Natural Language Processing

Episode 10-ish

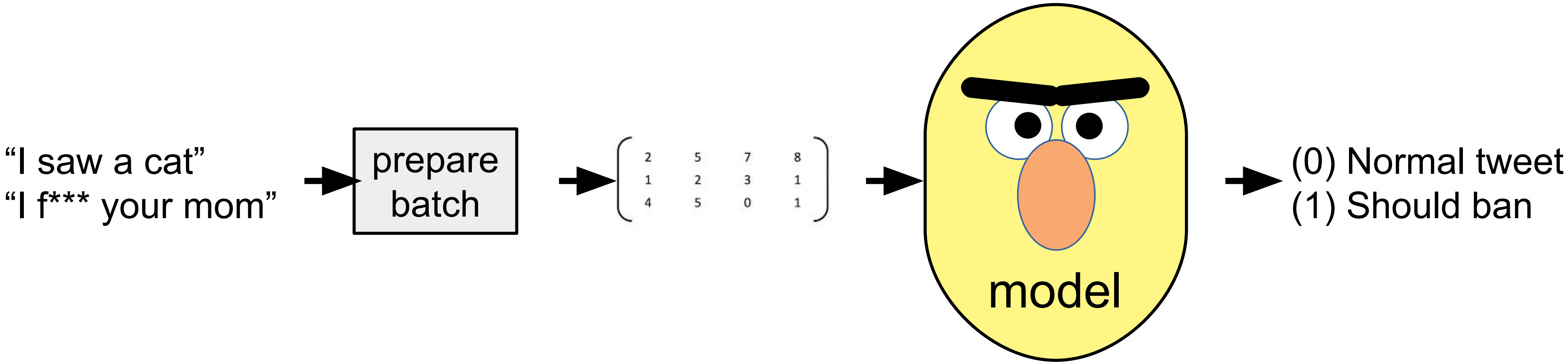
Model Compression & Acceleration

Institute of Science and Technology Austria

by Andrei Panferov

Chapter 1: Why Should You Care?

Case Study: Text Classification



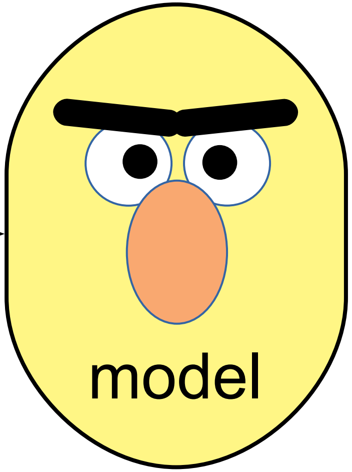
Case Study: Text Classification

"I saw a cat"
"I f*** your mom"

→ prepare batch →

$$\begin{pmatrix} 2 & 5 & 7 & 8 \\ 1 & 2 & 3 & 1 \\ 4 & 5 & 0 & 1 \end{pmatrix}$$

→

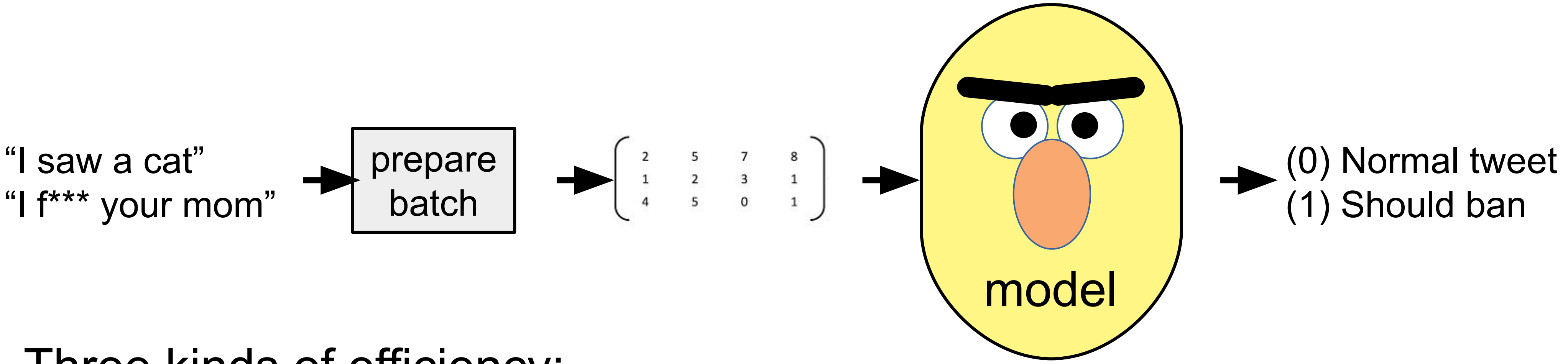


→ (0) Normal tweet
(1) Should ban

Three kinds of efficiency:

Model **S**ize
(mega)bytes

Case Study: Text Classification

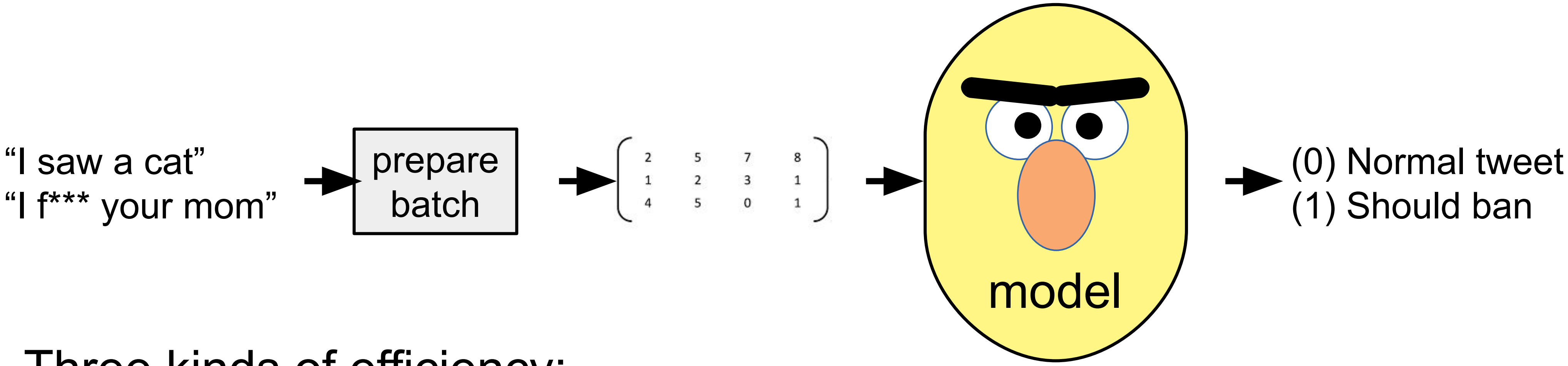


Three kinds of efficiency:

Model Size
(mega)bytes

Throughput
samples/second

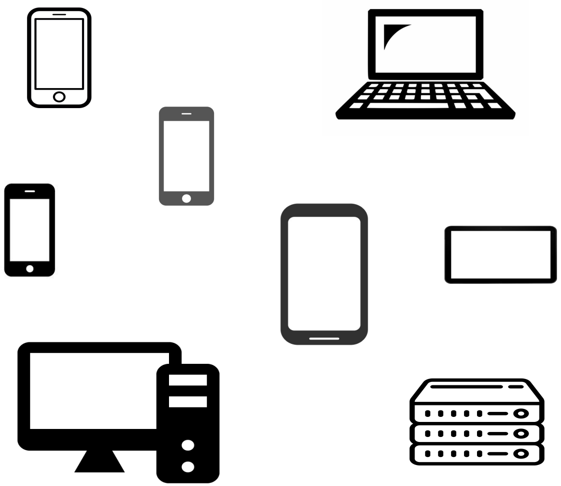
Case Study: Text Classification



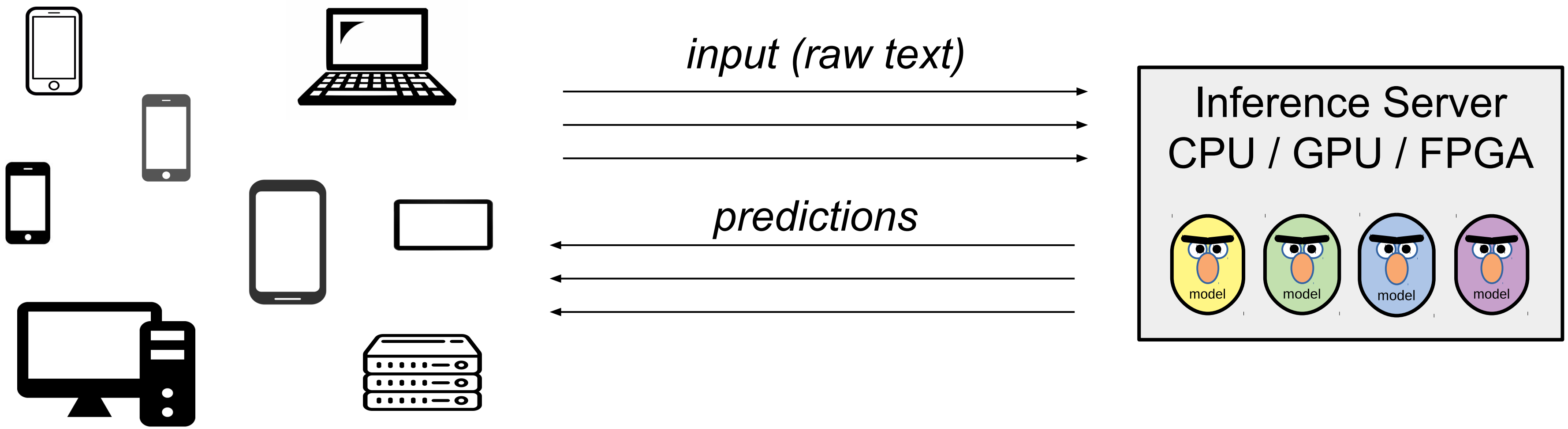
Three kinds of efficiency:

- **Model Size** (mega)bytes
- **Throughput** samples/second
- **Latency** ms@percentile

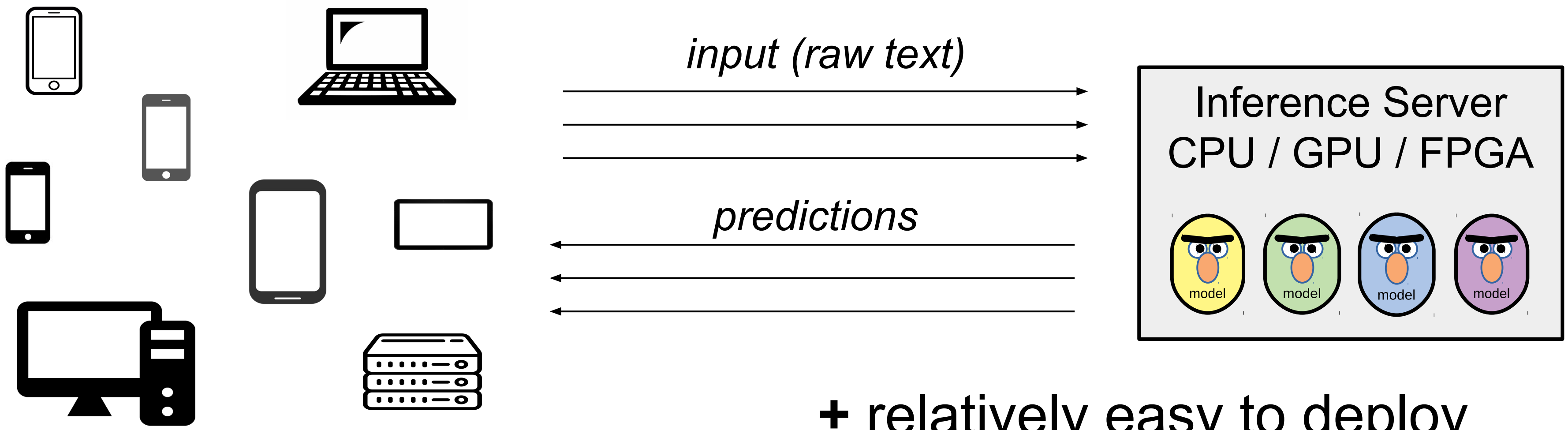
Scenario 1: Inference Server



Scenario 1: Inference Server

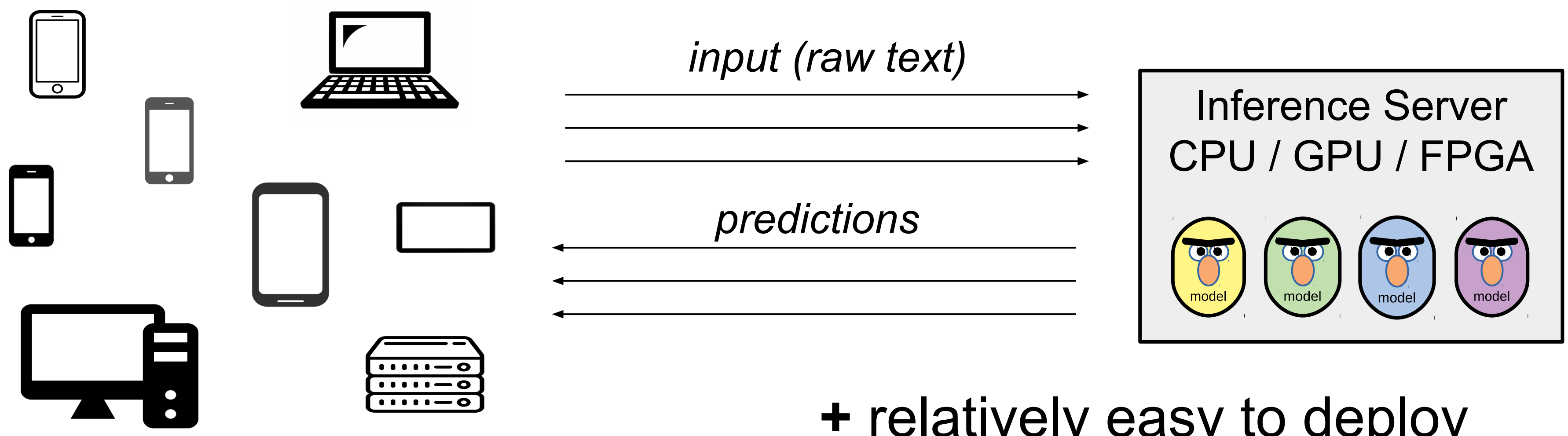


Scenario 1: Inference Server



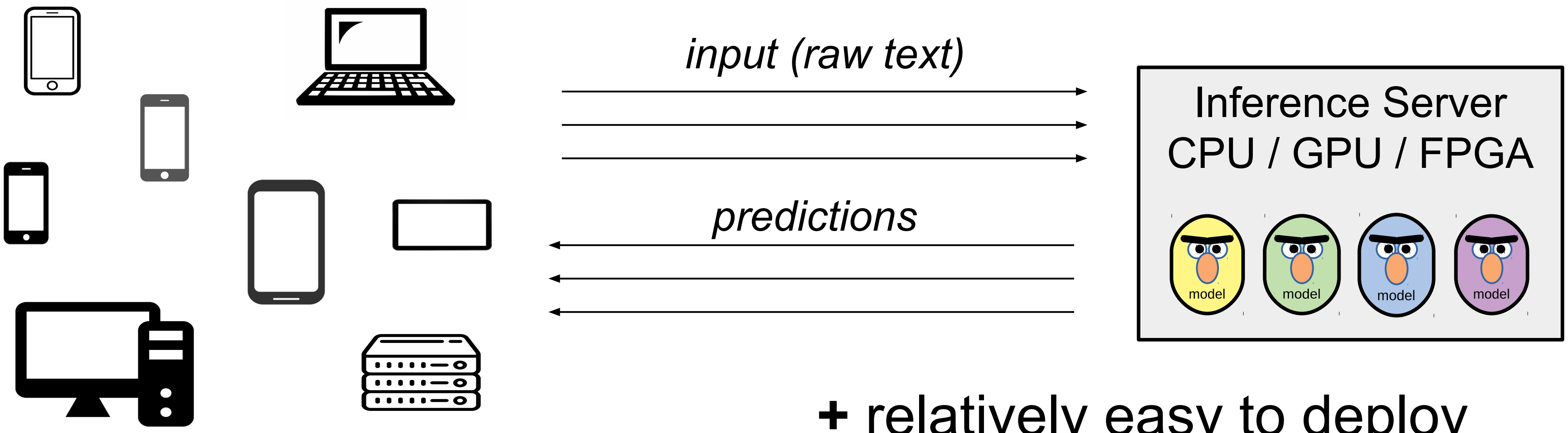
+ relatively easy to deploy
+ you control model & inference
+ clients don’t run compute

Scenario 1: Inference Server



+ relatively easy to deploy
+ you control model & inference
+ clients don’t run compute
– you pay for each inference
– clients can’t work offline
– network latency

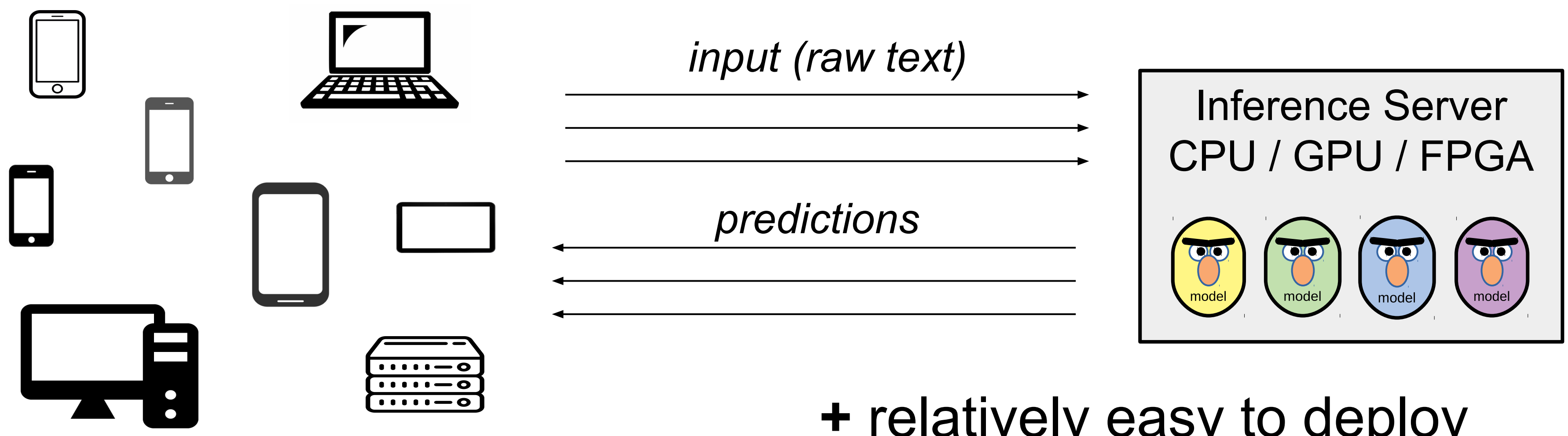
Scenario 1: Inference Server



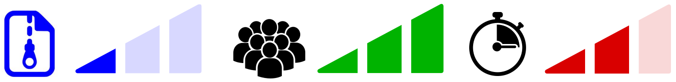
+ relatively easy to deploy
+ you control model & inference
+ clients don't run compute
– you pay for each inference
– clients can't work offline
– network latency

Which is the most important?

Scenario 1: Inference Server



Priorities:



+ relatively easy to deploy
+ you control model & inference
+ clients don’t run compute
– you pay for each inference
– clients can’t work offline
– network latency

Scenario 1: Inference Server

- Group inputs into batches (e.g. by length)
  - *improves throughput at the cost of latency*
- Multiple servers with load balancing

Scenario 1: Inference Server

- Group inputs into batches (e.g. by length)
  - *improves throughput at the cost of latency*

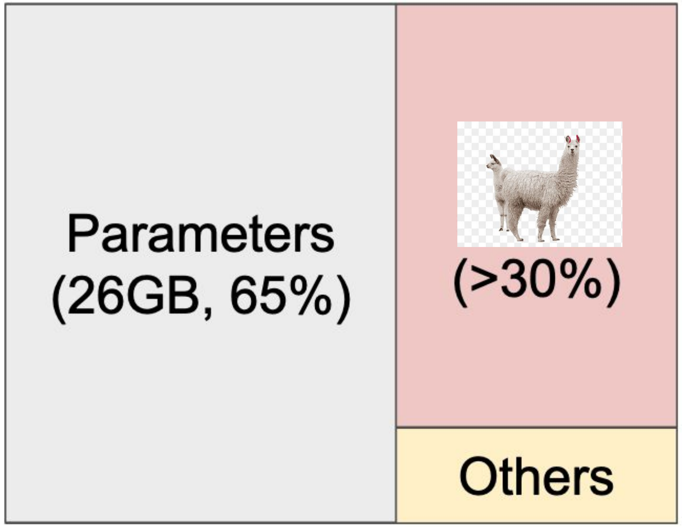
- Multiple servers with load balancing

  improves throughput at the cost of your budget :)

Popular frameworks:	priorities
vLLM	efficiency ≪ developer time
Triton Inference Server	efficiency ≈ developer time
 Custom model-dependent code	efficiency ≫ developer time

Scenario 1: Inference Server

Question: what did behind the Llama?



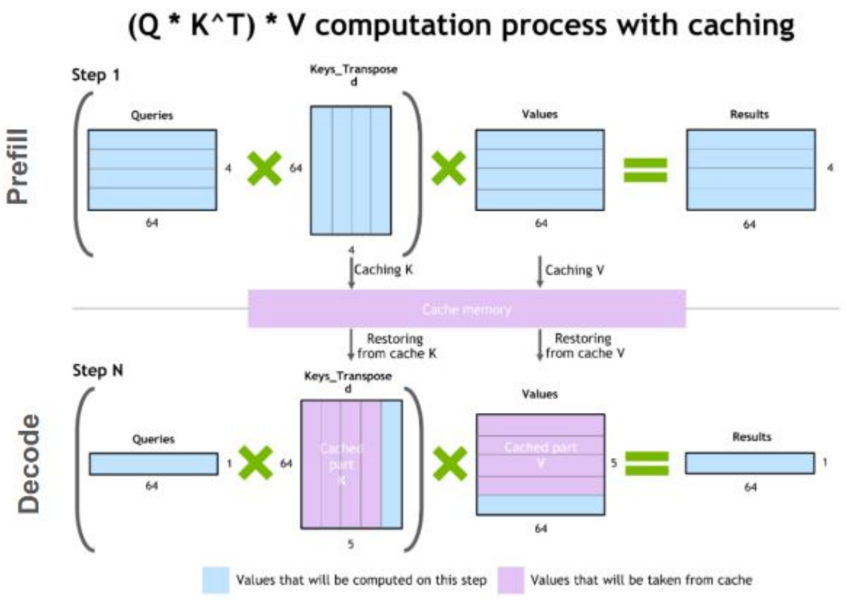
NVIDIA A100 40GB

Image: arxiv.org/pdf/2309.06180

Scenario 1: Inference Server

Question: what did behind the Llama?

Answer: KV Cache



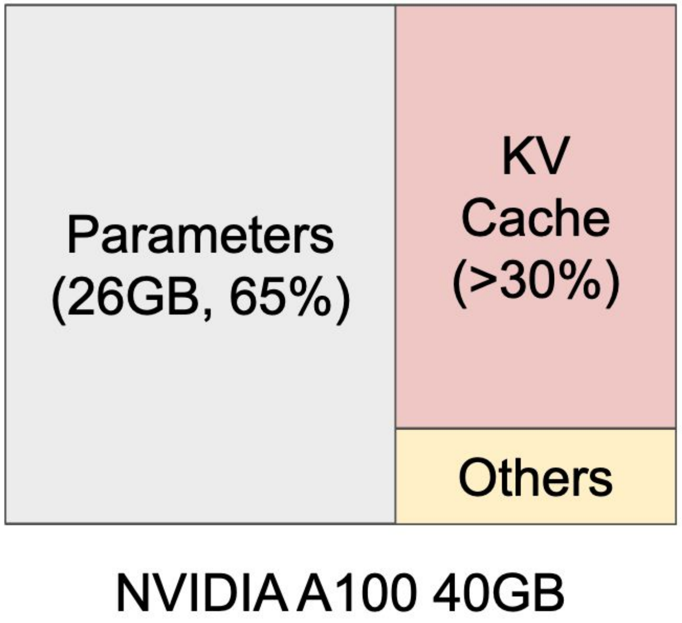
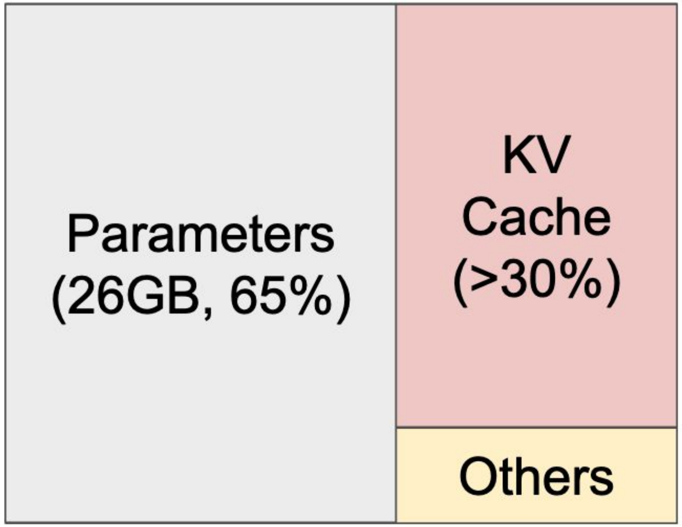


Image: arxiv.org/pdf/2309.06180

Scenario 1: Inference Server

Question: how many parallel requests we can serve if we have 20Gb of extra DRAM and one token takes up 200kb of cache?

Answer: it depends…



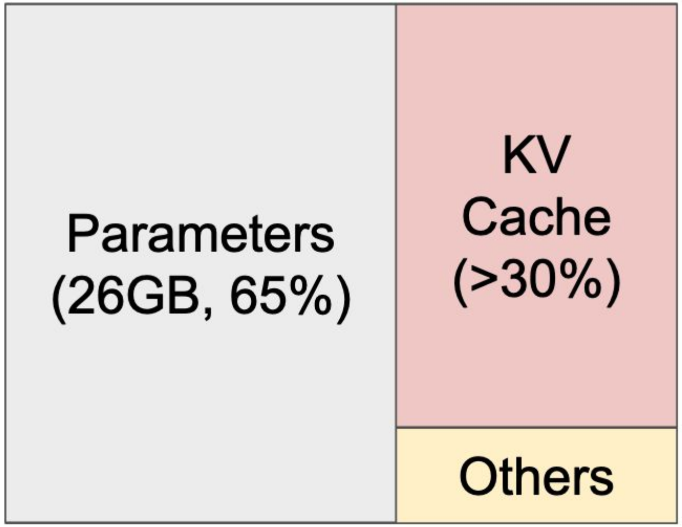
NVIDIA A100 40GB

Image: arxiv.org/pdf/2309.06180

Scenario 1: Inference Server

Question: how many parallel requests we can serve if we have 20Gb of extra DRAM and one token takes up 200kb of cache?
Answer: it depends…

What if requests were very diverse:
- 2000 prompt tok, 1 gen tok
- 2000 prompt tok, 2000 gen tok
- 50 prompt tok, 1 gen tok



NVIDIA A100 40GB

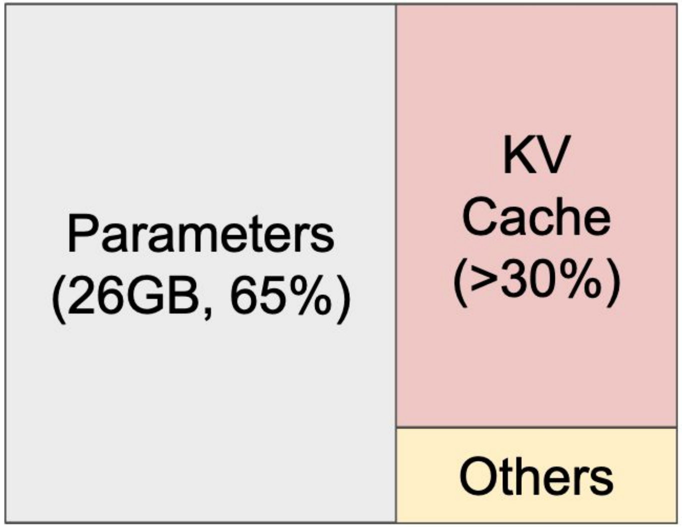
Image: arxiv.org/pdf/2309.06180

Scenario 1: Inference Server

Question: how many parallel requests we can serve if we have 20Gb of extra DRAM and one token takes up 200kb of cache?
Answer: it depends…

What if requests were very diverse:

- 2000 prompt tok, 1 gen tok
- 2000 prompt tok, 2000 gen tok
- 50 prompt tok, 1 gen tok



NVIDIA A100 40GB

Image: arxiv.org/pdf/2309.06180

Scenario 1: Inference Server

Question: how many parallel requests we can serve if we have 20Gb of extra DRAM and one token takes up 200kb of cache?
Answer: ~100000 tokens with Paged Attention

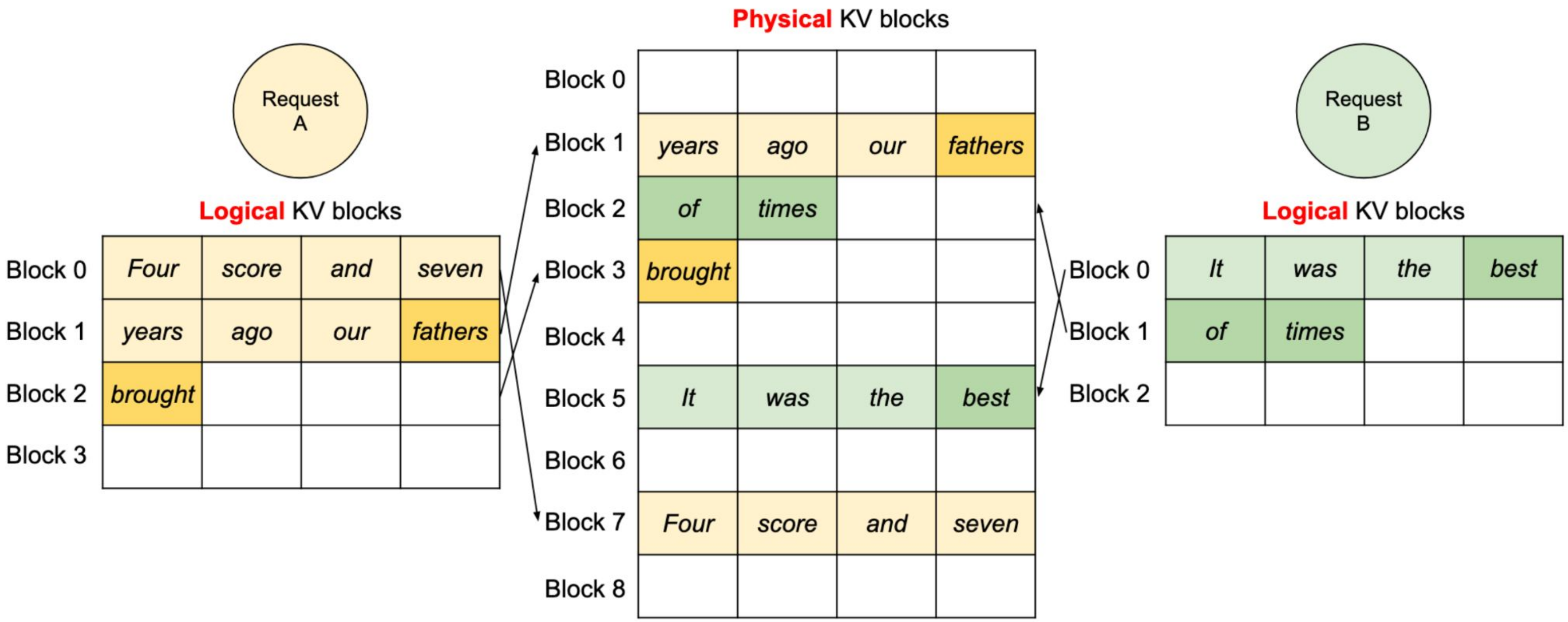


Image: arxiv.org/pdf/2309.06180

Scenario 1: Inference Server

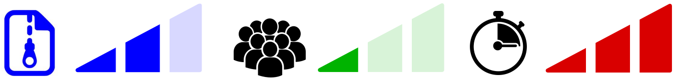
TLDR:

- Prioritizing throughput
- Many concurrent users with diverse requests
- Smart KV-Cache management is a must

Scenario 2: Workstation Deployment

- Preload model onto a dedicated device, infer locally
- Features:
  - Data privacy
- Requirements:
  - Predictable hardware
- LLM frameworks:
  - llama.cpp
  - ollama
  - exllama-v2

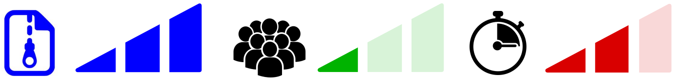
Priorities:



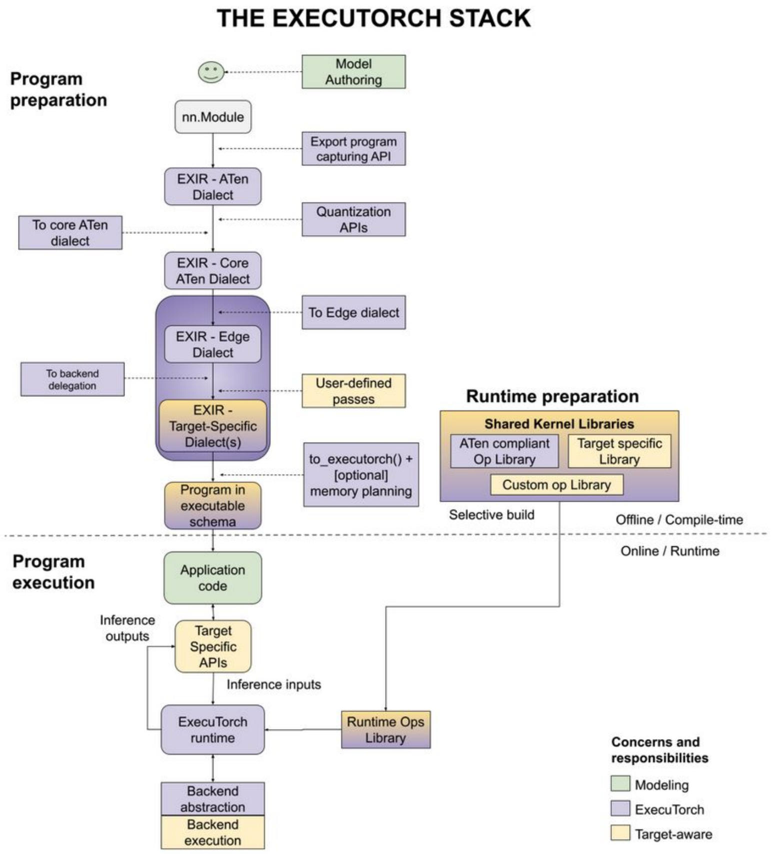
Scenario 3: Mobile Deployment

- Run model on a mobile device
- Features:
    - Diverse devices (or not if you’re Apple)
    - New priority: *power consumption*
    - Specialized matmul kernels

Priorities:



Scenario 3: Mobile Deployment



Same model…

- Tracing the model
- Compiling

Different backends

- Mapping to kernels

Scenario 3: Mobile Deployment

TensorFlow.js	All modern browsers
CoreML	iOS devices
NNAPI	Android devices
ExecuTorch	iOS/Android/Embedded
MLX	Apple Silicon

Scenario 3: Mobile Deployment

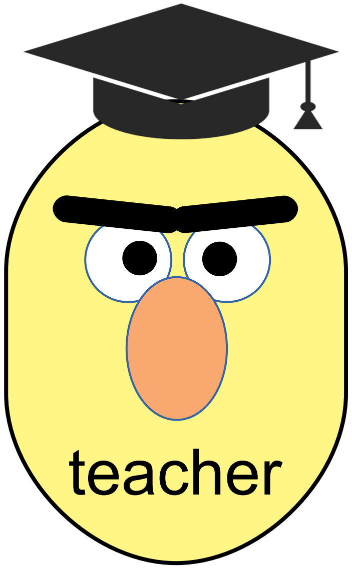
TLDR:
- Often limited by RAM/Latency
- Sensitive to power consumption
- Specialized/unique/diverse hardware

Chapter 2: How Do I Improve My Model?

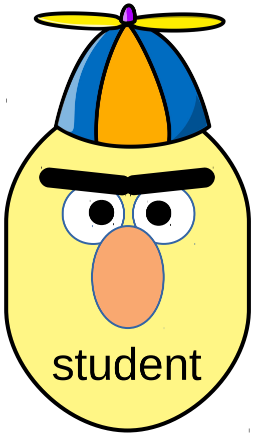
Compression by Distillation

Distillation…
Heard that word before?

Compression by Distillation

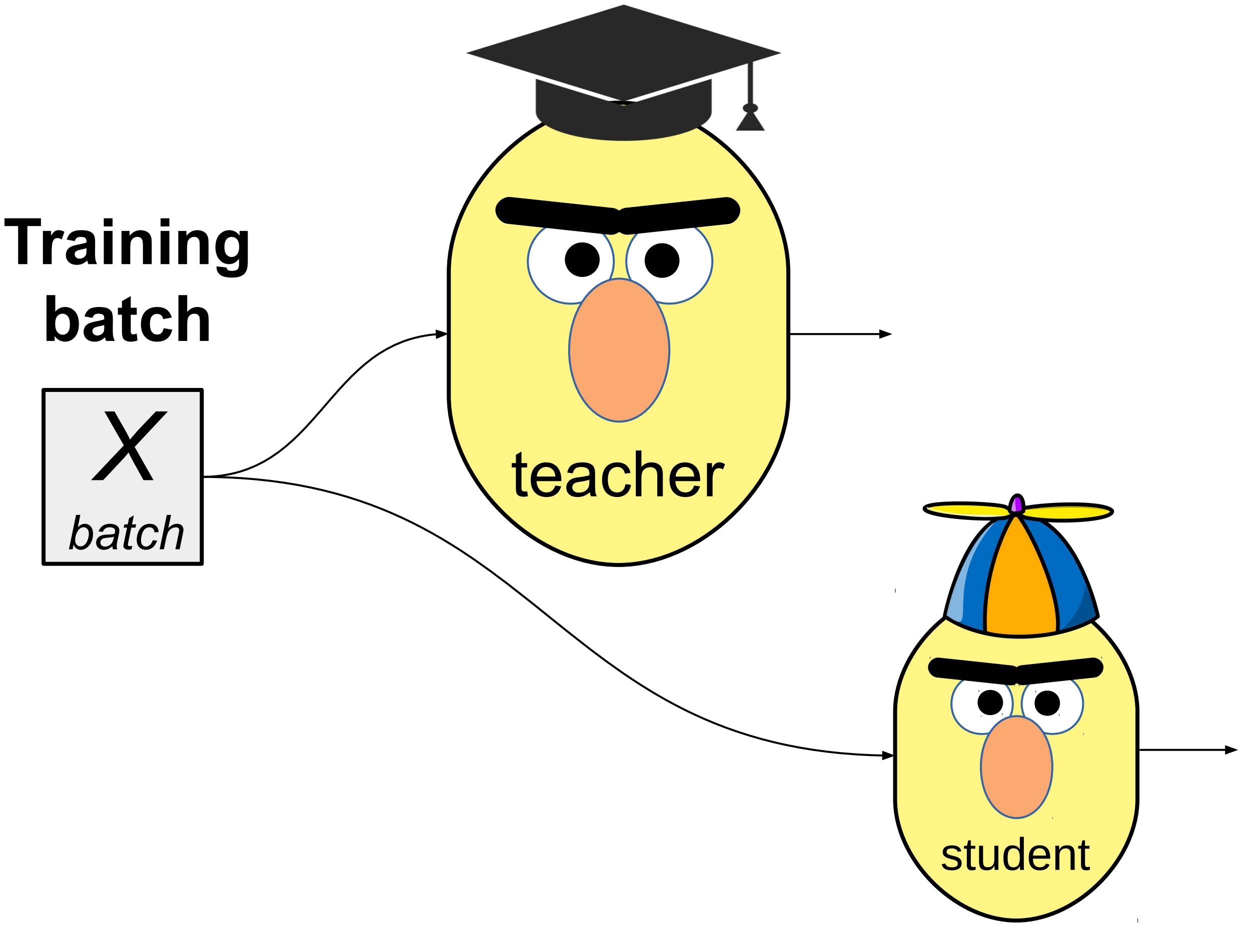


First, get the best performing model regardless of size

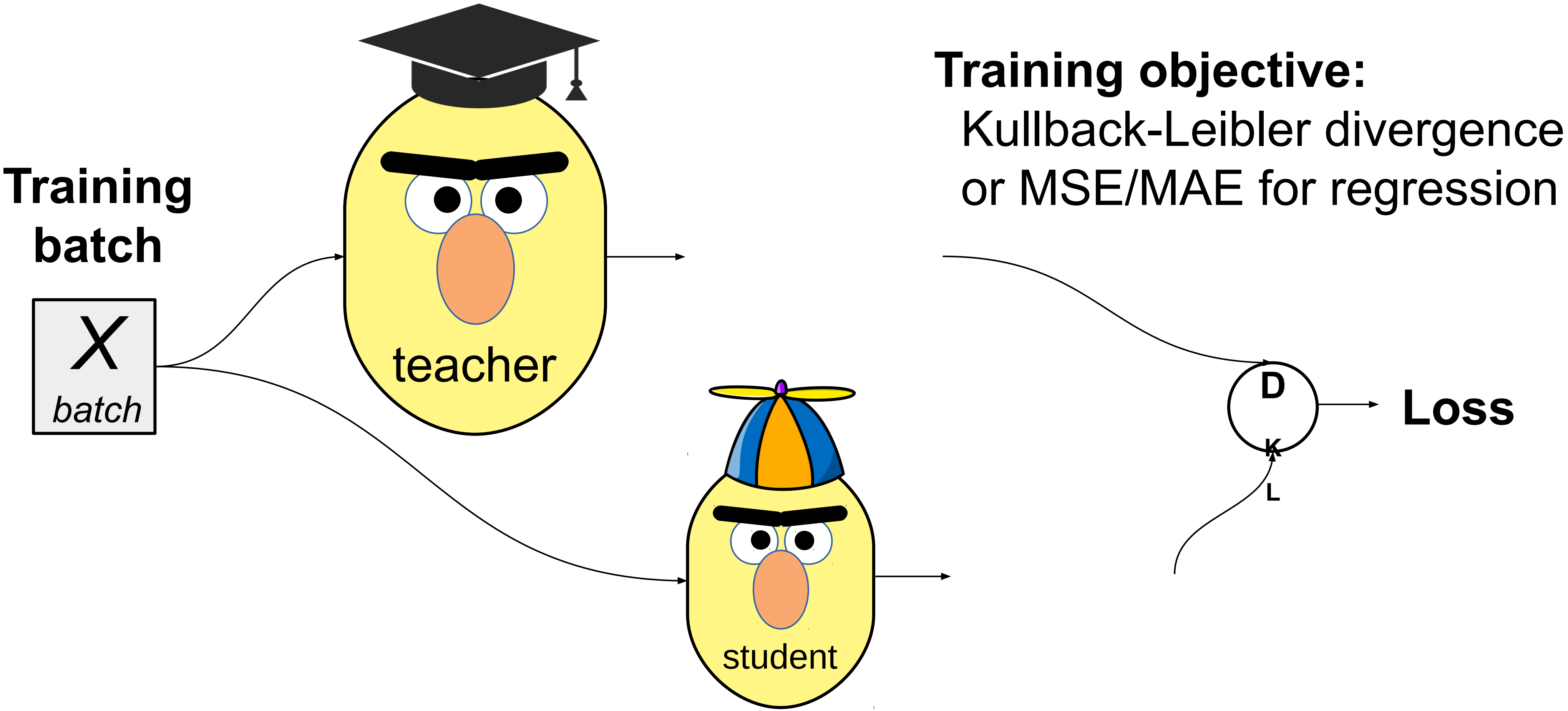


Then, train a more compact model to approximate it!

Compression by Distillation

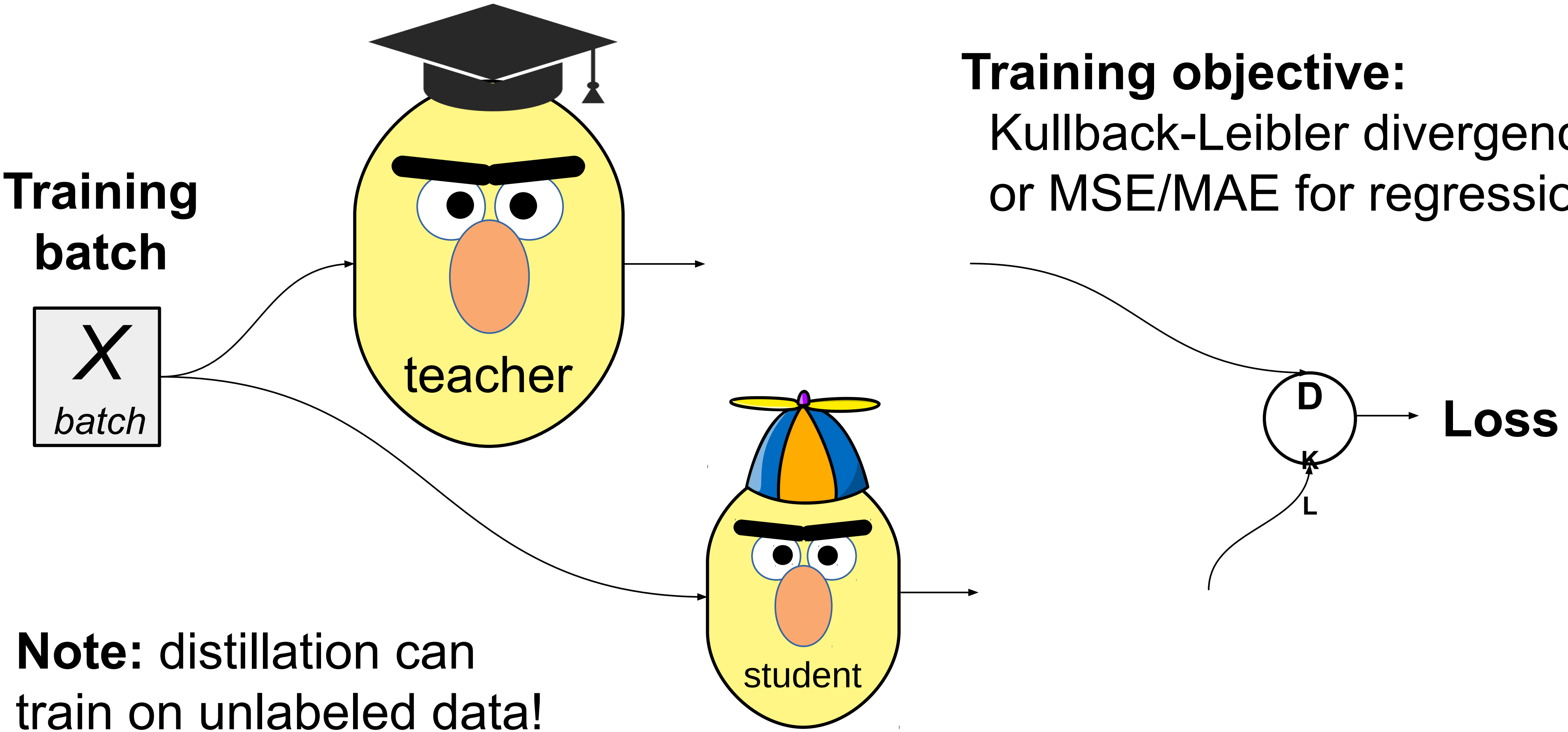


Compression by Distillation



Compression by Distillation

Training objective:
Kullback-Leibler divergence
or MSE/MAE for regression



Note: distillation can
train on unlabeled data!

Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
  - e.g. DistillBERT: https://arxiv.org/pdf/1910.01108.pdf
  - Same as BERT-base, but with
  - *half as many layers*
  - (and ≈1.5 times faster)

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

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

Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units, random init
- **Sparse along layers/attn_heads/mlp_neurons:** sparsify but obtain dense representation, init from teacher
  Let’s focus on that…

Minitron Approach

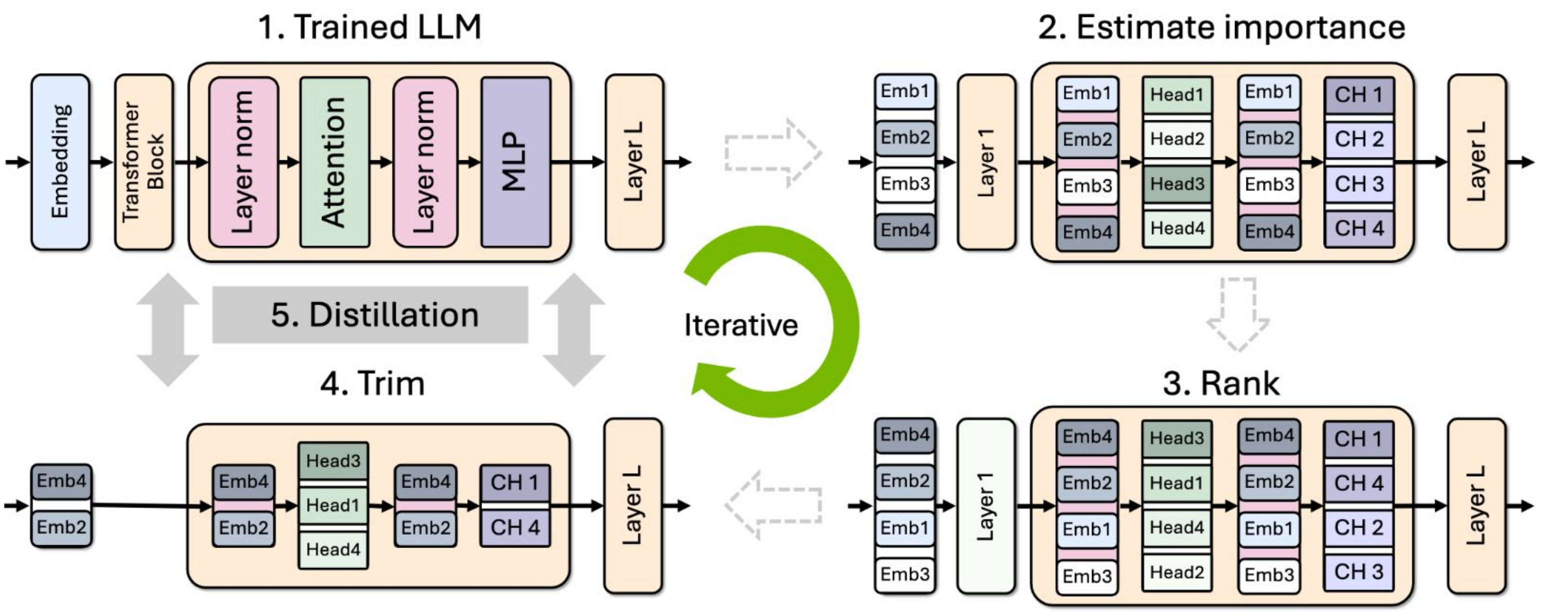
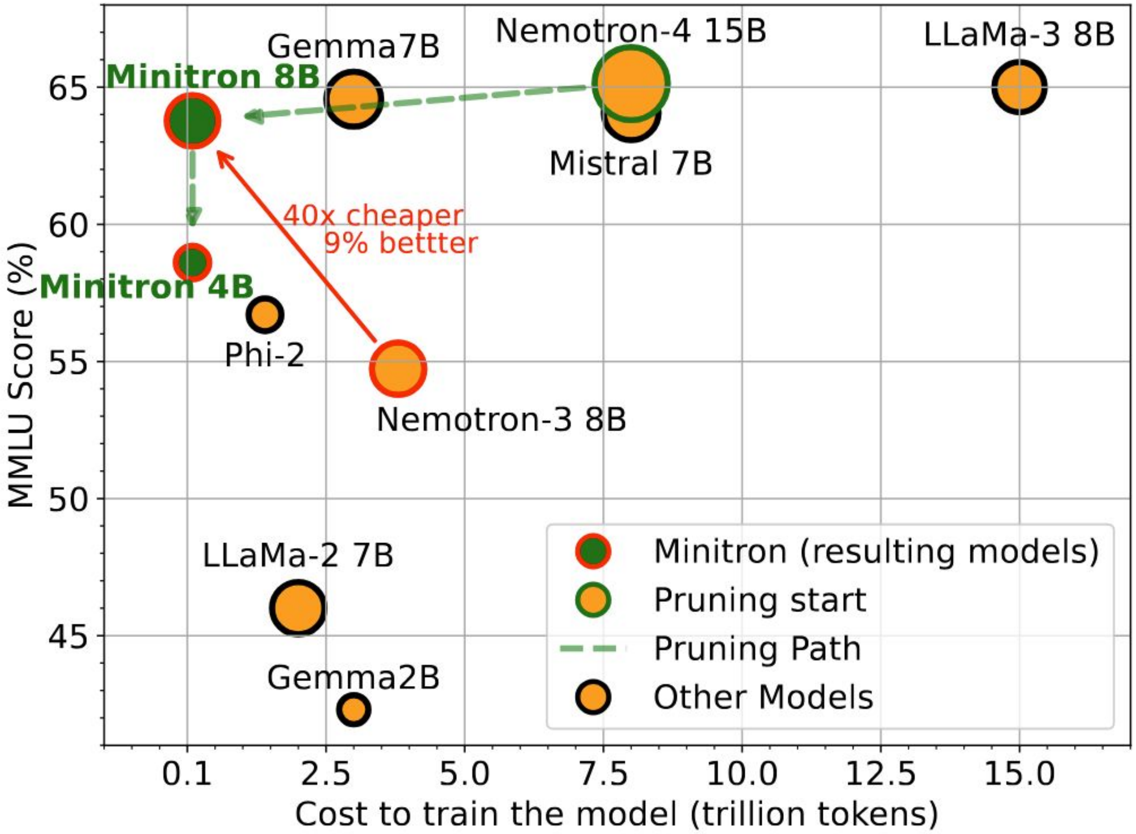


Image: arxiv.org/pdf/2407.14679

Figure 2: High-level overview of our proposed iterative pruning and distillation approach to train a family of smaller LLMs. On a pretrained LLM, we first evaluate importance of neurons, rank them, trim the least important neurons and distill the knowledge from the original LLM to the pruned model. The original model is replaced with the distilled model for the next iteration of compression.

Minitron Approach



Source: arxiv.org/pdf/2407.14679

Minitron Approach

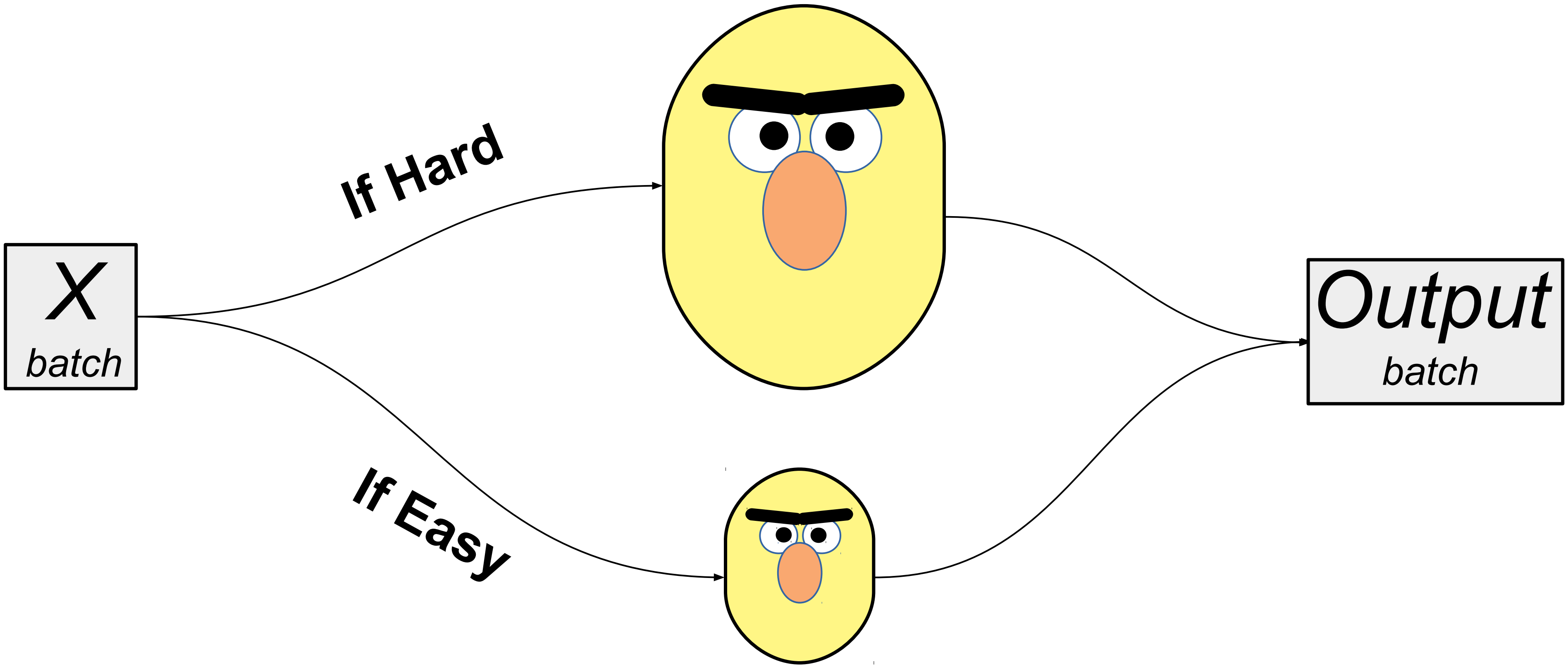
1. To train a family of LLMs, train the largest one and prune+distill iteratively to smaller LLMs.
2. Use (batch=L2, seq=mean) importance estimation for width axes and PPL/BI for depth.
3. Use single-shot importance estimation; iterative provides no benefit.
4. Prefer width pruning over depth for the model scales we consider (≤ 15B).
5. Retrain exclusively with distillation loss using KLD instead of conventional training.
6. Use (logit+intermediate state+embedding) distillation when depth is reduced significantly.
7. Use logit-only distillation when depth isn't reduced significantly.
8. Prune a model closest to the target size.
9. Perform lightweight retraining to stabilize the rankings of searched pruned candidates.
10. If the largest model is trained using a multi-phase training strategy, it is best to prune and retrain the model obtained from the final stage of training.

Source: arxiv.org/pdf/2407.14679

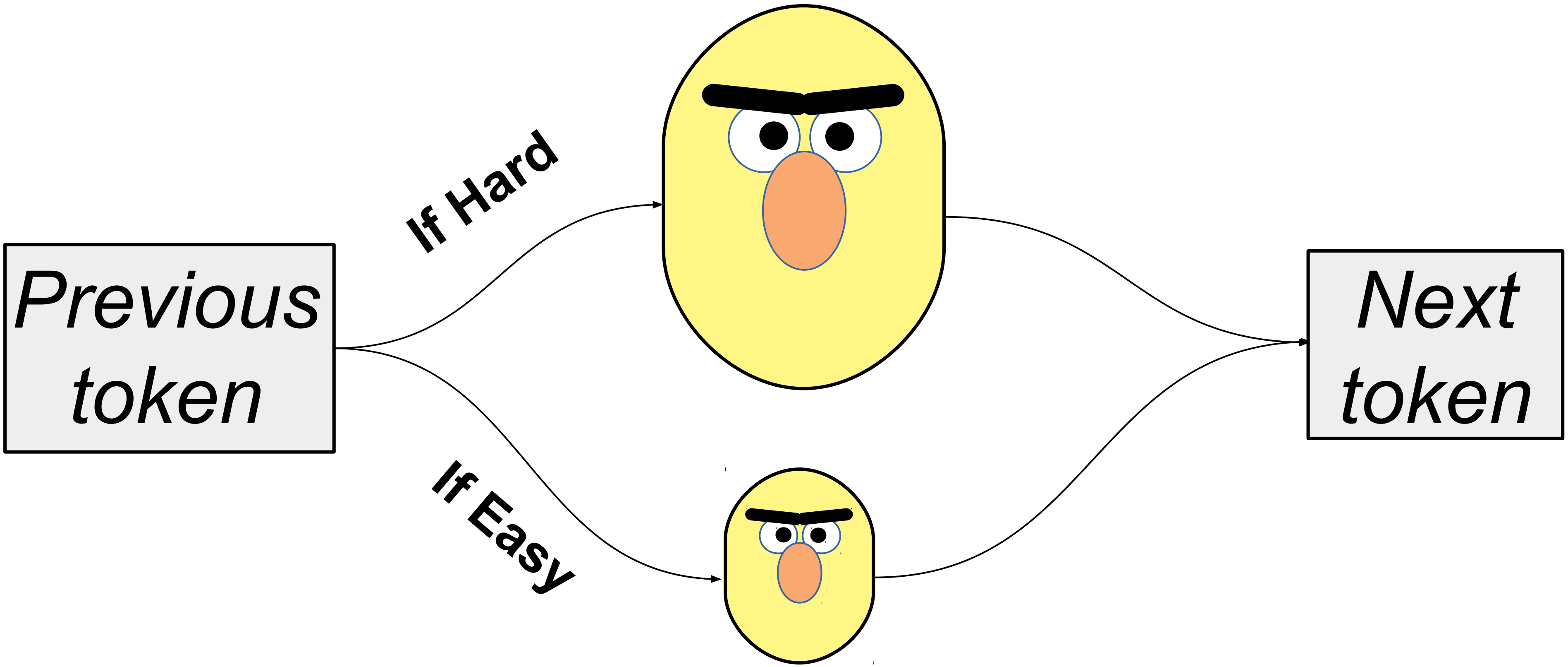
Compression by Distillation

- More distillation tricks:
  - Ensemble distillation https://arxiv.org/abs/1702.01802
    Dropout distillation http://proceedings.mlr.press/v48/bulo16.pdf
  - Co-distillation https://arxiv.org/abs/1804.03235

Smaller Model for Simpler Inputs



Smaller LLMs for Simpler Tokens?



Bottlenecks in GPU Inference

The math behind GPU utilization.

K tokens at a time:

- **Memory transfer** load of **O(N^2 + KxN)** per matrix-vector product.
- **Compute** load of **O(KxN^2)** per matrix-vector product.

Time cost of GPU operations:

- Bring million numbers from memory into kernels - **SLOW**
- Multiply million numbers with kernels - **FAST**

Bottlenecks in GPU Inference

K≈1 token at a time:

- The primary load in single user **chat applications**.
- **Memory transfer** load of **O(N^2)** per matrix-vector product.
- **Compute** load of **O(KxN^2)** per matrix-vector product.
- Memory transfer bottlenecked.

K>N tokens at a time:

- Prompt preprocessing, **highly parallel** inference.
- **Memory transfer** load of **O(KxN)** per matrix-matrix product.
- **Compute** load of **O(KxN^2)** per matrix-matrix product.
- Compute bottlenecked.

Bottlenecks in GPU Inference

K≈1 token at a time:

- Latency doesn’t depend on K!

K>N tokens at a time:

- Latency is linear with K.

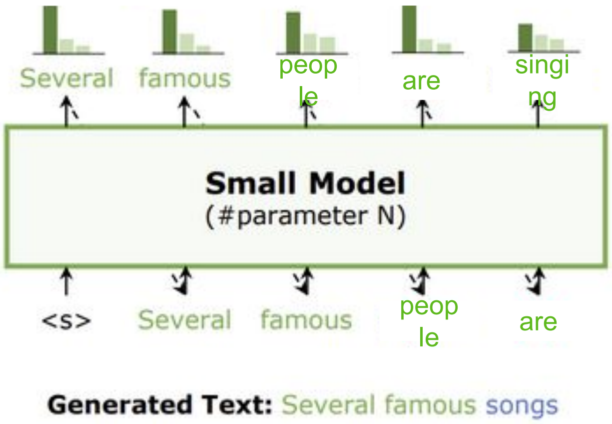
Speculative Decoding

https://arxiv.org/abs/2211.17192

Two models: main model (~Llama-70B) and draft model (~Llama-7B)

Greedy decoding:

Step 1: generate with draft model (sequential)



Speculative Decoding

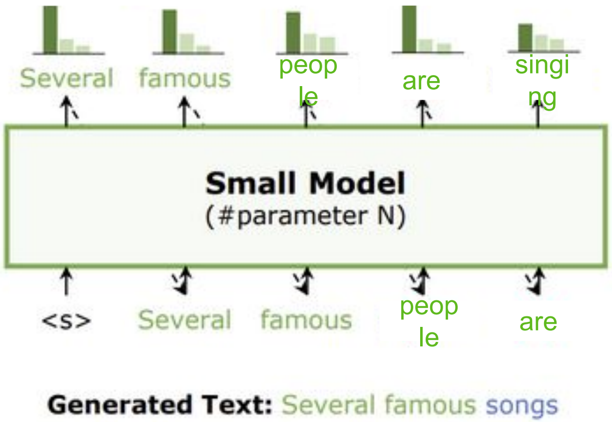
https://arxiv.org/abs/2211.17192

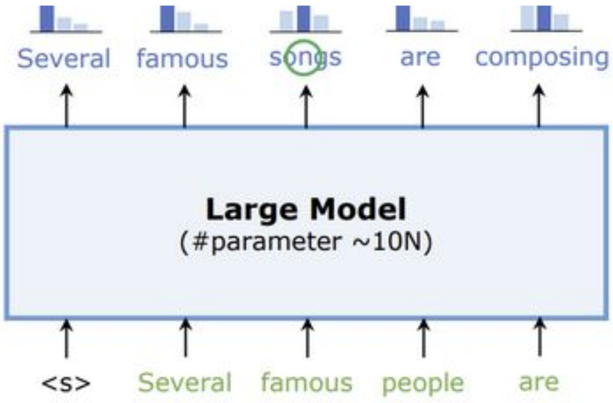
Two models: main model (~Llama-70B) and draft model (~Llama-7B)

Greedy decoding:

Step 1: generate with draft model (sequential)

Step 2: verify with large model (parallel)





Speculative Decoding

https://arxiv.org/abs/2211.17192

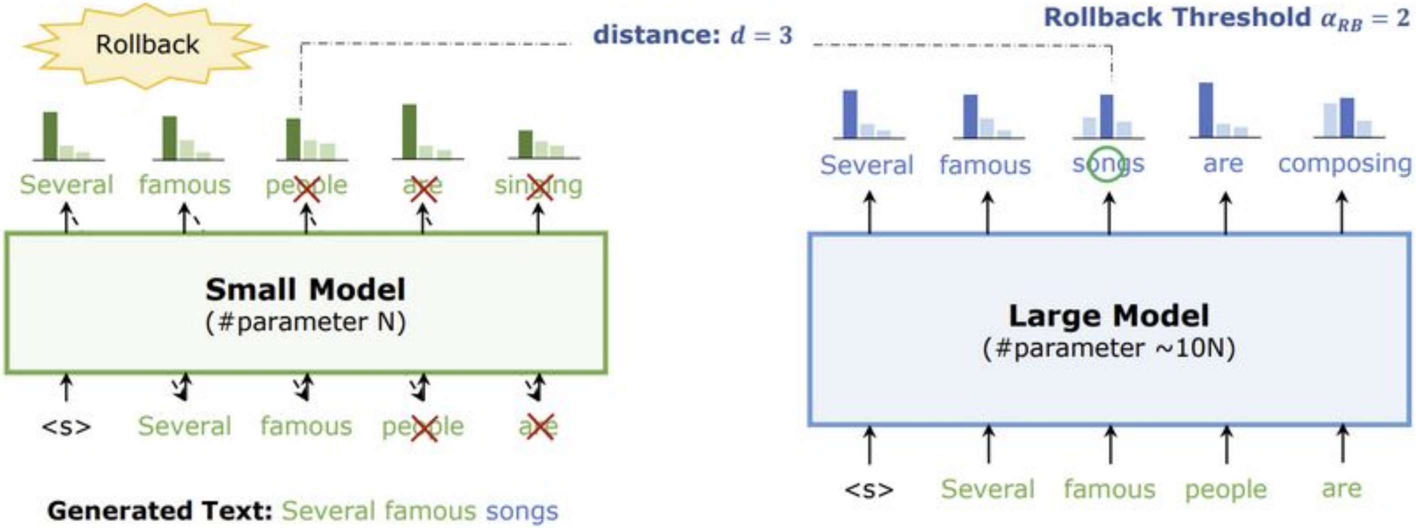
Two models: main model (~Llama-70B) and draft model (~Llama-7B)

Greedy decoding:

Step 1: generate with draft model (sequential)

Step 2: verify with large model (parallel)

Accept multiple tokens



Speculative Decoding

https://arxiv.org/abs/2211.17192

Two models: main model (~Llama-70B) and draft model (~Llama-7B)

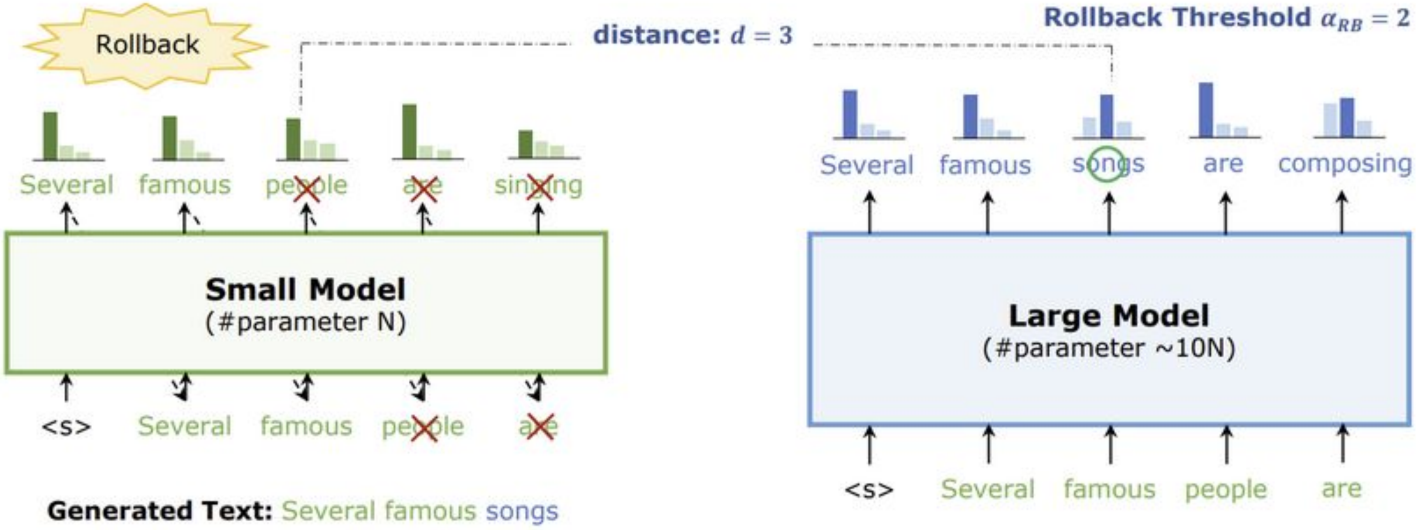
Greedy decoding:

Step 1: generate with draft model (sequential)

Step 2: verify with large model (parallel)

Accept multiple tokens

Repeat

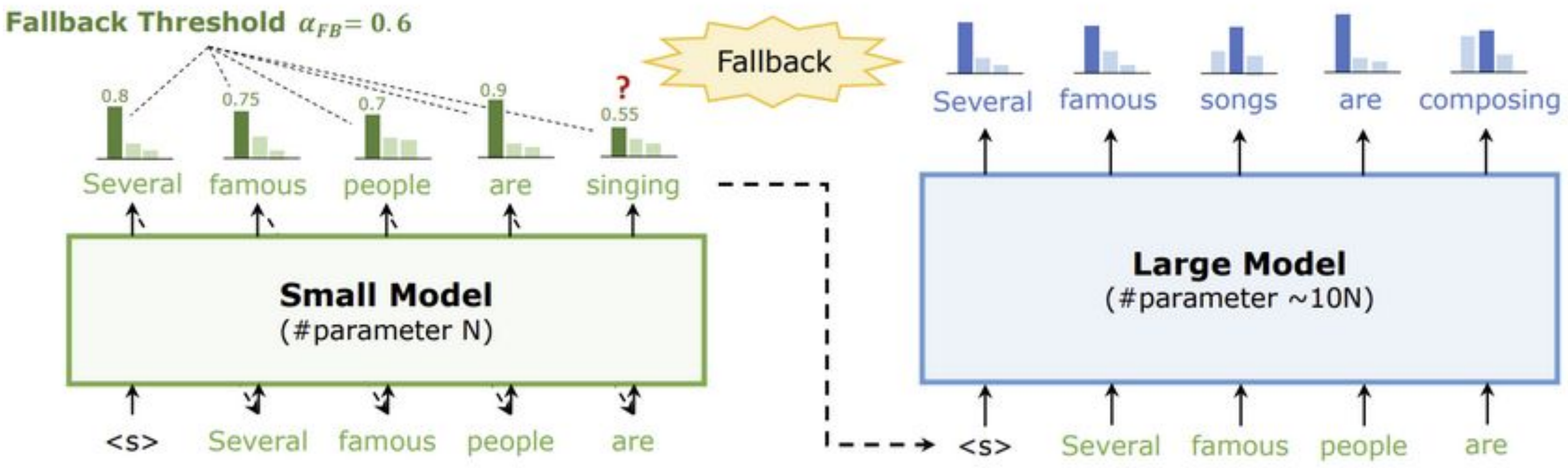


Speculative Decoding

https://arxiv.org/abs/2211.17192

Two models: main model (~Llama-70B) and draft model (~Llama-7B)

Sampling (temperature, top-p, top-k): generate, then reject with probability sampling probability proven equal to regular sampling

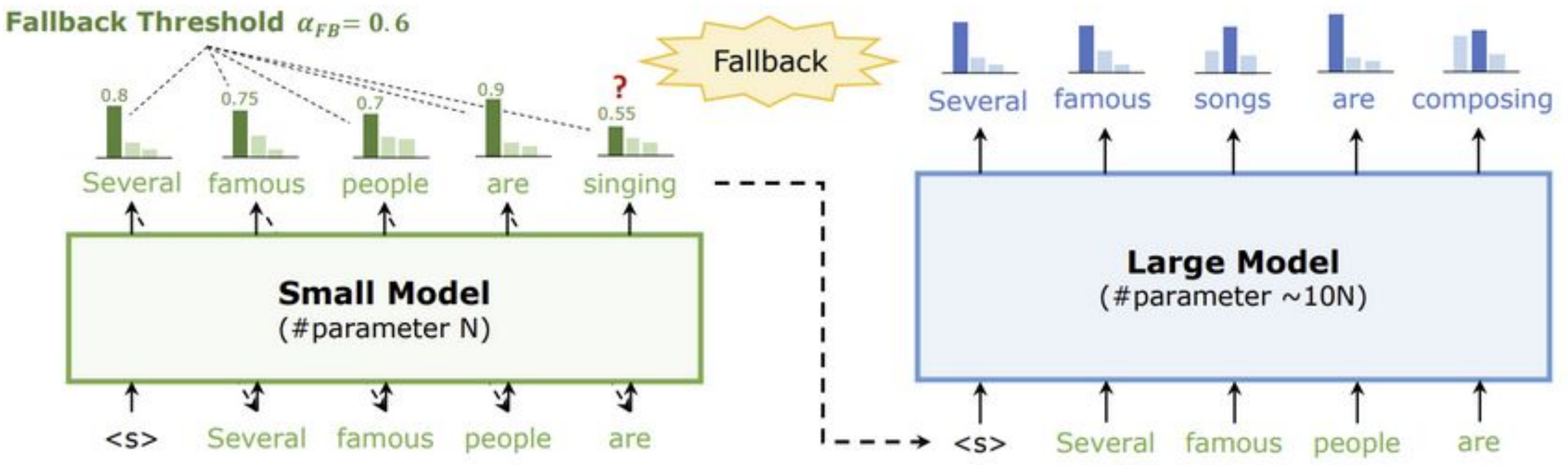


Speculative Decoding

That you'll implement

Two models: main model (Llama-8B) and draft model (of your choice)

Greedy Sampling: generate, then reject everything after exact match



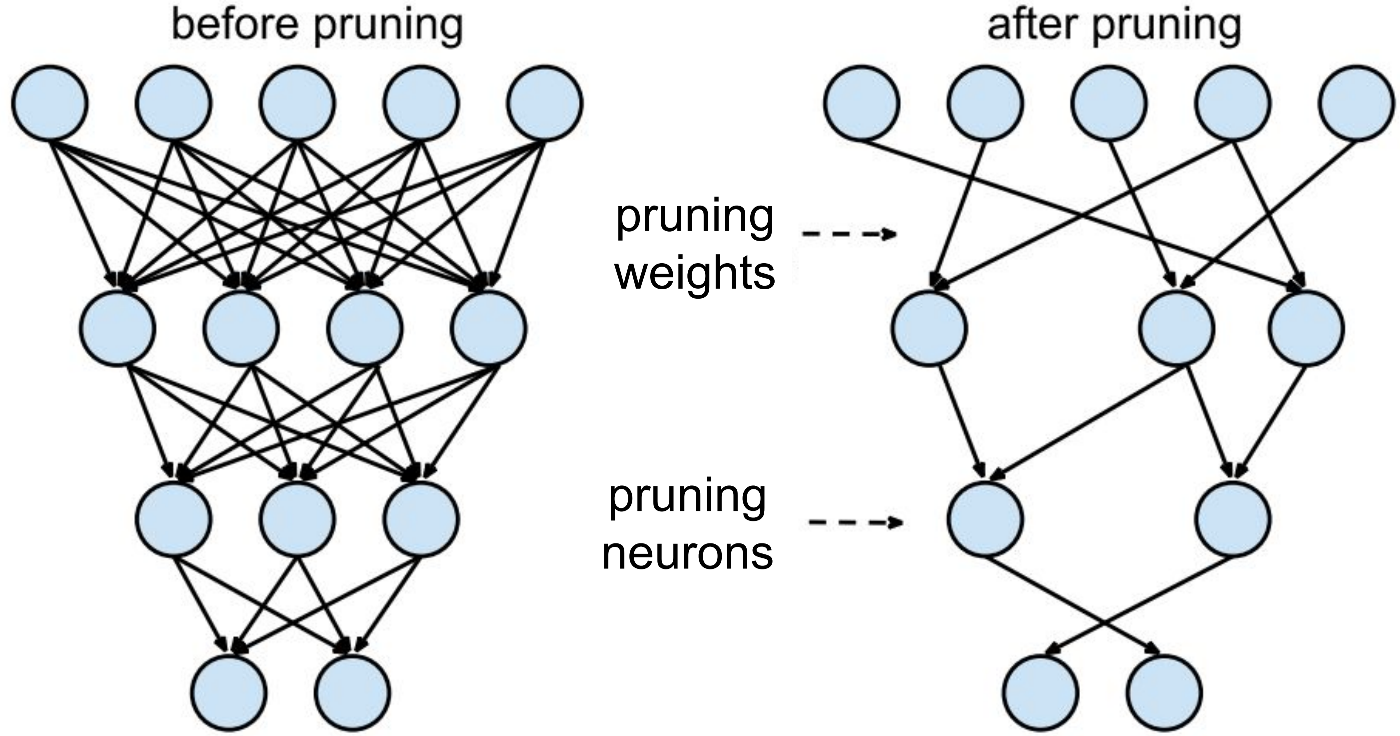
Going Inside Model Layers

Compression by Sparsification

Do we really need all D by D weights?

Compression by Pruning

Do we really need all D by D weights?



Magnitude Pruning

Drop ~5% smallest weights
from each layer every 1000 steps
(and keep training)

Importance Estimation

Minitron:

- attn_head/neuron level: activation scales
- Block level: PPL

L0: Learnable importance estimation:

$$w_i = w_i \times \sigma(a_i + N(0,1))$$

Read more: https://arxiv.org/abs/1712.01312
Alternative: https://arxiv.org/abs/1701.05369

Compression by Sparsification

Unstructured sparsity = prune individual weights
(minimal model size)

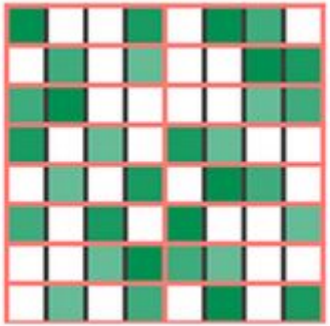
Structured sparsity= prune entire neurons/heads
(fastest inference)

Compression by Sparsification

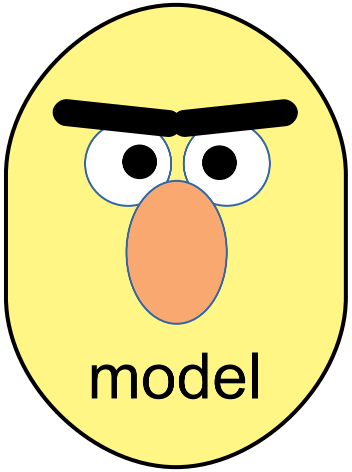
Unstructured sparsity = prune individual weights
(minimal model size)

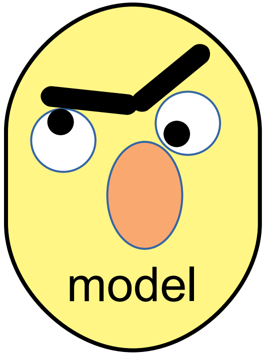
Structured sparsity= prune entire neurons/heads
(fastest inference)

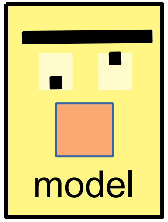
Note: some GPU/FPGAs also run fast with low-level structured sparsity, e.g. “Any 2 of 4 consecutive weights” (left)



Compression by quantization





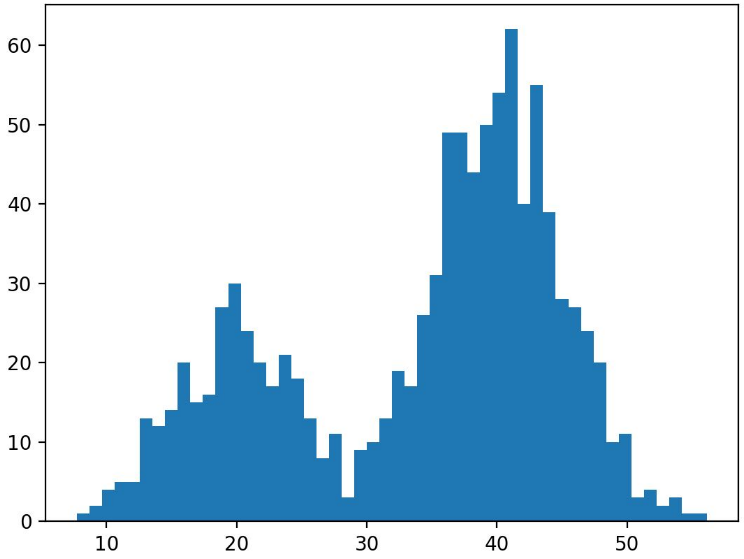


INT8

8 BITS

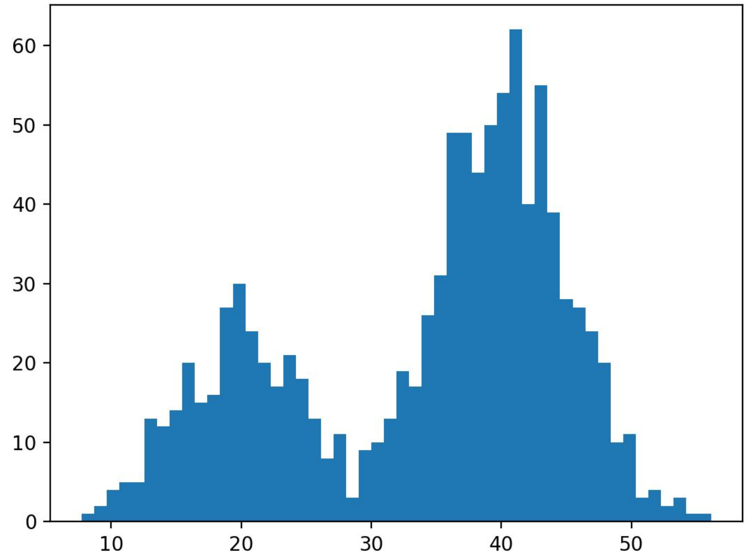
Quantization basics

Goal: encode data as int8 / int4



Quantization basics

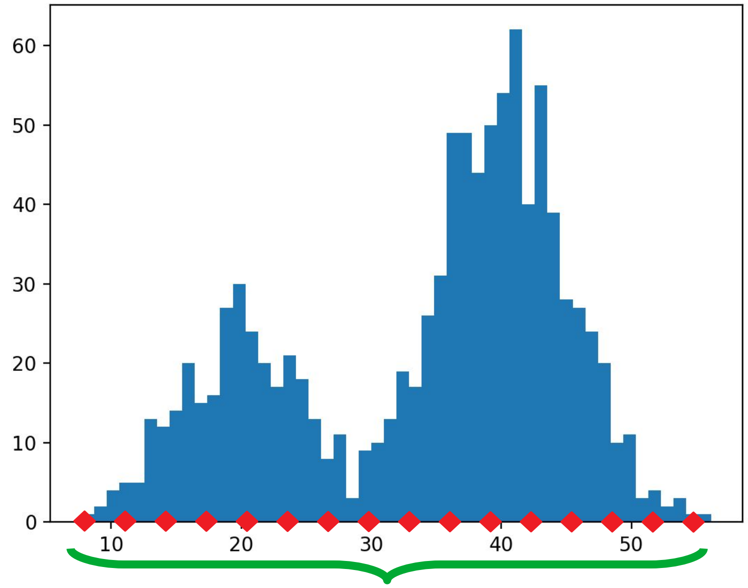
Goal: encode data as int8 / int4



not an ideal range for int4

Linear quantization

Fit a linear range to data



scale = (max(w) – min(w)) / 2^4

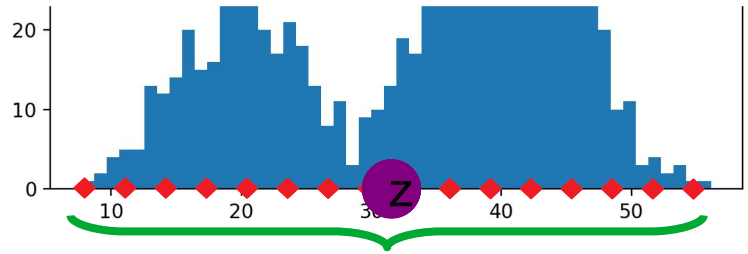
zero = -min(w) / scale

Linear quantization

Fit a linear range to data

Encode: $c_i = (w_i / \mathbf{s} + \mathbf{z}).\text{clip}(0, 15)$ uint4 range

Decode: $\mathbf{w_i} =$ **??? ideas?**



scale = (max(w) – min(w)) / 2^4

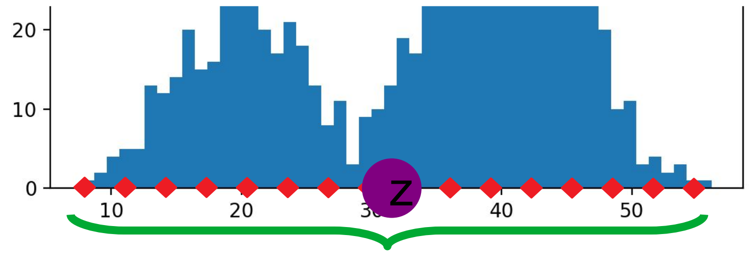
zero = -min(w) / scale

Linear quantization

Fit a linear range to data

Encode: $c_i = (w_i / s + z).\text{clip}(0, 15)$ uint4 range

Decode: $w_i \approx s * c_i - z$



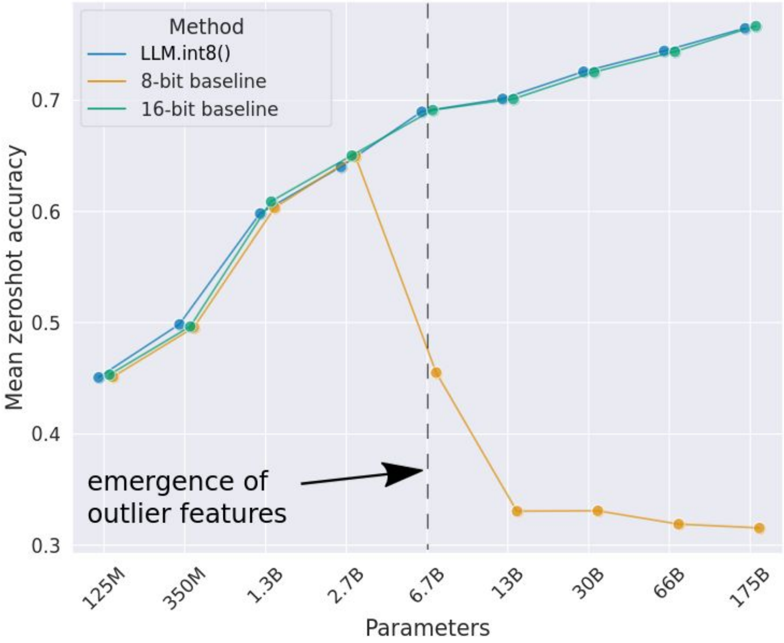
scale = (max(w) – min(w)) / 2^4

zero = -min(w) / scale

LLM.8bit(): some weights are more important

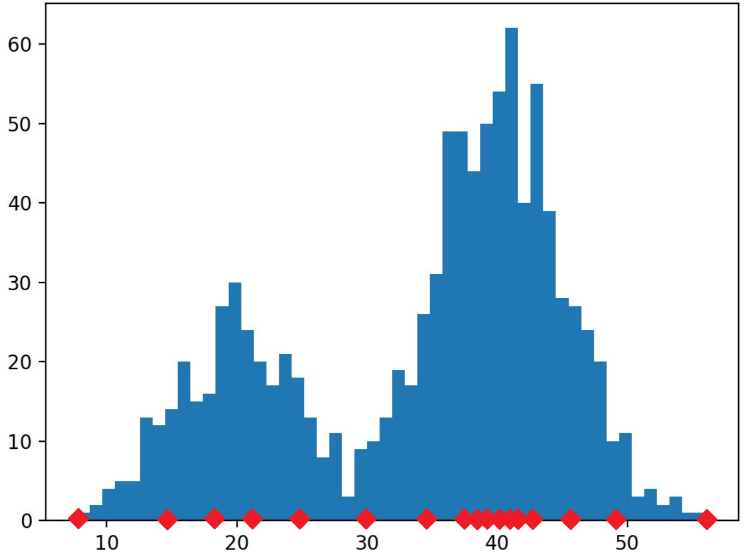
https://arxiv.org/abs/2208.07339

TL;DR in very LLM, some input features become outliers
Weights for those features are sensitive
KEEP <1% MOST SENSITIVE WEIGHTS IN 16-bit!



Nonlinear quantization

Fit codes to data



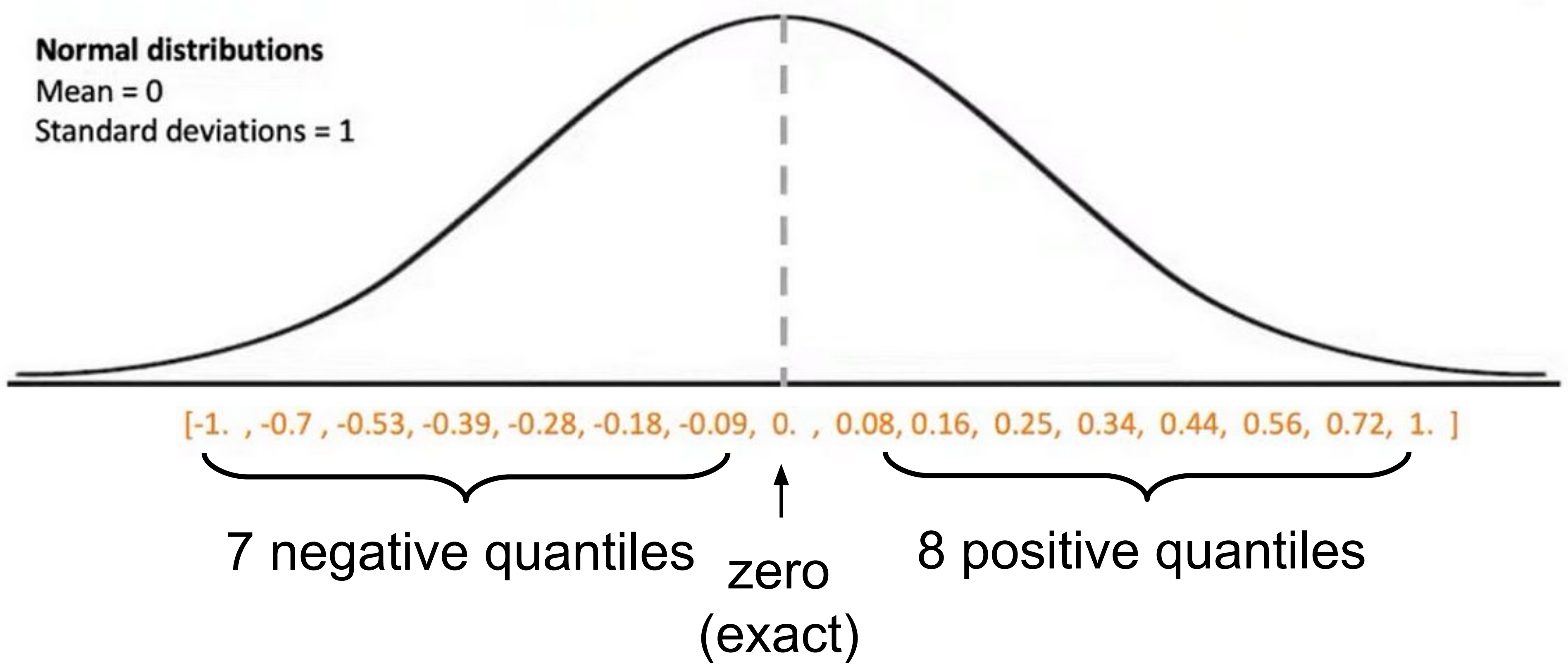
Compute a grid of percentiles or centroids (k-means 1d)

Store each weight as the index of nearest percentile/centroid

Static nonlinear case: NF4

https://arxiv.org/abs/2305.14314

https://arxiv.org/abs/2306.06965



Static nonlinear case: NF4

https://arxiv.org/abs/2305.14314

https://arxiv.org/abs/2306.06965

How to use:

```
model = transformers.AutoModelForCausalLM.from_pretrained(
    "Enoch/llama-7b-hf", load_in_4bit=True)
```

[-1. , -0.7 , -0.53, -0.39, -0.28, -0.18, -0.09, 0. , 0.08, 0.16, 0.25, 0.34, 0.44, 0.56, 0.72, 1.]

7 negative quantiles

zero (exact)

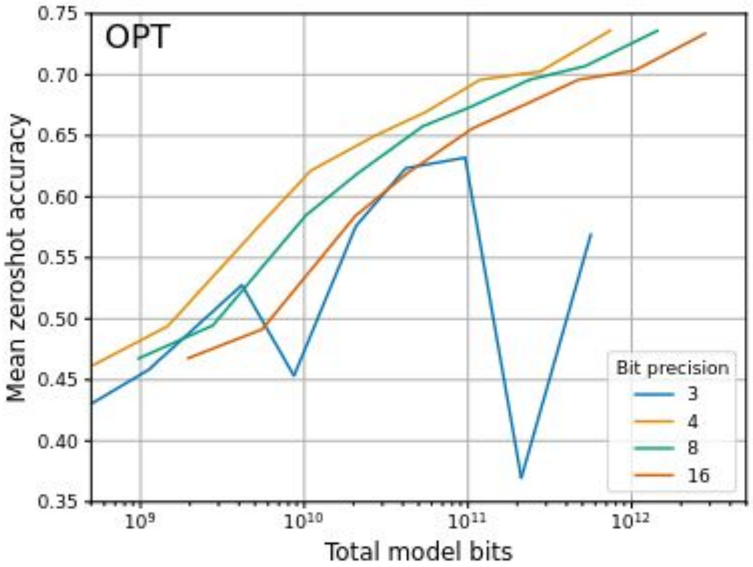
8 positive quantiles

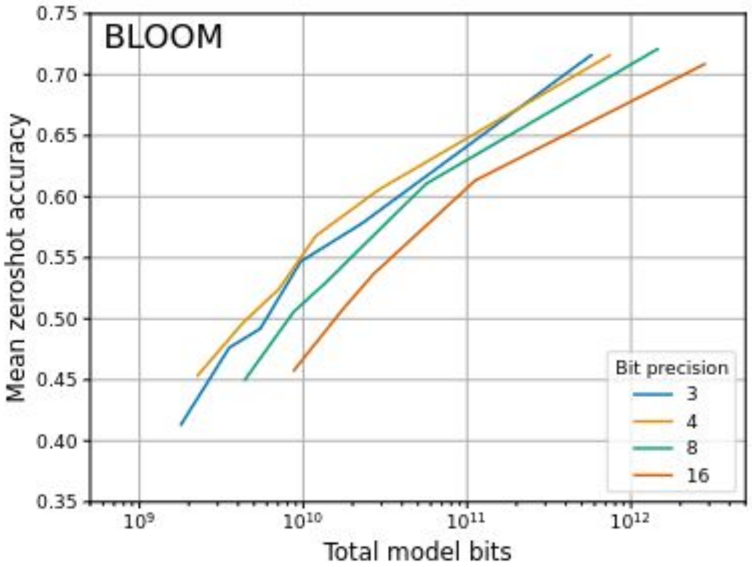
How many bits is best?

https://arxiv.org/abs/2212.09720

TL;DR 3-4 bits looks optimal

2-bit: Smaller models in 4 bits are better than larger models in 2 bits



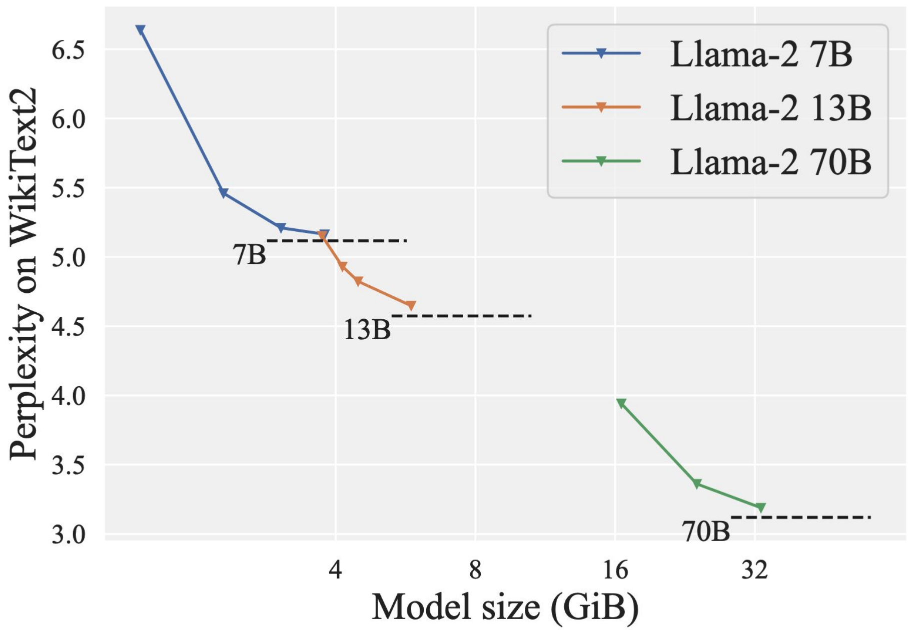


How many bits is best?

https://arxiv.org/abs/2401.06118

TL;DR ~2 bits looks optimal

2-bit: more expensive to quantize, more expensive to run



GPU Inference

K≈1 token at a time:

- The primary load in single user **chat applications**.
- **Memory transfer** load of **O(N^2)** per matrix-vector product.
- **Compute** load of **O(KxN^2)** per matrix-vector product.
- Memory transfer bottlenecked.

K>N tokens at a time:

- Prompt preprocessing, **highly parallel** inference.
- **Memory transfer** load of **O(KxN)** per matrix-matrix product.
- **Compute** load of **O(KxN^2)** per matrix-matrix product.
- Compute bottlenecked.

Quantized GPU Inference

With weight compression ratio **C**.

K≈1 token at a time:

- The primary load in single user **chat applications**.
- **Memory transfer** load of **O(N^2 / C)** per matrix-vector product.
- **Compute** load of **O(KxN^2)** per matrix-vector product.
- Memory transfer bottlenecked.

K>N tokens at a time:

- Prompt preprocessing, **highly parallel** inference.
- **Memory transfer** load of **O(KxN)** per matrix-matrix product.
- **Compute** load of **O(KxN^2)** per matrix-matrix product.
- Compute bottlenecked.

Quantized GPU Inference

With weight compression ratio **C**.

K≈1 token at a time:

- **Memory transfer** load of **O(N^2 / C)** per matrix-vector product.

- Memory transfer bottlenecked.

K>N tokens at a time:

- **Compute** load of **O(KxN^2)** per matrix-matrix product.
- Compute bottlenecked.

Quantized GPU Inference

With weight compression ratio **C**.

K≈1 token at a time:

- **C** times faster inference!

K>N tokens at a time:

- No speedup :(

Fully Quantized GPU Inference

With weight+activations compression ratio **C**.

K≈1 token at a time:

- **Memory transfer** load of **O(N^2 / C)** per matrix-vector product.
- Memory transfer bottlenecked.

K>N tokens at a time:

- **Compute** load of **O(KxN^2 / C)** per matrix-matrix product.
- Compute bottlenecked.

Fully Quantized GPU Inference

With weight+activations compression ratio **C**.

K≈1 token at a time:

- **C** times faster inference!

K>N tokens at a time:

- **C** times faster inference!

Model compression landscape

Goal: faster / smaller / both

Compression: quantize / prune / factorize

Setup: no data, some data, training data

Compress what: weights / activations /cache

Model compression landscape

Goal: faster / smaller / both

Compression: quantize / prune / factorize

Setup: no data, some data, training data

Compress what: weights / activations /cache

Q: can we take advantage of data
to improve quantization?

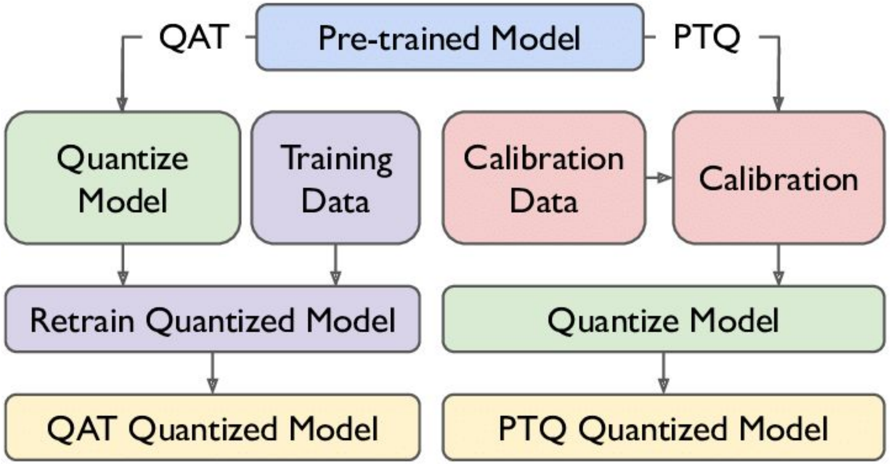
Model compression landscape

Goal: faster / smaller / both

Compression: quantize / prune / factorize

Setup: no data, some data, training data

Compress what: weights / activations /cache



Compression-aware training

Step 1: train normally for T steps

Step 2: prune 5% weights (or quantize 10% layers)

Step 3: freeze pruned/quantized parts

GoTo step 1

Post-training compression

Can we take advantage of **a few** data points without training the model?

Post-training compression

Can we take advantage of **a few** data points without training the model?

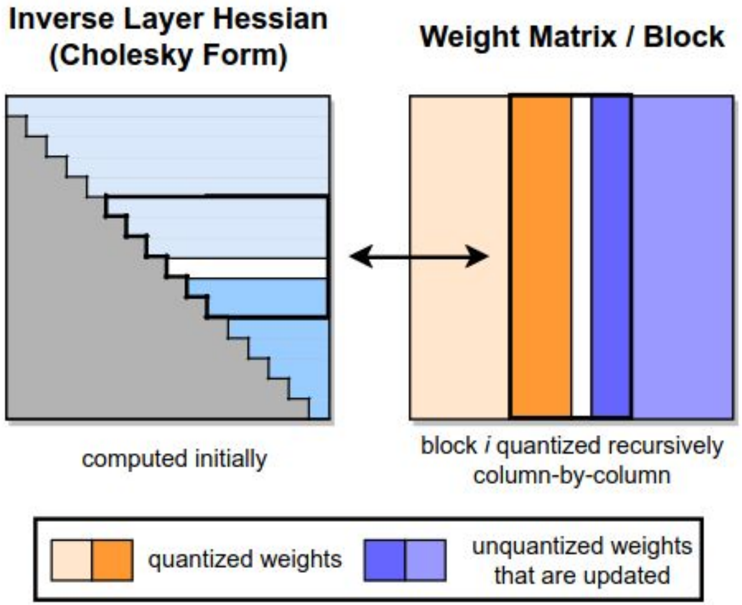
- find which weights are multiplied by larger inputs
- find correlated or anti-correlated weights
- try to “cancel out” quantization errors

GPTQ

[Frantar et al, 2022](#)

Minimize the same objective $\mathrm{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

For i = 1 … in_features:
- **quantize** i-th column of weight matrix
  (from one input feature and all outputs)
- freeze the quantized model forever
- update all remaining columns

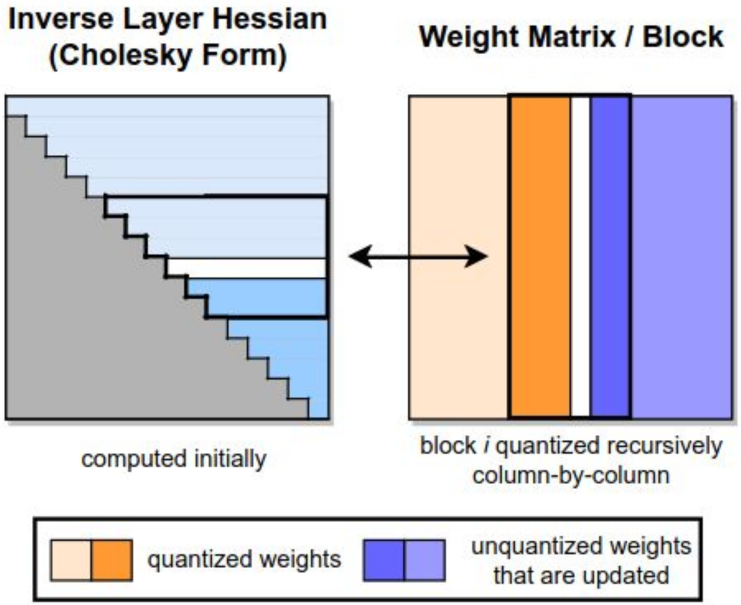


GPTQ

[Frantar et al, 2022](#)

Minimize the same objective $\mathrm{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

For i = 1 … in_features:
- **quantize** i-th column of weight matrix
  (from one input feature and all outputs)
- freeze the quantized model forever
- update all remaining columns



Use linear quantization with one scale & zero per each group of G weights

Tricks: process weights in “hacky” order

GPTQ

[Frantar et al, 2022](https://huggingface.co)

Minimize the same objective $\text{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

For i = 1

- **quantiz**

(from

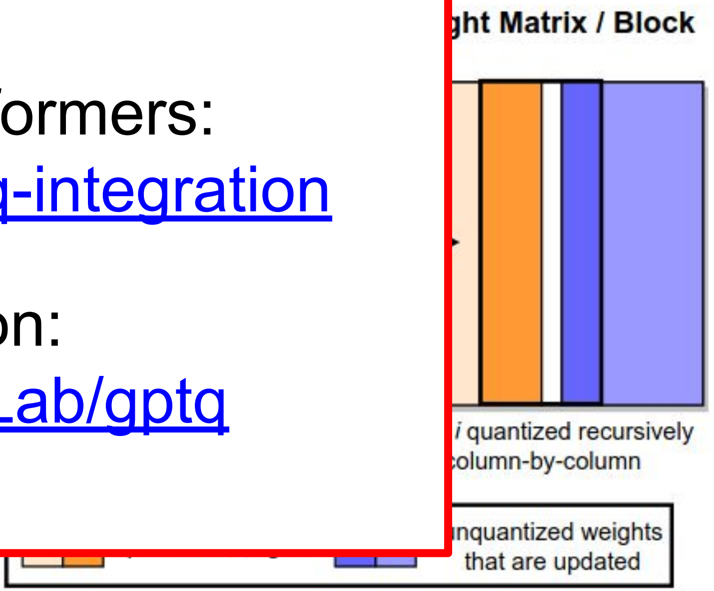
- freeze t
- update

Use linear

zero per ea

Native support in HF Transformers:
https://huggingface.co/blog/gptq-integration

Original implementation:
https://github.com/IST-DASLab/gptq



Tricks: process weights in “hacky” order

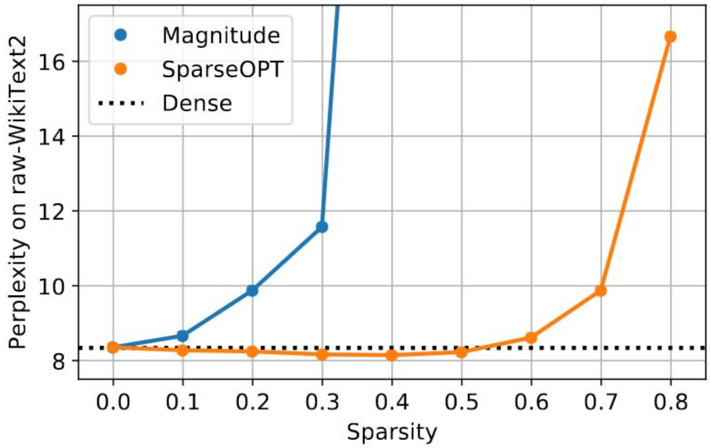
SparseGPT

[Frantar et al, 2023]

Minimize the same objective $\text{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

For i = 1 … in_features:
- sparsify i-th column of weight matrix
  (from one input feature and all outputs)
- freeze the quantized model forever
- update all remaining columns

OPT-175B



Dynamically choose how many weights to prune on each step (threshold on error)

Trick: can do 2-out-of-4 sparsity for A100

QuIP#/AQLM/SpinQuant

Minimize the same objective $\text{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

QuIP#/AQLM/SpinQuant

Minimize the same objective $\text{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

But also finetune while quantizing!

QuIP#

Minimize the same objective $\mathrm{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

But also finetune while quantizing!

On the level of blocks, after quantizing each linear layer:

$$\mathrm{argmin}_{\mathrm{Unquantized} \in B} ||B(X) - \widehat{B}(X)||_2^2$$

More Quantization Papers

PV-Tuning – SOTA weight-only quantization optimization procedure
https://arxiv.org/abs/2405.14852

QTIP – SOTA weight-only quantization algorithm
https://arxiv.org/abs/2406.11235

QuaRot - SOTA GPTQ-based weight+act quantization algorithm
https://arxiv.org/abs/2404.00456

SpinQuant – SOTA weight+act quantization algorithm with finetuning
https://arxiv.org/abs/2405.16406

Many more cool papers

Homework:

- `hw_speculative.ipynb` - Implement simple speculative decoding
  - *4+(?)* Points
- `hw_quantization.ipynb` - Implement GPTQ
  - *6+(3)* Points