Decoding strategies

CS 4804: Introduction to AI

Fall 2025

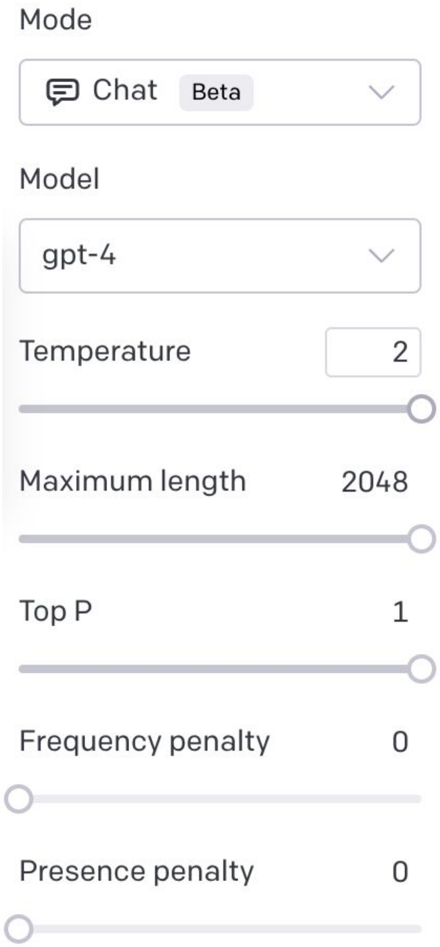
https://tuvllms.github.io/ai-fall-2025/

Tu Vu

VIRGINIA TECH.

Logistics

- Final Project proposal due 10/7
- Homework 1 due 10/14



Mode

Chat Beta

Model

gpt-4

Temperature 2

Maximum length 2048

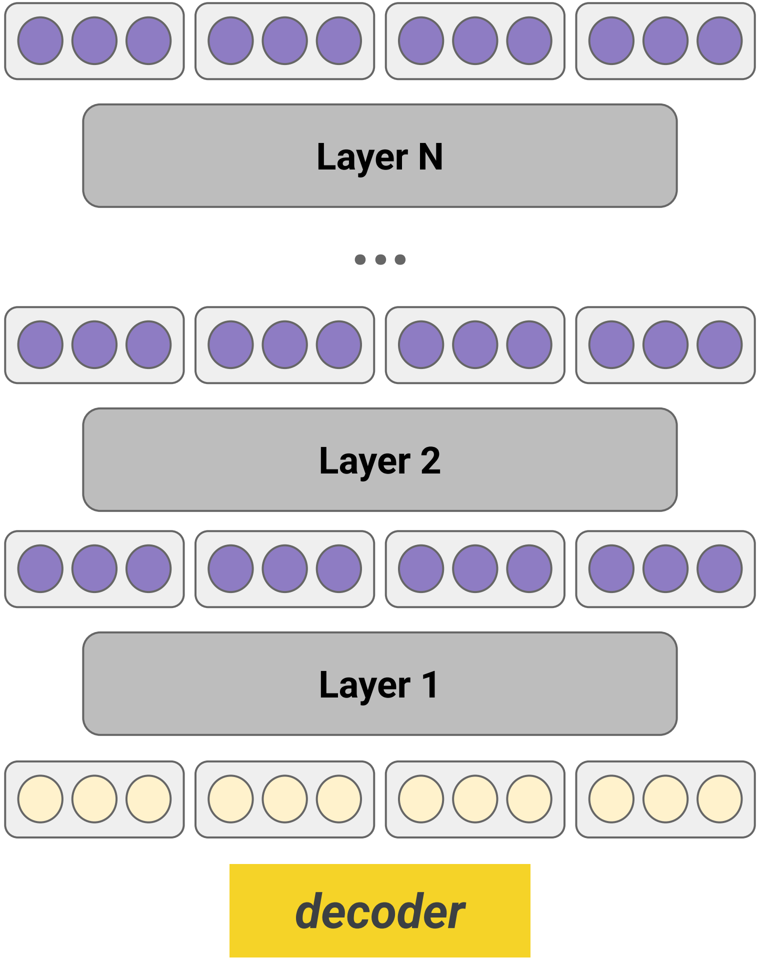
Top P 1

Frequency penalty 0

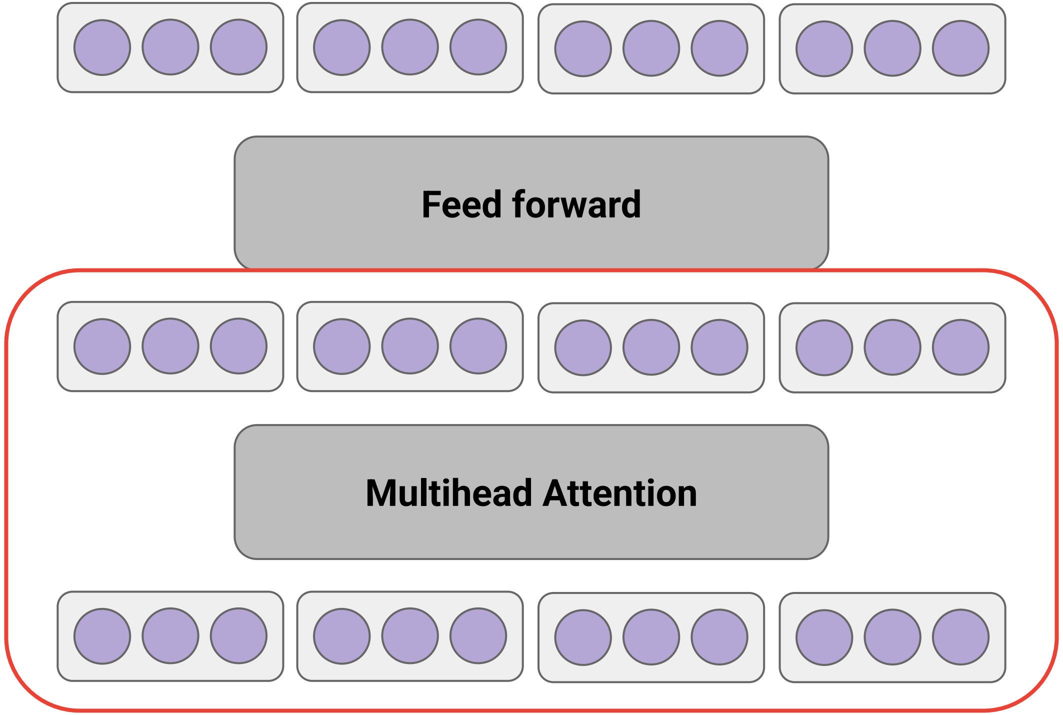
Presence penalty 0

Decoder-only Transformer review

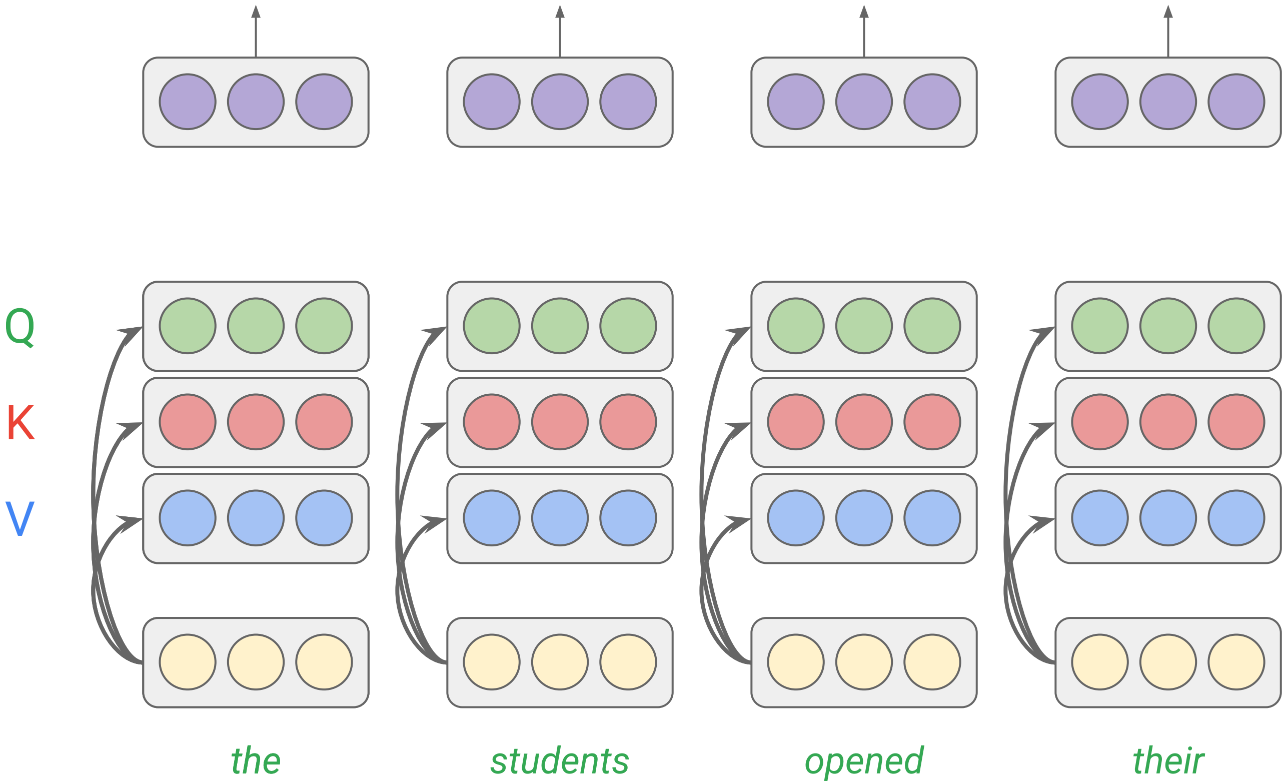
Transformer (N layers)



Transformer decoder



Attention



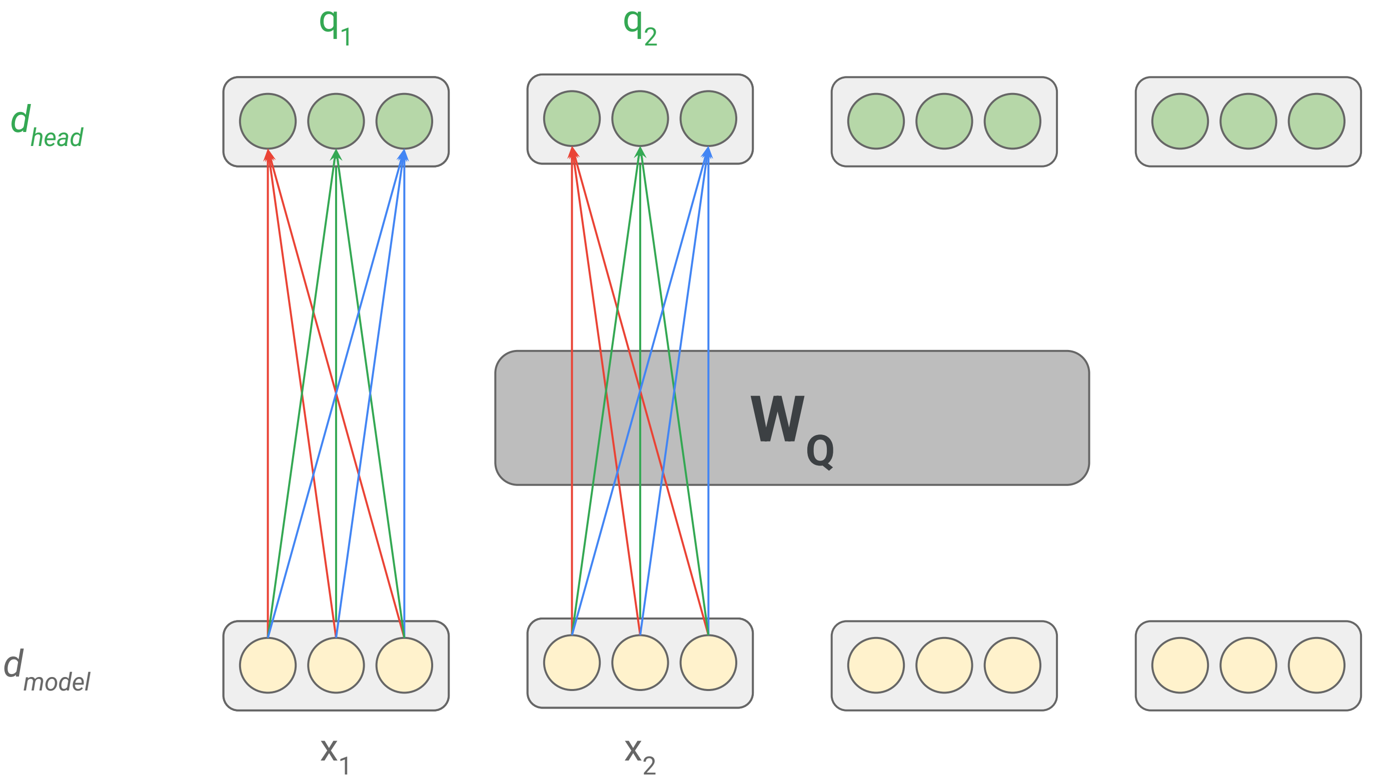
$$Q = X \cdot W_Q$$

$$K = X \cdot W_K$$

$$V = X \cdot W_V$$

linear projections

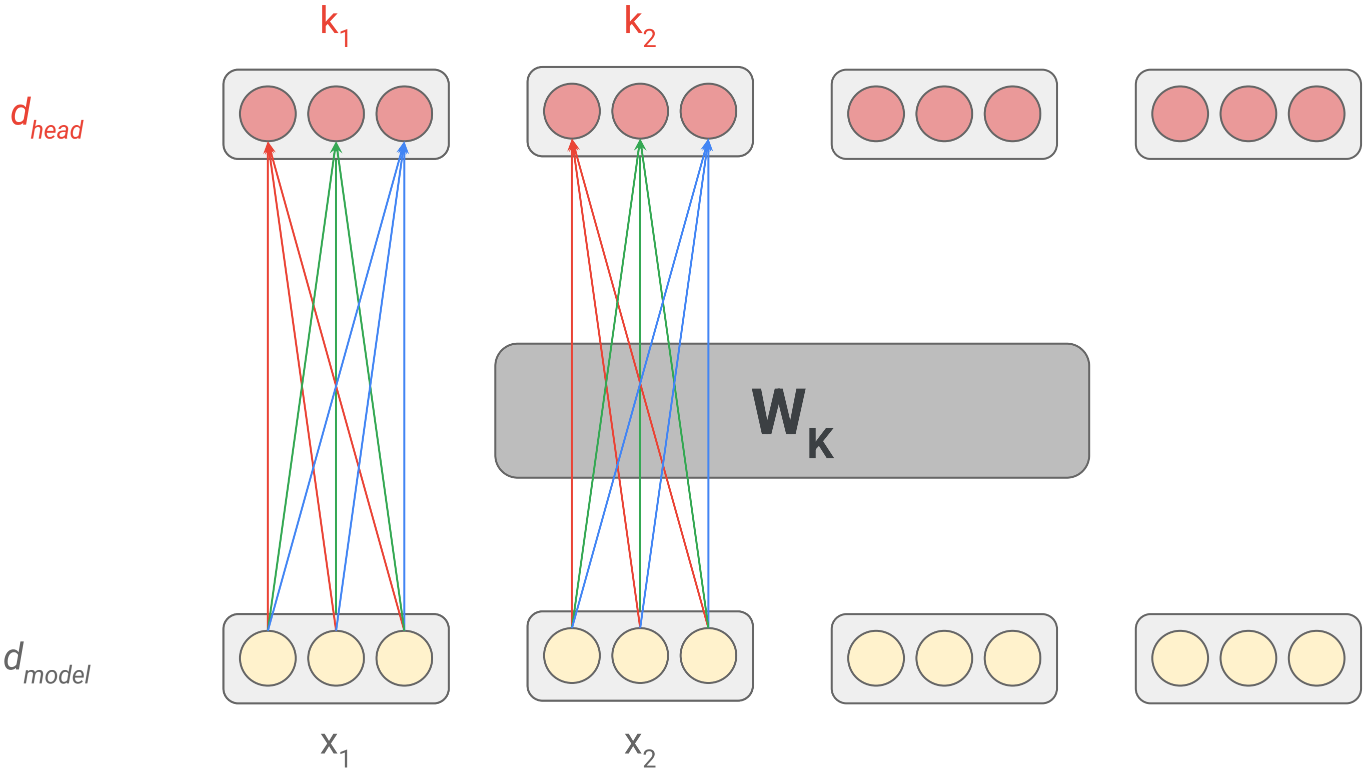
Query vectors



$$Q = X \cdot W_Q$$

linear projections

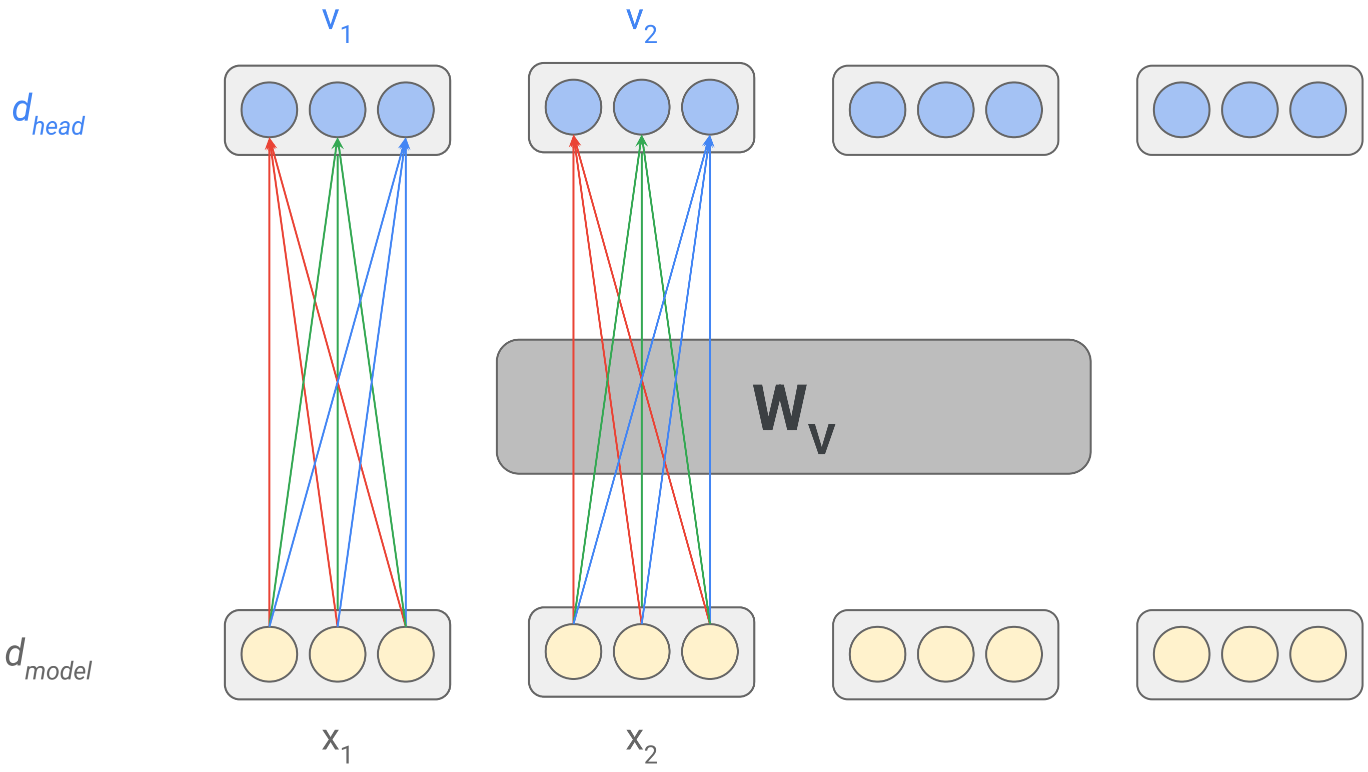
Key vectors



$$K = X \cdot W_K$$

linear projections

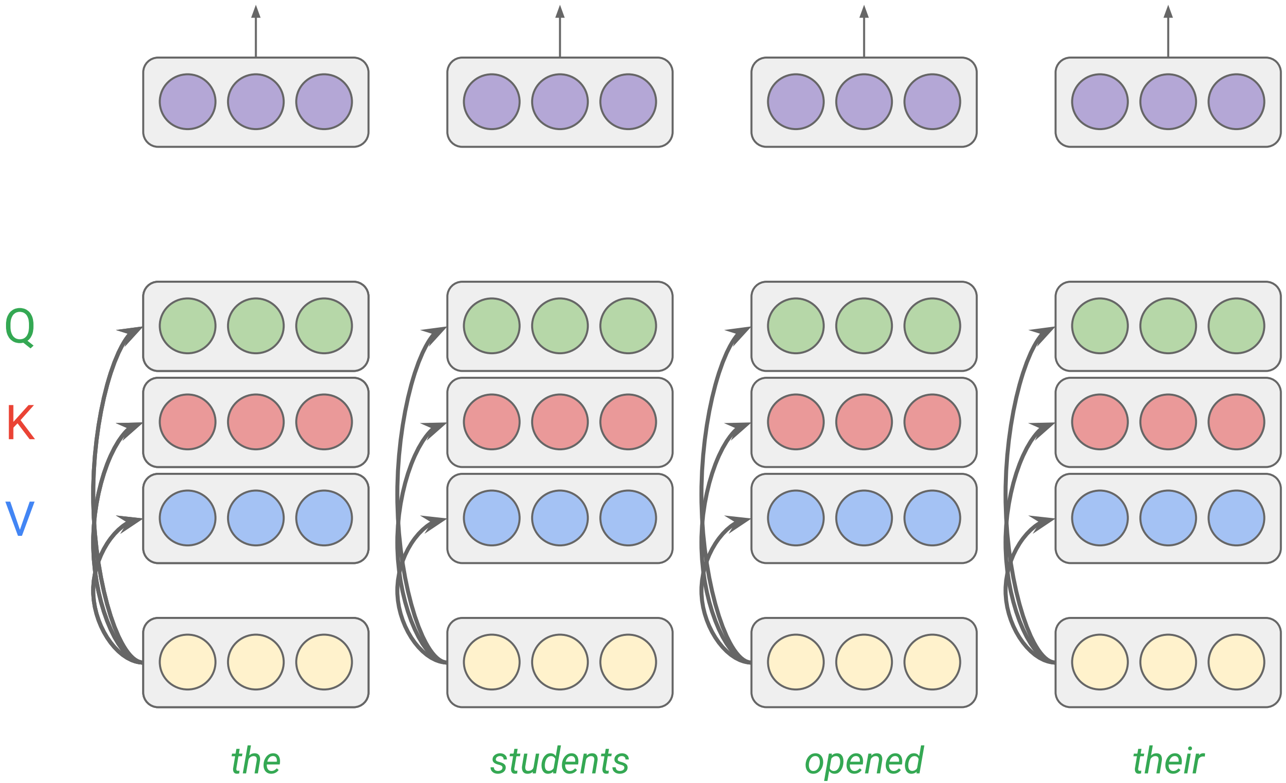
Value vectors



$$V = X \cdot W_V$$

linear projections

Attention (cont’d)



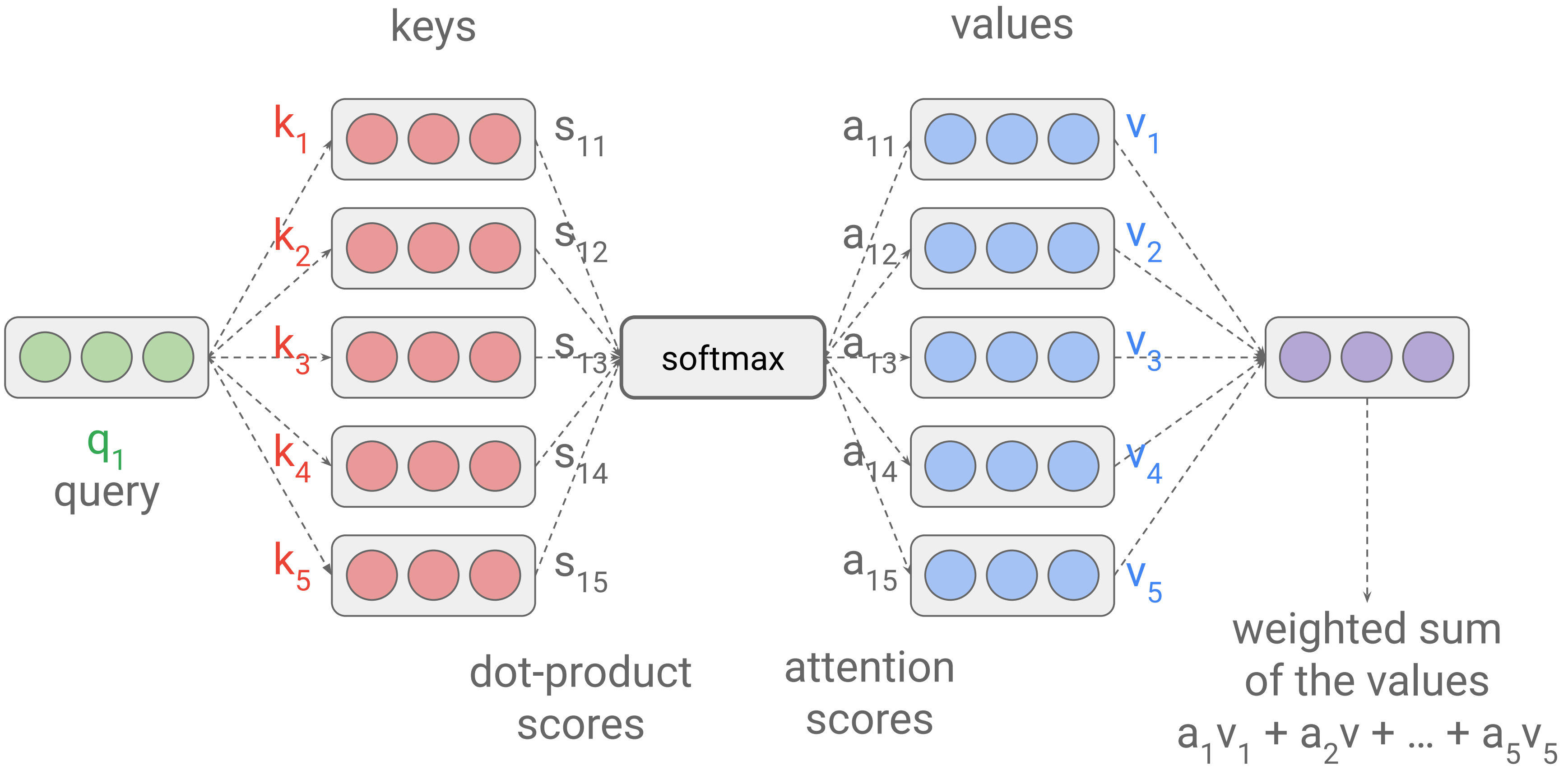
$$Q = X \cdot W_Q$$

$$K = X \cdot W_K$$

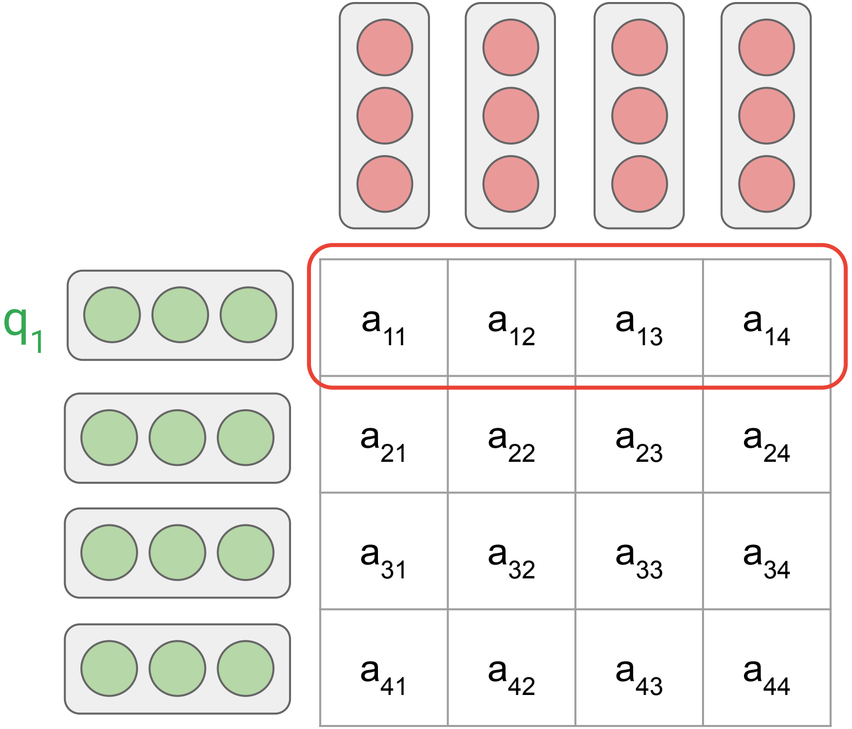
$$V = X \cdot W_V$$

linear projections

Attention (cont'd)



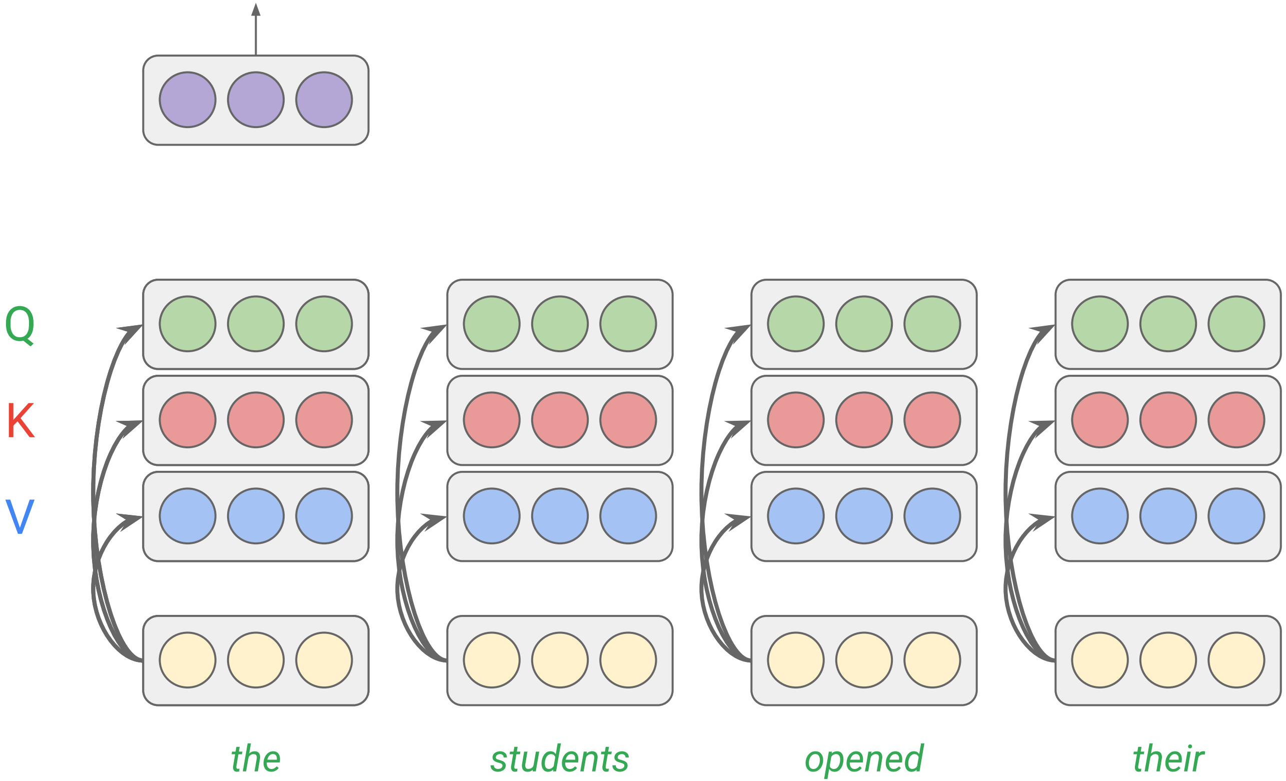
Attention matrix



a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

The time complexity of self-attention is quadratic in the input length $O(n^2)$

Attention (cont'd)



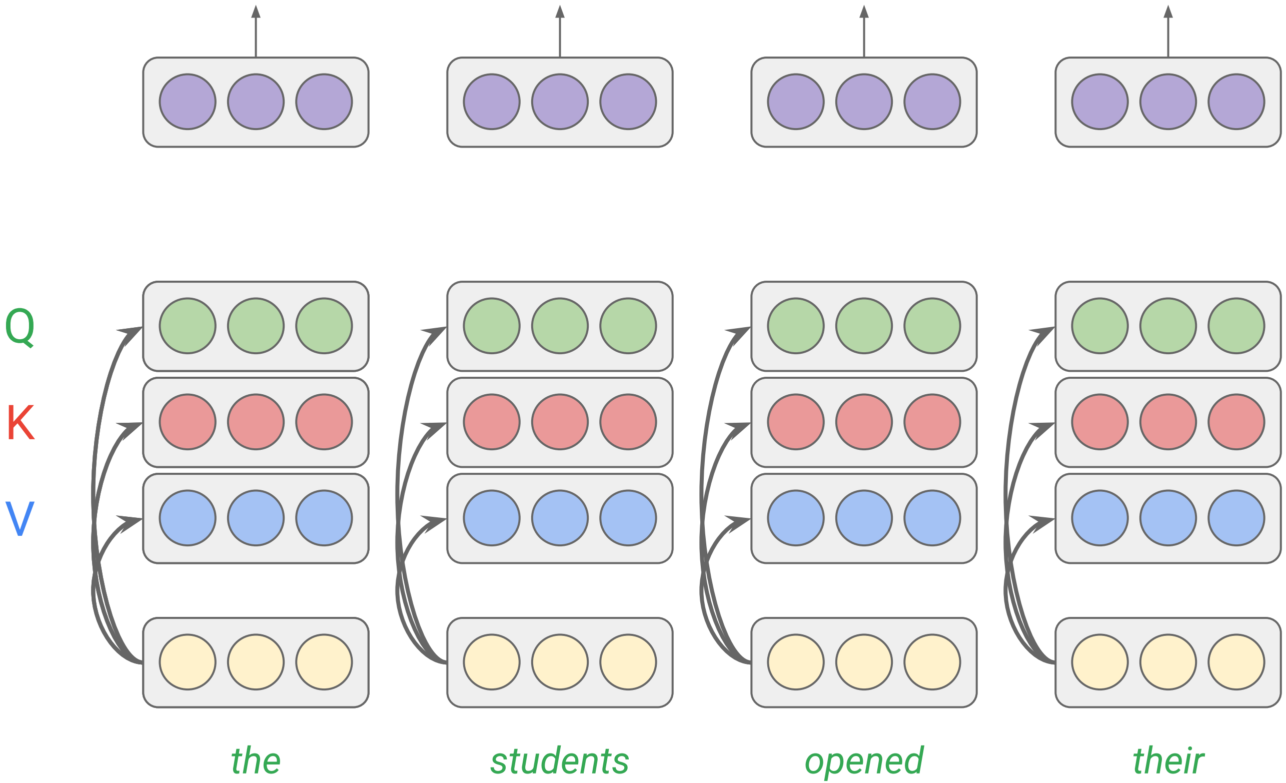
$Q = X \cdot W_Q$

$K = X \cdot W_K$

$V = X \cdot W_V$

linear projections

Attention (cont'd)



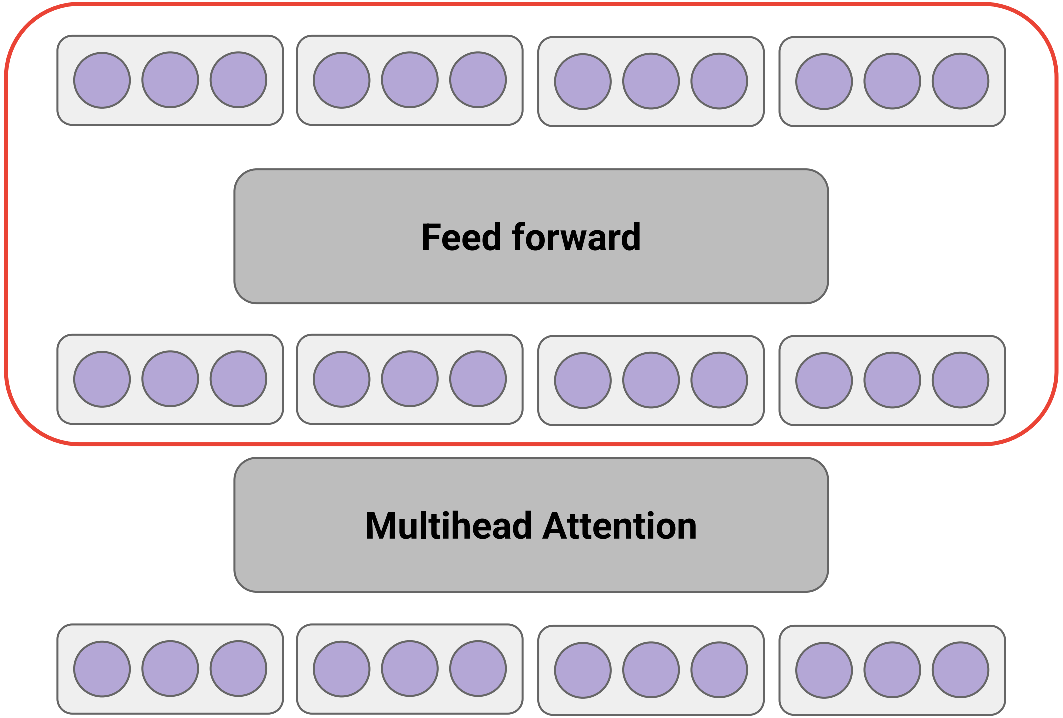
$$Q = X \cdot W_Q$$

$$K = X \cdot W_K$$

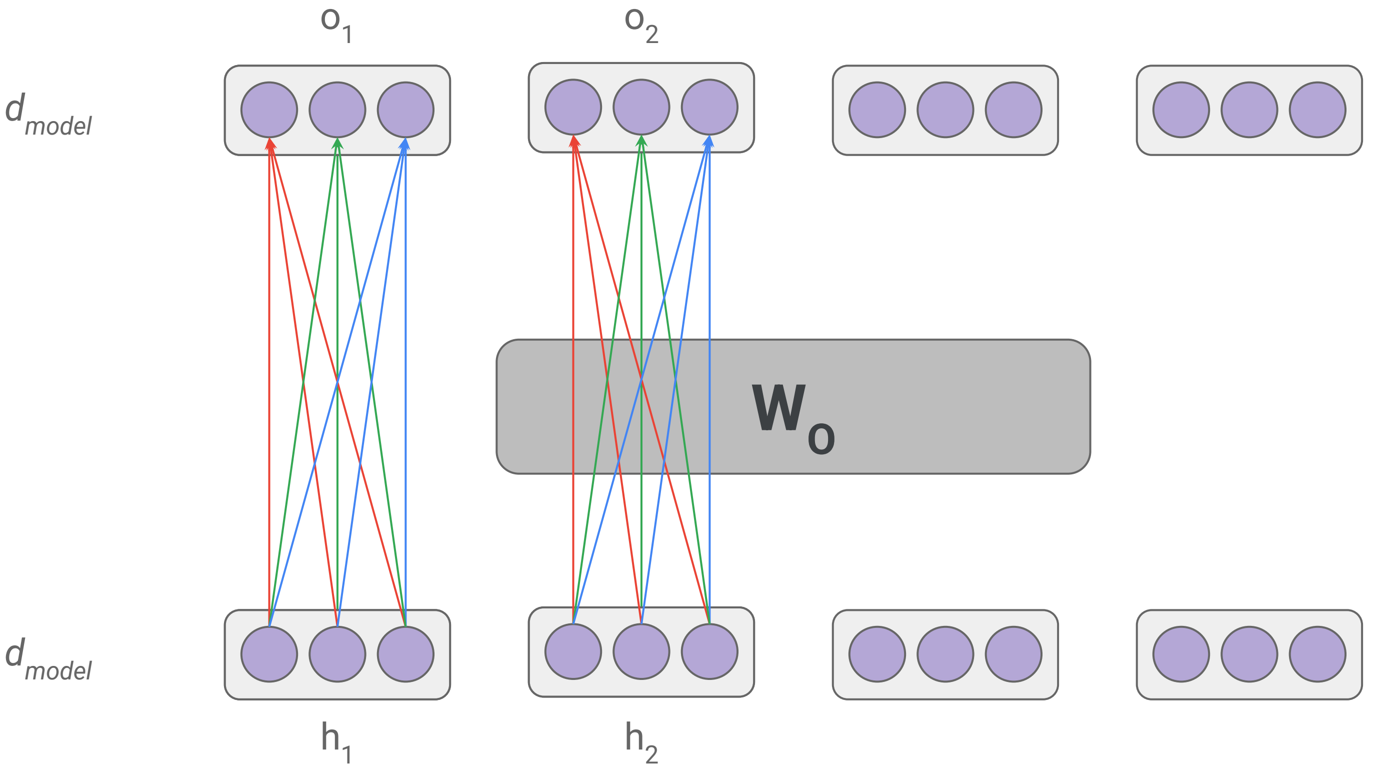
$$V = X \cdot W_V$$

linear projections

Transformer decoder



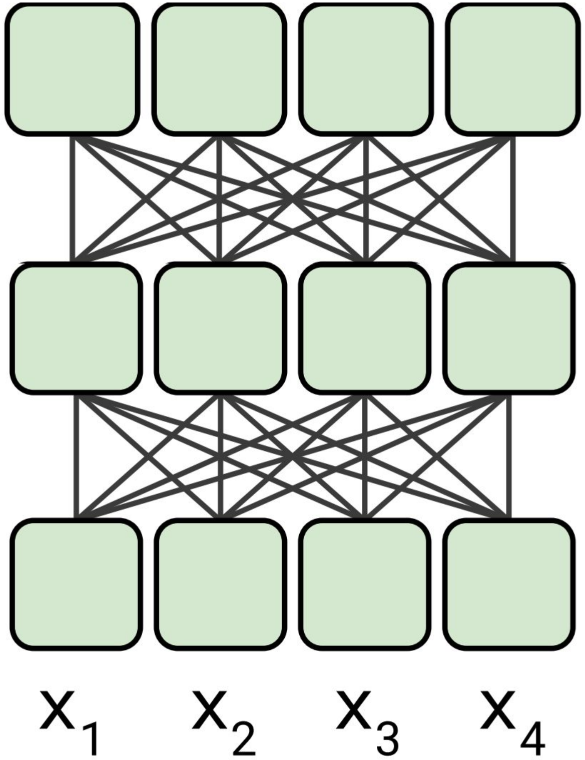
output vectors



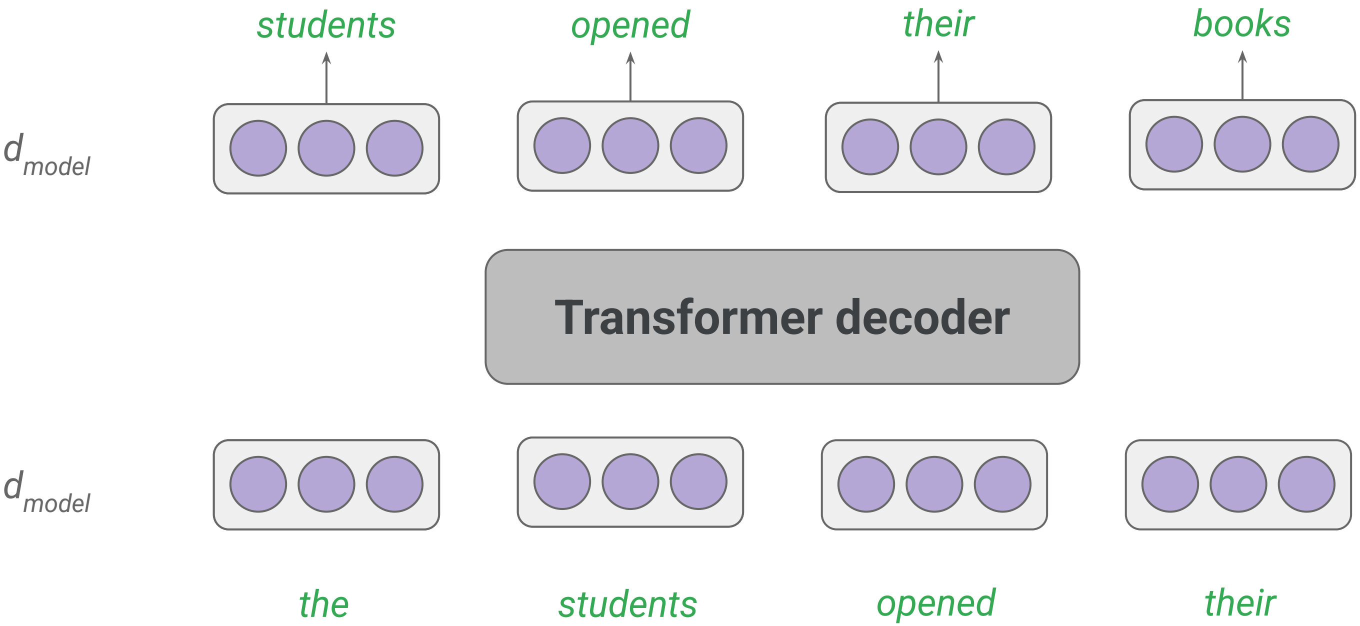
$$O = H \cdot W_O$$

linear projections

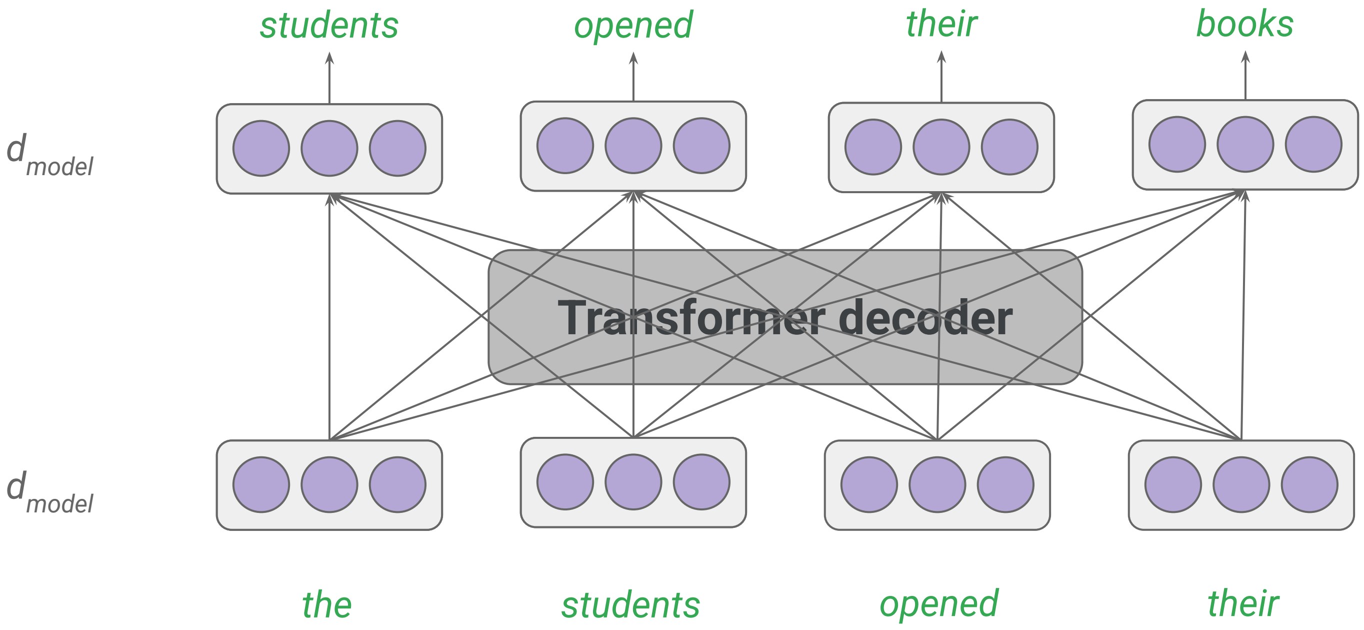
Transformer decoder (unmasked)



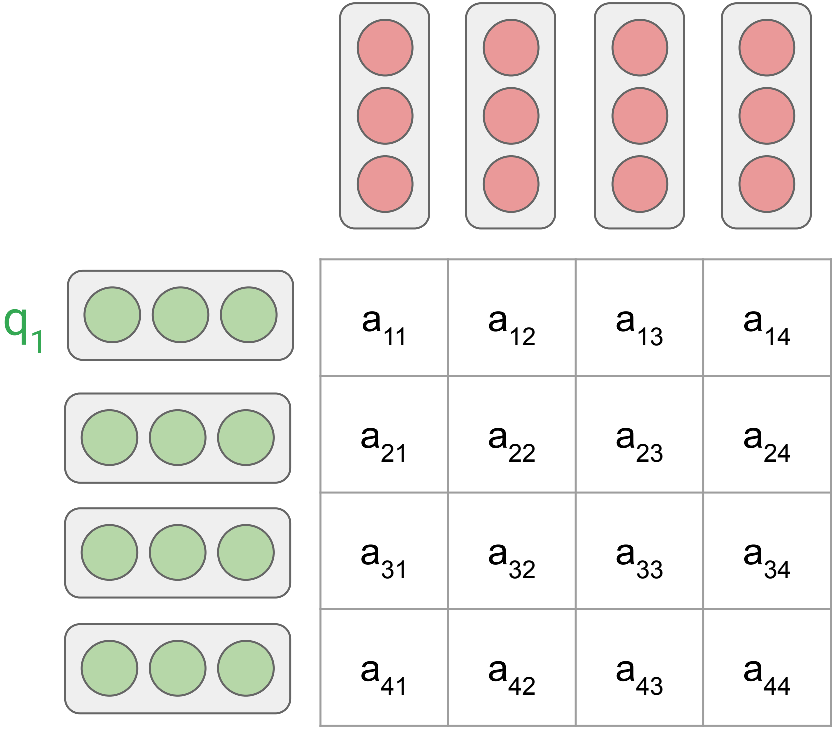
Transformer decoder



Transformer decoder (unmasked)



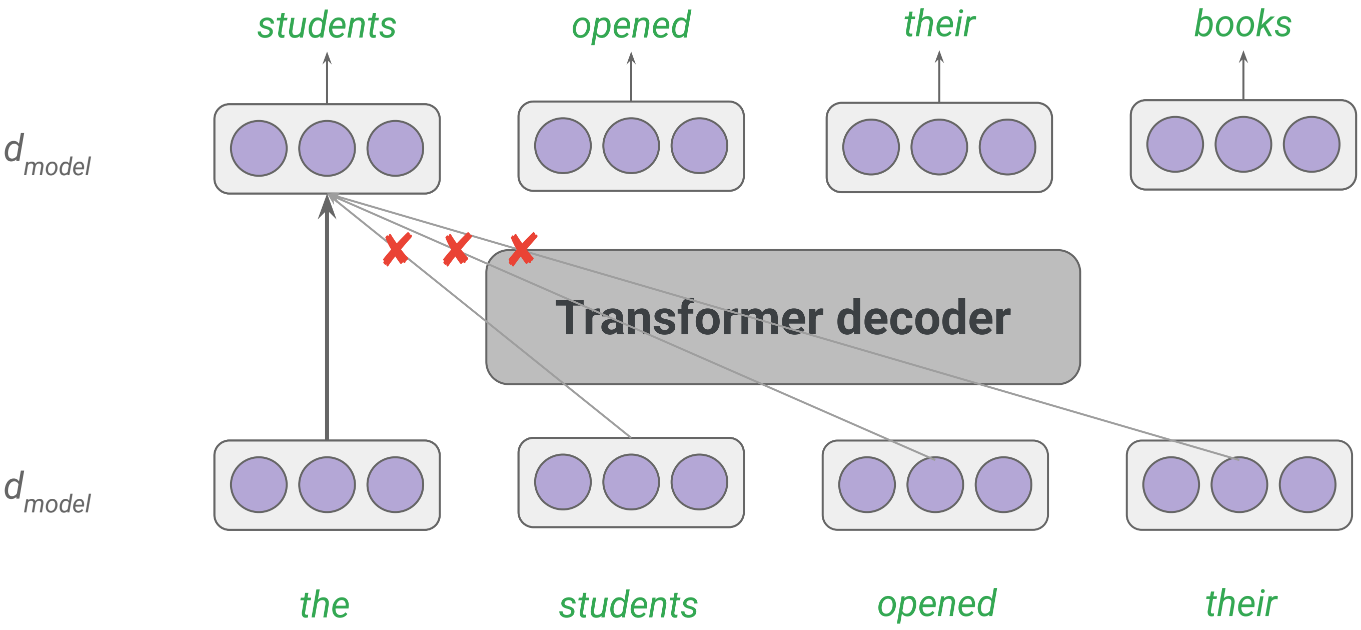
Attention matrix



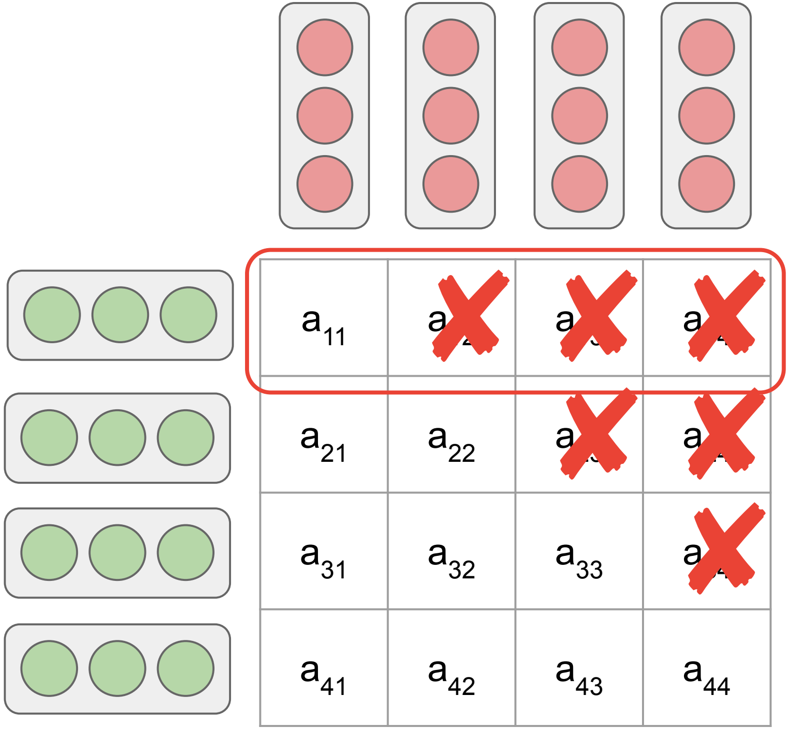
a_{11}	a_{12}	a_{13}	a_{14}
a_{21}	a_{22}	a_{23}	a_{24}
a_{31}	a_{32}	a_{33}	a_{34}
a_{41}	a_{42}	a_{43}	a_{44}

The time complexity of self-attention is quadratic in the input length $O(n^2)$

Transformer decoder

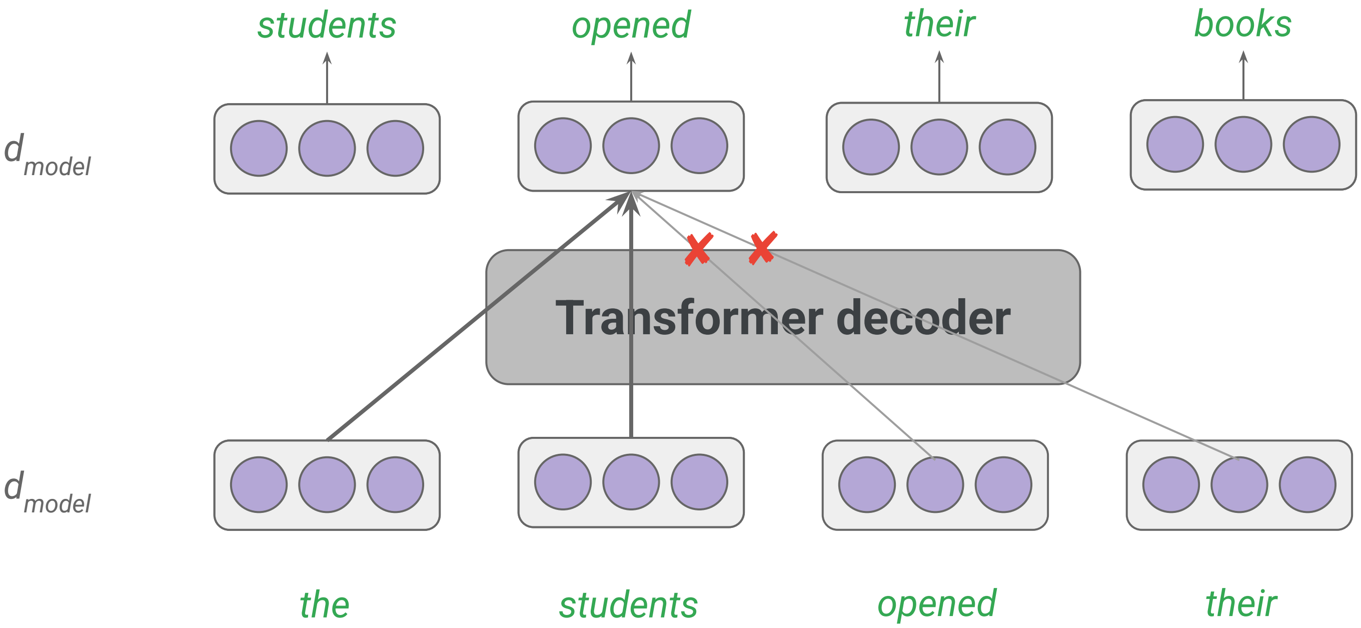


Attention matrix

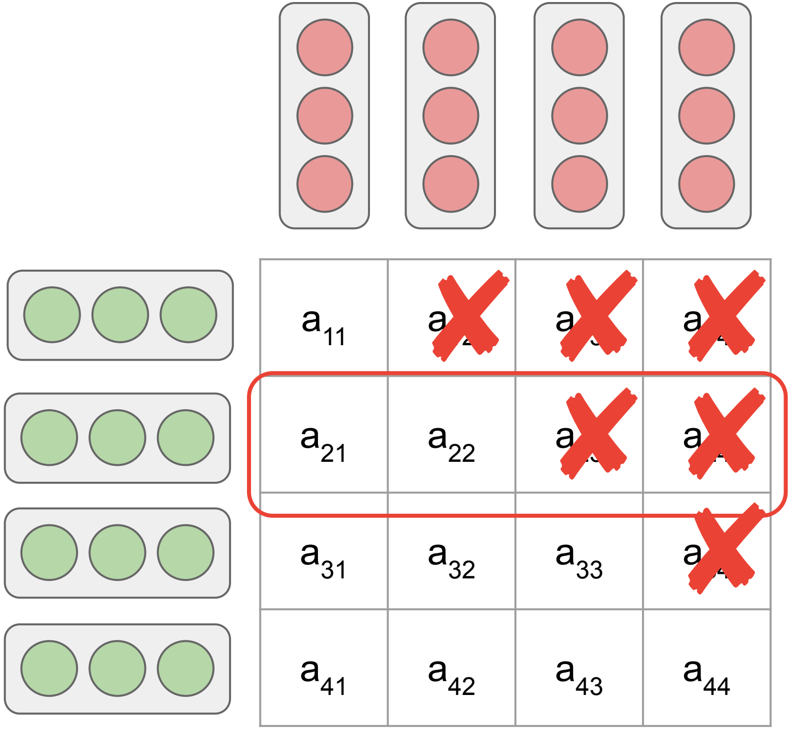


masking out all values in the input of the softmax which correspond to illegal connections

Transformer decoder



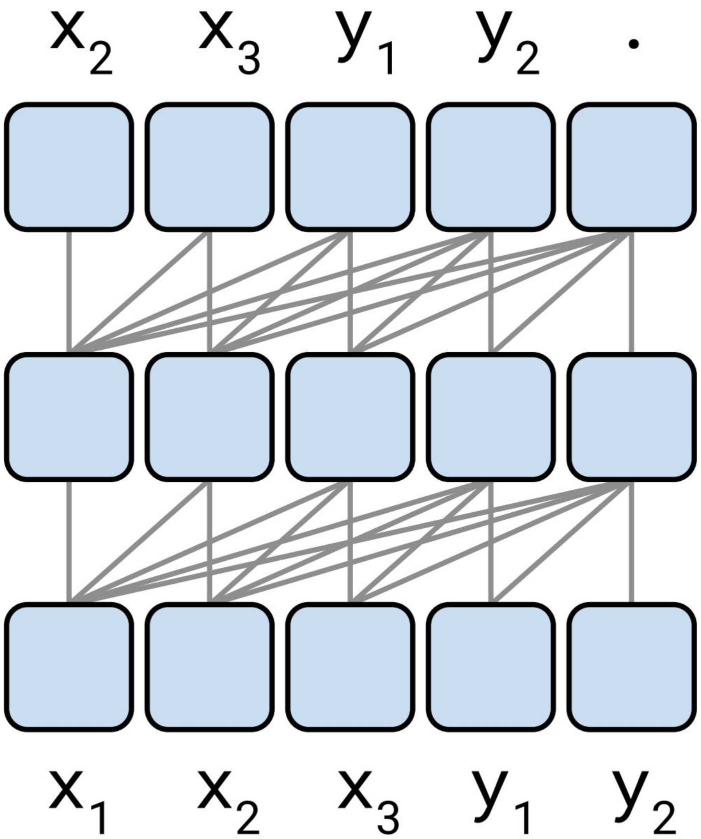
Attention matrix



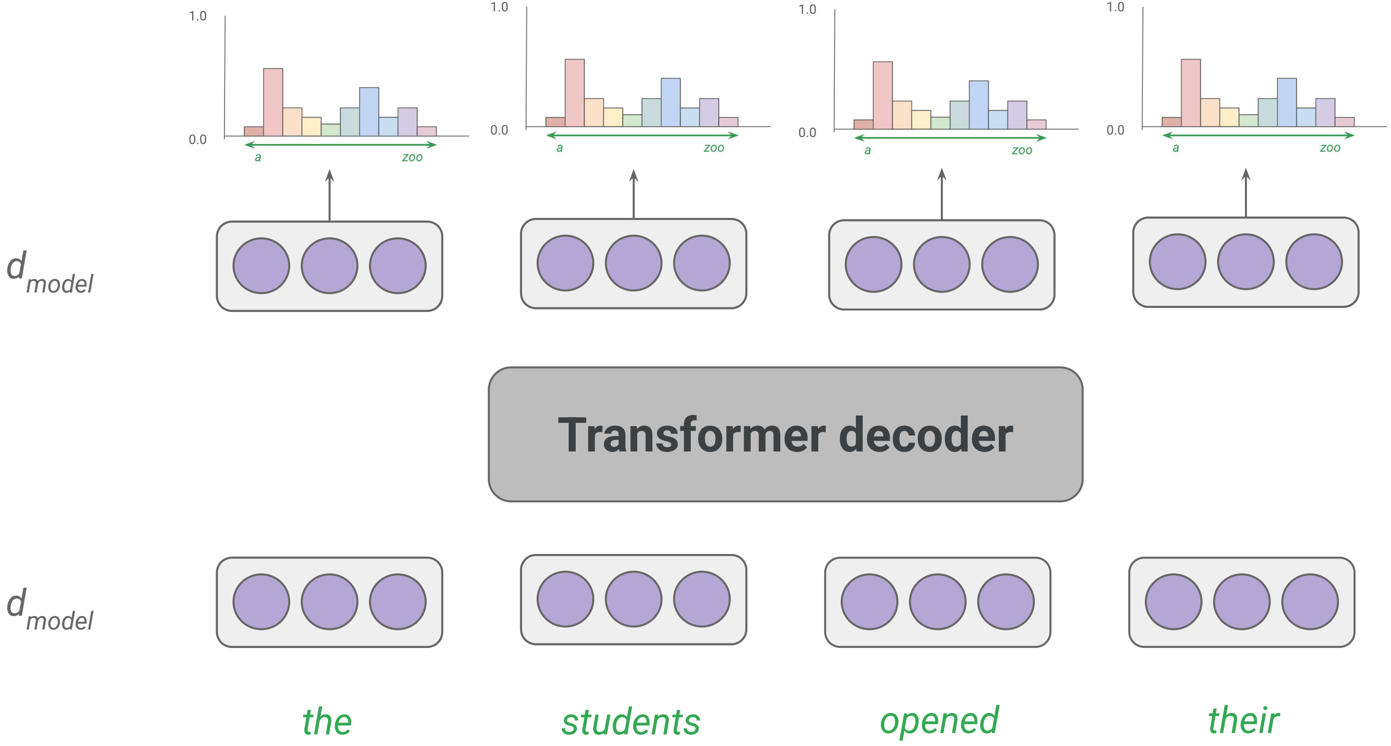
$$\begin{array}{cccc} a_{11} & \cancel{a_{12}} & \cancel{a_{13}} & \cancel{a_{14}} \\ a_{21} & a_{22} & \cancel{a_{23}} & \cancel{a_{24}} \\ a_{31} & a_{32} & a_{33} & \cancel{a_{34}} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}$$

masking out all values in the input of the softmax which correspond to illegal connections

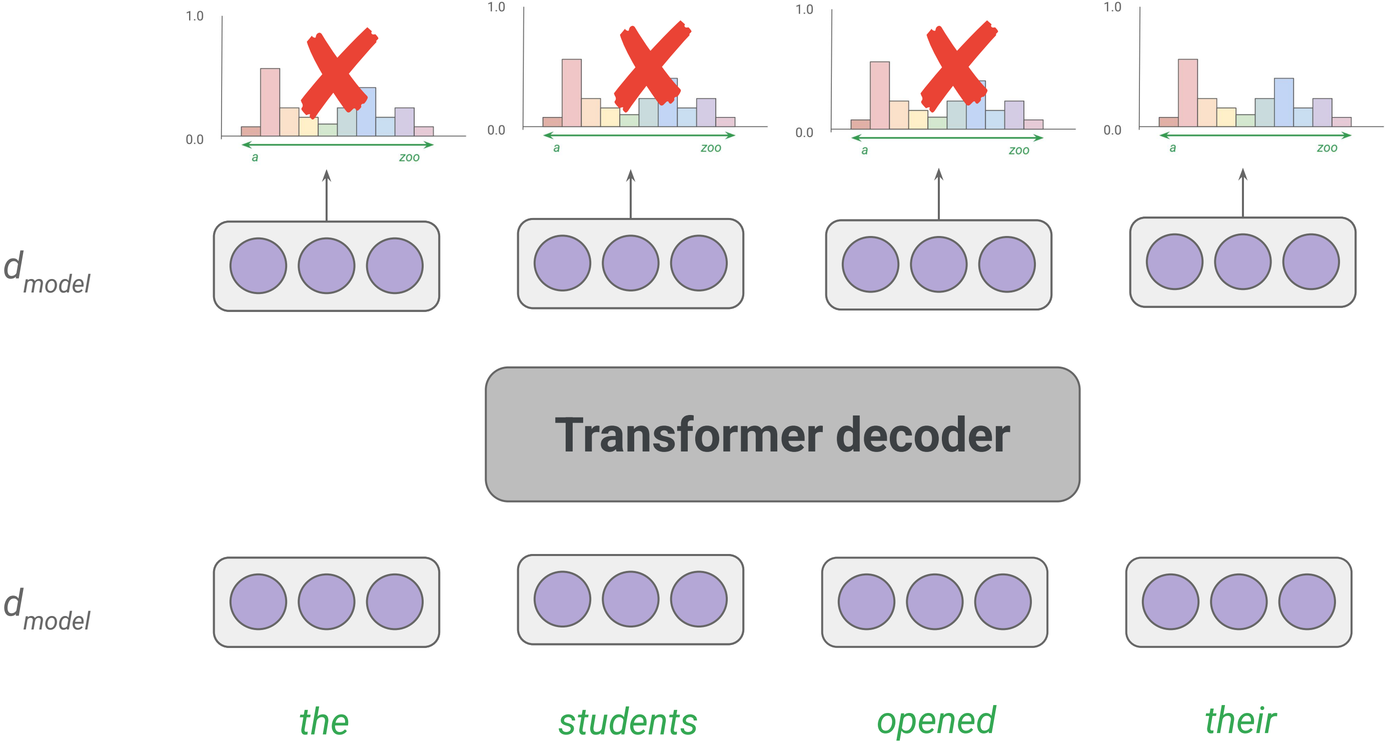
Transformer decoder (masked)



Transformer decoder: training



Transformer decoder: inference



Autoregressive decoding

- https://research.google/blog/looking-back-at-speculative-decoding/

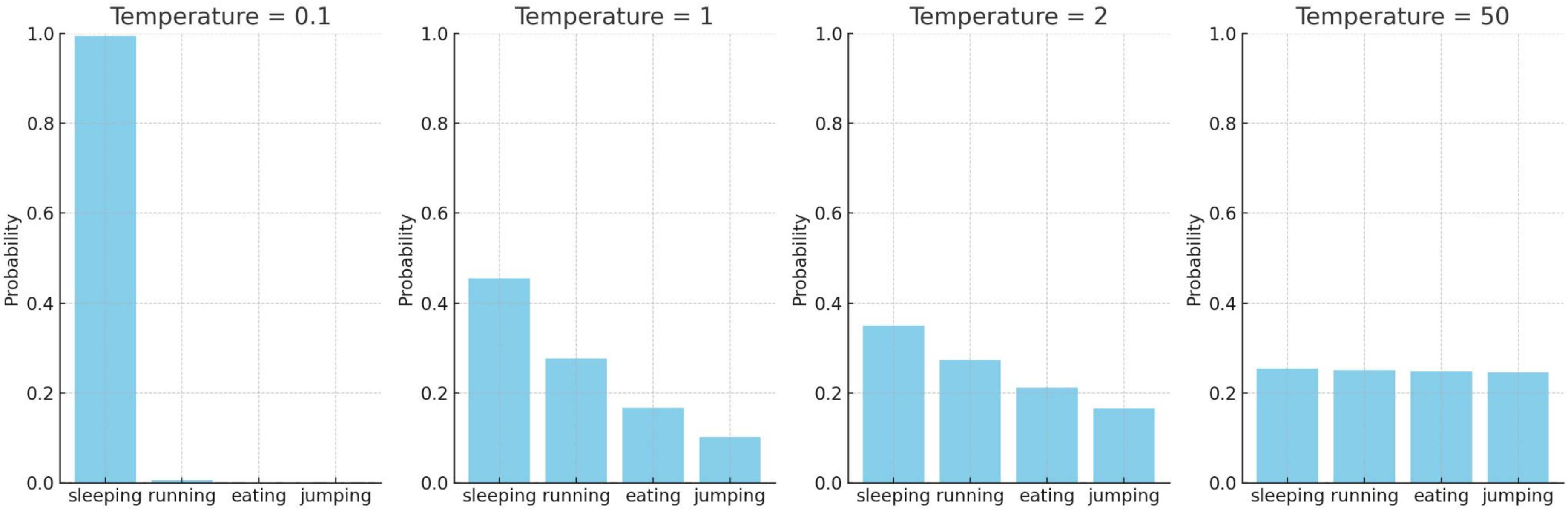
Temperature

$$P(y_i|\mathbf{x}) = \frac{\exp\left(\frac{z_i}{T}\right)}{\sum_j \exp\left(\frac{z_j}{T}\right)}$$

where:

- $P(y_i|\mathbf{x})$ is the probability of token y_i given the input $\mathbf{x}$
- z_i is the logit (raw score before softmax) for token y_i
- T is the temperature (where $T = 1$ is the default, and $T < 1$ reduces randomness while $T > 1$ increases randomness)
- The summation in the denominator is over all possible tokens j

Temperature (cont'd)



Temperature = 0.1

Temperature = 1

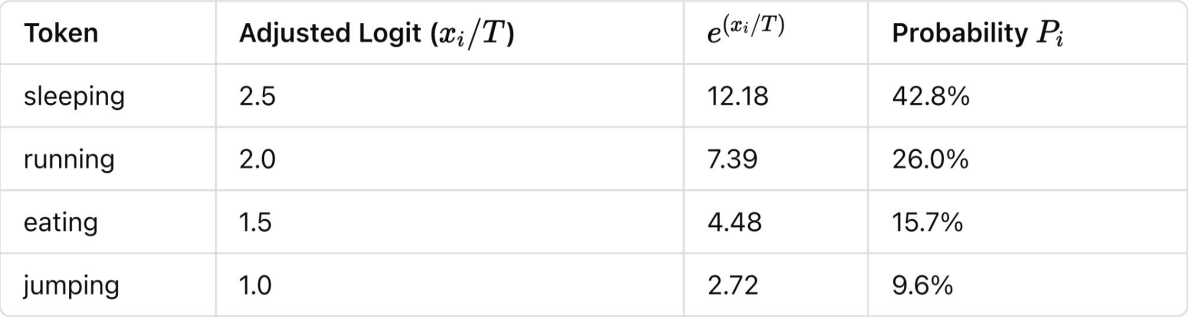
Temperature = 2

Temperature = 50

peaked distribution (more deterministic)

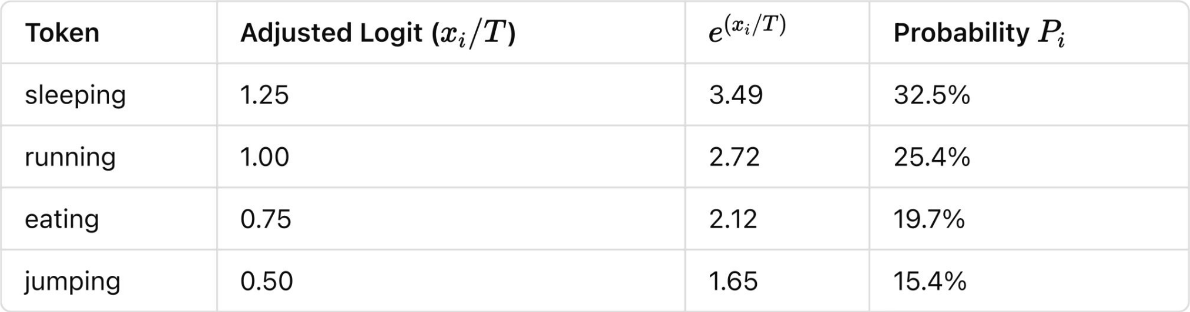
flatter distribution (more randomness)

“The cat is” → [sleeping, running, eating, jumping]



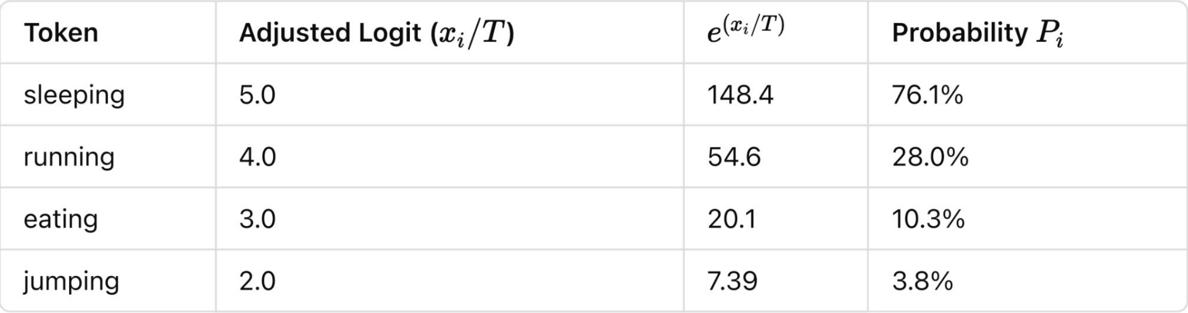
Token	Adjusted Logit (x_i/T)	$e^{(x_i/T)}$	Probability P_i
sleeping	2.5	12.18	42.8%
running	2.0	7.39	26.0%
eating	1.5	4.48	15.7%
jumping	1.0	2.72	9.6%

default T = 1.0
→ balanced



Token	Adjusted Logit (x_i/T)	$e^{(x_i/T)}$	Probability P_i
sleeping	1.25	3.49	32.5%
running	1.00	2.72	25.4%
eating	0.75	2.12	19.7%
jumping	0.50	1.65	15.4%

T = 2.0
→ flatter distribution
(more randomness)



Token	Adjusted Logit (x_i/T)	$e^{(x_i/T)}$	Probability P_i
sleeping	5.0	148.4	76.1%
running	4.0	54.6	28.0%
eating	3.0	20.1	10.3%
jumping	2.0	7.39	3.8%

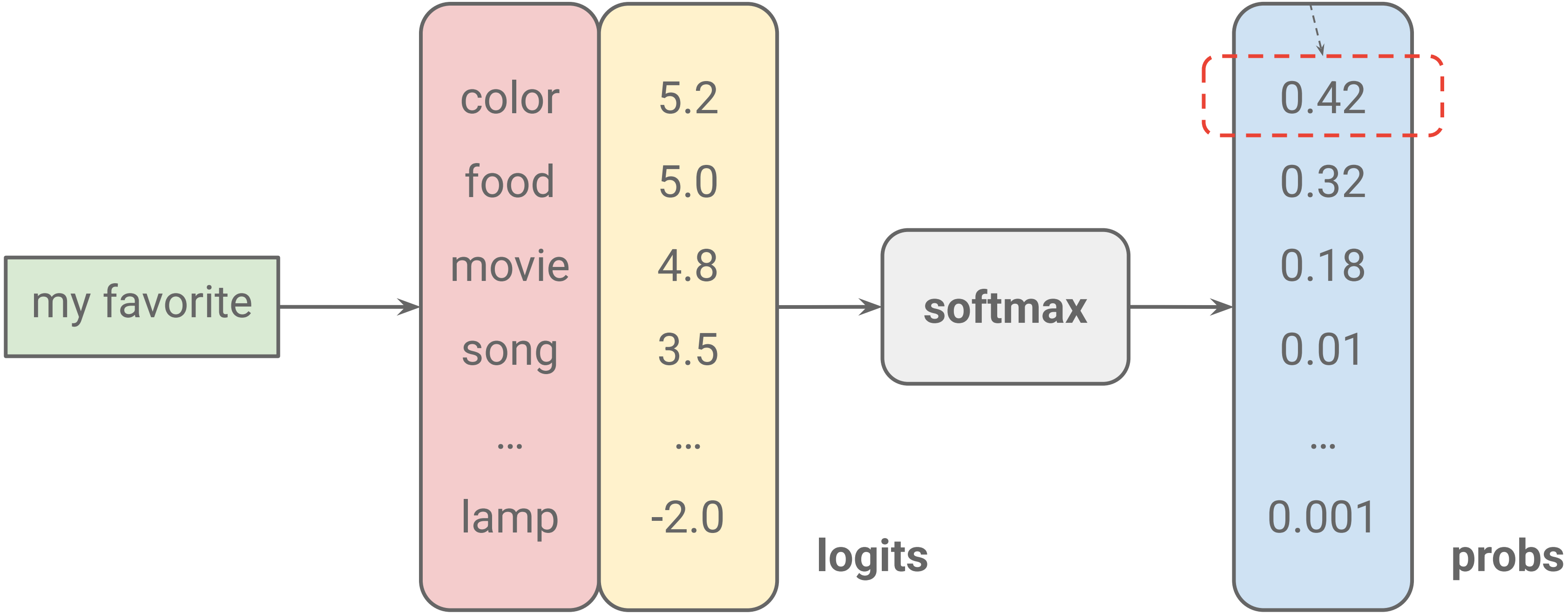
T = 0.5
→ peaked distribution
(more deterministic)

Temperature

- **Low temperature (T < 1, e.g., 0.2-0.5):**
  - more deterministic and predictable, favoring high-probability predictions
  - more factual but less diverse, resulting in repetitive or conservative responses
  - useful for tasks requiring precise answers (e.g., factual QA)
- **High temperature (T > 1, e.g., 1.2-2.0):**
  - more random and diverse, making token probabilities more uniform
  - increases creativity but may also result in less coherent or more unpredictable text
  - useful for tasks like storytelling or brainstorming
- **T = 1 (default setting):**
  - keeps the original probability distribution unchanged.
  - provides a balance between randomness and determinism.

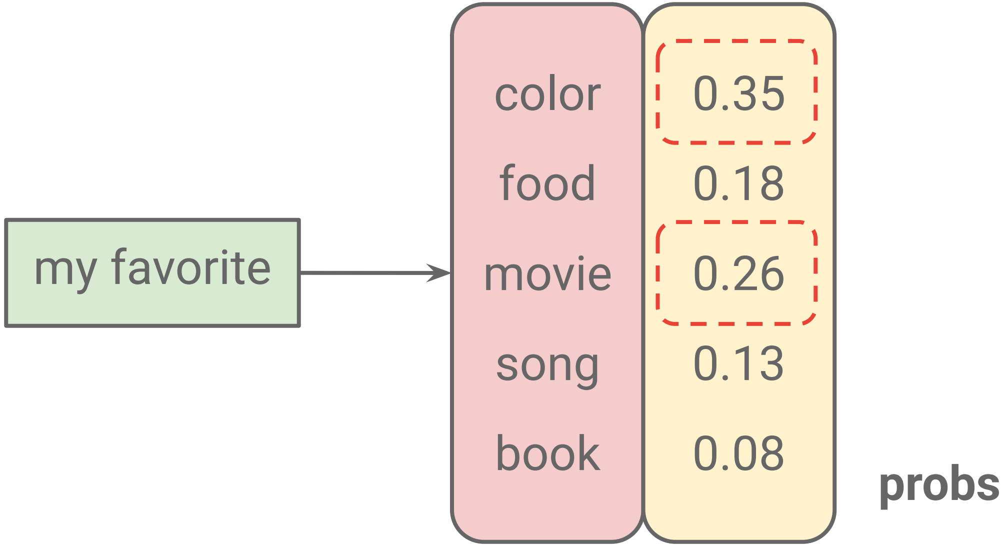
Greedy decoding

Selects the token with the highest probability at each step



Beam search

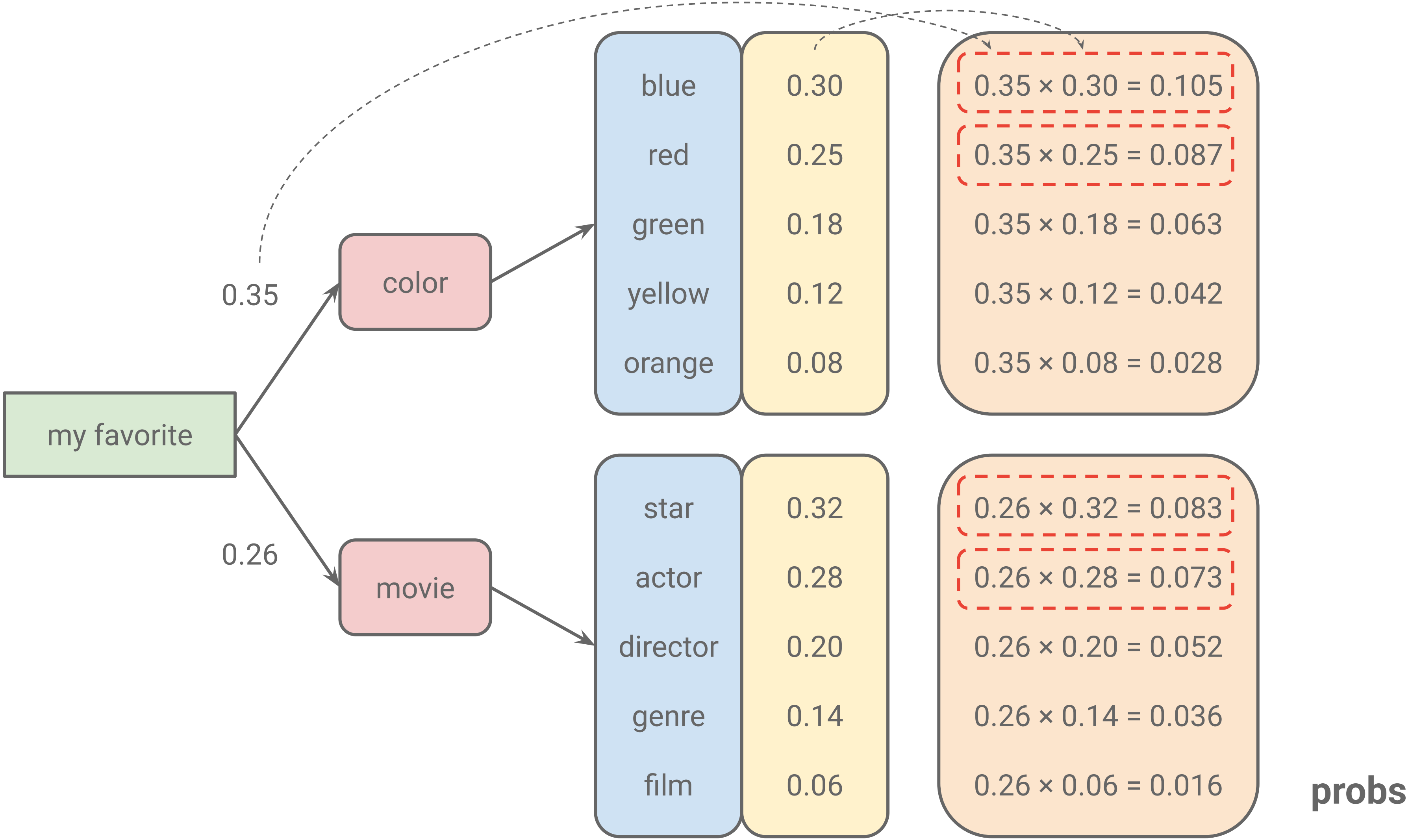
Maintains a set of *b* candidate sequences at each step instead of just keeping the single best one.



my favorite

color	0.35
food	0.18
movie	0.26
song	0.13
book	0.08

probs



my favorite

0.35

color

0.26

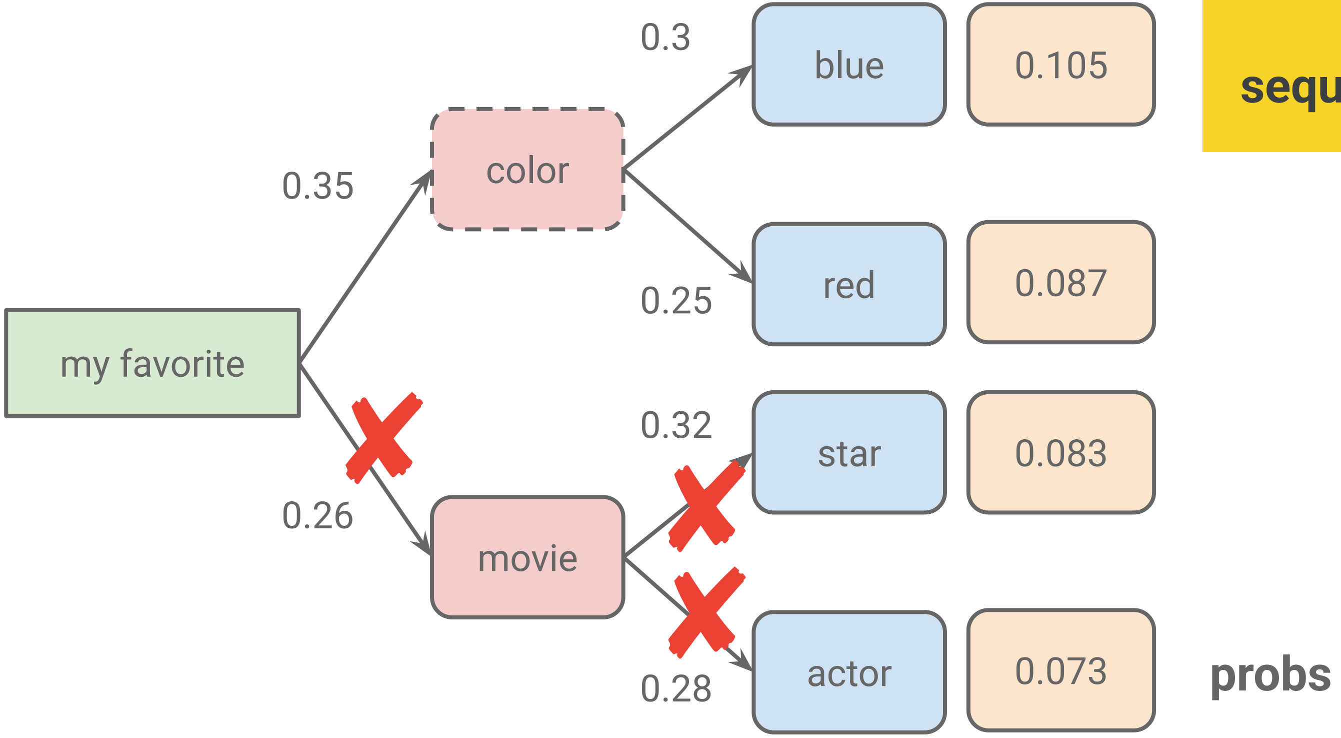
movie

blue	0.30	0.35 × 0.30 = 0.105
red	0.25	0.35 × 0.25 = 0.087
green	0.18	0.35 × 0.18 = 0.063
yellow	0.12	0.35 × 0.12 = 0.042
orange	0.08	0.35 × 0.08 = 0.028

star	0.32	0.26 × 0.32 = 0.083
actor	0.28	0.26 × 0.28 = 0.073
director	0.20	0.26 × 0.20 = 0.052
genre	0.14	0.26 × 0.14 = 0.036
film	0.06	0.26 × 0.06 = 0.016

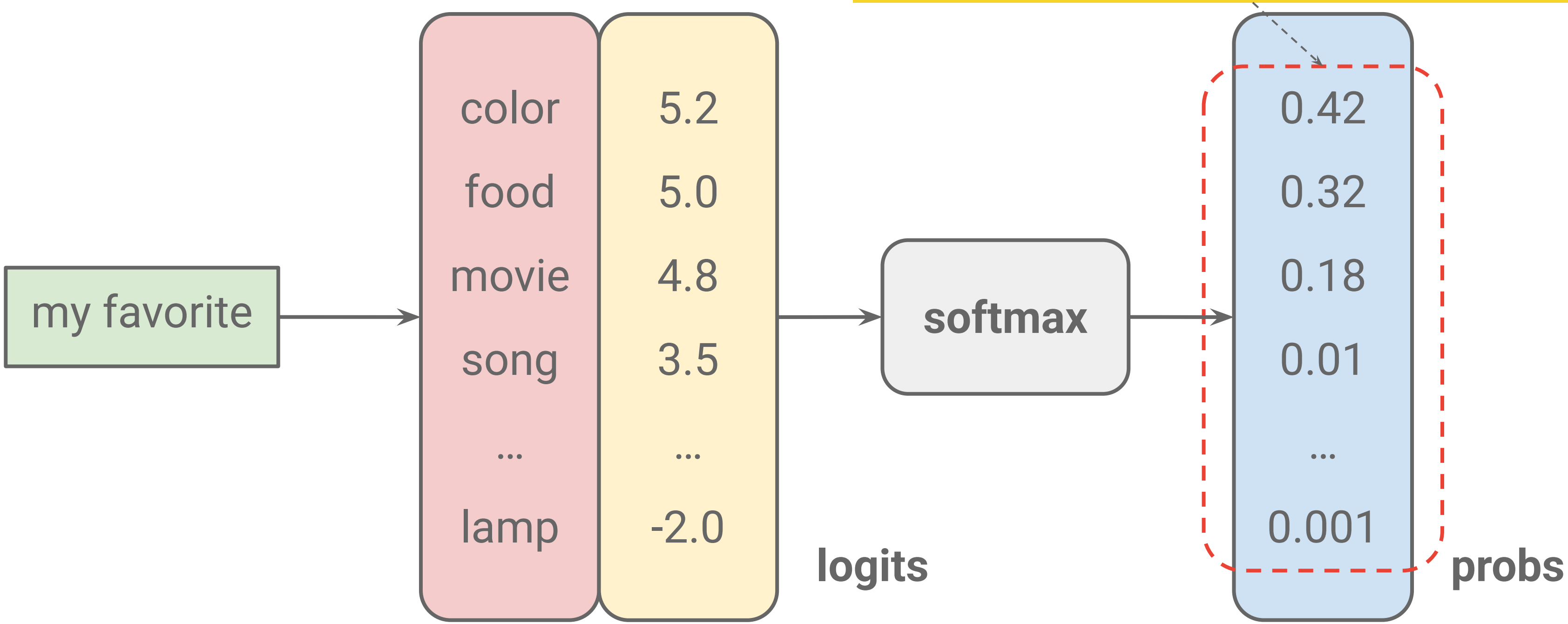
probs

Pruning: maintains a set of *b* candidate sequences at each step



Pure sampling

Samples from the *entire* probability distribution over the next token, with each token sampled according to its own probability, not uniformly



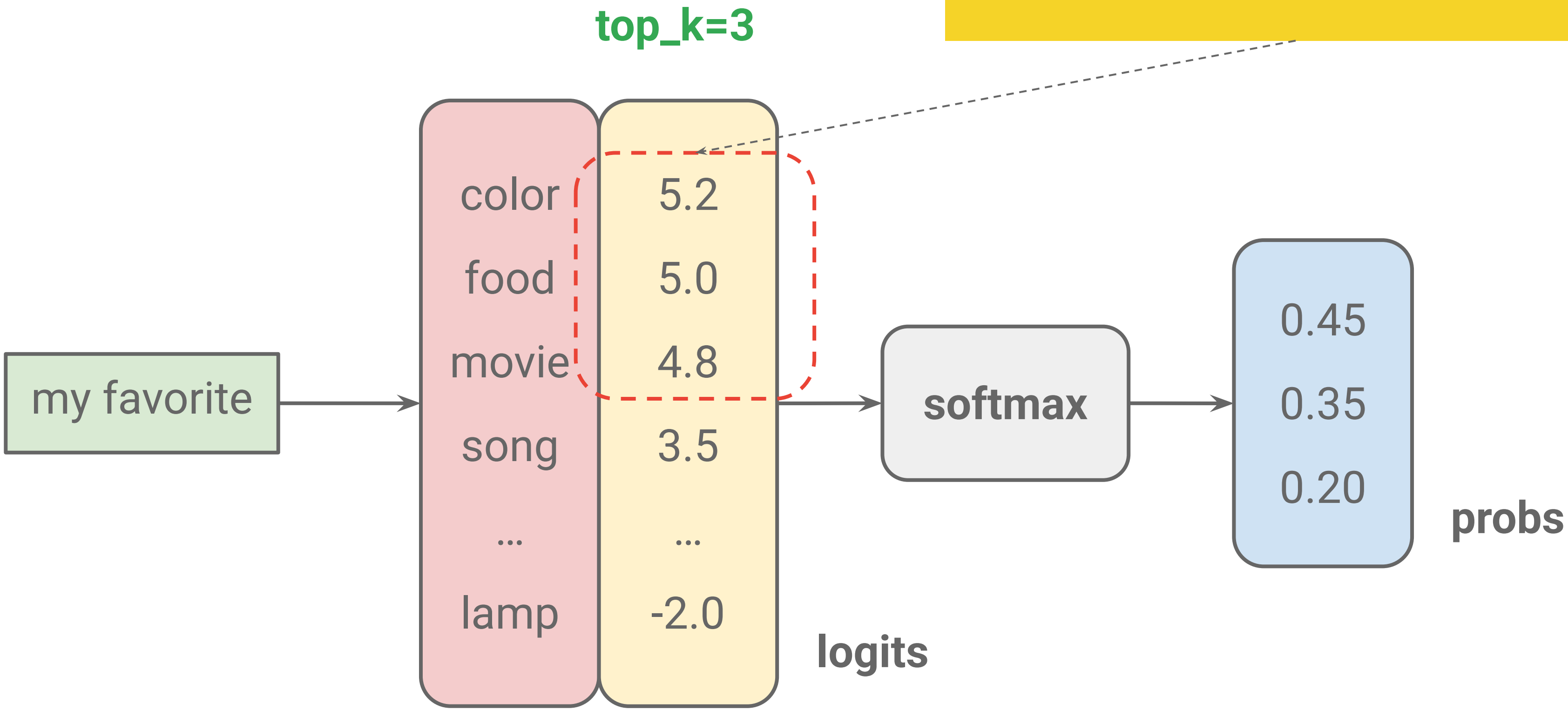
my favorite

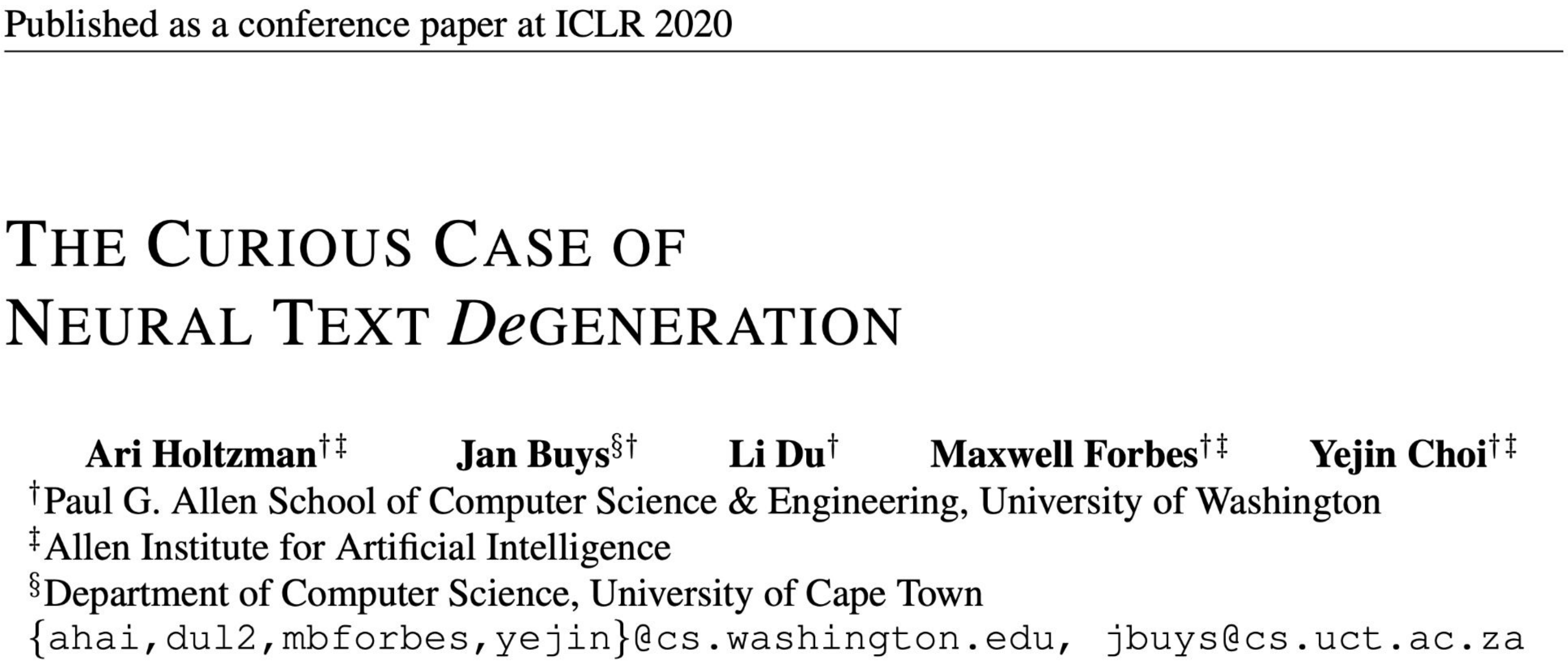
softmax

	logits	probs
color	5.2	0.42
food	5.0	0.32
movie	4.8	0.18
song	3.5	0.01
…	…	…
lamp	-2.0	0.001

Top-k sampling

Limits the vocabulary to the *k* most probable words at each step before applying softmax





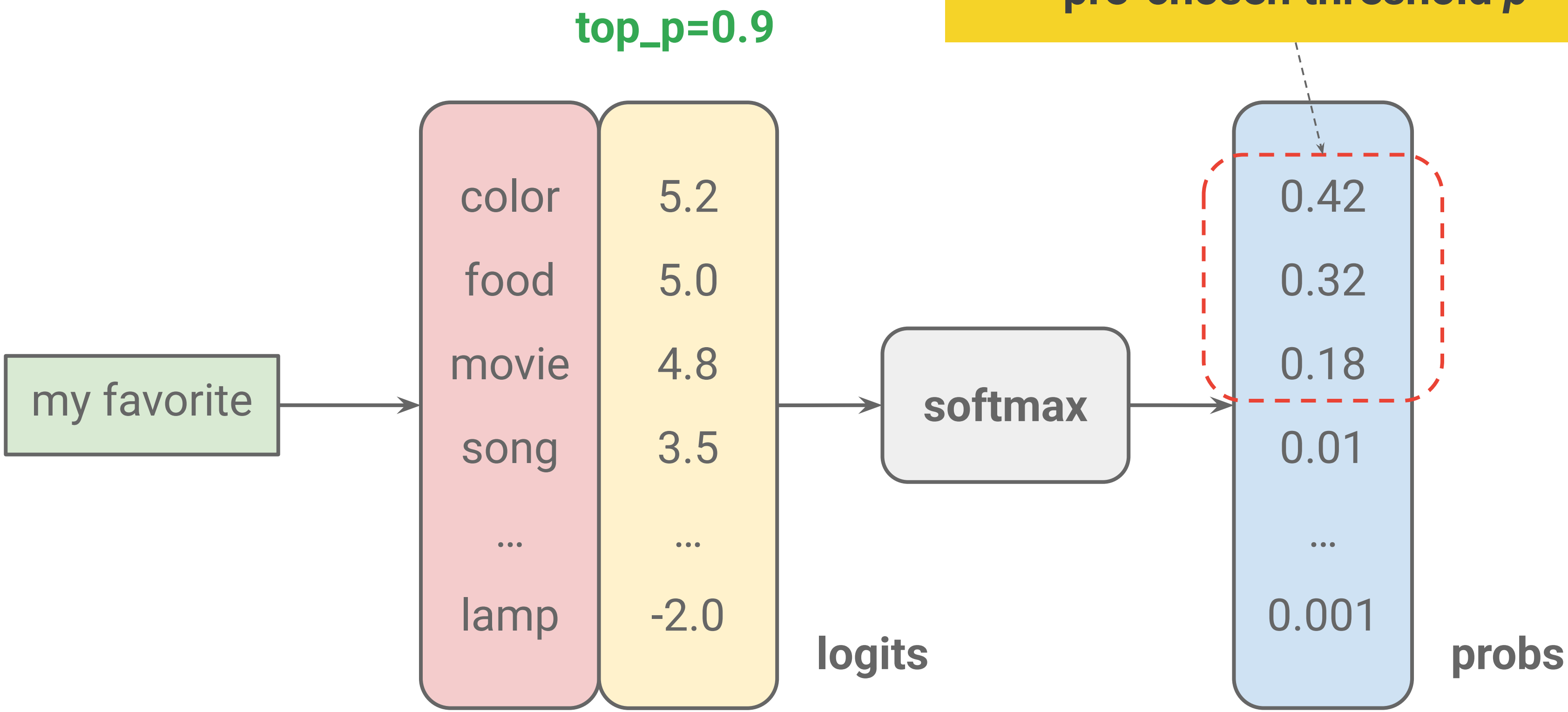
Published as a conference paper at ICLR 2020

THE CURIOUS CASE OF NEURAL TEXT *De*GENERATION

Ari Holtzman[†‡] **Jan Buys**[§†] **Li Du**[†] **Maxwell Forbes**[†‡] **Yejin Choi**[†‡]
[†]Paul G. Allen School of Computer Science & Engineering, University of Washington
[‡]Allen Institute for Artificial Intelligence
[§]Department of Computer Science, University of Cape Town
{ahai,dul2,mbforbes,yejin}@cs.washington.edu, jbuys@cs.uct.ac.za

Top-p (nucleus) sampling

Selects the highest probability tokens whose cumulative probability mass exceeds the pre-chosen threshold *p*



top_p=0.9

my favorite

color
food
movie
song
…
lamp

5.2
5.0
4.8
3.5
…
-2.0

logits

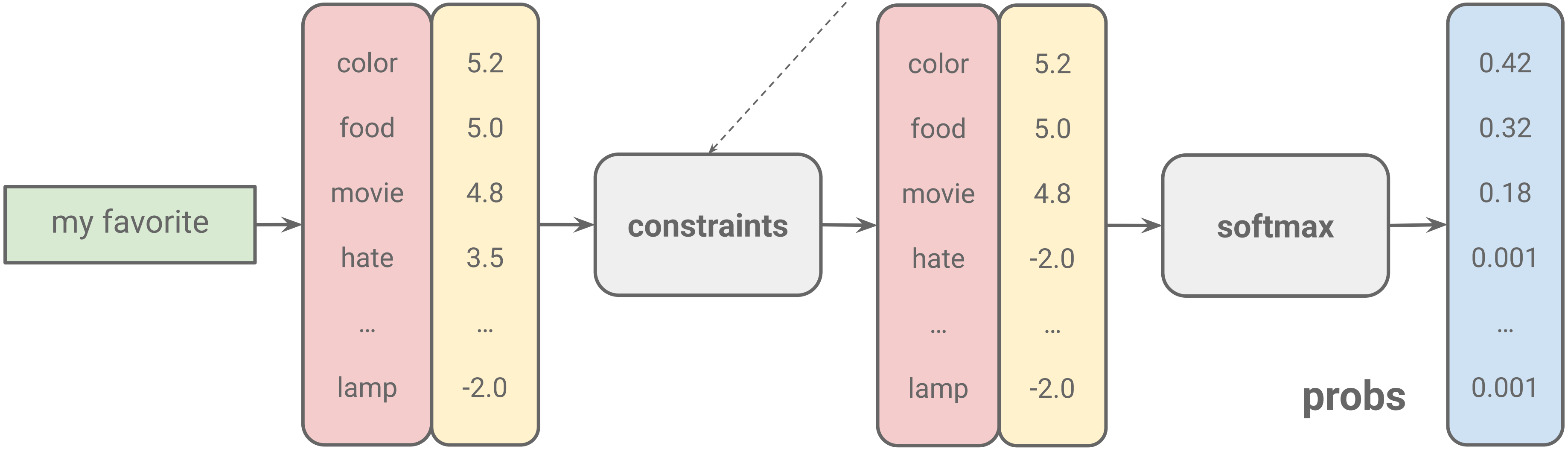
softmax

0.42
0.32
0.18
0.01
…
0.001

probs

Constrained decoding

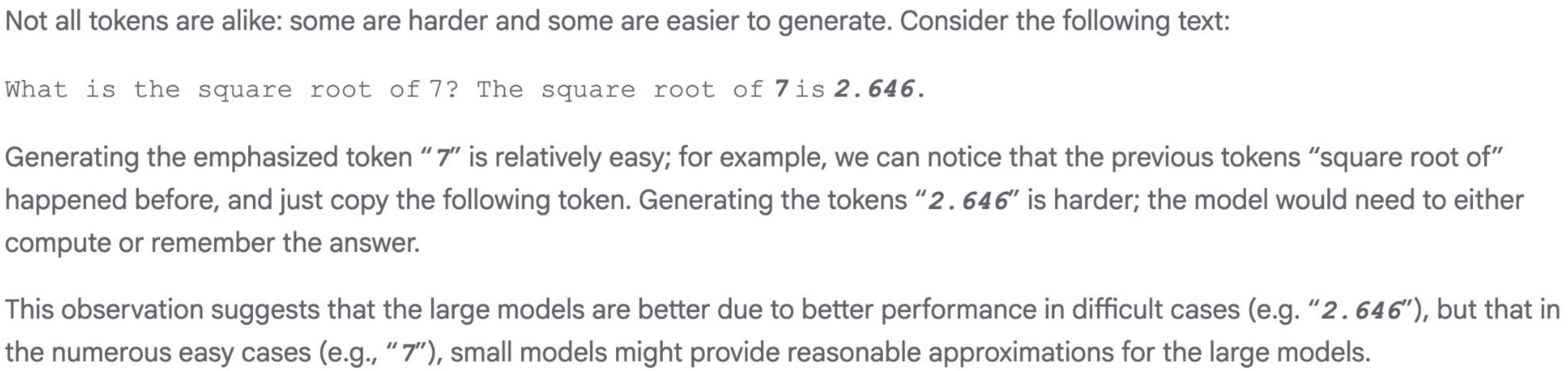
generates sequences that must satisfy certain predefined conditions or constraints



Speculative decoding

- https://research.google/blog/looking-back-at-speculative-decoding/

Observation 1: Some tokens are easier to generate than others



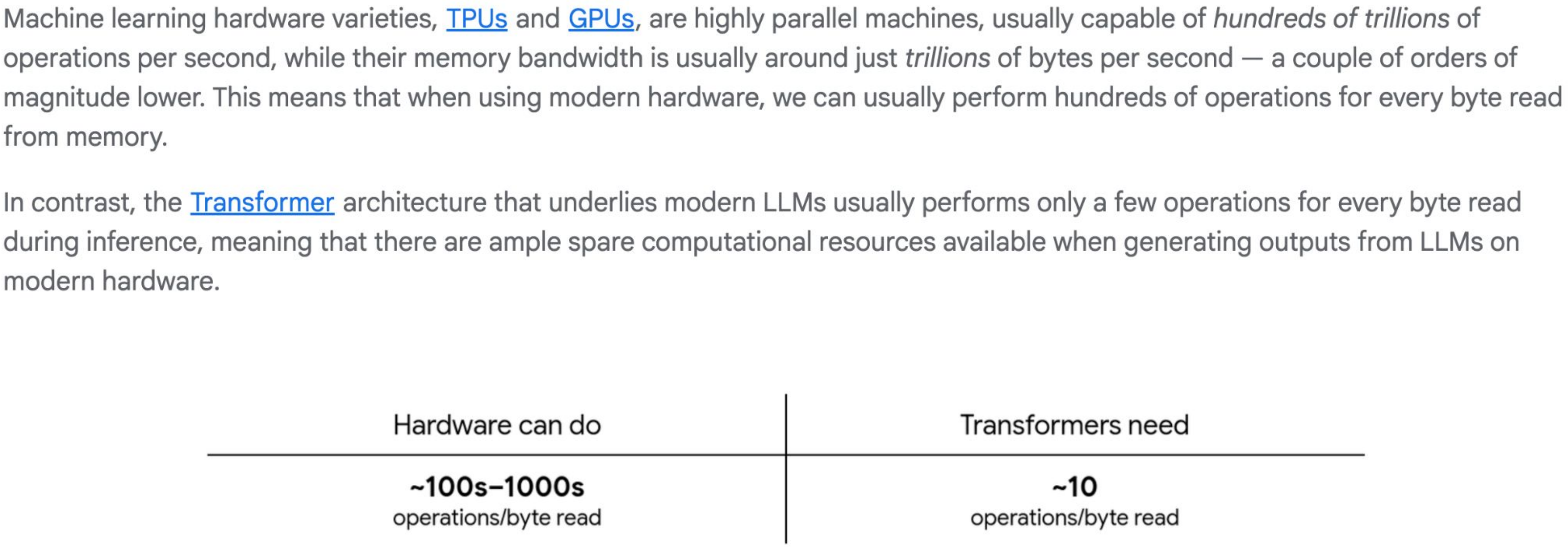
Not all tokens are alike: some are harder and some are easier to generate. Consider the following text:

What is the square root of 7? The square root of ***7*** is ***2.646.***

Generating the emphasized token "*7*" is relatively easy; for example, we can notice that the previous tokens "square root of" happened before, and just copy the following token. Generating the tokens "***2.646***" is harder; the model would need to either compute or remember the answer.

This observation suggests that the large models are better due to better performance in difficult cases (e.g. "***2.646***"), but that in the numerous easy cases (e.g., "*7*"), small models might provide reasonable approximations for the large models.

Observation 2: The bottleneck for LLM inference is usually memory

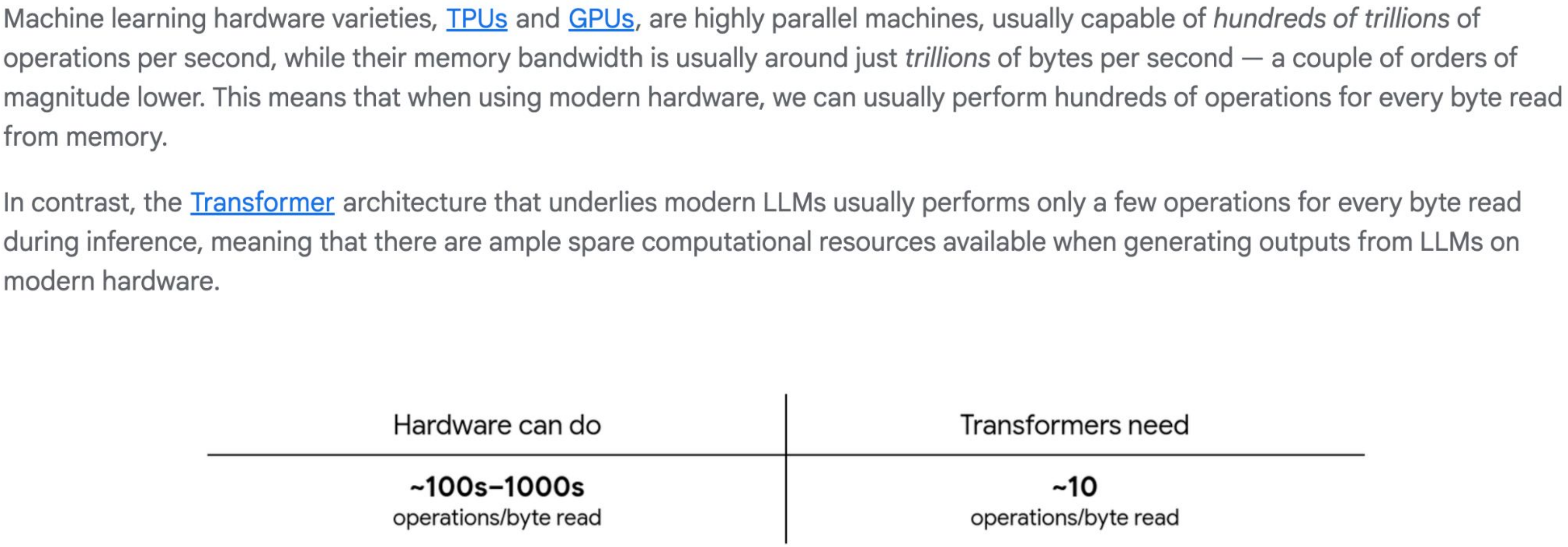


Machine learning hardware varieties, TPUs and GPUs, are highly parallel machines, usually capable of *hundreds of trillions* of operations per second, while their memory bandwidth is usually around just *trillions* of bytes per second — a couple of orders of magnitude lower. This means that when using modern hardware, we can usually perform hundreds of operations for every byte read from memory.

In contrast, the Transformer architecture that underlies modern LLMs usually performs only a few operations for every byte read during inference, meaning that there are ample spare computational resources available when generating outputs from LLMs on modern hardware.

Hardware can do	Transformers need
~100s–1000s operations/byte read	**~10** operations/byte read

Observation 2: The bottleneck for LLM inference is usually memory

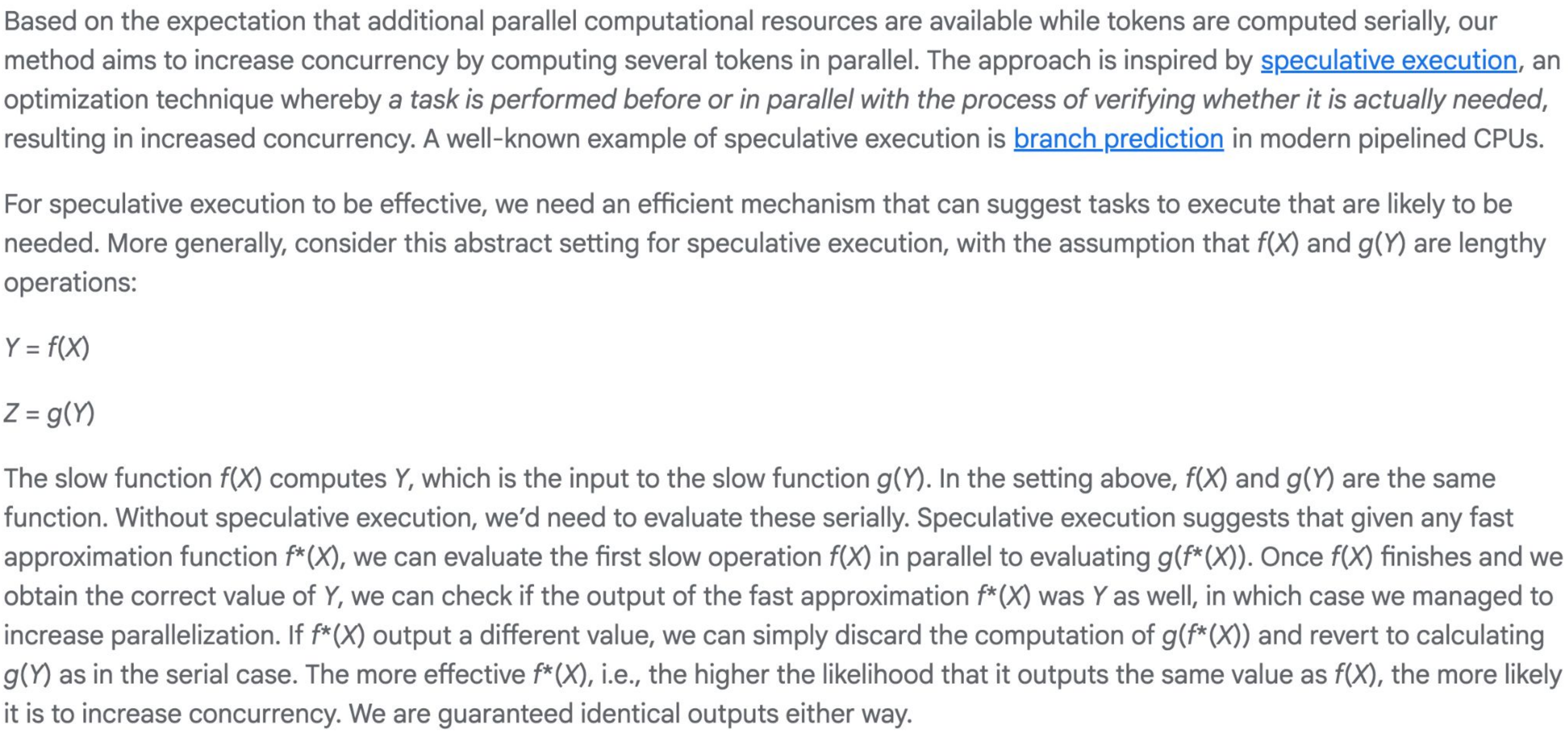


Machine learning hardware varieties, TPUs and GPUs, are highly parallel machines, usually capable of *hundreds of trillions* of operations per second, while their memory bandwidth is usually around just *trillions* of bytes per second — a couple of orders of magnitude lower. This means that when using modern hardware, we can usually perform hundreds of operations for every byte read from memory.

In contrast, the Transformer architecture that underlies modern LLMs usually performs only a few operations for every byte read during inference, meaning that there are ample spare computational resources available when generating outputs from LLMs on modern hardware.

Hardware can do	Transformers need
~100s–1000s operations/byte read	**~10** operations/byte read

Speculative execution



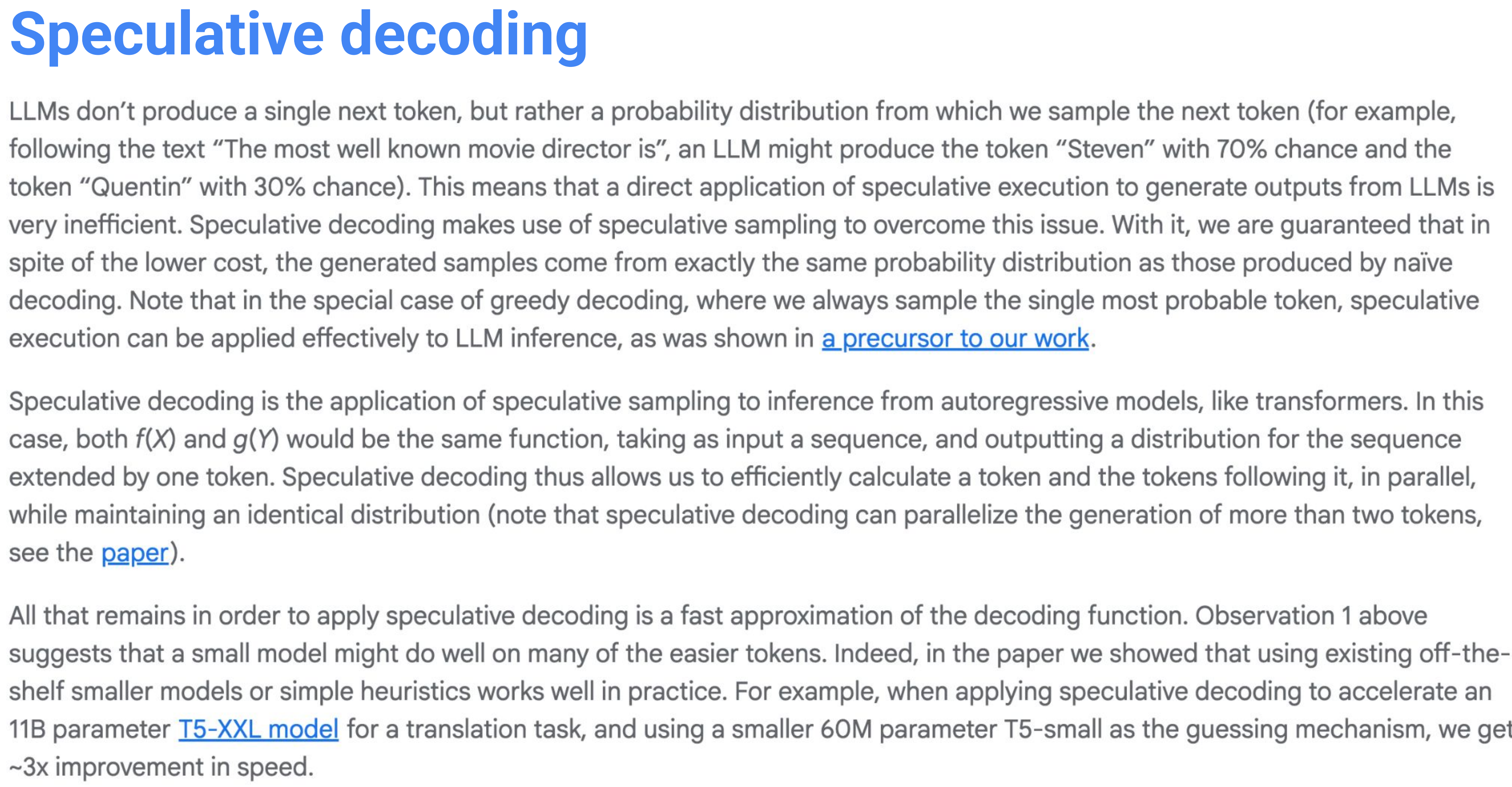
Based on the expectation that additional parallel computational resources are available while tokens are computed serially, our method aims to increase concurrency by computing several tokens in parallel. The approach is inspired by speculative execution, an optimization technique whereby *a task is performed before or in parallel with the process of verifying whether it is actually needed*, resulting in increased concurrency. A well-known example of speculative execution is branch prediction in modern pipelined CPUs.

For speculative execution to be effective, we need an efficient mechanism that can suggest tasks to execute that are likely to be needed. More generally, consider this abstract setting for speculative execution, with the assumption that $f(X)$ and $g(Y)$ are lengthy operations:

$Y = f(X)$

$Z = g(Y)$

The slow function $f(X)$ computes Y, which is the input to the slow function $g(Y)$. In the setting above, $f(X)$ and $g(Y)$ are the same function. Without speculative execution, we'd need to evaluate these serially. Speculative execution suggests that given any fast approximation function $f^*(X)$, we can evaluate the first slow operation $f(X)$ in parallel to evaluating $g(f^*(X))$. Once $f(X)$ finishes and we obtain the correct value of Y, we can check if the output of the fast approximation $f^*(X)$ was Y as well, in which case we managed to increase parallelization. If $f^*(X)$ output a different value, we can simply discard the computation of $g(f^*(X))$ and revert to calculating $g(Y)$ as in the serial case. The more effective $f^*(X)$, i.e., the higher the likelihood that it outputs the same value as $f(X)$, the more likely it is to increase concurrency. We are guaranteed identical outputs either way.



Speculative decoding

LLMs don't produce a single next token, but rather a probability distribution from which we sample the next token (for example, following the text "The most well known movie director is", an LLM might produce the token "Steven" with 70% chance and the token "Quentin" with 30% chance). This means that a direct application of speculative execution to generate outputs from LLMs is very inefficient. Speculative decoding makes use of speculative sampling to overcome this issue. With it, we are guaranteed that in spite of the lower cost, the generated samples come from exactly the same probability distribution as those produced by naïve decoding. Note that in the special case of greedy decoding, where we always sample the single most probable token, speculative execution can be applied effectively to LLM inference, as was shown in a precursor to our work.

Speculative decoding is the application of speculative sampling to inference from autoregressive models, like transformers. In this case, both *f*(*X*) and *g*(*Y*) would be the same function, taking as input a sequence, and outputting a distribution for the sequence extended by one token. Speculative decoding thus allows us to efficiently calculate a token and the tokens following it, in parallel, while maintaining an identical distribution (note that speculative decoding can parallelize the generation of more than two tokens, see the paper).

All that remains in order to apply speculative decoding is a fast approximation of the decoding function. Observation 1 above suggests that a small model might do well on many of the easier tokens. Indeed, in the paper we showed that using existing off-the-shelf smaller models or simple heuristics works well in practice. For example, when applying speculative decoding to accelerate an 11B parameter T5-XXL model for a translation task, and using a smaller 60M parameter T5-small as the guessing mechanism, we get ~3x improvement in speed.

Speculative decoding

- https://research.google/blog/looking-back-at-speculative-decoding/

Thank you!