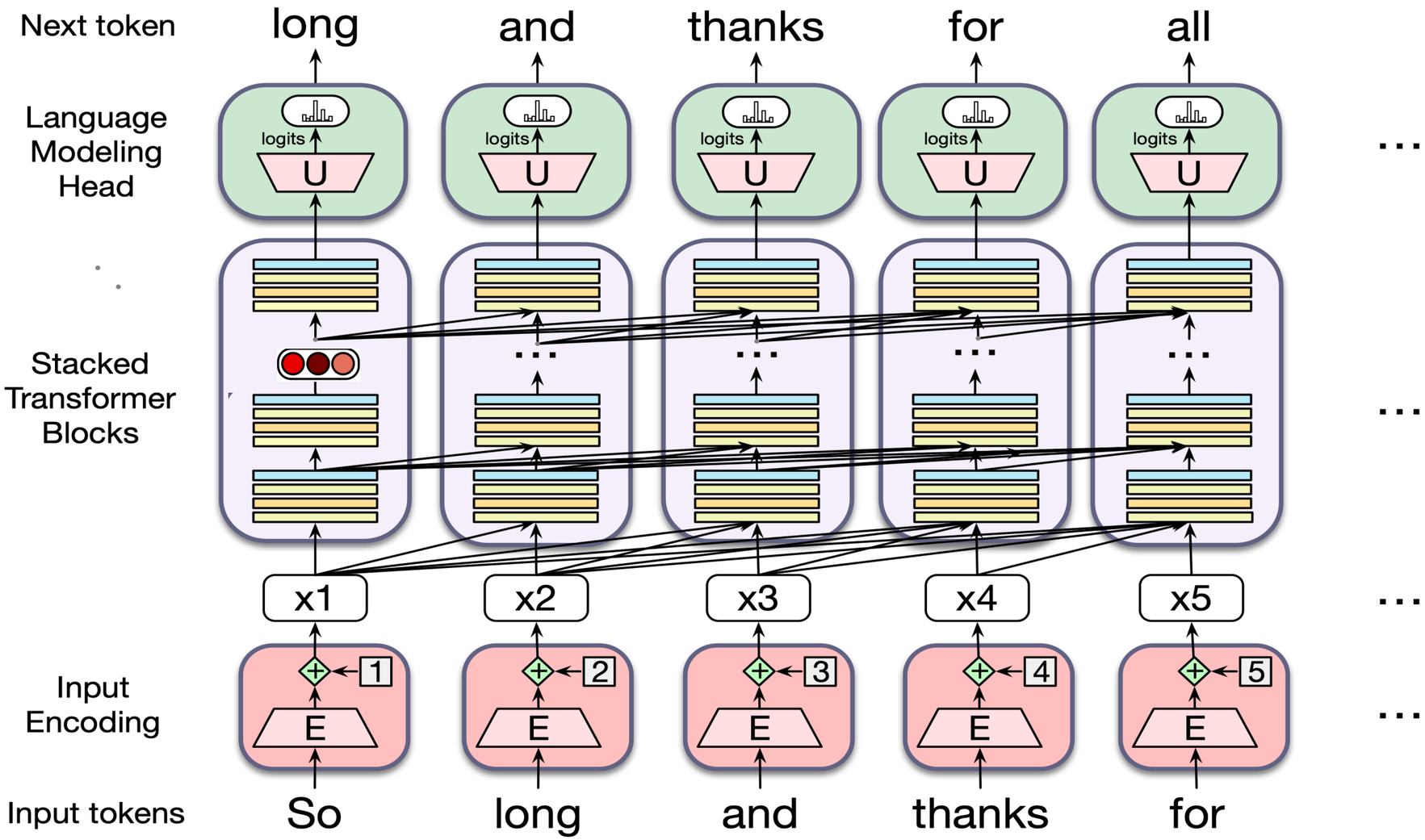
How to compute contextual embeddings

- **Contextual embeddings**: Each word has a different vector in each different context that represents different meanings depending on surrounding words
- How to compute contextual embeddings?

Attention

Reminder: contextual embeddings

Let's consider the embeddings for an individual word from a particular layer



Reminder: problem with static embeddings (word2vec)

They are static! The embedding for a word doesn't reflect how its meaning changes in context.

The chicken didn't cross the road because it was too tired

What is the meaning represented in the static embedding for "it"?

Contextual Embeddings

`The chicken didn't cross the road because it`

What should be the properties of "it"?

`The chicken didn't cross the road because it was too` **`tired`**
`The chicken didn't cross the road because it was too` **`wide`**

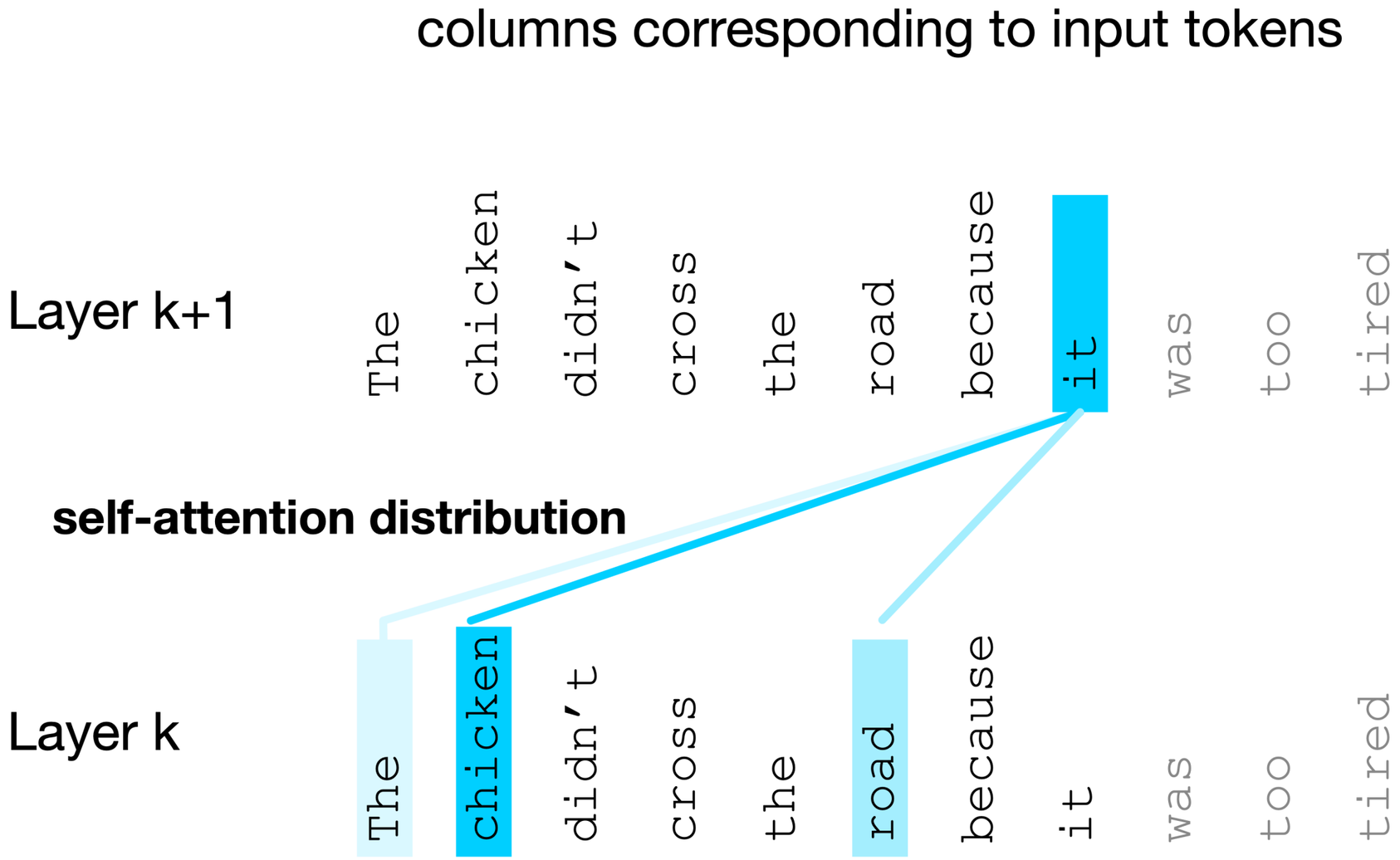
At this point in the sentence, it's probably referring to either the chicken or the street

Intuition of attention

Build up the contextual embedding from a word by selectively integrating information from all the neighboring words

We say that a word "attends to" some neighboring words more than others

Intuition of attention:

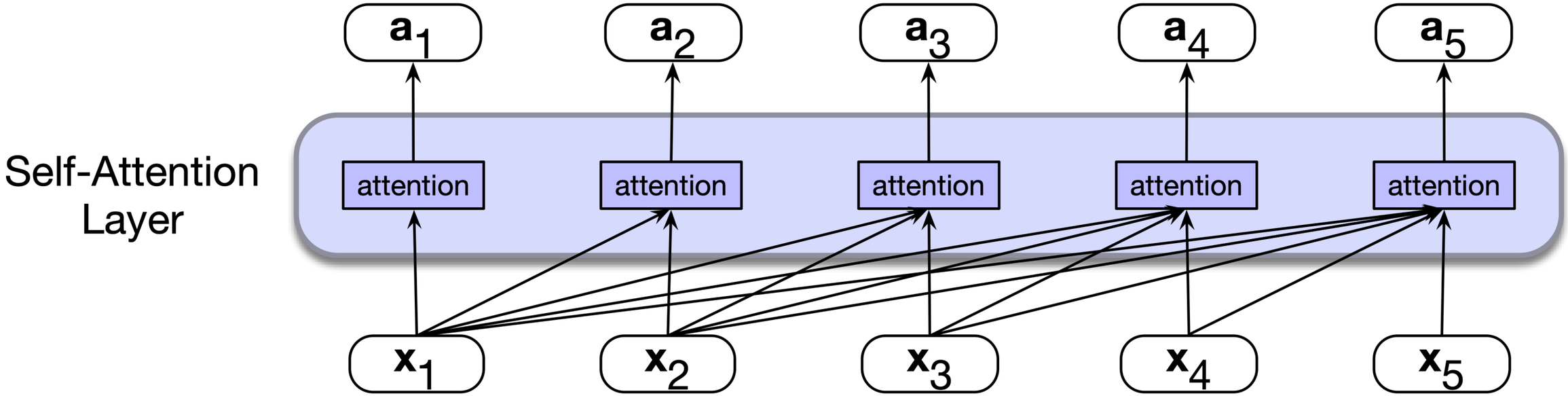


Attention definition

A mechanism for helping compute the embedding for a token by selectively attending to and integrating information from surrounding tokens (at the previous layer).

More formally: a method for doing a weighted sum of vectors.

Attention is left-to-right



Simplified version of attention: a sum of prior words weighted by their similarity with the current word

Given a sequence of token embeddings:

$$\mathbf{x}_1 \quad \mathbf{x}_2 \quad \mathbf{x}_3 \quad \mathbf{x}_4 \quad \mathbf{x}_5 \quad \mathbf{x}_6 \quad \mathbf{x}_7 \quad \mathbf{x}_i$$

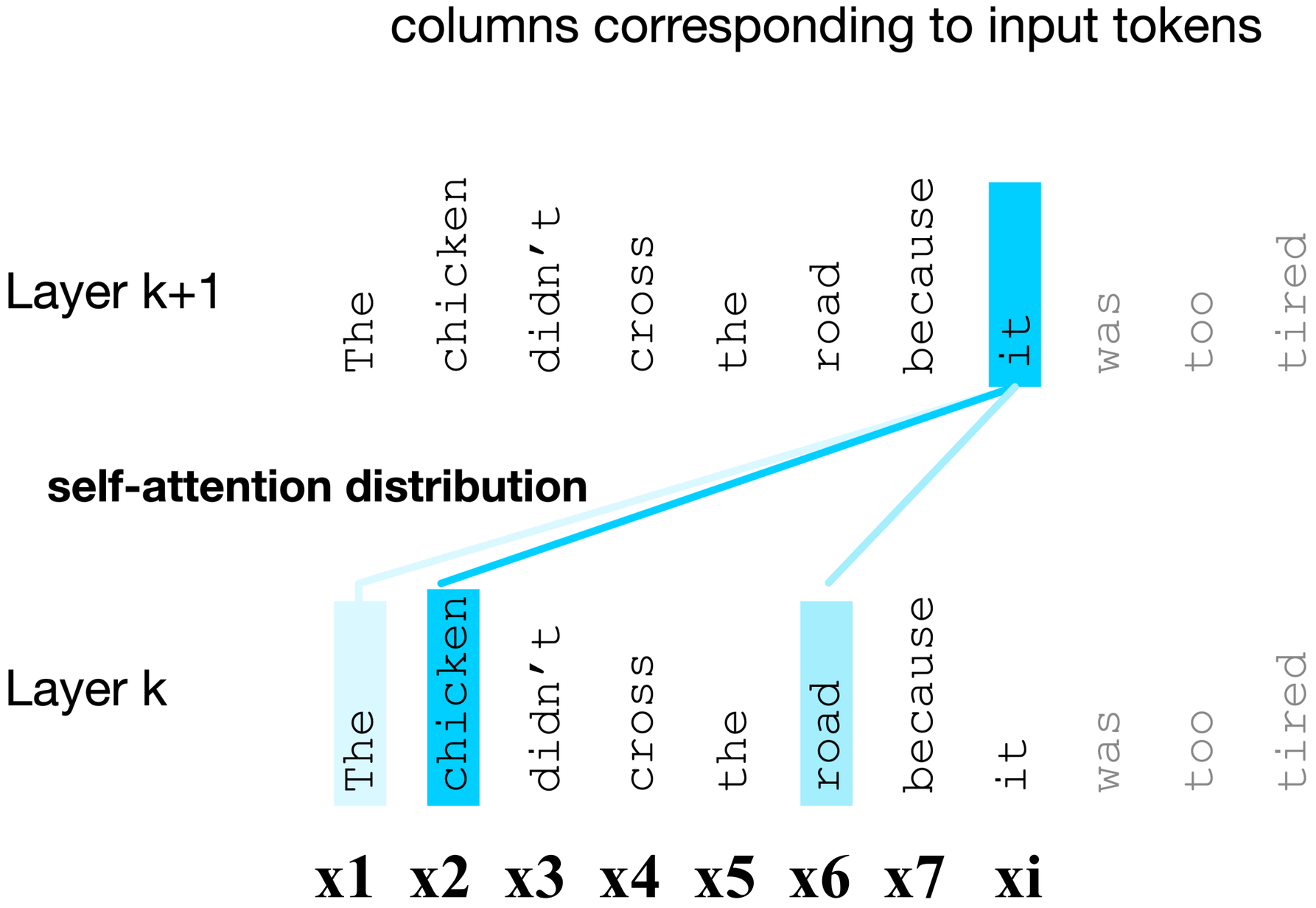
Produce: $\mathbf{a}_i$ = a weighted sum of $\mathbf{x}_1$ through $\mathbf{x}_7$ (and $\mathbf{x}_i$)
Weighted by their similarity to $\mathbf{x}_i$

$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i \cdot \mathbf{x}_j$$

$$\alpha_{ij} = \text{softmax}(\text{score}(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \leq i$$

$$\mathbf{a}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{x}_j$$

Intuition of attention:

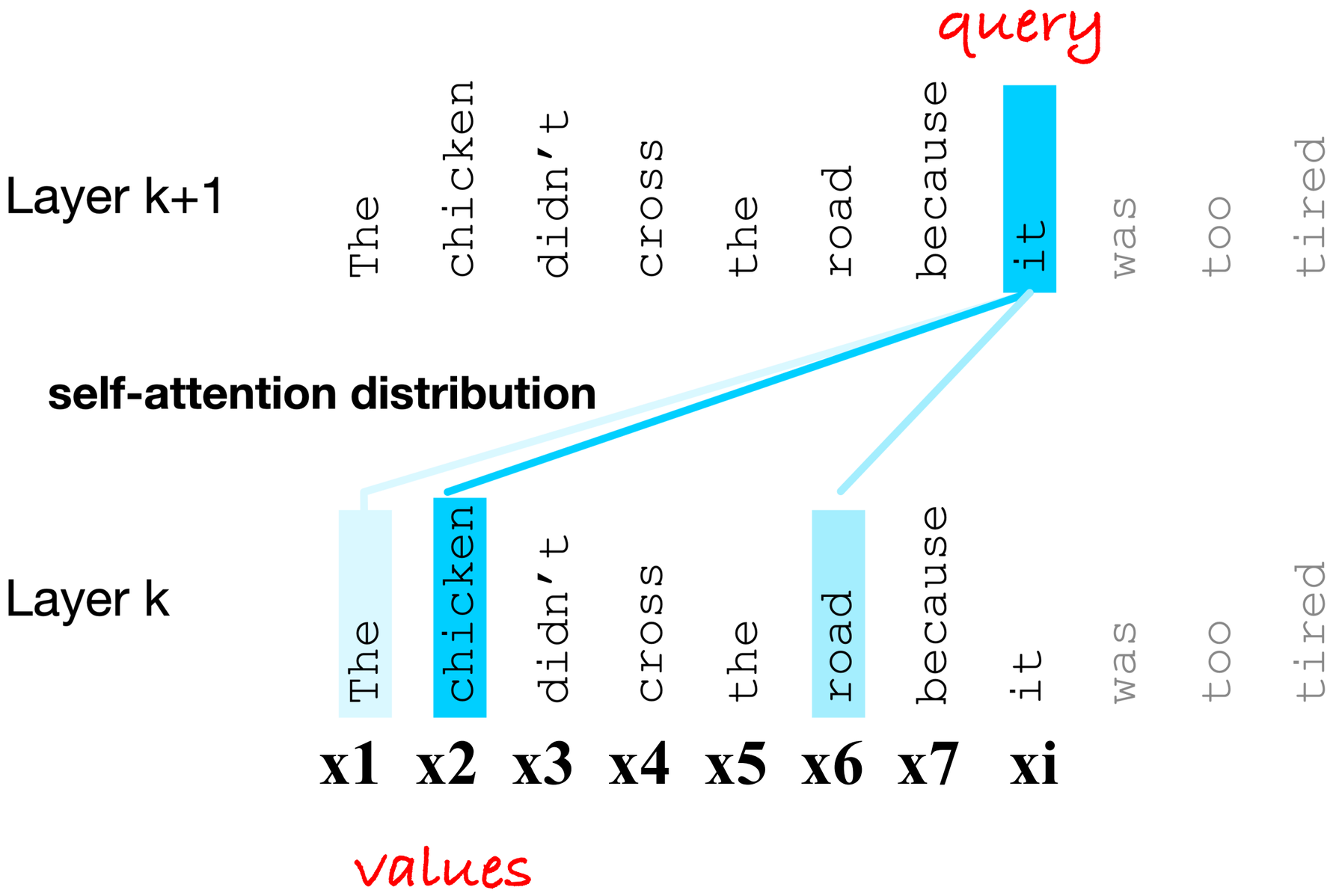


An Actual Attention Head: slightly more complicated

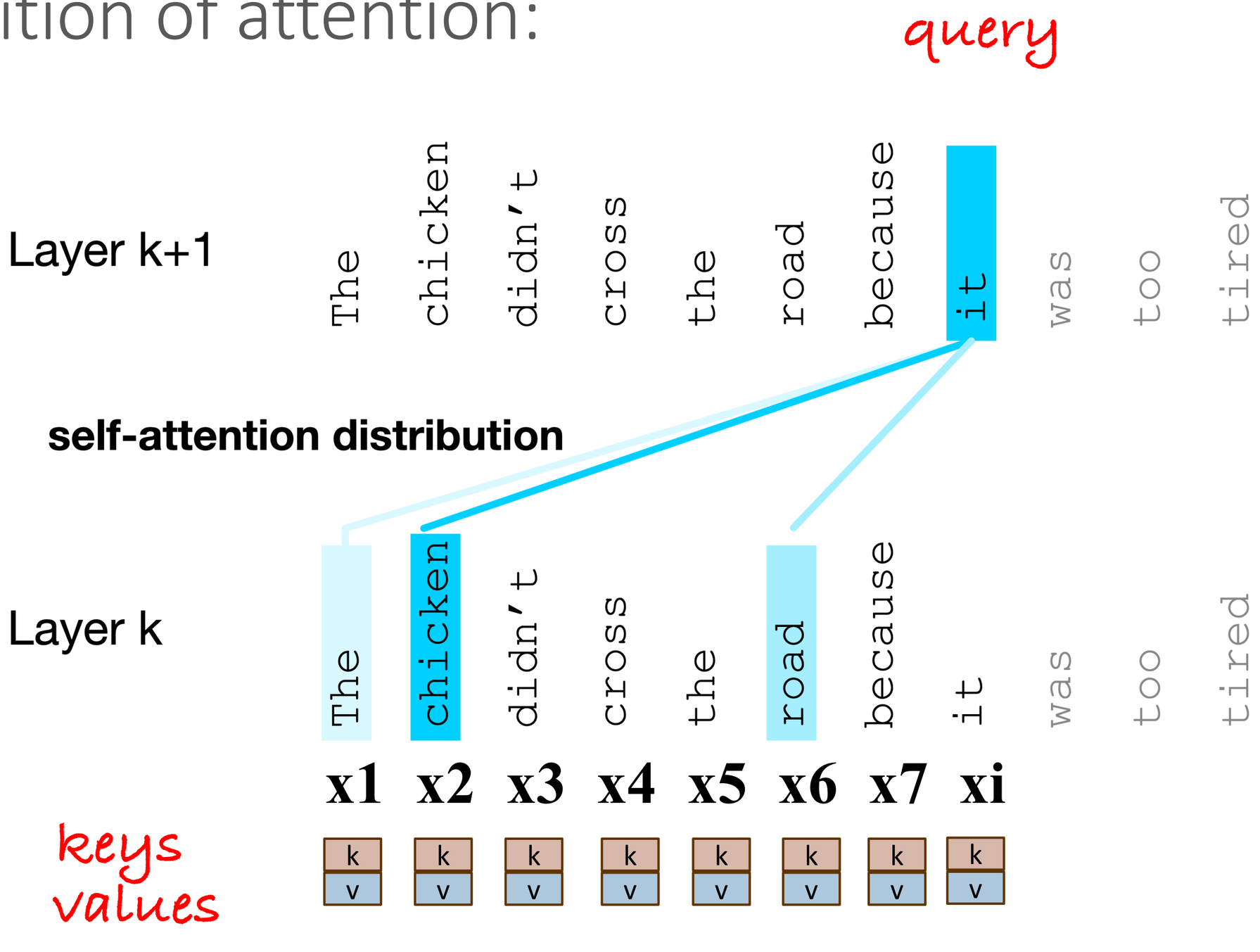
High-level idea: instead of using vectors (like x_i and x_4) directly, we'll represent 3 separate roles each vector $\mathbf{x}_i$ plays:

- **query**: As *the current element* being compared to the preceding inputs.
- **key**: as *a preceding input* that is being compared to the current element to determine a similarity
- **value**: a value of a preceding element that gets weighted and summed

Attention intuition



Intuition of attention:



An Actual Attention Head: slightly more complicated

We'll use matrices to project each vector $\mathbf{x}_i$ into a representation of its role as query, key, value:

- **query**: $\mathbf{W}^Q$
- **key**: $\mathbf{W}^K$
- **value**: $\mathbf{W}^V$

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^Q; \quad \mathbf{k}_i = \mathbf{x}_i\mathbf{W}^K; \quad \mathbf{v}_i = \mathbf{x}_i\mathbf{W}^V$$

An Actual Attention Head: slightly more complicated

Given these 3 representation of $\mathbf{x}_i$

$$\mathbf{q}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{Q}}; \quad \mathbf{k}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{K}}; \quad \mathbf{v}_i = \mathbf{x}_i\mathbf{W}^{\mathbf{V}}$$

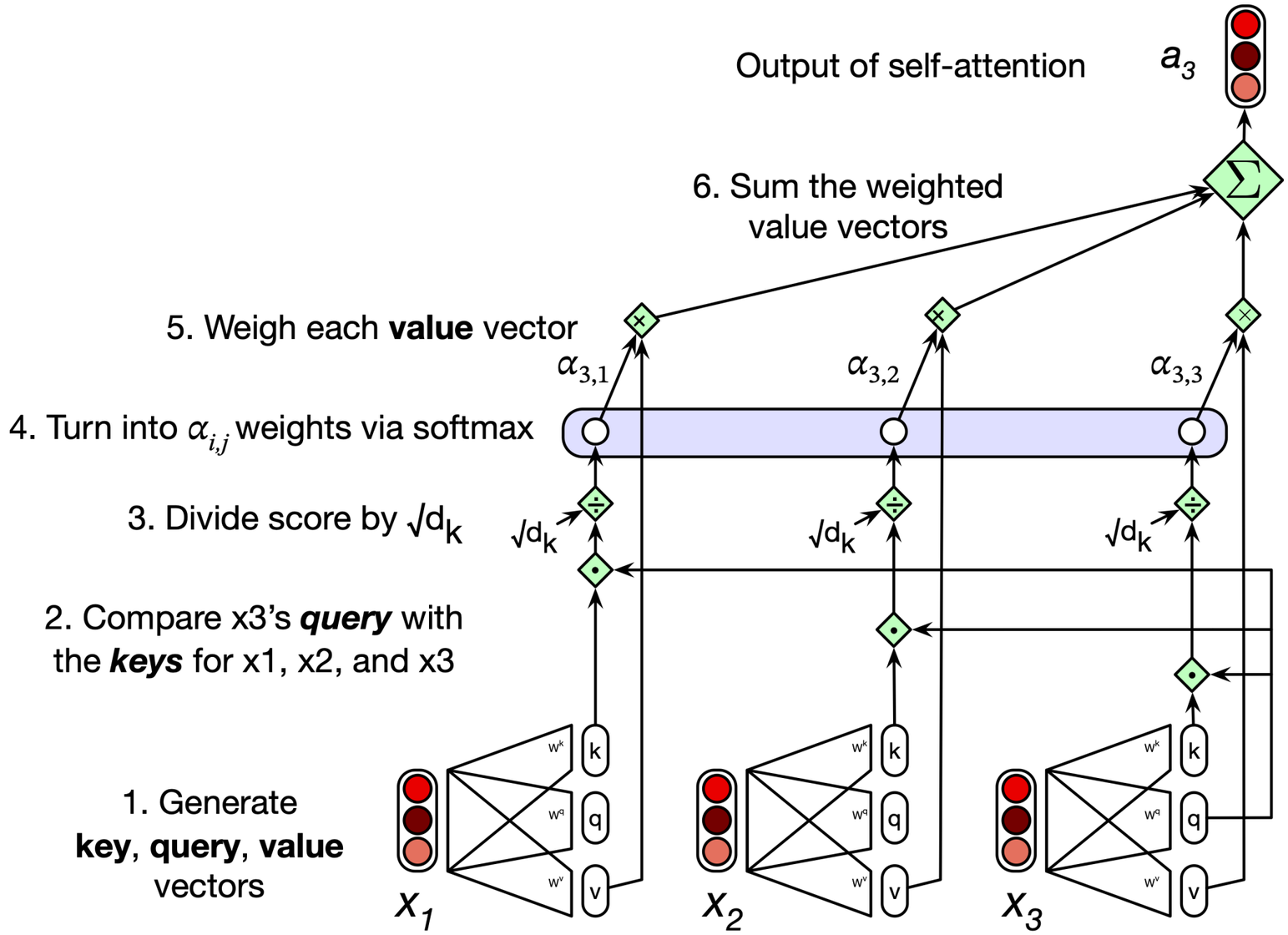
To compute similarity of current element $\mathbf{x}_i$ with some prior element $\mathbf{x}_j$

We'll use dot product between $\mathbf{q}_i$ and $\mathbf{k}_j$.

And instead of summing up $\mathbf{x}_j$, we'll sum up $\mathbf{v}_j$

Final equations for one attention head

Calculating the value of a3



Actual Attention: slightly more complicated

- Instead of one attention head, we'll have lots of them!
- Intuition: each head might be attending to the context for different purposes
  - Different linguistic relationships or patterns in the context

$$\mathbf{q}_i^c = \mathbf{x}_i\mathbf{W}^{\mathbf{Qc}};\quad \mathbf{k}_j^c = \mathbf{x}_j\mathbf{W}^{\mathbf{Kc}};\quad \mathbf{v}_j^c = \mathbf{x}_j\mathbf{W}^{\mathbf{Vc}};\quad \forall\, c \quad 1 \le c \le h$$

$$\text{score}^c(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i^c \cdot \mathbf{k}_j^c}{\sqrt{d_k}}$$

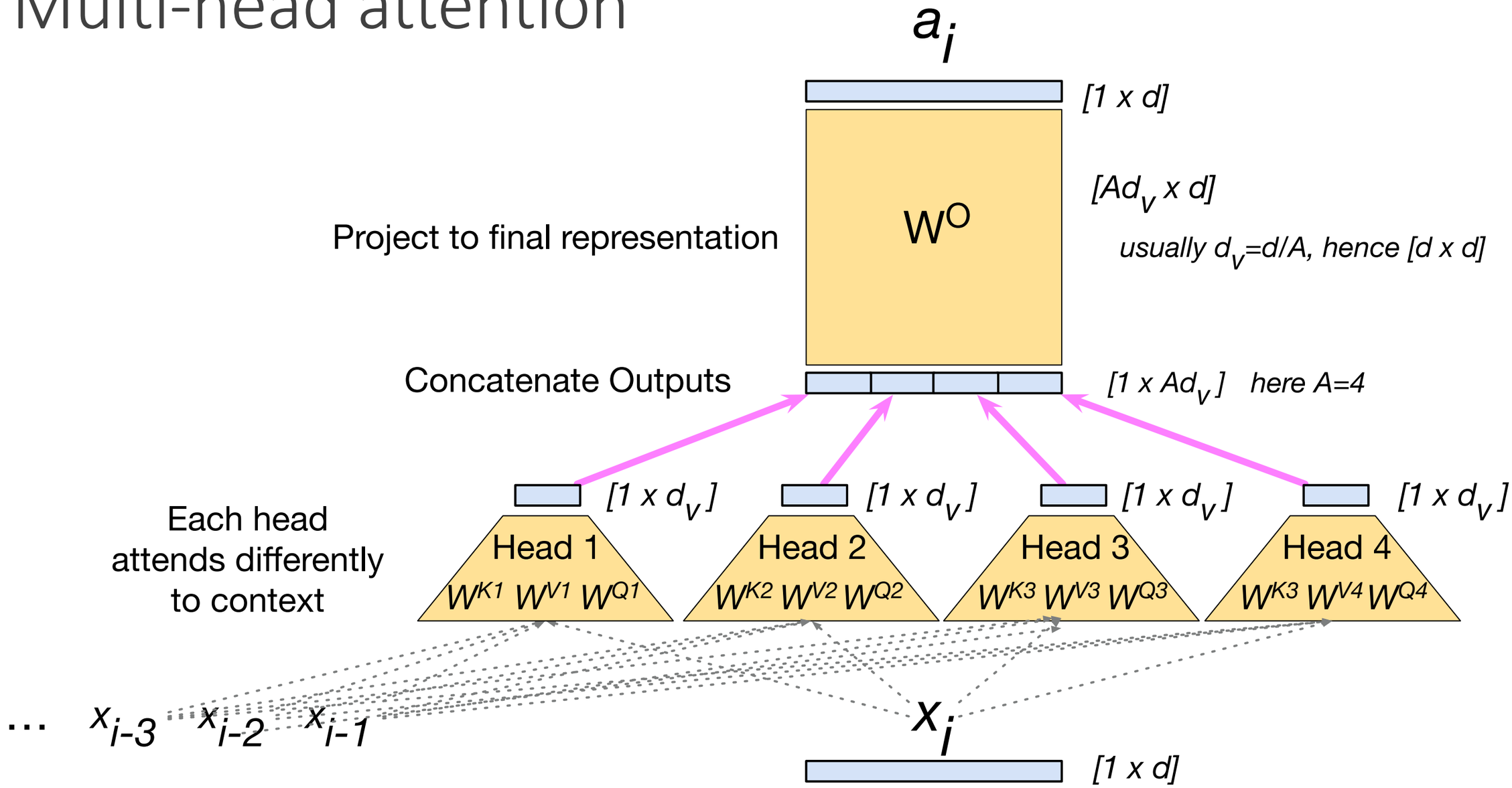
$$\alpha_{ij}^c = \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \le i$$

$$\mathbf{head}_i^c = \sum_{j \le i} \alpha_{ij}^c \mathbf{v}_j^c$$

$$\mathbf{a}_i = (\mathbf{head}^1 \oplus \mathbf{head}^2 \ldots \oplus \mathbf{head}^h)\mathbf{W}^O$$

$$\text{MultiHeadAttention}(\mathbf{x}_i, [\mathbf{x}_1, \cdots, \mathbf{x}_N]) = \mathbf{a}_i$$

Multi-head attention



Summary

Attention is a method for enriching the representation of a token by incorporating contextual information

The result: the embedding for each word will be different in different contexts!

Contextual embeddings: a representation of word meaning in its context.

We'll see in the next lecture that attention can also be viewed as a way to move information from one token to another.

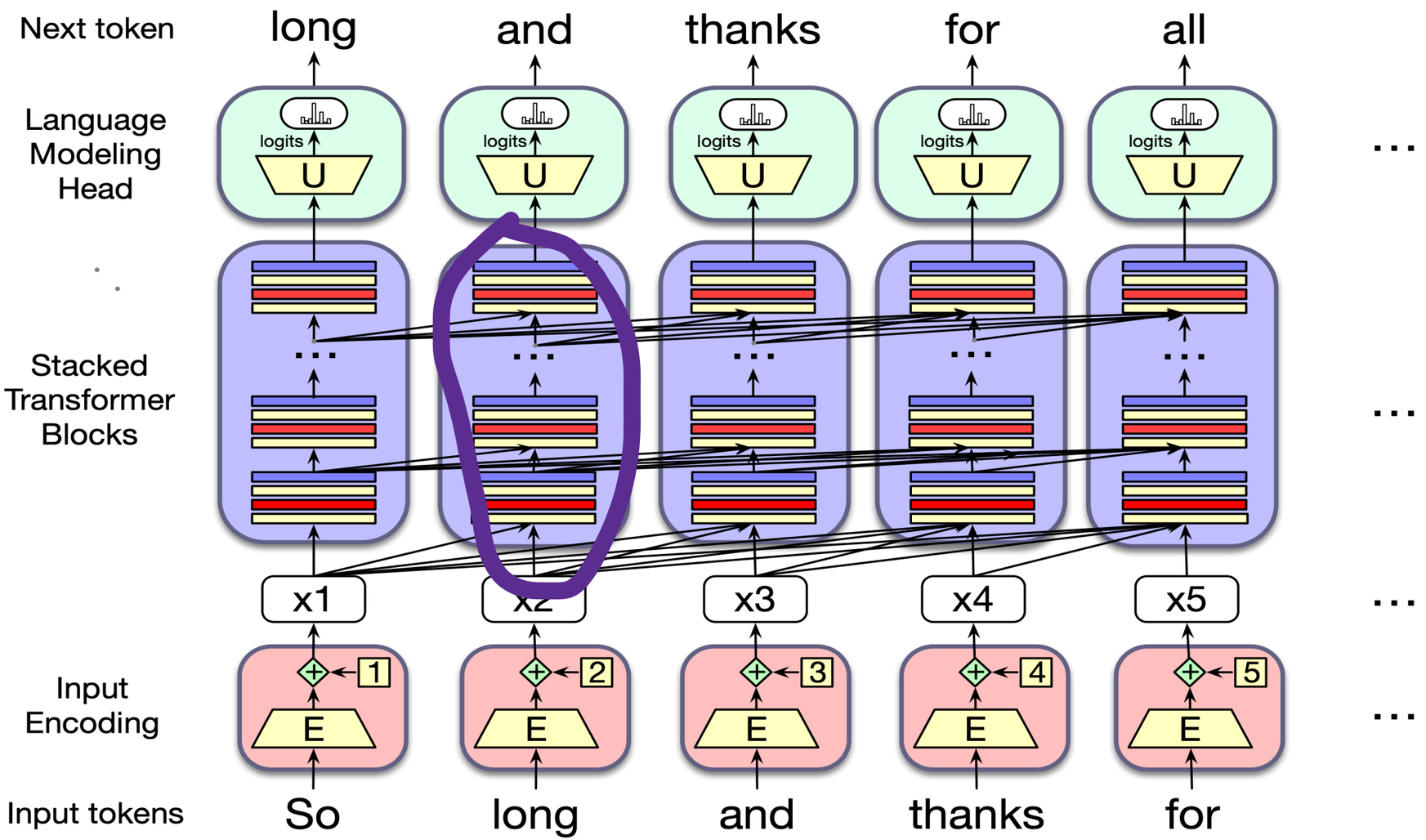
Transformers

Attention

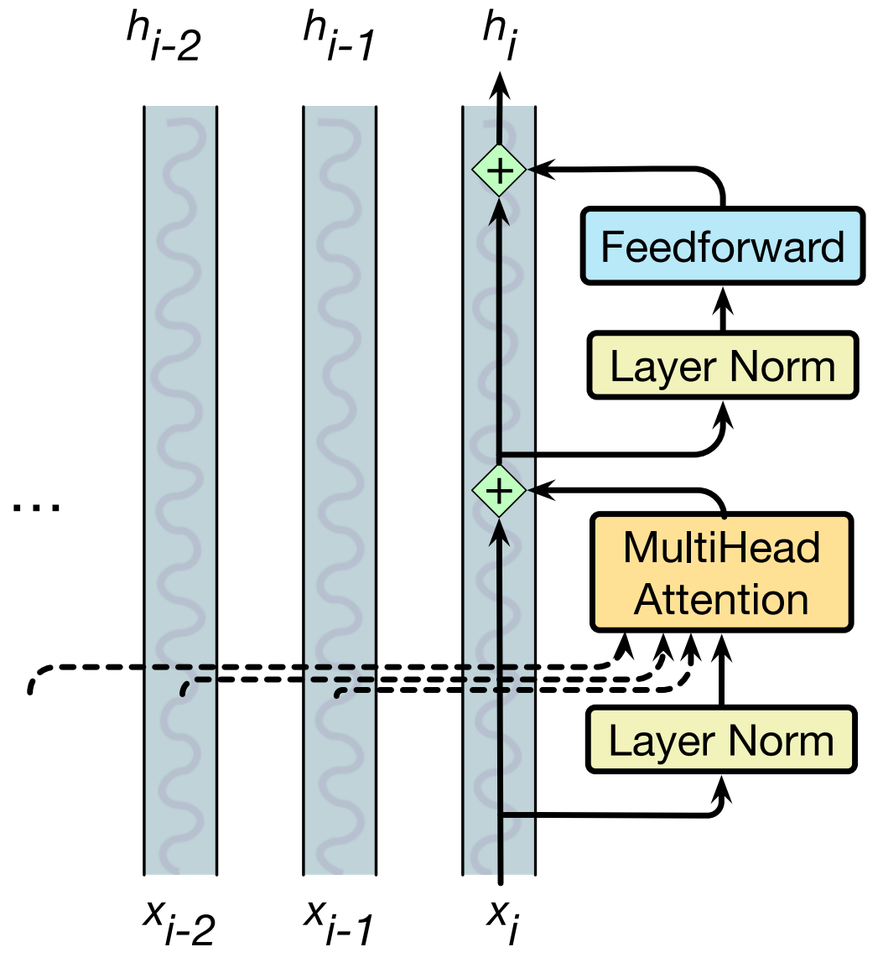
Transformers

The Transformer Block

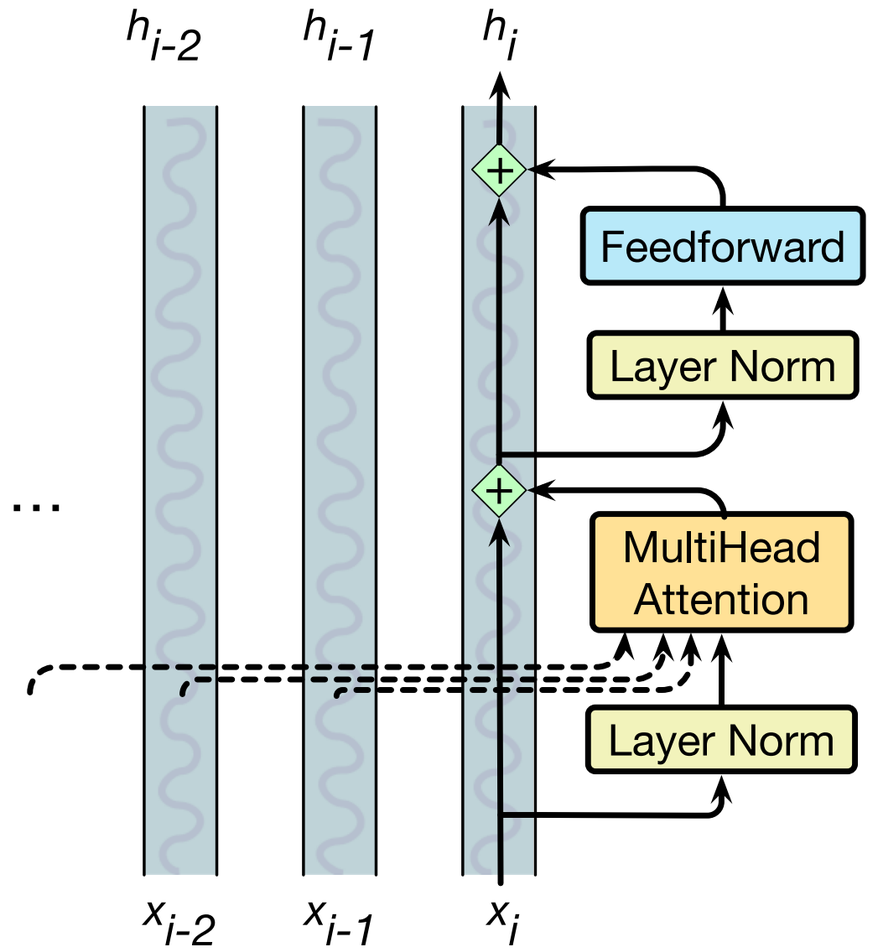
Reminder: transformer language model



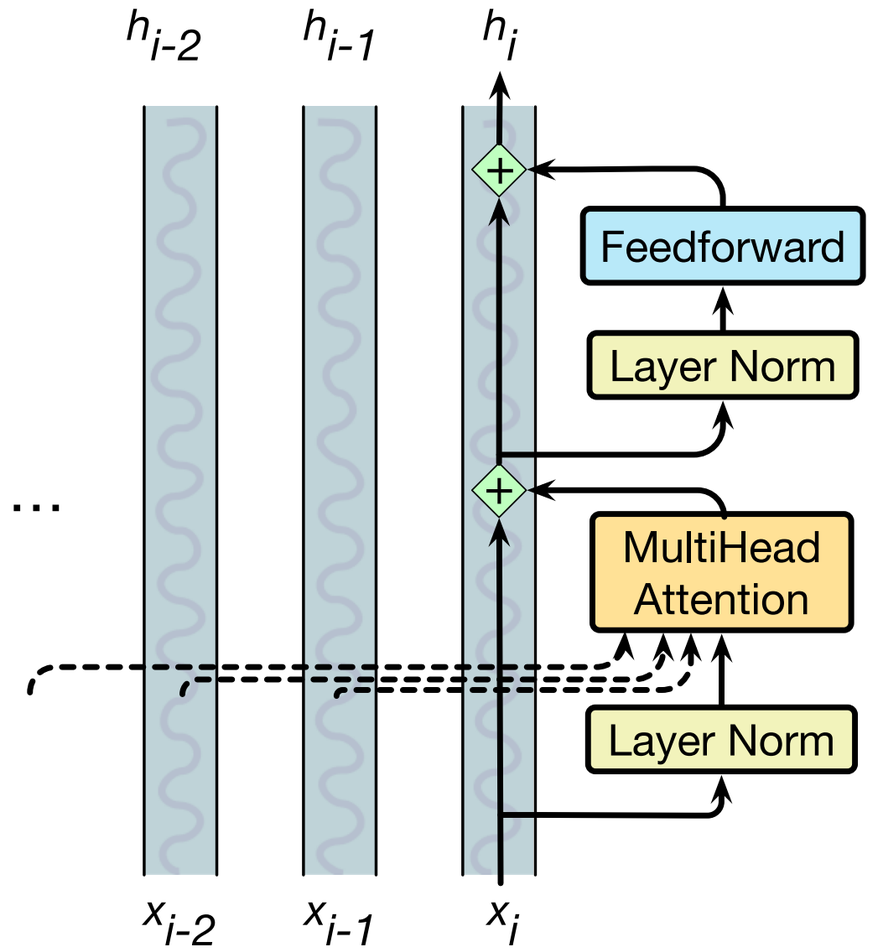
The residual stream: each token gets passed up and modified



We'll need nonlinearities, so a feedforward layer



Layer norm: the vector $\mathbf{x}_i$ is normalized twice



Layer Norm

Layer norm is a variation of the z-score from statistics, applied to a single vec- tor in a hidden layer

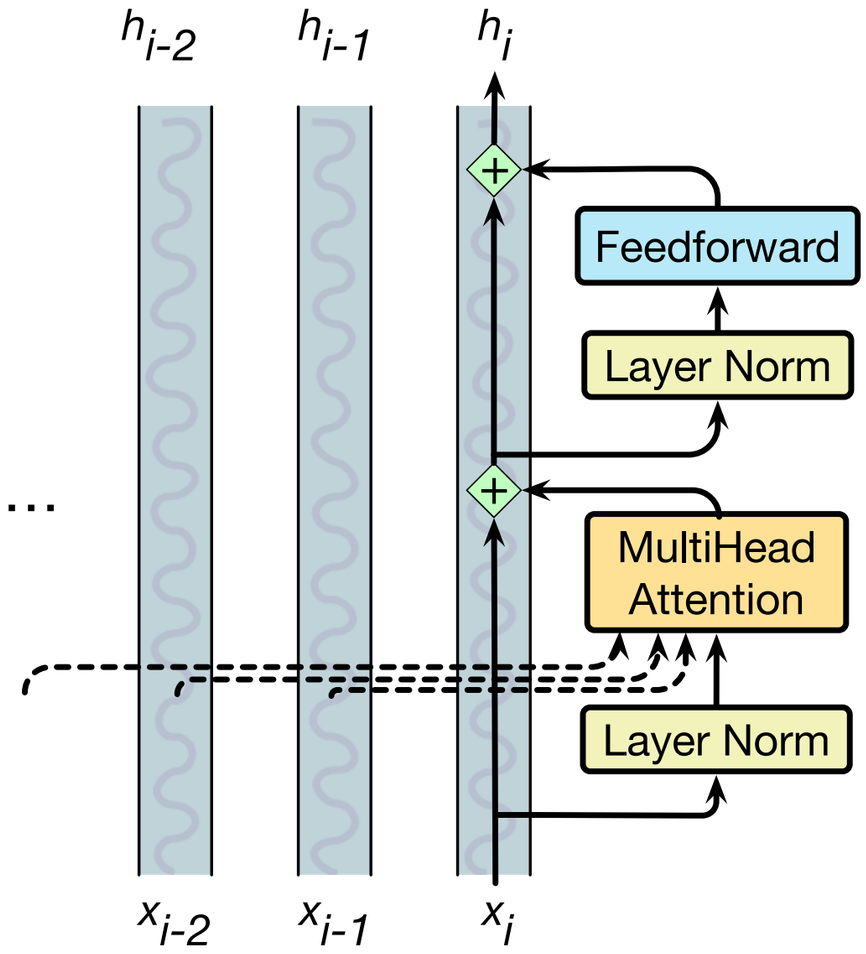
$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

$$\sigma = \sqrt{\frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2}$$

$$\hat{\mathbf{x}} = \frac{(\mathbf{x} - \mu)}{\sigma}$$

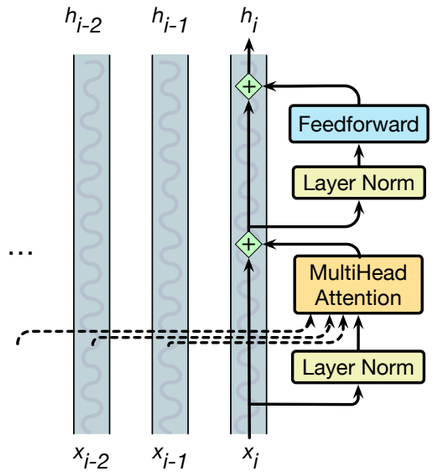
$$\text{LayerNorm}(\mathbf{x}) = \gamma \frac{(\mathbf{x} - \mu)}{\sigma} + \beta$$

Putting together a single transformer block

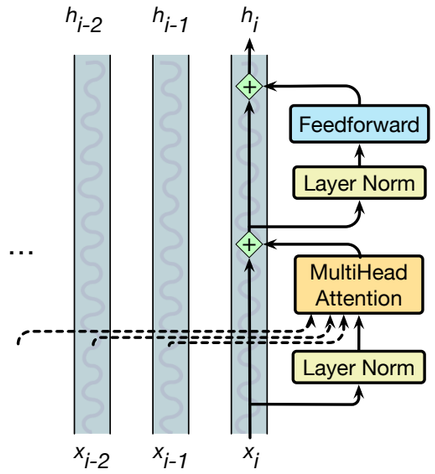


A transformer is a stack of these blocks
so all the vectors are of the same dimensionality d

Block 2



Block 1

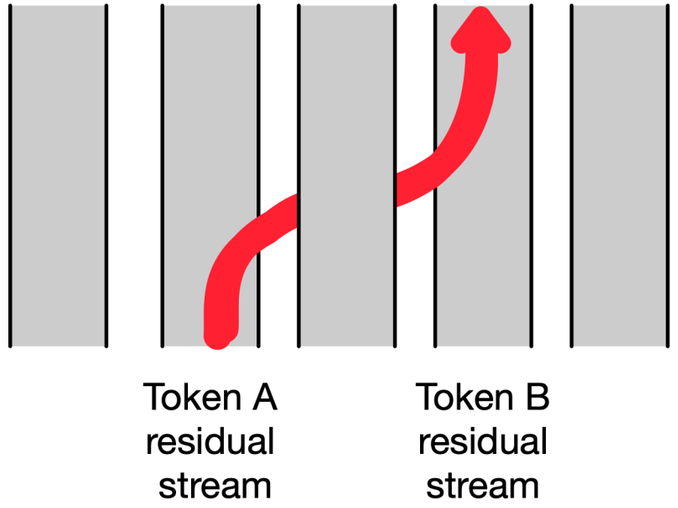


Residual streams and attention

Notice that all parts of the transformer block apply to 1 residual stream (1 token).

Except attention, which takes information from other tokens

Elhage et al. (2021) show that we can view attention heads as literally moving information from the residual stream of a neighboring token into the current stream .



Transformers

The Transformer Block

Transformers

Parallelizing Attention Computation

Parallelizing computation using X

For attention/transformer block we've been computing a **single** output at a **single** time step *i* in a **single** residual stream.

But we can pack the *N* tokens of the input sequence into a single matrix **X** of size [$N \times d$].

Each row of X is the embedding of one token of the input.

X can have 1K-32K rows, each of the dimensionality of the embedding *d* (the **model dimension**)

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^{\mathbf{Q}};\ \ \mathbf{K} = \mathbf{X}\mathbf{W}^{\mathbf{K}};\ \ \mathbf{V} = \mathbf{X}\mathbf{W}^{\mathbf{V}}$$

QK^T

Now can do a single matrix multiply to combine Q and K^T

N

q1·k1	**q1·k2**	**q1·k3**	**q1·k4**
q2·k1	**q2·k2**	**q2·k3**	**q2·k4**
q3·k1	**q3·k2**	**q3·k3**	**q3·k4**
q4·k1	**q4·k2**	**q4·k3**	**q4·k4**

N

Parallelizing attention

- Scale the scores, take the softmax, and then multiply the result by V resulting in a matrix of shape $N \times d$
  - An attention vector for each input token

$$\mathbf{A} = \text{softmax}\left(\text{mask}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_k}}\right)\right)\mathbf{V}$$

Masking out the future

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

- What is this mask function?
  QK^T has a score for each query dot every key, *including those that follow the query*.
- Guessing the next word is pretty simple if you already know it!

Masking out the future

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

Add $-\infty$ to cells in upper triangle

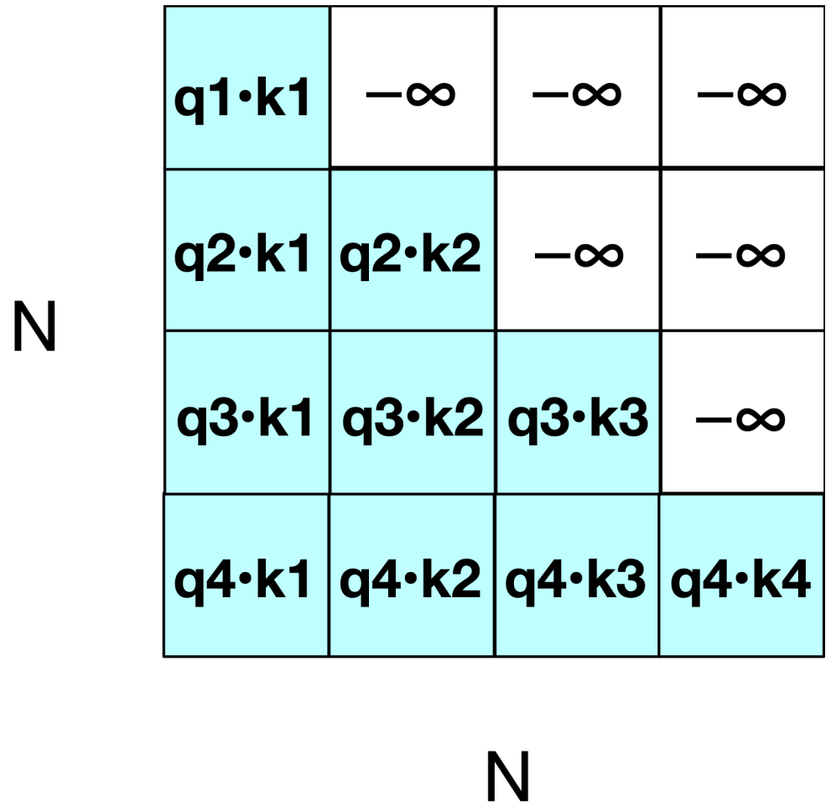
The softmax will turn it to 0

q1·k1	−∞	−∞	−∞
q2·k1	q2·k2	−∞	−∞
q3·k1	q3·k2	q3·k3	−∞
q4·k1	q4·k2	q4·k3	q4·k4

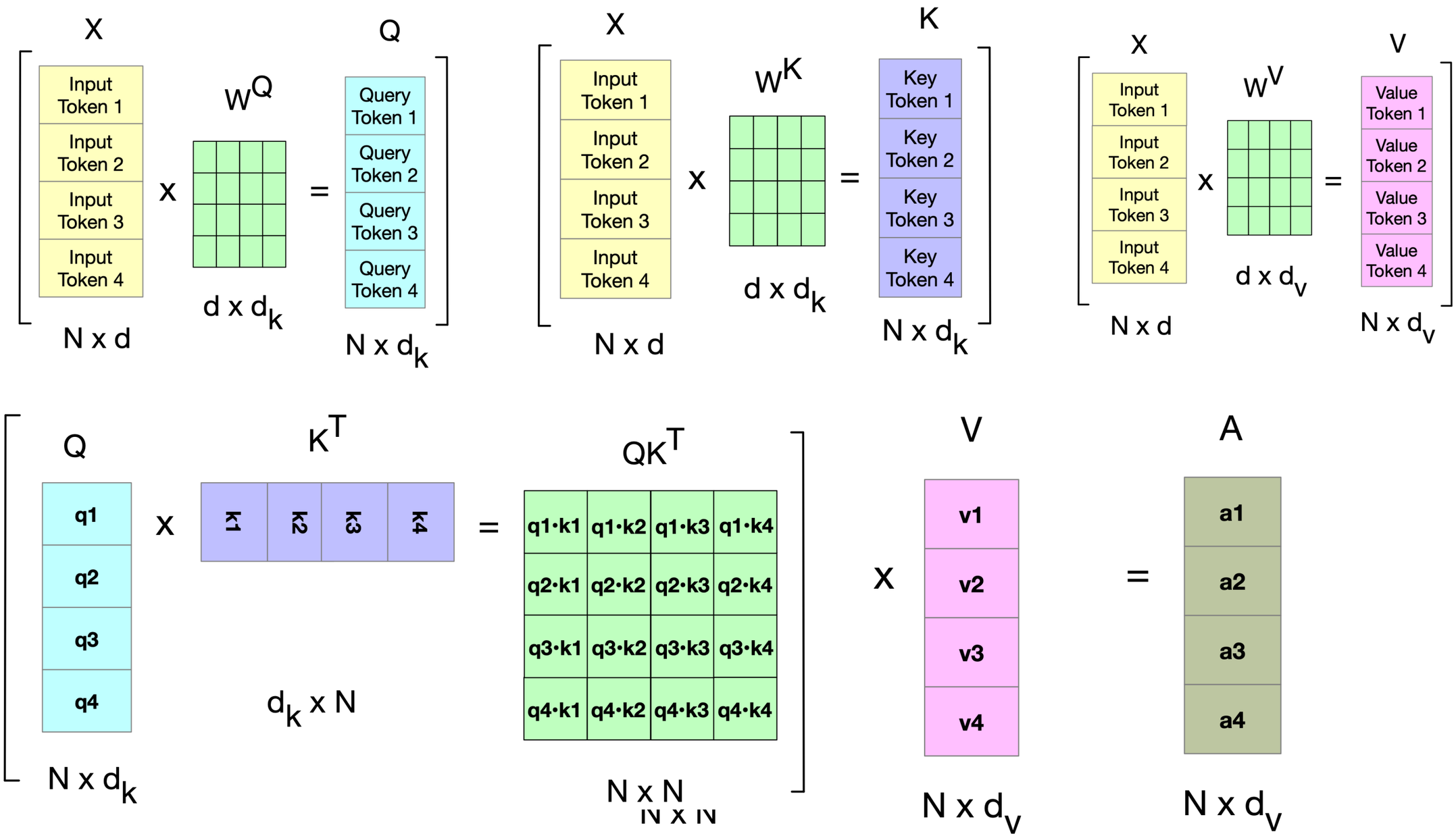
N

N

Another point: Attention is quadratic in length



Attention again



Parallelizing Multi-head Attention

$$\mathbf{Q^i} = \mathbf{X}\mathbf{W^{Qi}}\,;\quad \mathbf{K^i} = \mathbf{X}\mathbf{W^{Ki}}\,;\quad \mathbf{V^i} = \mathbf{X}\mathbf{W^{Vi}}$$

$$\mathbf{head}_i = \mathrm{SelfAttention}(\mathbf{Q^i}, \mathbf{K^i}, \mathbf{V^i}) = \mathrm{softmax}\left(\frac{\mathbf{Q^i}\mathbf{K^{i\,T}}}{\sqrt{d_k}}\right)\mathbf{V^i}$$

$$\mathrm{MultiHeadAttention}(\mathbf{X}) = (\mathbf{head}_1 \oplus \mathbf{head}_2 \ldots \oplus \mathbf{head}_h)\mathbf{W^O}$$

Parallelizing Multi-head Attention

$$\mathbf{O} = \text{LayerNorm}(\mathbf{X} + \text{MultiHeadAttention}(\mathbf{X}))$$
$$\mathbf{H} = \text{LayerNorm}(\mathbf{O} + \text{FFN}(\mathbf{O}))$$

or

$$\mathbf{T}^1 = \text{MultiHeadAttention}(\mathbf{X})$$
$$\mathbf{T}^2 = \mathbf{X} + \mathbf{T}^1$$
$$\mathbf{T}^3 = \text{LayerNorm}(\mathbf{T}^2)$$
$$\mathbf{T}^4 = \text{FFN}(\mathbf{T}^3)$$
$$\mathbf{T}^5 = \mathbf{T}^4 + \mathbf{T}^3$$
$$\mathbf{H} = \text{LayerNorm}(\mathbf{T}^5)$$

Transformers

Parallelizing Attention Computation

Transformers

Input and output: Position embeddings and the Language Model Head

Token and Position Embeddings

The matrix X (of shape $[N \times d]$) has an embedding for each word in the context.

This embedding is created by adding two distinct embedding for each input

- token embedding
- positional embedding

Token Embeddings

Embedding matrix E has shape [$|V| \times d$].

- One row for each of the $|V|$ tokens in the vocabulary.
- Each word is a row vector of d dimensions

Given: string "*Thanks for all the*"

1. Tokenize with BPE and convert into vocab indices

w = [5,4000,10532,2224]

2. Select the corresponding rows from E, each row an embedding

- (row 5, row 4000, row 10532, row 2224).

Position Embeddings

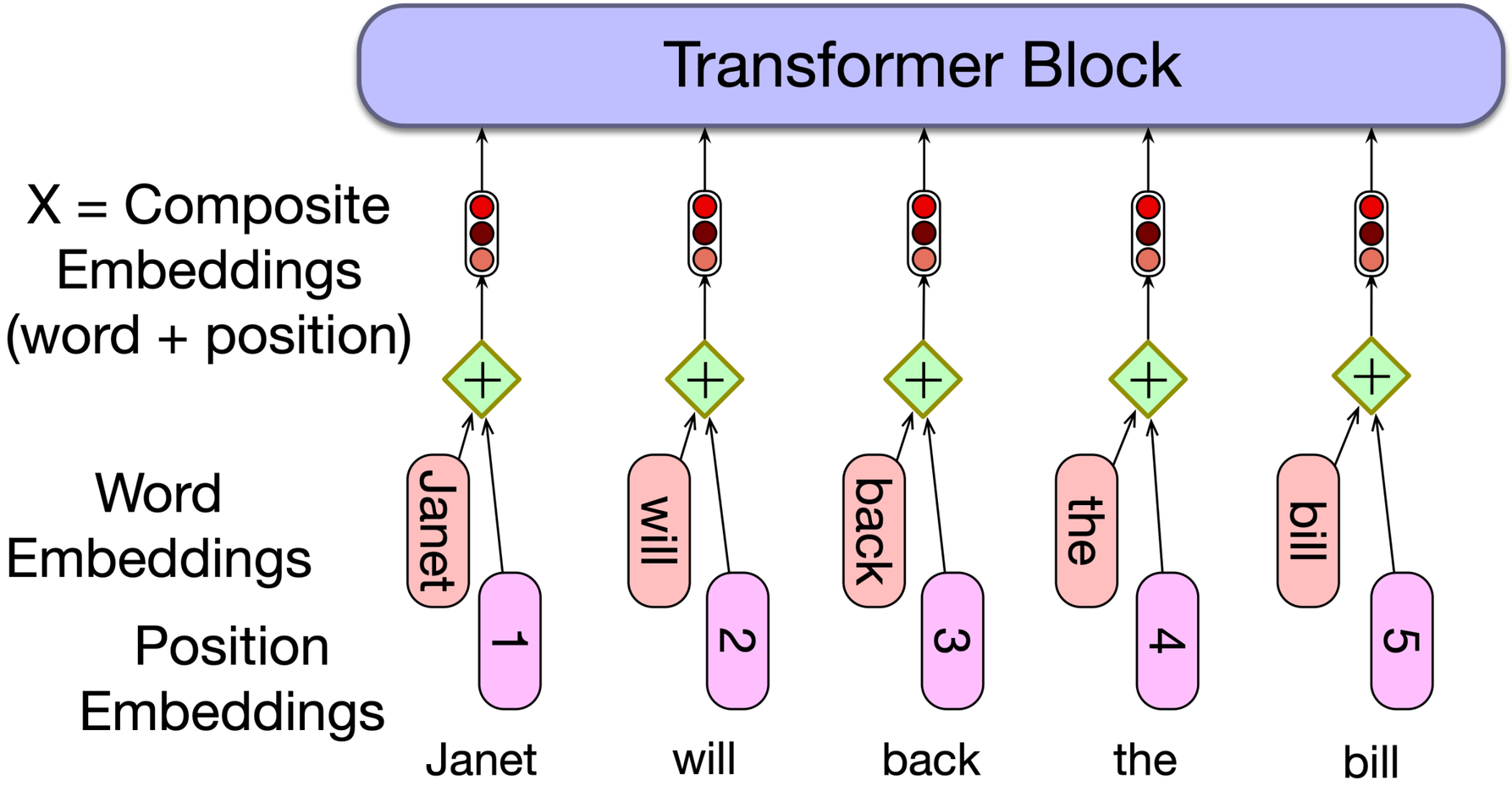
There are many methods, but we'll just describe the simplest: absolute position.

Goal: learn a position embedding matrix Epos of shape $[1 \times N]$.

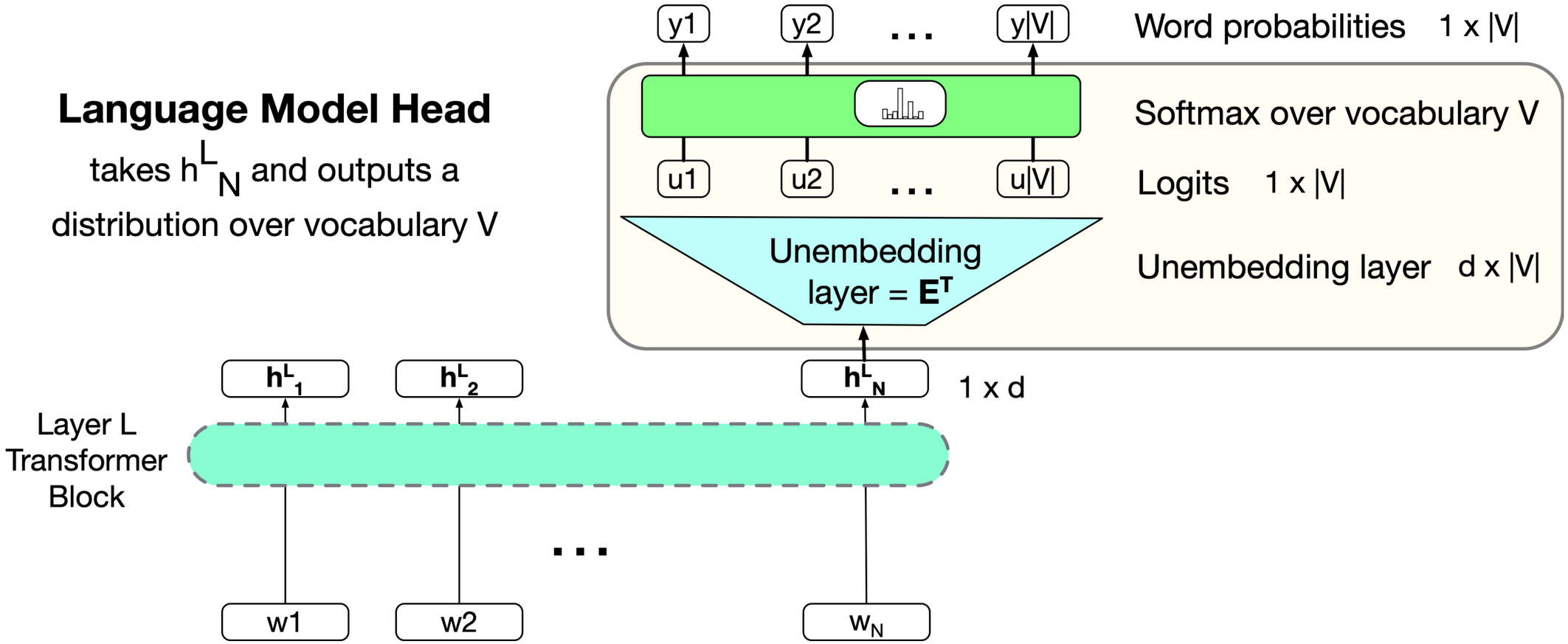
Start with randomly initialized embeddings

- one for each integer up to some maximum length.
- i.e., just as we have an embedding for token *fish*, we'll have an embedding for position 3 and position 17.
- As with word embeddings, these position embeddings are learned along with other parameters during training.

Each **x** is just the sum of word and position embeddings

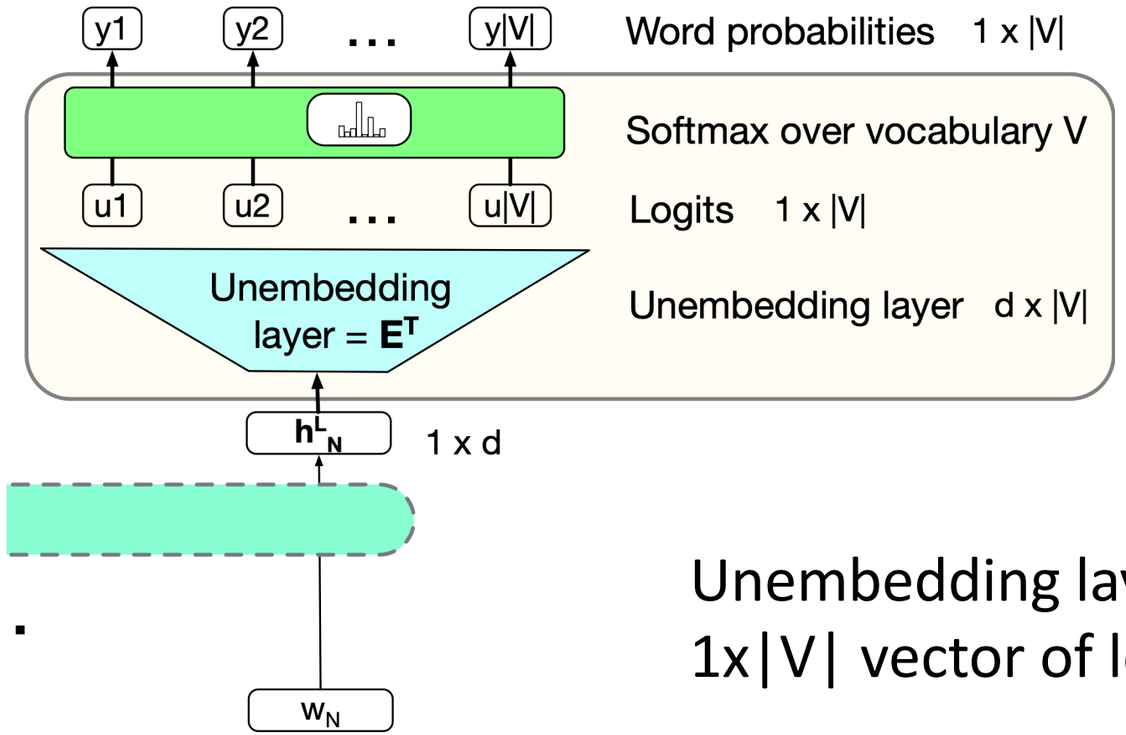


Language modeling head



Language modeling head

Unembedding layer: linear layer projects from h^L_N (shape $[1 \times d]$) to logit vector

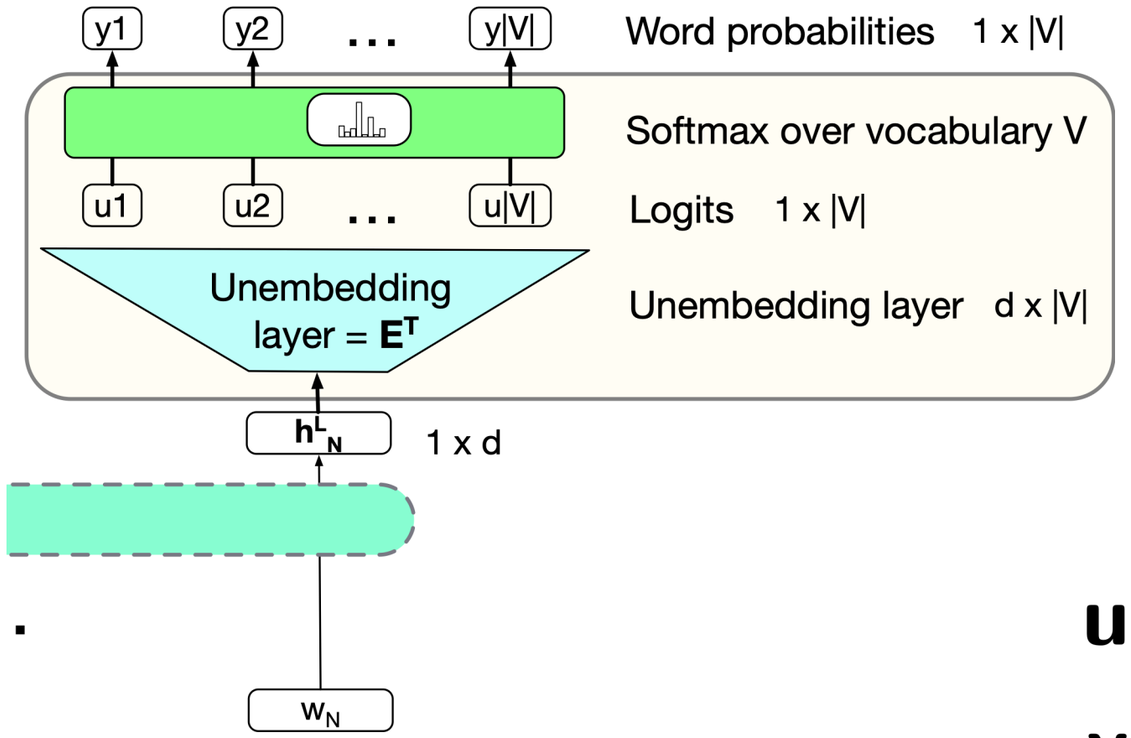


Why "unembedding"? **Tied** to $\mathbf{E}^T$

Weight tying, we use the same weights for two different matrices

Unembedding layer maps from an embedding to a 1x|V| vector of logits

Language modeling head



Logits, the score vector u

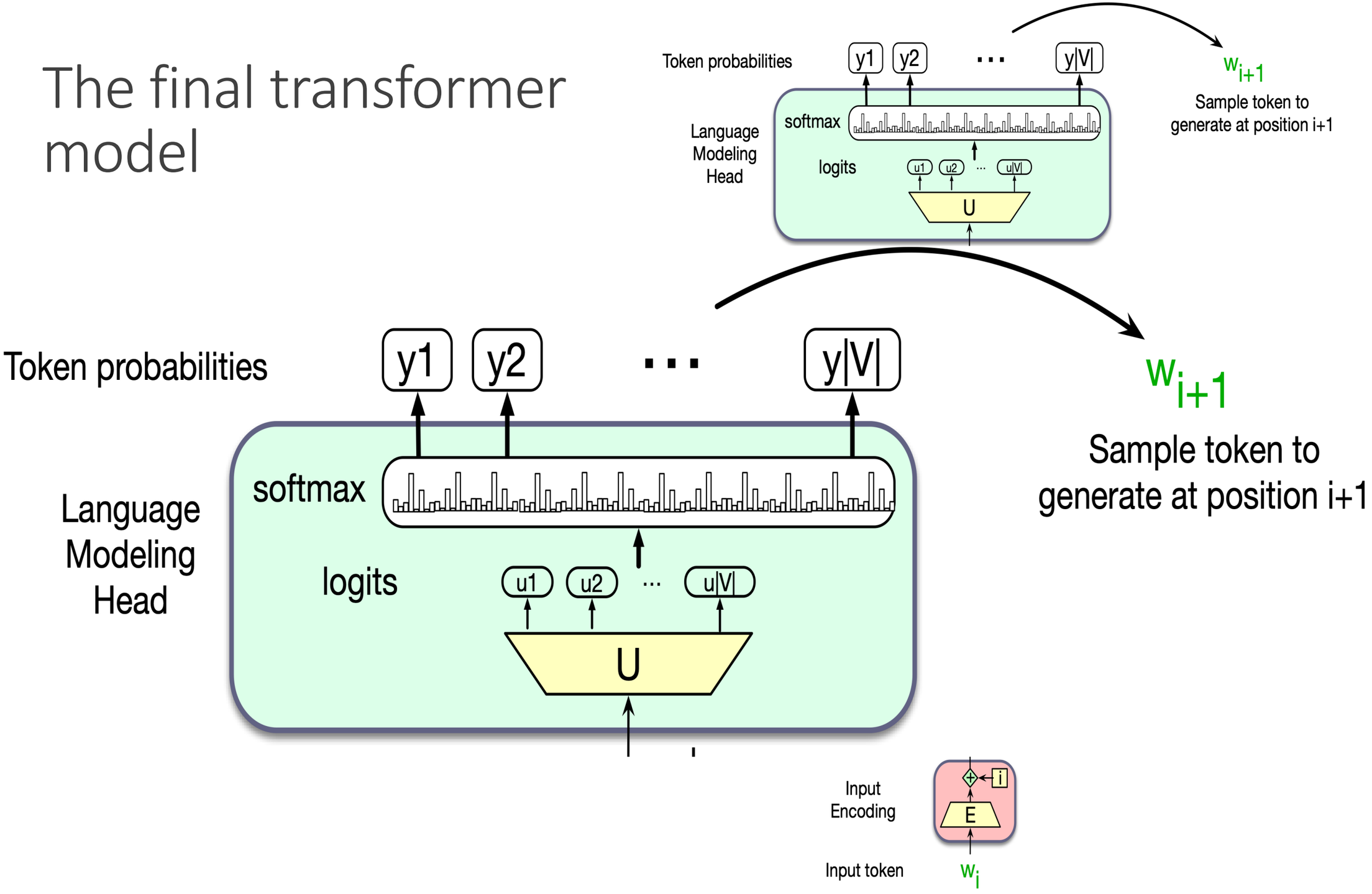
One score for each of the $|V|$ possible words in the vocabulary V. Shape $1 \times |V|$.

Softmax turns the logits into probabilities over vocabulary. Shape $1 \times |V|$.

$$\mathbf{u} = \mathbf{h}_N^L \mathbf{E}^T$$

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

The final transformer model



Transformers

Input and output: Position embeddings and the Language Model Head

Large Language Models

Sampling for LLM Generation

Recall: Conditional Generation: Generating text conditioned on previous text!

1. Give the LLM an input **prompt**
2. Have it generate token by token
   - conditioned on the prompt and the generated tokens

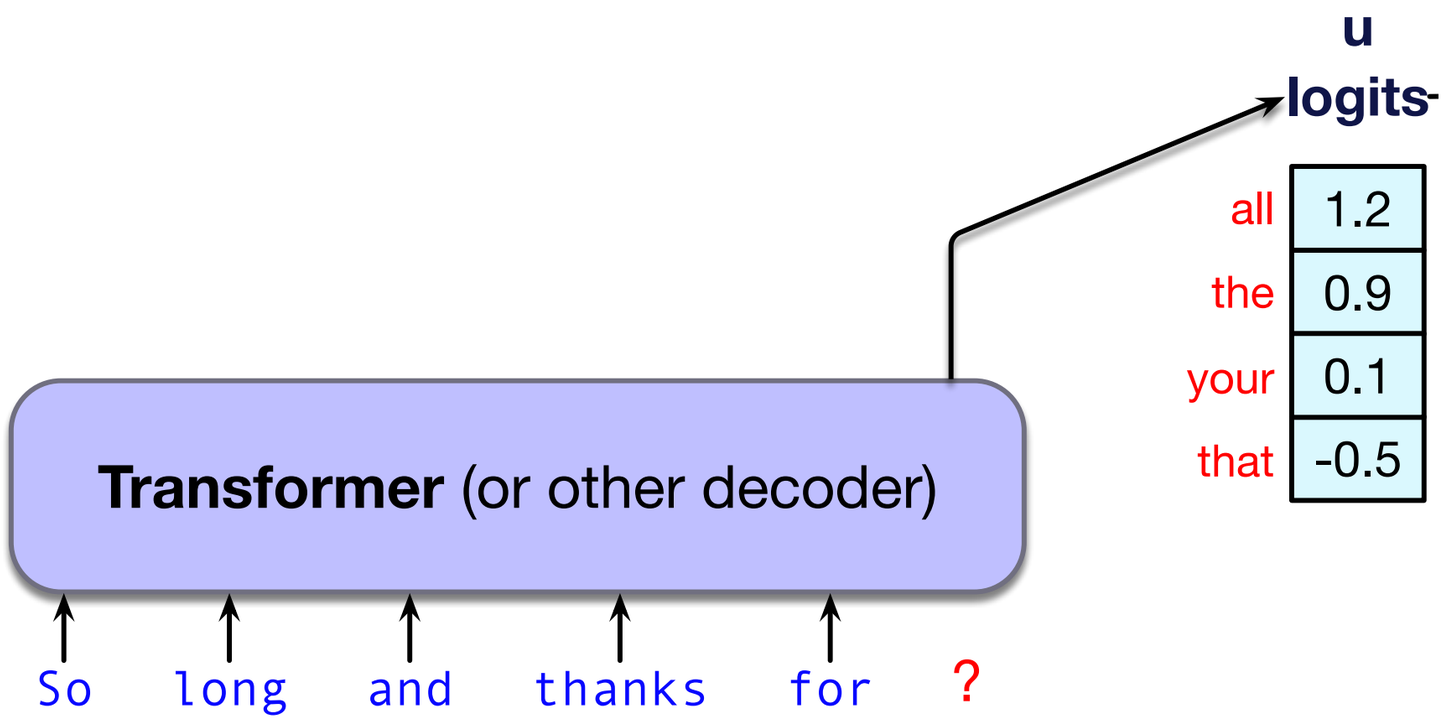
We generate from a model by

1. computing the probability of the next token w_i from the prior context: $P(w_i|w_{<i})$
2. sampling from that distribution to generate a token

Where does token probability come from?

LLMs internally generate real-valued **logits** for each token in the vocabulary.

Score vector **u** of shape $[1 \times |V|]$

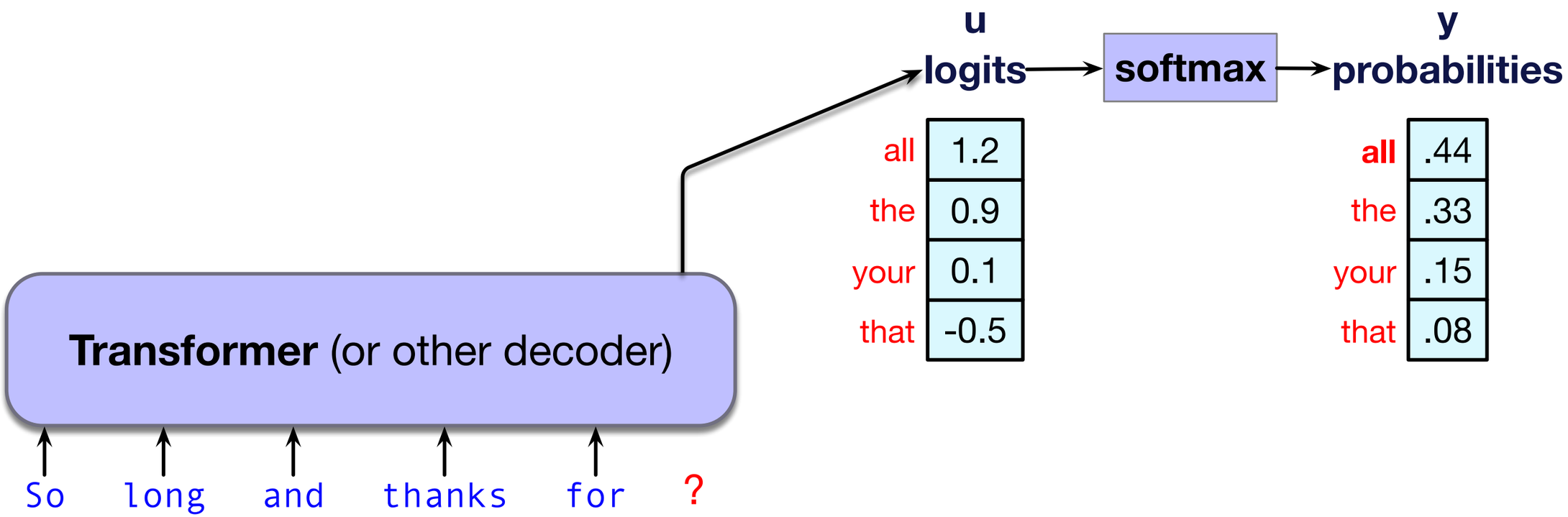


Where does token probability come from?

LLMs internally generate real-valued **logits** for each token in the vocabulary.

Score vector **u** of shape $[1 \times |V|]$ is turned into a probability by softmax

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



Decoding

Choosing a word to generate based on model probabilities

Autoregressive generation:

- Decoding from a model
- left-to-right
- repeatedly choosing the next token
- conditioned on our previous choices

Greedy decoding

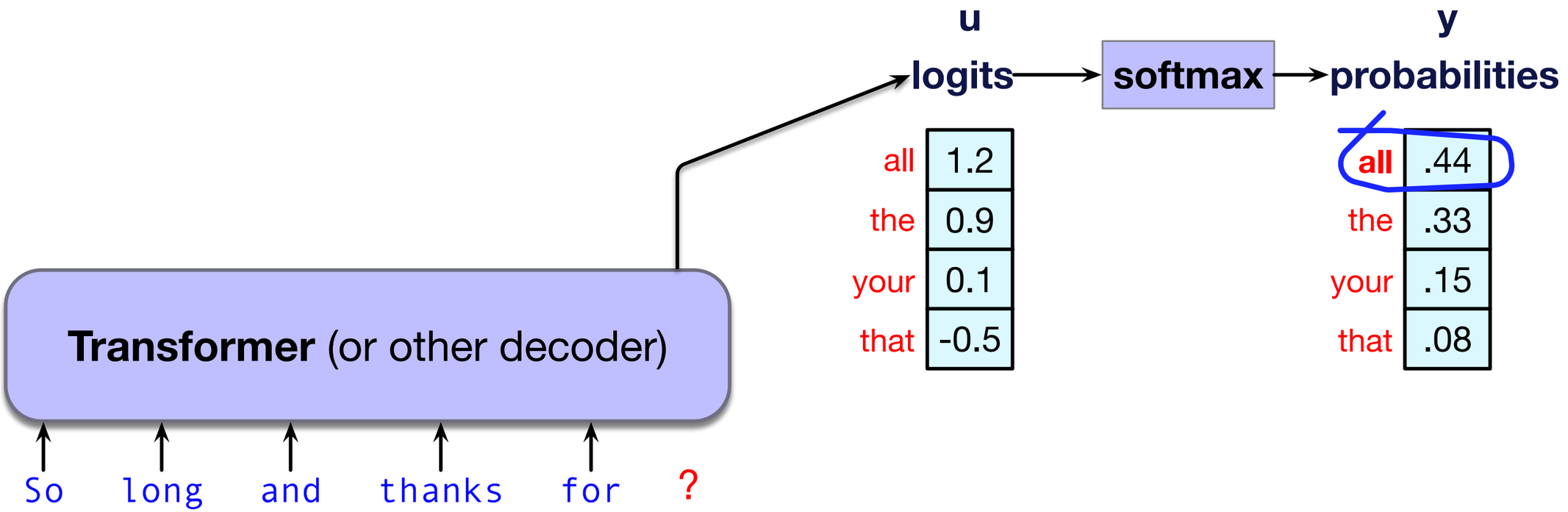
A **greedy algorithm** is one that makes a choice that is locally optimal

- (whether or not it will turn out to have been the best choice with hindsight)

Simply generate the most probable word:

$$\hat{w}_t = \text{argmax}_{w \in V} P(w|\mathbf{w}_{<t})$$

Greedy decoding: choosing "all"



We don't use greedy decoding

It chooses extremely predictable tokens resulting in **generic** and **repetitive** text

Instead, people prefer more diverse text, so we do **sampling**

Random sampling

Sampling from a distribution:

- Choosing random points according to their likelihood.

Sampling from an LM:

- Choosing the next token to generate according to its probability assigned by the LM

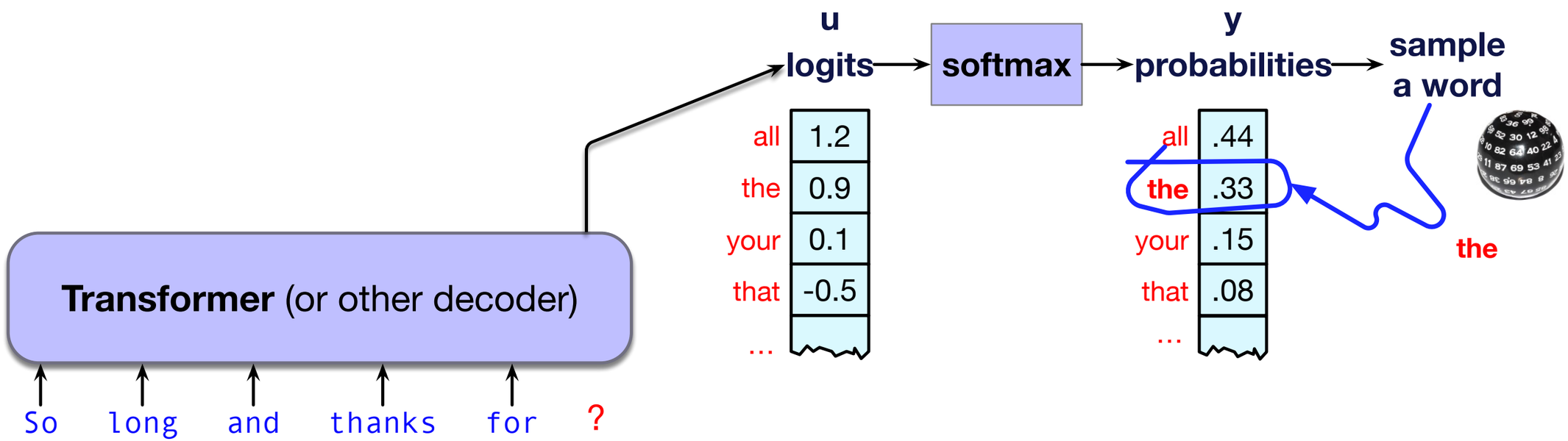
Random (multinomial) sampling:

We randomly select a token to generate

- according to its probability defined by the LM
- conditioned on our previous choices,

Generate it, and iterate.

Random Sampling



Sampling a word from a distribution

Random sampling

Alas, 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

Temperature sampling

Reshape the probability distribution

- increase the probability of the high probability tokens
- decrease the probability of the low probability tokens

Temperature sampling

Divide the logit by a temperature parameter τ before passing it through the softmax.

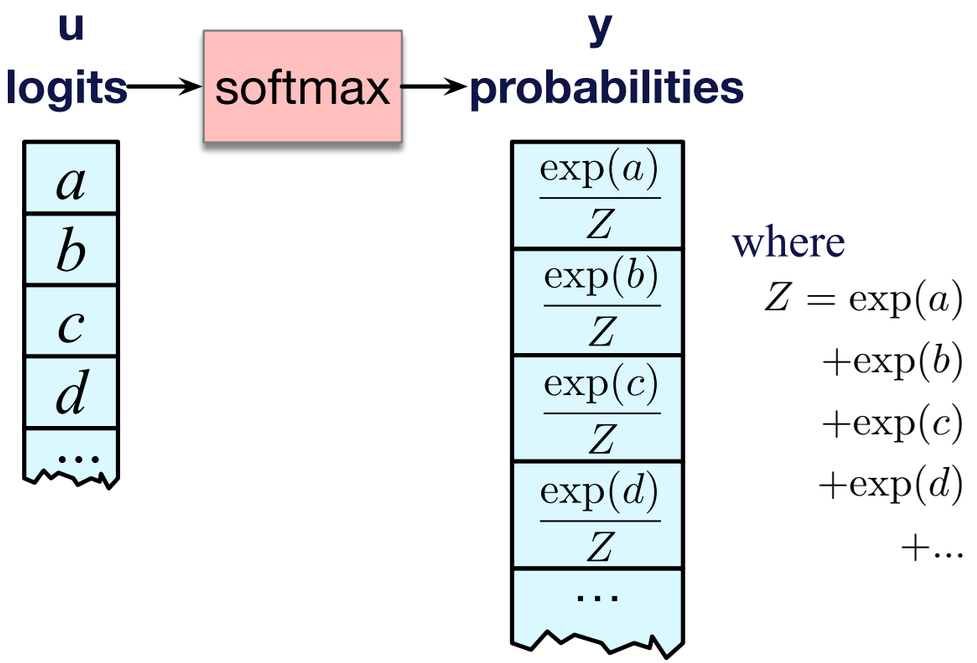
Instead of

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

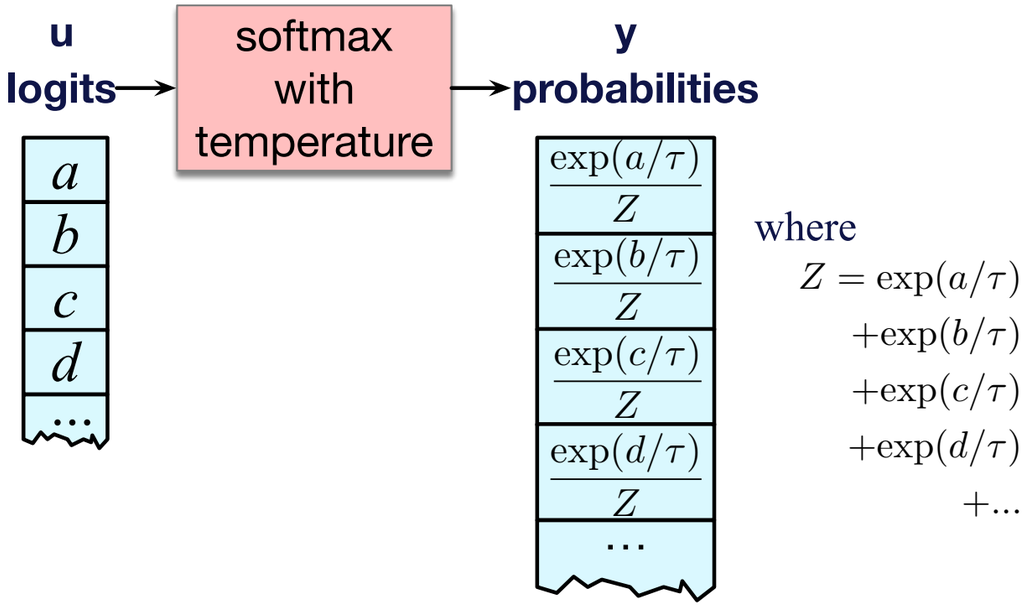
We do

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

Temperature sampling



(a)



(b)

Temperature sampling

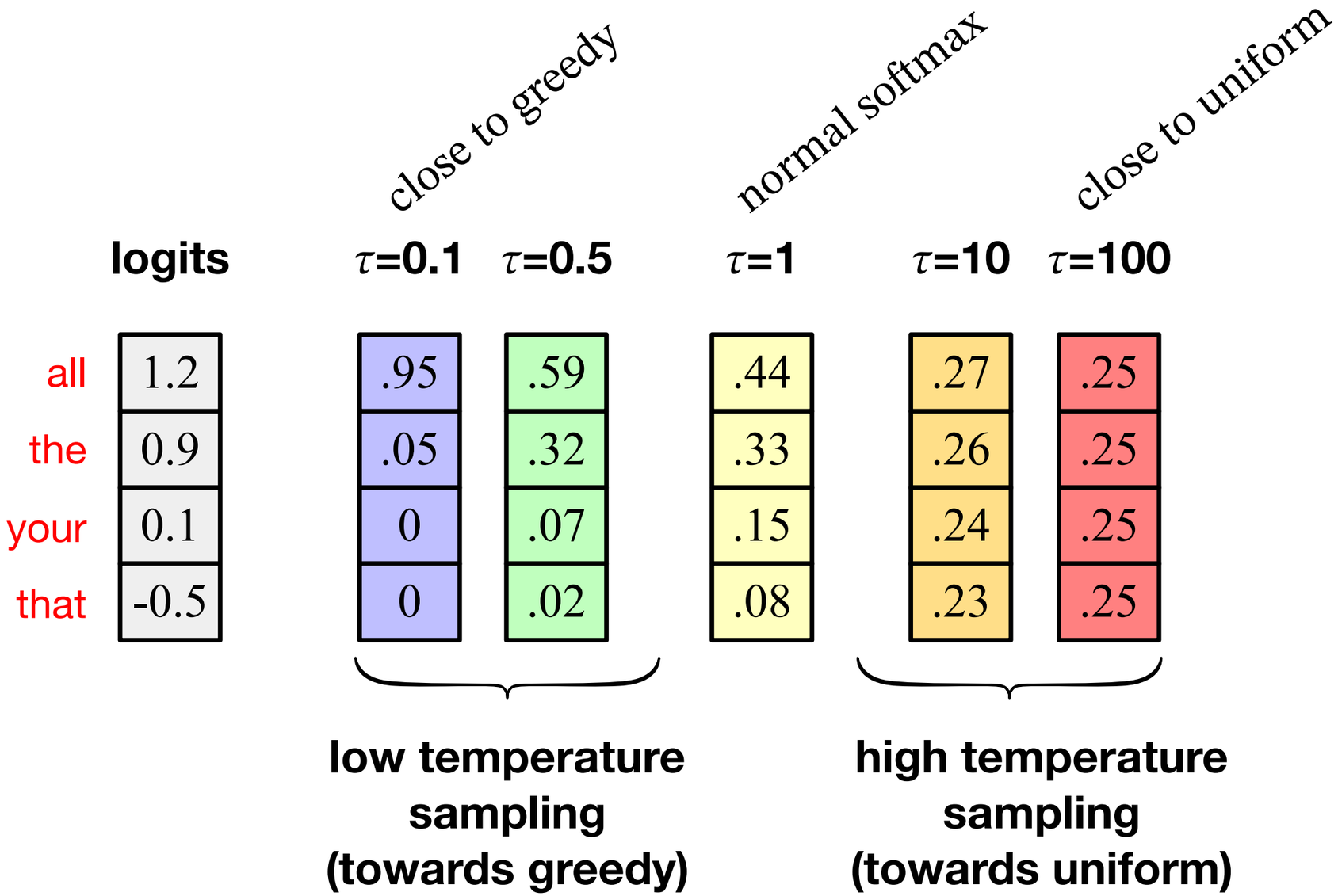
$0 \leq \tau \leq 1$

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

Why does this work?

- When τ is close to 1 the distribution doesn't change much.
- The lower τ 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 τ approaches 0, the probability of most likely word approaches 1

softmax output with temperature τ



Temperature sampling comes 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.

Large Language Models

Sampling for LLM Generation

Large Language Models

Pretraining Transformers

Self-supervised pre-training algorithm

We simply 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!

The language modeling loss

- We want the model to learn to guess the next word
- = to assign a high probability to true word w_{t+1}
- = high loss if the model assigns low probability to w_{t+1}
- So LM loss should go opposite of probability
- The negative log probability of next word

$$-\log p(w_{t+1})$$

- If the model assigns too low a probability to w_{t+1}
  - We move the model weights in the direction that assigns a higher probability to w_{t+1}

From chapter: Why –log prob turns out to be cross-entropy loss

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 **it is 1** for the actual next word and **it is 0** for the others.

- Example: correct next word w_{t+1} is "fish" which is word #3
- Y_t = [0 0 0 1 0 0 0 0 0 0 …..]
- L_{CE} = 0 (log $\hat{y}_t[0]$) + 0 (log $\hat{y}_t[1]$) + 0 (log $\hat{y}_t[2]$) + **1 (log $\hat{y}_t[3]$)** + 0….

So all terms get multiplied by zero except one:

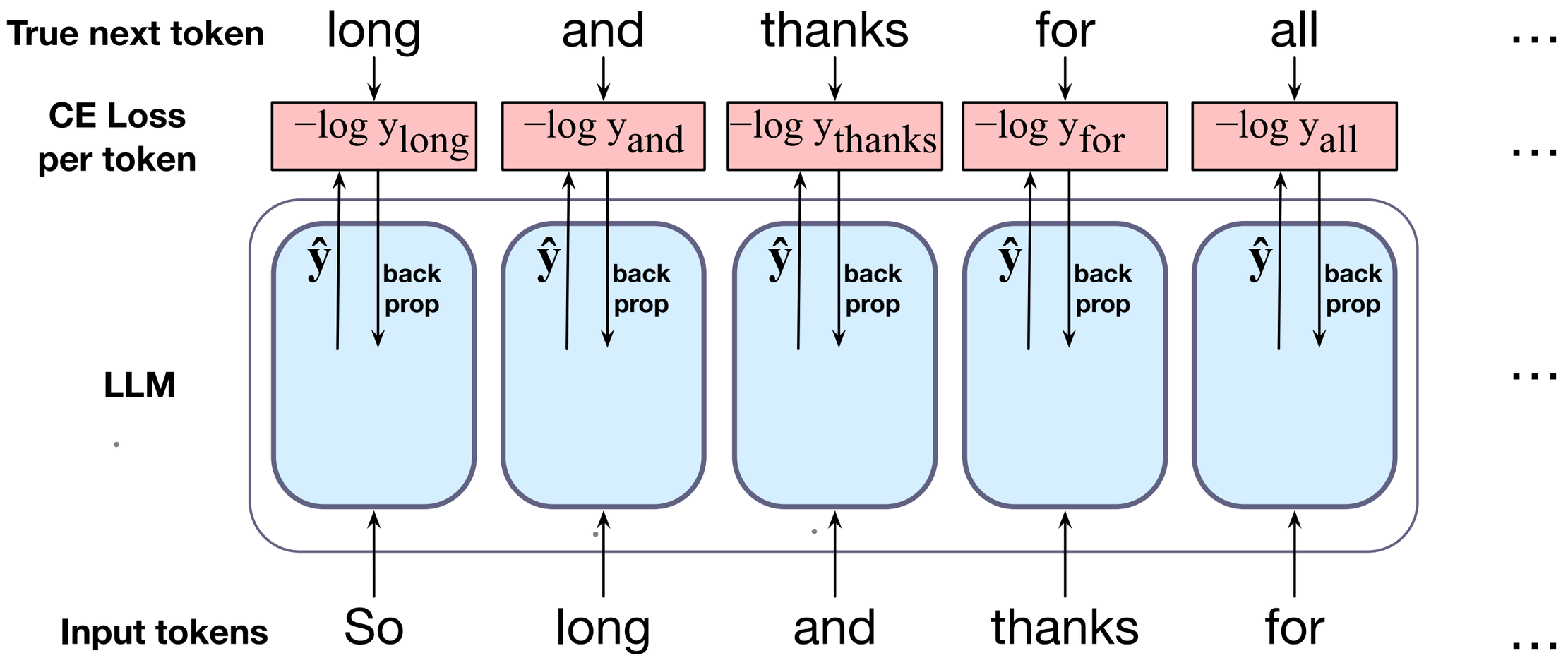
- the logp the model assigns to the correct next word

$$-\log \hat{\mathbf{y}}_t[w_{t+1}]$$

Teacher forcing

- At each token position t,
  - model sees correct tokens $w_{1:t}$,
  - computes loss (–log probability) for the next true 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, and move on

Training a transformer language model



LLMs are mainly trained on the web

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

Fineweb corpus (2025)

18.5 trillion tokens of English

From 96 CommonCrawl dumps from 2013-2024.

How to make an LLM multilingual?

Surprising answer

Just throw all the languages data in a big file and pretrain on it!

FineWeb2 dataset

- 1000 languages
- 5 billion documents
- From Common Crawl

ISO 639-3	Script	Name	Language Family	Words	Documents	Disk size
rus	Cyrl	Russian	Indo-European	588,579,493,780	699,083,579	5.82TB
cmn	Hani	Mandarin Chinese	Sino-Tibetan	543,543,038,750	636,058,984	2.42TB
deu	Latn	German	Indo-European	262,271,052,199	495,964,485	1.51TB
jpn	Jpan	Japanese	Japonic	331,144,301,801	400,138,563	1.50TB
spa	Latn	Spanish	Indo-European	261,523,749,595	441,287,261	1.32TB
fra	Latn	French	Indo-European	220,662,584,640	360,058,973	1.11TB
ita	Latn	Italian	Indo-European	139,116,026,491	238,984,437	739.24GB
por	Latn	Portuguese	Indo-European	109,536,087,117	199,737,979	569.24GB
pol	Latn	Polish	Indo-European	73,119,437,217	151,966,724	432.01GB
nld	Latn	Dutch	Indo-European	74,634,633,118	147,301,270	397.51GB
ind	Latn	Indonesian	Austronesian	60,264,322,142	100,238,529	348.65GB
vie	Latn	Vietnamese	Austro-Asiatic	50,886,874,358	61,064,248	319.83GB
fas	Arab	Persian	Indo-European	39,705,799,658	58,843,652	304.62GB
arb	Arab	Standard Arabic	Afro-Asiatic	32,812,858,120	61,977,525	293.59GB
tur	Latn	Turkish	Turkic	41,933,799,420	95,129,129	284.52GB
tha	Thai	Thai	Kra-Dai	24,662,748,945	35,897,202	278.68GB
ukr	Cyrl	Ukrainian	Indo-European	25,586,457,655	53,101,726	254.86GB
ell	Grek	Modern Greek (1453-)	Indo-European	22,827,957,288	47,421,073	222.05GB
kor	Hang	Korean	Koreanic	48,613,120,582	60,874,355	213.43GB
ces	Latn	Czech	Indo-European	35,479,428,809	66,067,904	206.33GB
swe	Latn	Swedish	Indo-European	35,745,969,364	59,485,306	202.96GB
hun	Latn	Hungarian	Uralic	30,919,839,164	49,935,986	199.69GB
ron	Latn	Romanian	Indo-European	35,017,893,659	58,303,671	186.19GB
nob	Latn	Norwegian Bokmål	Indo-European	32,008,904,934	38,144,343	172.05GB
dan	Latn	Danish	Indo-European	28,055,948,840	45,391,655	150.72GB
bul	Cyrl	Bulgarian	Indo-European	16,074,326,712	25,994,731	145.75GB
fin	Latn	Finnish	Uralic	20,343,096,672	36,710,816	143.03GB
hin	Deva	Hindi	Indo-European	11,173,681,651	22,095,985	120.98GB
ben	Beng	Bengali	Indo-European	6,153,579,265	15,185,742	87.04GB

Large Language Models

Pretraining Large Language Models

Large Language Models

Evaluating Large Language Models

Better LMs are better at predicting text

Reminder of the chain rule:

$$P(w_{1:n}) = P(w_1)P(w_2|w_1)P(w_3|w_{1:2})\ldots P(w_n|w_{1:n-1})$$
$$= \prod_{i=1}^{n} P(w_i|w_{<i})$$

Given text $w_{1:n}$ we could just compare the log likelihood from two LMs:

$$\text{LM}_\text{A}\ \text{log likelihood}(w_{1:n}) = \log \prod_{i=1}^{n} P_\text{A}(w_i|w_{<i})$$

$$\text{LM}_\text{B}\ \text{log likelihood}(w_{1:n}) = \log \prod_{i=1}^{n} P_\text{B}(w_i|w_{<i})$$

But raw log-likelihood has problems

Probability depends on size of test set

- Probability gets smaller the longer the text
- We would prefer a metric that is **per-word**, normalized by length

Perplexity is normalized for 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,∞]

Perplexity

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

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

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

$$\begin{aligned} \text{Perplexity}(w_{1:T}) &= P(w_{1:T})^{-\frac{1}{T}} \\ &= \left(\prod_{t=1}^{T} P(w_t|w_{<t})^{-1}\right)^{\frac{1}{T}} \end{aligned}$$

Perplexity

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

Also:

1. Perplexity turns out to be the exp of the cross-entropy

$$L_{CE}(\text{batch of length T}) = \frac{1}{T}\sum_{t=1}^{T} -\log P(w_t|w_{<t})$$

2. Perplexity is sensitive to length/tokenization so best used when comparing LMs that use the same tokenizer.

(as we said in lecture 1)
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_2 emitted

Fairness

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