CSCI567 Machine Learning (Fall 2025)

Haipeng Luo

University of Southern California

Nov 7, 2025

Administration

Will discuss HW3 solutions in today's discussion session.

HW4 (last homework) will be released soon.

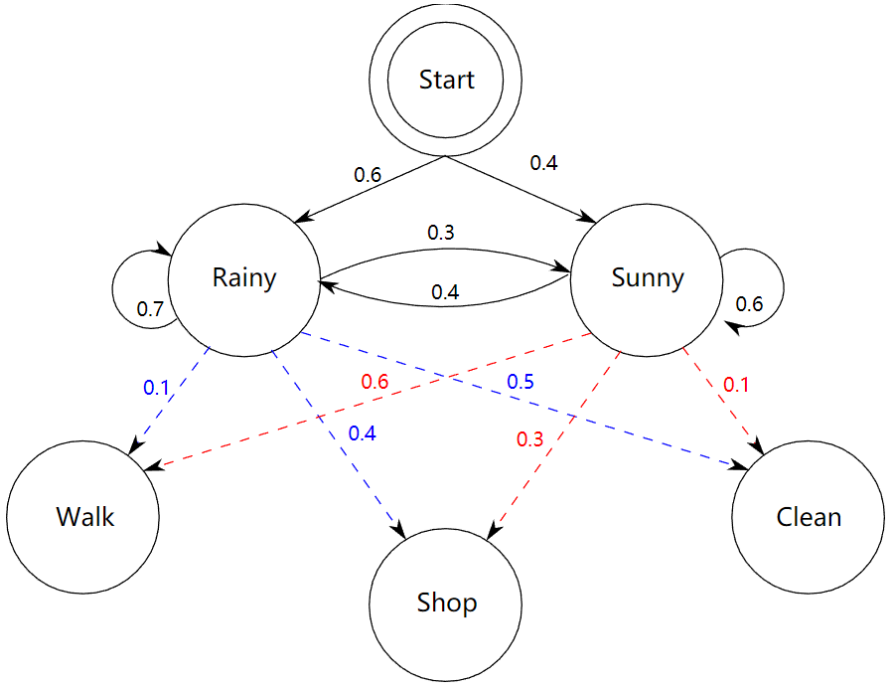
Outline

Outline

Hidden Markov Models

Model parameters:

- **initial distribution**
  $P(Z_1 = s) = \pi_s$
- **transition distribution**
  $P(Z_{t+1} = s' \mid Z_t = s) = a_{s,s'}$
- **emission distribution**
  $P(X_t = o \mid Z_t = s) = b_{s,o}$



Baum–Welch algorithm

Step 0 Initialize the parameters $(\boldsymbol{\pi}, \boldsymbol{A}, \boldsymbol{B})$

Step 1 (E-Step) Fixing the parameters, compute forward and backward messages for all sample sequences, then use these to compute $\gamma_s^{(n)}(t)$ and $\xi_{s,s'}^{(n)}(t)$ for each n, t, s, s'.

Step 2 (M-Step) Update parameters:

$$\pi_s \propto \sum_n \gamma_s^{(n)}(1), \quad a_{s,s'} \propto \sum_n \sum_{t=1}^{T-1} \xi_{s,s'}^{(n)}(t), \quad b_{s,o} \propto \sum_n \sum_{t:x_t=o} \gamma_s^{(n)}(t)$$

Step 3 Return to Step 1 if not converged

Viterbi Algorithm

Viterbi Algorithm

For each $s \in [S]$, compute $\delta_s(1) = \pi_s b_{s,x_1}$.

For each $t = 2, \ldots, T$,

- for each $s \in [S]$, compute

$$\delta_s(t) = b_{s,x_t} \max_{s'} a_{s',s} \delta_{s'}(t-1)$$

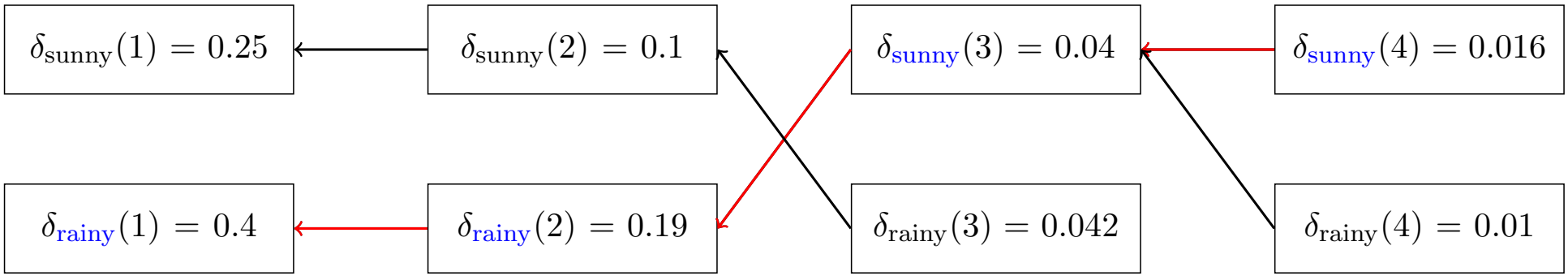
$$\Delta_s(t) = \operatorname{argmax}_{s'} a_{s',s} \delta_{s'}(t-1)$$

Backtracking: let $z_T^* = \operatorname{argmax}_s \delta_s(T)$.
For each $t = T, \ldots, 2$: set $z_{t-1}^* = \Delta_{z_t^*}(t)$.

Output the most likely path $z_1^*, \ldots, z_T^*$.

Example

Arrows represent $\Delta_s(t)$, backtracking = follow the arrows.



The most likely path is **"rainy, rainy, sunny, sunny".**

Outline

Recall: language models via HMM

- today the _____
- today the price _____
- today the price of _____
- today the price of gold _____

company	0.153
bank	0.153
price	0.077
italian	0.039
emirate	0.039
…	

of	0.308
for	0.050
it	0.046
to	0.046
is	0.031
…	

the	0.072
18	0.043
oil	0.043
its	0.036
gold	0.018
…	

final result: *today the price of gold per ton, while production of shoe lasts and shoe industry, the bank intervened just after it considered and rejected an IMF demand to rebuild depleted European stocks.*

Surprisingly grammatical! but *incoherent*...

How to improve this?

Key ideas for improvement:

- represent **words as vectors**, enabling differentiable operations
- flexible **sequence to sequence** architecture (not just vector to vector)
- **shared components** (just like filters in CNN)

Recurrent Neural Network (RNN) is one solution (popular before transformers).

Acknowledgements

Very useful resources:

- RNN cheatsheet from Stanford CS 230
  - `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks`
- Visualizing a tiny RNN
  - `https://joshvarty.github.io/VisualizingRNNs/`
- Character-level RNN
  - `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`

Words as vectors

Simplest approach: **one-hot sparse encoding**

- suppose there are d words in the vocabulary
- represent the i-th of them by the d-dimensional basis vector

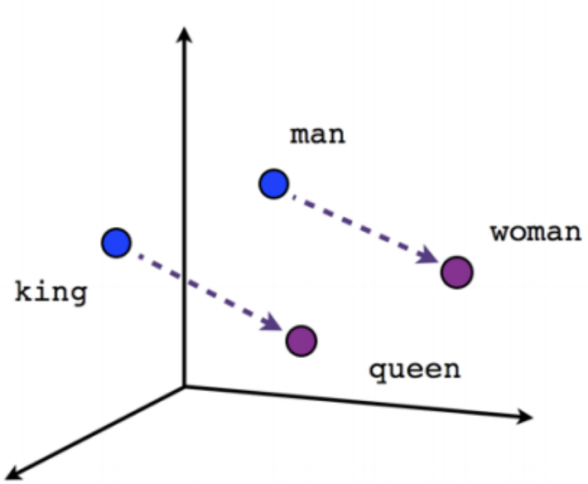
$$e_i = (0, \ldots, 0, \underset{\substack{\uparrow \\ i\text{-th entry}}}{1}, 0, \ldots, 0) \in \mathbb{R}^d$$

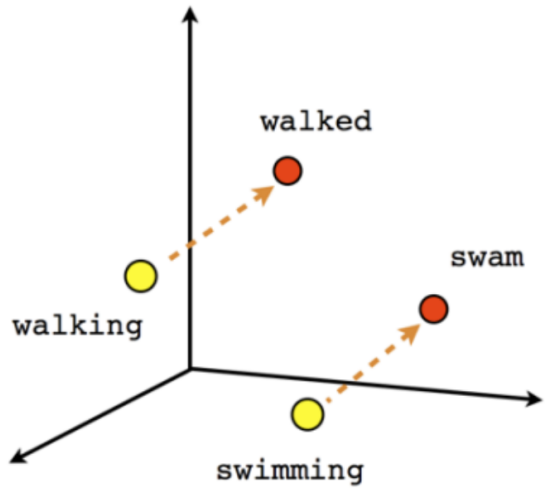
Issues: does not convey any semantic meanings

Words as vectors: embedding

Word embedding: similar words are closer in their vector representation

- popular approaches: **word2vec**, **GloVe** (see project)
- can even perform meaningful algebraic operations
  - example: *man is to woman as king is to?*
  - can be answered by finding the word with the closest embedding to $\text{vec}(\text{woman}) - \text{vec}(\text{man}) + \text{vec}(\text{king})$, which happens to be "queen"





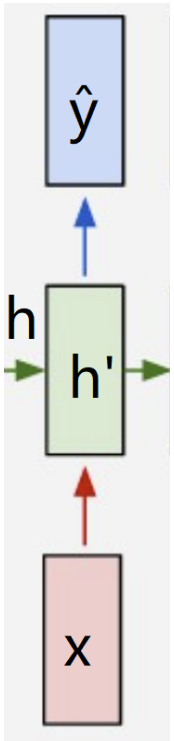
Unifying two approaches

Let $\boldsymbol{x} \in \mathbb{R}^d$ be the one-hot encoding of a word, and matrix $\boldsymbol{E} \in \mathbb{R}^{d_e \times d}$ be some **embedding matrix**, then $\boldsymbol{E}\boldsymbol{x}$ is the embedding for this word

- $\boldsymbol{E}$ can be fixed (i.e., from word2vec or GloVe), where the i-th column is the embedding for the i-th word
- or $\boldsymbol{E}$ can be *learned* (e.g., via backpropagation in DL pipeline), making it application specific (common especially if data are huge)

In the remaining, we simply use one-hot representation, but keep in mind it could be passed through some $\boldsymbol{E}$ implicitly

A recurrent layer



from $\widehat{\boldsymbol{y}} = \boldsymbol{f}(\boldsymbol{x})$ to $(\widehat{\boldsymbol{y}}, \boldsymbol{h}') = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{h})$

- $\boldsymbol{h}$ is "hidden state" (like HMM), updated via

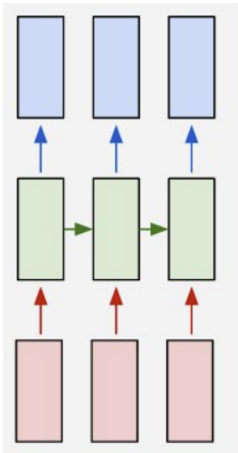
$$\boldsymbol{h}' = \boldsymbol{\sigma}(\boldsymbol{W}\boldsymbol{h} + \boldsymbol{U}\boldsymbol{x} + \boldsymbol{b}_h)$$

  where $\boldsymbol{\sigma}$ is an activation function

- $\widehat{\boldsymbol{y}} = \boldsymbol{V}\boldsymbol{h}' + \boldsymbol{b}_y$ is the output
- $\boldsymbol{x}, \widehat{\boldsymbol{y}} \in \mathbb{R}^d, \boldsymbol{h}, \boldsymbol{h}' \in \mathbb{R}^{d_h}$
- weight matrices $\boldsymbol{W} \in \mathbb{R}^{d_h \times d_h}, \boldsymbol{U} \in \mathbb{R}^{d_h \times d}, \boldsymbol{V} \in \mathbb{R}^{d \times d_h}$
- bias terms $\boldsymbol{b}_h \in \mathbb{R}^{d_h}$, $\boldsymbol{b}_y \in \mathbb{R}^d$

Recurrent layer applied recursively

Given a **sequence** $x_1, x_2, \ldots$, can apply f recursively:



- $h_0 = 0$
- $(\hat{y}_1, h_1) = f(x_1, h_0)$
- $(\hat{y}_2, h_2) = f(x_2, h_1)$
- $\ldots$

This is **one** recurrent layer unfolded (over steps), *not many different layers*.

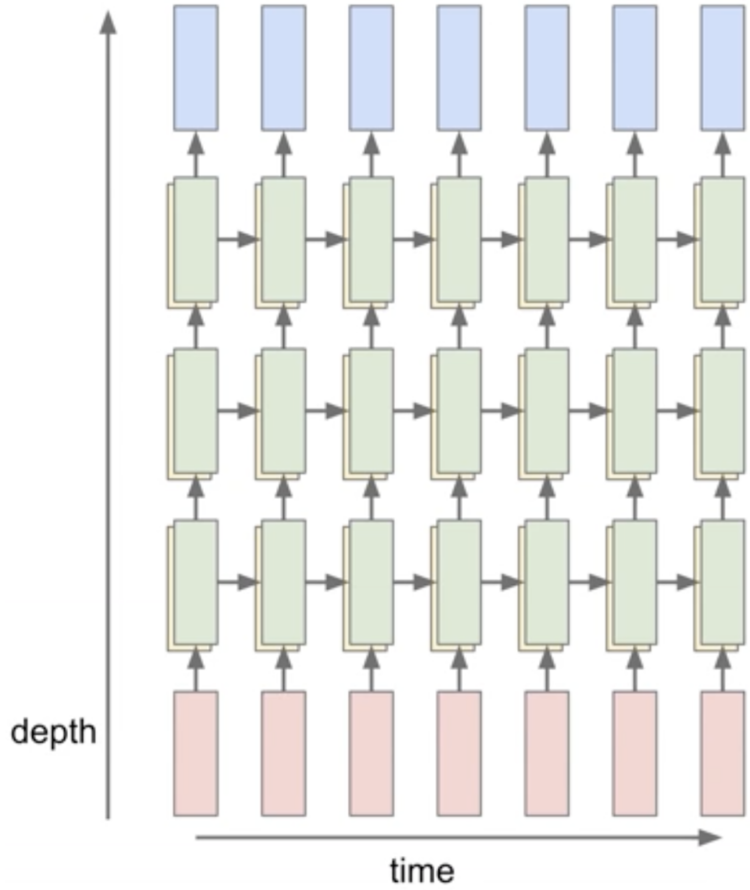
The same f (i.e, W, U, V, b_h, b_y) is **shared** in all steps (similar to CNN's filters shared across different spatial locations).

Hidden state h_t summarizes information up to step t.

Making it "deep"

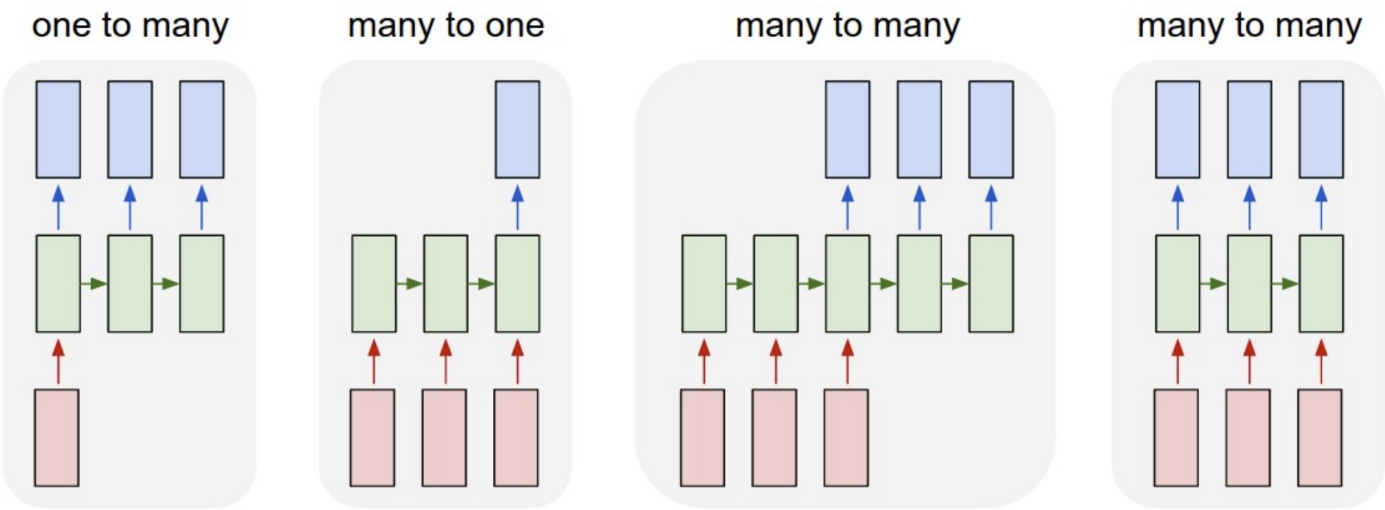
Stack multiple recurrent layers:

- hidden states become the inputs of the next layer
- different layers learn different $\boldsymbol{W}, \boldsymbol{U}, \boldsymbol{b}_h$
- last layer learns $\boldsymbol{V}, \boldsymbol{b}_y$ and output $\widehat{\boldsymbol{y}}$



A flexible sequence-to-sequence model

Many possible structures and applications:



- **one-to-many**: image captioning
- **many-to-one**: sentiment classification
- **many-to-many**: machine translation, question answering
- **(aligned) many-to-many**: POS tagging, name entity recognition

How to train an RNN

Take **text generation** (unsupervised learning) as an example:

- given a corpus, train an RNN that learns $P(\boldsymbol{x}_t \mid \boldsymbol{x}_{1:t-1})$

For each sequence $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \in \mathbb{R}^d$ (one-hot representation) in the corpus

- feed $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{T-1}$ into the current RNN to get $\widehat{\boldsymbol{y}}_1, \ldots, \widehat{\boldsymbol{y}}_{T-1} \in \mathbb{R}^d$
- each $\widehat{\boldsymbol{y}}_t$ defines a distribution over the next word via softmax: $P(\text{next word} = i) \propto \exp(\widehat{\boldsymbol{y}}_{t,i})$
- based on the true label $\boldsymbol{x}_{t+1}$, each $\widehat{\boldsymbol{y}}_t$ incurs cross-entropy loss

$$-\ln\left(\frac{\exp(\widehat{\boldsymbol{y}}_t^\top \boldsymbol{x}_{t+1})}{\sum_{i=1}^d \exp(\widehat{\boldsymbol{y}}_{t,i})}\right)$$

- update the RNN parameters using backpropagation over the total loss

Demo

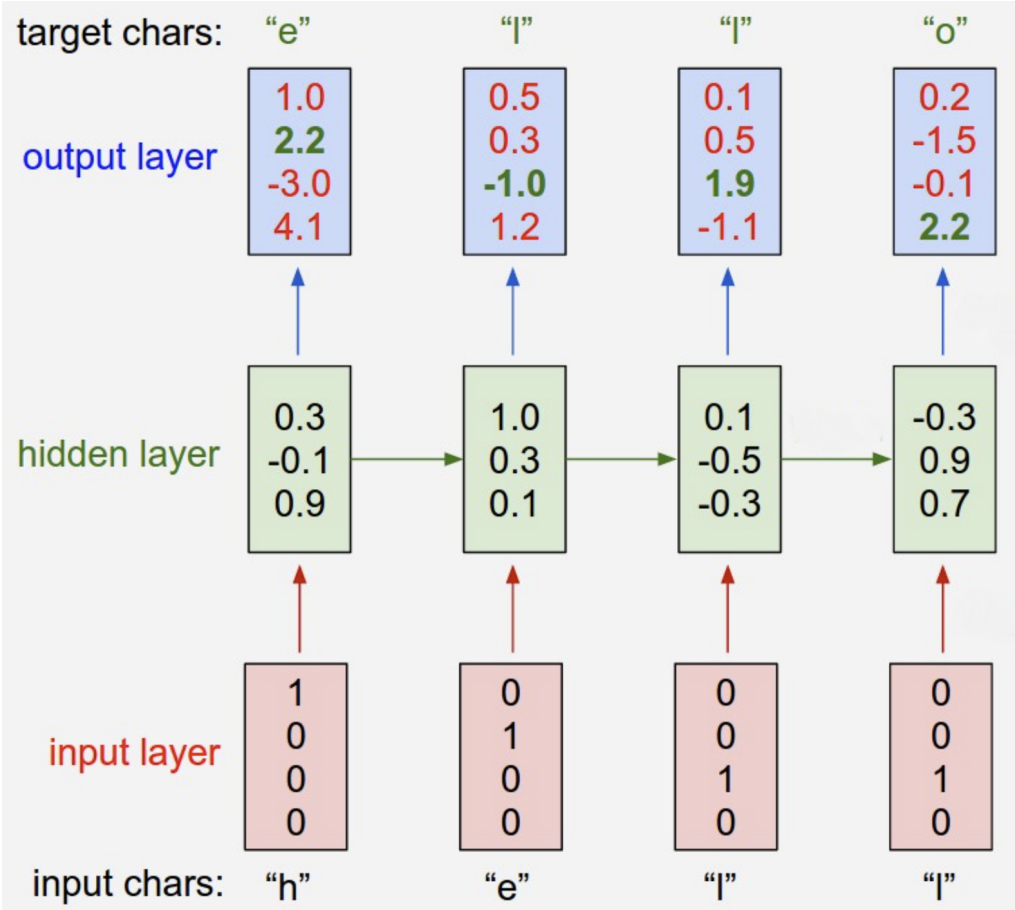
Tiny RNN, predicting the **next bit of a binary sequence**

- `https://joshvarty.github.io/VisualizingRNNs/`
- the entire vocabulary is just $\{0, 1\}$ $(d = 2)$
- one-layer RNN with $d_h = 3$, so parameters are $\boldsymbol{W} \in \mathbb{R}^{3\times3}, \boldsymbol{U} \in \mathbb{R}^{3\times2}, \boldsymbol{V} \in \mathbb{R}^{2\times3}, \boldsymbol{b}_h \in \mathbb{R}^3, \boldsymbol{b}_y \in \mathbb{R}^2$

Another demo

Min-Char RNN, predicting the **next character of a sequence**

- `https://karpathy.github.io/2015/05/21/rnn-effectiveness/`



Generation after training

Keep sampling from softmax($\widehat{\boldsymbol{y}}_t$) as the next input $\boldsymbol{x}_{t+1}$ to RNN

Can control how "random" the generation is via softmax($\beta \cdot \widehat{\boldsymbol{y}}_t$)

- $1/\beta$ is called temperature
- larger temperature (smaller β) leads to more random outputs
  - $\beta = 0$, uniform output (maximum entropy)
  - $\beta = \infty$, deterministically output $\operatorname{argmax}_i \widehat{\boldsymbol{y}}_{t,i}$ ("hard" max)

Generation after training

A few remarkable examples from Min-Char RNN:

- corpus: LaTeX source code of an algebraic geometry book (16MB)
- generate source code that almost complies
- the model understands complex syntactic structures

For $\bigoplus_{n=1,\ldots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the compariciely in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, **??** and the fact that any U affine, see Morphisms, Lemma **??**. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}'_{X',x'}$ is separated. By Algebra, Lemma **??** we can define a map of complexes $\mathrm{GL}_{S'}(x'/S'')$ and we win. □

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_p$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1}\mathcal{F})$$

Generation after training

A few remarkable examples from Min-Char RNN:

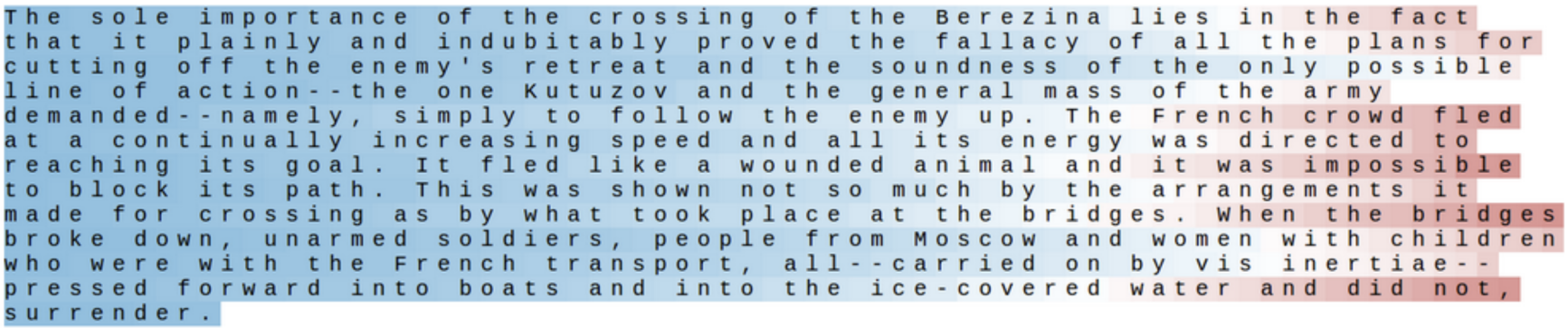
- corpus: Linux source code (474MB of C code); 10M parameters
- generate codes with very few syntactic errors
- uses strings/pointers properly, open/close brackets correctly, good indentation, even add comments

```
static int indicate_policy(void)
{
  int error;
  if (fd == MARN_EPT) {
    /*
     * The kernel blank will coeld it to userspace.
     */
    if (ss->segment < mem_total)
      unblock_graph_and_set_blocked();
    else
      ret = 1;
    goto bail;
  }
  segaddr = in_SB(in.addr);
  selector = seg / 16;
  setup_works = true;
  for (i = 0; i < blocks; i++) {
    seq = buf[i++];
    bpf = bd->bd.next + i * search;
    if (fd) {
      current = blocked;
    }
  }
  rw->name = "Getjbbregs";
  bprm_self_clearl(&iv->version);
  regs->new = blocks[(BPF_STATS << info->historidac)] | PFMR_CLOBATHINC_SECONDS << 12;
  return segtable;
}
```

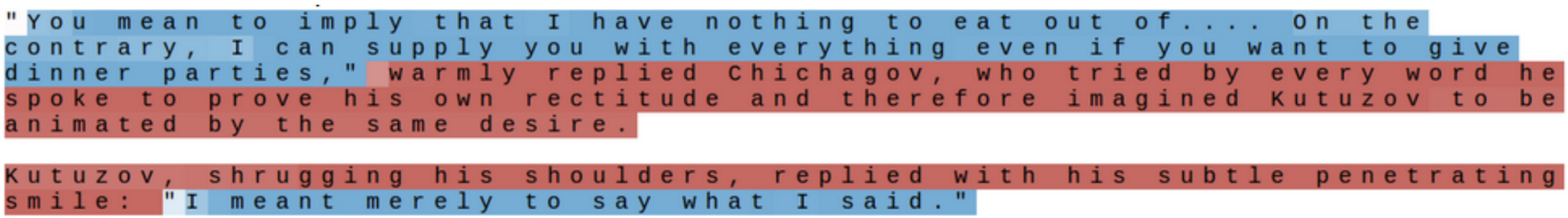
A closer look at some neurons

Some neurons (entries of the hidden state h) are quite **interpretable** (even though most are not)

- can visualize this by coloring the input character based on the value of this neuron (red = large value, blue = small value)



A neuron sensitive to the **position** in line



Final notes

Instead of using a character or a word as each $\boldsymbol{x}$, often use a **token** (word or sub-word)

- "apple" is a token
- "unbelievable" is 3 tokens ("un", "believ", "able")
- can reduce the size of vocabulary

Directly applying backpropagation to RNN leads to *vanishing/exploding gradient* issues when T is large

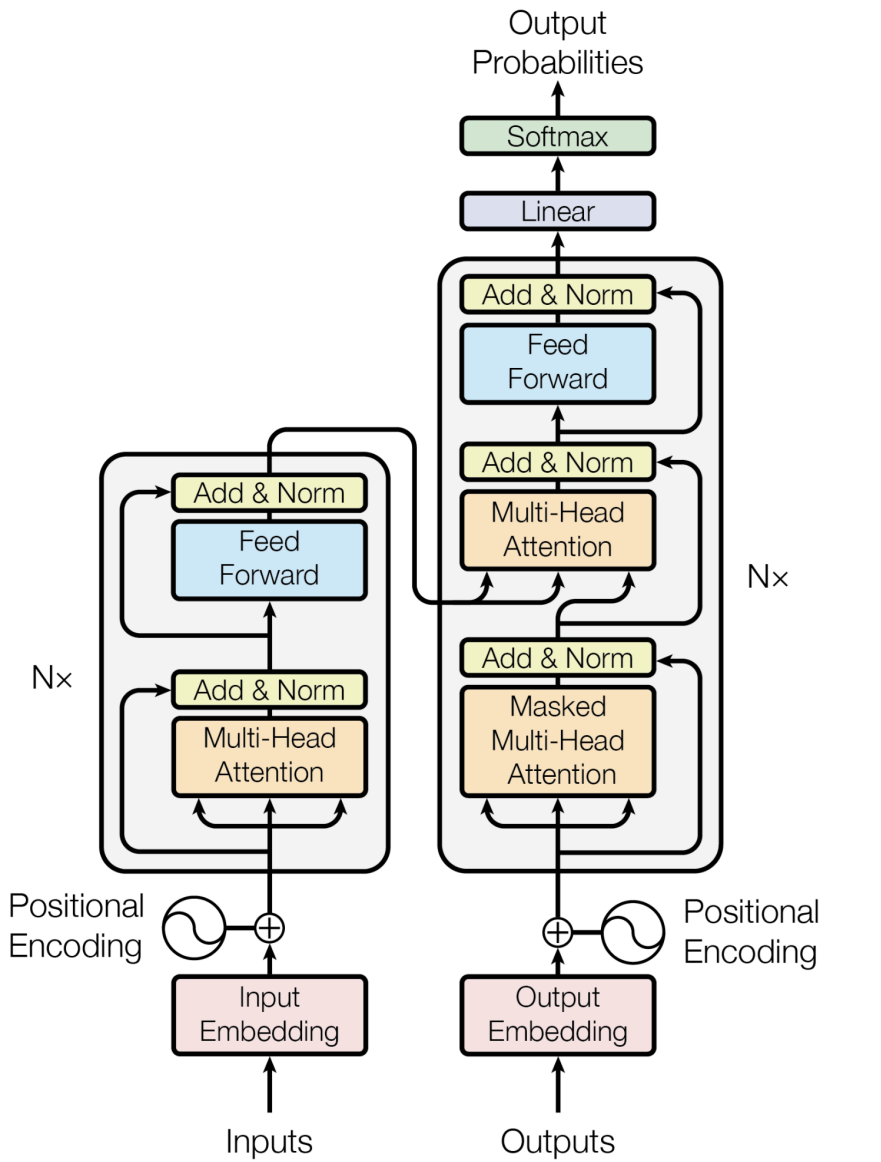
- $\boldsymbol{W}$ is applied T times at the end of the sequence (so roughly $\boldsymbol{W}^T$)
- some fixes: **Long Short-Term Memory** (LSTM) and **Gated Recurrent Units** (GRU)

Outline

Transformers

Issues of RNN: *must compress all previous info into a single state* $\boldsymbol{h}$

A solution that dominates all other models currently: **transformers**



Acknowledgements

Very useful resources:

- original paper: "Attention Is All You Need" (**200K+** citation by now)
  - `https://arxiv.org/pdf/1706.03762`
- The Illustrated Transformer (most pictures are from here)
  - `https://jalammar.github.io/illustrated-transformer/`
- a super cool Nano-GPT visualization
  - `https://bbycroft.net/llm`
- A Multiscale Visualization of Attention
  - `https://arxiv.org/pdf/1906.05714`

Key idea: self-attention

Example: "*The animal didn't cross the street because it was too tired*"

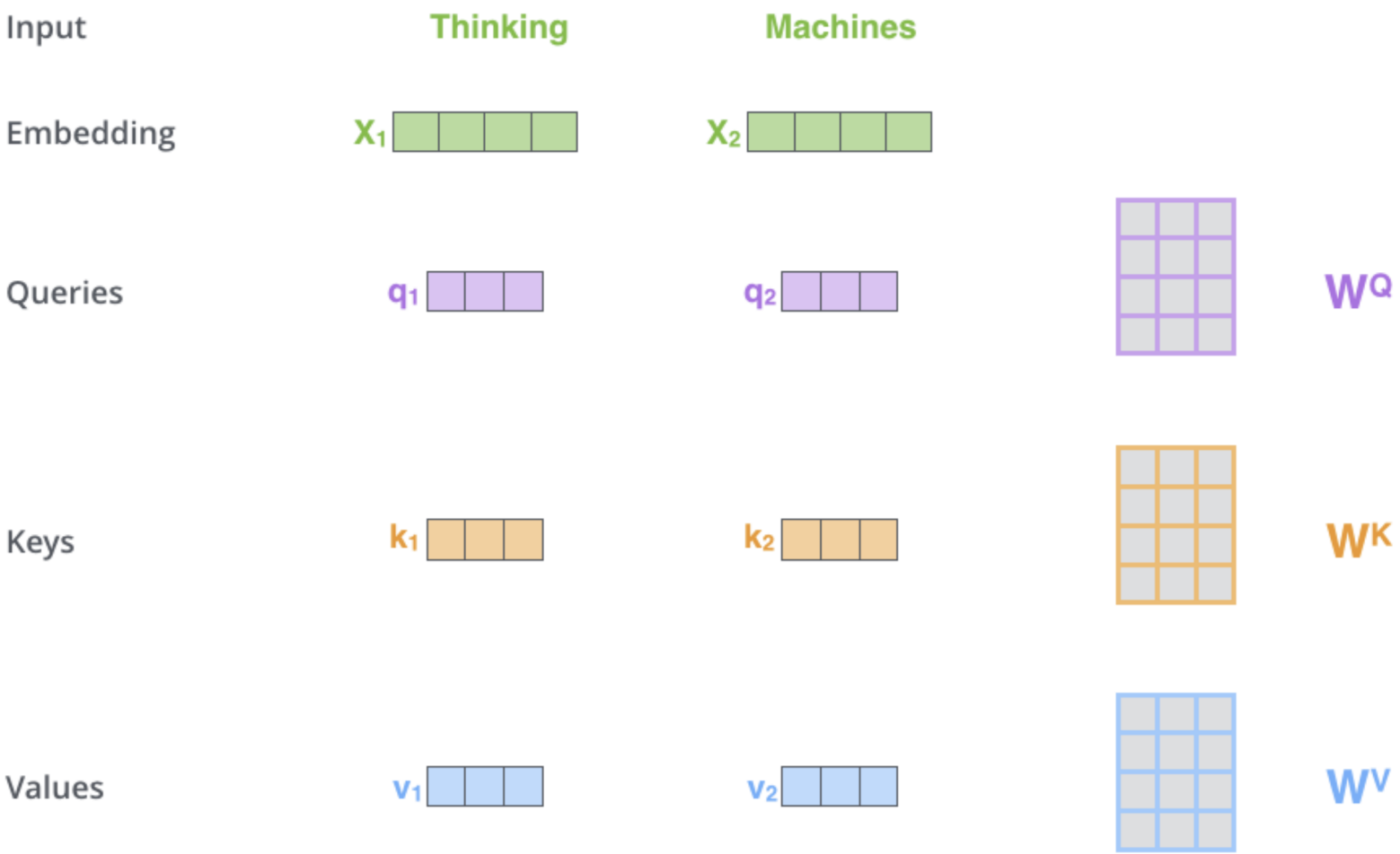
- Does "it" refer to "animal" or "street"?
- trivial for human, but how to design a model that understands this?
- intuitively, when looking at the word "it", the model should pay attention to the word "animal"
- An **attention head** does exactly this

Attention head

An attention head

- takes a sequence of inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T \in \mathbb{R}^d$ and outputs another sequence $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_T \in \mathbb{R}^{d_v}$ (similar to **hidden states** of RNN)
- parametrized by three matrices (and corresponding biases, omitted for simplicity): $\boldsymbol{W}_Q \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}_K \in \mathbb{R}^{d \times d_k}$, $\boldsymbol{W}_V \in \mathbb{R}^{d \times d_v}$
  - computes a query vector for each input $\boldsymbol{x}_t$ as $\boldsymbol{q}_t = \boldsymbol{W}_Q^\top \boldsymbol{x}_t \in \mathbb{R}^{d_k}$
  - computes a key vector for each input $\boldsymbol{x}_t$ as $\boldsymbol{k}_t = \boldsymbol{W}_K^\top \boldsymbol{x}_t \in \mathbb{R}^{d_k}$
  - computes a value vector for each input $\boldsymbol{x}_t$ as $\boldsymbol{v}_t = \boldsymbol{W}_V^\top \boldsymbol{x}_t \in \mathbb{R}^{d_v}$
- the output $\boldsymbol{z}_t$ is the "**answer**" to the query of $\boldsymbol{q}_t$

Attention head (cont.)



Attention head (cont.)

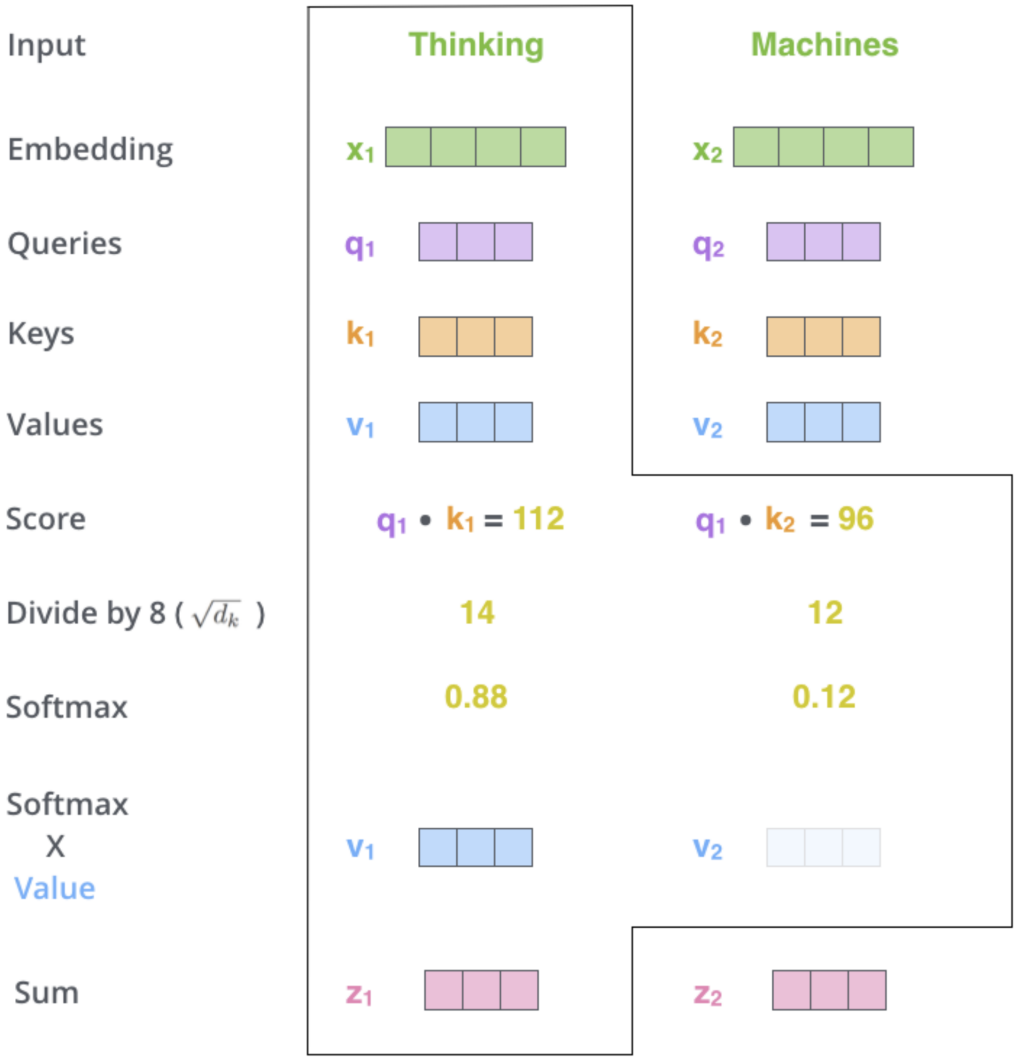
The output $\boldsymbol{z}_t$ is the "**answer**" to the query of $\boldsymbol{q}_t$. *How?*

- imagine: you make a Google query ($\boldsymbol{q}_t$), and it returns a list of website titles ($\boldsymbol{k}_{1:T}$); clicking a title ($\boldsymbol{k}_\tau$) leads you to a website ($\boldsymbol{v}_\tau$).
- You then summarize the answer using all websites ($\boldsymbol{v}_{1:T}$), each with a different **weight** based on how relevant/close its title is to your query
- formally, the final answer $\boldsymbol{z}_t$ is the **weighted sum** of $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_T$, with weights computed via

$$\text{softmax}\left(\frac{\boldsymbol{q}_t{}^\top \boldsymbol{k}_1}{\sqrt{d_k}}, \ldots, \frac{\boldsymbol{q}_t{}^\top \boldsymbol{k}_T}{\sqrt{d_k}}\right)$$

where $\boldsymbol{q}_t^\top \boldsymbol{k}_\tau$ is the **attention score** from input $\boldsymbol{x}_t$ to input $\boldsymbol{x}_\tau$

Attention head (cont.)



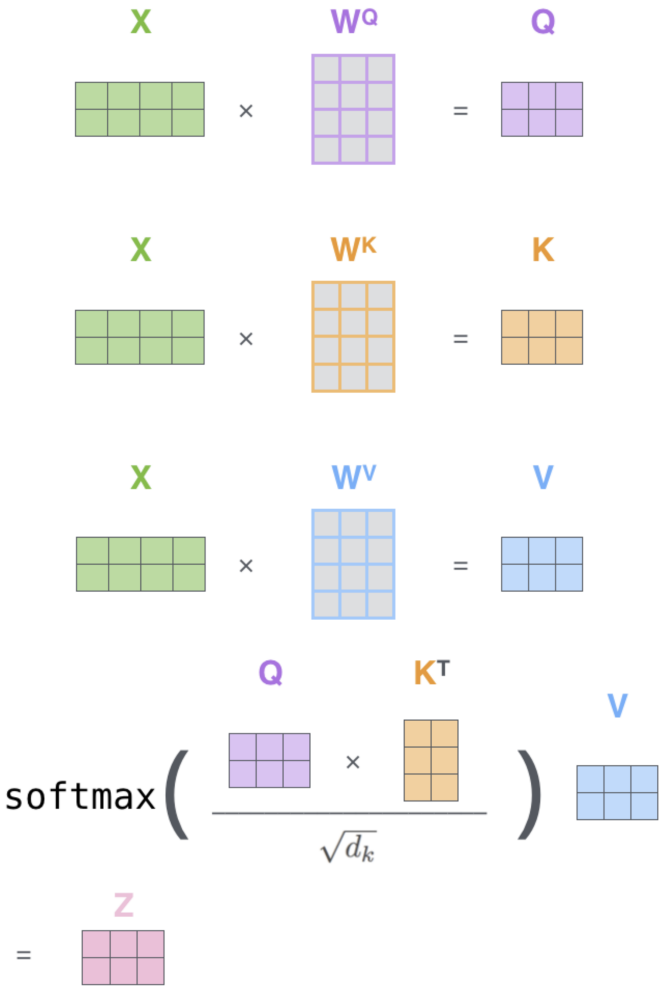
Attention head (cont.)

Matrix notation:

- input matrix $\boldsymbol{X} \in \mathbb{R}^{T \times d}$, obtained by stacking $\boldsymbol{x}_1^\top, \ldots, \boldsymbol{x}_T^\top$
- query matrix $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q \in \mathbb{R}^{T \times d_k}$
- key matrix $\boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K \in \mathbb{R}^{T \times d_k}$
- value matrix $\boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V \in \mathbb{R}^{T \times d_v}$
- attention score matrix $\boldsymbol{Q}\boldsymbol{K}^\top \in \mathbb{R}^{T \times T}$
- output matrix $\boldsymbol{Z} \in \mathbb{R}^{T \times d_v}$ is

$$\text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}$$

where softmax is *applied row-wise*

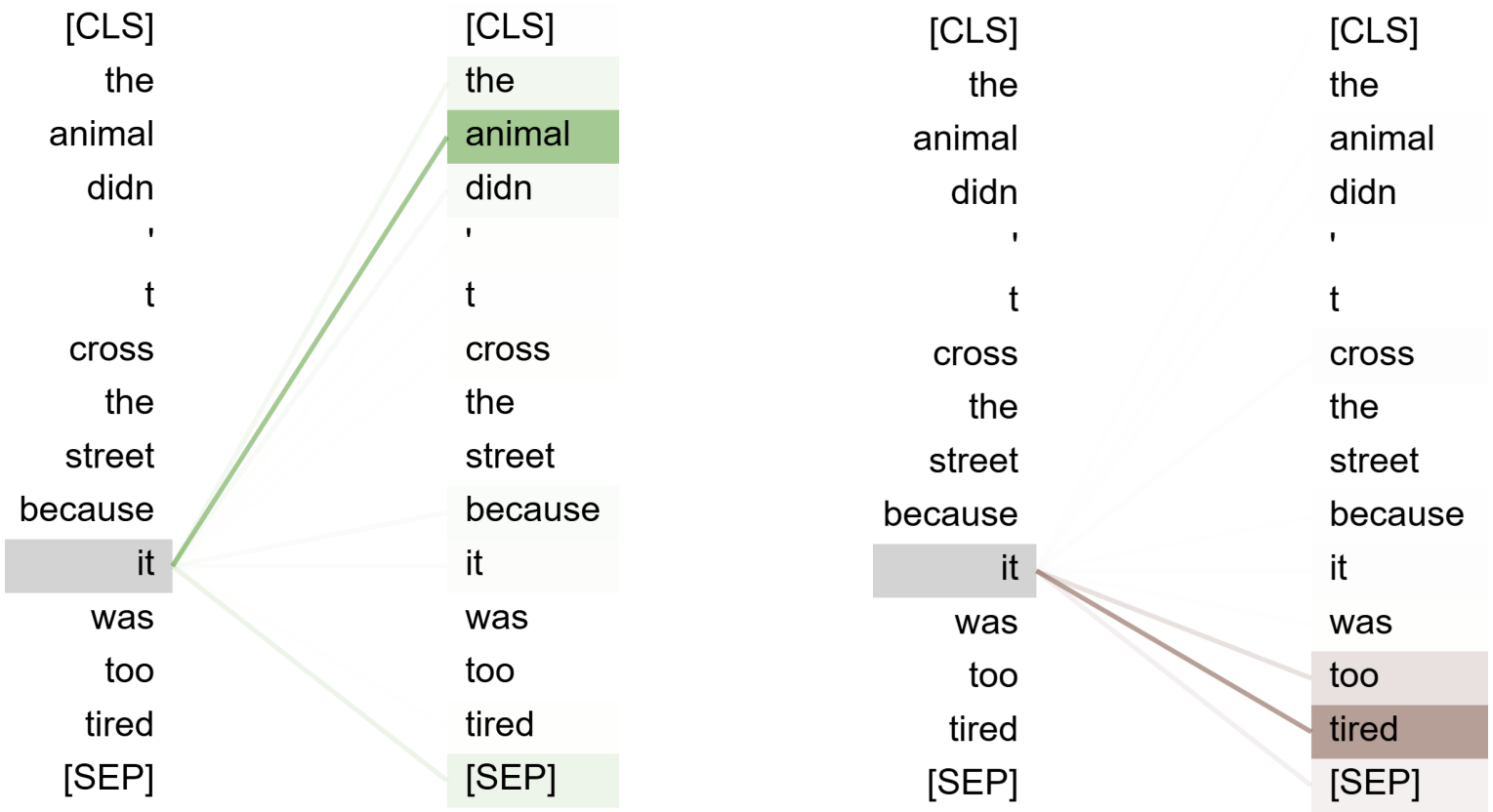


$O(T^2)$ complexity (ignoring d, d_k, d_v)

Visualization of an attention head

link

- the darker the color, the larger the attention score
- "it" attends to "animal" in one head, and "tired" in another head

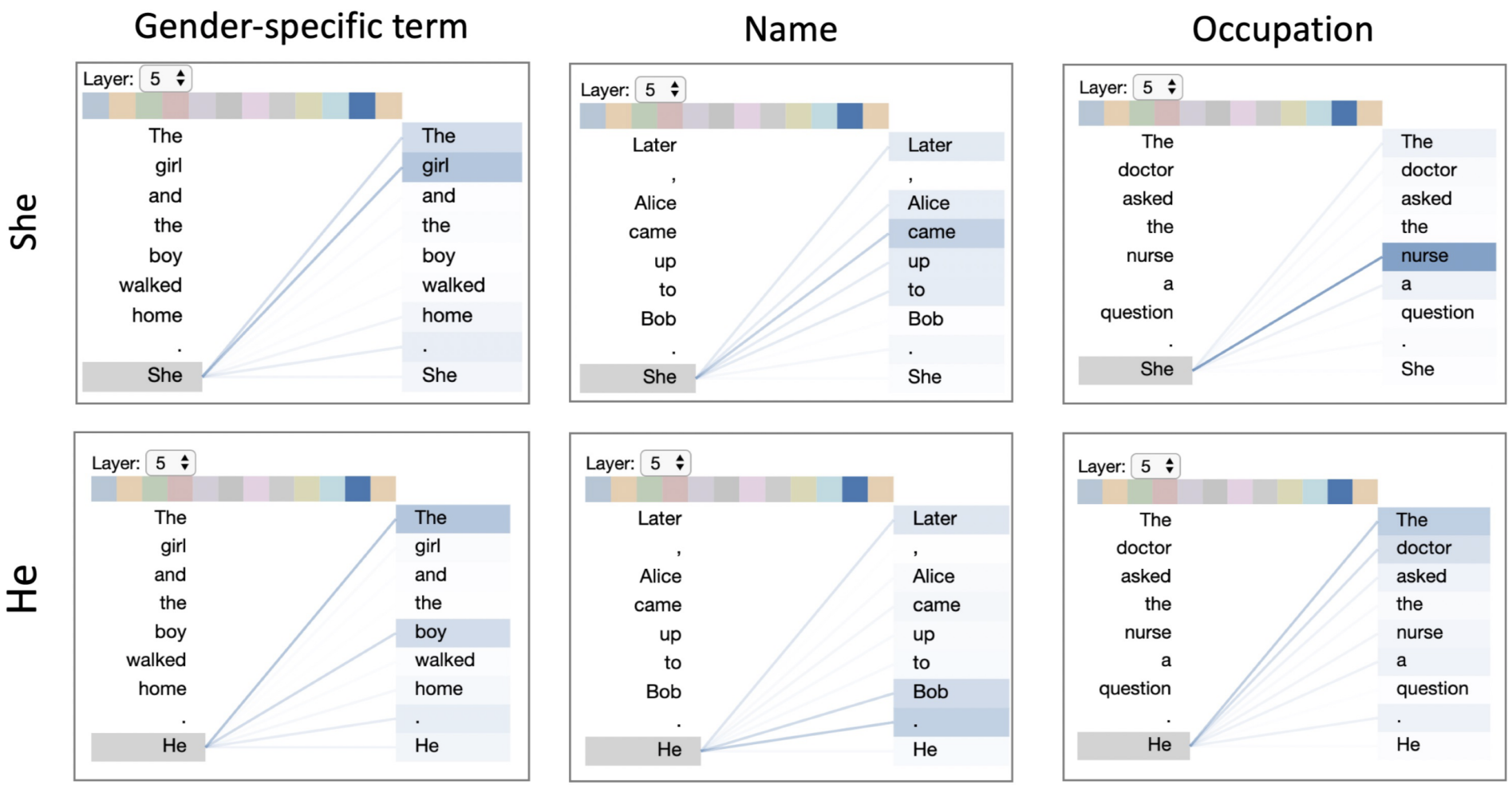


Visualization of an attention head

link

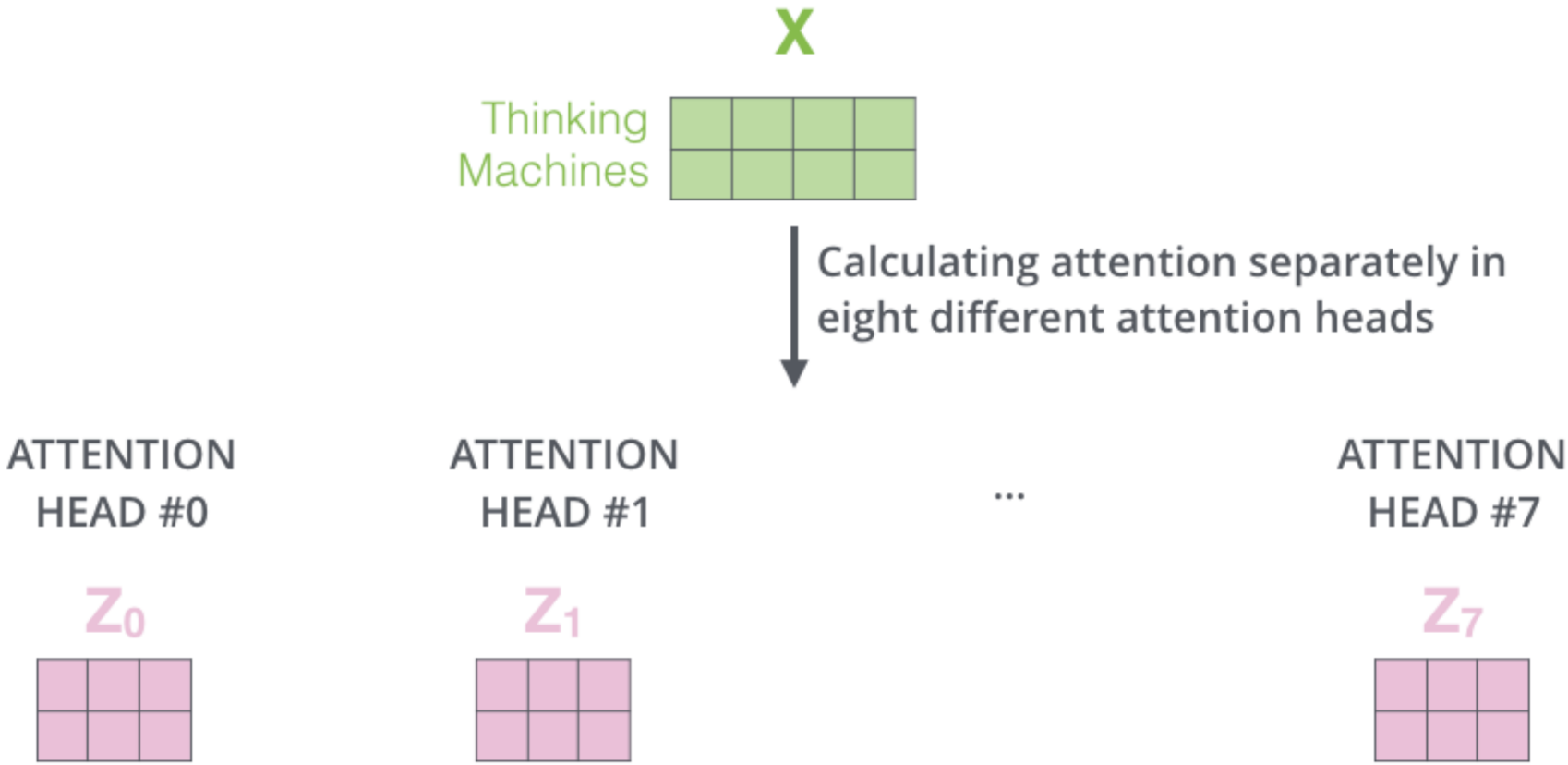
More examples

- all from unsupervised learning; *no one tells the model to learn these!*



Multi-head attention

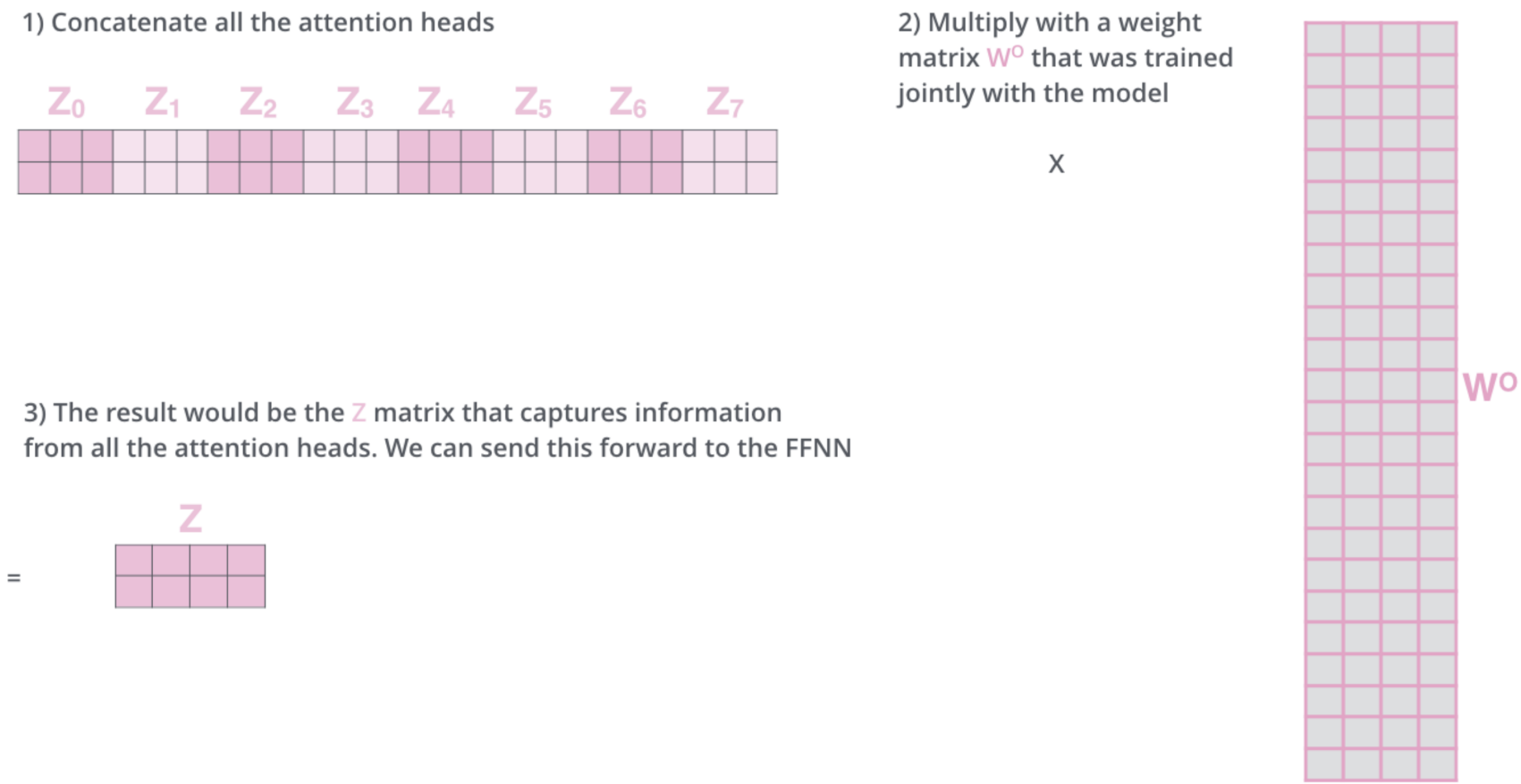
Pass $\boldsymbol{X}$ to multiple attention-heads, each with **different parameters**



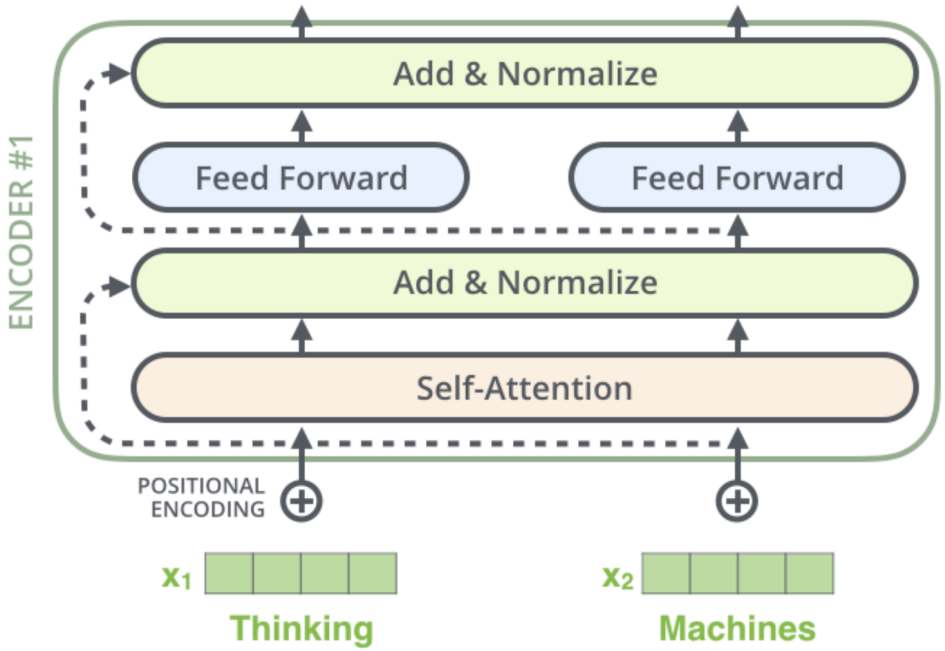
Multi-head attention (cont.)

Concatenate outputs of different heads and then **project** again

- final output dimension is $\mathbb{R}^{T \times d}$, same as inputs $\boldsymbol{X}$



Zooming out: a complete encoder



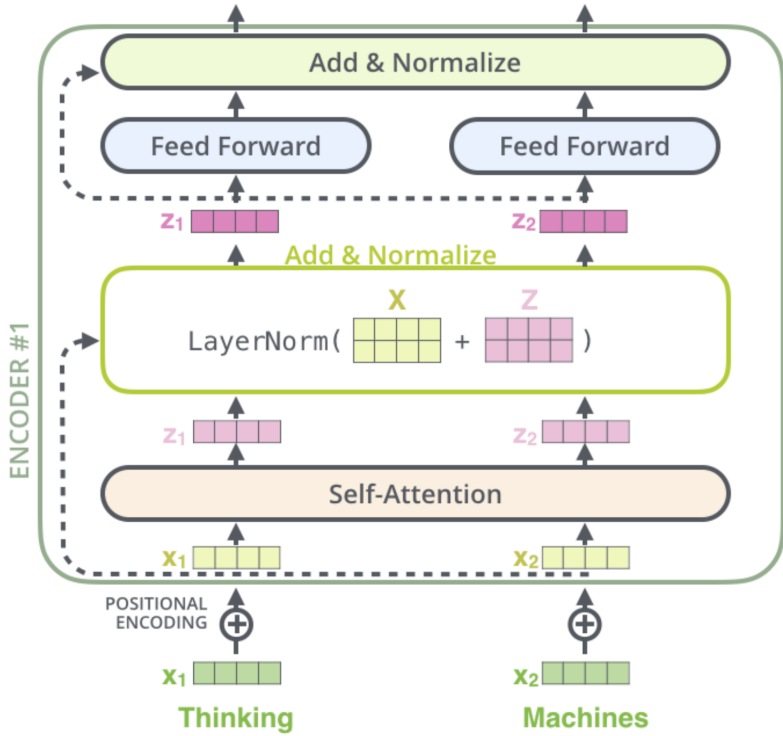
Positional encoding

- fix the issue that attention-head *does not have positional info*
- via a **positional embedding matrix** $\boldsymbol{E}_P \in \mathbb{R}^{d \times T}$, fixed or learned,
- $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_t + \boldsymbol{E}_P e_t$, i.e., add the t-th column of $\boldsymbol{E}_P$ to $\boldsymbol{x}_t$

Zooming out: a complete encoder (cont.)

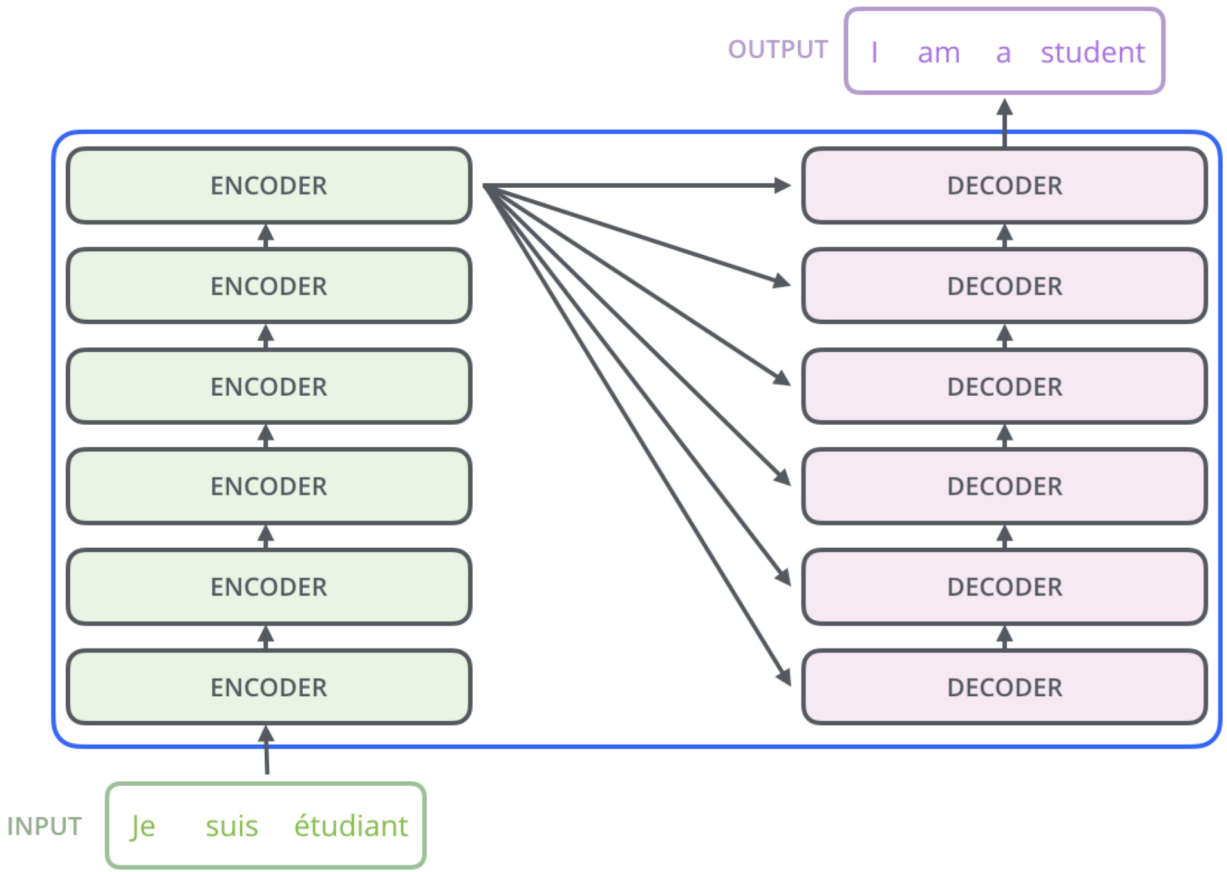
Two more components to stabilize and speed up training:

1. **Residual pathway**
   - add input to output, $\boldsymbol{Z} \leftarrow \boldsymbol{Z} + \boldsymbol{X}$
   - an idea from **Residual Networks** to deal with vanishing gradients



2. **Layer normalization**:
   - for each $\boldsymbol{z}_t \in \mathbb{R}^d$, normalize it to zero-mean and unit-variance (across features)
   - similar but different from batch normalization (where you normalize each feature to zero-mean and unit-variance across samples)

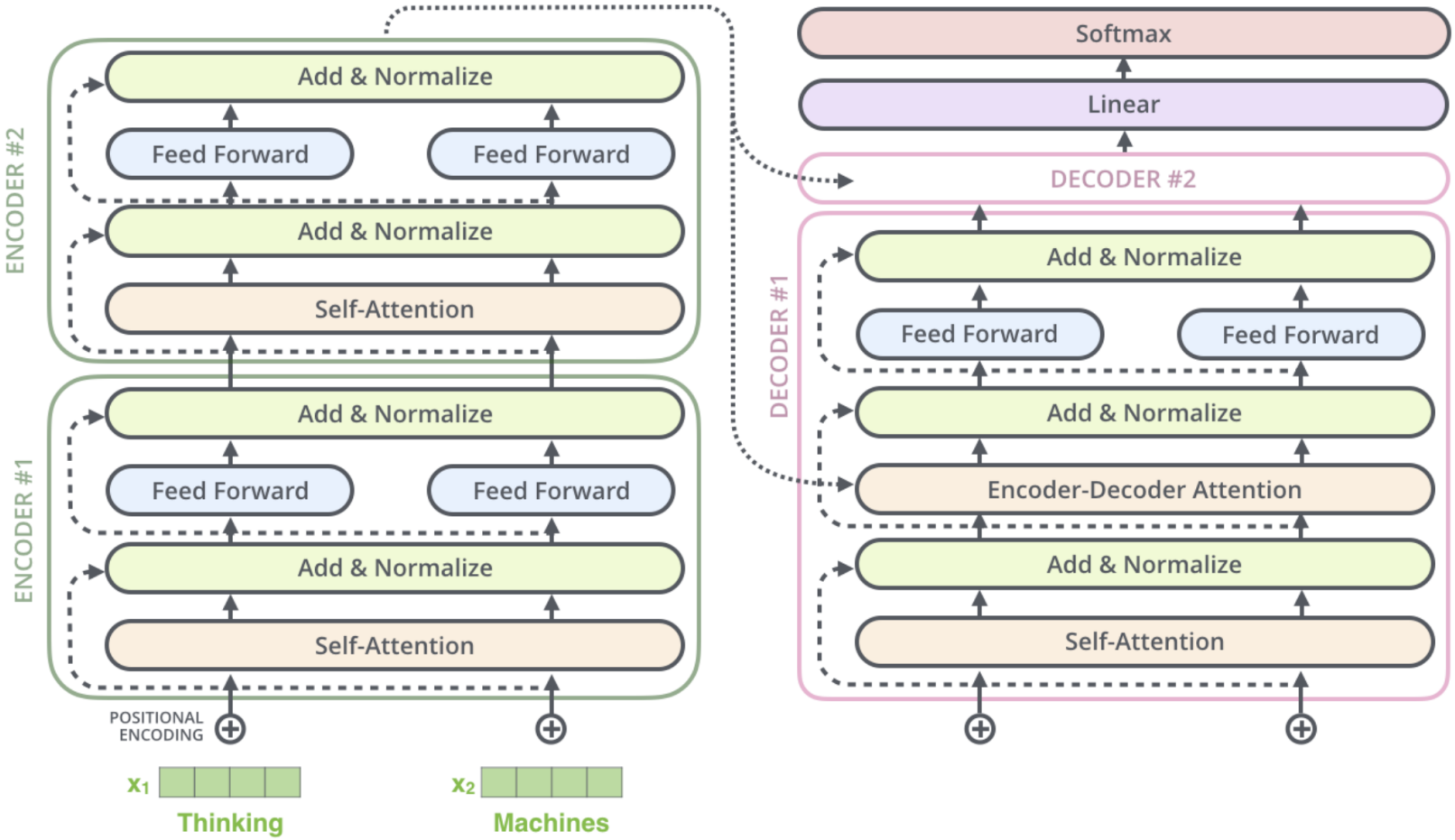
Zooming out: stacking encoders and decoders



Encoder: summarizes the input into a useful representation

Decoder: generates outputs

A closer look at decoders



Extra component: **encoder-decoder attention**

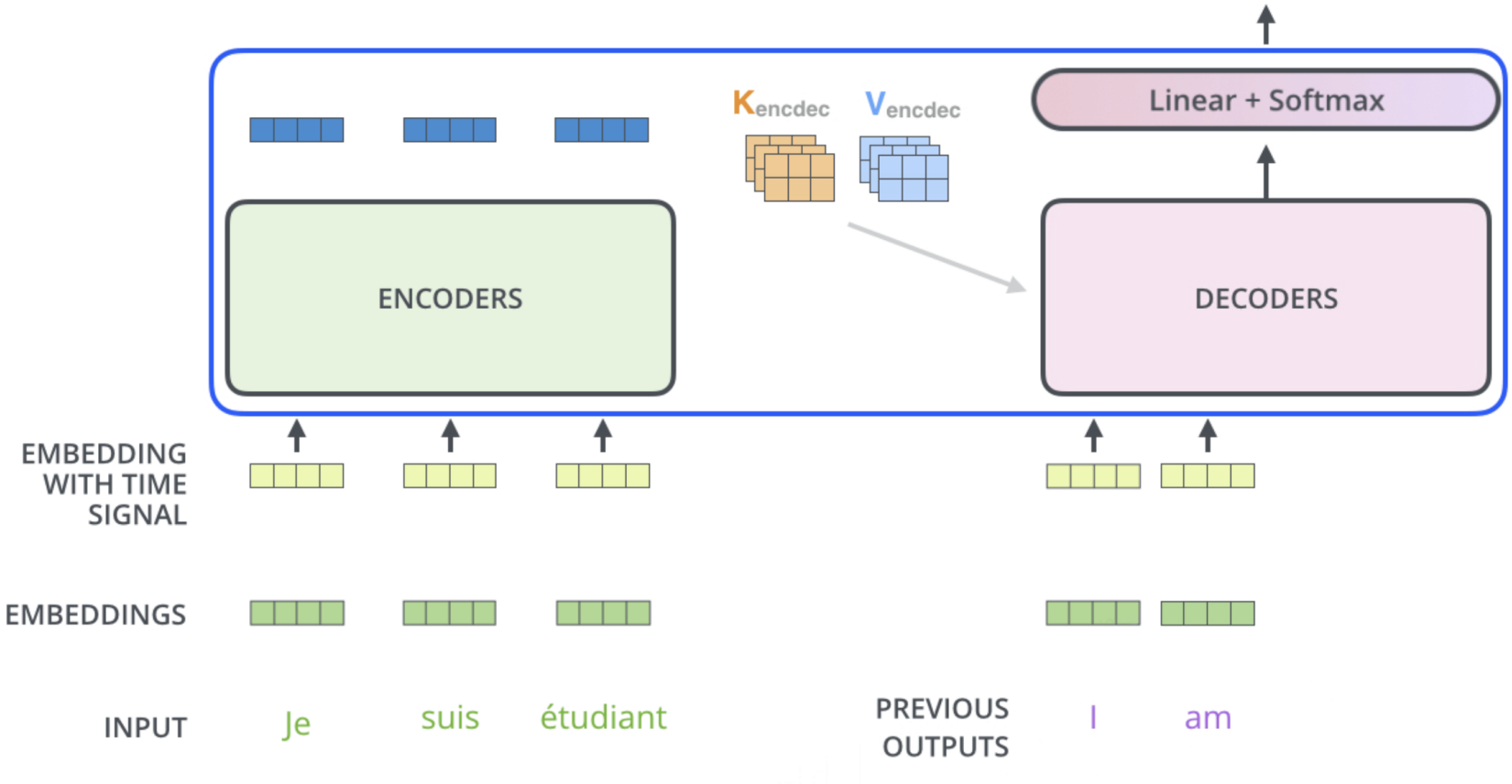
Encoder-decoder attention

Same idea as self-attention, except key and value matrices are computed using output $\boldsymbol{Z}_{\text{enc}}$ of the final encoder:

- query matrix $\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q \in \mathbb{R}^{T_{\text{dec}} \times d_k}$ (as usual)
- key matrix $\boldsymbol{K}_{\text{enc}} = \boldsymbol{Z}_{\text{enc}}\boldsymbol{W}_K \in \mathbb{R}^{T_{\text{enc}} \times d_k}$
- value matrix $\boldsymbol{V}_{\text{enc}} = \boldsymbol{Z}_{\text{enc}}\boldsymbol{W}_V \in \mathbb{R}^{T_{\text{enc}} \times d_v}$

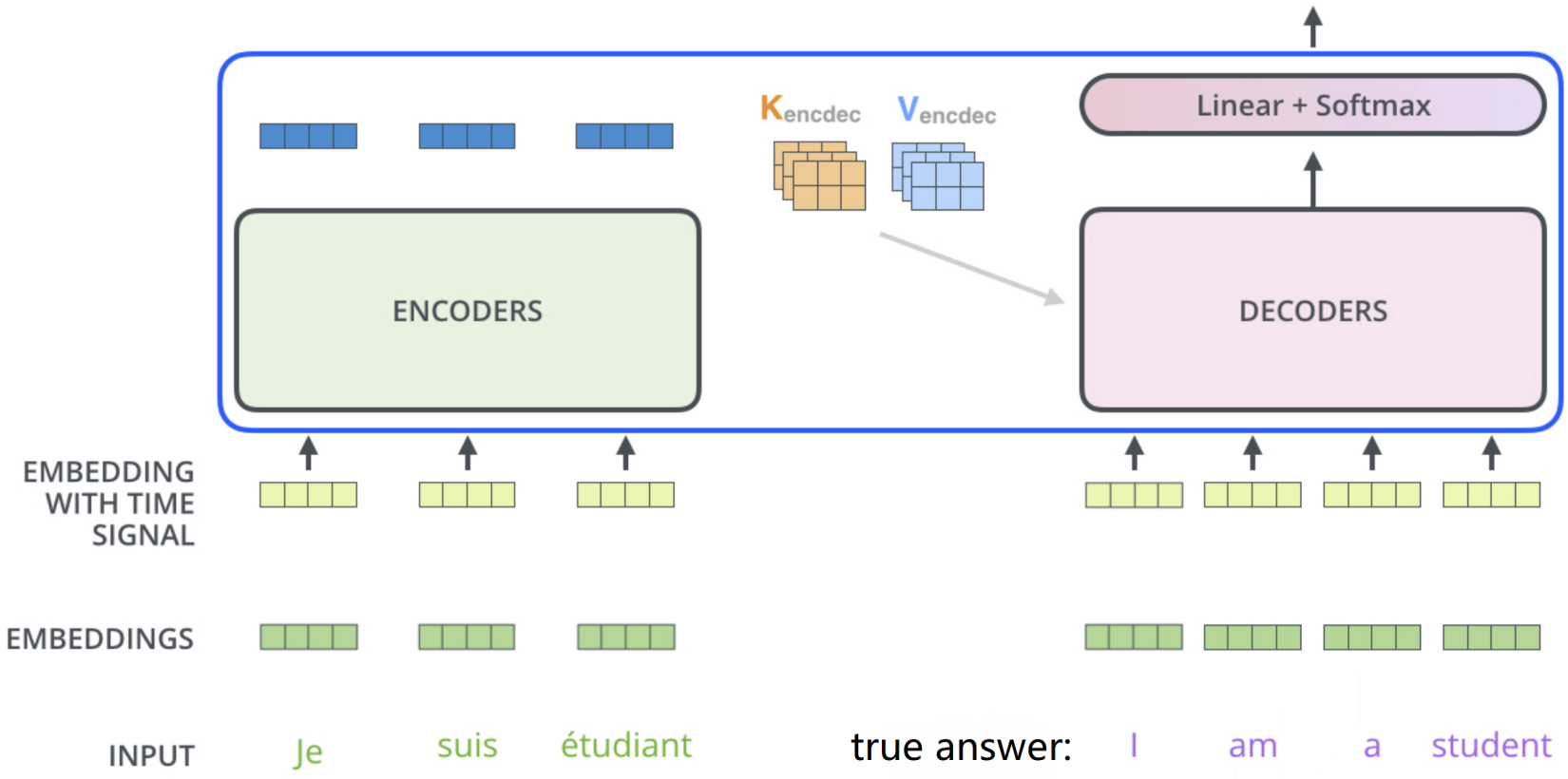
Intuition: find answer from the encoded representation of original inputs

Generating answers



Use previously generated text as inputs of the decoder

Training



Use cross-entropy loss again, and apply

- **teacher forcing**: use the true answer as inputs of the decoder

Teaching forcing and causal mask

In teaching forcing, to avoid generating a word by peeking at future words, need to use a causal mask in the decoder's self-attention heads:

$$\boldsymbol{Q}\boldsymbol{K}^\top \leftarrow \boldsymbol{Q}\boldsymbol{K}^\top + \boldsymbol{M}$$

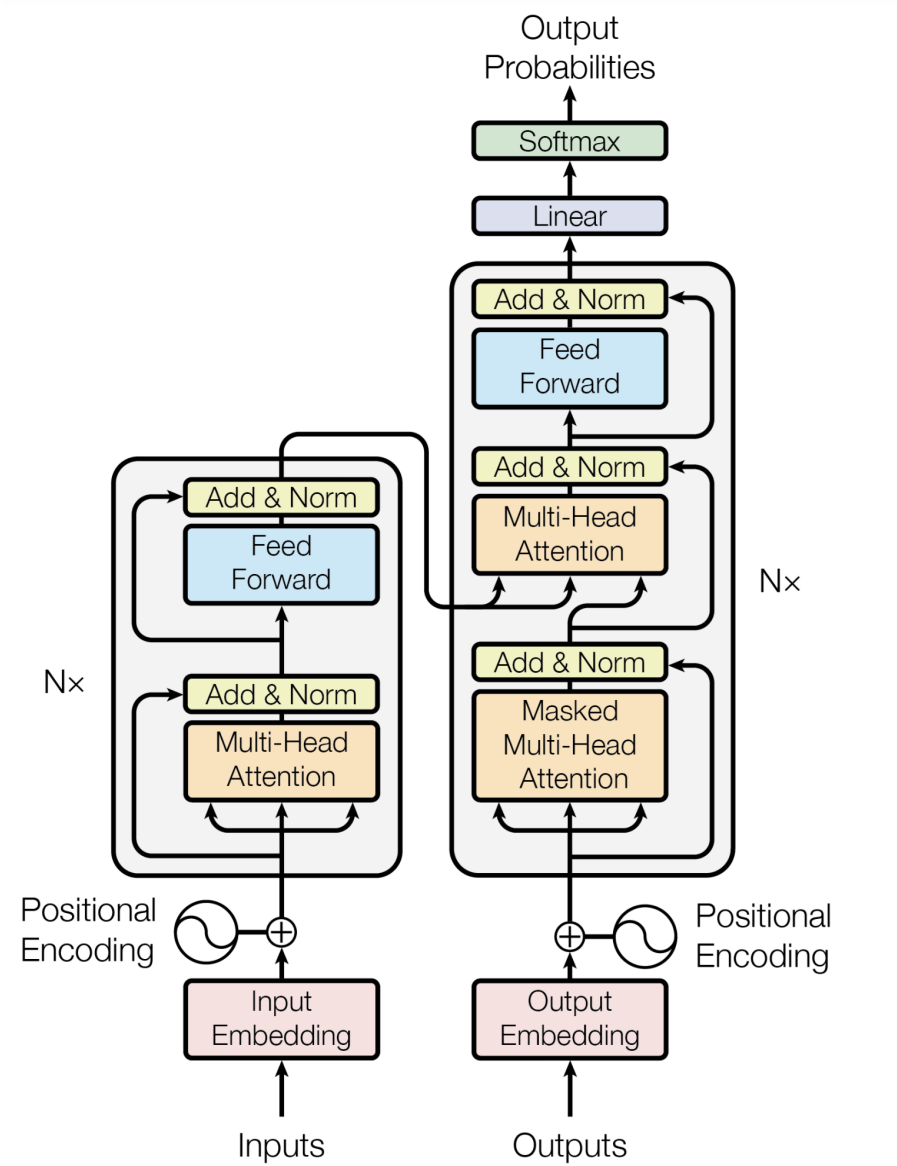
where

$$\boldsymbol{M} = \begin{bmatrix} 0 & -\infty & -\infty & \cdots & -\infty \\ 0 & 0 & -\infty & \cdots & -\infty \\ 0 & 0 & 0 & \cdots & -\infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix} \in \mathbb{R}^{T_{\text{dec}} \times T_{\text{dec}}}$$

so a word at position t never attends to words at positions $> t$

Q: should we use causal mask for encoder-decoder attention heads?

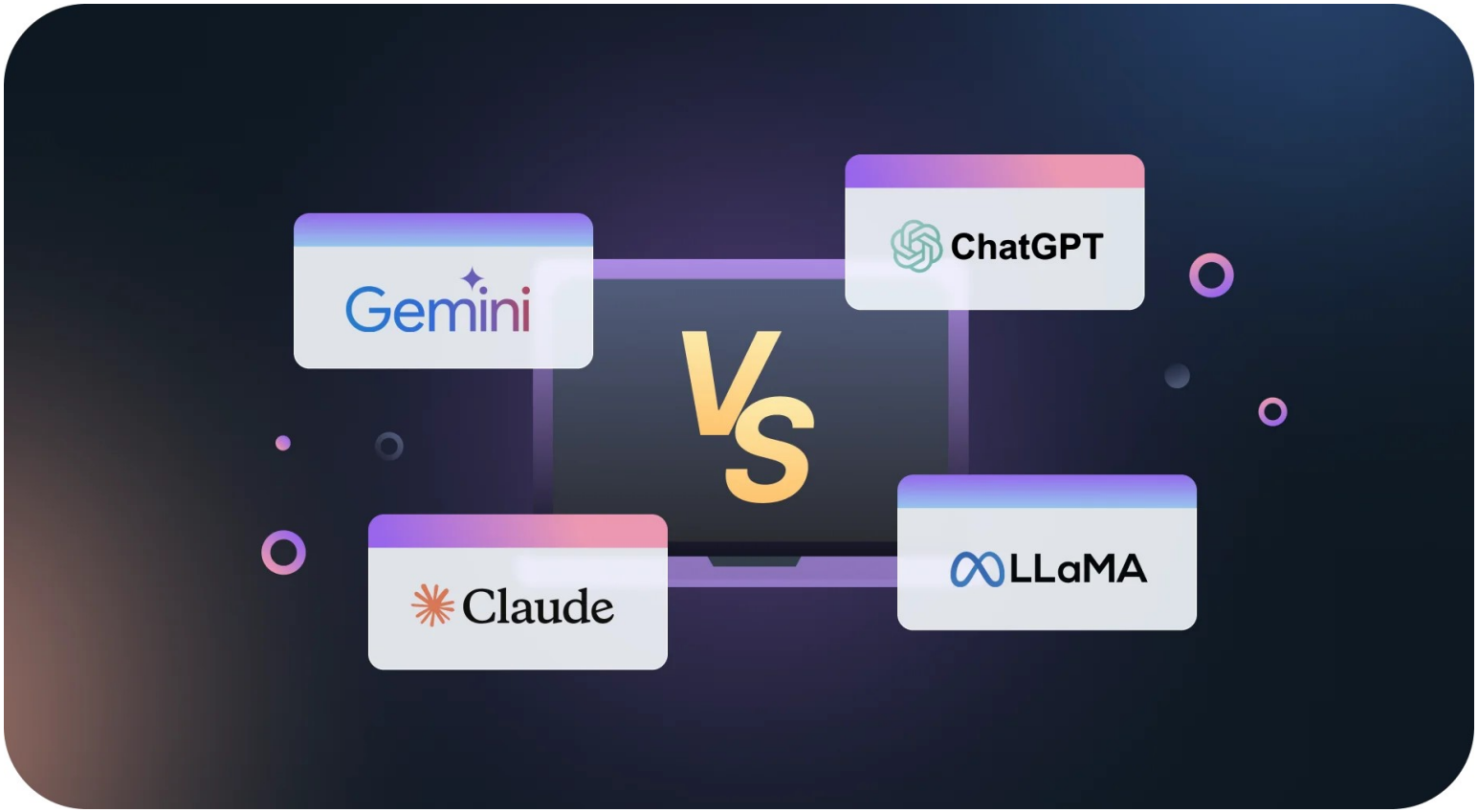
That's it!


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
N×
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs

Large language models

Large language models (LLMs) are all based on transformers

- estimated #parameters for GPT5: **trillions**



A cool 3D visualization of a **nano-GPT**: `https://bbycroft.net/llm`

Training Large language models

Unsupervised pre-training

- via next word prediction using a *huge* training set (e.g., the entire internet)

Fine-tuning

- using a labeled dataset for a specific task (translation, question answering, etc.)

Reinforcement Learning with Human Feedback (RLHF)

- get preference feedback from human: *which answer is better?*
- more on this in the next two weeks