Large Language Models

Introduction to Large Language Models

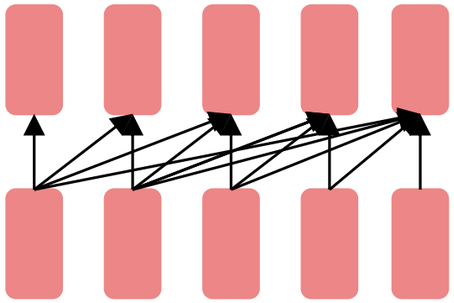
Language models

- 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

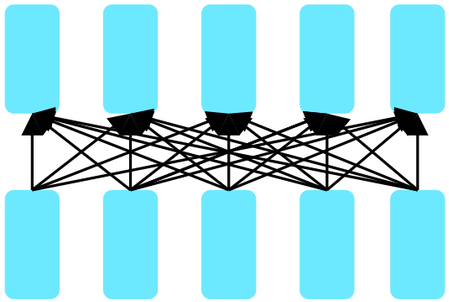
- Even through pretrained only to predict words
- Learn a lot of useful language knowledge
- Since training on a **lot** of text

Three architectures for large language models



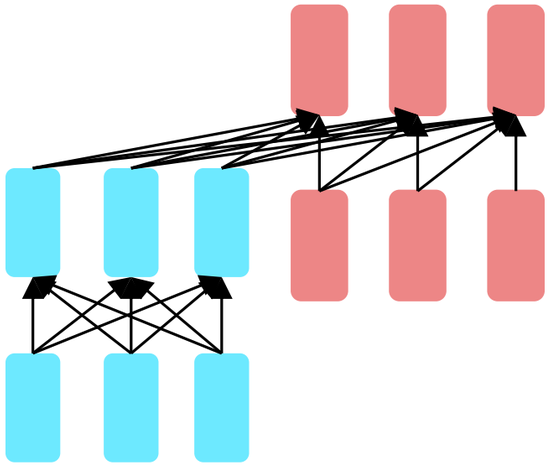
Decoders

GPT, Claude, Llama Mixtral



Encoders

BERT family, HuBERT



Encoder-decoders

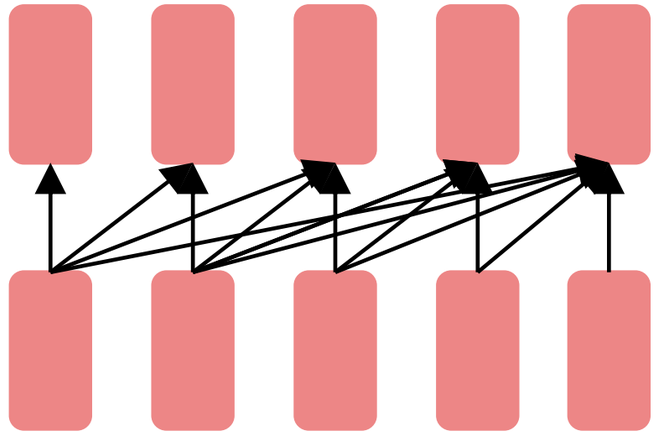
Flan-T5, Whisper

Today's lecture: decoder-only models

What people usually mean when they say "LLM"
(GPT, Claude, Gemini, etc)

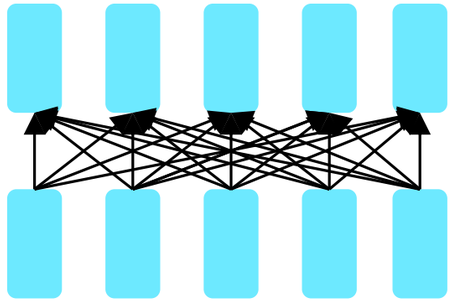
Also called:

- Causal LLMs
- Autoregressive LLMs
- Left-to-right LLMs



- Predict words left to right

Encoders

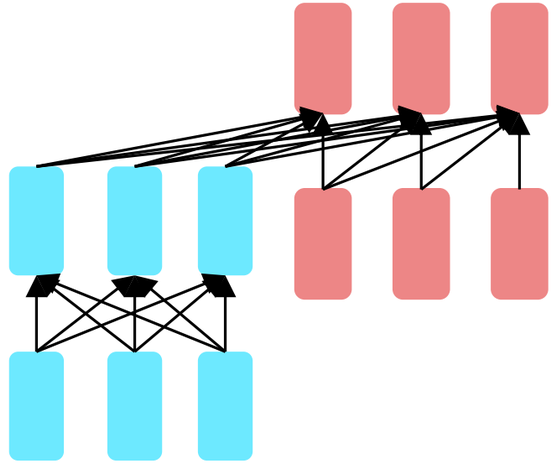


Many varieties!

- Popular: Masked Language Models (MLMs)
- BERT family (what you used in PA5!)

- 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)

Big idea

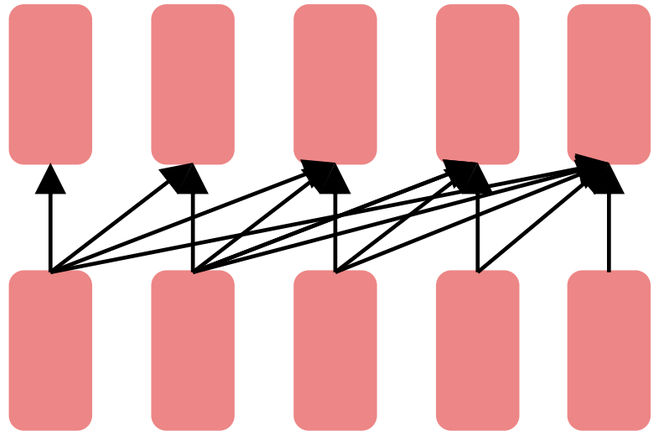
Many tasks can be turned into tasks of predicting words!

Today's lecture: decoder-only models

What people usually mean when they say "LLM" (ChatGPT, Claude, Gemini, etc)

Also called:

- Causal LLMs
- Autoregressive LLMs
- Left-to-right LLMs



- Predict words left to right

The intuition (before we see the details)

1. Given the current prefix

   So long and thanks for all

2. Use the LM to compute a probability distribution over all words in V, conditioned on all the prior words

   P(aardvark | So long and thanks for all)

   P(abaft | So long and thanks for all)

   …

3. Choose one to generate

   So long and thanks for all **the**

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:

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

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:

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

Charles

3. Now iterate:

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

So to do language modeling

We just need a system

That can take a prefix string of words *s*

And compute the probability that word *w* follows *s*

Modern LMs are built out of stacks of neural networks called transformers

Large Language Models

Introduction to Large Language Models

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

2004-6 GPUs begin to be used

2008 Multi-Task Learning

2012 GPUs take off

2013 Static Word Embeddings re-discovered

2015 Attention

2017 Transformer

2018 Contextual Word Embeddings and Pretraining

2019 Prompting

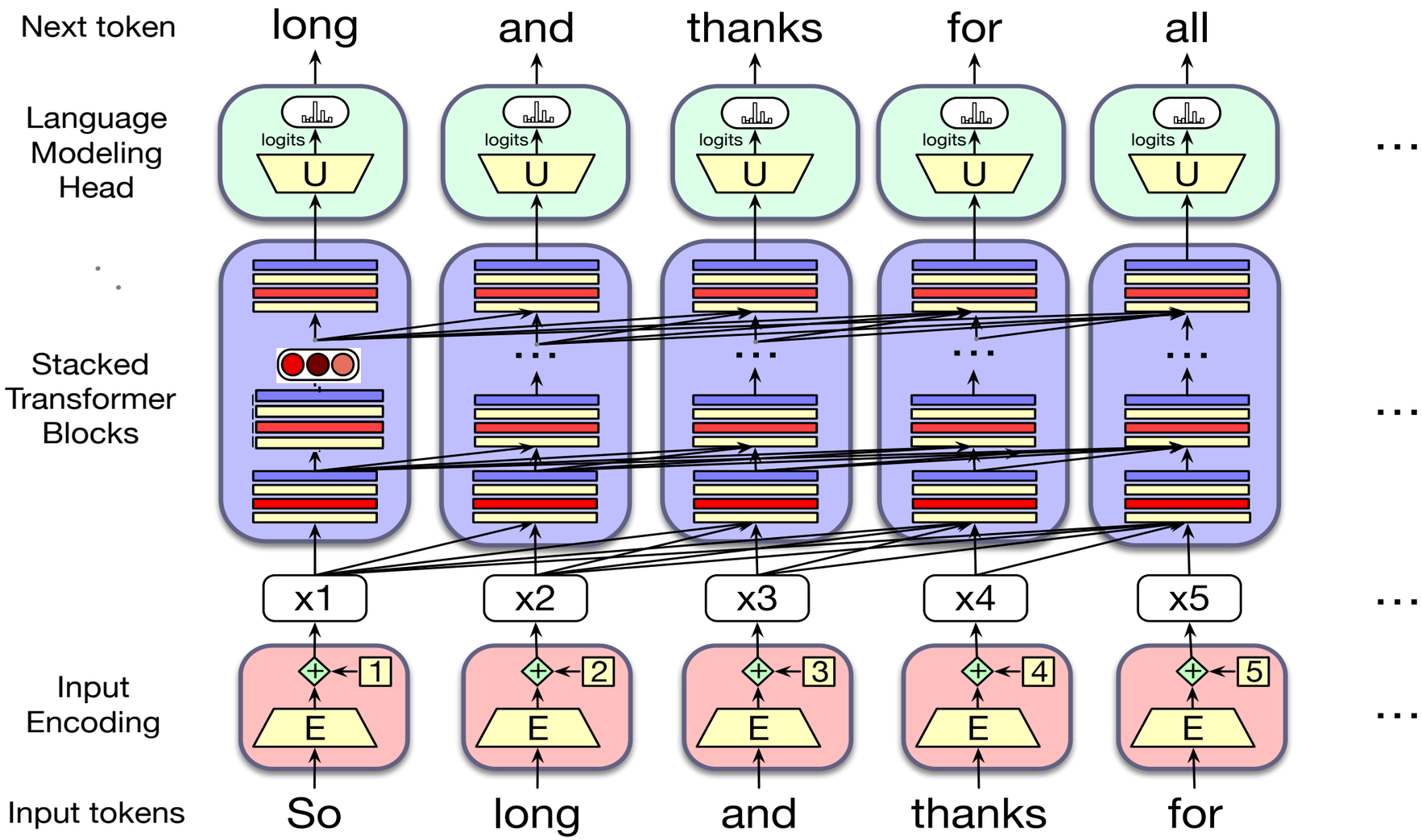
2022 ChatGPT

Transformers

Attention

Instead of starting with the big picture

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



Problem with static embeddings (word2vec)

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

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

Contextual Embeddings

- Intuition: a representation of meaning of a word should be different in different contexts!
- **Contextual Embedding**: each word has a different vector that expresses different meanings depending on the surrounding words
- How to compute contextual embeddings?
  - **Attention**

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`**

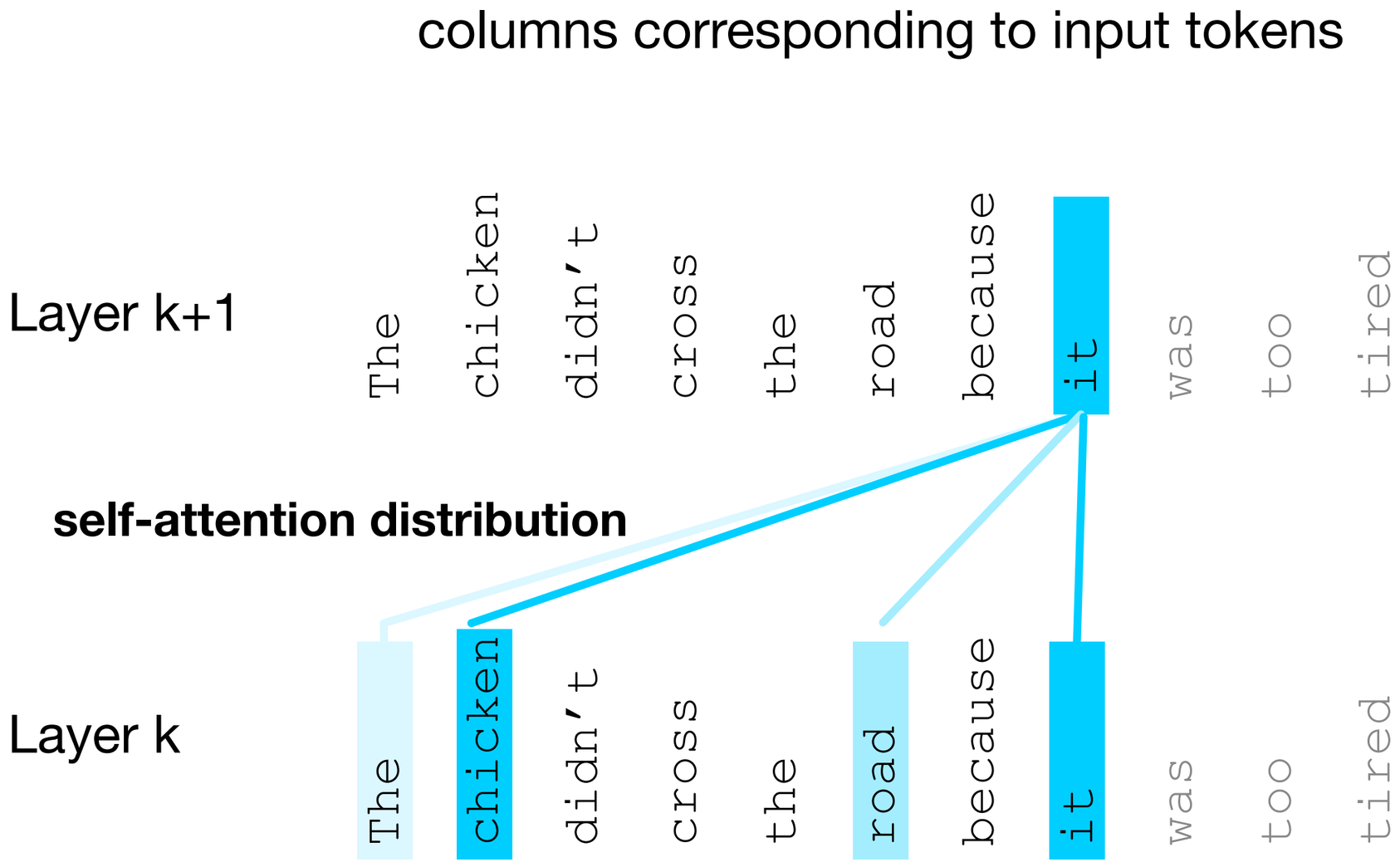
At this point in the sentence, it's probably referring to either the chicken or the street

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:

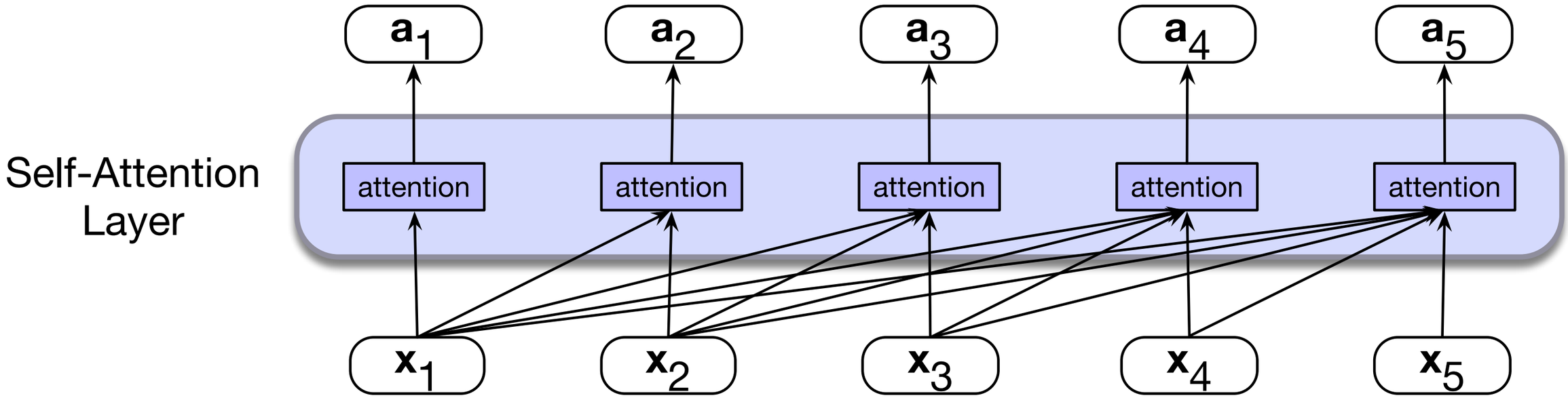


Attention definition

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:

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

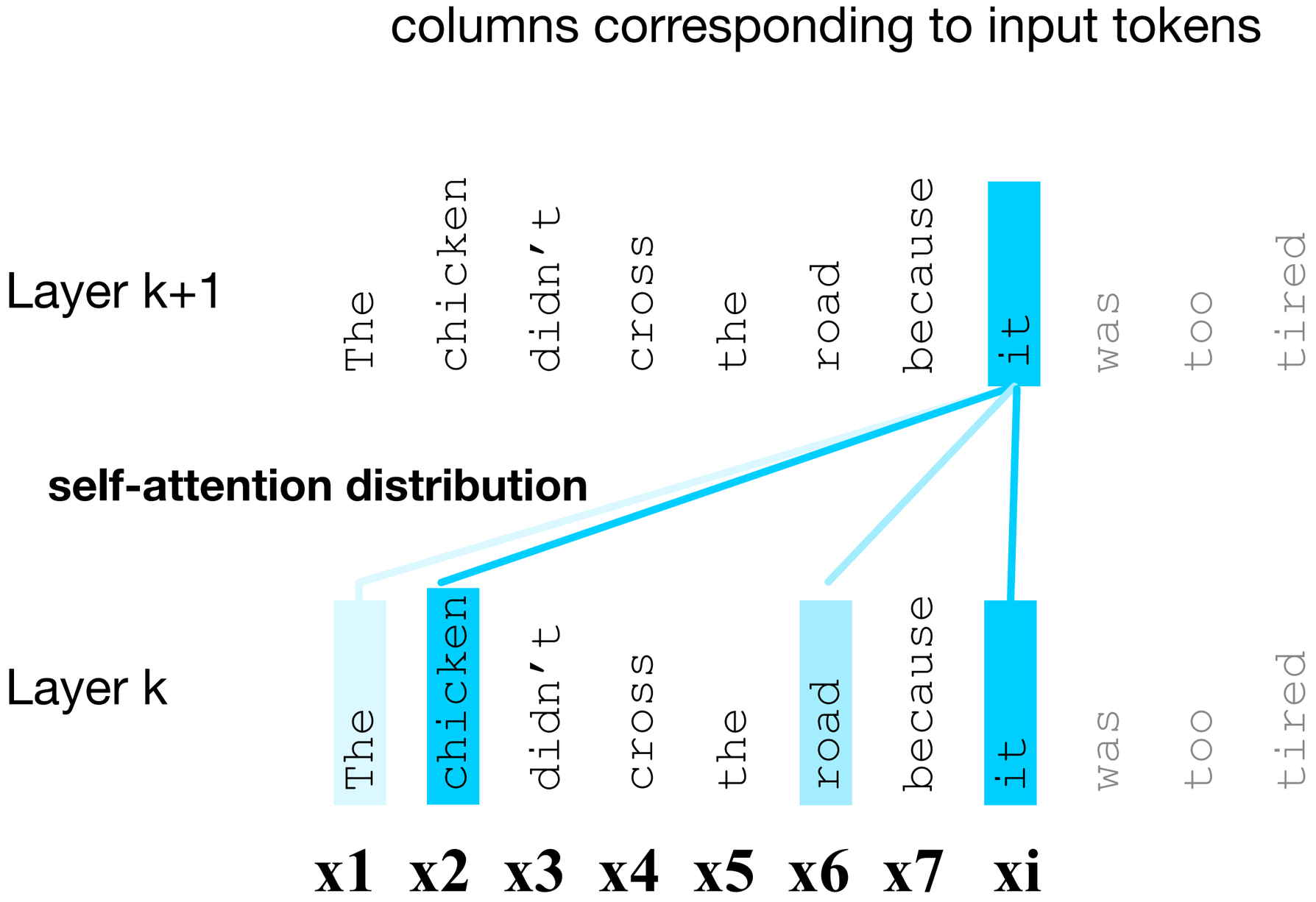
Produce: $\mathbf{a}_i$ = a weighted sum of $\mathbf{x}_1$ through $\mathbf{x}_7$ (and $\mathbf{x}_i$)
Weighted by their similarity to $\mathbf{x}_i$

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

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

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

Intuition of attention:

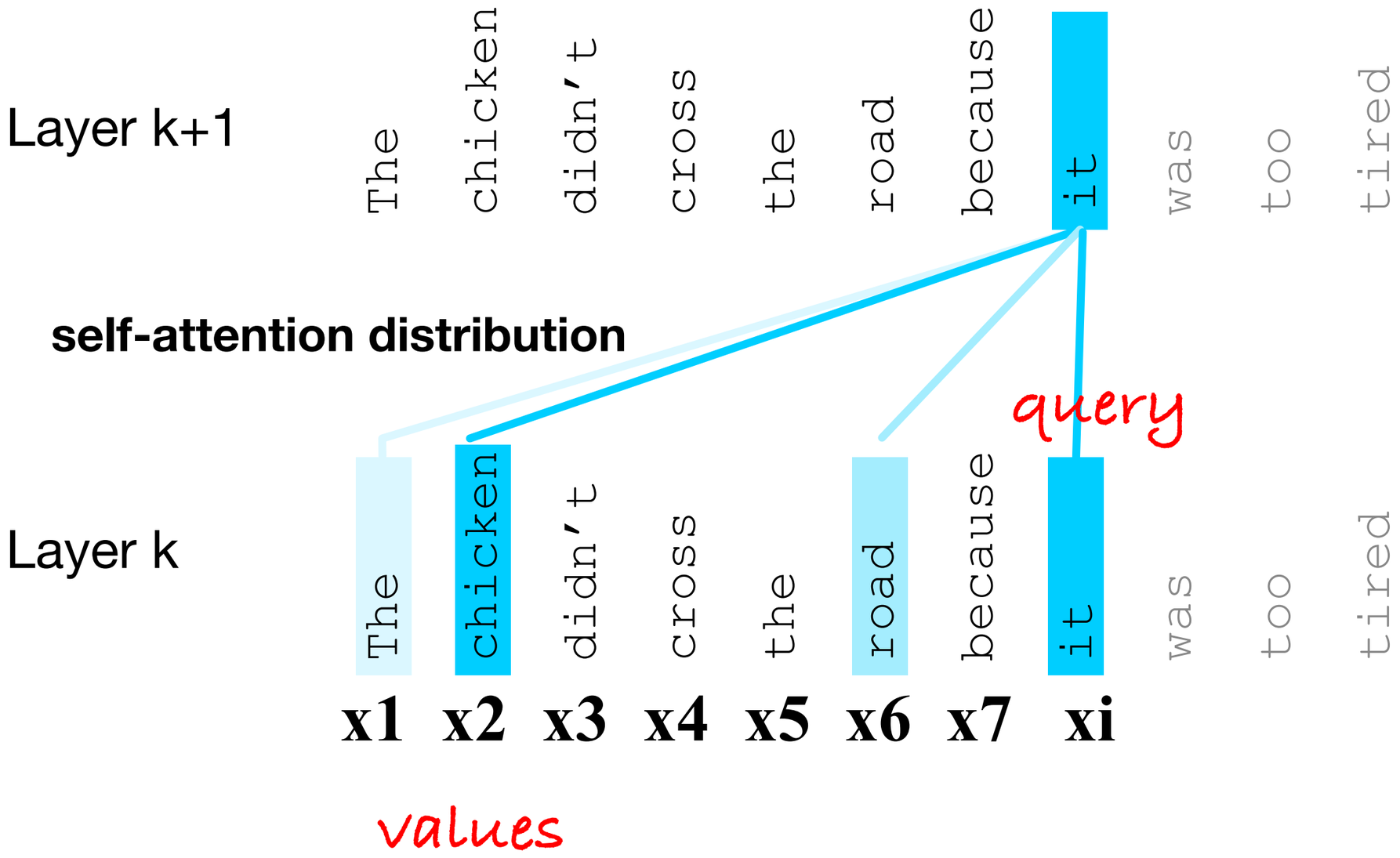


An Actual Attention Head: slightly more complicated

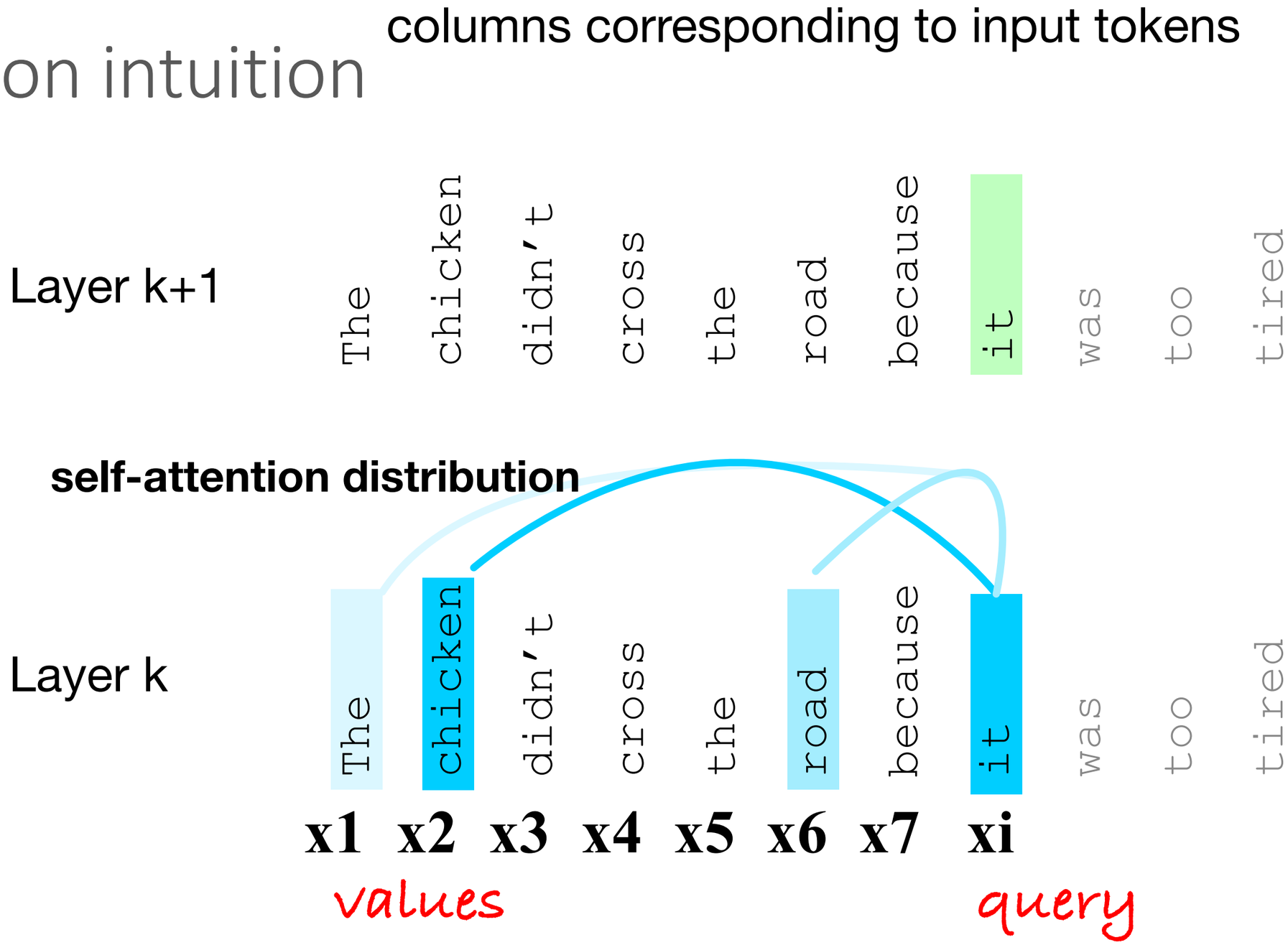
High-level idea: instead of using vectors (like x_i and x_4) directly, we'll represent 3 separate roles each vector $\mathbf{x}_i$ plays:

- **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

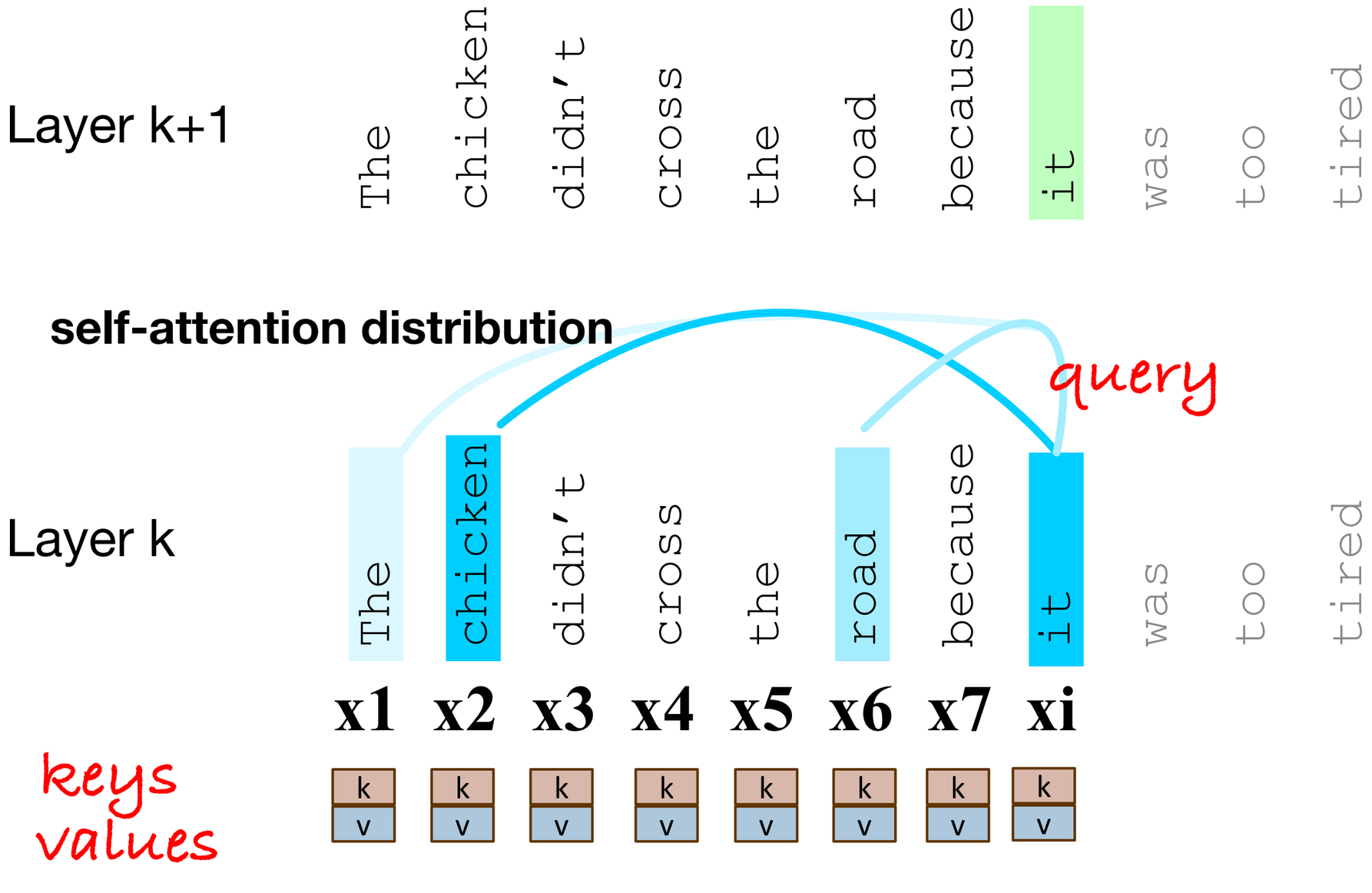
Attention intuition



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

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

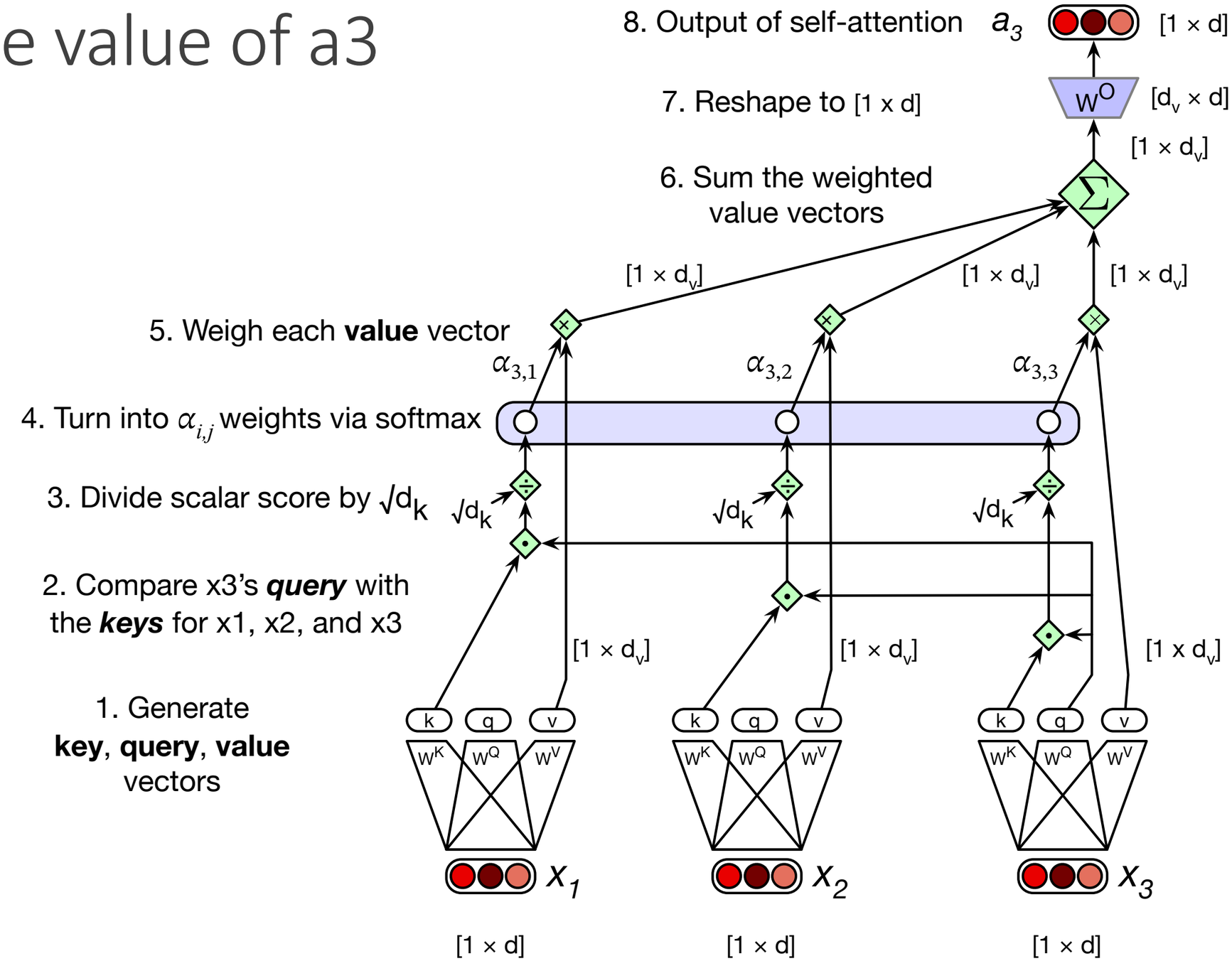
$$\text{score}(\mathbf{x}_i, \mathbf{x}_j) = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$$

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

$$\mathbf{head}_i = \sum_{j \leq i} \alpha_{ij} \mathbf{v}_j$$

$$\mathbf{a}_i = \mathbf{head}_i \mathbf{W}^{\mathbf{O}}$$

Calculating the value of a3



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!

And enriched representation can be passed up layer by layer.

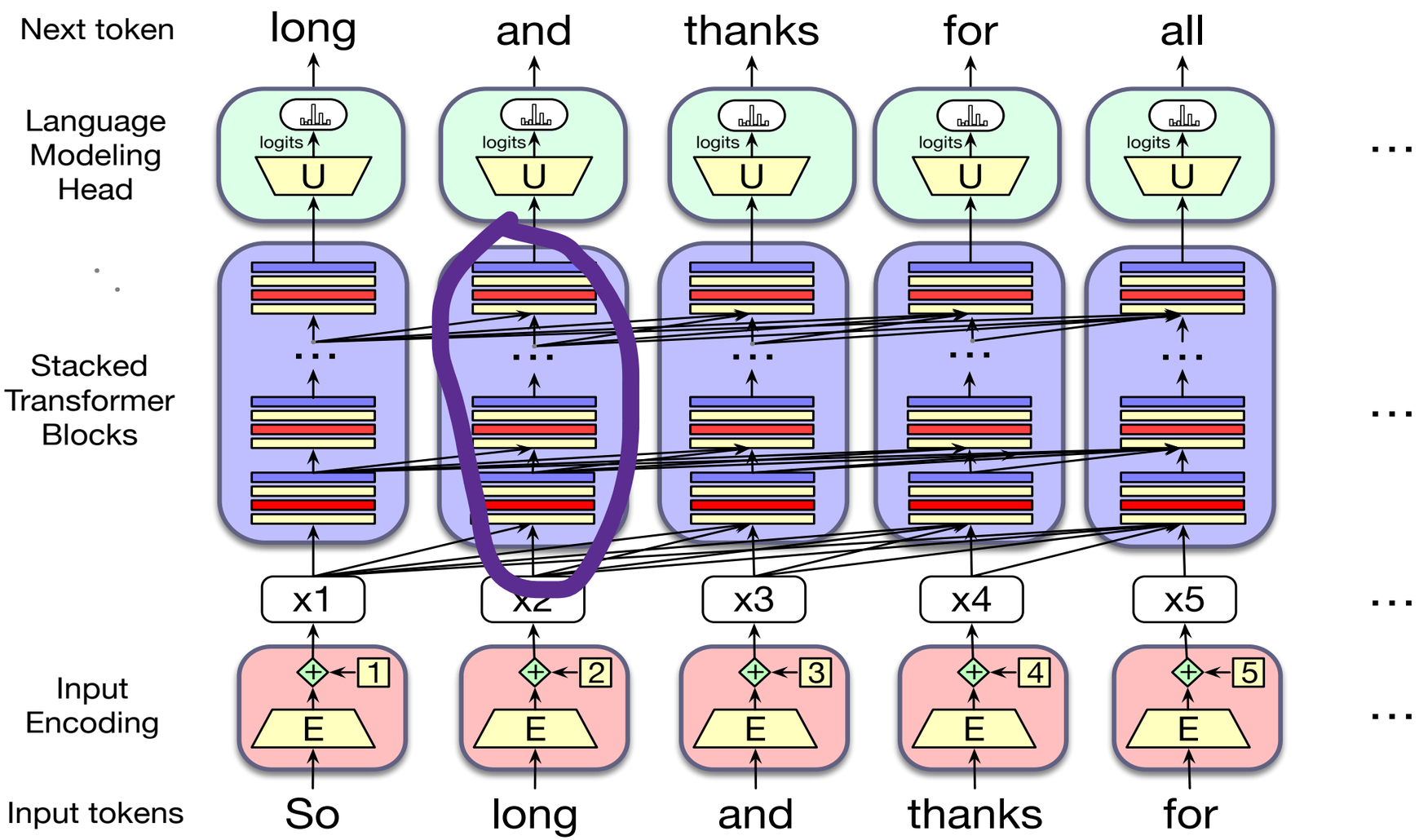
Transformers

Attention

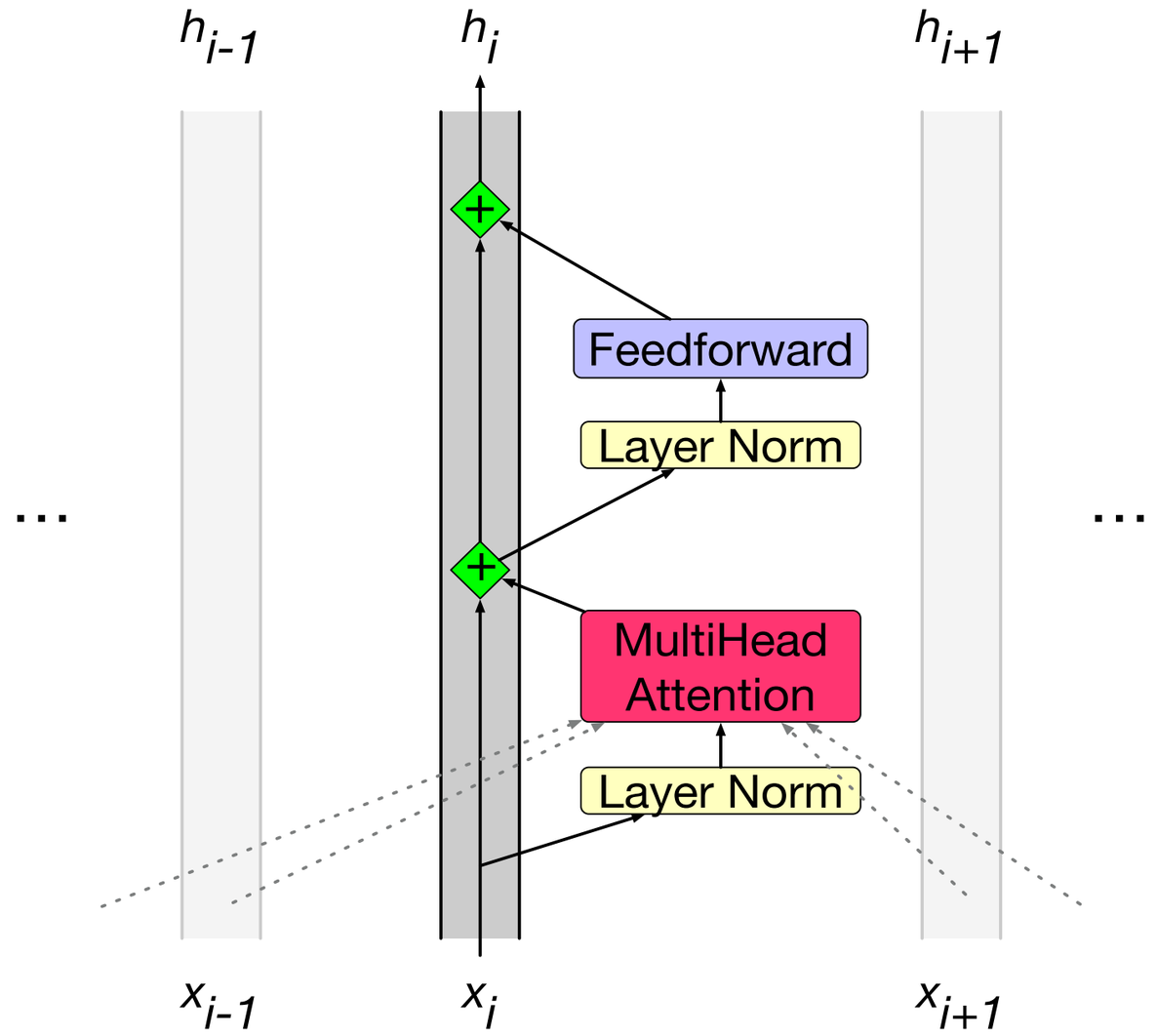
Transformers

The Transformer Block

Transformer language model

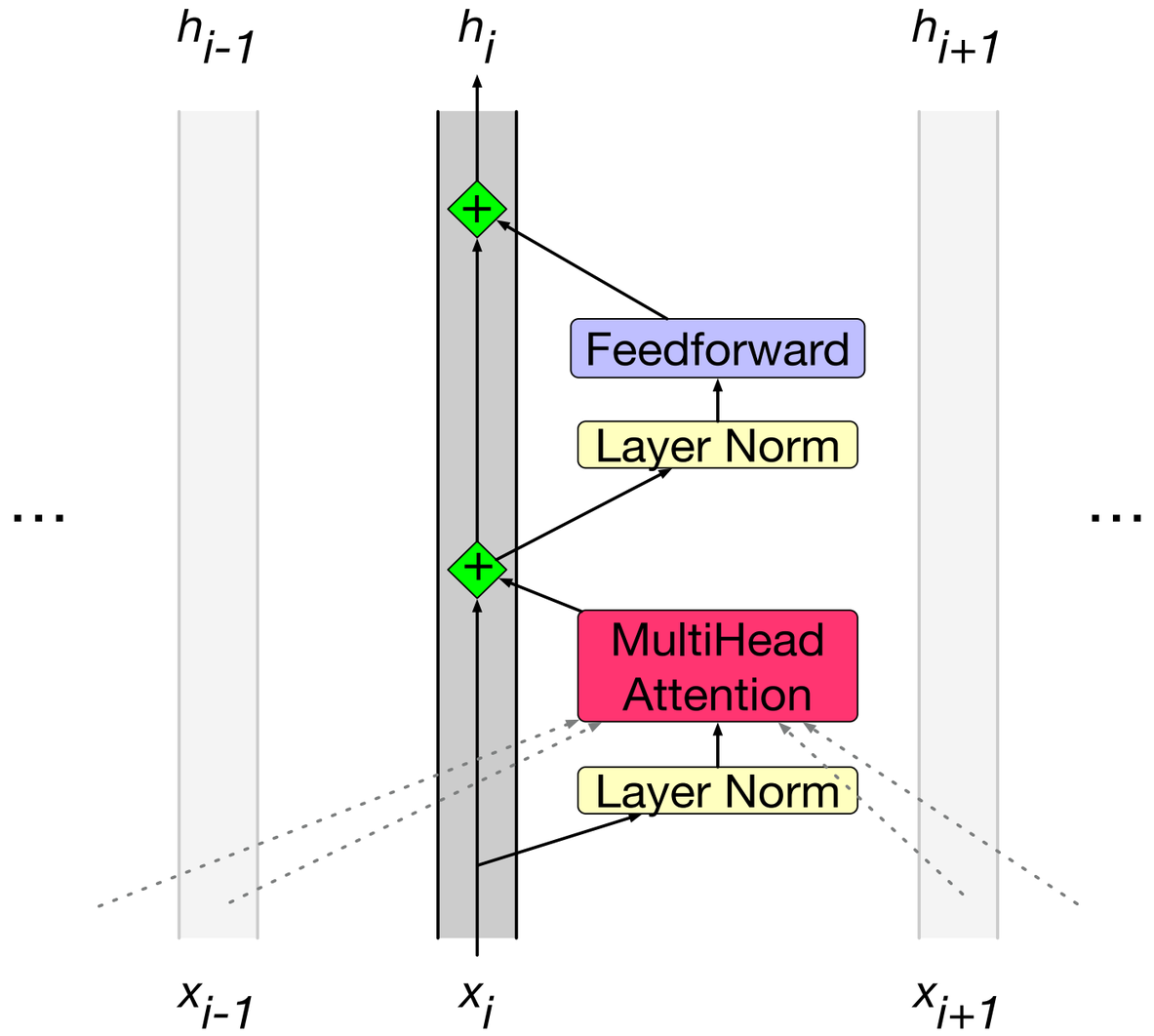


The residual stream: each token gets passed up and modified

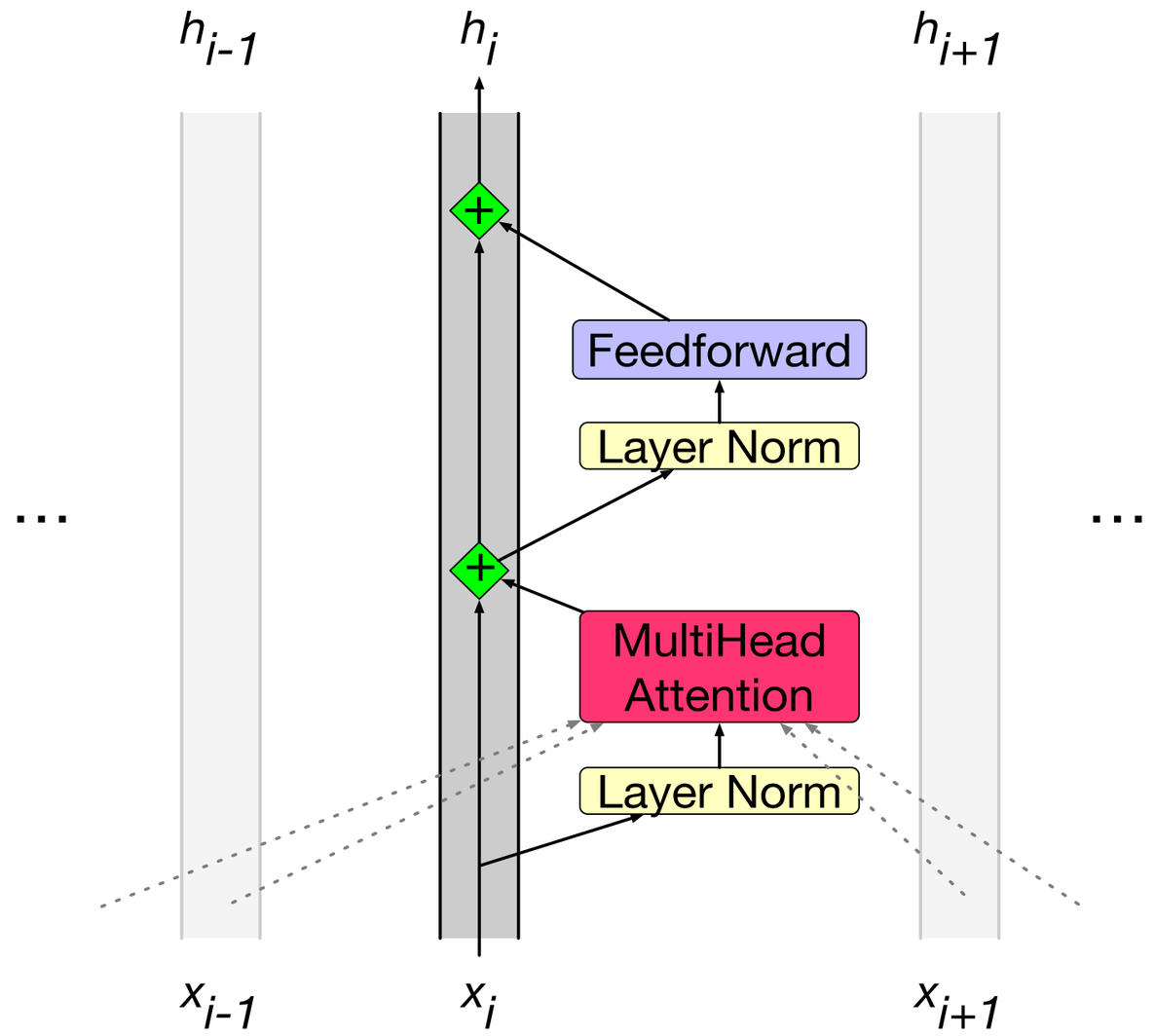


We'll need nonlinearities, so a feedforward layer

$$\mathrm{FFN}(\mathbf{x}_i) = \mathrm{ReLU}(\mathbf{x}_i\mathbf{W}_1 + b_1)\mathbf{W}_2 + b_2$$



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 vector in a hidden layer

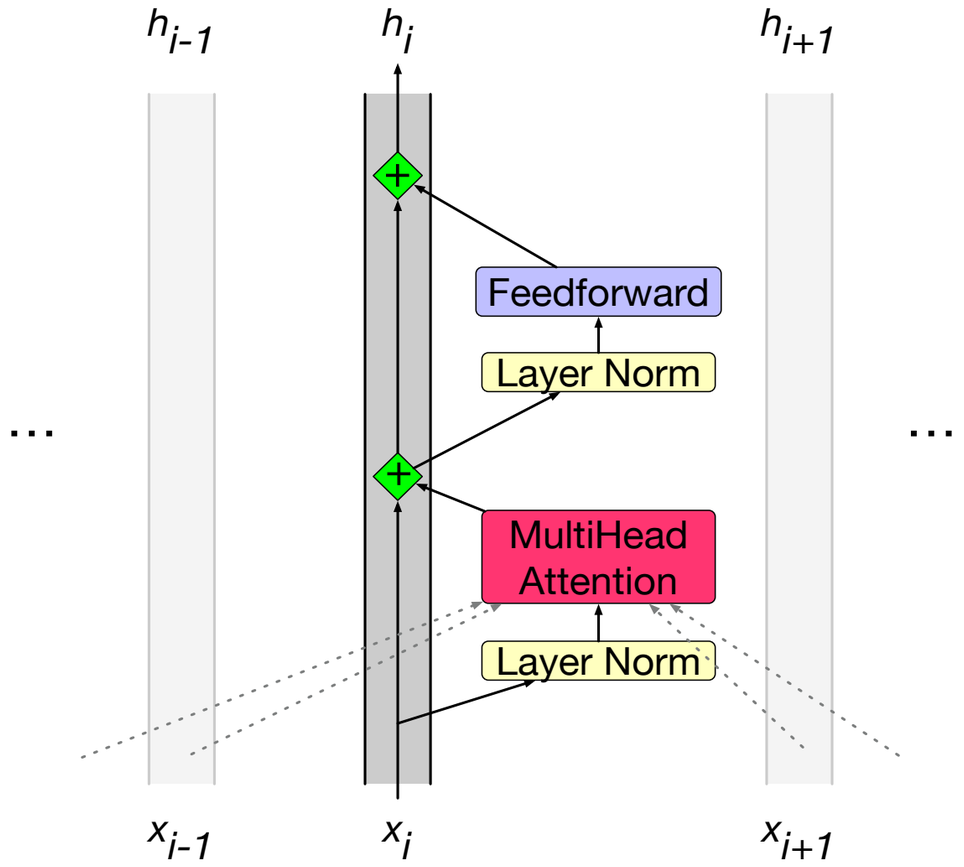
$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

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

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

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

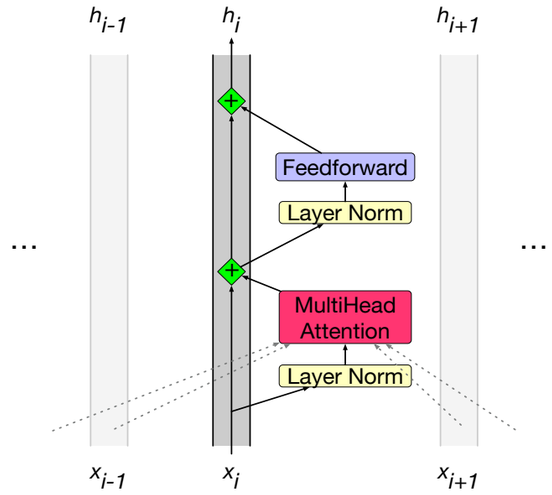
Putting together a single transformer block



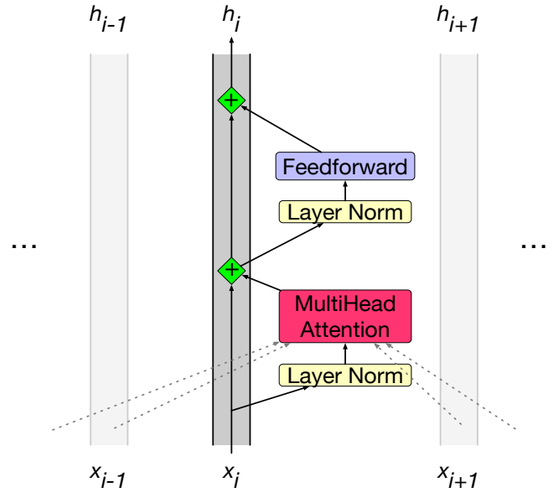
$$
\begin{aligned}
\mathbf{t}_i^1 &= \text{LayerNorm}(\mathbf{x}_i) \\
\mathbf{t}_i^2 &= \text{MultiHeadAttention}(\mathbf{t}_i^1, [\mathbf{x}_1^1, \cdots, \mathbf{x}_N^1]) \\
\mathbf{t}_i^3 &= \mathbf{t}_i^2 + \mathbf{x}_i \\
\mathbf{t}_i^4 &= \text{LayerNorm}(\mathbf{t}_i^3) \\
\mathbf{t}_i^5 &= \text{FFN}(\mathbf{t}_i^4) \\
\mathbf{h}_i &= \mathbf{t}_i^5 + \mathbf{t}_i^3
\end{aligned}
$$

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

Block 2



Block 1



Transformers

The Transformer Block

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.

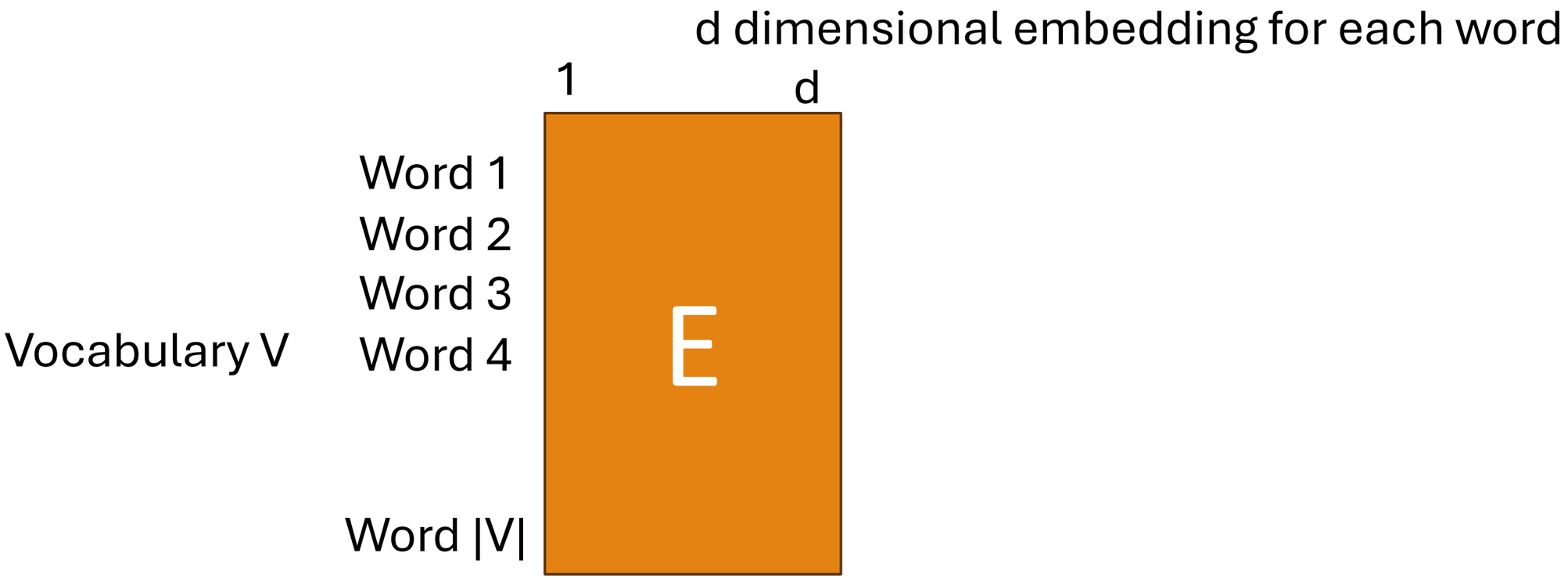
Each embedding of shape [1× d] is created by adding two distinct embedding for each input

- token embedding
- positional embedding

Embedding matrix E

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

- One row for each of the $|V|$ tokens in the vocabulary.
- Each word embedding is a row vector of shape [1 × d]



Token Embeddings

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

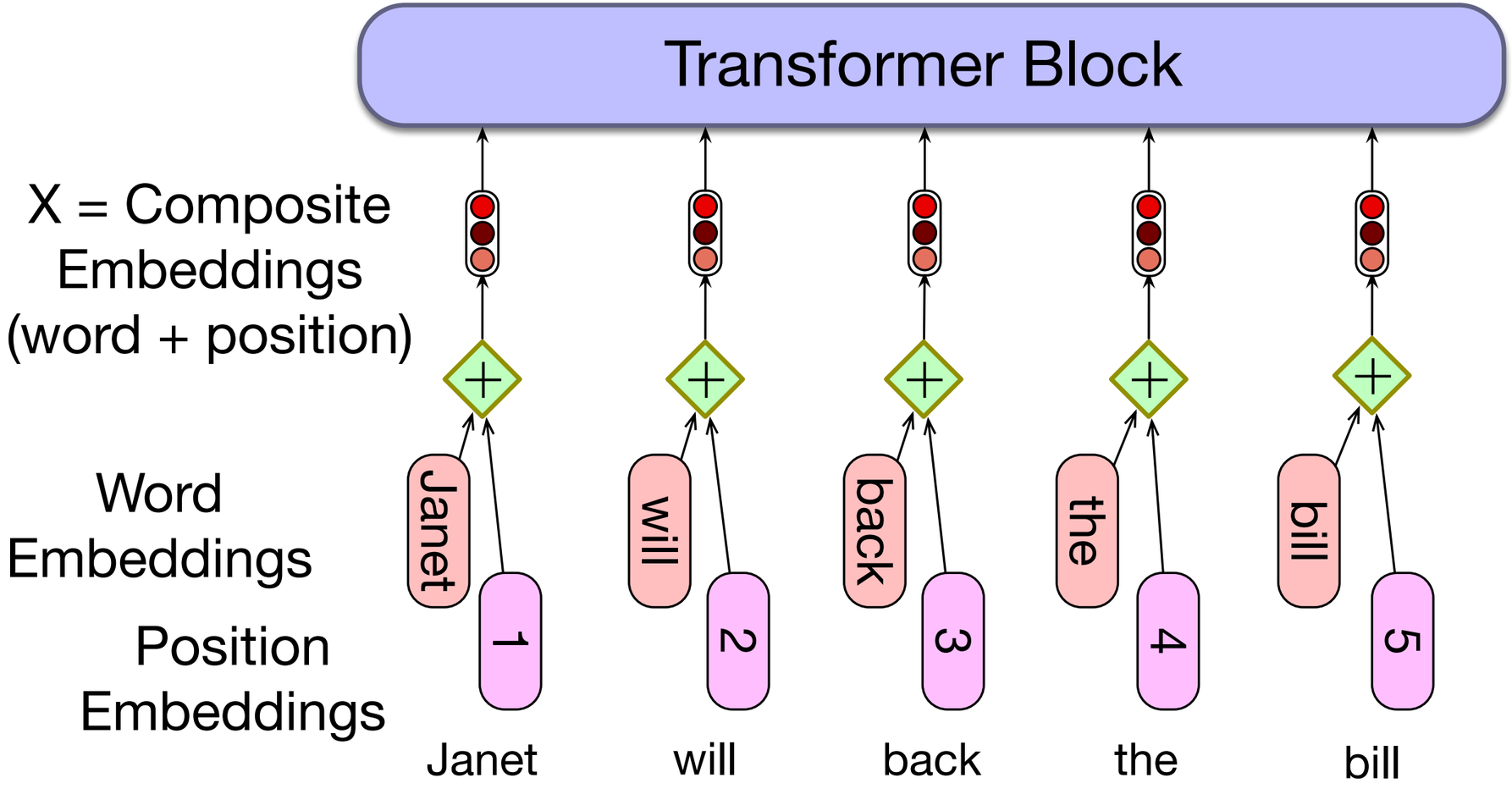
There are many methods, the simplest is absolute position.

Goal: learn a position embedding matrix E_{pos} 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

Each transformer block outputs a [1 x d] vector.

How do we turn it into a probability distribution over the vocabulary?

This is the job of the **language modeling head**

- **"Head"** means "extra circuits on top of a transformer"
- The job of the language modeling head:
  - Map from a [1 x d] vector
  - To a [1 x |V|] probability distribution over vocab

The job of the language modeling head:

- Map
  - From a [1 x d] vector
  - To a [1 x |V|] probability distribution over vocab

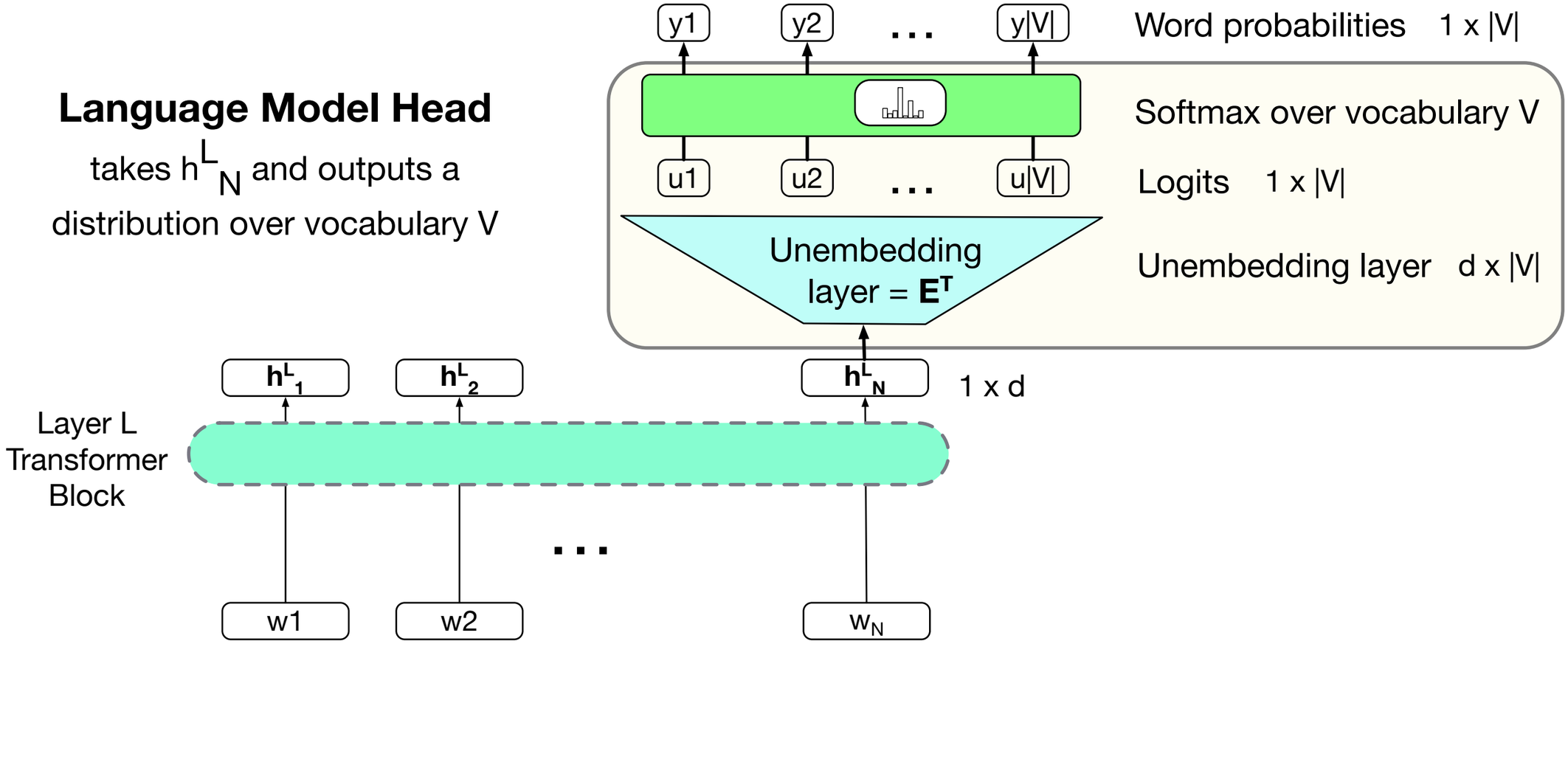
We do this in two steps

1. Use the "**unembedding matrix**" to map from [1 x d] vector to a [1 x |V|] vector of **logits**
2. Use **softmax** to map from a [1 x |V|] vector of logits to a a [1 x |V|] vector of probabilities

Language modeling head

Language Model Head

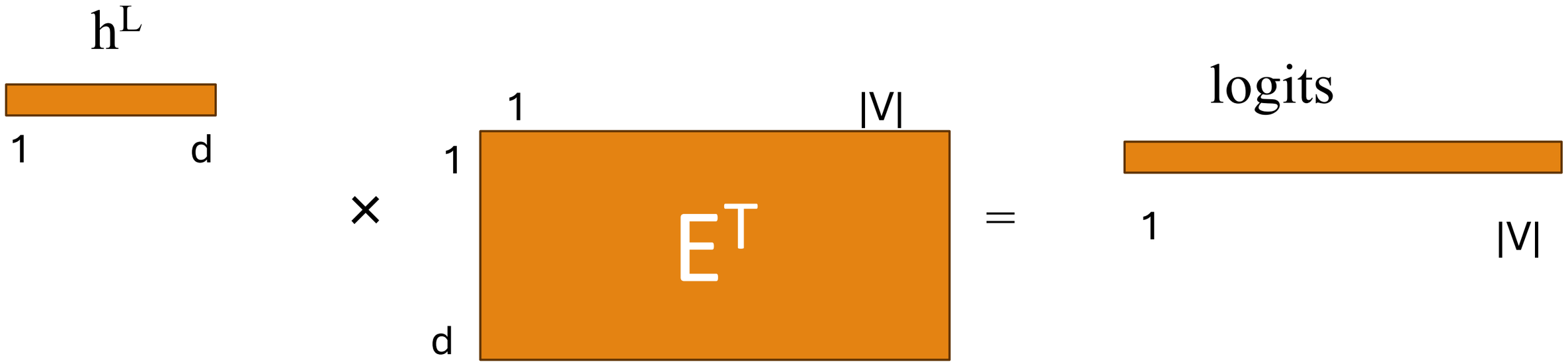
takes h^L_N and outputs a distribution over vocabulary V



Unembedding matrix E^T

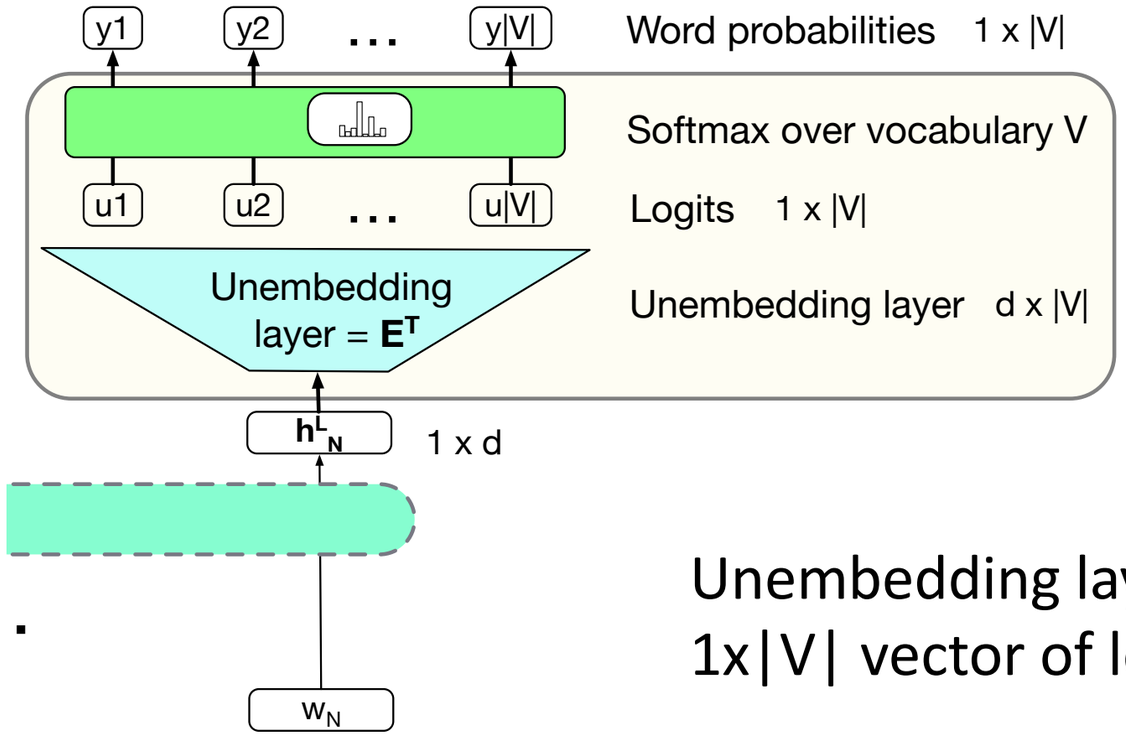
Embedding matrix E has shape [$|V| \times d$].
Unembedding matrix has shape [$d \times |V|$].

- One column for each of the $|V|$ tokens in the vocabulary.



Language modeling head

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

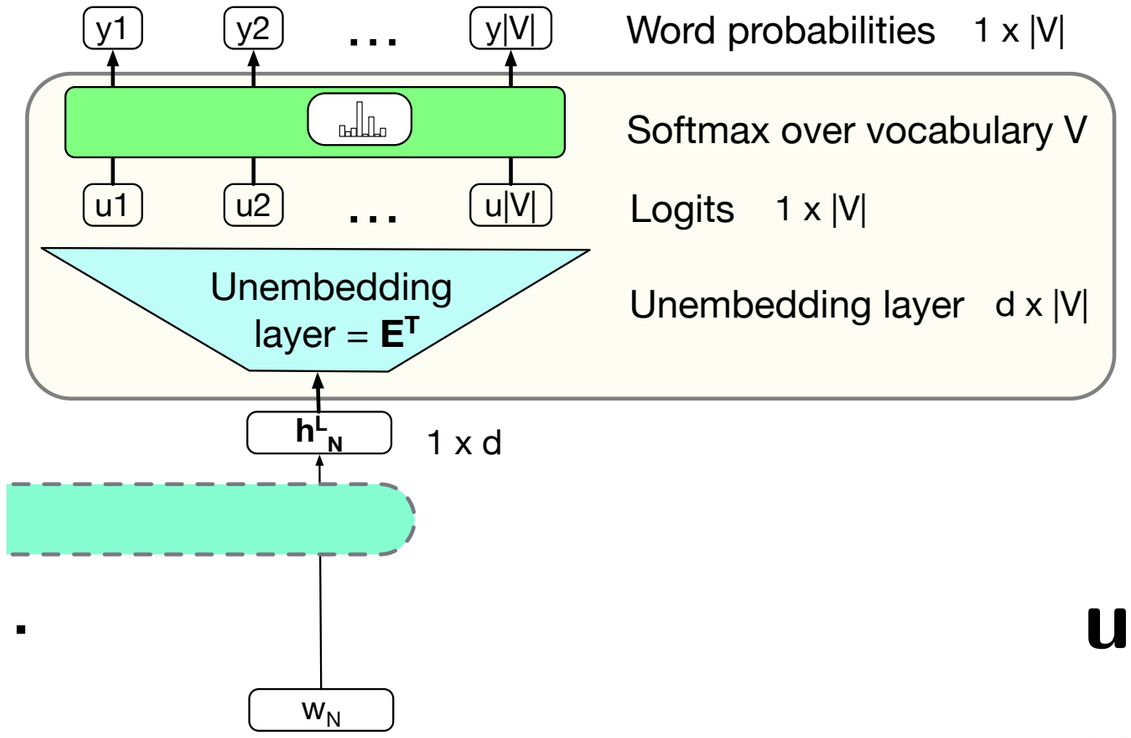


Why "unembedding"? **Tied** to $\mathbf{E}^T$

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

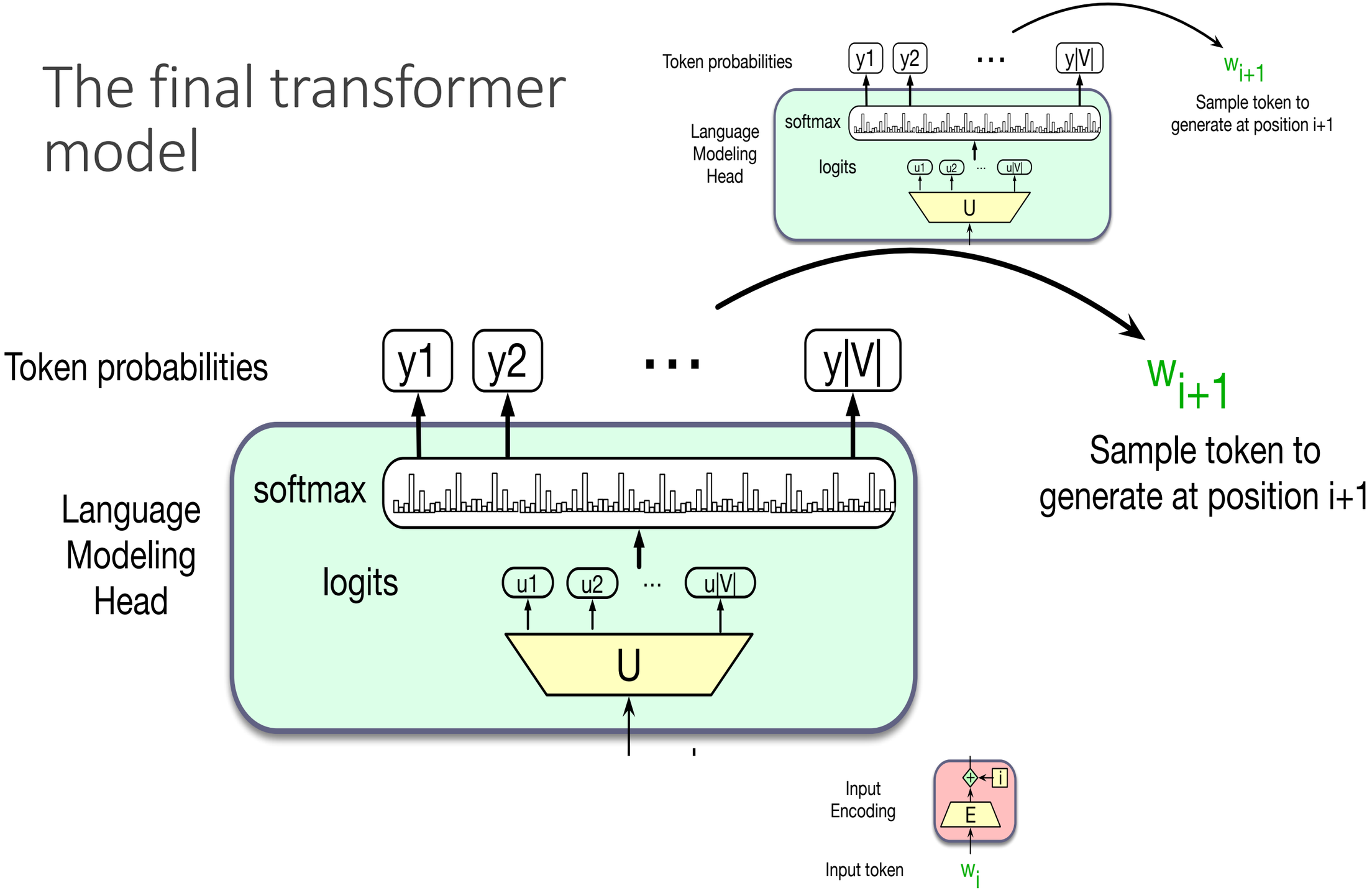
One score for each of the $|V|$ possible words in the vocabulary V. Shape $1 \times |V|$.

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

Pretraining Large Language Models

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.

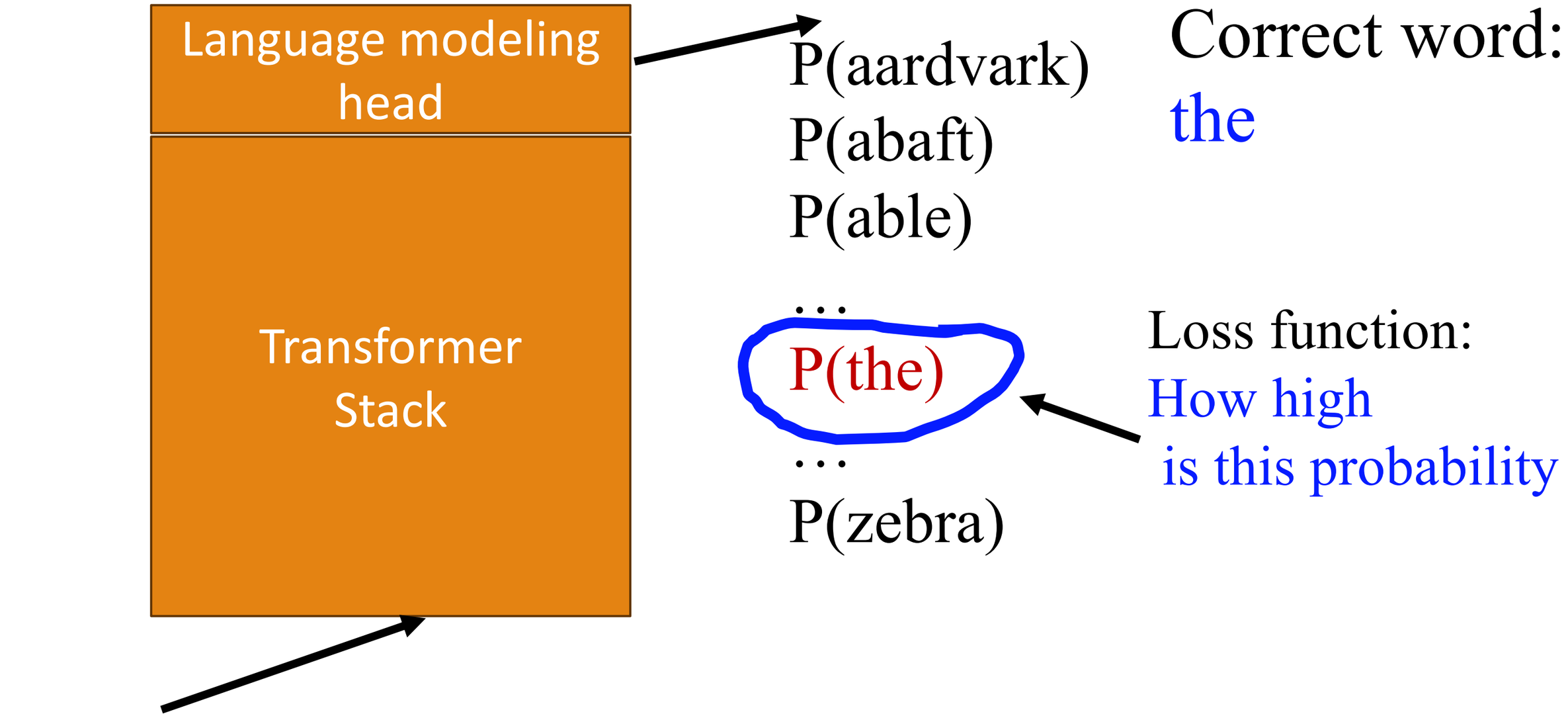
Self-supervised training algorithm

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!

The output of the LLM: Probability distribution over the vocabulary: possible next words



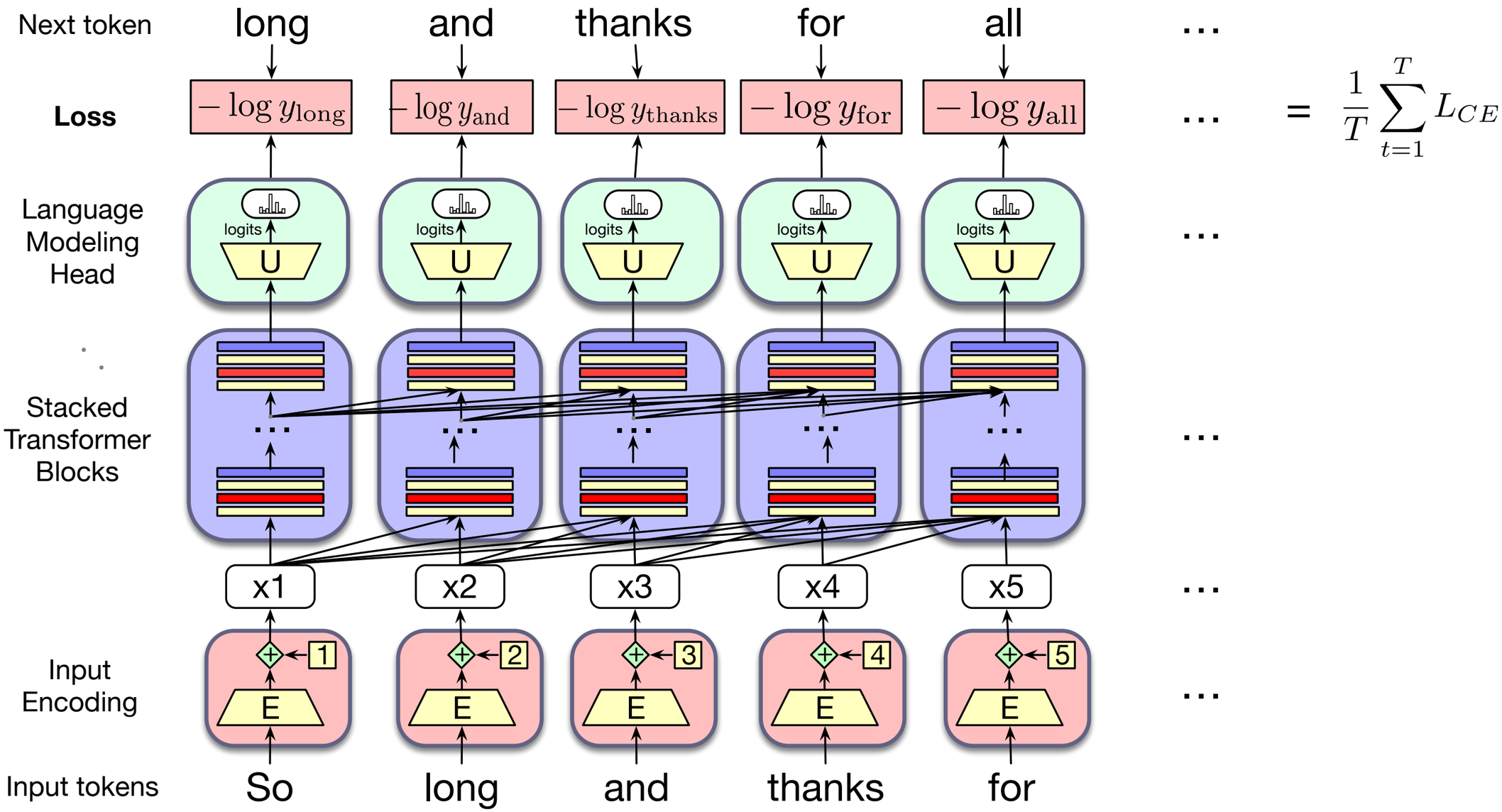
Cross-entropy loss

- Same loss from regression + neural nets
  - We want to assign a high probability to true word *w*
  - = high loss if model assigns too low a probability to w
- CE Loss : The negative log probability that the model assign to the true word w_t
- $L_{CE} = -\log p(w_i)$
- CE Loss for a whole sentence:
- $\frac{1}{T}\sum_{t=1}^{T} -\log p(w_t)$
- If loss is high (model assigns too low a probability to w)
  - We move model weights to give higher probability to w

Teacher forcing

- At each token position t, model sees correct tokens $w_{1:t}$,
  - Computes loss (neg log prob) 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



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

It's filtered to remove boilerplate, adult content, toxicity

What does a model learn from pretraining?

- There are canines everywhere! One dog in the front room, and two dogs
- It wasn't just big it was enormous
- The author of "A Room of One's Own" is Virginia Woolf
- The doctor told me that he
- The square root of 4 is 2

Big idea

Text contains enormous amounts of knowledge

Pretraining on lots of text with all that knowledge is what gives language models their ability to do so much

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

Pretraining Large Language Models: Algorithm and Data

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

Reminder: 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 *n* tokens $w_{1:n}$ the perplexity is :

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

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,∞]

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_2 emitted

Fairness

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

Large Language Models

Harms of Large Language Models

Hallucination

Chatbots May 'Hallucinate' More Often Than Many Realize

What Can You Do When A.I. Lies About You?

People have little protection or recourse when the technology creates and spreads falsehoods about them.

Air Canada loses court case after its chatbot hallucinated fake policies to a customer

The airline argued that the chatbot itself was liable. The court disagreed.

Copyright

Authors Sue OpenAI Claiming Mass Copyright Infringement of Hundreds of Thousands of Novels

The Times Sues OpenAI and Microsoft Over A.I. Use of Copyrighted Work

Millions of articles from The New York Times were used to train chatbots that now compete with it, the lawsuit said.

Privacy

Toxicity and Abuse

The New AI-Powered Bing Is Threatening Users.

Cleaning Up ChatGPT Takes Heavy Toll on Human Workers

Contractors in Kenya say they were traumatized by effort to screen out descriptions of violence and sexual abuse during run-up to OpenAI's hit chatbot

Fraud, Misinformation, Phishing

Italian tycoons targeted by fake defence minister in suspected AI scam

How AI is making phishing attacks more dangerous

Large Language Models

Harms of Large Language Models

Neural Networks

A quick walkthough of a backprop example

(that should help with PA6)

Intuition of neural net training (from Canvas lecture)

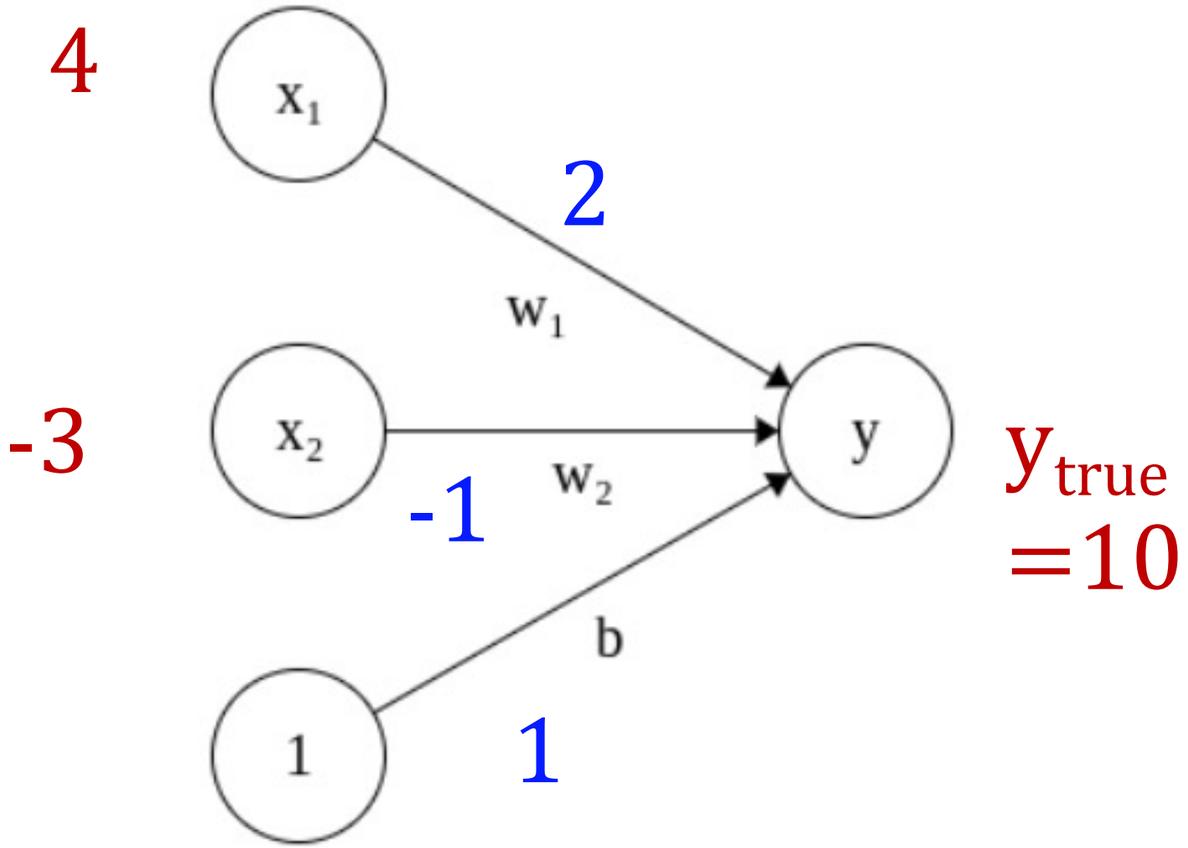
Goal: update network weights to minimize loss

For every training tuple (x, y_{true})

1. Run *forward* computation to find our estimate y
2. Compute loss L between y_{true} and the estimated y
3. Run *backward* computation; update weights to decrease L
   - For each weight w in the network
     - Assess how much blame it deserves for the current loss: $\frac{\partial L}{\partial w}$
       - (Using chain rule to compute partial derivatives)
     - Update w: $w_{new} = w - \eta \frac{\partial L}{\partial w}$ Where $\eta = .01$

New network will have lower loss on example

Example for today: simple 1-layer network, no activation function, squared loss



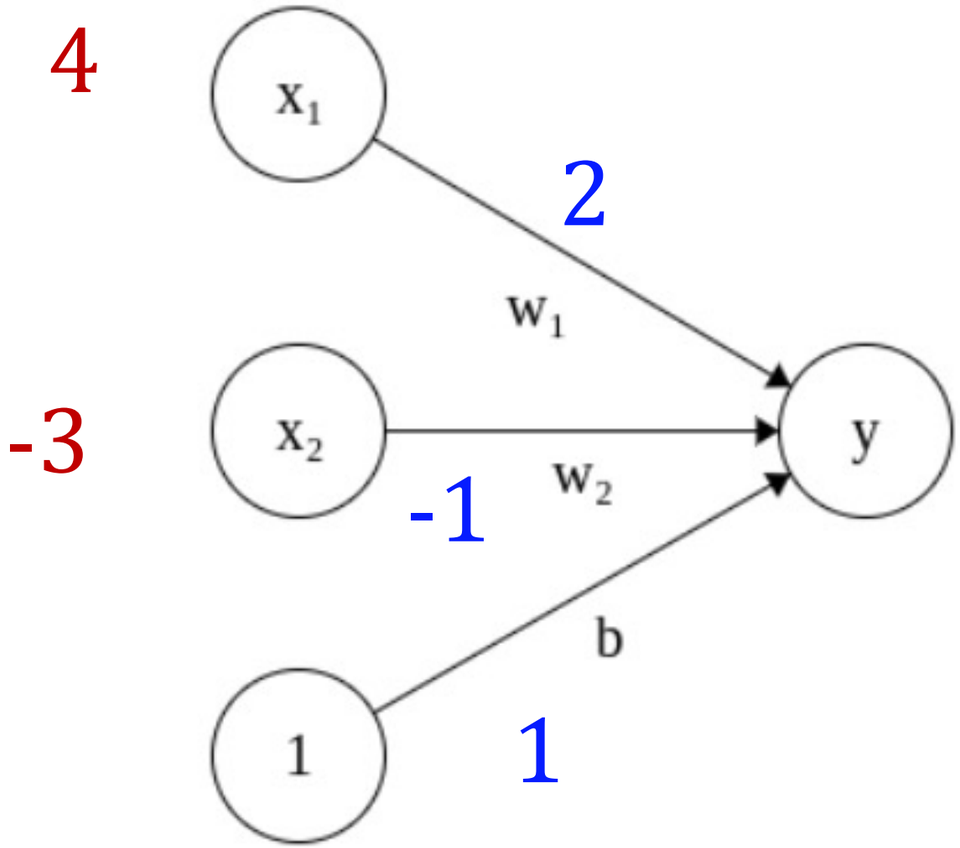
$y = w_1 x_1 + w_2 x_2 + b$

$L = (y_{true} - y)^2$

Initial weights:
$w_1 = 2$, $w_2 = -1$, $b = 1$

We'll train on one example:
$(x_1, x_2, y_{true}) = (4, -3, 10)$

1. Forward pass: compute y



$y=w_1x_1+w_2x_2+b$

Initial weights:

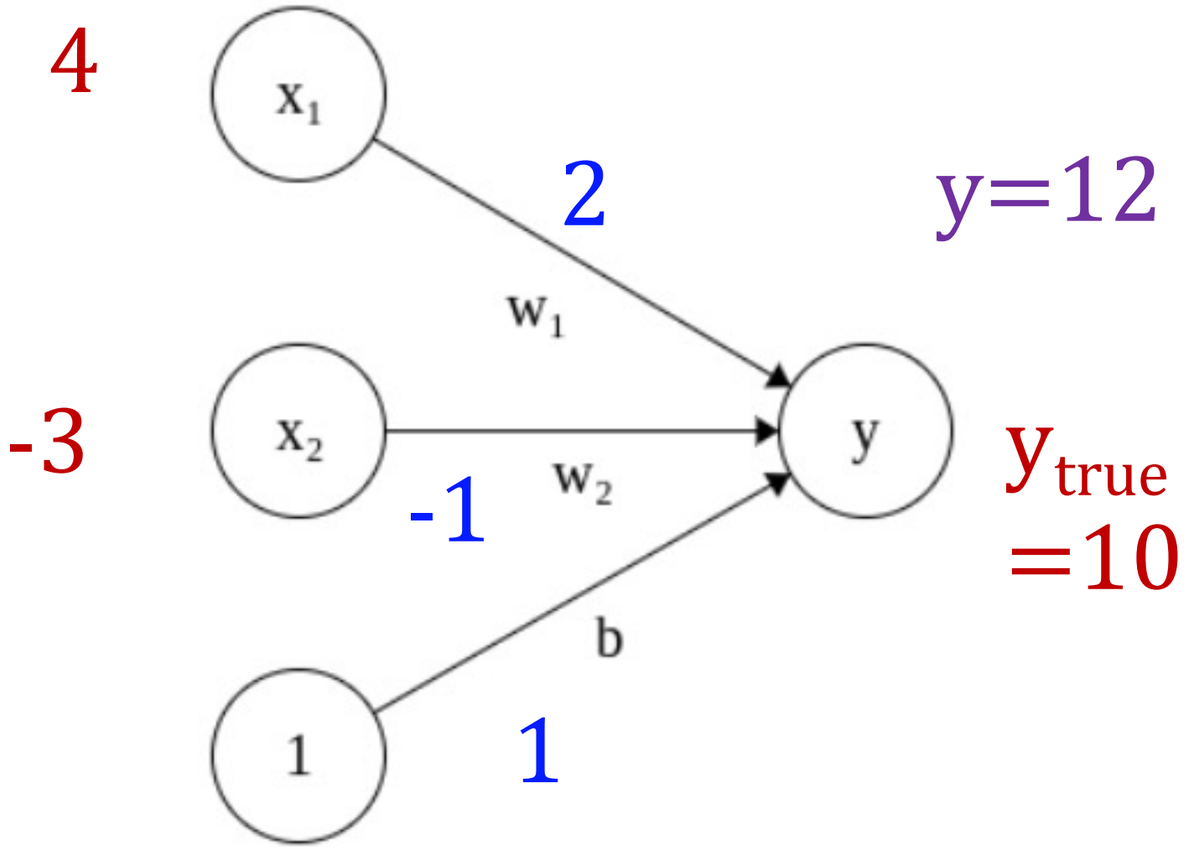
$w_1=2$, $w_2=-1$, $b=1$

Input:

$(x_1, x_2) = (4, -3)$

$y = 2*4 + -1*-3 + 1 = 12$

2. Compute Loss



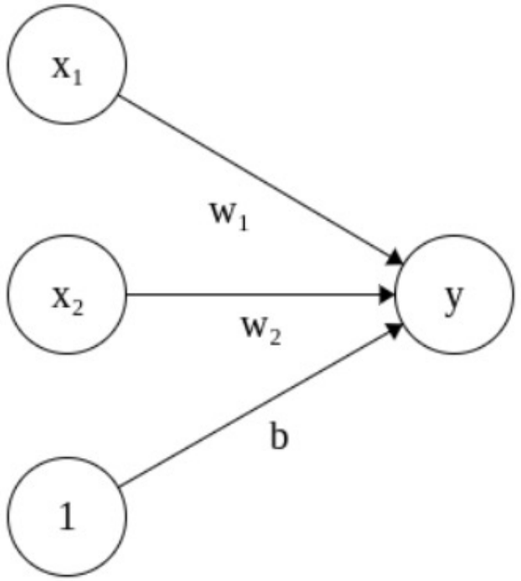
$$y=w_1x_1+w_2x_2+b$$
$$=12$$

$$L=(y_{true}-y)^2$$
$$=(10-12)^2=4$$

3. Backward pass: Create computation graph

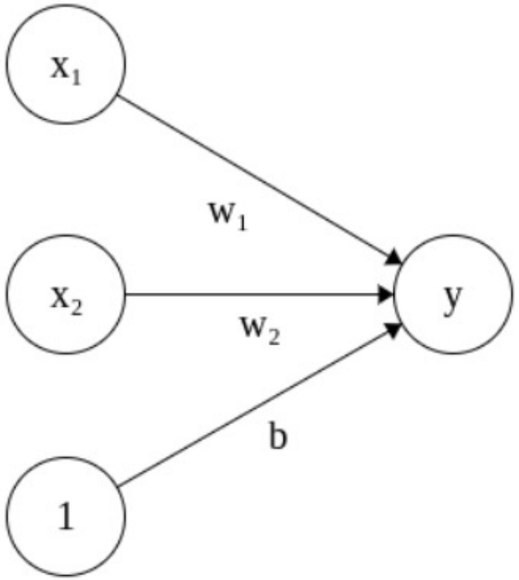
First: make computation graph including intermediate variables

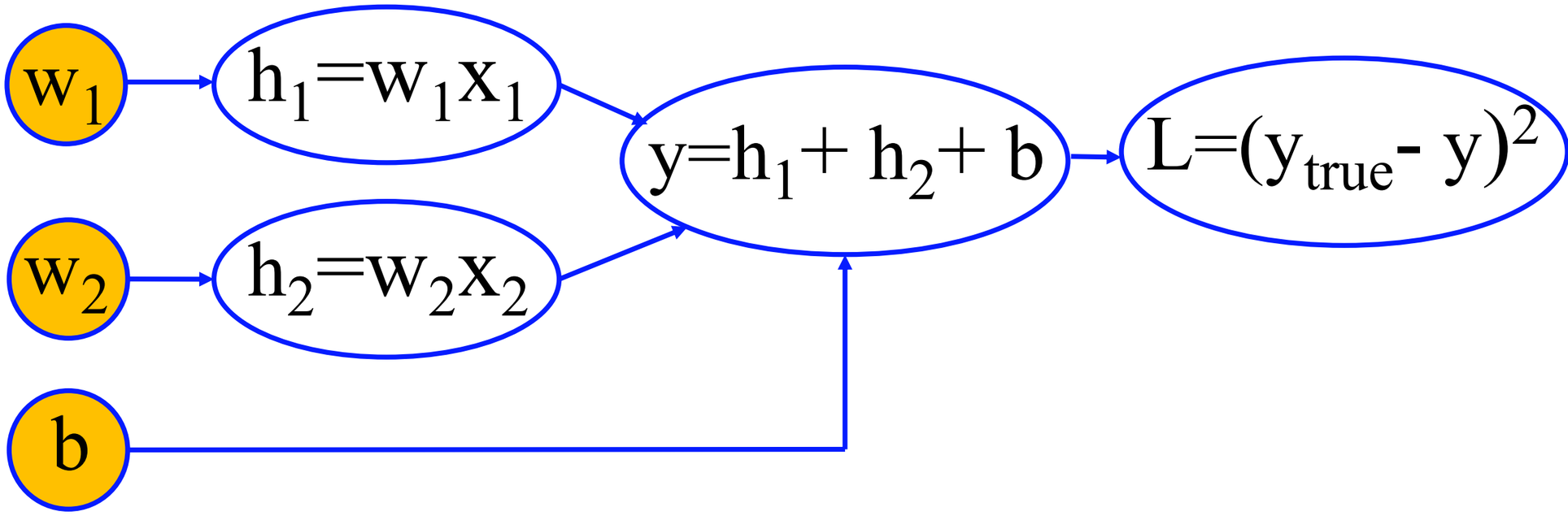
The key nodes are the parameters of the model that we need gradients to update



3. Backward pass: Create computation graph

We're going to make nodes for w_1, w_2, b, which we need to update
Plus we'll need the nodes between them and the loss L
We'll create 2 convenient intermediate nodes $h_1=w_1x_1$ and $h_2=w_2x_2$
Why not x_1, x_2?

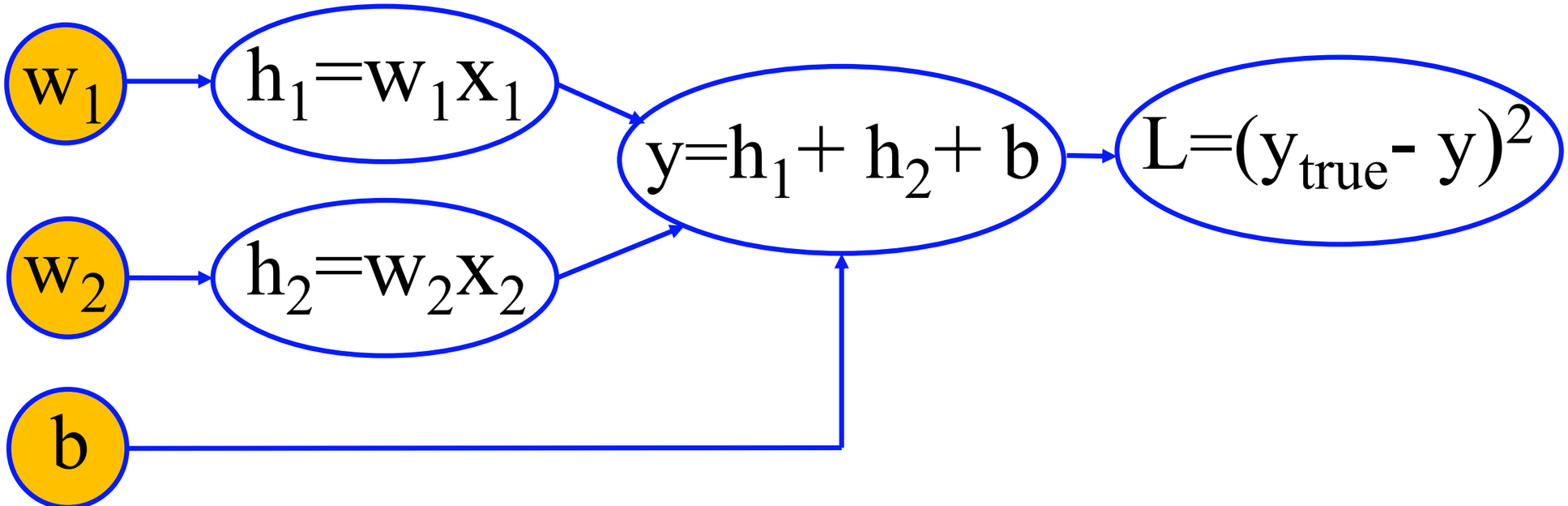




3. Backprop: compute loss gradients for weights

We want $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b}$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial h_1}\frac{\partial h_1}{\partial w_1}$$



3. Backprop: compute loss gradients for weights

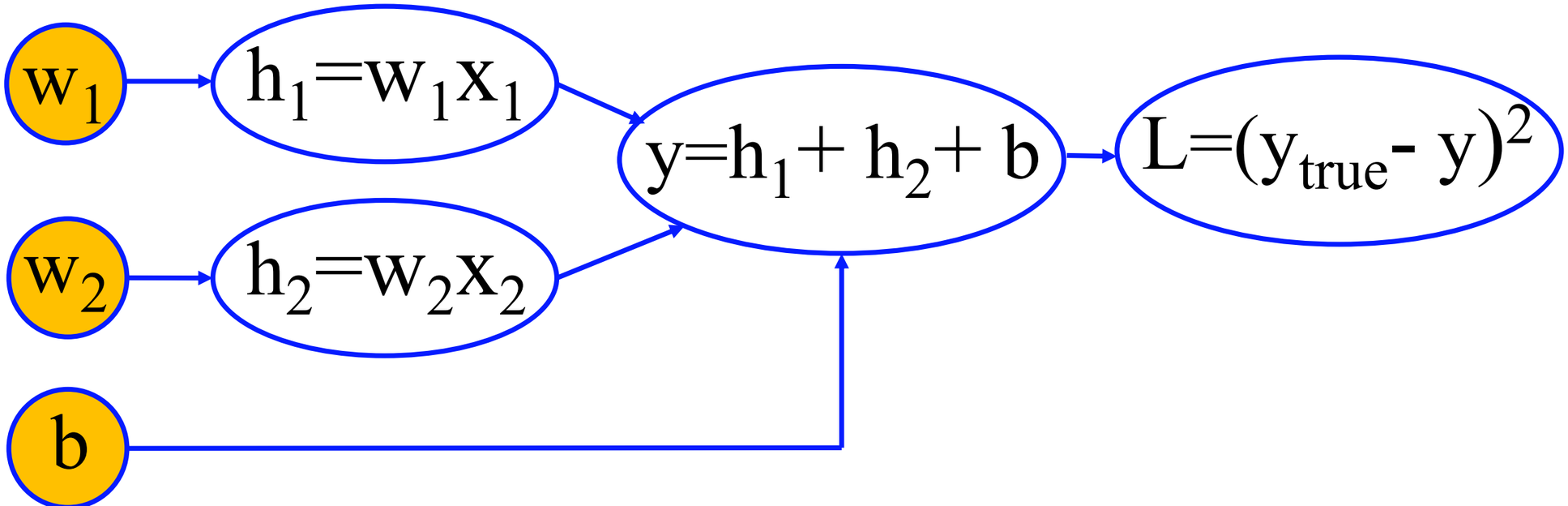
We want $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b}$

$$L=(y_{true}-y)^2$$

$$\frac{\partial L}{\partial w_1}=\frac{\partial L}{\partial y}\frac{\partial y}{\partial h_1}\frac{\partial h_1}{\partial w_1}$$

$$y = h_1 + h_2 + b$$

$$h_1 = w_1 x_1$$



3. Backprop: compute loss gradients for weights

We want $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b}$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial h_1} \frac{\partial h_1}{\partial w_1}$$

$$L = (y_{true} - y)^2$$

$$y = h_1 + h_2 + b$$

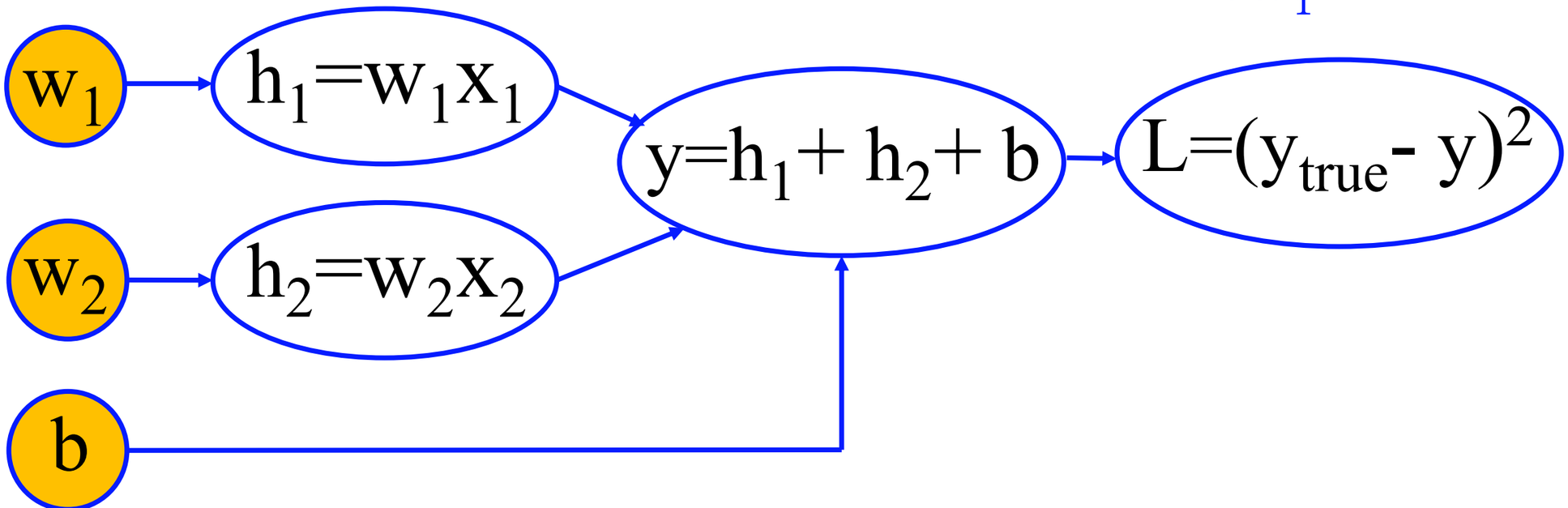
$$h_1 = w_1 x_1$$

$$\frac{\partial L}{\partial y} = 2\,(y_{true} - y) * -1$$

$$= 2y - 2y_{true}$$

$$\frac{\partial y}{\partial h_1} = 1$$

$$\frac{\partial h_1}{\partial w_1} = x_1$$



3. Backprop: compute loss gradients for weights

We want $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b}$

$$L=(y_{true}-y)^2$$

$$\frac{\partial L}{\partial y} = 2\,(y_{true}-y) * -1$$

$$= 2(y - y_{true}) = 4$$

$$y = h_1 + h_2 + b$$

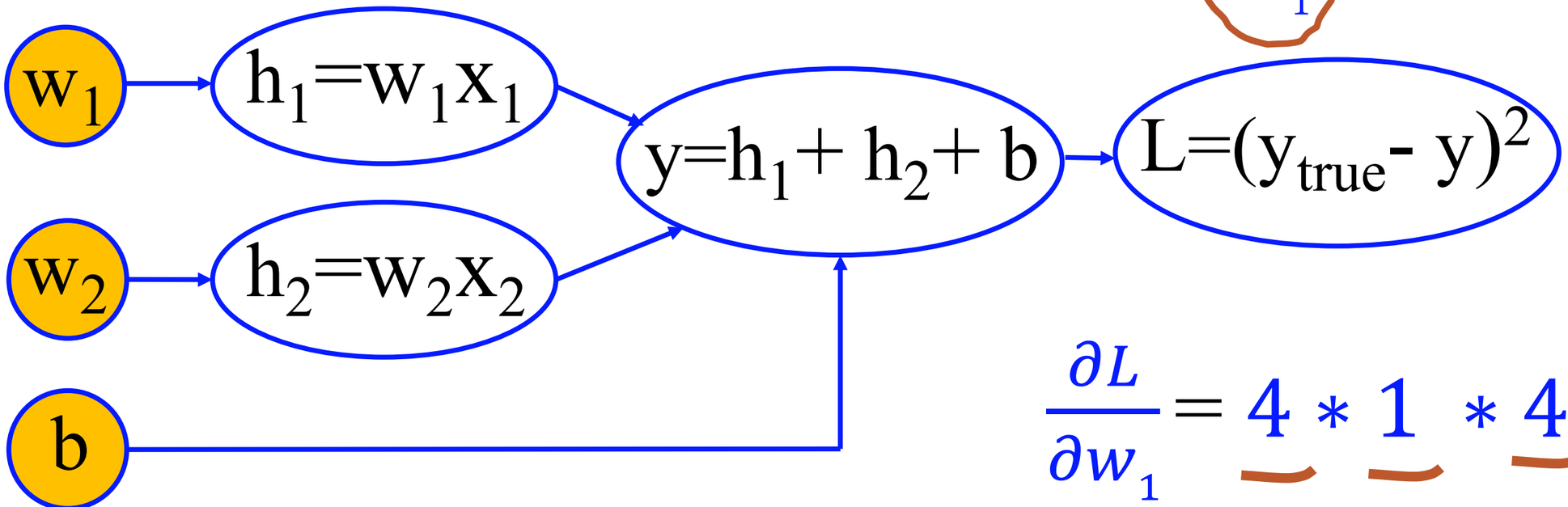
$$\frac{\partial y}{\partial h_1} = 1$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y}\,\frac{\partial y}{\partial h_1}\,\frac{\partial h_1}{\partial w_1}$$

$$h_1 = w_1 x_1$$

$$\frac{\partial h_1}{\partial w_1} = x_1 = 4$$

$x_1 = 4$
$x_2 = -3$
$y = 12$
$y_{true} = 10$
$w_1 = 2$
$w_2 = -1$
$b = 1$



$$\frac{\partial L}{\partial w_1} = 4 * 1 * 4 = 16$$

3. Backprop: compute loss gradients for weights

We want $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b}$

$$L=(y_{true}-y)^2$$

$$\frac{\partial L}{\partial y} = 2\,(y_{true}-y) * -1$$

$$= 2(y - y_{true}) = 4$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial y}\,\frac{\partial y}{\partial h_2}\,\frac{\partial h_2}{\partial w_2}$$

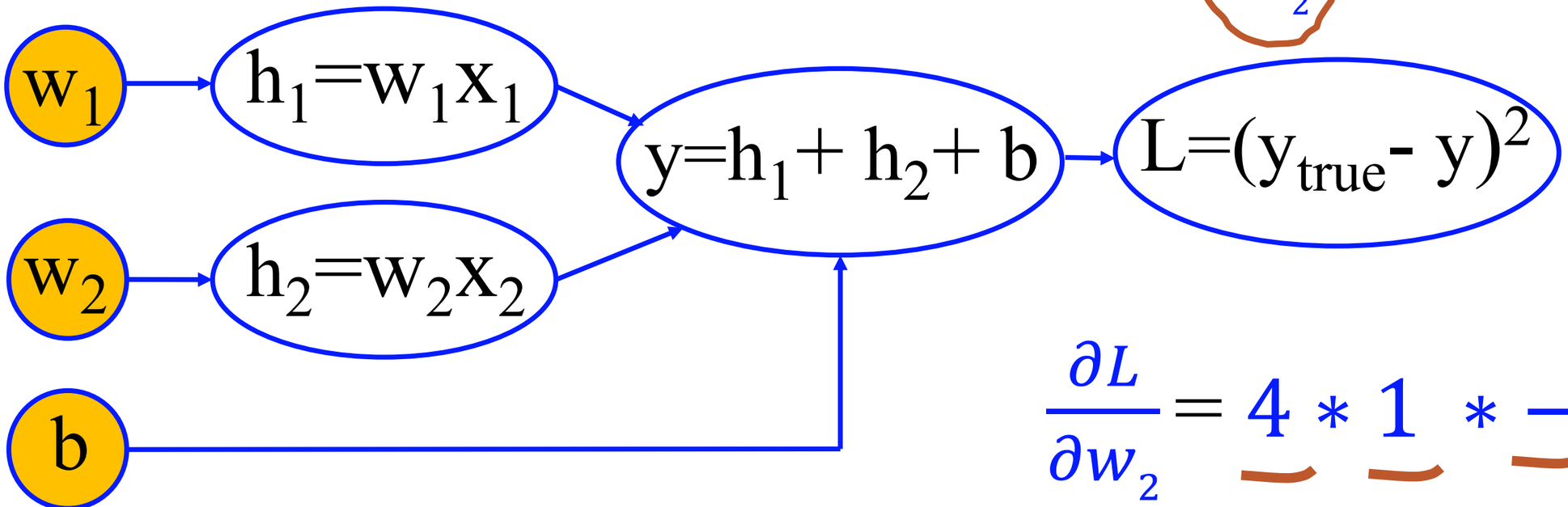
$$y = h_1 + h_2 + b$$

$$\frac{\partial y}{\partial h_2} = 1$$

$$h_2 = w_2 x_2$$

$$\frac{\partial h_1}{\partial w_2} = x_2 = -3$$

$x_1 = 4$
$x_2 = -3$
$y = 12$
$y_{true} = 10$
$w_1 = 2$
$w_2 = -1$
$b = 1$



$$\frac{\partial L}{\partial w_2} = 4 * 1 * -3 = -12$$

3. Backprop: compute loss gradients for weights

We want $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \frac{\partial L}{\partial b}$

$$L=(y_{true}-y)^2$$

$$\frac{\partial L}{\partial y}= 2\,(y_{true}-y) * -1$$

$$= 2(y - y_{true})= 4$$

$$y = h_1 + h_2 + b$$

$$\frac{\partial y}{\partial b}=1$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial y}\,\frac{\partial y}{\partial b}$$

$x_1 = 4$

$x_2 = -3$

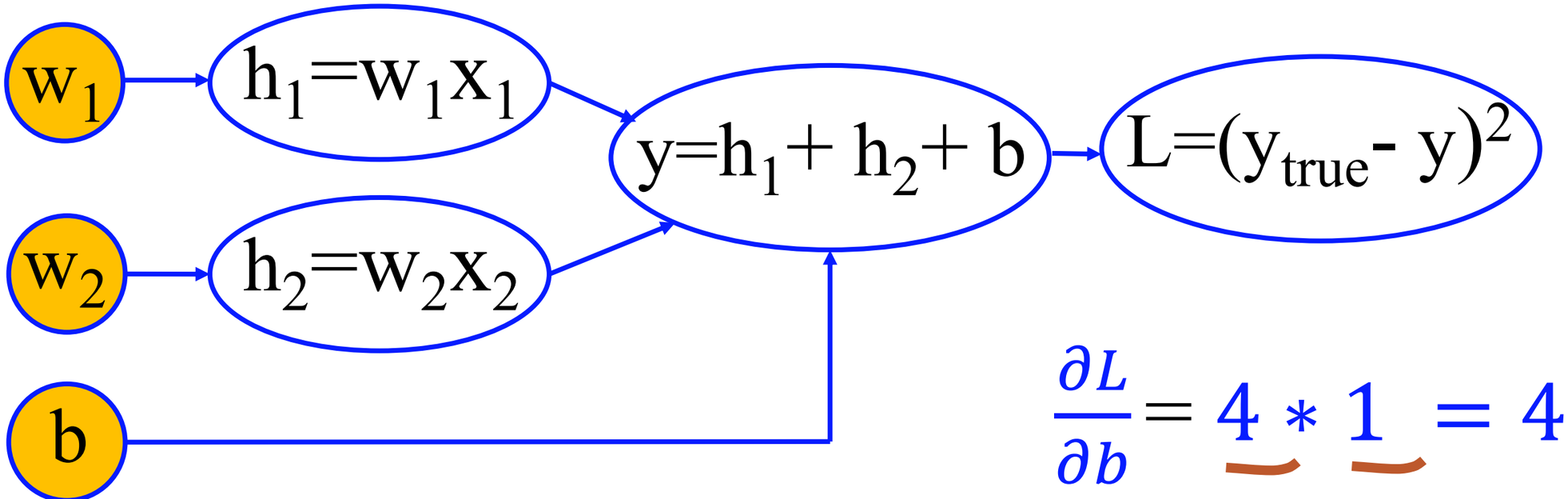
$y = 12$

$y_{true} = 10$

$w_1=2$

$w_2=-1$

$b=1$



$$\frac{\partial L}{\partial b} = 4 * 1 = 4$$

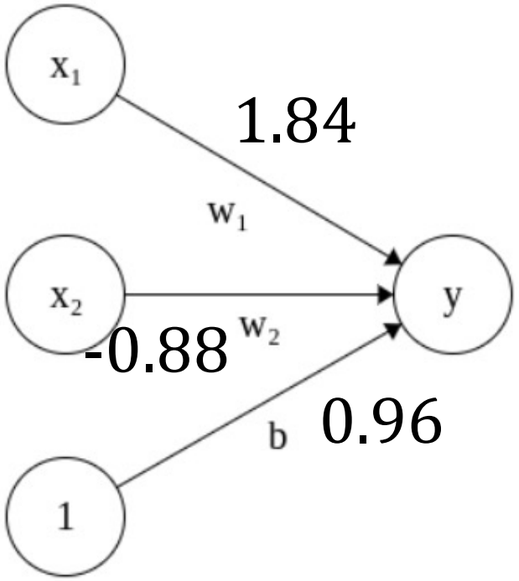
3. Backprop: compute new weights

$$\frac{\partial L}{\partial w_1} = 16 \qquad w_1 = w_1 - \eta \frac{\partial L}{\partial w_1} = 2 - 0.01 * 16 = 1.84$$

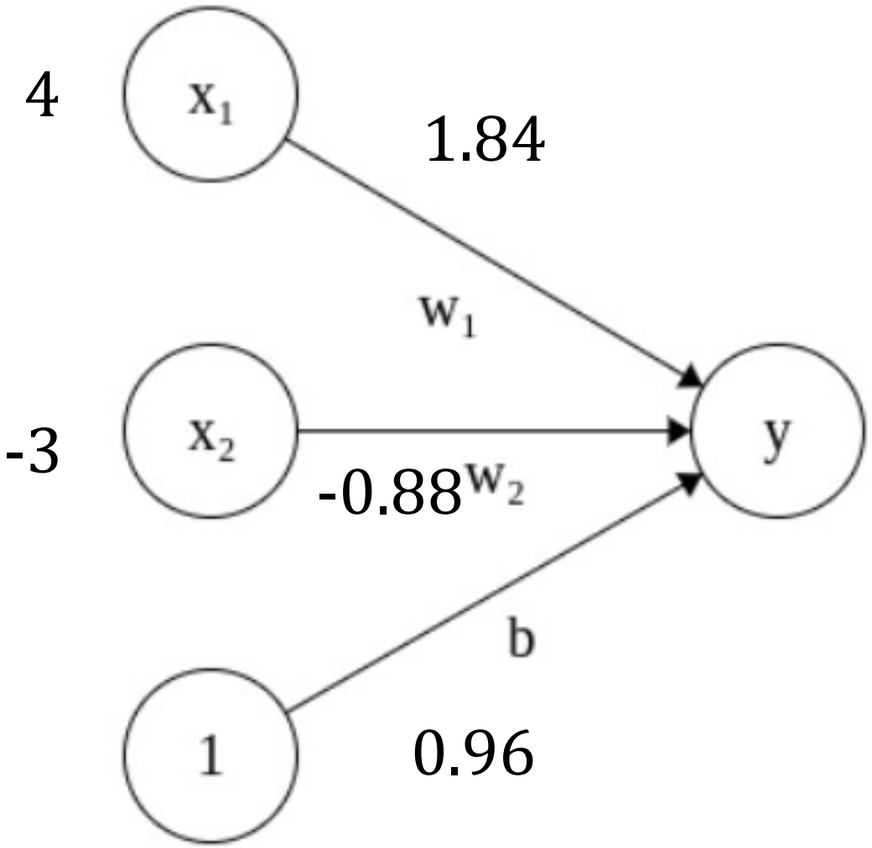
$$\frac{\partial L}{\partial w_2} = -12 \qquad w_2 = w_2 - \eta \frac{\partial L}{\partial w_2} = -1 - 0.01 * -12 = -0.88$$

$$\frac{\partial L}{\partial b} = 4 \qquad b = b - \eta \frac{\partial L}{\partial b} = 1 - 0.01 * 4 = 0.96$$

$x_1 = 4$
$x_2 = -3$
$y = 12$
$y_{true} = 10$
$w_1 = 2$
$w_2 = -1$
$b = 1$
$\eta = .01$



If we rerun forward pass, result should be closer to $y_{true}=10$



$$y=w_1x_1+w_2x_2+b$$
$$= 10.96$$

Success!!!

Neural Networks

A backprop example

Transformers

Advanced: (I won't get to this on Tuesday) Multihead Attention and Parallelizing the Attention Computation

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

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

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

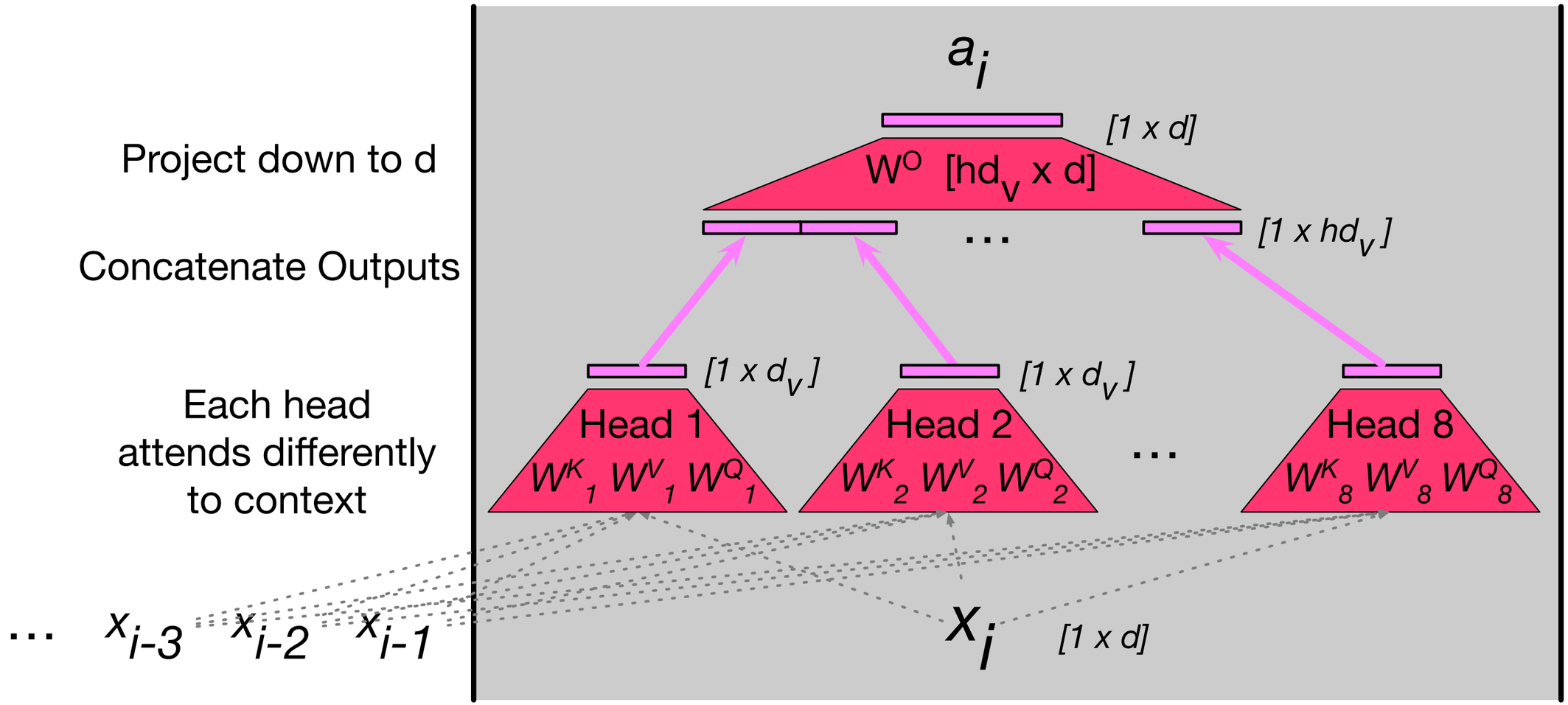
$$\alpha_{ij}^c = \text{softmax}(\text{score}^c(\mathbf{x}_i, \mathbf{x}_j)) \;\; \forall j \le i$$

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

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

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

Multi-head attention



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

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

$$\mathbf{A} = \mathbf{head}\,\mathbf{W}^{\mathbf{O}}$$

Masking out the future

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

$$\mathbf{A} = \mathbf{head}\,\mathbf{W}^O$$

- 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

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

$$\mathbf{A} = \mathbf{head}\,\mathbf{W}^O$$

Add $-\infty$ to cells in upper triangle

The softmax will turn it to 0

N

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

N

Another point: Attention is quadratic in length

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

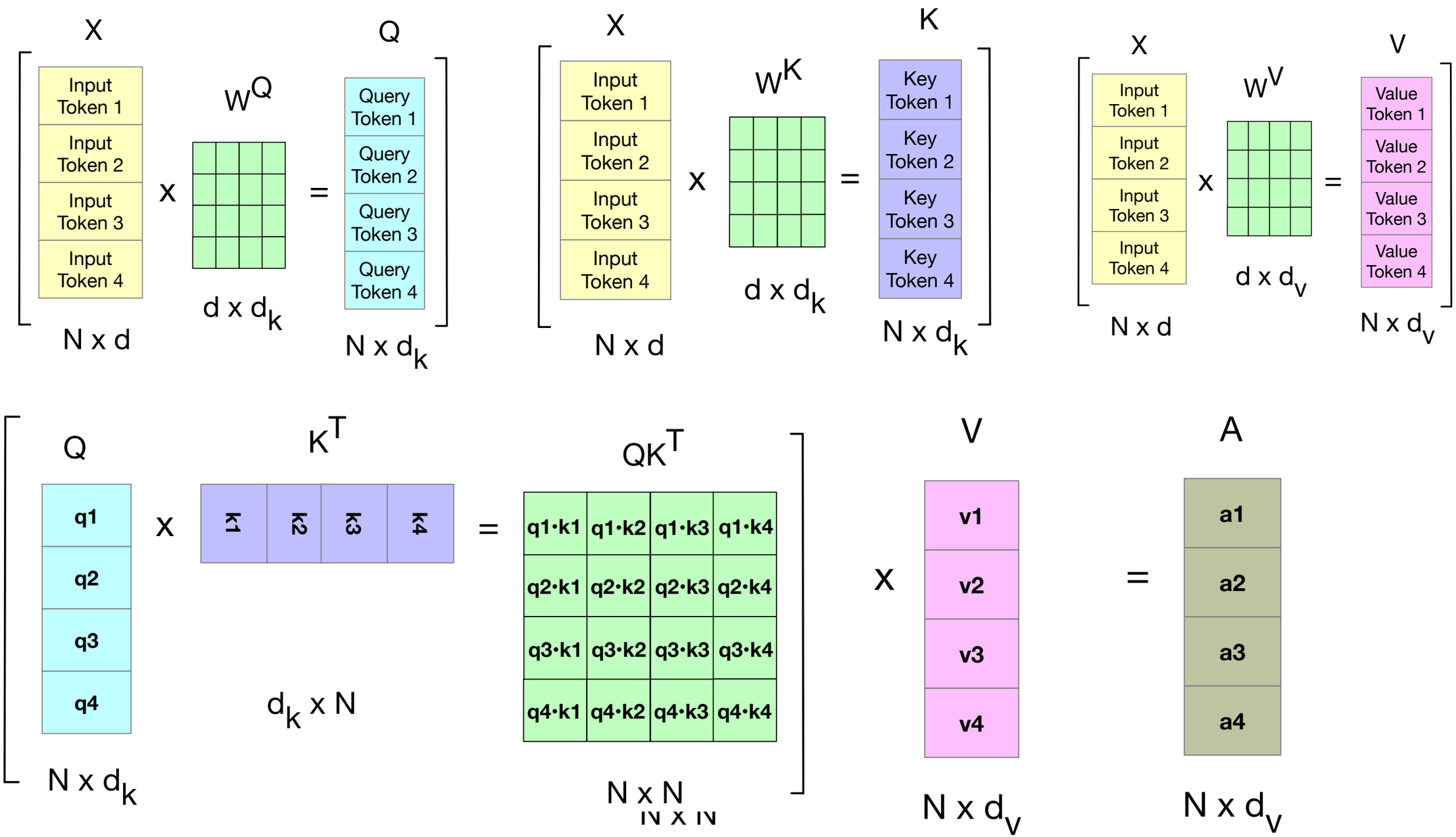
$$\mathbf{A} = \mathbf{head}\,\mathbf{W}^O$$

N

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

N

Attention again



Parallelizing Multi-head Attention

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

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

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

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

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

Transformers

Parallelizing Attention Computation