CS388: Natural Language Processing

Lecture 21: Efficiency and LLMs

Greg Durrett

TEXAS
The University of Texas at Austin

Announcements

- Check-ins due today, will be graded as promptly as we can

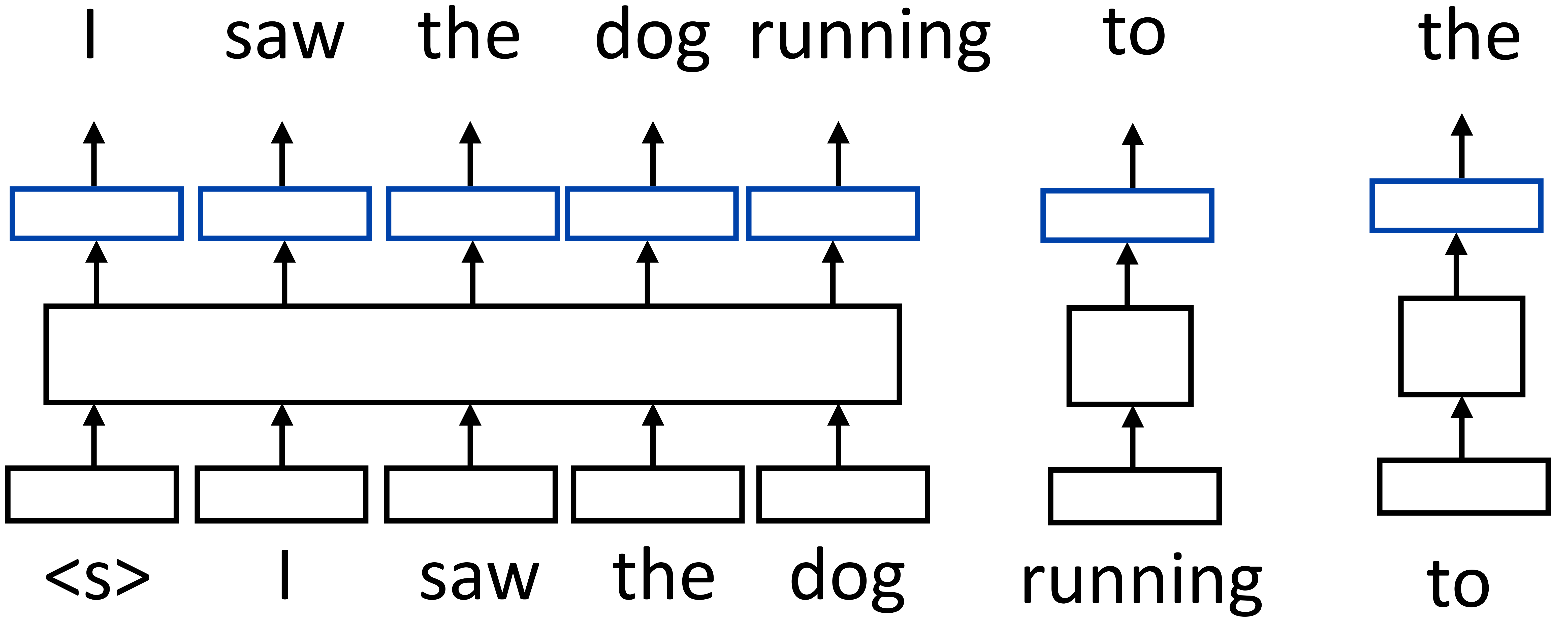
- Final presentations start in 2.5 weeks, reports due May 3

This Lecture

- Decoding optimizations: exact decoding, but faster
  - Speculative decoding
  - Medusa heads
  - Flash attention
- Model pruning
  - Pruning LLMs
  - Distilling LLMs
- Model compression

Decoding Optimizations

Decoding Basics



L transformer layers

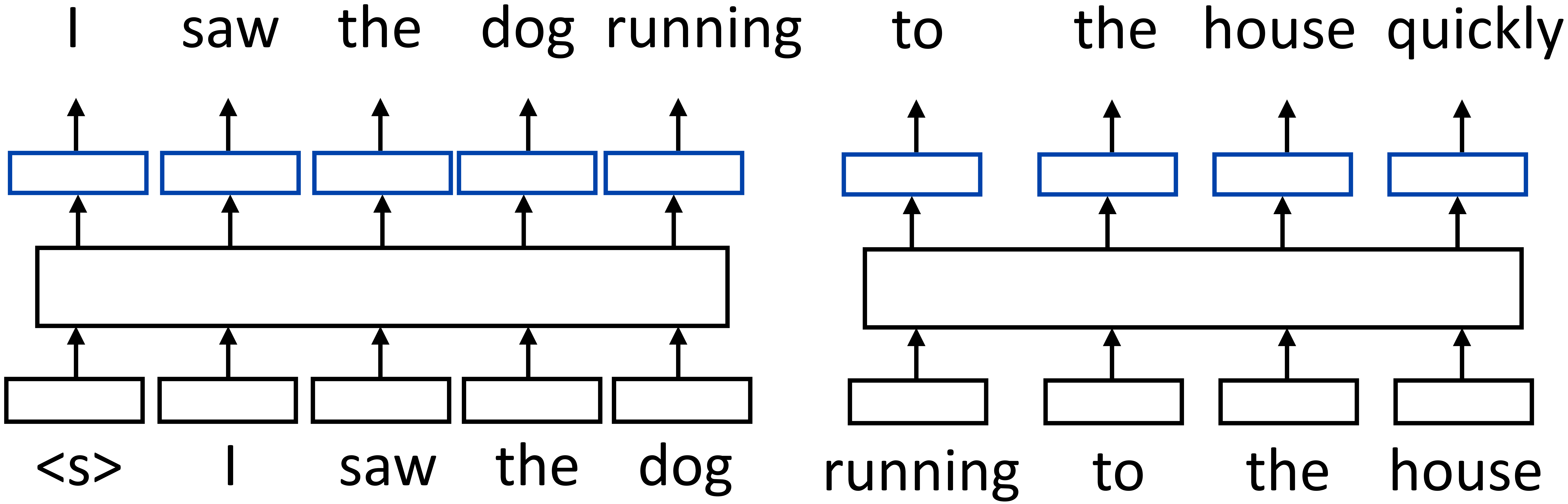
Prompt (prefix of *p* tokens) Decoded tokens (*k*)

Operations for one decoder pass: $O(pL)$

Operations for *k* decoder passes: $O(pk^2L)$

Transformer layers (non-parallelizable ops): $O(kL)$

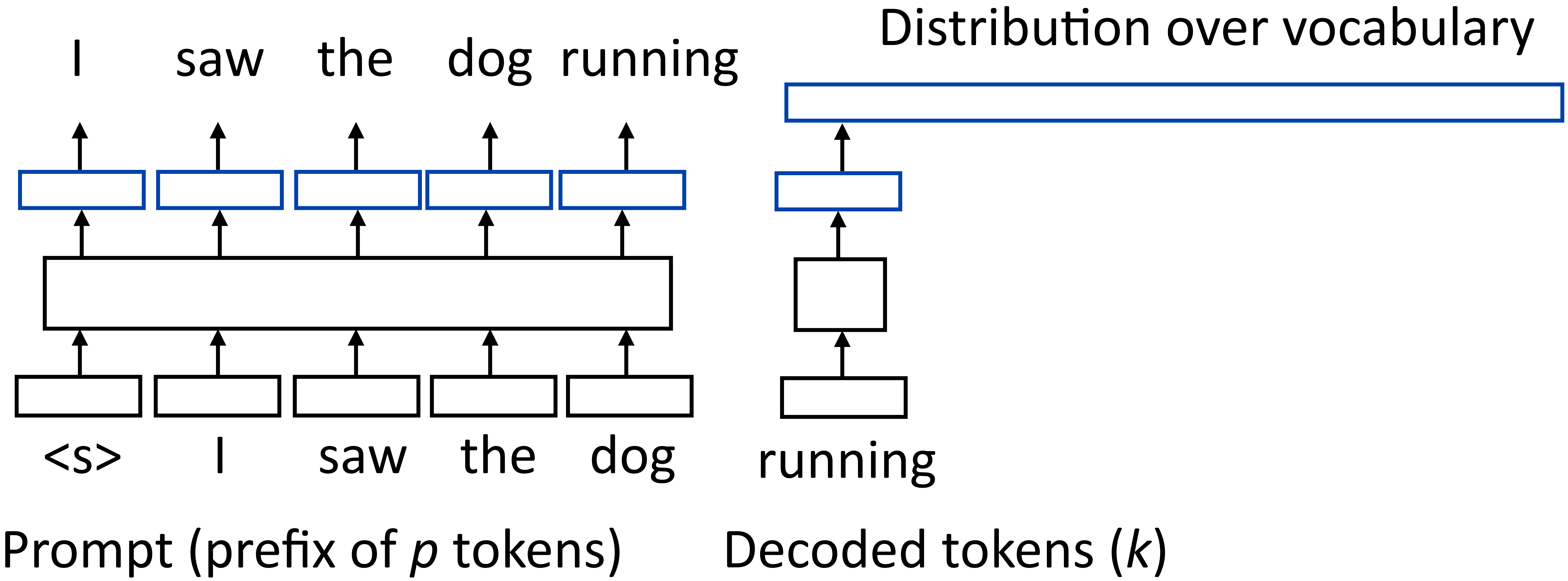
Speculative Decoding



Prompt (prefix of *p* tokens) Decoded tokens (*k*)

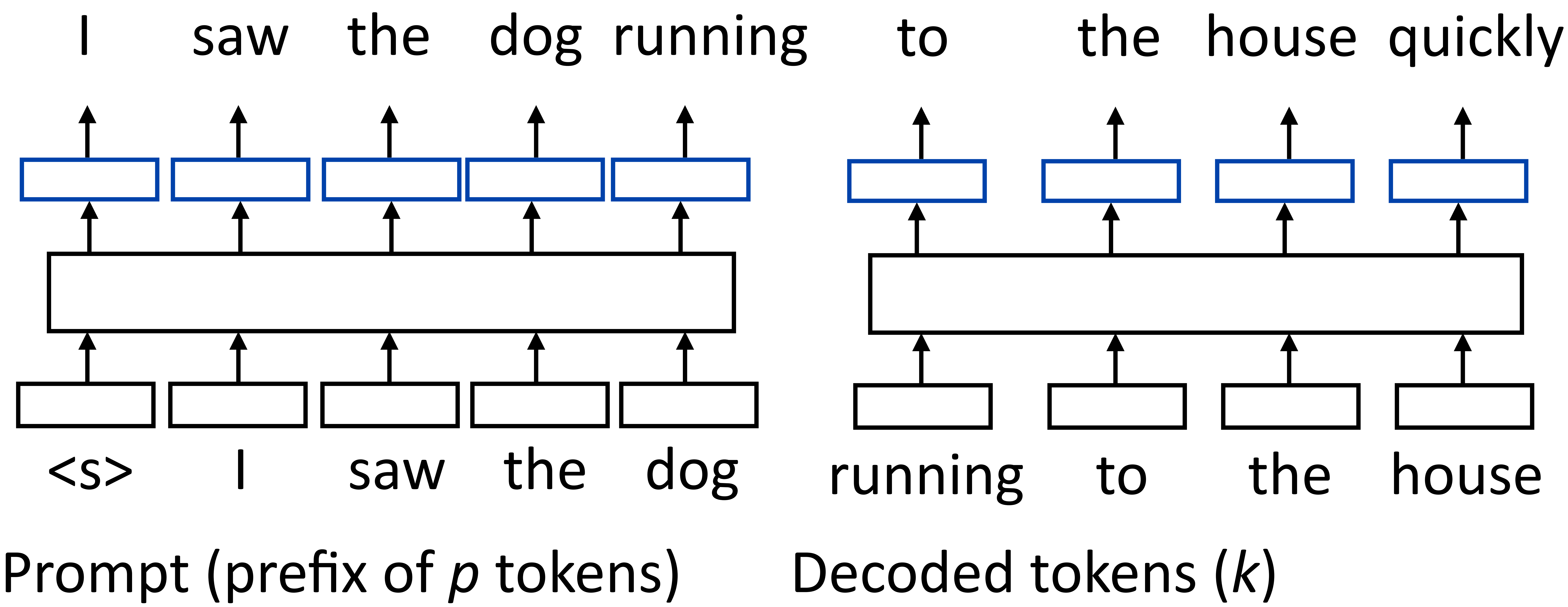
- Key idea a forward pass for several tokens at a time is O(*L*) serial steps, since the tokens can be computed in parallel
- Can we predict many tokens with a weak model and then "check" them with a single forward pass?

Speculative Decoding



- When sampling, we need the whole distribution
- When doing greedy decoding, we only need to know what token was the max

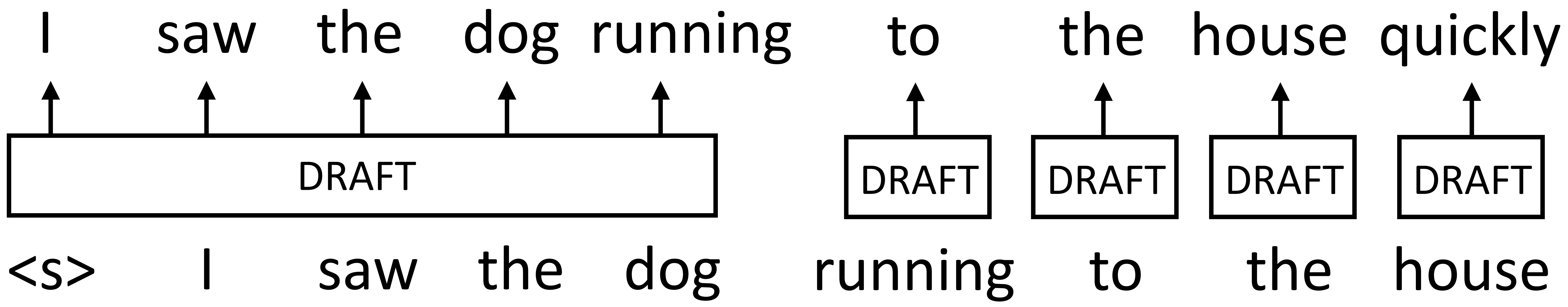
Speculative Decoding



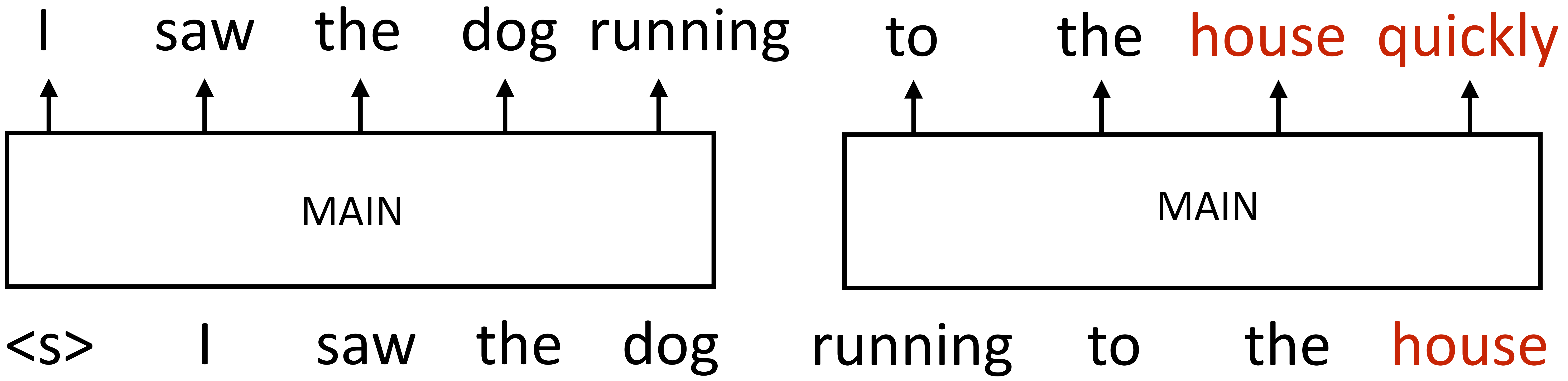
- **We can use a small, cheap model to do inference, then check that "to", "the", "house", "quickly" are really the best tokens from a bigger model**

Leviathan et al. (2023)

Speculative Decoding: Flow



- Produce decoded tokens one at a time from a fast draft model…



- Confirm that the tokens are the max tokens from the slower main model. Any “wrong” token invalidates the rest of the sequence

Speculative Decoding

Leviathan et al. (2023)

[START] japan ' s benchmark ~~bond~~ n

[START] japan ' s benchmark nikkei 22 ~~,~~5

[START] japan ' s benchmark nikkei 225 index rose 22 ~~.~~6

[START] japan ' s benchmark nikkei 225 index rose 226 . 69 ~~,~~ points

[START] japan ' s benchmark nikkei 225 index rose 226 . 69 points , or 0 1

[START] japan ' s benchmark nikkei 225 index rose 226 . 69 points , or 1 . 5 percent , to 10 , 9859

- Can also adjust this to use sampling. Treat this as a proposal distribution *q*(x) and may need to reject + resample (rejection sampling)

Speculative Decoding

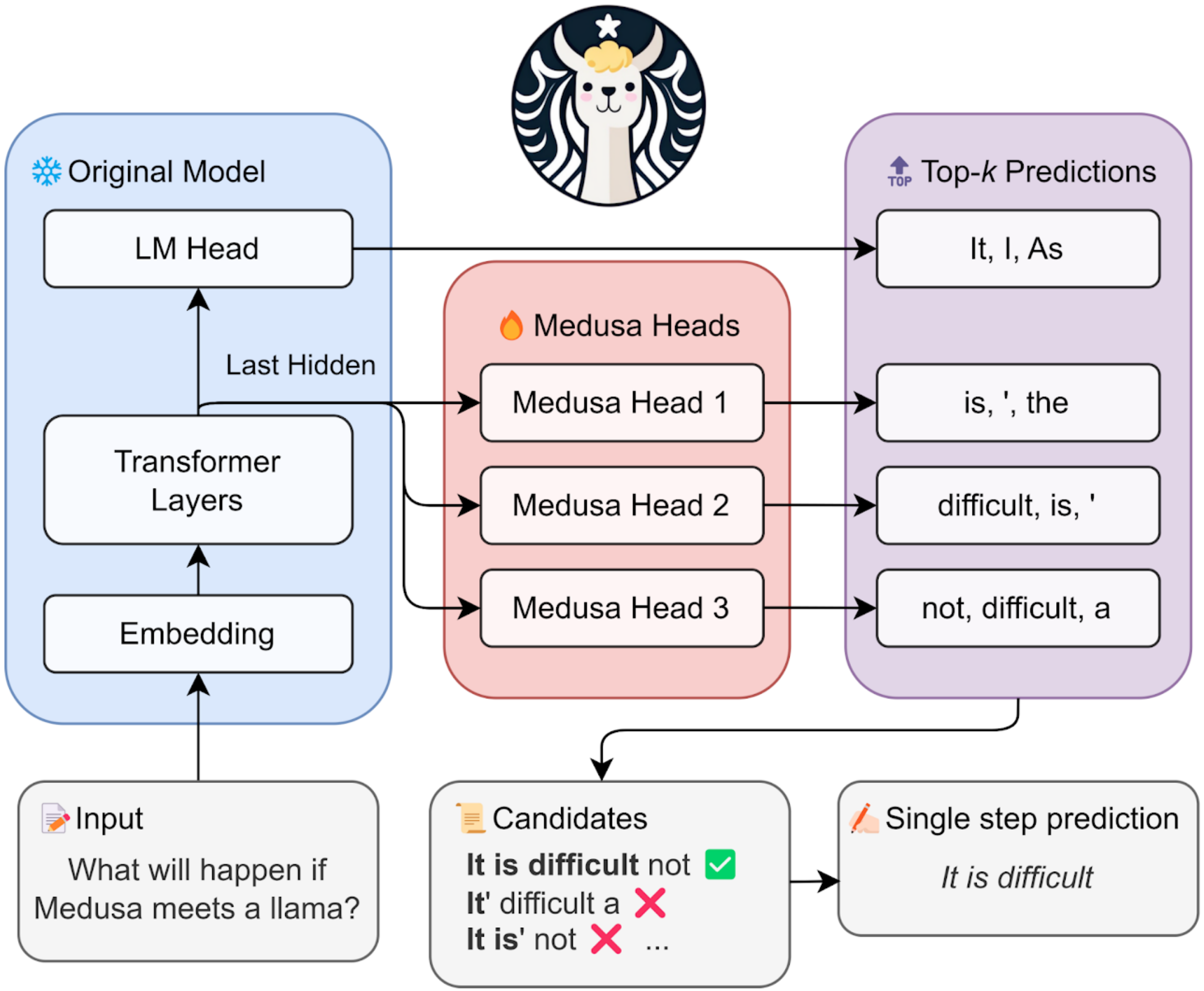
- Find the first index that was rejected by the sampling procedure, then resample from there

Inputs: $M_p, M_q, prefix$.
$\triangleright$ Sample γ guesses $x_{1,\ldots,\gamma}$ from M_q autoregressively.
for $i = 1$ **to** γ **do**
  $q_i(x) \leftarrow M_q(prefix + [x_1, \ldots, x_{i-1}])$
  $x_i \sim q_i(x)$
end for
$\triangleright$ Run M_p in parallel.
$p_1(x), \ldots, p_{\gamma+1}(x) \leftarrow$
  $M_p(prefix), \ldots, M_p(prefix + [x_1, \ldots, x_\gamma])$
$\triangleright$ Determine the number of accepted guesses n.
$r_1 \sim U(0,1), \ldots, r_\gamma \sim U(0,1)$
$n \leftarrow \min(\{i - 1 \mid 1 \leq i \leq \gamma, r_i > \frac{p_i(x)}{q_i(x)}\} \cup \{\gamma\})$
$\triangleright$ Adjust the distribution from M_p if needed.
$p'(x) \leftarrow p_{n+1}(x)$
if $n < \gamma$ **then**
  $p'(x) \leftarrow norm(max(0, p_{n+1}(x) - q_{n+1}(x)))$
end if
$\triangleright$ Return one token from M_p, and n tokens from M_q.
$t \sim p'(x)$
return $prefix + [x_1, \ldots, x_n, t]$

Leviathan et al. (2023)

Medusa Heads

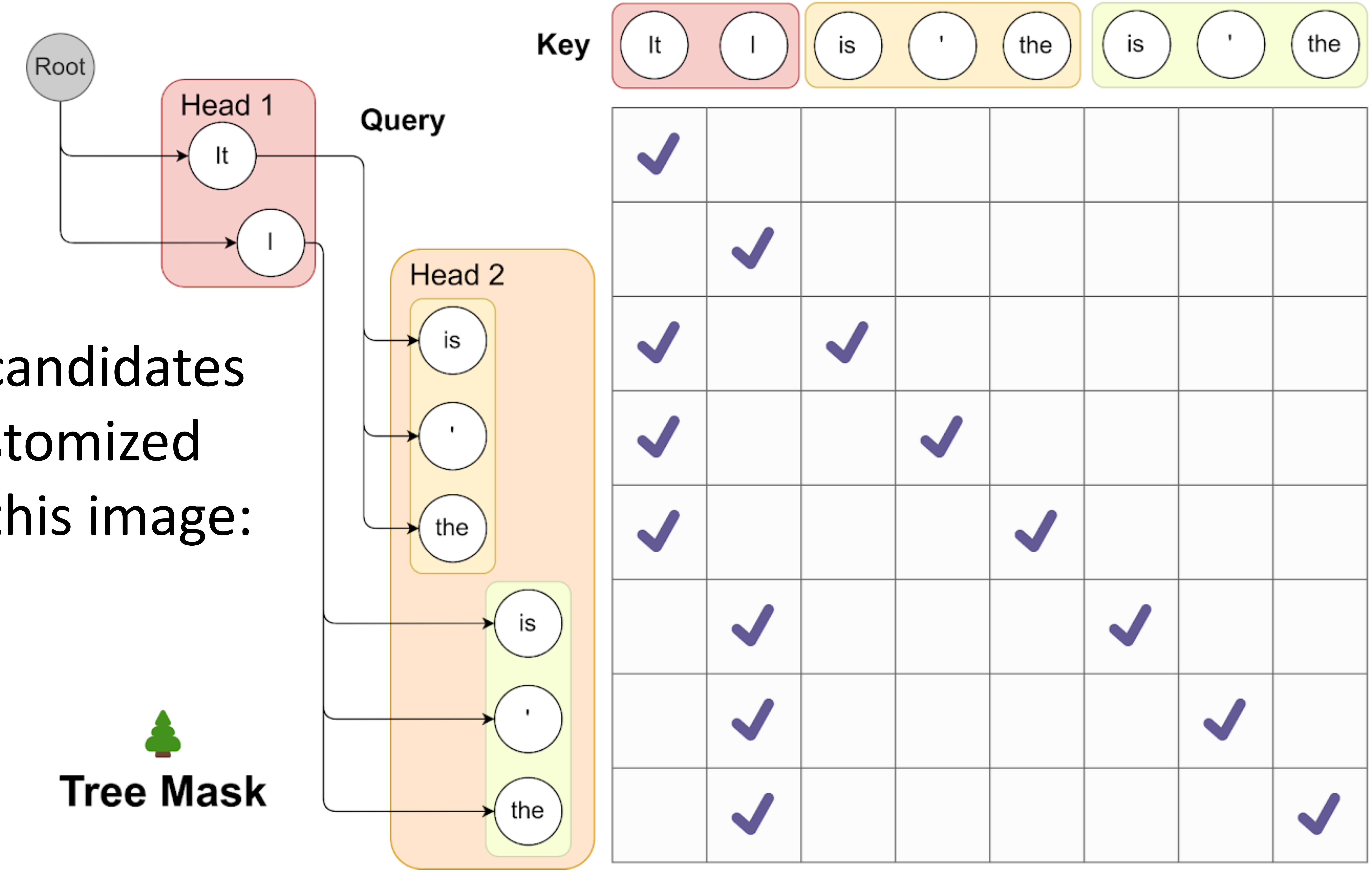
- The “draft model” consists of multiple prediction heads trained to predict the next k tokens



https://www.together.ai/blog/medusa

Medusa Heads

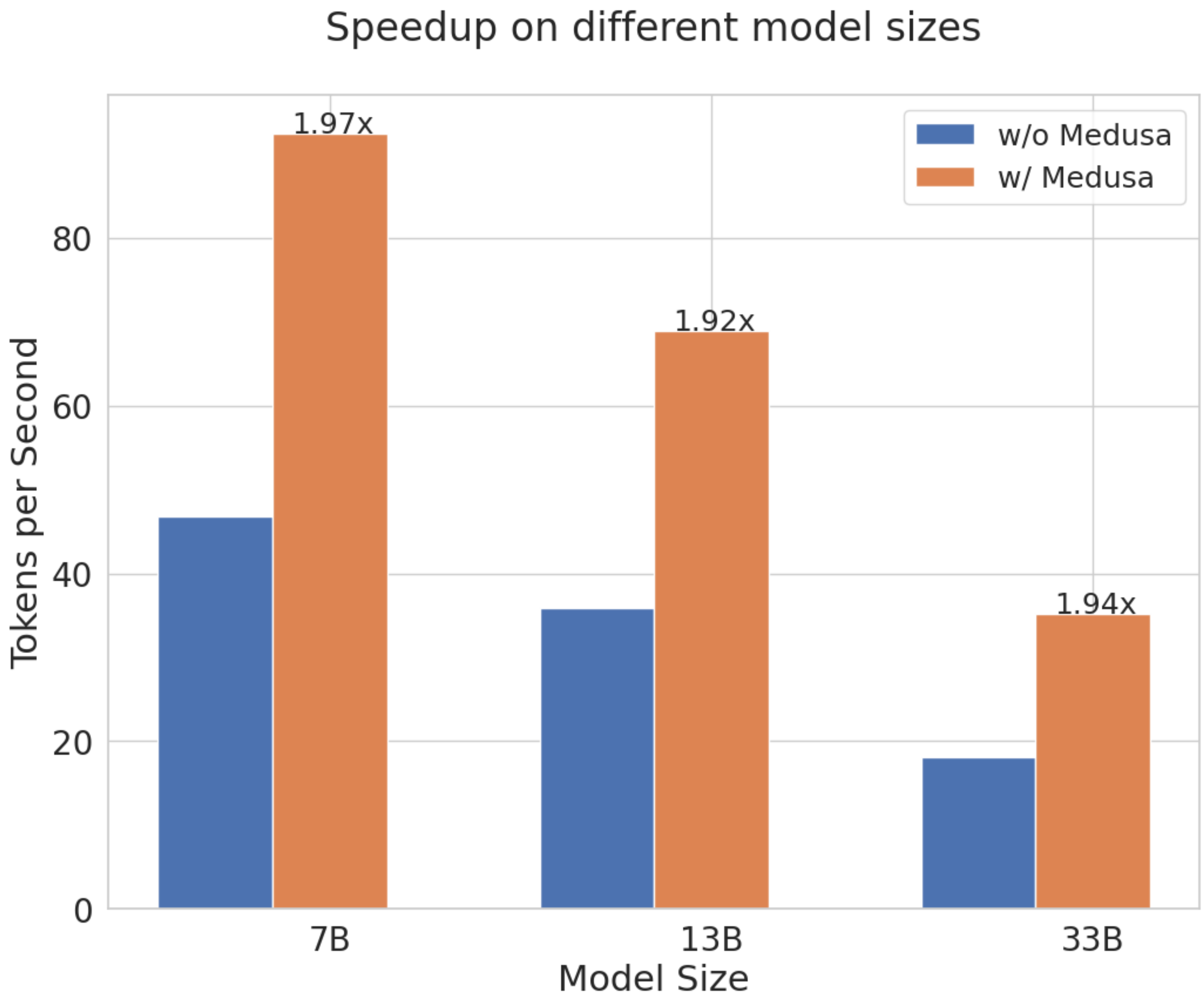
- Evaluate multiple candidates at once using a customized attention layer. In this image: 2 x 3 candidates



https://www.together.ai/blog/medusa

Medusa Heads

- Speedup with no loss in accuracy!

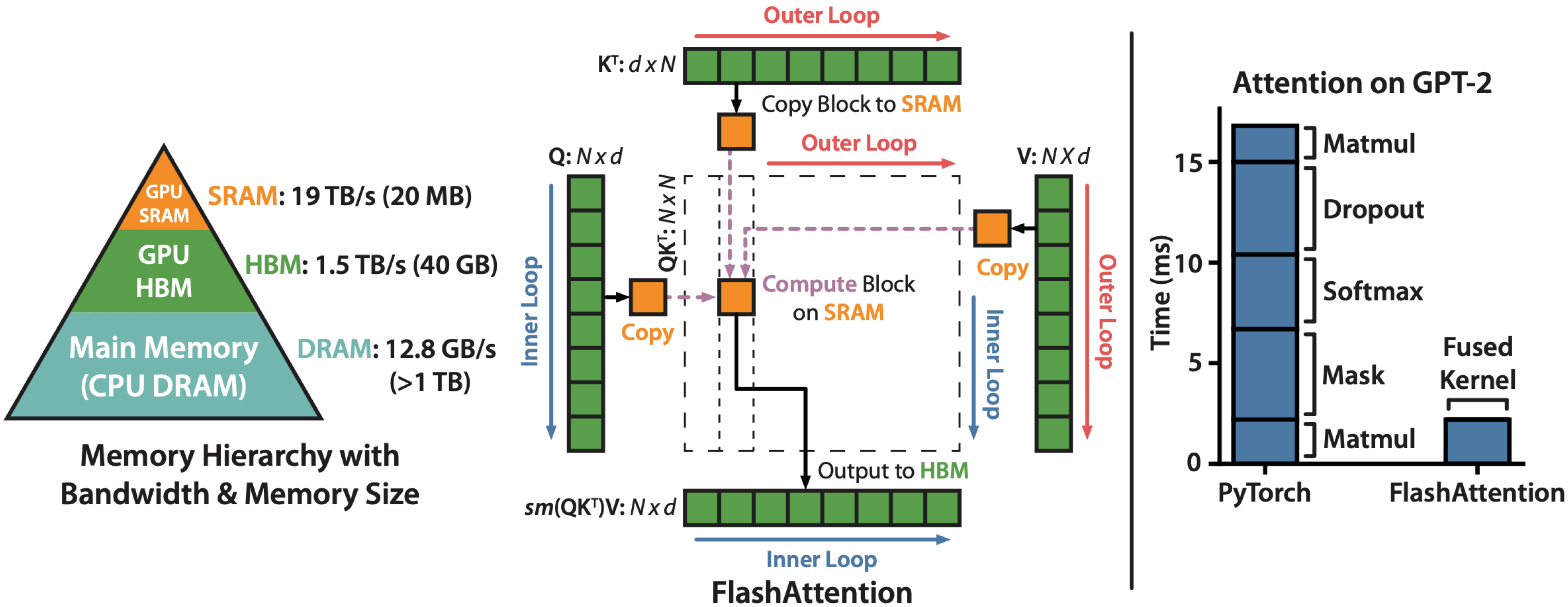


https://www.together.ai/blog/medusa

Other Decoding Improvements

- Most other approaches to speeding up require changing the model (making a faster Transformer) or making it smaller (distillation, pruning; discussed next)
- Batching parallelism: improve throughput by decoding many examples in parallel. (Does not help with latency, and it's a little bit harder to do in production if requests are coming in asynchronously)
- Low-level hardware optimizations?
  - Easy things like caching (KV cache: keys + values for context tokens are cached across multiple tokens)

Flash Attention



- Does extra computation during attention, but avoids expensive reads/writes to GBU “high-bandwidth memory.” Recomputation is all in SRAM and is very fast
- Essentially: store a running sum for the softmax, compute values as needed

Flash Attention

Algorithm 0 Standard Attention Implementation

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM.

1: Load $\mathbf{Q}, \mathbf{K}$ by blocks from HBM, compute $\mathbf{S} = \mathbf{Q}\mathbf{K}^\top$, write $\mathbf{S}$ to HBM.
2: Read $\mathbf{S}$ from HBM, compute $\mathbf{P} = \mathrm{softmax}(\mathbf{S})$, write $\mathbf{P}$ to HBM.
3: Load $\mathbf{P}$ and $\mathbf{V}$ by blocks from HBM, compute $\mathbf{O} = \mathbf{P}\mathbf{V}$, write $\mathbf{O}$ to HBM.
4: Return $\mathbf{O}$.

Algorithm 1 FlashAttention

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, on-chip SRAM of size M.

[dividing stuff into blocks]

5: **for** $1 \le j \le T_c$ **do**
6: Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.
7: **for** $1 \le i \le T_r$ **do**
8: Load $\mathbf{Q}_i, \mathbf{O}_i, \ell_i, m_i$ from HBM to on-chip SRAM.
9: On chip, compute $\mathbf{S}_{ij} = \mathbf{Q}_i\mathbf{K}_j^T \in \mathbb{R}^{B_r \times B_c}$.

[more computation, writes to HBM]

Flash Attention

Models	ListOps	Text	Retrieval	Image	Pathfinder	Avg	Speedup
Transformer	36.0	63.6	81.6	42.3	72.7	59.3	-
FlashAttention	37.6	63.9	81.4	43.5	72.7	59.8	2.4×
Block-sparse FlashAttention	37.0	63.0	81.3	43.6	73.3	59.6	**2.8**×
Linformer [84]	35.6	55.9	77.7	37.8	67.6	54.9	2.5×
Linear Attention [50]	38.8	63.2	80.7	42.6	72.5	59.6	2.3×
Performer [12]	36.8	63.6	82.2	42.1	69.9	58.9	1.8×
Local Attention [80]	36.1	60.2	76.7	40.6	66.6	56.0	1.7×
Reformer [51]	36.5	63.8	78.5	39.6	69.4	57.6	1.3×
Smyrf [19]	36.1	64.1	79.0	39.6	70.5	57.9	1.7×

- Gives a speedup for free — with no cost in accuracy (modulo numeric instability)
- Outperforms the speedup from many other approximate Transformer methods, which perform substantially worse

Model Compression

Approaches to Compression

- Pruning: can we reduce the number of neurons in the model?
  - Basic idea: remove low-magnitude weights
  - Issue: sparse matrices are not fast, matrix multiplication is very fast on GPUs so you don't save any time!

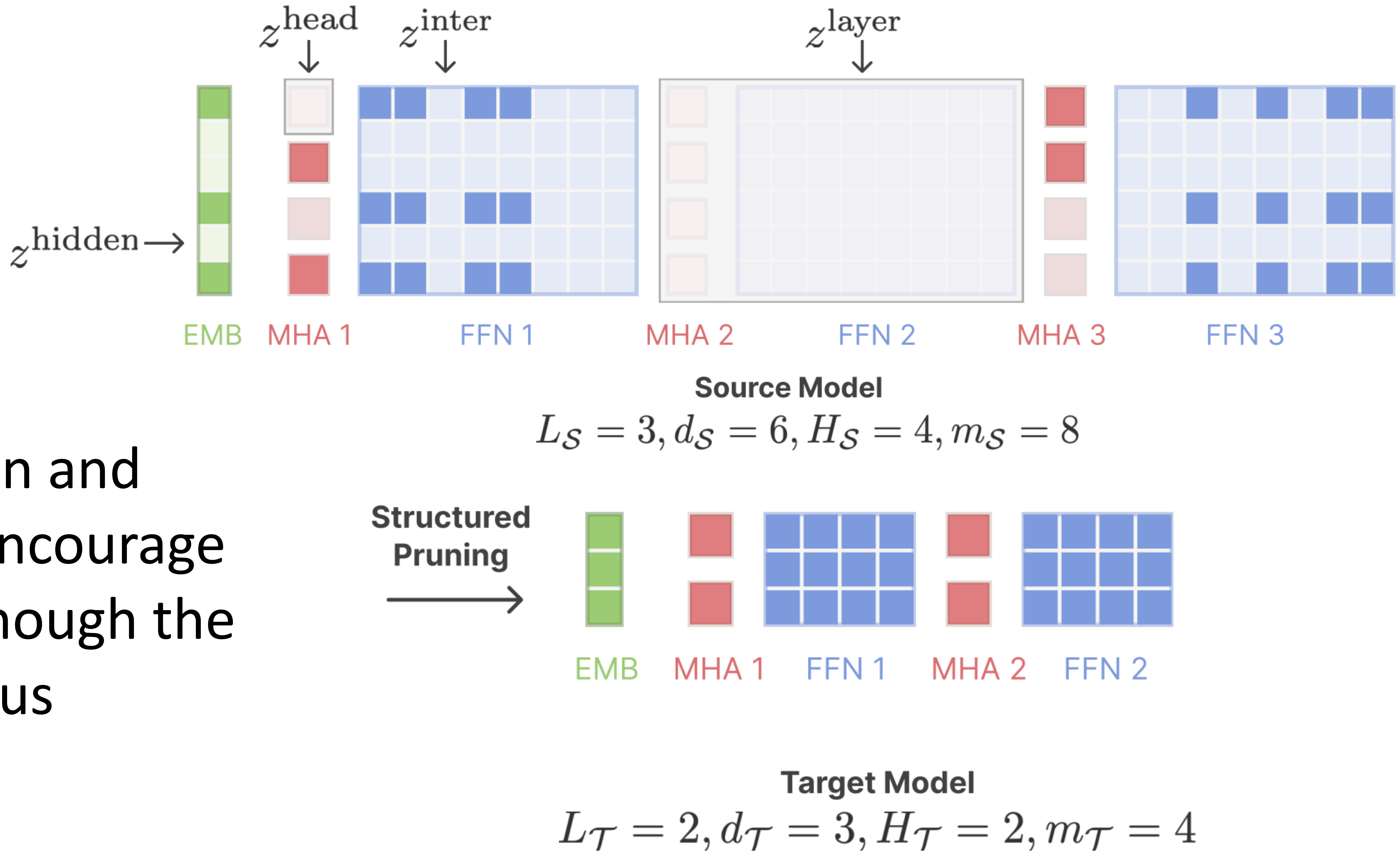
Approaches to Compression

- Pruning: can we reduce the number of neurons in the model?
  - ~~Basic idea: remove low-magnitude weights~~
  - Instead, we want some kind of structured pruning. What does this look like?
- Still a challenge: if different layers have different sizes, your GPU utilization may go down

Sheared Llama

- Idea 1: targeted structured pruning

- Parameterization and regularization encourage sparsity, even though the z's are continuous



Mengzhou Xia et al. (2023)

Sheared Llama

- Train for a while with the z's, then prune the network. Then enter stage 2: continued pre-training on new data

- Idea 2: dynamic batch loading. Update the weights controlling the mix of data you use during pre-training (sample more from domains of data with high loss)

Mengzhou Xia et al. (2023)

Sheared Llama

Model (#tokens for training)	Continued: LogiQA	Continued: BoolQ (32)	LM: LAMBADA	World Knowledge: NQ (32)	World Knowledge: MMLU (5)	Average
LLaMA2-7B (2T)†	30.7	82.1	28.8	73.9	46.6	64.6
OPT-1.3B (300B)†	**26.9**	57.5	58.0	6.9	24.7	48.2
Pythia-1.4B (300B)†	27.3	57.4	**61.6**	6.2	**25.7**	48.9
Sheared-LLaMA-1.3B (50B)	**26.9**	**64.0**	61.0	**9.6**	**25.7**	**51.0**
OPT-2.7B (300B)†	26.0	63.4	63.6	10.1	25.9	51.4
Pythia-2.8B (300B)†	28.0	66.0	64.7	9.0	26.9	52.5
INCITE-Base-3B (800B)	27.7	65.9	65.3	14.9	**27.0**	54.7
Open-LLaMA-3B-v1 (1T)	28.4	70.0	65.4	**18.6**	**27.0**	55.1
Open-LLaMA-3B-v2 (1T)†	28.1	69.6	66.5	17.1	26.9	55.7
Sheared-LLaMA-2.7B (50B)	**28.9**	**73.7**	**68.4**	16.5	26.4	**56.7**

- (Slightly) better than models that were “organically” trained at these larger scales

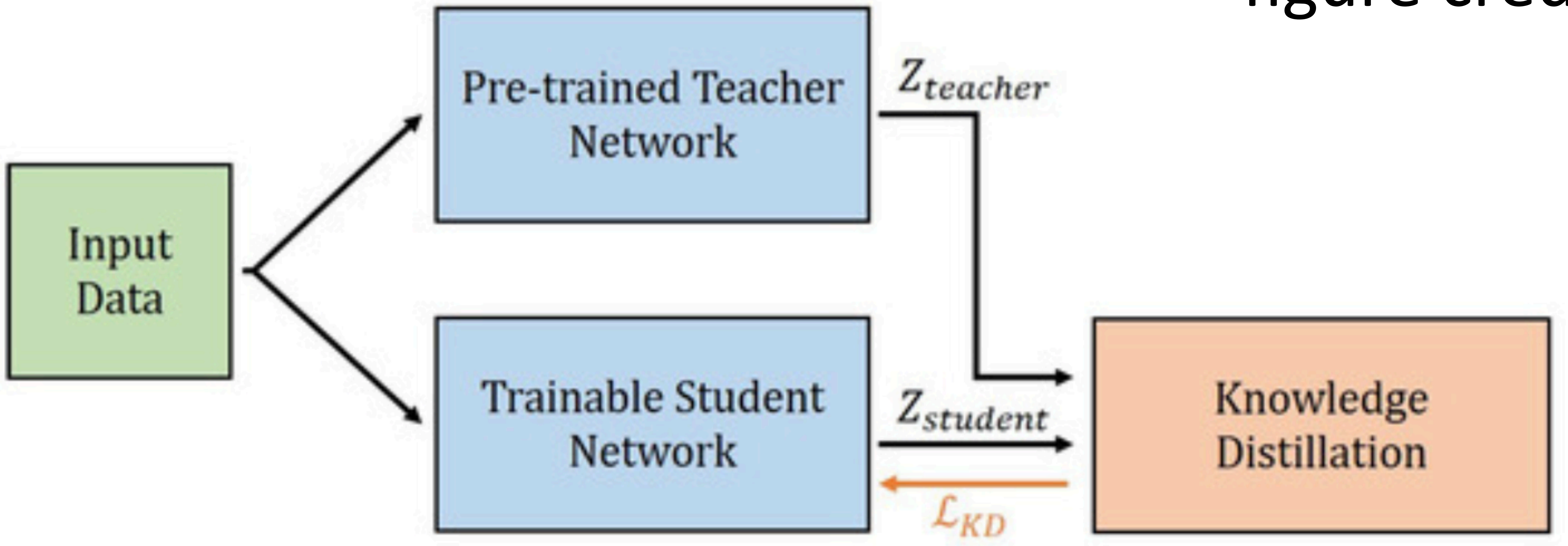
Mengzhou Xia et al. (2023)

Approaches to Compression

- Pruning: can we reduce the number of neurons in the model?
  - ~~Basic idea: remove low-magnitude weights~~
  - Instead, we want some kind of structured pruning. What does this look like?
- Knowledge distillation
  - Classic approach from Hinton et al.: train a *student* model to match distribution from *teacher*

DistilBERT

figure credit: Tianjian Li



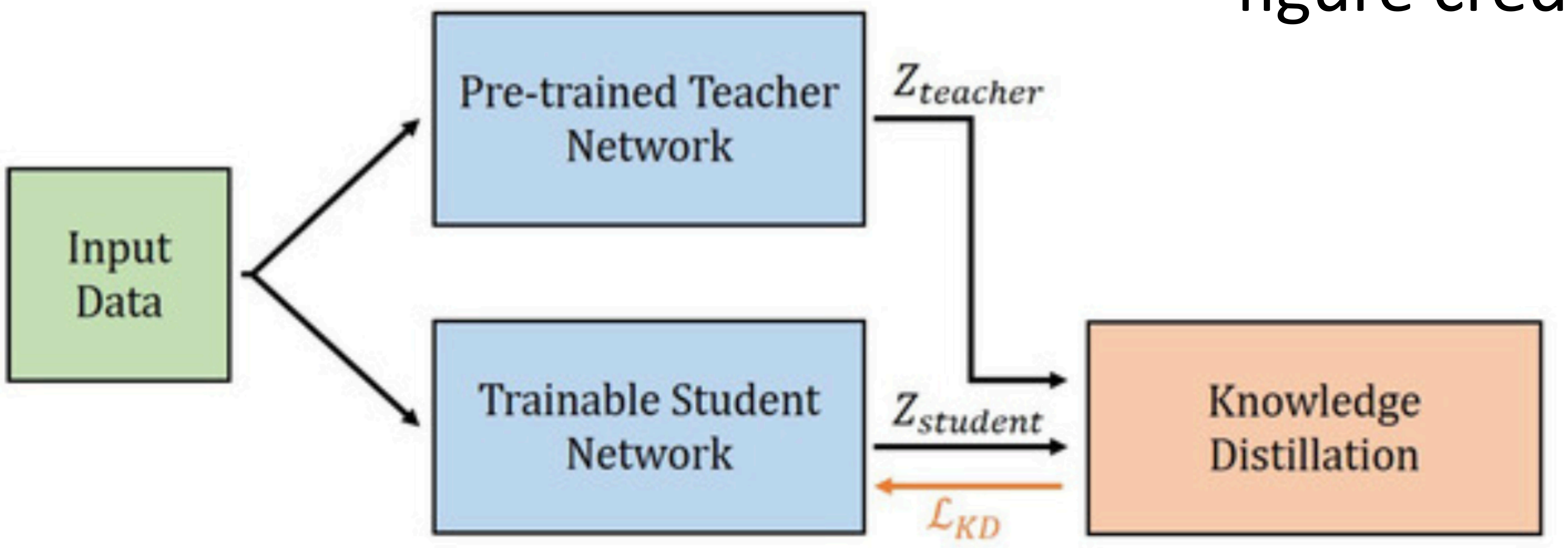
Suppose we have a classification model with output $P_{teacher}(y \mid \boldsymbol{x})$

Minimize KL($P_{teacher}$ || $P_{student}$) to bring student dist close to teacher

Note that this is not using labels — it uses the teacher to "pseudo-label" data, and we label an entire distribution, not just a top-one label

DistilBERT

figure credit: Tianjian Li



- Use a teacher model as a large neural network, such as BERT
- Make a small student model that is half the layers of BERT. Initialize with every other layer from the teacher

Sanh et al. (2019)

DistilBERT

Model	**Score**	CoLA	MNLI	MRPC	QNLI	QQP	RTE	SST-2	STS-B	WNLI
ELMo	68.7	44.1	68.6	76.6	71.1	86.2	53.4	91.5	70.4	56.3
BERT-base	79.5	56.3	86.7	88.6	91.8	89.6	69.3	92.7	89.0	53.5
DistilBERT	77.0	51.3	82.2	87.5	89.2	88.5	59.9	91.3	86.9	56.3

Table 2: **DistilBERT yields to comparable performance on downstream tasks.** Comparison on downstream tasks: IMDb (test accuracy) and SQuAD 1.1 (EM/F1 on dev set). D: with a second step of distillation during fine-tuning.

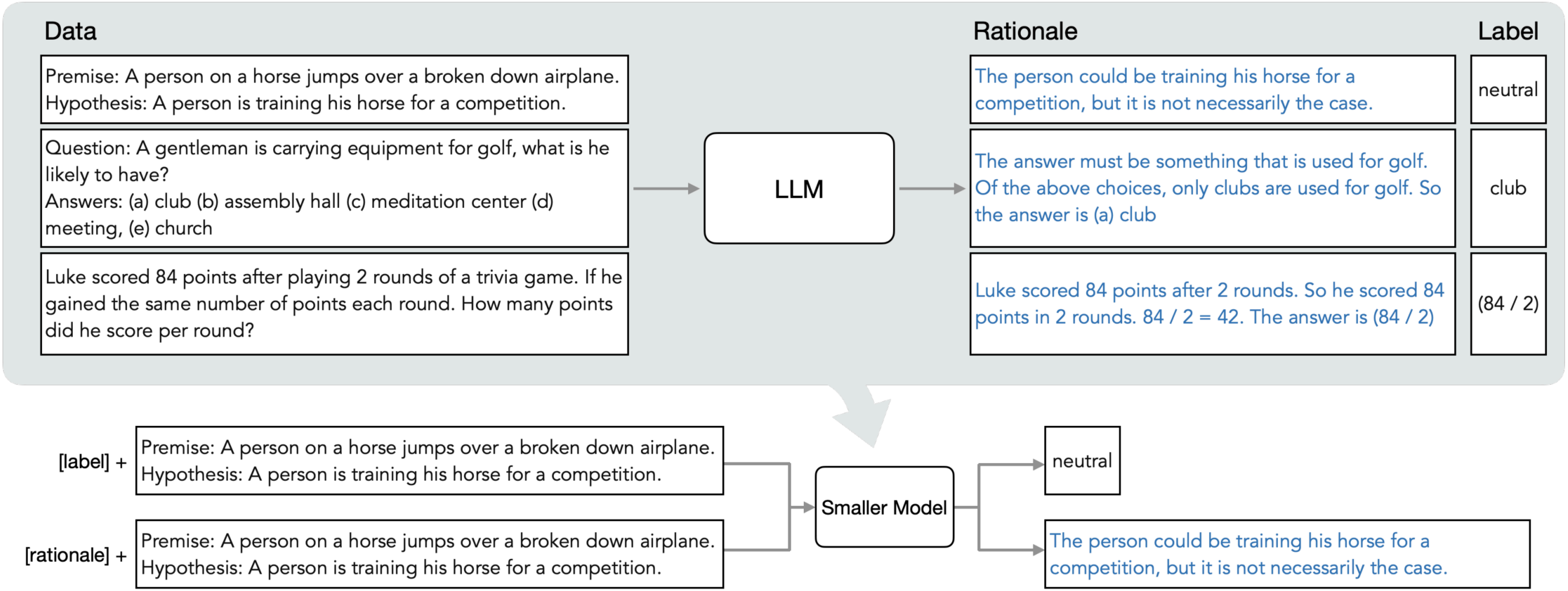
Model	IMDb (acc.)	SQuAD (EM/F1)
BERT-base	93.46	81.2/88.5
DistilBERT	92.82	77.7/85.8
DistilBERT (D)	-	79.1/86.9

Table 3: **DistilBERT is significantly smaller while being constantly faster.** Inference time of a full pass of GLUE task STS-B (sentiment analysis) on CPU with a batch size of 1.

Model	# param. (Millions)	Inf. time (seconds)
ELMo	180	895
BERT-base	110	668
DistilBERT	66	410

Sanh et al. (2019)

Other Distillation



- How to distill models for complex reasoning settings? Still an open problem!

Cheng-Yu Hsieh et al. (2023)

Parameter-Efficient Tuning

Parameter-Efficient Tuning

- Rather than train all model parameters at once, can we get away with just training a small number of them?

- What are the advantages of this?

- Typical advantages: lower memory, easier to serve many models for use cases like personalization or multitasking

- Not an advantage: faster (it's not)

BitFit

$$\mathbf{Q}^{m,\ell}(\mathbf{x}) = \mathbf{W}_q^{m,\ell}\mathbf{x} + \mathbf{b}_q^{m,\ell}$$
$$\mathbf{K}^{m,\ell}(\mathbf{x}) = \mathbf{W}_k^{m,\ell}\mathbf{x} + \mathbf{b}_k^{m,\ell}$$
$$\mathbf{V}^{m,\ell}(\mathbf{x}) = \mathbf{W}_v^{m,\ell}\mathbf{x} + \mathbf{b}_v^{m,\ell}$$

$$\mathbf{h}_1^\ell = att\big(\mathbf{Q}^{1,\ell}, \mathbf{K}^{1,\ell}, \mathbf{V}^{1,\ell}, .., \mathbf{Q}^{m,\ell}, \mathbf{K}^{m,\ell}, \mathbf{V}^{m,l}\big)$$

and then fed to an MLP with layer-norm (LN):

$$\mathbf{h}_2^\ell = \mathrm{Dropout}\big(\mathbf{W}_{m_1}^\ell \cdot \mathbf{h}_1^\ell \;+\; \mathbf{b}_{m_1}^\ell\big) \quad (1)$$
$$\mathbf{h}_3^\ell = \mathbf{g}_{LN_1}^\ell \odot \frac{(\mathbf{h}_2^\ell + \mathbf{x}) - \mu}{\sigma} + \mathbf{b}_{LN_1}^\ell \quad (2)$$
$$\mathbf{h}_4^\ell = \mathrm{GELU}\big(\mathbf{W}_{m_2}^\ell \cdot \mathbf{h}_3^\ell \;+\; \mathbf{b}_{m_2}^\ell\big) \quad (3)$$
$$\mathbf{h}_5^\ell = \mathrm{Dropout}\big(\mathbf{W}_{m_3}^\ell \cdot \mathbf{h}_4^\ell \;+\; \mathbf{b}_{m_3}^\ell\big) \quad (4)$$
$$\mathrm{out}^\ell = \mathbf{g}_{LN_2}^\ell \odot \frac{(\mathbf{h}_5^\ell + \mathbf{h}_3^\ell) - \mu}{\sigma} + \mathbf{b}_{LN_2}^\ell \quad (5)$$

- Tune only the bias terms of the Transformer architecture, don't fine-tune the weights
- How many parameters do you think this is?

Zaken et al. (2022)

BitFit

		%Param	QNLI	SST-2	$MNLI_m$	$MNLI_{mm}$	…	Avg.
	Train size		105k	67k	393k	393k		
(V)	Full-FT†	100%	**93.5**	**94.1**	**86.5**	**87.1**		**84.8**
(V)	Full-FT	100%	91.7±0.1	93.4±0.2	85.5±0.4	85.7±0.4		84.1
(V)	Diff-Prune†	0.5%	**93.4**	**94.2**	**86.4**	**86.9**		**84.6**
(V)	BitFit	0.08%	91.4±2.4	93.2±0.4	84.4±0.2	84.8±0.1		84.2
(T)	Full-FT‡	100%	91.1	**94.9**	86.7	85.9		**81.8**
(T)	Full-FT†	100%	**93.4**	94.1	86.7	**86.0**		81.5
(T)	Adapters‡	3.6%	90.7	94.0	84.9	85.1		81.1
(T)	Diff-Prune†	0.5%	**93.3**	94.1	**86.4**	**86.0**		**81.5**
(T)	BitFit	0.08%	92.0	**94.2**	84.5	84.8		80.9

- Degraded performance, but only train <0.1% of the parameters of the full model!

Zaken et al. (2022)

LoRA

- Alternative: learn weight matrices as (W + BA), where BA is a product of two low-rank matrices.
  - If we have a d x d matrix and we use a rank reduction of size r, what is the parameter reduction from LoRA?
- Allows adding low-rank matrix on top of existing high-rank model
- Unlike some other methods, LoRA can be "compiled down" into the model (just add BA into W)

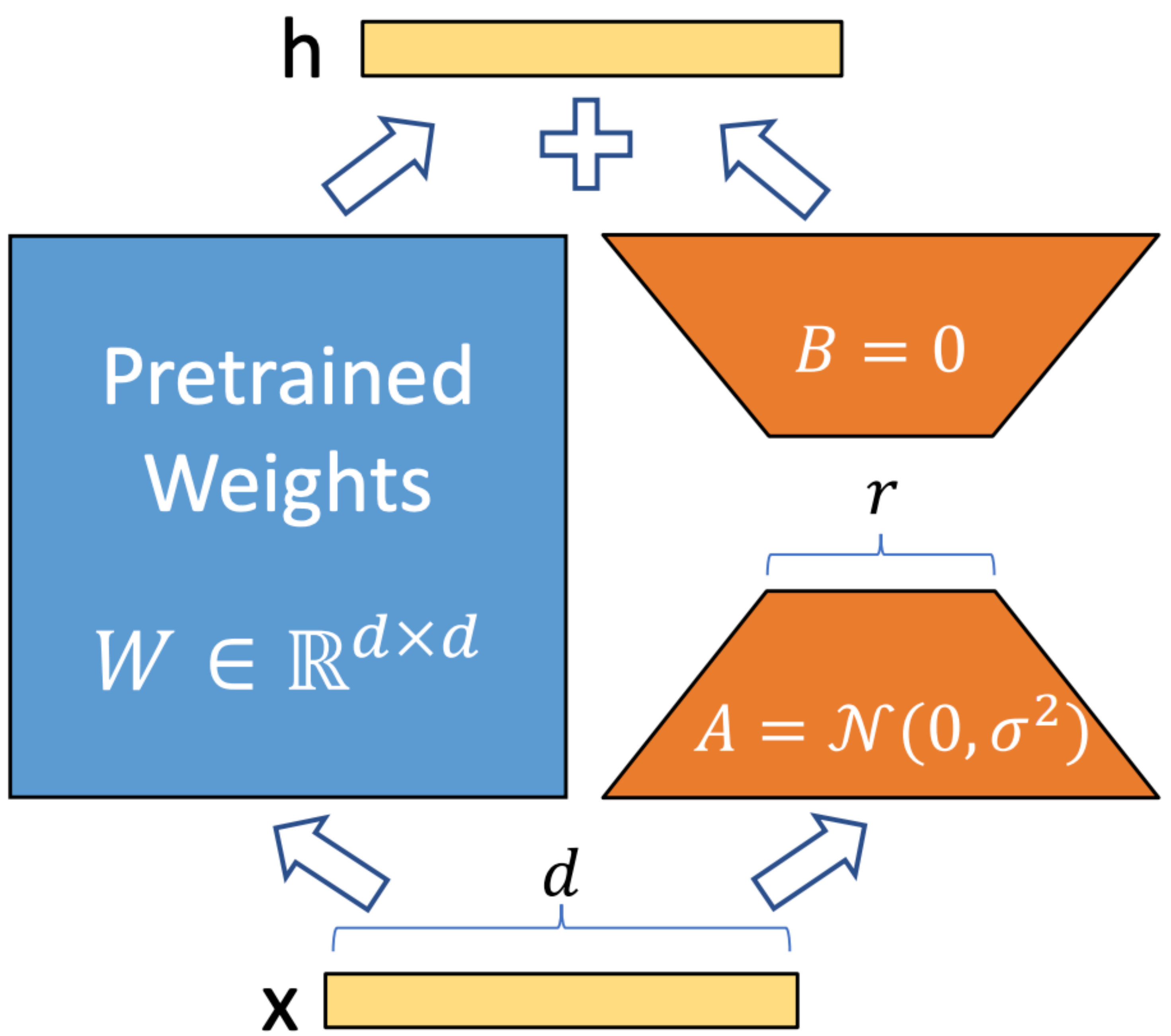


Figure 1: Our reparametrization. We only train A and B.

Hu et al. (2021)

LoRA

Model & Method	# Trainable Parameters	MNLI	SST-2	MRPC	CoLA	QNLI	QQP	RTE	STS-B	Avg.
RoB_{base} (FT)*	125.0M	**87.6**	94.8	90.2	**63.6**	92.8	**91.9**	78.7	91.2	86.4
RoB_{base} (BitFit)*	0.1M	84.7	93.7	**92.7**	62.0	91.8	84.0	81.5	90.8	85.2
RoB_{base} (Adpt^{D})*	0.3M	$87.1_{\pm.0}$	$94.2_{\pm.1}$	$88.5_{\pm1.1}$	$60.8_{\pm.4}$	$93.1_{\pm.1}$	$90.2_{\pm.0}$	$71.5_{\pm2.7}$	$89.7_{\pm.3}$	84.4
RoB_{base} (Adpt^{D})*	0.9M	$87.3_{\pm.1}$	$94.7_{\pm.3}$	$88.4_{\pm.1}$	$62.6_{\pm.9}$	$93.0_{\pm.2}$	$90.6_{\pm.0}$	$75.9_{\pm2.2}$	$90.3_{\pm.1}$	85.4
RoB_{base} (LoRA)	0.3M	$87.5_{\pm.3}$	$\mathbf{95.1}_{\pm.2}$	$89.7_{\pm.7}$	$63.4_{\pm1.2}$	$\mathbf{93.3}_{\pm.3}$	$90.8_{\pm.1}$	$\mathbf{86.6}_{\pm.7}$	$\mathbf{91.5}_{\pm.2}$	**87.2**
$\text{RoB}_{\text{large}}$ (FT)*	355.0M	90.2	**96.4**	**90.9**	68.0	94.7	**92.2**	86.6	92.4	88.9
$\text{RoB}_{\text{large}}$ (LoRA)	0.8M	$\mathbf{90.6}_{\pm.2}$	$96.2_{\pm.5}$	$\mathbf{90.9}_{\pm1.2}$	$\mathbf{68.2}_{\pm1.9}$	$\mathbf{94.9}_{\pm.3}$	$91.6_{\pm.1}$	$\mathbf{87.4}_{\pm2.5}$	$\mathbf{92.6}_{\pm.2}$	**89.0**

- LoRA is much better than BitFit, even better than vanilla fine-tuning on GLUE!

Hu et al. (2021)

LLM Quantization

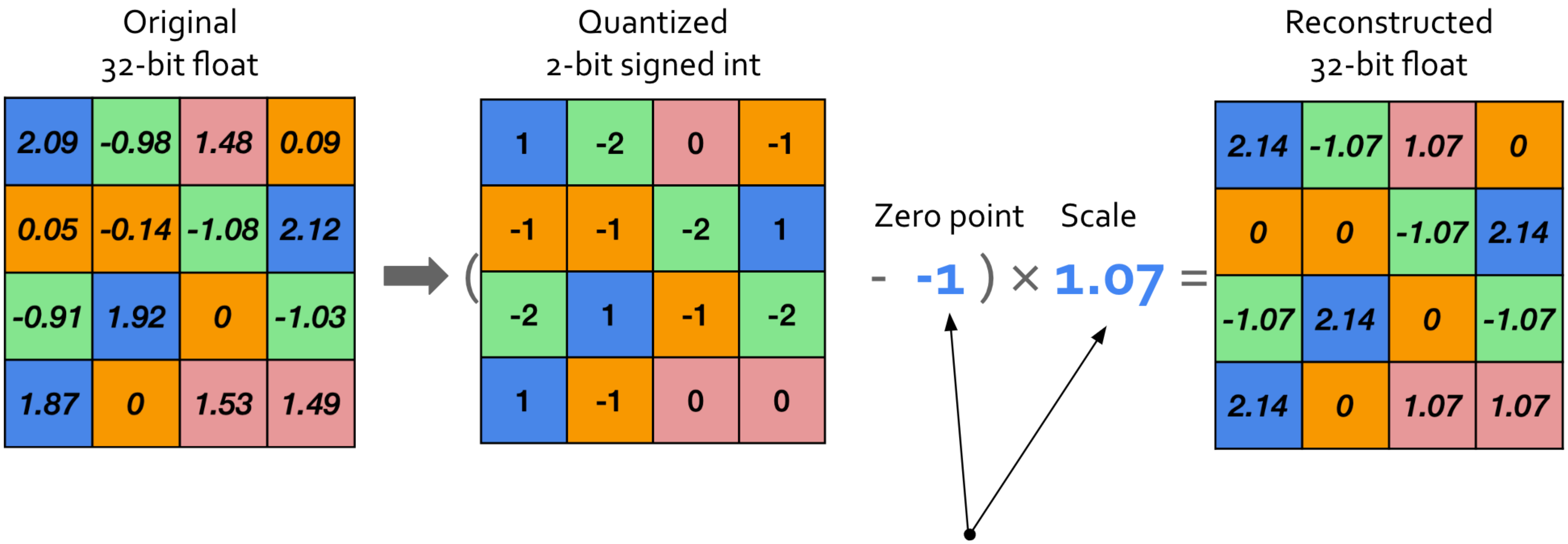
- A significant fraction of LLM training is just storing the weights
  - Normal floating-point precision: 4 bytes per weight, gets large for 10B+ parameter models!
- How much is needed for fine-tuning?
  - The Adam optimizer has to store at least 2 additional values for each parameter (first- and second-moment estimates)
  - Memory gets very large! Can we reduce this?

LLM Quantization

		Exponent	Fraction
IEEE 754 Single Precision 32-bit Float (FP32)		8	23
IEEE 754 Half Precision 16-bit Float (FP16)		5	10
Google Brain Float (BF 16)		8	7
Nvidia FP8 (E4M3)		4	3

slide credit: Tianjian Li

LLM Quantization

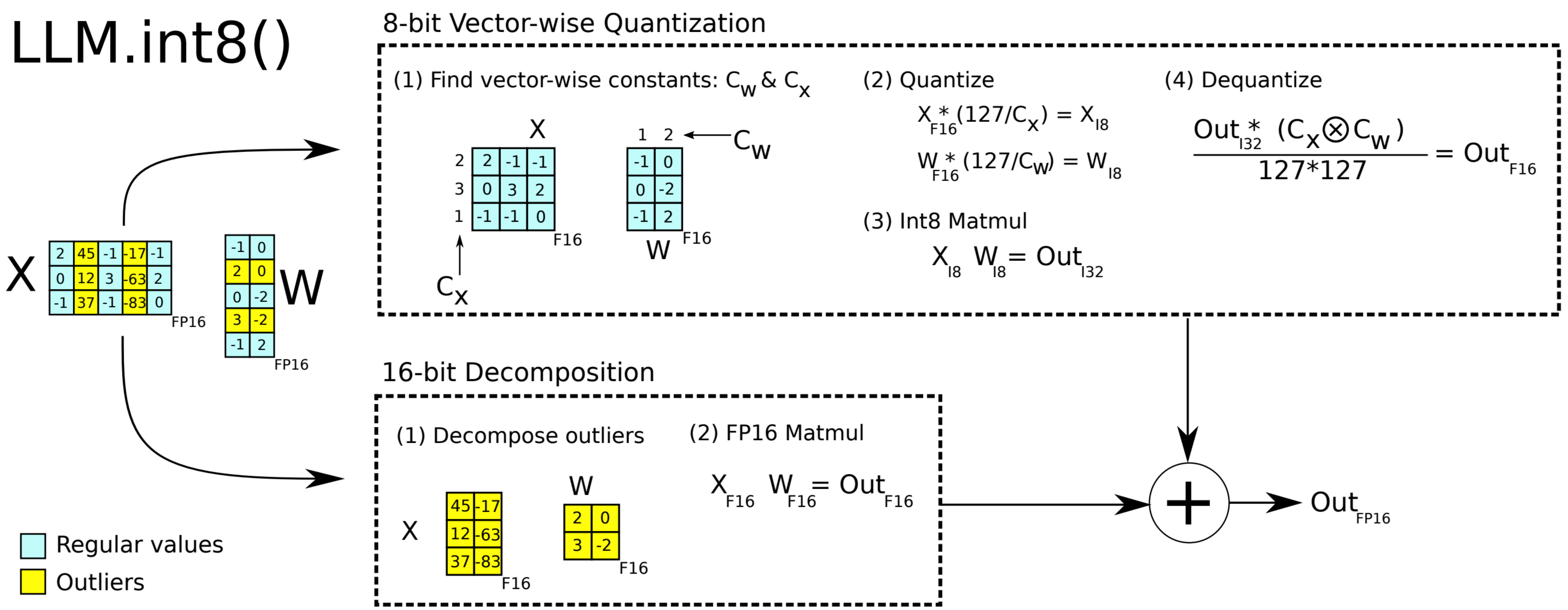


- Outlier weights can make it hard to find a good zero point/scale

slide credit: Tianjian Li

LLM Quantization

LLM.int8()



- Solution: combine 8-bit and 16-bit quantization, where most stuff is 8-bit quantized

Dettmers et al. (2022)

LLM Quantization

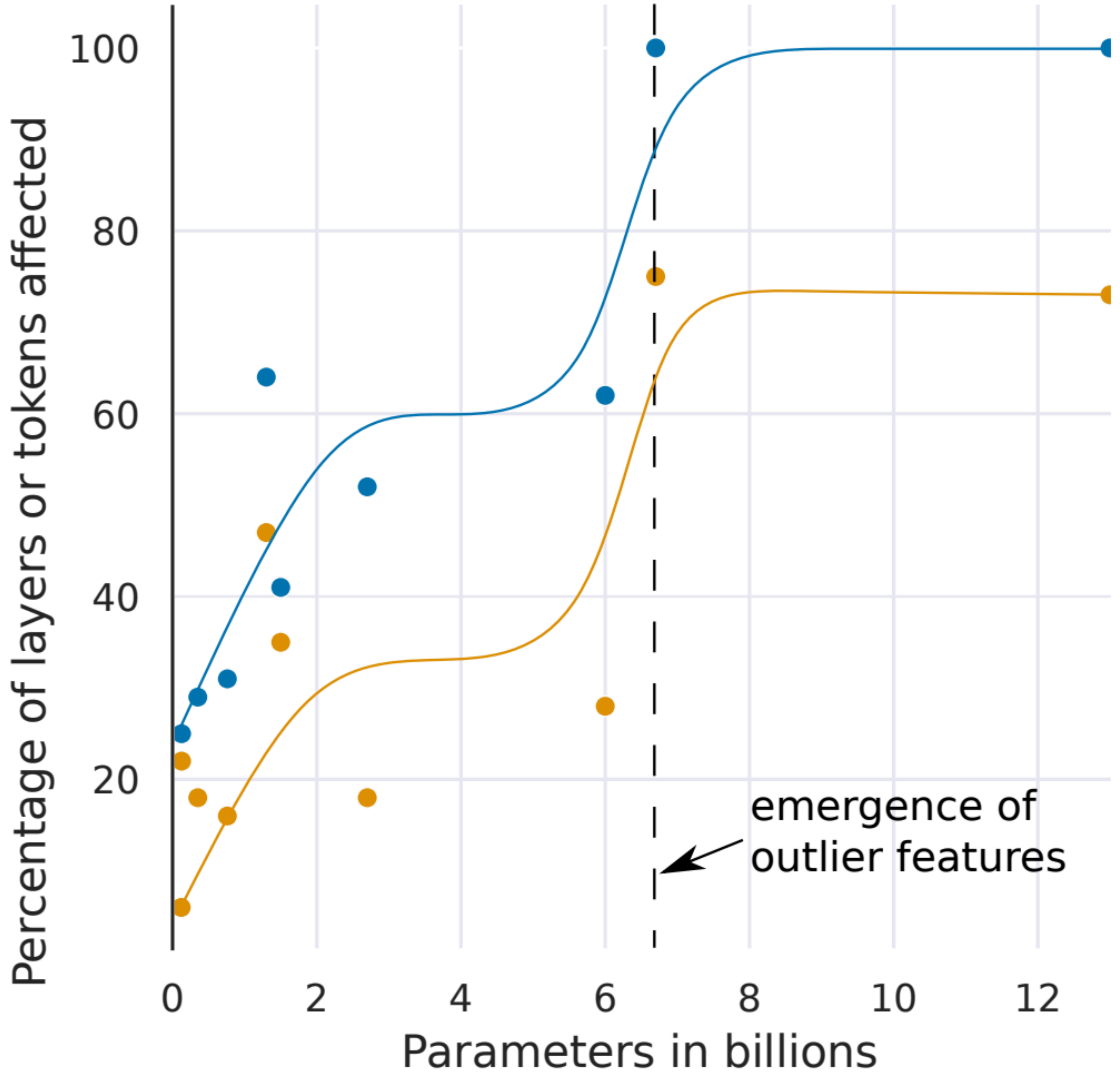
Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

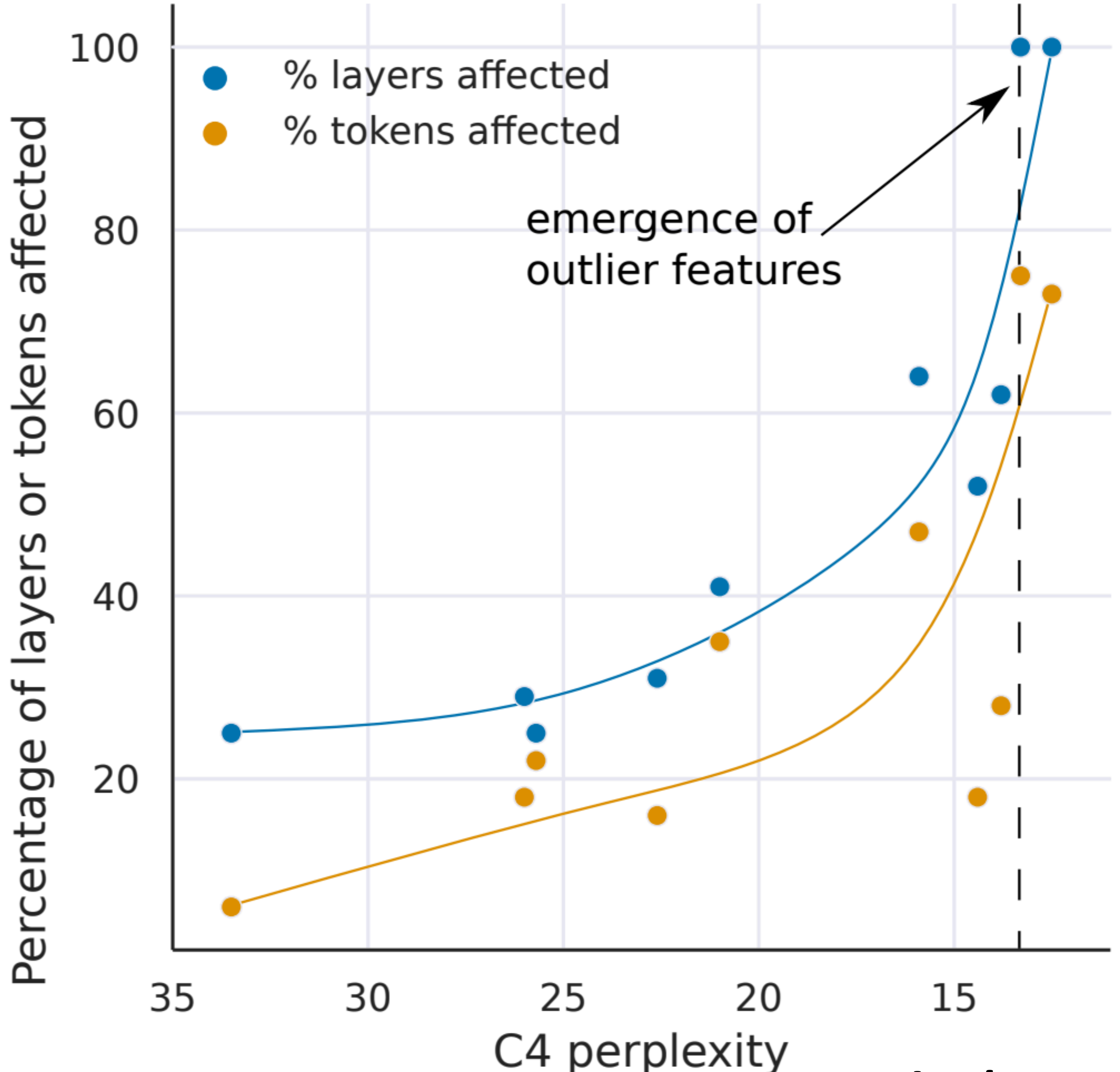
- Validation perplexity on language modeling. Prior Int8 techniques degrade, the decomposition maintains performance

Dettmers et al. (2022)

LLM Quantization

- Interestingly, the outlier features that require 16-bit quantization emerge at large scale

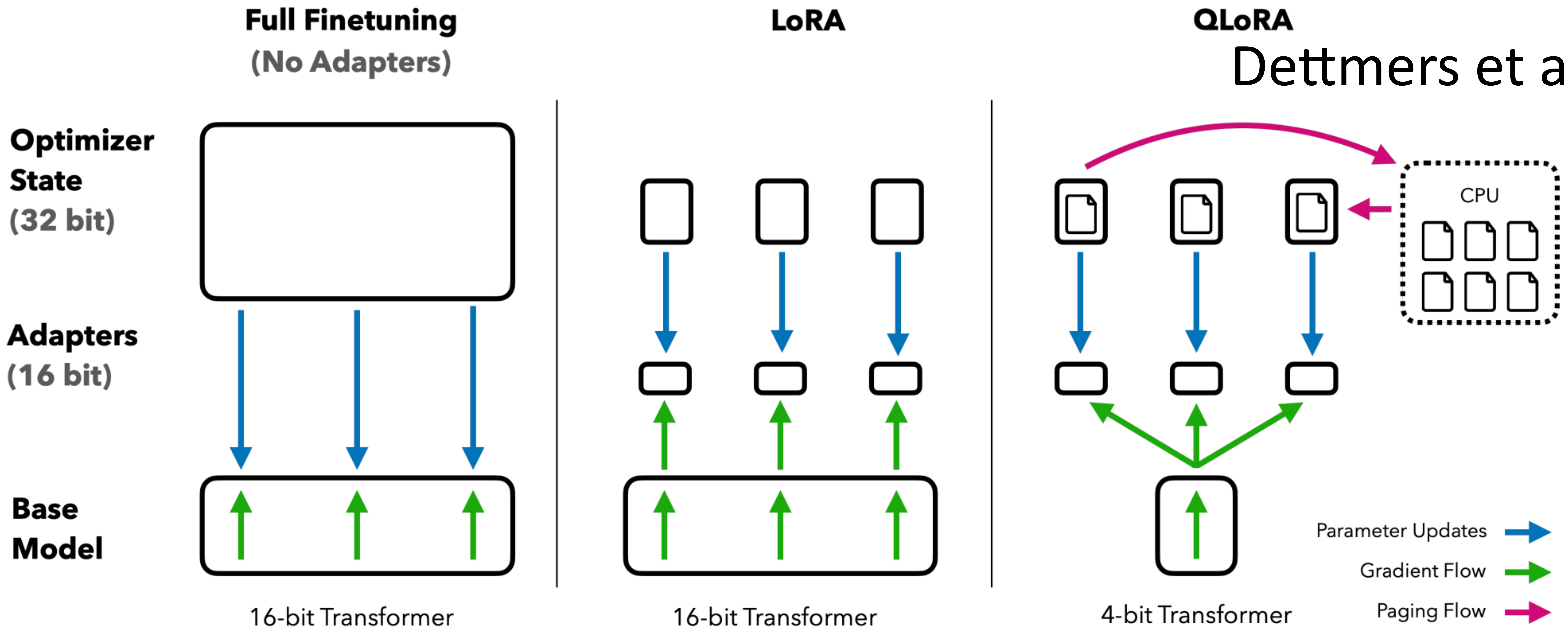




Dettmers et al. (2022)

QLoRA: Memory-efficient training

Dettmers et al. (2023)



- 4-bit "normal float", takes advantage of the fact that NN weights typically have a zero-centered normal distribution
- Paged optimizer state to avoid memory spikes (due to training examples with long sequence length)

Where is this going?

- **Better GPU programming:** as GPU performance starts to saturate, we'll probably see more algorithms tailored very specifically to the affordances of the hardware

- **Small models**, either distilled or trained from scratch: as LLMs gets better, we can do with ~7B scale what used to be only doable with ChatGPT (GPT-3.5)

- **Continued focus on faster inference**: faster inference can be highly impactful across all LLM applications

Takeaways

- Decoding optimizations: speculative decoding gives a fast way to exactly sample from a smaller model. Also techniques like Flash Attention

- Model optimizations to make models smaller: pruning, distillation

- Model compression and quantization: standard compression techniques, but adapted to work really well for GPUs