Building LLM Reasoners

Lecture 2: Tokenizers, Optimizers, Tricks

Greg Durrett

Slide credit: Tatsu Hashimoto & Percy Liang, Stanford CS 336

Administrative details and recap

Tokenizers

Byte Pair Encoding

Decoding

Optimizers

Efficiency

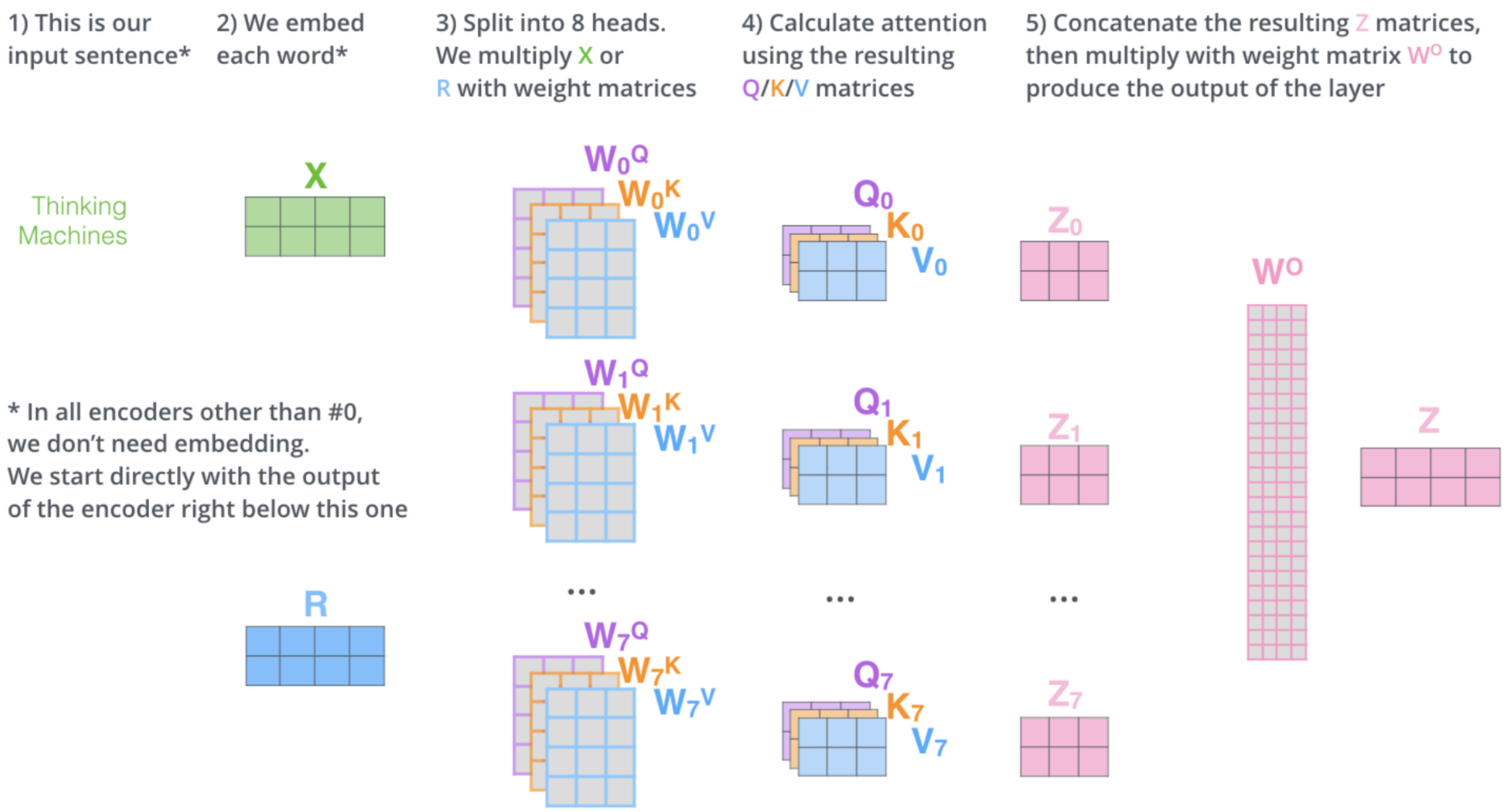
Implementing Transformers

Administrative details and recap

Administrivia

- Assignment 1 due in two weeks
- HPC cloud bursting now available (guide from TAs in Discord)
- Office hours

Recall: Multi-head Self-Attention



1) This is our input sentence*

2) We embed each word*

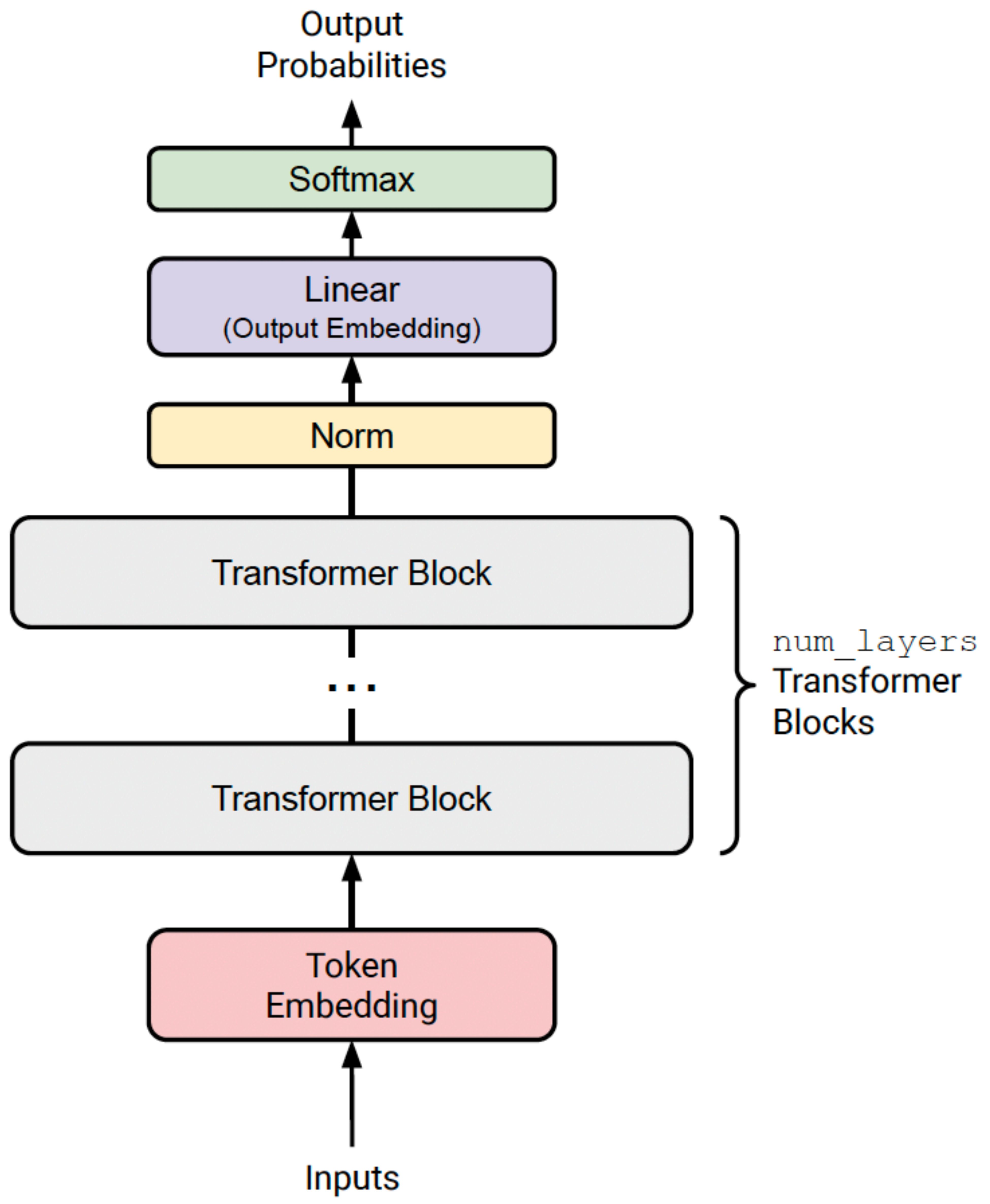
3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer

* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

Recall: Architecture



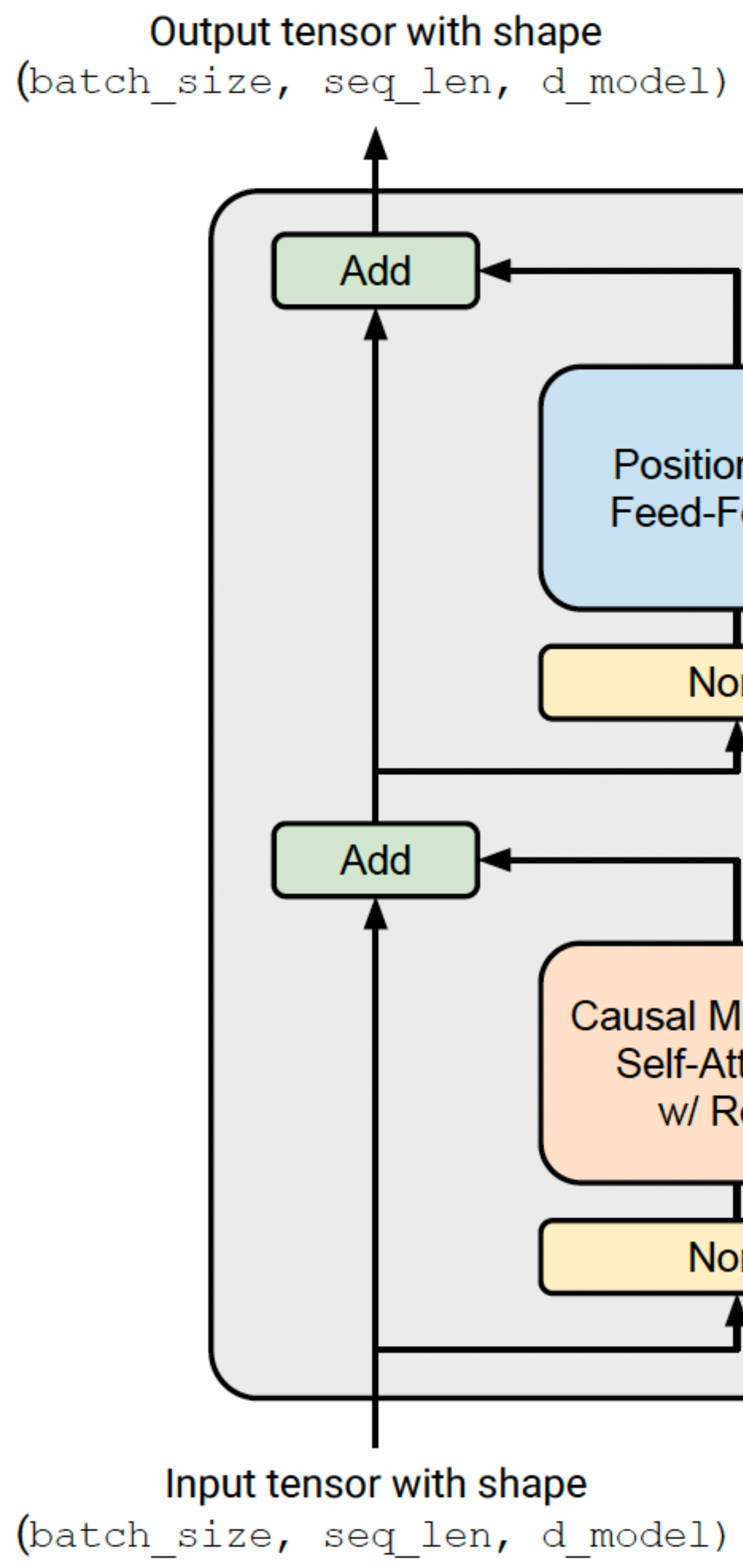
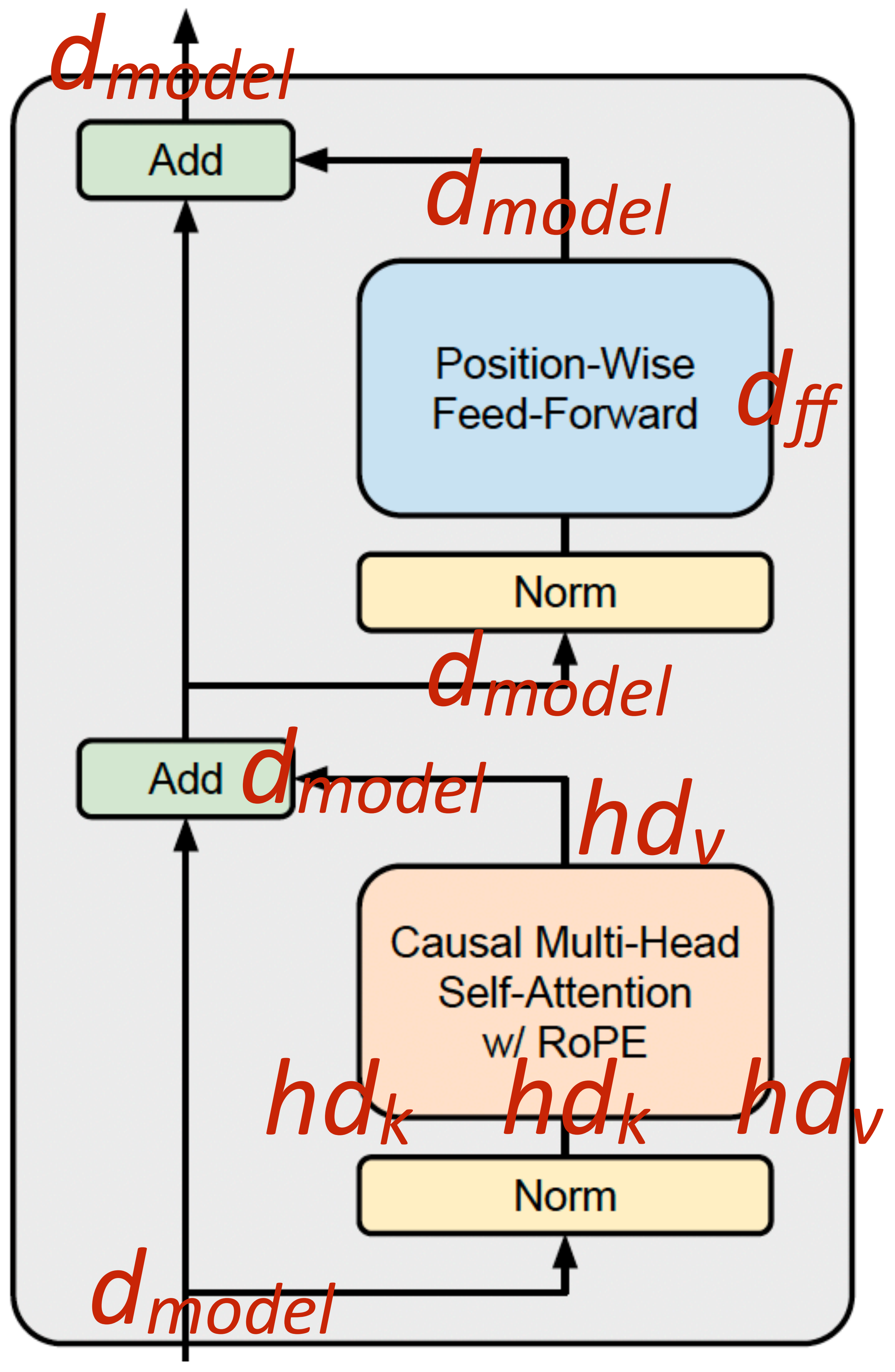

Output tensor with shape
(batch_size, seq_len, d_model)
Add
Position-Wise Feed-Forward
Norm
Add
Causal Multi-Head Self-Attention w/ RoPE
Norm
Input tensor with shape
(batch_size, seq_len, d_model)

Figure credit:
Stanford CS336 A1

Recall: Dimensions

- Main vector size d_{model}
- Queries/keys: d_k, always smaller than d_{model}, often d_{model}/h (number of heads)
- Values: separate dimension d_v, output is multiplied by W^O which is $(d_v \times h) \times d_{model}$ so we can get back to d_{model}
- FFN can use a higher latent dimension

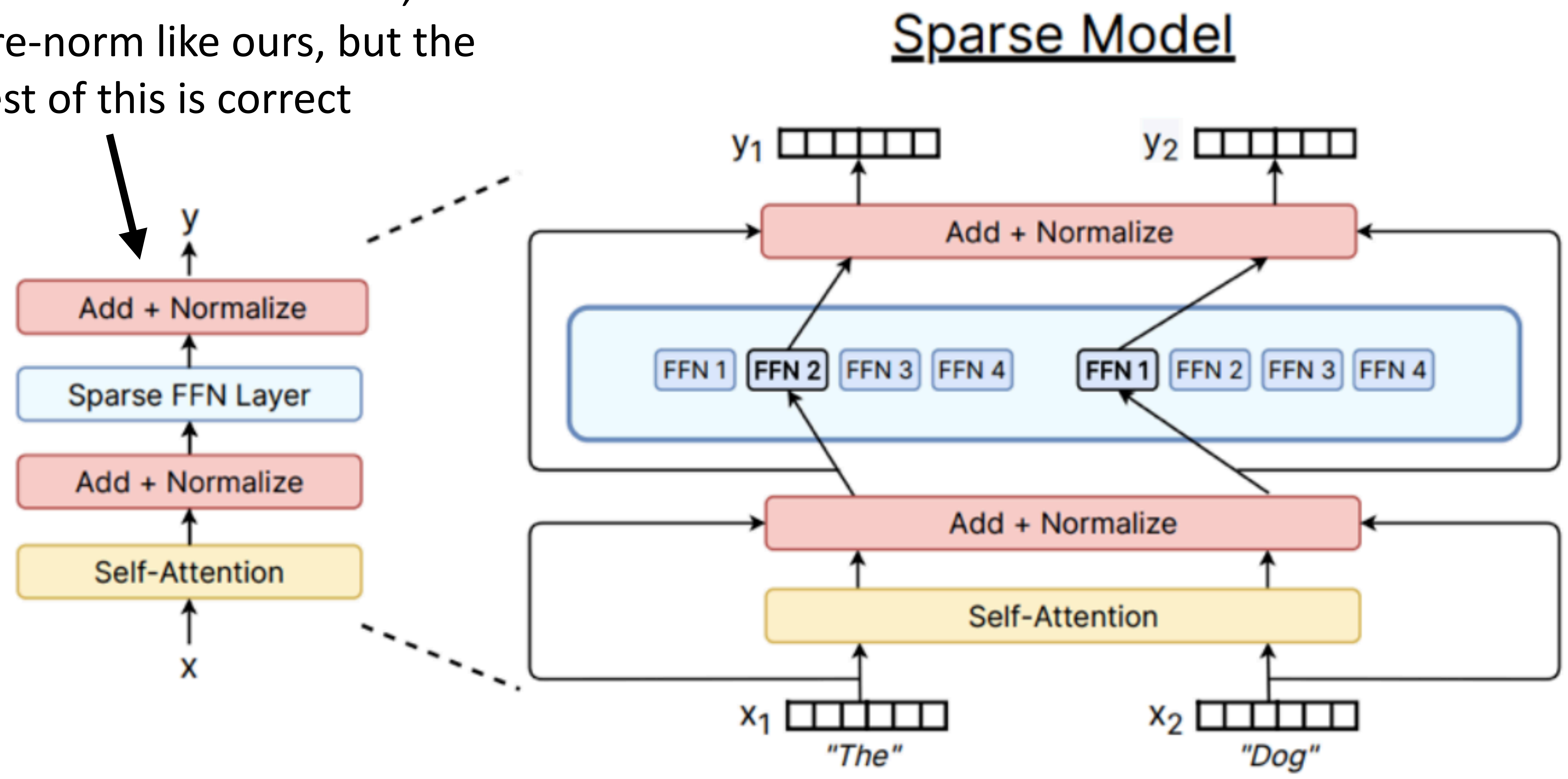


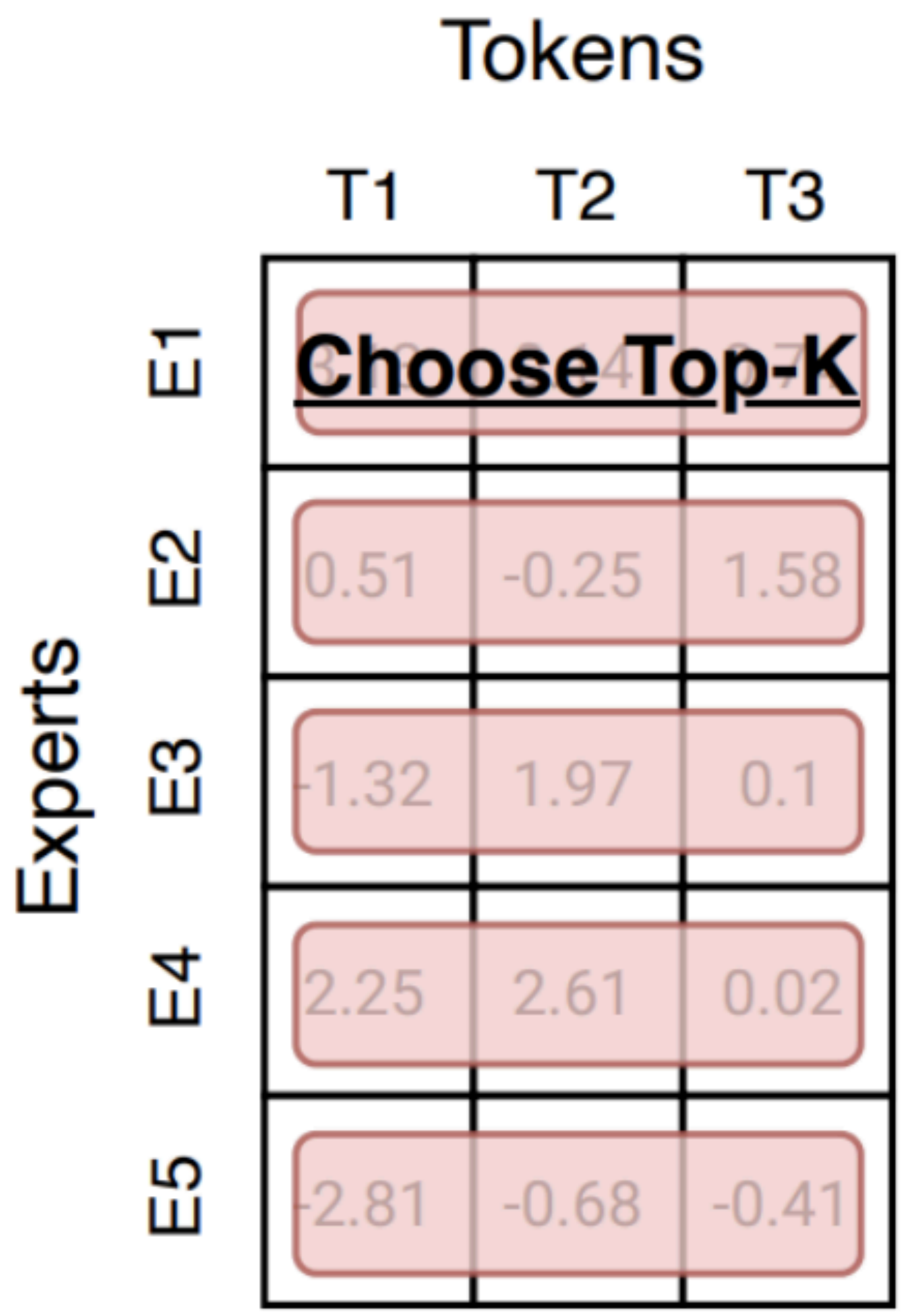
$$\mathrm{FFN}(x) = \mathrm{SwiGLU}(x, W_1, W_2, W_3) = W_2(\mathrm{SiLU}(W_1 x) \odot W_3 x)$$

Figure credit: Stanford CS336 A1

Mixture of Experts

post-norm Transformer, not pre-norm like ours, but the rest of this is correct



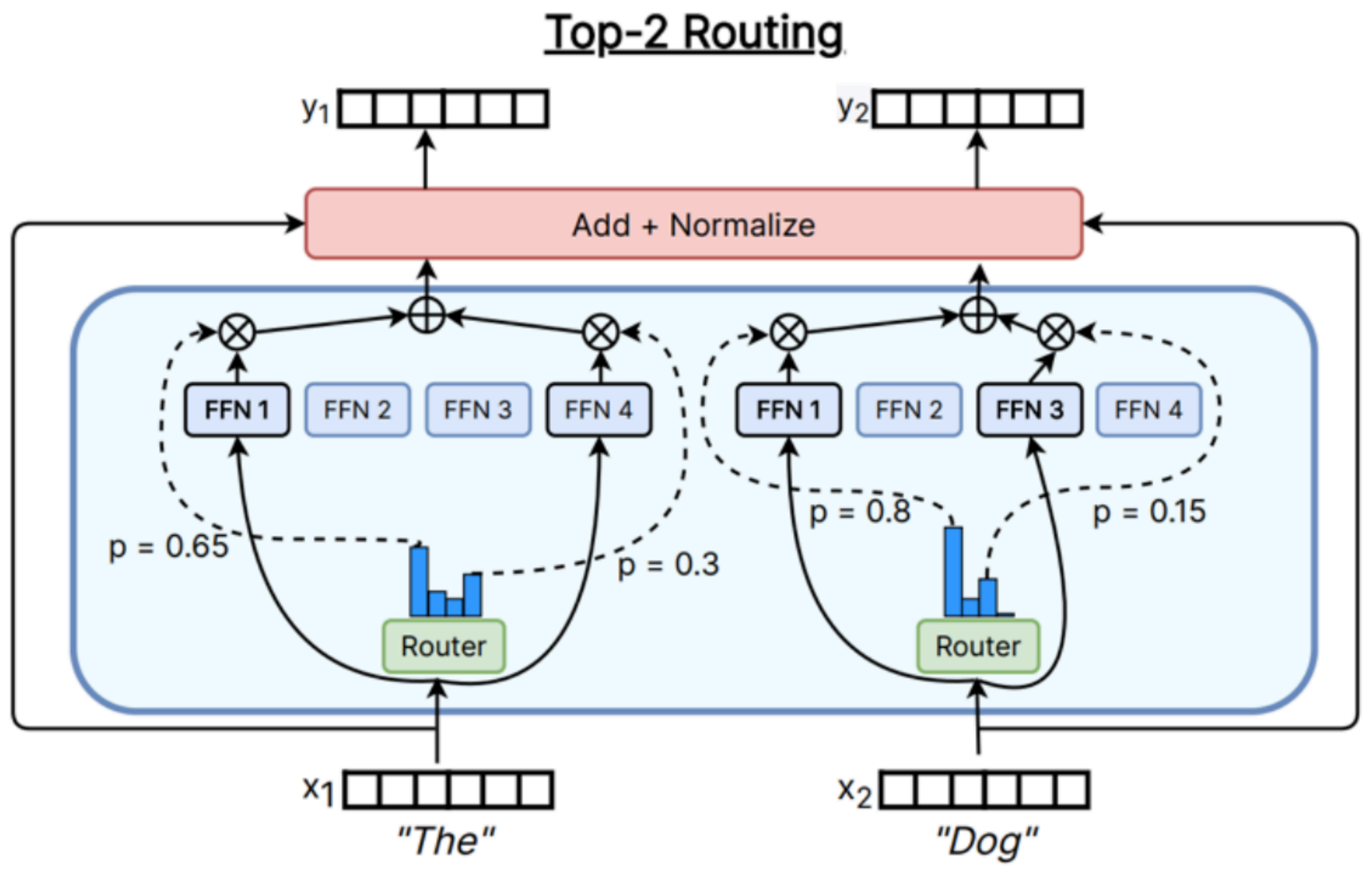

Tokens
Experts
Choose Top-K

	T1	T2	T3
E1	[illegible]	[illegible]	[illegible]
E2	0.51	-0.25	1.58
E3	1.32	1.97	0.1
E4	2.25	2.61	0.02
E5	2.81	-0.68	-0.41

- Router scores each expert (FFNN), only the top-k scoring ones are used, take weighted combination of their outputs

Slide credit: Tatsu Hashimoto/CS 336

Mixture of Experts



Used in *most* MoEs

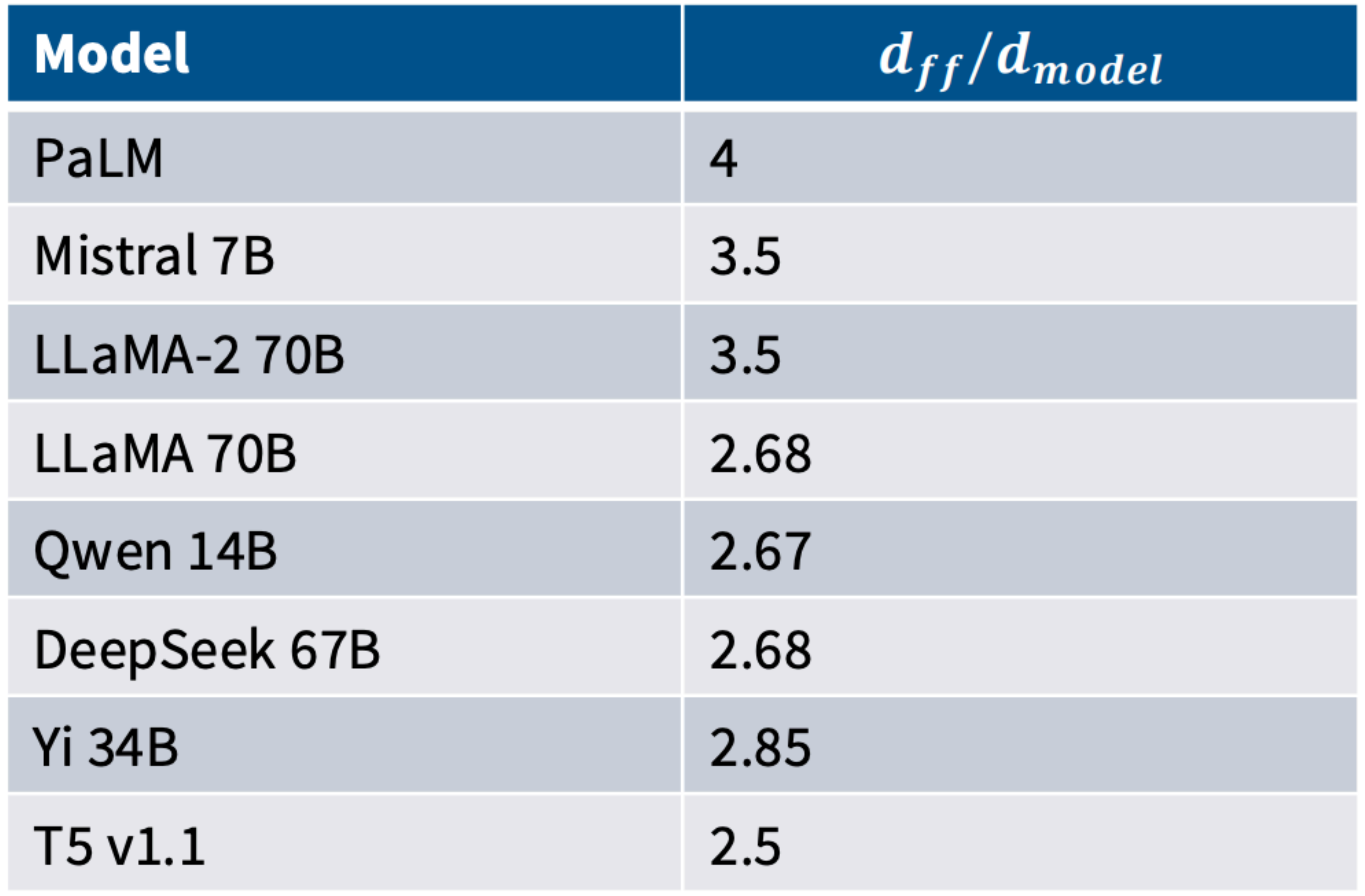
Switch Transformer (k=1)
Gshard (k=2), Grok (2), Mixtral (2),
Qwen (4), DBRX (4),
DeepSeek (7)

- Variant in DeepSeek/Qwen: have 1 shared expert that's always used

Slide credit: Tatsu Hashimoto/CS 336

Hyperparameters: d_{ff}

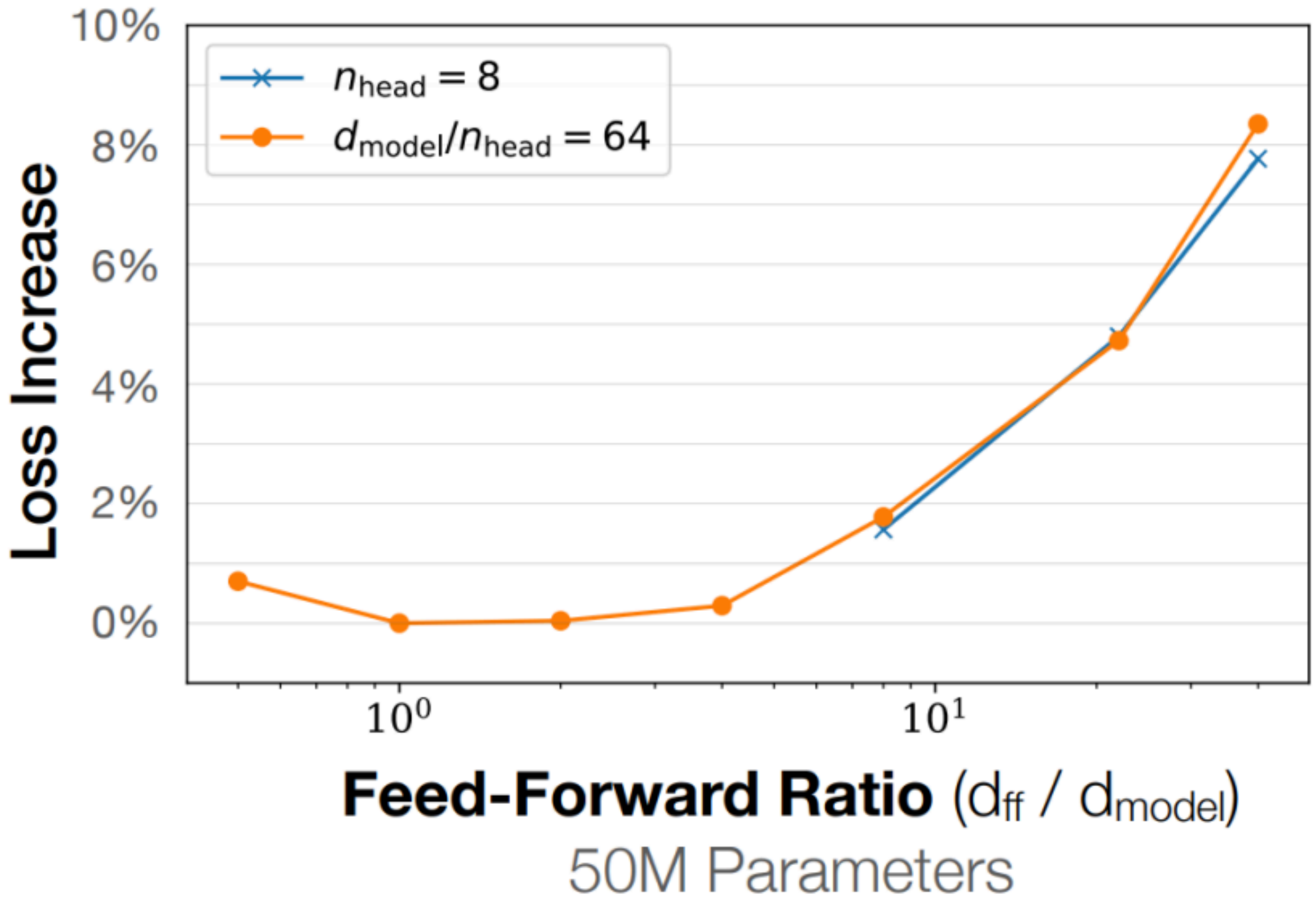
- $d_{ff} = 4\ d_{model}$ is common, except on GLU variants where $d_{ff} = 8/3\ d_{model}$



Model	d_{ff}/d_{model}
PaLM	4
Mistral 7B	3.5
LLaMA-2 70B	3.5
LLaMA 70B	2.68
Qwen 14B	2.67
DeepSeek 67B	2.68
Yi 34B	2.85
T5 v1.1	2.5

Slide credit: Tatsu Hashimoto/CS 336

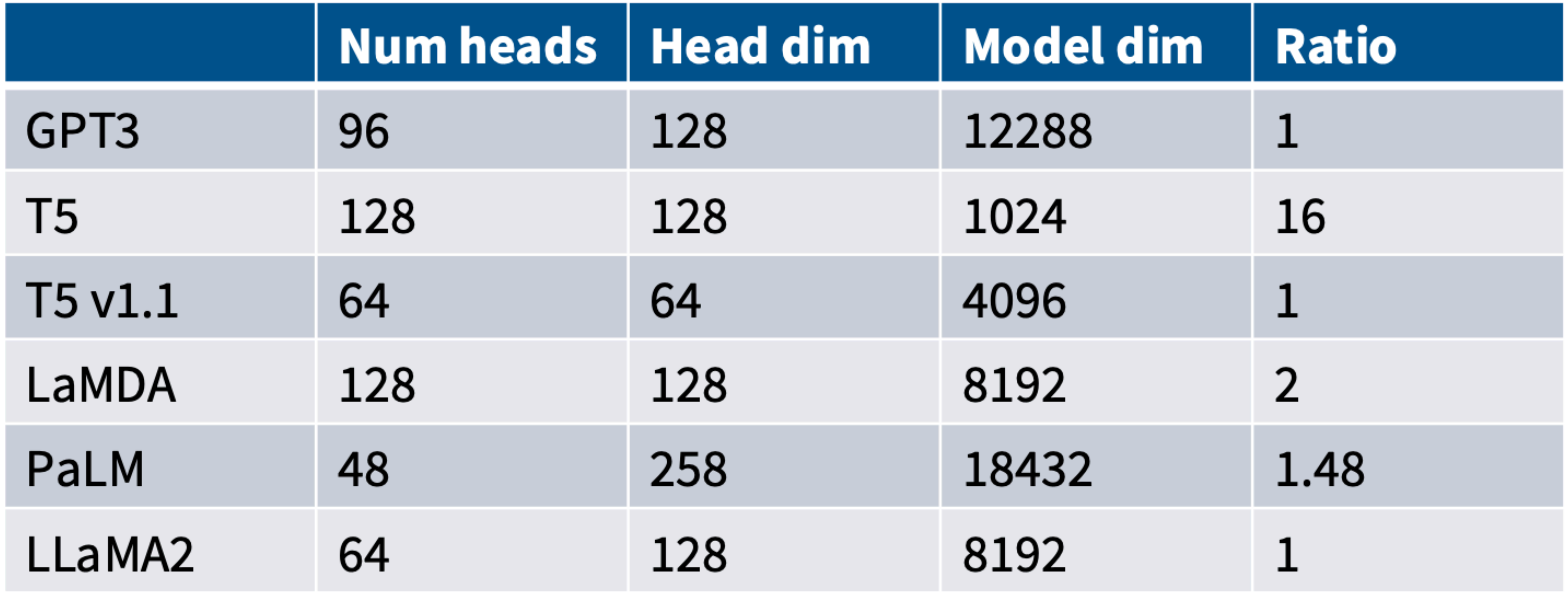
Hyperparameters



Slide credit: Tatsu Hashimoto/CS 336, original figure: Kaplan et al., 2020

Hyperparameters: Heads

- Ratio of num heads * head dim / model dim

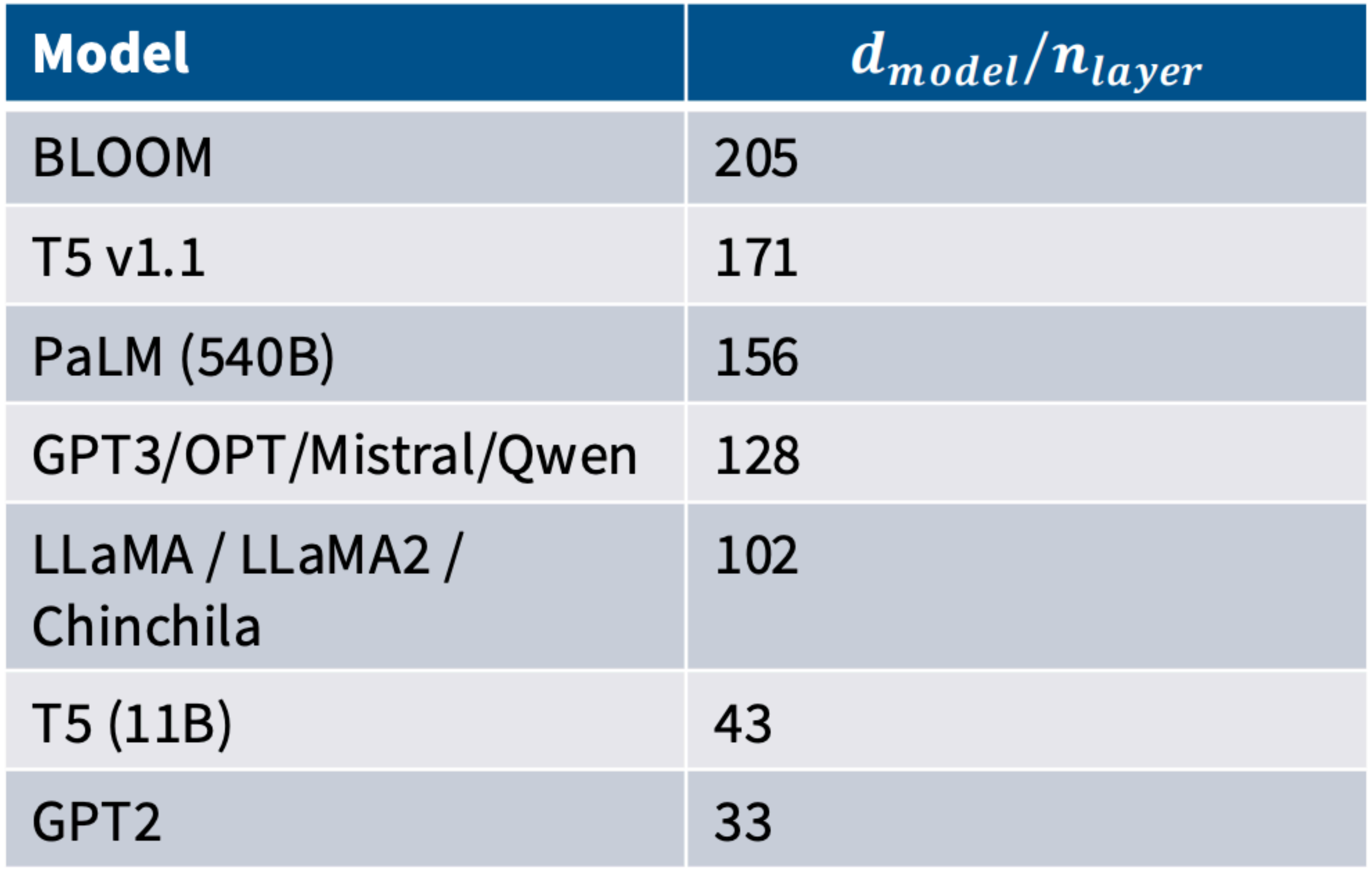


	Num heads	Head dim	Model dim	Ratio
GPT3	96	128	12288	1
T5	128	128	1024	16
T5 v1.1	64	64	4096	1
LaMDA	128	128	8192	2
PaLM	48	258	18432	1.48
LLaMA2	64	128	8192	1

Slide credit: Tatsu Hashimoto/CS 336

Hyperparameters: Layers

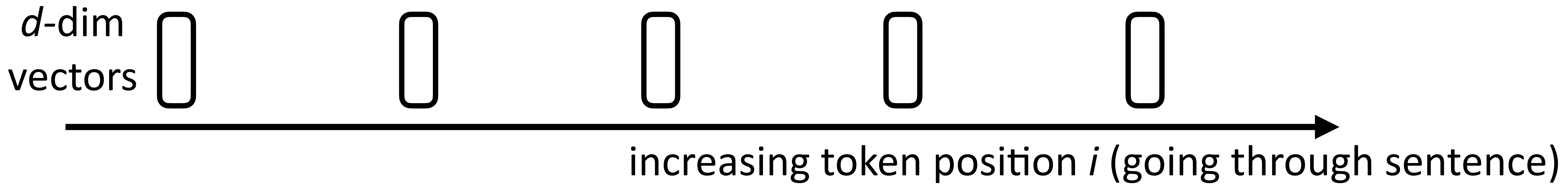
- Ratio of d_{model} and number of layers: wider or deeper?



Model	d_{model}/n_{layer}
BLOOM	205
T5 v1.1	171
PaLM (540B)	156
GPT3/OPT/Mistral/Qwen	128
LLaMA / LLaMA2 / Chinchila	102
T5 (11B)	43
GPT2	33

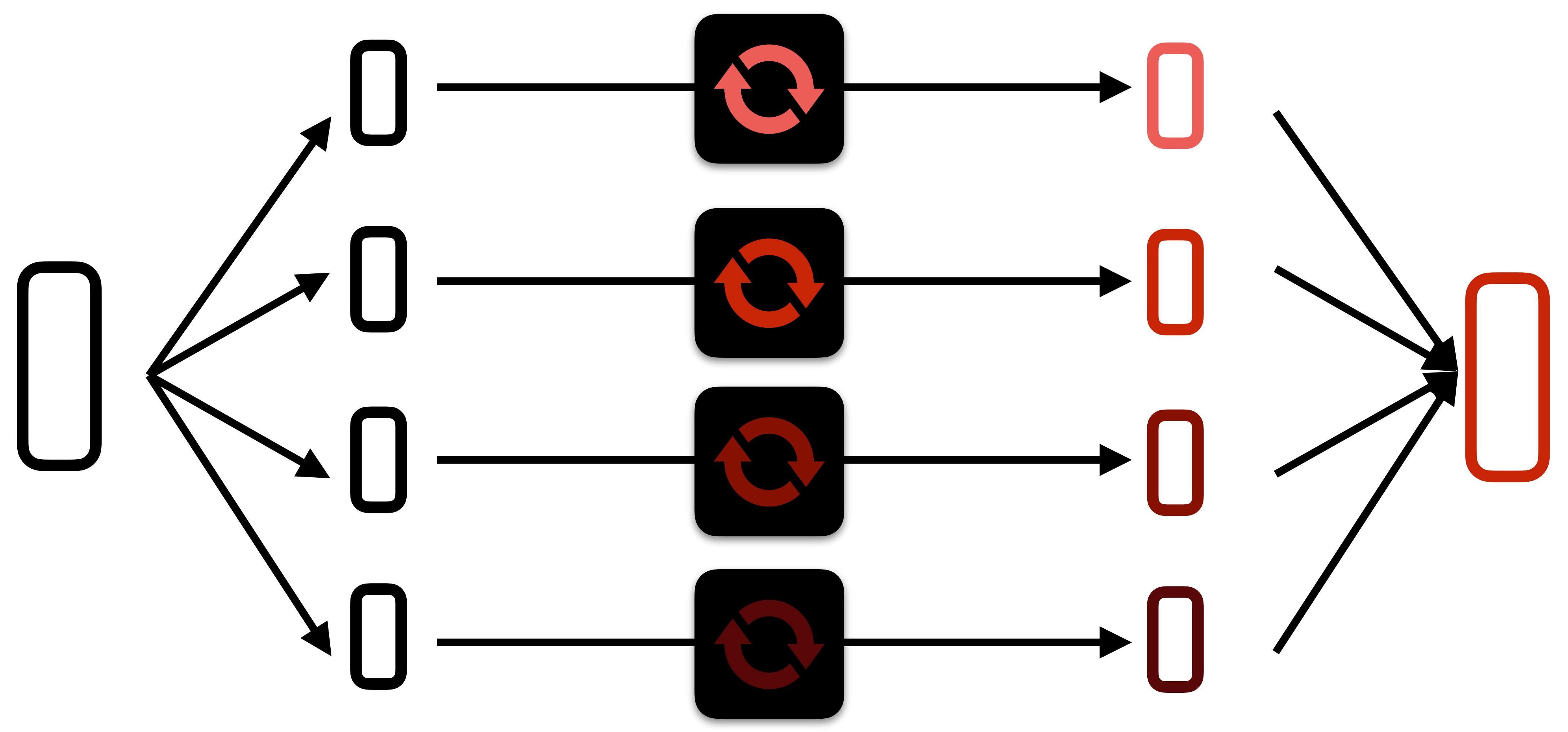
Slide credit: Tatsu Hashimoto/CS 336

Recall: RoPE (Jianlin Su et al., 2021)



Goal: encode positional information in each vector.

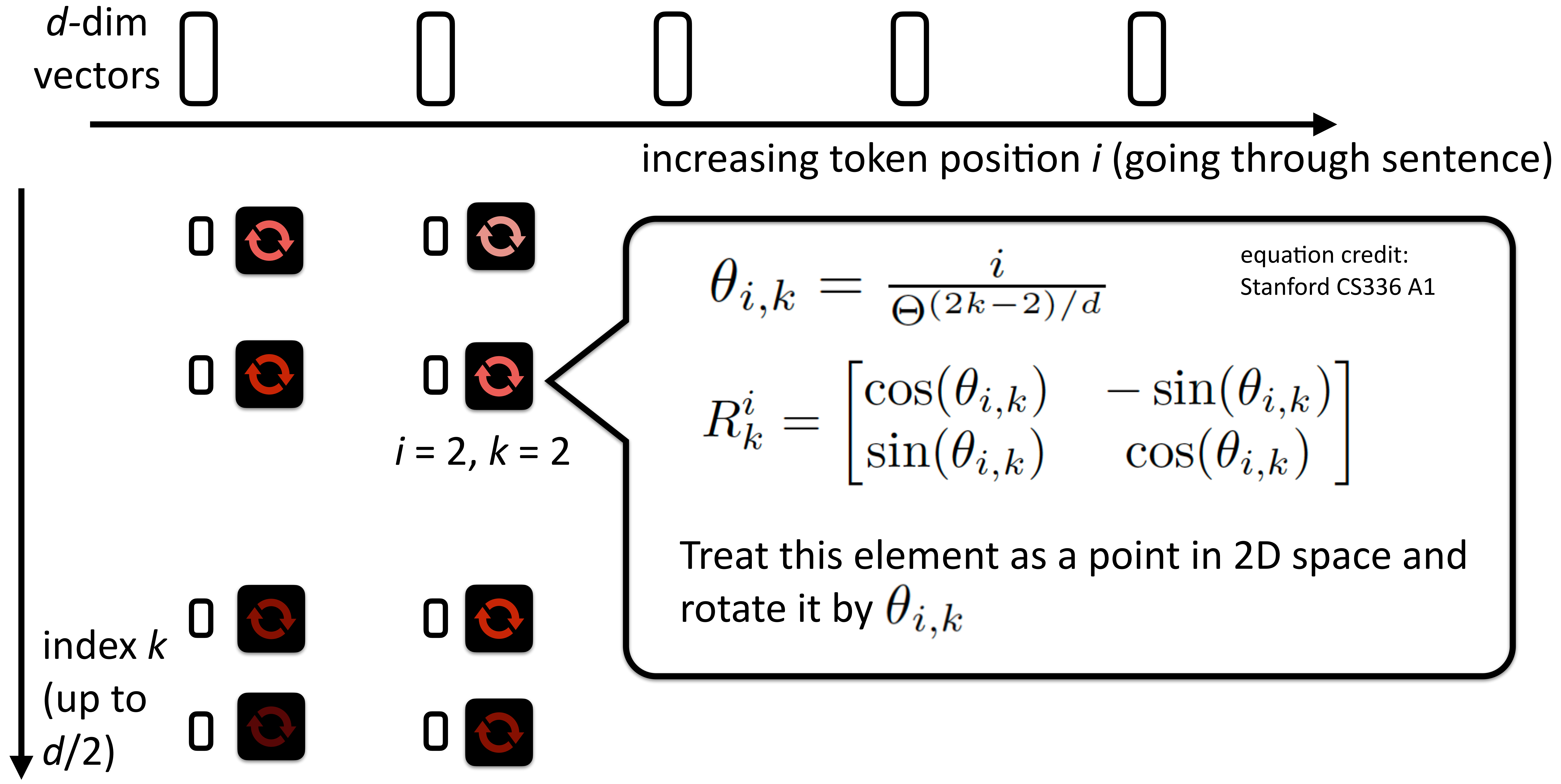
For vector at position *i*:



Step 1: Break into *d*/2 vectors in $\mathbb{R}^2$

Step 2: Rotate each one by an amount depending on *i* and the vector index

Recall: RoPE (Jianlin Su et al., 2021)



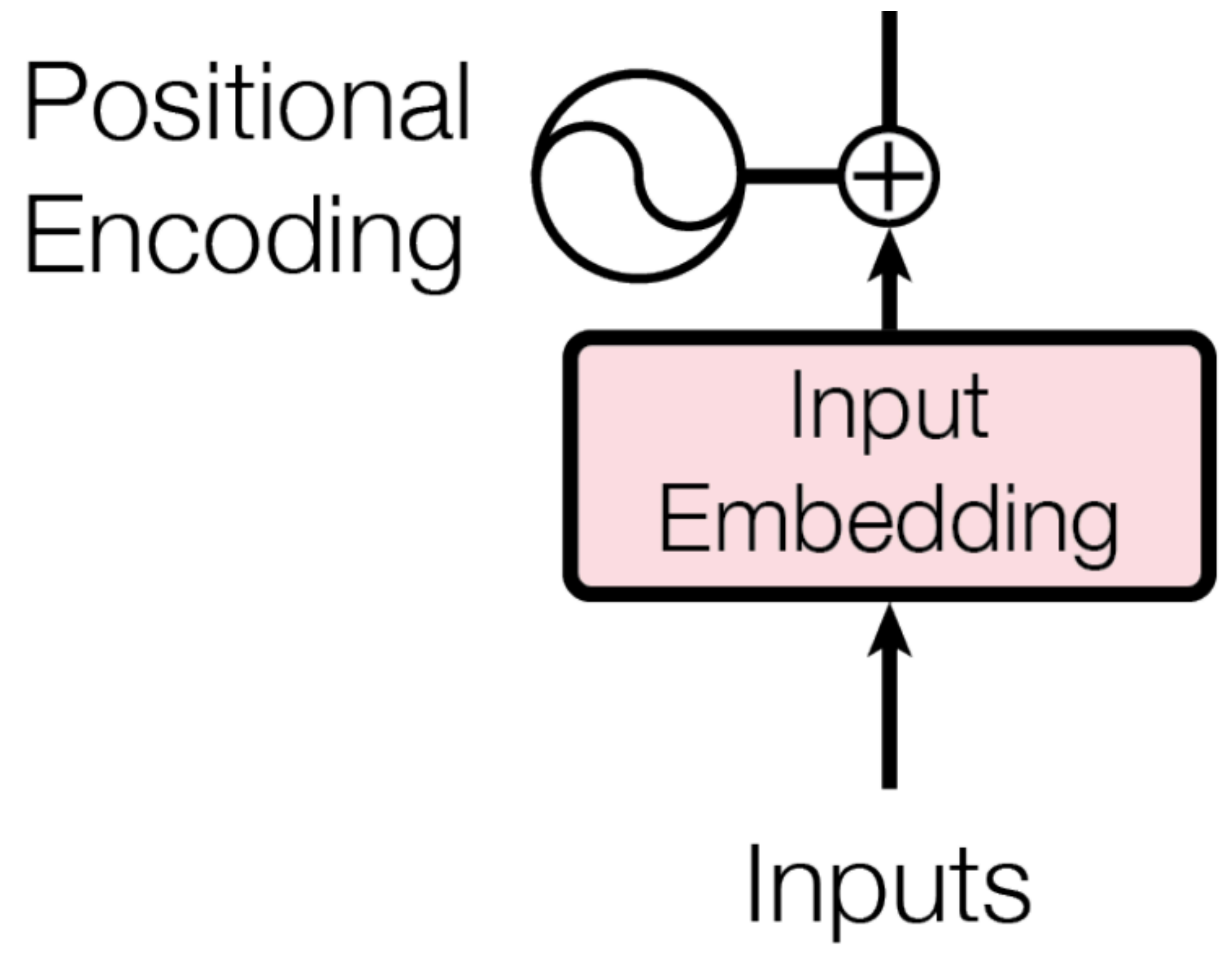
$$\theta_{i,k} = \frac{i}{\Theta^{(2k-2)/d}}$$

equation credit: Stanford CS336 A1

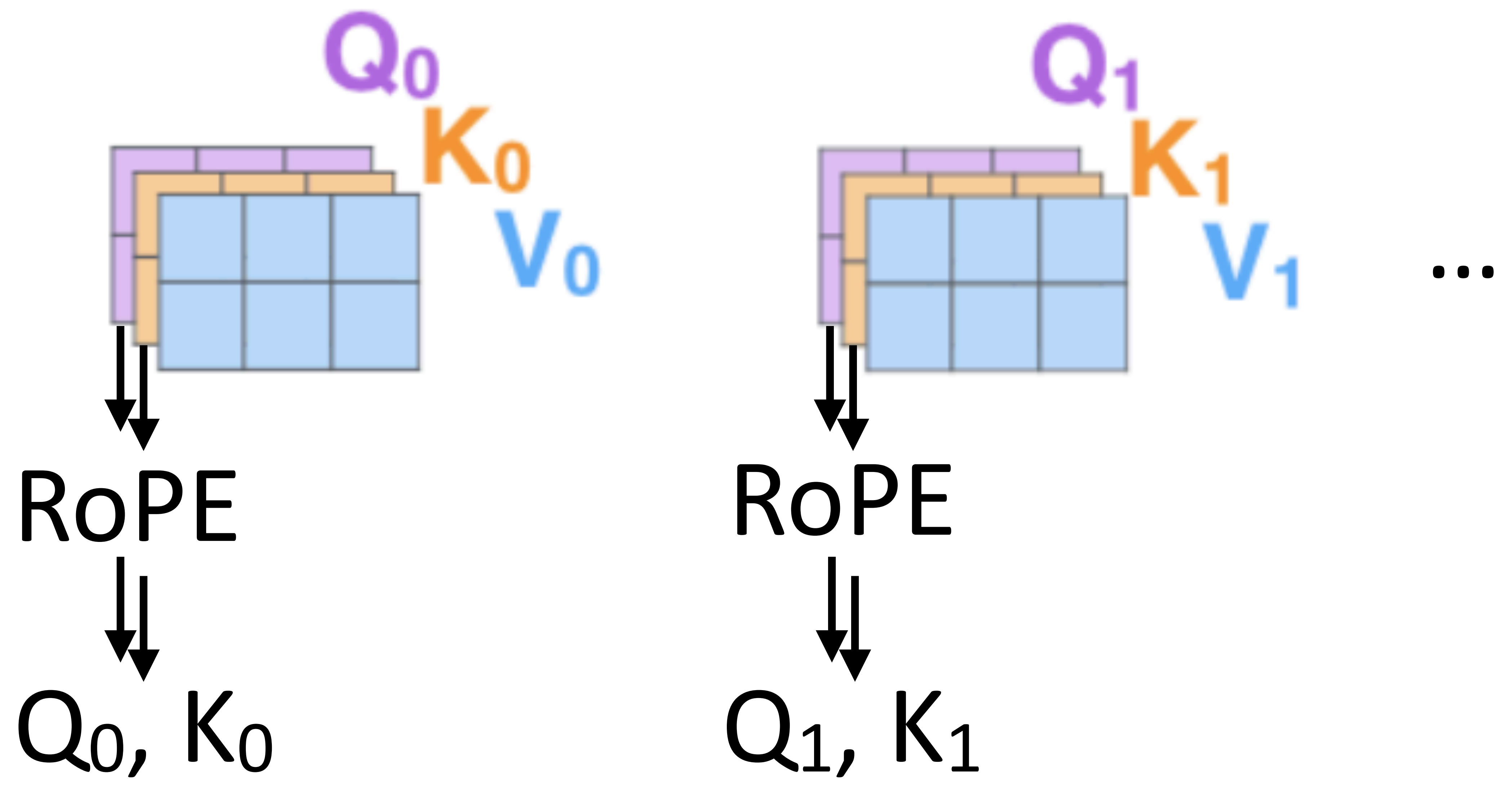
$$R_k^i = \begin{bmatrix} \cos(\theta_{i,k}) & -\sin(\theta_{i,k}) \\ \sin(\theta_{i,k}) & \cos(\theta_{i,k}) \end{bmatrix}$$

Treat this element as a point in 2D space and rotate it by $\theta_{i,k}$

Recall: Where are PEs used?



Classical Vaswani et al. Transformer (2017): added to input



Modern practice: Apply RoPE to Qs and Ks right before self-attention

LM Evaluation

- Accuracy doesn't make sense — predicting the next word is generally impossible so accuracy values would be very low
- Evaluate LMs on the likelihood of held-out data (averaged to normalize for length)

$$\frac{1}{n}\sum_{i=1}^{n} \log P(w_i | w_1, \ldots, w_{i-1})$$

- Perplexity: exp(average negative log likelihood). Lower is better
  - Suppose we have probs 1/4, 1/3, 1/4, 1/3 for 4 predictions
  - Avg NLL (base e) = 1.242 Perplexity = 3.464 <== geometric mean of denominators

Tokenizers

Bag-of-words Features

this movie was great! would watch again Positive

How do classical sentiment analysis models handle this?

[contains *the*]	[contains *a*]	[contains *was*]	[contains *movie*]	[contains *film*]	...
position 0	position 1	position 2	position 3	position 4	
f(***x***) = [0	0	1	1	0	...

- Very large vector space (size of vocabulary), sparse features (how many per example?)

Feature Representation

What are some preprocessing operations we might want to do before we map to words?

Feature Extraction Details

Tokenization:

"I thought it wasn't that great!" critics complained.

" I thought it was n't that great ! " critics complained .

- Split out punctuation, contractions; handle hyphenated compounds
- Lowercasing (maybe)
- Filtering stopwords (maybe)
- Building the feature vector requires *indexing* the words (mapping them to axes). Store an invertible map from string -> index
- Can we use a similar process as bag-of-words to build Transformer LMs?

Tokenization for Transformers

Input: raw string

Output: sequence of token IDs that will be embedded through the embedding matrix

What are some options? Word-level? Character-level?

Word Tokenization for Transformers

Where do we get a vocabulary from?

Will we encounter new tokens at test time that we can’t map to our vocab?

How well will this work in our Transformer?

Character (Byte) Tokenization

Where do we get a vocabulary from?

Will we encounter new tokens at test time that we can’t map to our vocab?

How well will this work in our Transformer?

Tokenization Desiderata

Input: raw string

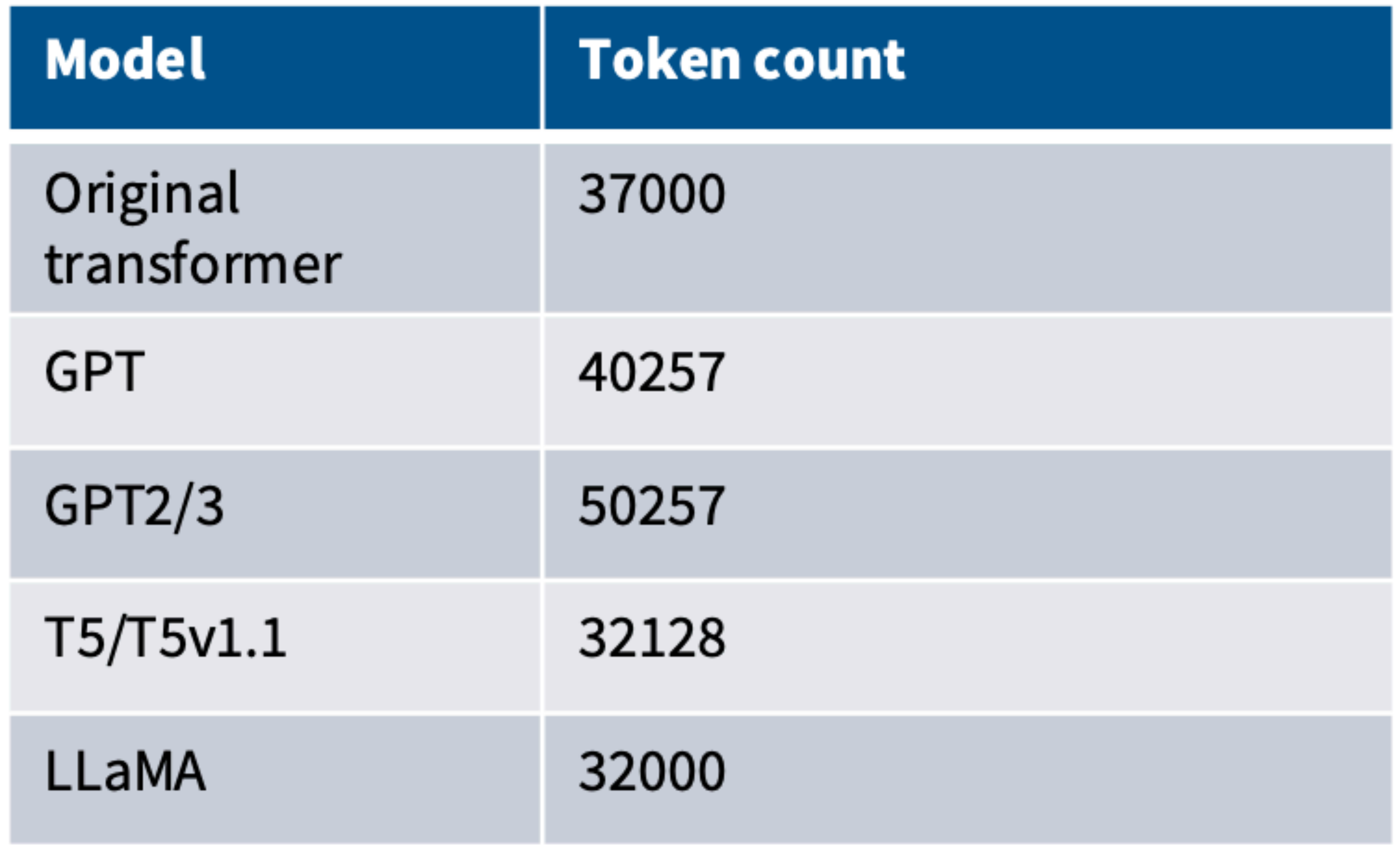
Output: sequence of token IDs that will be embedded through the embedding matrix

Desiderata

- Moderate vocabulary size: 10k ~ 500K depending on scale of LLM
- Ability to represent every string in the language

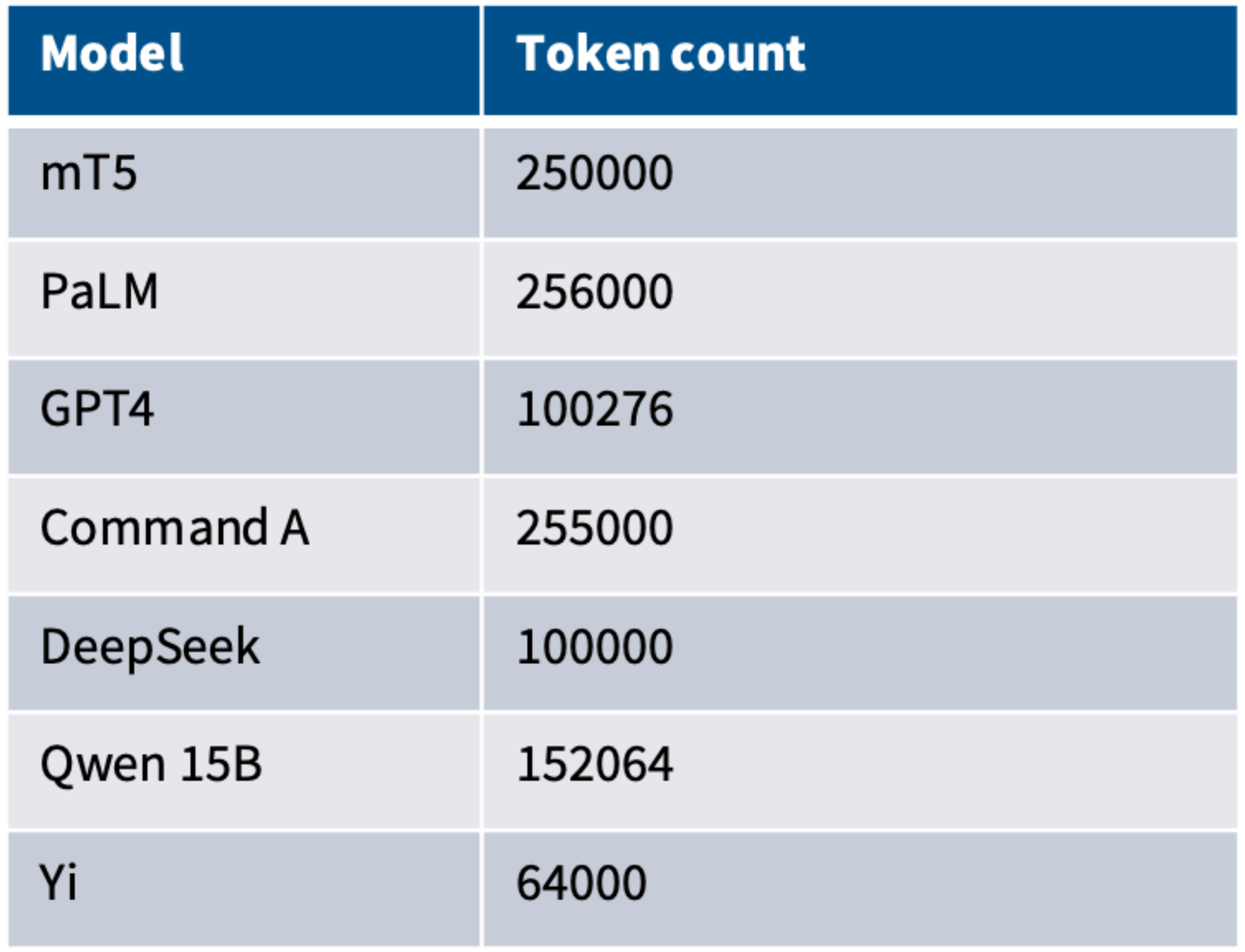
Vocabulary Sizes

Monolingual models – 30-50k vocab



Model	Token count
Original transformer	37000
GPT	40257
GPT2/3	50257
T5/T5v1.1	32128
LLaMA	32000

Multilingual / production systems 100-250k



Model	Token count
mT5	250000
PaLM	256000
GPT4	100276
Command A	255000
DeepSeek	100000
Qwen 15B	152064
Yi	64000

See GPT2 in practice: https://tiktokenizer.vercel.app/?encoder=gpt2

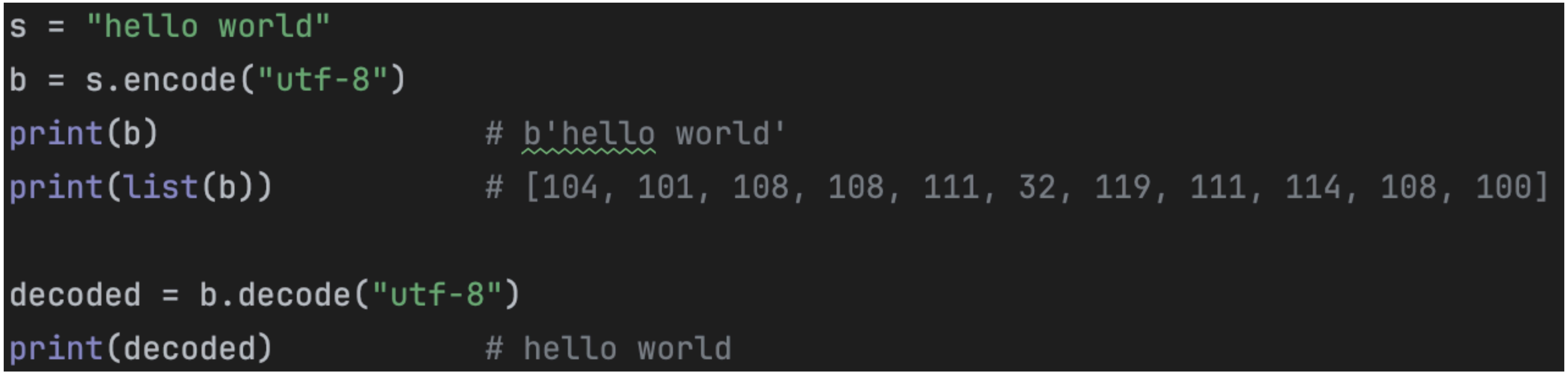
Slide credit: Tatsu Hashimoto/CS 336

Byte Pair Encoding

Bytes and Character Encodings

`bytes:` Python datatype, think of it as list[byte]

Strings are mappable to/from bytes via “encoding”: UTF-8



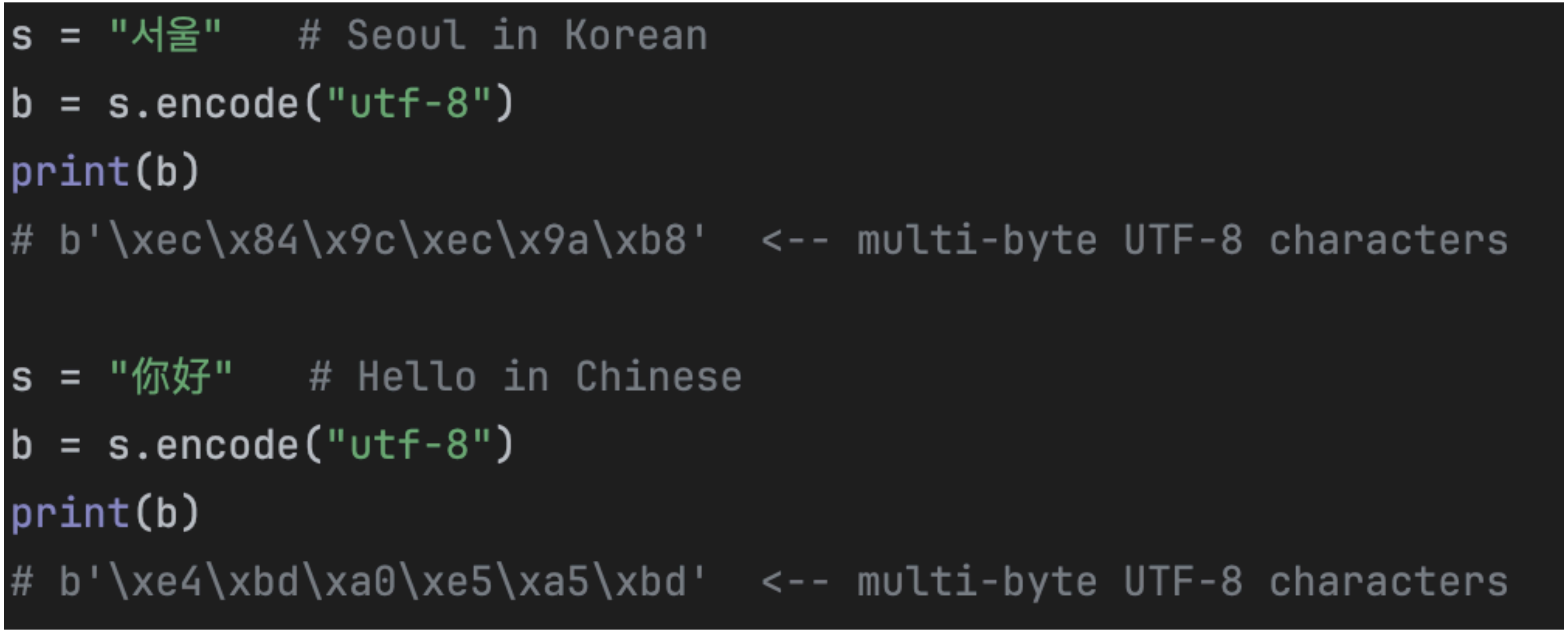
```
s = "hello world"
b = s.encode("utf-8")
print(b)                 # b'hello world'
print(list(b))           # [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]

decoded = b.decode("utf-8")
print(decoded)           # hello world
```

Example: ChatGPT

UTF-8: 256 tokens

How does UTF-8 handle the fact that there are more than 256 tokens?



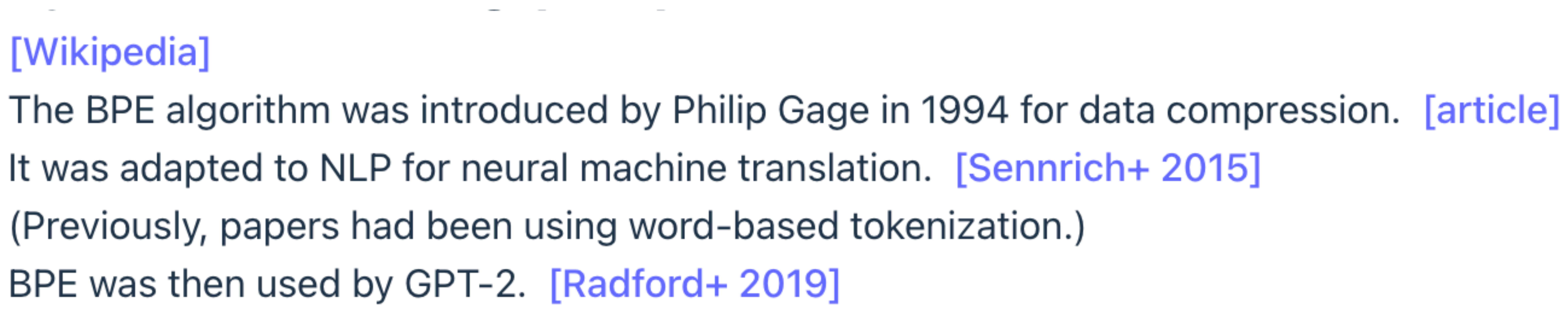
```
s = "서울"   # Seoul in Korean
b = s.encode("utf-8")
print(b)
# b'\xec\x84\x9c\xec\x9a\xb8'  <-- multi-byte UTF-8 characters

s = "你好"   # Hello in Chinese
b = s.encode("utf-8")
print(b)
# b'\xe4\xbd\xa0\xe5\xa5\xbd'  <-- multi-byte UTF-8 characters
```

Example: ChatGPT

Tokenization Desiderata

BPE: merge bytes into subword tokens, these become the vocabulary.
Commonly cooccurring bytes are the first to be merged



[Wikipedia]

The BPE algorithm was introduced by Philip Gage in 1994 for data compression. [article]

It was adapted to NLP for neural machine translation. [Sennrich+ 2015]

(Previously, papers had been using word-based tokenization.)

BPE was then used by GPT-2. [Radford+ 2019]

Source: Percy Liang / CS336

BPE

`bytes:` Python datatype, think of it as list[byte]

A BPE tokenizer is defined by:

- A set of merges of bytes: list[tuple[bytes, bytes]]
- A vocabulary dict[int,bytes]

Vocab = initial characters (256 bytes) + special characters + every token created by a merge

BPE tokenizer is “trained” on a corpus (but not with gradient descent!)

Step 0: Chunking

TinyStories corpus:

Once upon a time, there was a reliable otter named Ollie. He lived in a river with his family. They all loved to play and swim together.

One day, Ollie's mom said, "Ollie, hurry and get some fish for dinner!" Ollie swam fast to catch fish. He saw his friend, the duck. "Hi, Ollie!" said the duck. "Hi, duck!" said Ollie. "I need to hurry and catch fish for my family."

While Ollie was catching fish, he found a big shiny stone. He thought, "This is not a fish, but it is so pretty!" Ollie took the shiny stone home to show his family. They all looked at the shiny stone and smiled. The shiny stone made everyone happy, and they forgot about the fish for dinner.

<|endoftext|>

One day, a little boy named Tim went to the park. He saw a big tiger. The tiger was not mean, but very easy to play with. Tim and the tiger played all day. They had lots of fun.

Then, something unexpected happened. The tiger started to shake. Tim was scared. He did not know what was going on. But then, the tiger turned into a nice dog. Tim was very surprised.

Tim and the dog played together now. They were very happy. The dog was easy to play with too. At the end of the day, Tim went home with his new friend.

<|endoftext|>

$\longrightarrow$ Break into *n* chunks to process in parallel

Each chunk will still contain many of these stories, but <|endoftext|> should be stripped out — special characters are never tokenized

Step 1: Pretokenization

We don’t produce tokens across space boundaries. “ing to” would never be a token. So we break things into words to start.

```
r"""'(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+"""
```

?\p{L}+ finds one or more letters with leading space, ?\p{N}+ finds numbers, etc.

What does this do?

Step 1: Pretokenization

```
>>> # requires `regex` package
>>> import regex as re
>>> re.findall(PAT, "some text that i'll pre-tokenize")
['some', ' text', ' that', ' i', "'ll", ' pre', '-', 'tokenize']
```

Important: preserves leading space!

Step 1: Pretokenization

One chunk:

Once upon a time, there was a reliable otter named Ollie. He lived in a river with his family. They all loved to play and swim together.

One day, Ollie's mom said, "Ollie, hurry and get some fish for dinner!" Ollie swam fast to catch fish. He saw his friend, the duck. "Hi, Ollie!" said the duck. "Hi, duck!" said Ollie. "I need to hurry and catch fish for my family."

While Ollie was catching fish, he found a big shiny stone. He thought, "This is not a fish, but it is so pretty!" Ollie took the shiny stone home to show his family. They all looked at the shiny stone and smiled. The shiny stone made everyone happy, and they forgot about the fish for dinner.

<|endoftext|>

One day, a little boy named Tim went to the park. He saw a big tiger. The tiger was not mean, but very easy to play with. Tim and the tiger played all day. They had lots of fun.

Then, something unexpected happened. The tiger started to shake. Tim was scared. He did not know what was going on. But then, the tiger turned into a nice dog. Tim was very surprised.

Tim and the dog played together now. They were very happy. The dog was easy to play with too. At the end of the day, Tim went home with his new friend.

<|endoftext|>

$\longrightarrow$ Split on <|endoftext|>

$\downarrow$

Run pretokenization

$\downarrow$

Collapse into dict

{“ tiger”: 427, “ dog”: 416, …}

Why?

Step 2: Compute Merges

Corpus:

```
low low low low low
lower lower widest widest widest
newest newest newest newest newest newest
```

Count dict: `{low: 5, lower: 2, widest: 3, newest: 6}`

Count dict (after byte-tokenizing words): $\{$(l,o,w): 5 …$\}$

Pairwise counts:

$\{$lo: 7, ow: 7, we: 8, er: 2, wi: 3, id: 3, de: 3, es: 9, st: 9, ne: 6, ew: $6\}$

Pick most frequent: “es”, “st” tied -> prefer “st” from lex ordering

Apply the merge, **efficiently update your count dict** (need to be smart!)

Repeat until enough merges are made

Step 3: Encode and Decode

Final tokenizer:

- A set of merges of bytes: list[tuple[bytes, bytes]]
- A vocabulary dict[int,bytes]

Decoding: sequence of BPE ids -> string

How should we do this?

Step 3: Encode and Decode

Final tokenizer:

- A set of merges of bytes: list[tuple[bytes, bytes]]
- A vocabulary dict[int,bytes]

Encoding: string -> sequence of BPE ids

How should we do this?

Do not apply the "standard" greedy heuristic of matching the longest token that applies! This will not return the right sequence in general. (This heuristic comes from WordPiece / BERT, but is not used here.)

Tips

Profile your code! Identify bottlenecks and fix them; think about what tokens / data structures / etc. need to be touched after every merge

Use multiprocessing to parallelize the initial encoding (saves ~minutes)

You're allowed to ask LLMs for help on optimizing things…but pay attention to the course policies

Percy Liang shows some code here (slightly different than the project code):

https://stanford-cs336.github.io/spring2025-lectures/?trace=var/traces/lecture_01.json

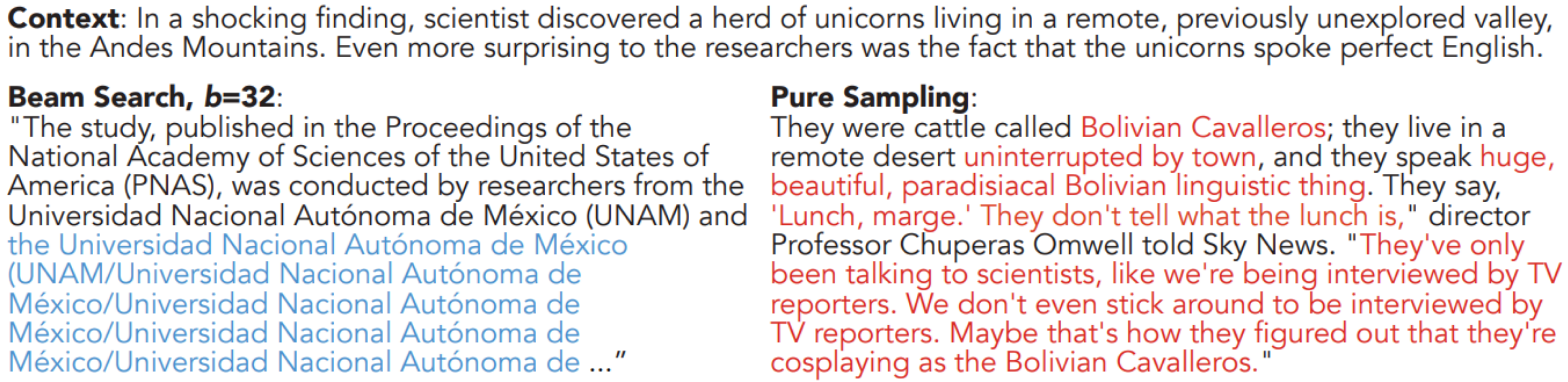
Decoding

Decoding Strategies

- LMs place a distribution $P(y_i \mid y_1, \ldots, y_{i-1})$
- seq2seq models place a distribution $P(y_i \mid \mathbf{x}, y_1, \ldots, y_{i-1})$
- Generation from both models looks similar; how do we do it?
  - Option 1: $\max_{y_i} P(y_i \mid y_1, \ldots, y_{i-1})$ — take greedily best option
  - Option 2: use beam search to find the sequence with the highest prob.
  - Option 3: sample from the model; draw y_i from that distribution
- When should we use these different approaches?

Decoding Strategies

- Story generation with GPT-2:



Context: In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

Beam Search, *b*=32:
"The study, published in the Proceedings of the National Academy of Sciences of the United States of America (PNAS), was conducted by researchers from the Universidad Nacional Autónoma de México (UNAM) and the Universidad Nacional Autónoma de México (UNAM/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de ..."

Pure Sampling:
They were cattle called Bolivian Cavalleros; they live in a remote desert uninterrupted by town, and they speak huge, beautiful, paradisiacal Bolivian linguistic thing. They say, 'Lunch, marge.' They don't tell what the lunch is," director Professor Chuperas Omwell told Sky News. "They've only been talking to scientists, like we're being interviewed by TV reporters. We don't even stick around to be interviewed by TV reporters. Maybe that's how they figured out that they're cosplaying as the Bolivian Cavalleros."

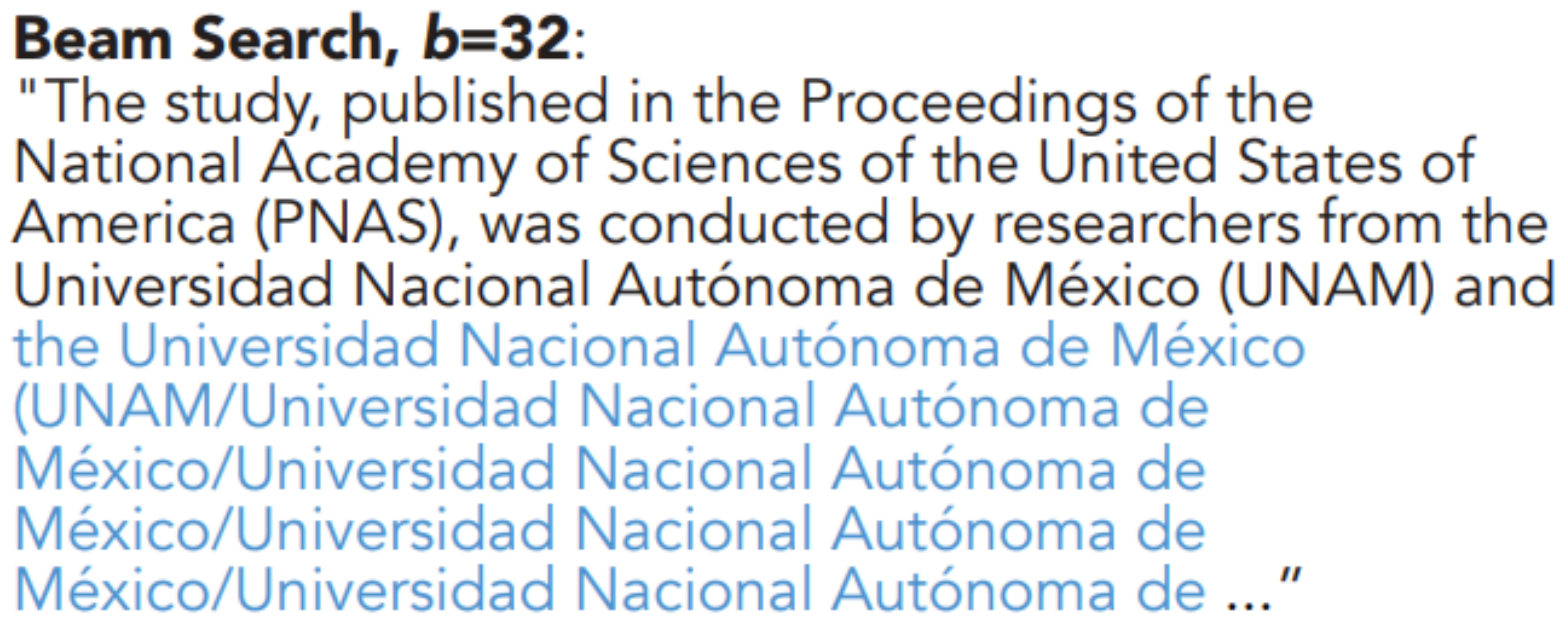
- Beam search degenerates and starts repeating. If you see a fragment repeated 2-3x, it has very high probability to keep repeating

- Sampling is too noisy — introduces many grammatical errors

Holtzman et al. (2019)

Degeneration

- Beam search fails because the model is *locally normalized*
- Let's look at all the individual decisions that get made here



Beam Search, *b*=32:
"The study, published in the Proceedings of the National Academy of Sciences of the United States of America (PNAS), was conducted by researchers from the Universidad Nacional Autónoma de México (UNAM) and the Universidad Nacional Autónoma de México (UNAM/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de México/Universidad Nacional Autónoma de ..."

P(Nacional | … Universidad) is high

P(Autónoma | … Universidad Nacional) is high

P(de | … Universidad Nacional Autónoma) is high

P(México | Universidad Nacional Autónoma de) is high

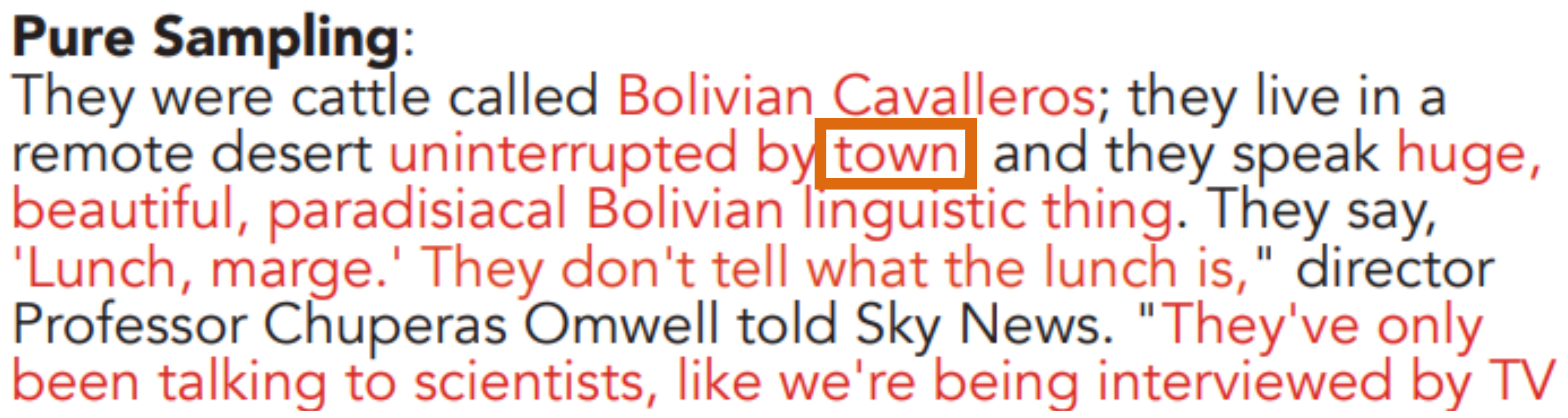
P(/ | … México) and P(Universidad | … México /) — these probabilities may be low. But those are just 2/6 words of the repeating fragment

- **Each word is likely given the previous words but the sequence is bad**

Holtzman et al. (2019)

Drawbacks of Sampling

- Sampling is “too random”



Pure Sampling:
They were cattle called Bolivian Cavalleros; they live in a remote desert uninterrupted by town and they speak huge, beautiful, paradisiacal Bolivian linguistic thing. They say, 'Lunch, marge.' They don't tell what the lunch is," director Professor Chuperas Omwell told Sky News. "They've only been talking to scientists, like we're being interviewed by TV

P(*y* | … they live in a remote desert uninterrupted by)

0.01	roads
0.01	towns
0.01	people
0.005	civilization
…	
0.0005	town

Good options, maybe accounting for 90% of the total probability mass. So a 90% chance of getting something good

Long tail with 10% of the mass

Holtzman et al. (2019)

Nucleus Sampling

P(y | … they live in a remote desert uninterrupted by)

0.01	roads
0.01	towns
0.01	people
0.005	civilization

$\longrightarrow$ renormalize and sample

cut off after p% of mass

- Define a threshold p. Keep the most probable options account for p% of the probability mass (the *nucleus*), then sample among these.
- To implement: sort options by probability, truncate the list once the total exceeds p, then renormalize and sample from it

Holtzman et al. (2019)

Decoding Strategies

- LMs place a distribution $P(y_i \mid y_1, \ldots, y_{i-1})$
- seq2seq models place a distribution $P(y_i \mid \mathbf{x}, y_1, \ldots, y_{i-1})$
- How to generate sequences?
  - Option 1: $\max_{y_i} P(y_i \mid y_1, \ldots, y_{i-1})$ — take greedily best option
  - Option 2: use beam search to find the sequence with the highest prob.
  - ~~Option 3: sample from the model; draw y_i from that distribution~~
  - Option 4: nucleus sampling

Holtzman et al. (2019)

Administrative details and recap

Tokenizers

Byte Pair Encoding

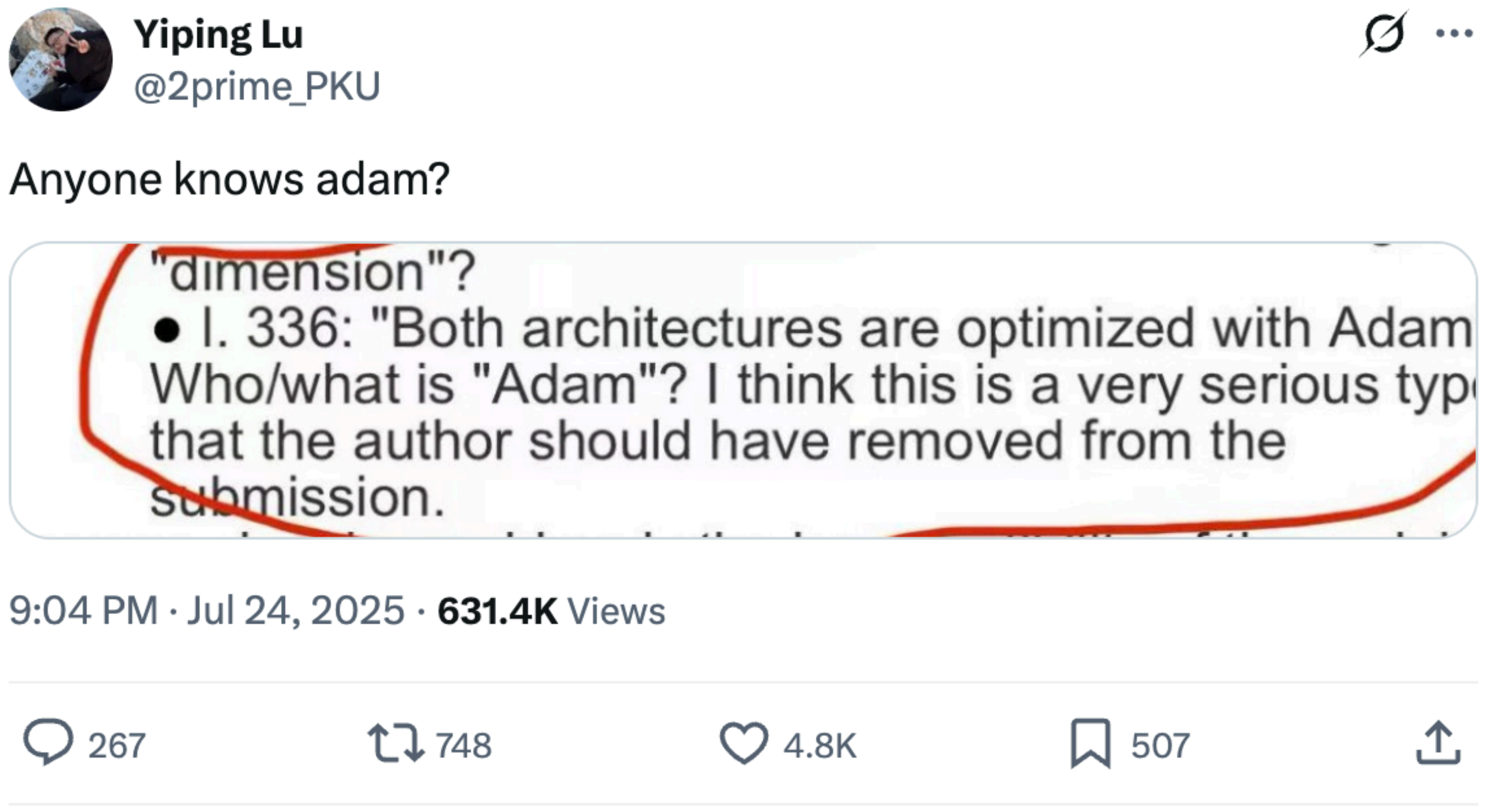
Decoding

Optimizers

Efficiency

Implementing Transformers

Optimizers


Yiping Lu
@2prime_PKU

Anyone knows adam?

"dimension"?
● l. 336: "Both architectures are optimized with Adam
Who/what is "Adam"? I think this is a very serious typ
that the author should have removed from the
submission.

9:04 PM · Jul 24, 2025 · 631.4K Views

267 748 4.8K 507

Optimization

Stochastic gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \mathbf{g} \qquad \mathbf{g} = \frac{\partial}{\partial \mathbf{w}} \mathcal{L}$$

- Very simple to code up
- "First-order" technique: only relies on having gradient
- Can avg gradient over a few examples and apply update once (minibatch)
- Setting step size is hard (decrease when held-out performance worsens?)

Newton's method

$$\mathbf{w} \leftarrow \mathbf{w} - \left(\frac{\partial^2}{\partial \mathbf{w}^2} \mathcal{L} \right)^{-1} \mathbf{g}$$

- Second-order technique
- Optimizes quadratic instantly

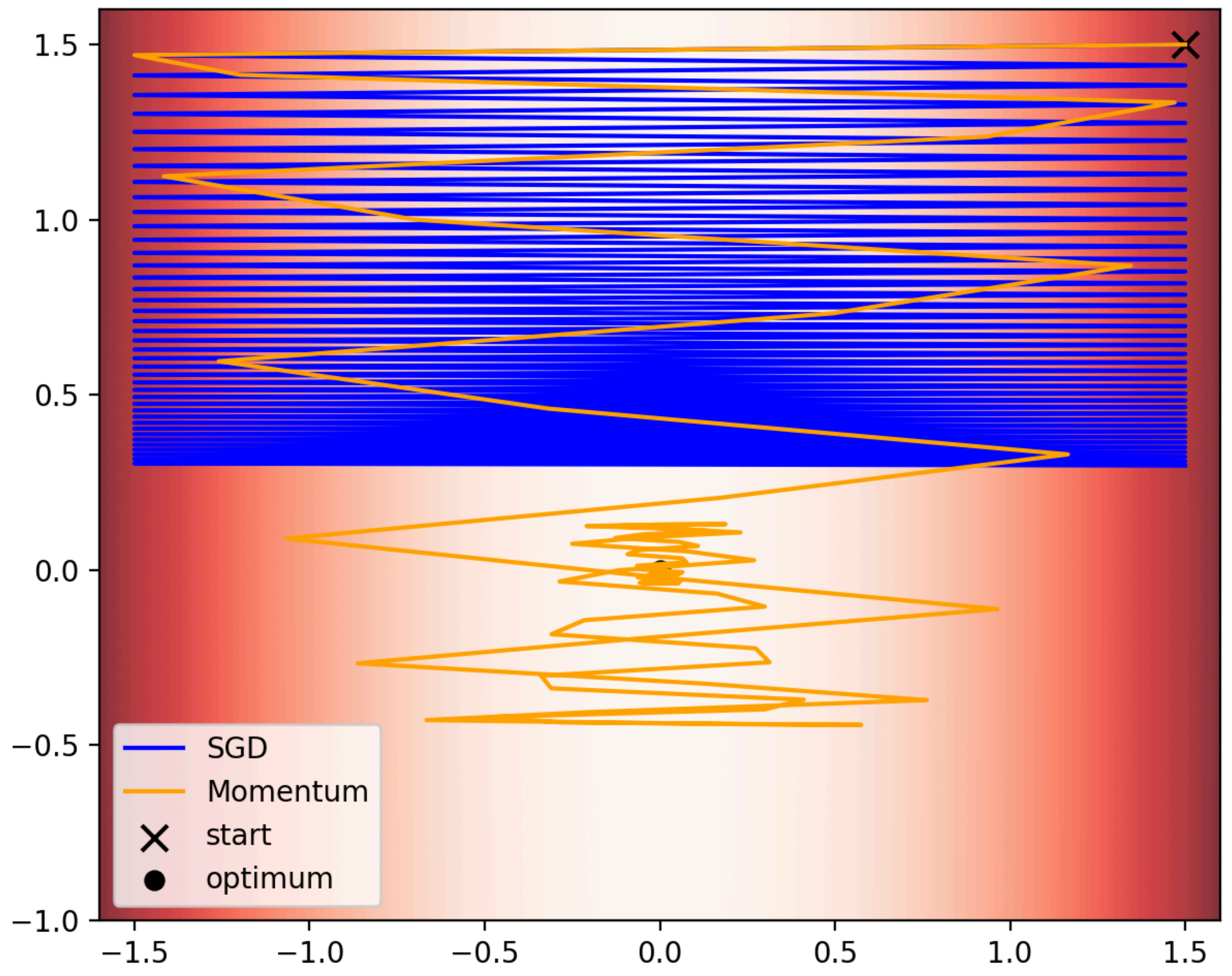
Inverse Hessian: *n* x *n* mat, expensive!

Quasi-Newton methods: L-BFGS, etc. approximate inverse Hessian

Adam

Idea 1: momentum

- Rather than apply the gradient directly, use an exponential weighted moving average of the gradient $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$



Adam

Idea 1: momentum

- Rather than apply the gradient directly, use an exponential weighted moving average of the gradient $m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$

Idea 2: adaptive scaling based on second moment

- Reduces update on features that have high variance

$$v \leftarrow \beta_2 v + (1 - \beta_2) g^2$$

$$\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v} + \epsilon}$$

AdamW

```
init($\theta$) (Initialize learnable parameters)
$m \leftarrow 0$ (Initial value of the first moment vector; same shape as $\theta$)
$v \leftarrow 0$ (Initial value of the second moment vector; same shape as $\theta$)
for $t = 1, \ldots, T$ do
    Sample batch of data $B_t$
    $g \leftarrow \nabla_\theta \ell(\theta; B_t)$ (Compute the gradient of the loss at the current time step)
    $m \leftarrow \beta_1 m + (1 - \beta_1) g$ (Update the first moment estimate)
    $v \leftarrow \beta_2 v + (1 - \beta_2) g^2$ (Update the second moment estimate)
    $\alpha_t \leftarrow \alpha \frac{\sqrt{1-(\beta_2)^t}}{1-(\beta_1)^t}$ (Compute adjusted $\alpha$ for iteration $t$)
    $\theta \leftarrow \theta - \alpha_t \frac{m}{\sqrt{v}+\epsilon}$ (Update the parameters)
    $\theta \leftarrow \theta - \alpha\lambda\theta$ (Apply weight decay)
end for
```

beta1: 0.9

beta2: 0.999

← This fix makes it "AdamW": changes where weight decay is used (Adam does it on gradient, scaled by rt(v))

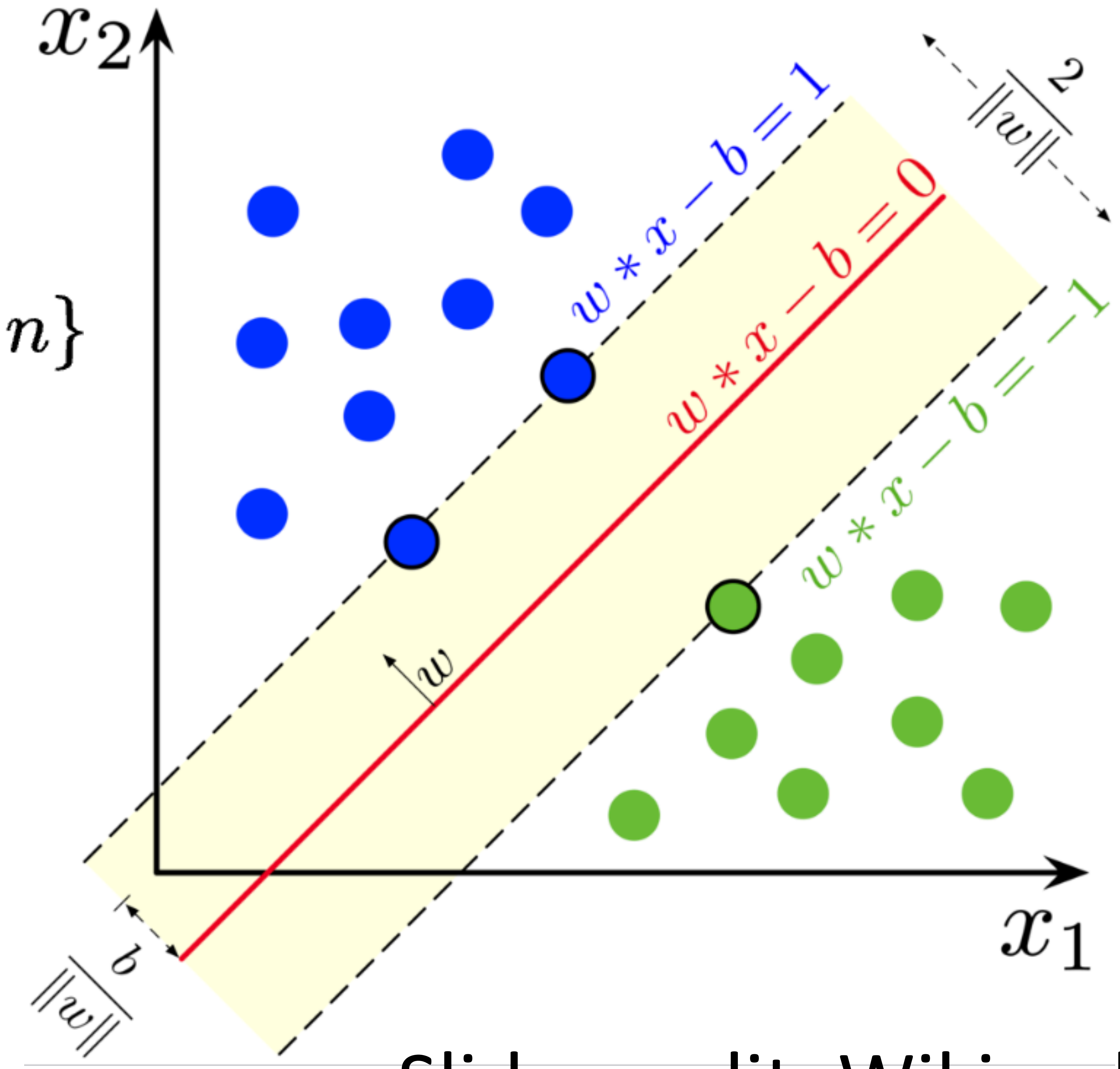
Let's look at weight decay…

Regularization in Classical ML

Support vector machines: classifier that maximizes the margin

$$\begin{aligned} &\underset{\mathbf{w},\, b}{\text{minimize}} && \frac{1}{2}\|\mathbf{w}\|^2 \\ &\text{subject to} && y_i(\mathbf{w}^\top \mathbf{x}_i - b) \geq 1 \quad \forall i \in \{1, \ldots, n\} \end{aligned}$$

(Introduce and minimize slack and you get an objective which involves optimizing for correctness + minimizing the norm of ***w***)



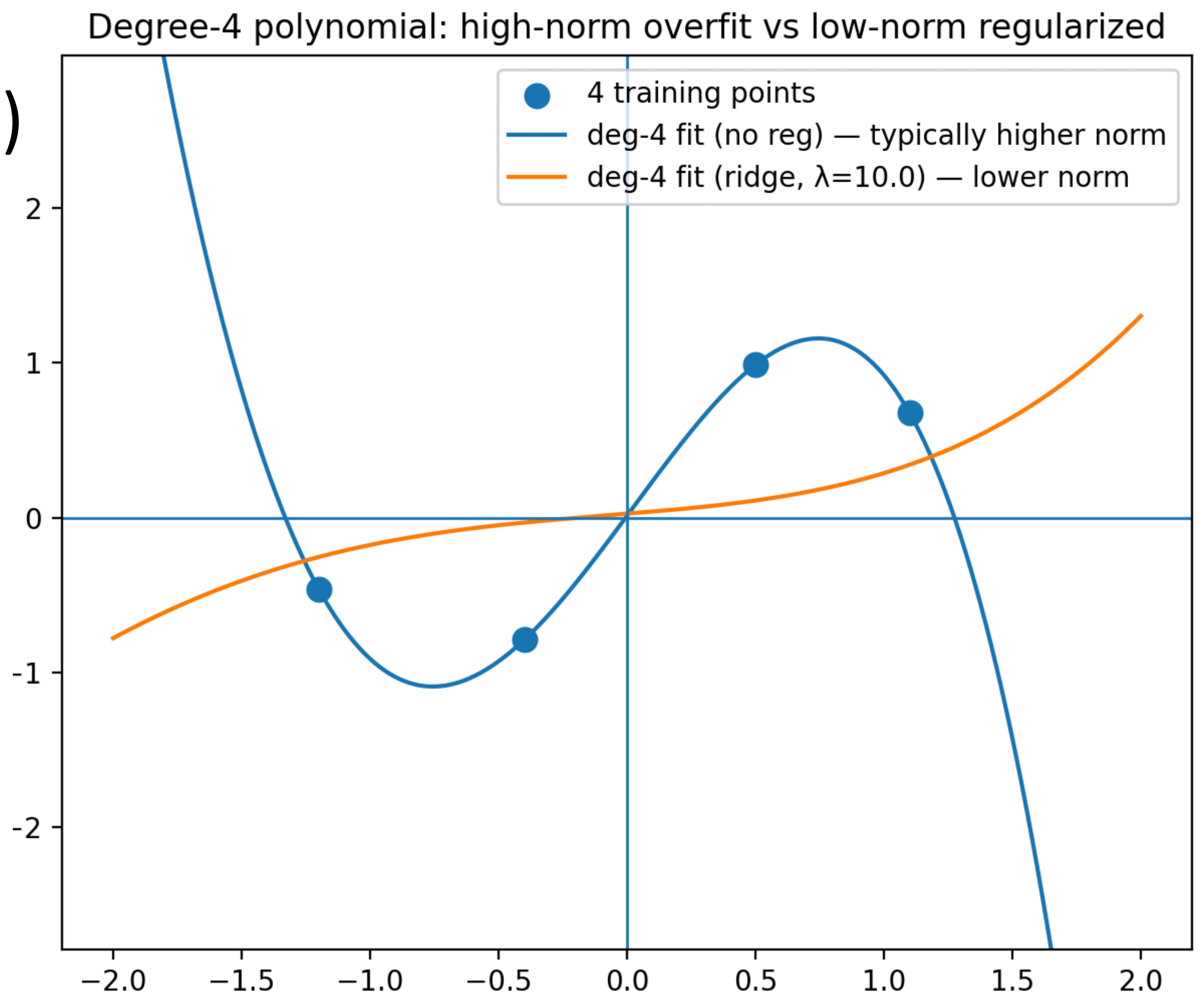
Slide credit: Wikipedia

Why is low weight norm regularizing?

Blue: high-norm.
Orange: low-norm (regularized)

Often we get the orange line anyway due to our choice of optimizer (SGD with low learning rates, not trained to full convergence)

Does this hold for deep learning?



Weight Decay

What's going on in this plot?

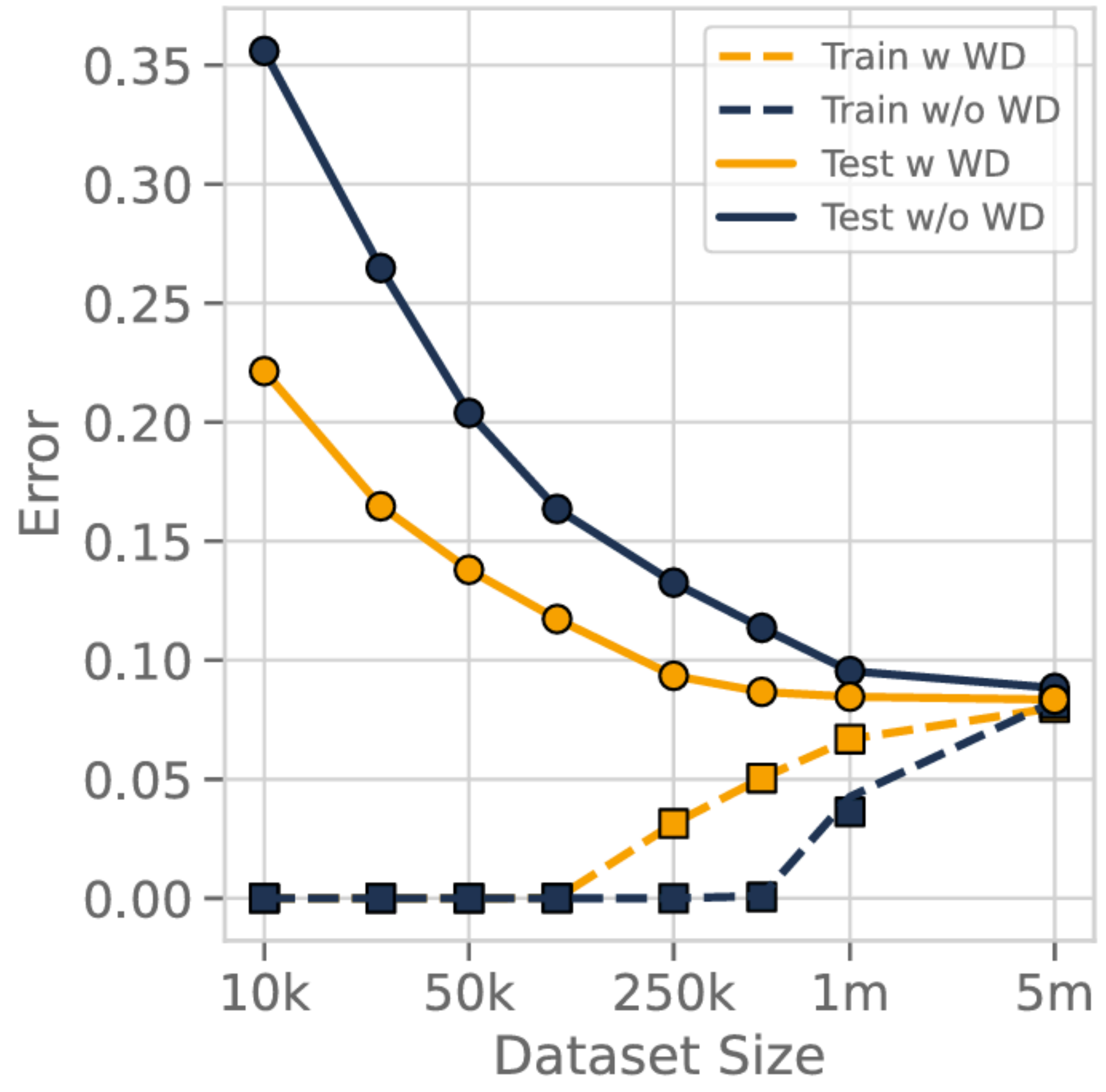
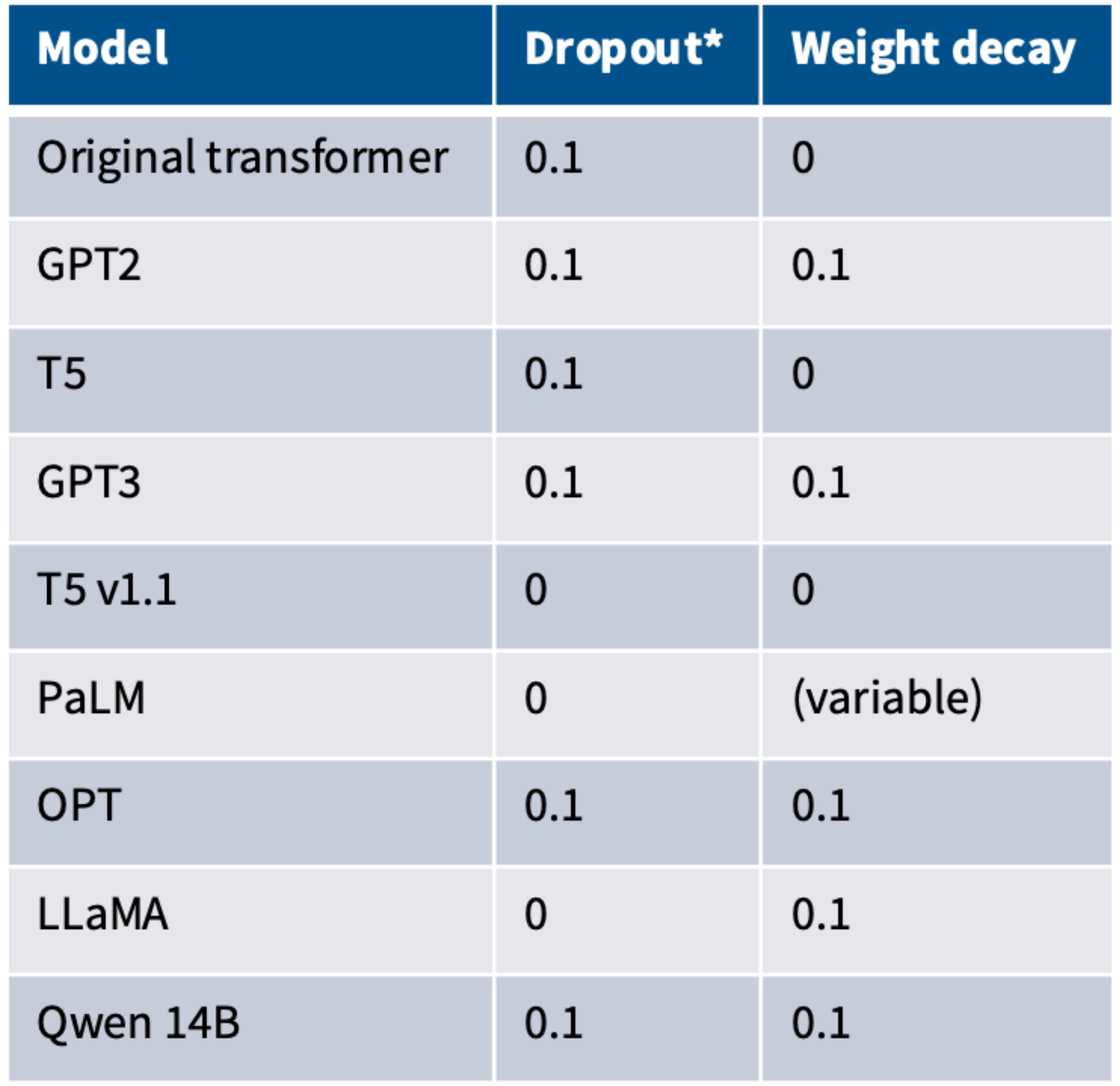


Figure 1: Test error vs. dataset size on CIFAR-10-5m for a *fixed* number of training iteration.

D'Angelo, Andriuschenko et al. (2023)

Hyperparameters

- Many open models do not discuss dropout and don’t use it, unclear if closed models do or don’t…



Model	Dropout*	Weight decay
Original transformer	0.1	0
GPT2	0.1	0.1
T5	0.1	0
GPT3	0.1	0.1
T5 v1.1	0	0
PaLM	0	(variable)
OPT	0.1	0.1
LLaMA	0	0.1
Qwen 14B	0.1	0.1

Slide credit: Tatsu Hashimoto / CS 336

Learning Rate Schedule

Warm-up phase: starting with full learning rate can lead to “destructive” updates (from batch norm era, 2015 or so)

Decreasing learning rate: classical optimization theory suggests that learning rate should decrease, decreasing smoothly is better than stepwise

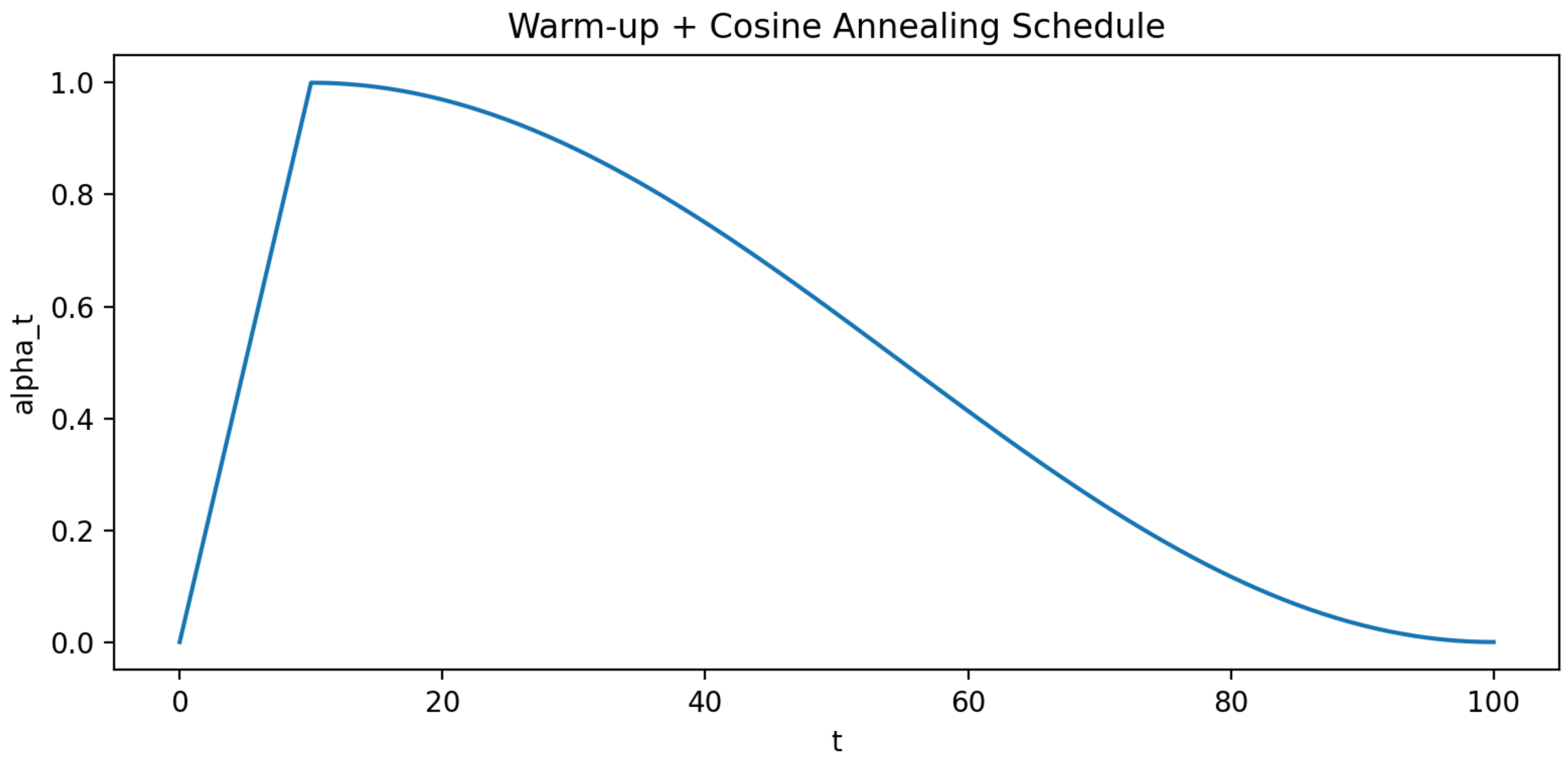
(Warm-up) If $t < T_w$, then $\alpha_t = \frac{t}{T_w}\alpha_{\max}$.

(Cosine annealing) If $T_w \le t \le T_c$, then $\alpha_t = \alpha_{\min} + \frac{1}{2}\left(1 + \cos\left(\frac{t - T_w}{T_c - T_w}\pi\right)\right)(\alpha_{\max} - \alpha_{\min})$

(Post-annealing) If $t > T_c$, then $\alpha_t = \alpha_{\min}$.

Where does cos annealing start in terms of magnitude? Where does it end?

Learning Rate Schedule



Our complete LLM

Train the BPE tokenizer: chunk, pretokenize in parallel, compute merges, serialize tokenizer

Tokenize the corpus: big file comes in, sequences of token ids come out, save as binary

Define architecture

Stream data batches using mmap, **compute gradients + optimize** on them

Decode to sample tokens

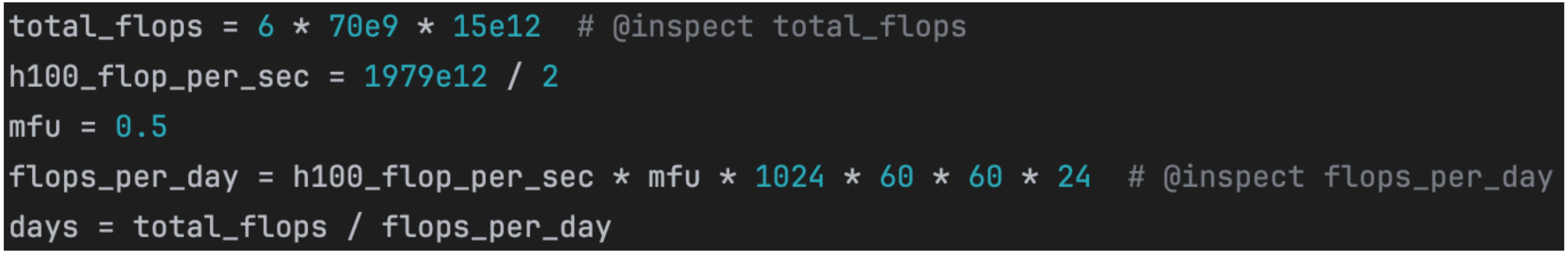
Efficiency

Slide credits for this section:
Percy Liang / CS 336

Resource Accounting

Question: How long would it take to train a 70B parameter model on 15T tokens on 1024 H100s?

where does 6 come from?

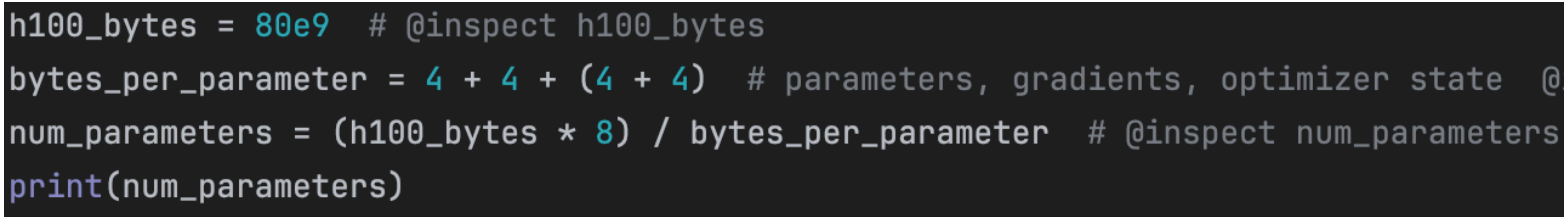


```
total_flops = 6 * 70e9 * 15e12  # @inspect total_flops
h100_flop_per_sec = 1979e12 / 2
mfu = 0.5
flops_per_day = h100_flop_per_sec * mfu * 1024 * 60 * 60 * 24  # @inspect flops_per_day
days = total_flops / flops_per_day
```

Answer: Around 143

what is MFU?

Question: What's the largest model trainable on 8 H100s with AdamW (naively)?



```
h100_bytes = 80e9  # @inspect h100_bytes
bytes_per_parameter = 4 + 4 + (4 + 4)  # parameters, gradients, optimizer state  @
num_parameters = (h100_bytes * 8) / bytes_per_parameter  # @inspect num_parameters
print(num_parameters)
```

Answer: 40B

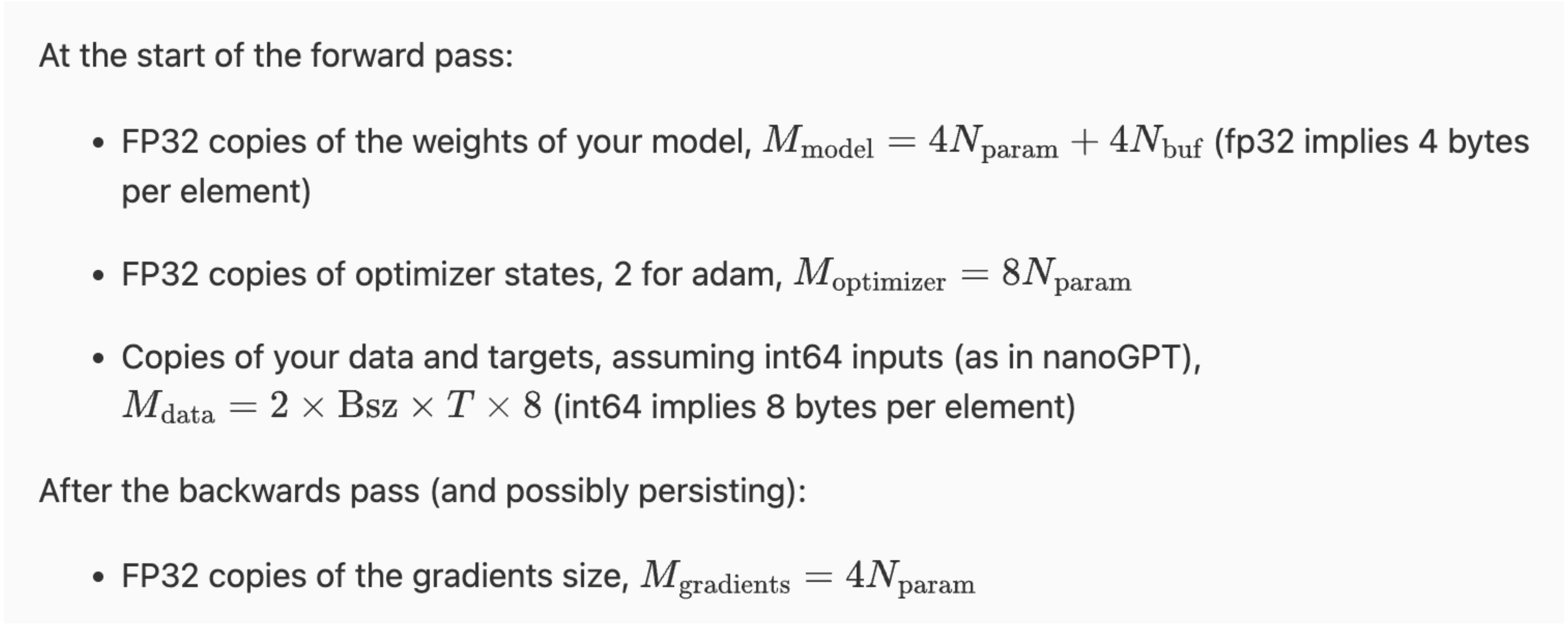
Resource Accounting

What has to be stored? (let's brainstorm)

https://erees.dev/transformer-memory/

Resource Accounting

What has to be stored?



At the start of the forward pass:

- FP32 copies of the weights of your model, $M_{\text{model}} = 4N_{\text{param}} + 4N_{\text{buf}}$ (fp32 implies 4 bytes per element)
- FP32 copies of optimizer states, 2 for adam, $M_{\text{optimizer}} = 8N_{\text{param}}$
- Copies of your data and targets, assuming int64 inputs (as in nanoGPT), $M_{\text{data}} = 2 \times \mathrm{Bsz} \times T \times 8$ (int64 implies 8 bytes per element)

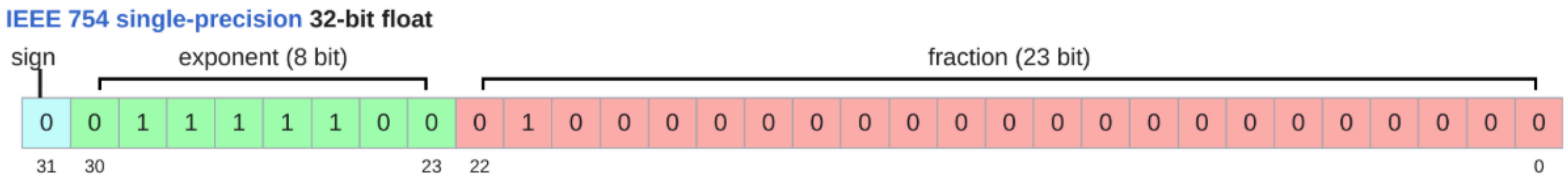
After the backwards pass (and possibly persisting):

- FP32 copies of the gradients size, $M_{\text{gradients}} = 4N_{\text{param}}$

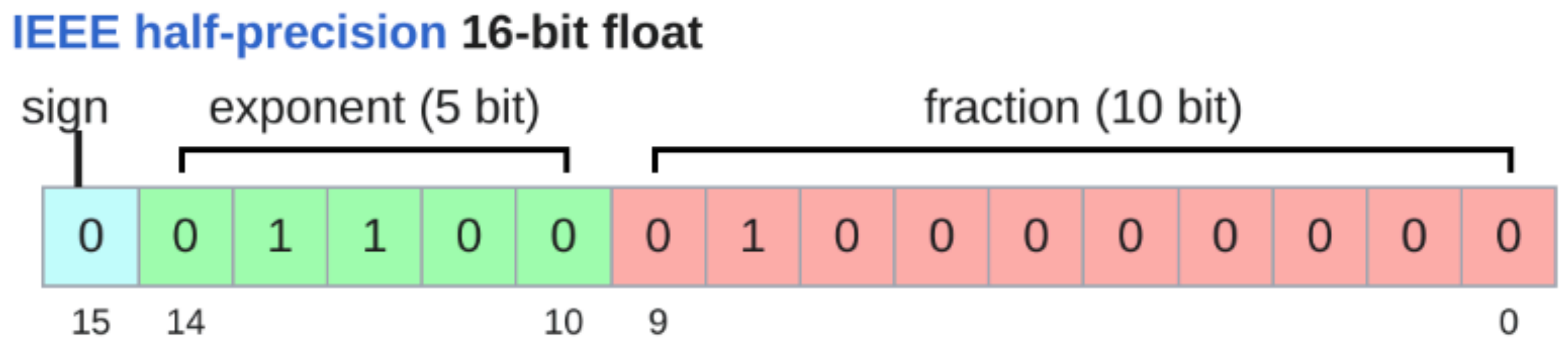
What is fp32, int64, …?

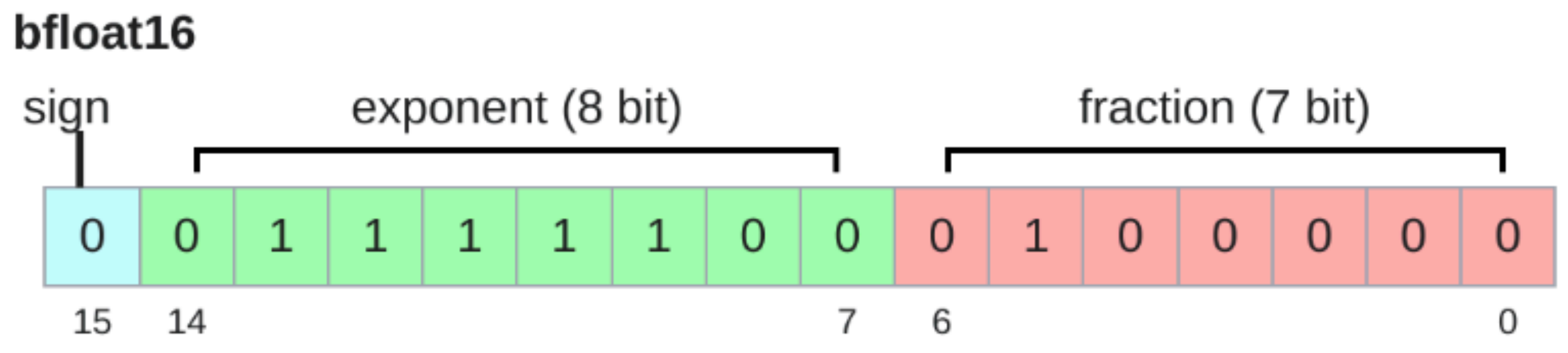
https://erees.dev/transformer-memory/

Data Types



"brain float" from Google





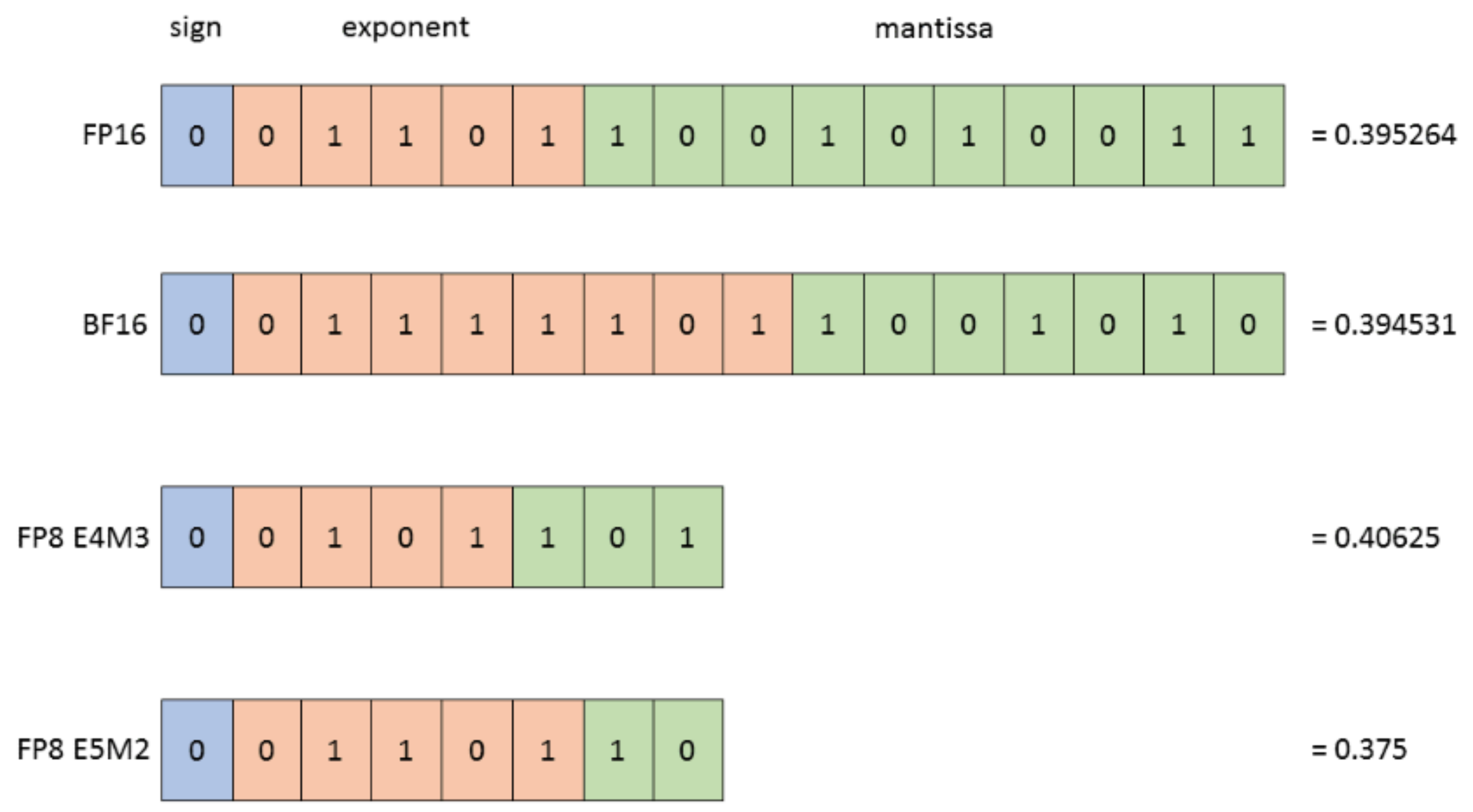
1e-8 is an underflow

Goes down to 1e-38 but lower resolution

How much memory to store a linear layer from d_{model} to 4 d_{model} if dmodel is 12k?

Data Types

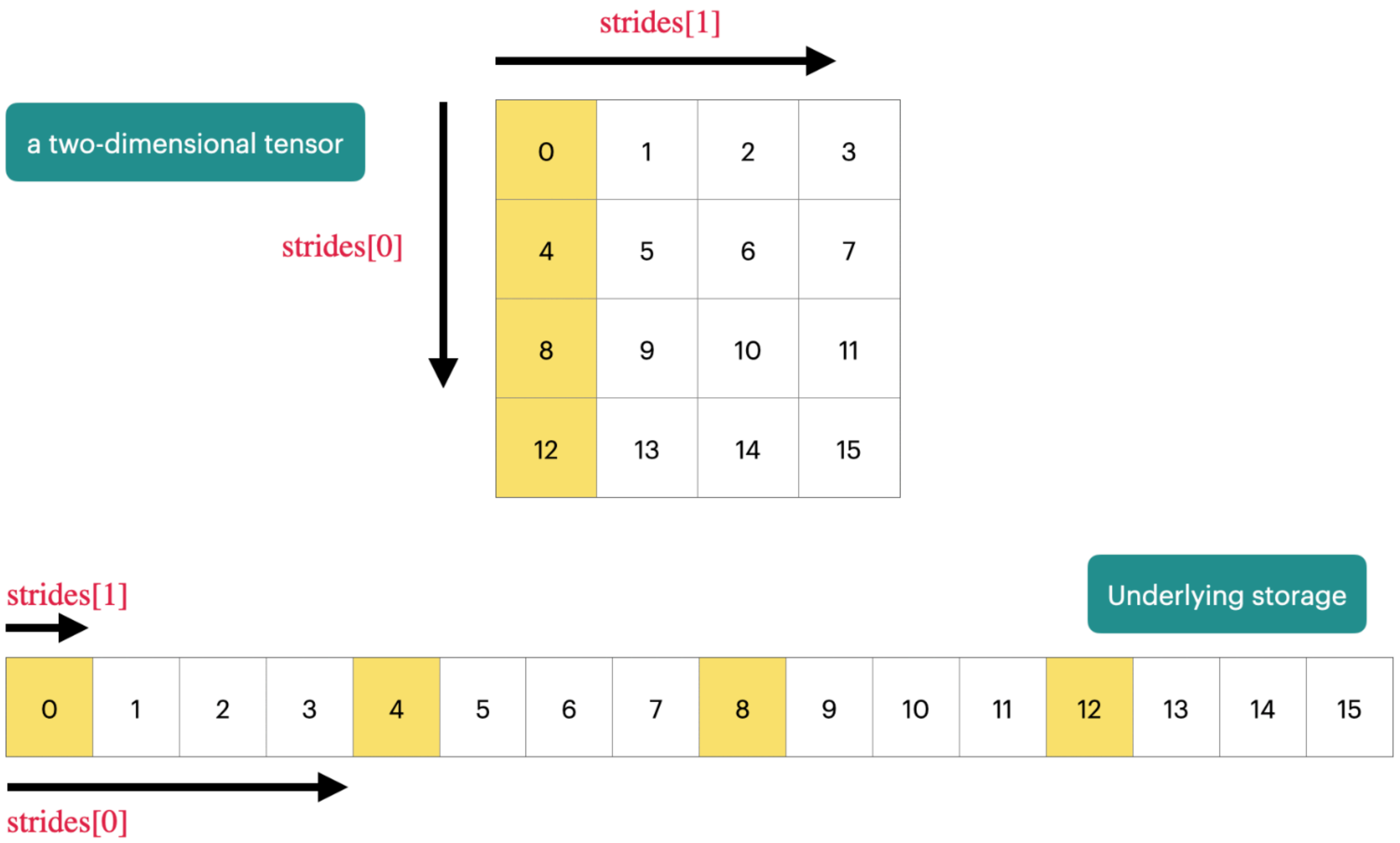
https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html



Both E4M3 and E5M2 supported on H100

Mixed precision training gets the best of both worlds: fast without losing accuracy

Tensors



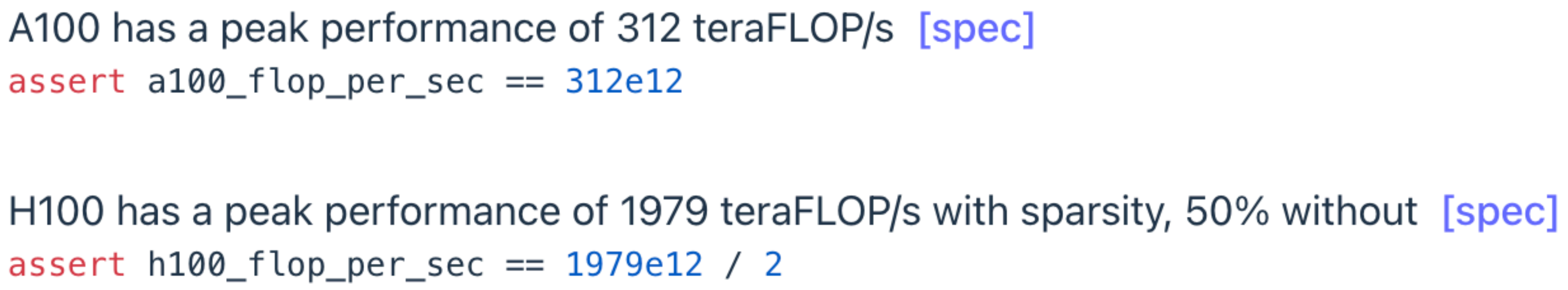
Slicing, transpose, etc. do not reallocate memory, but just change the view

Resource Accounting

FLOPs: floating-point operations (measure of computation done)

- How expensive is adding two M x N matrices A and B?
- How expensive is A^TB?

FLOP/s: floating-point operations per second (also written as FLOPS), which is used to measure the speed of hardware.



A100 has a peak performance of 312 teraFLOP/s [spec]

```
assert a100_flop_per_sec == 312e12
```

H100 has a peak performance of 1979 teraFLOP/s with sparsity, 50% without [spec]

```
assert h100_flop_per_sec == 1979e12 / 2
```

Resource Accounting: Forward

Case study: linear layer. How many FLOPs for the forward pass?

Batch = 1024

Dim = 256

Hidden dim size = 64

2 * 1024 * 256 * 64

Resource Accounting: Backward

Case study: two linear layers. How many FLOPs for forward+backward passes?

Batch = 1024

Dim = 256

Hidden dim size = 64

Output size = 64

Model: x --w1--> h1 --w2--> h2 -> loss

Compute grad for w2: 2 * 1024 * 64 * 64
Do backward pass to w1: 2 * 1024 * 64 * 64
Overall this is 2x as much as the forward pass

Total F+B:
6 * data points * params

Model FLOPs Utilization (MFU)

Further reading:

https://erees.dev/transformer-memory/

https://www.adamcasson.com/posts/transformer-flops

Implementing Transformers

Notation

Math: we think of operations as going right to left

Wg(VX) : take X, multiply by V, apply g, multiply by W

If X has dimension *m* x *n*, what dimension can V have?

PyTorch: we think of operations as going left to right

Y = Linear(X): X is a tensor ending in the dimension d_{in} to be manipulated (often batch size x seq len x dimension).
Write this in math as XV^T

einsum

- Language for tensor manipulation using labels for indices
- Specify indices of input tensor and which indices remain in the output. “Leftover” indices are summed out.

```
einsum(X, W, ‘i, o i -> o’)

einsum(X, W, ‘i, i o -> o’)

einsum(X, W, ‘o i, o i ->')

einsum(X, W, ‘l i, i -> l')

einsum(X, W, '... i, o i -> ... o')
```

- Can also use einops.rearrange to change shape, introduce new dimensions, etc. Try this when manipulating the heads in multi-head attention

einsum

How do you keep track of tensor dimensions?

Old way:

```
x = torch.ones(2, 2, 1, 3)  # batch seq heads hidden  @inspect x
```

New (jaxtyping) way:

```
x: Float[torch.Tensor, "batch seq heads hidden"] = torch.ones(2, 2, 1, 3)  # @inspect x
```

Note: this is just documentation (no enforcement).

Numerical Stability

Log softmax: distribute log to numerator and denominator, don’t apply directly

Compare:

[-30, -29, -29, -30] =>

[-1, 0, 0, -1] =>

Exercise: try this with different datatypes, see which ones underflow

Random seeds

Fixing random seeds can help debugging

```
# Torch
seed = 0
torch.manual_seed(seed)

# NumPy
import numpy as np
np.random.seed(seed)

# Python
import random
random.seed(seed)
```

Administrative details and recap

Tokenizers

Byte Pair Encoding

Decoding

Optimizers

Efficiency

Implementing Transformers

Next time

We'll start understanding GPUs and acceleration a bit better

Flash Attention, other inference-time speedups