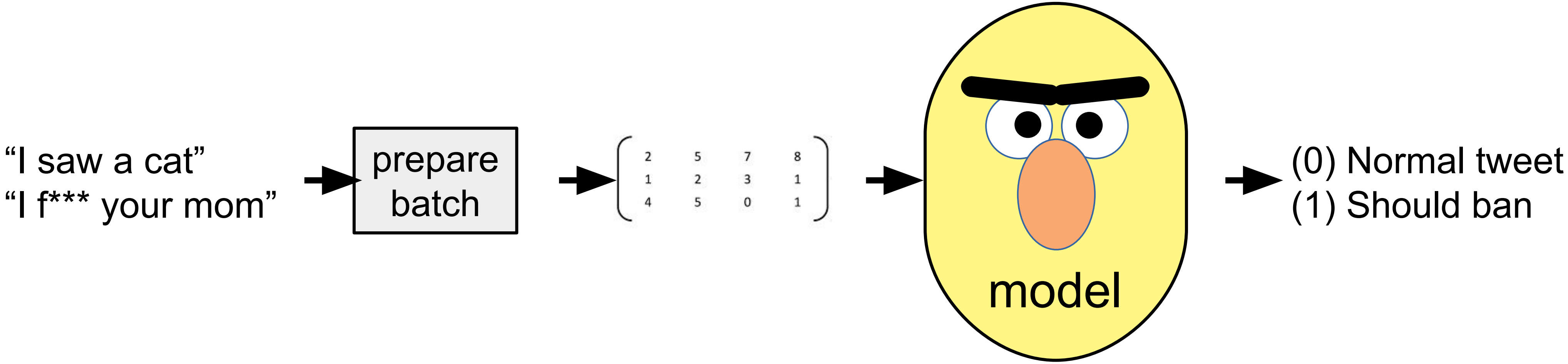
Lecture 09: Model compression & acceleration

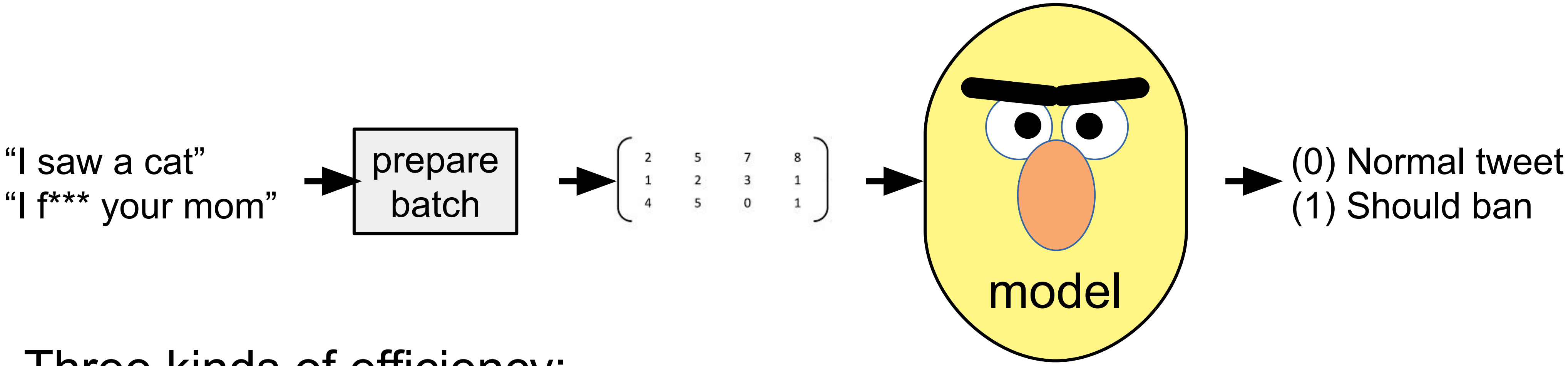
Michael Diskin (with the help of one shy hedgehog)

Chapter 1: why should you care?

Case study: text classification



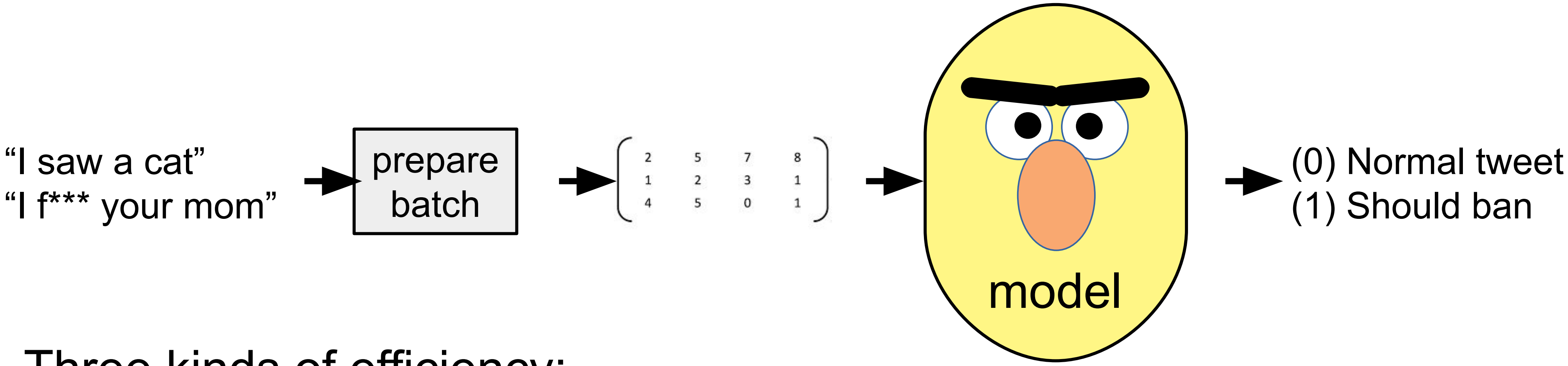
Case study: text classification



Three kinds of efficiency:

Model Size
(mega)bytes

Case study: text classification

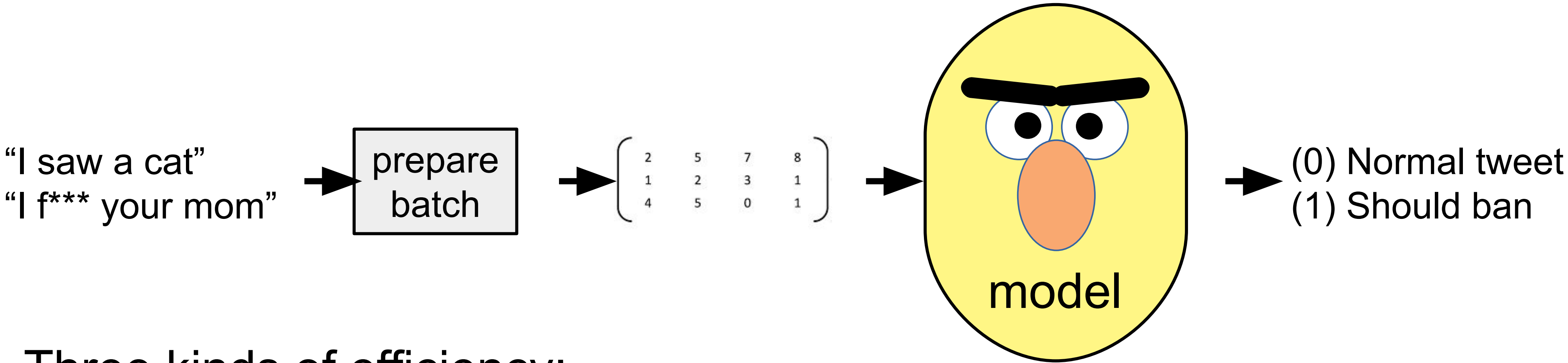


Three kinds of efficiency:

Model Size
(mega)bytes

Throughput
samples/second

Case study: text classification



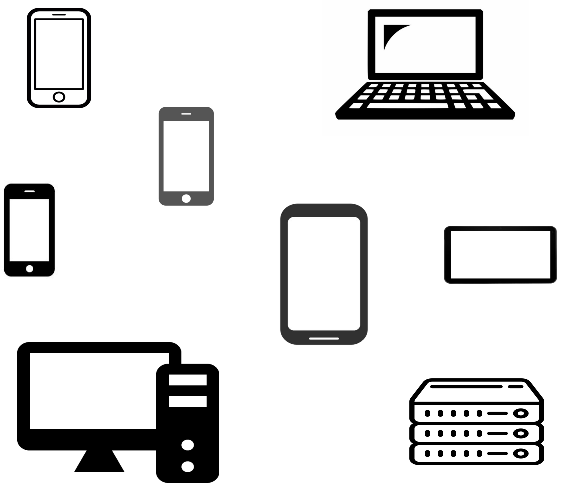
Three kinds of efficiency:

Model Size
(mega)bytes

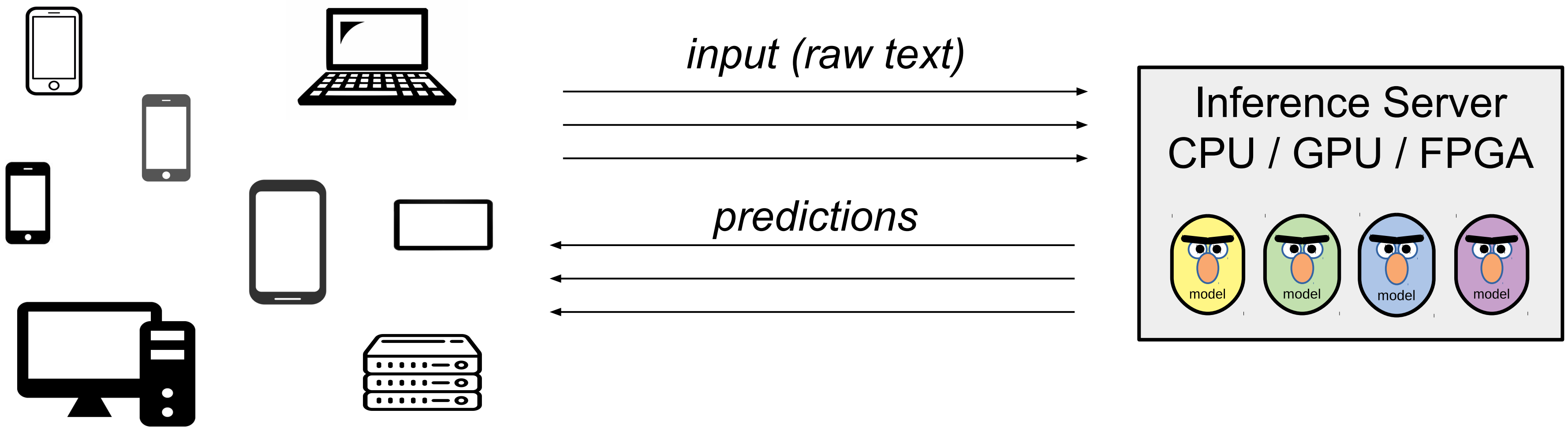
Throughput
samples/second

Latency
ms@percentile

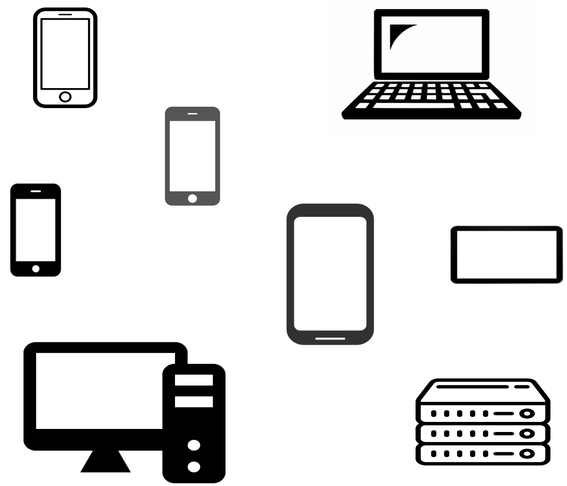
Scenario 1: inference server

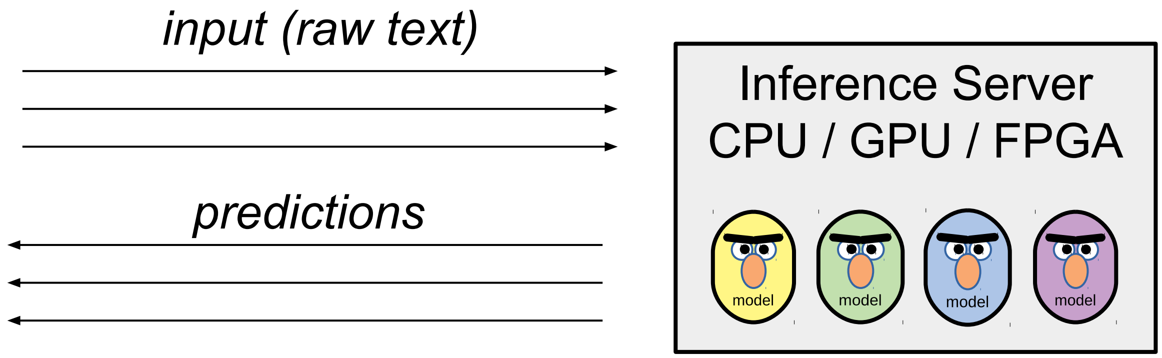


Scenario 1: inference server



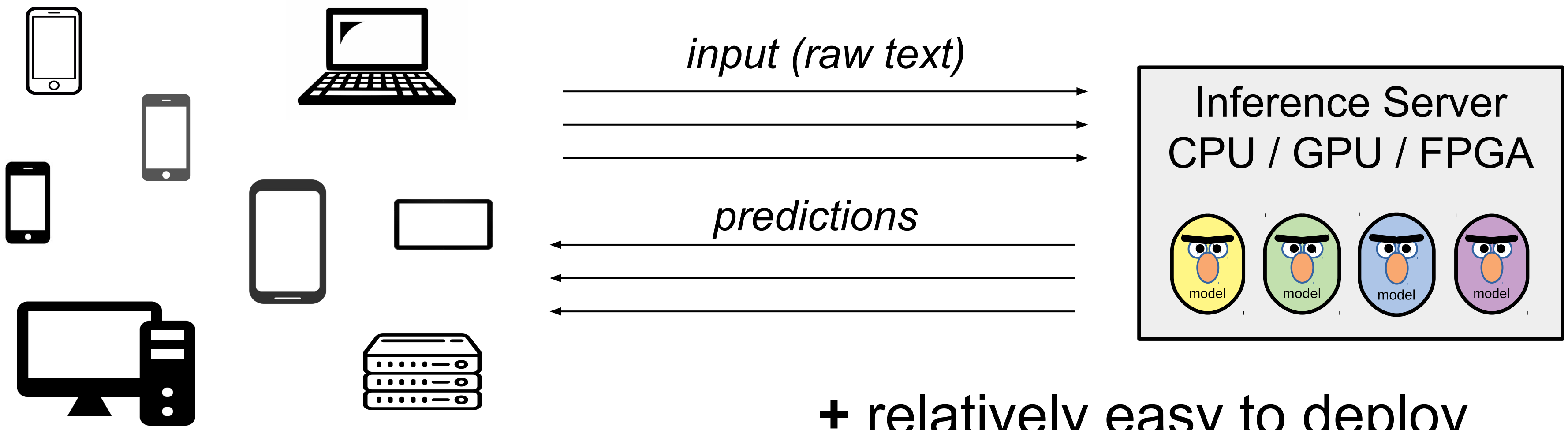
Scenario 1: inference server





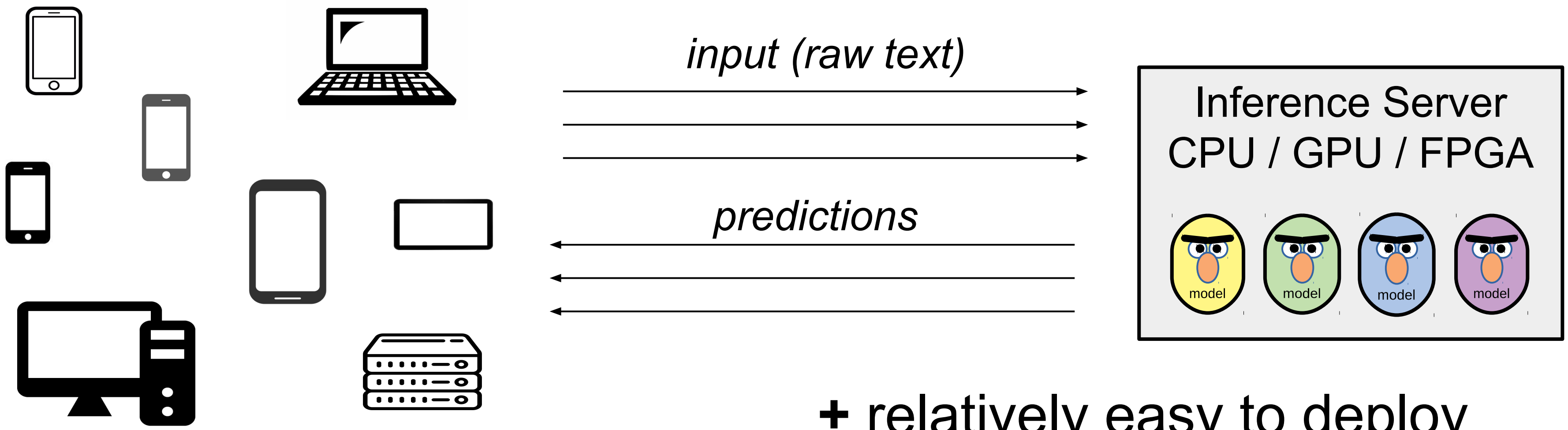
+ relatively easy to deploy
+ you control model & inference
+ clients don't run compute

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

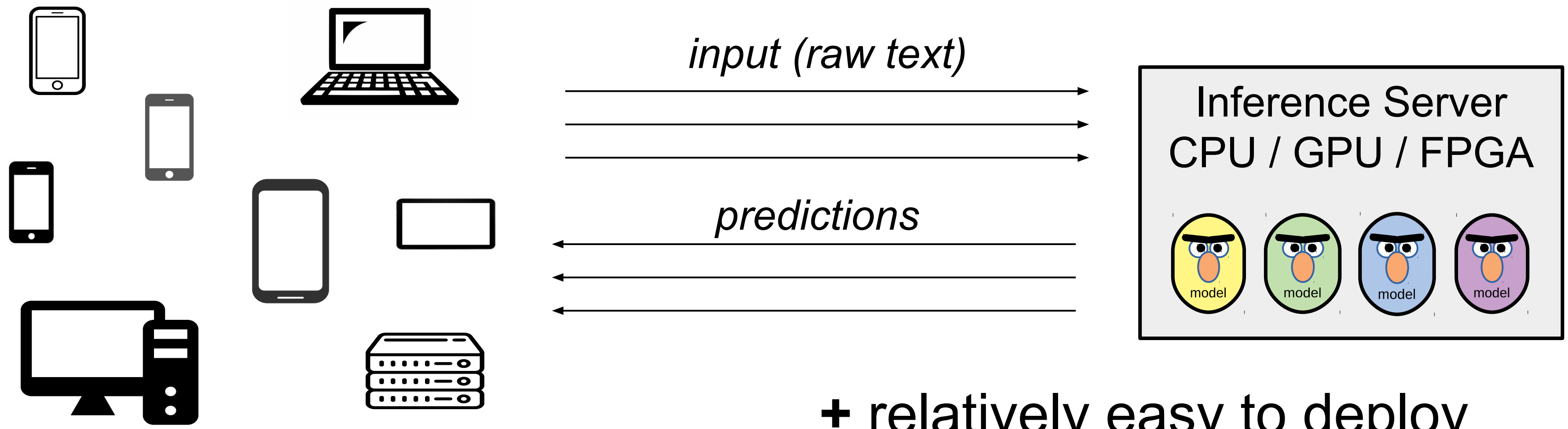
Scenario 1: inference server



Which is the most important?

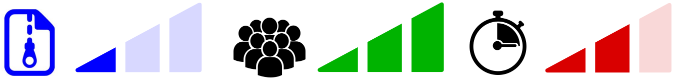
+ relatively easy to deploy
+ you control model & inference
+ clients don't run compute
– you pay for each inference
– clients can't work offline
– network latency

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

Priorities:



Note: smaller model = you can fit more models in the same memory

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 :)

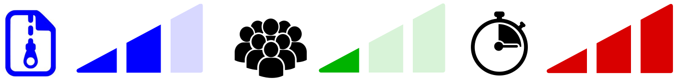
Popular frameworks:

	priorities
TensorFlow Serving	efficiency $\ll$ developer time
TensorRT Inference Server (Triton)	efficiency $\approx$ developer time
Custom model-dependent code	efficiency $\gg$ developer time

Scenario 2: local inference

- Preload model onto a dedicated device, infer locally using that device
- Typical use cases:
  - Parallel speech recognition
  - “Smart” cameras
  - Autonomous drones
  - Self-driving cars

Priorities:



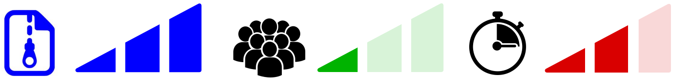
Scenario 3: web/smartphone app

- Load model weights on the fly and infer locally
- Model size is critical for both you and the user

Scenario 3: web/smartphone app

- Load model weights on the fly and infer locally
- Model size is critical for both you and the user

- Autonomous machine translation (tinyurl.com/yandex-translate-app)
- Pix2pix demo in a browser (https://affinelayer.com/pixsrv)

- Priorities: 

Scenario 3: web/smartphone app

- Load model weights on the fly and infer locally
- Model size is critical for both you and the user

- Autonomous machine translation (tinyurl.com/yandex-translate-app)
- Pix2pix demo in a browser (https://affinelayer.com/pixsrv)

- Priorities:

- Popular frameworks:

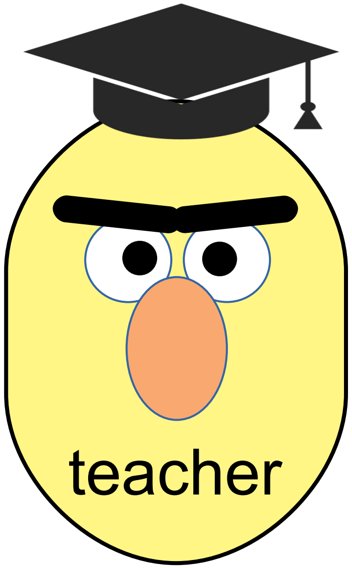
	Platform
TensorFlow.js	All modern browsers
CoreML	iOS devices
NNAPI	Android devices

Chapter 2: how do I improve my model?

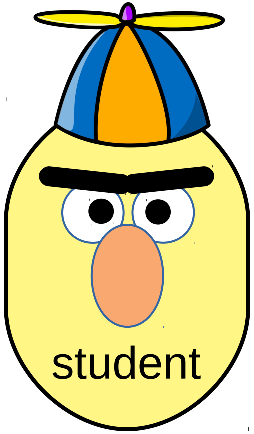
Compression by Distillation

Distillation…
Heard that word before?

Compression by Distillation

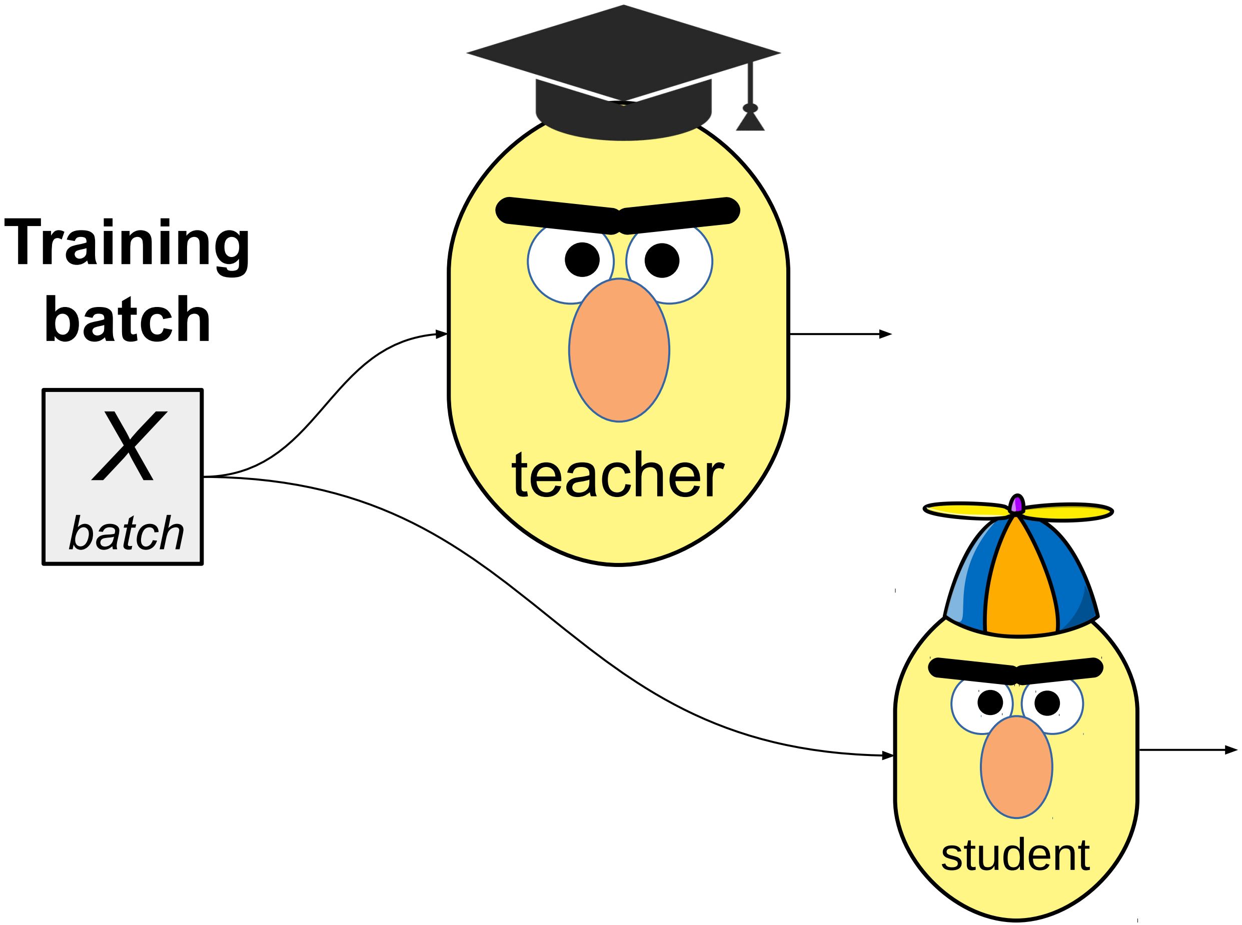


First, get the best performing model regardless of size



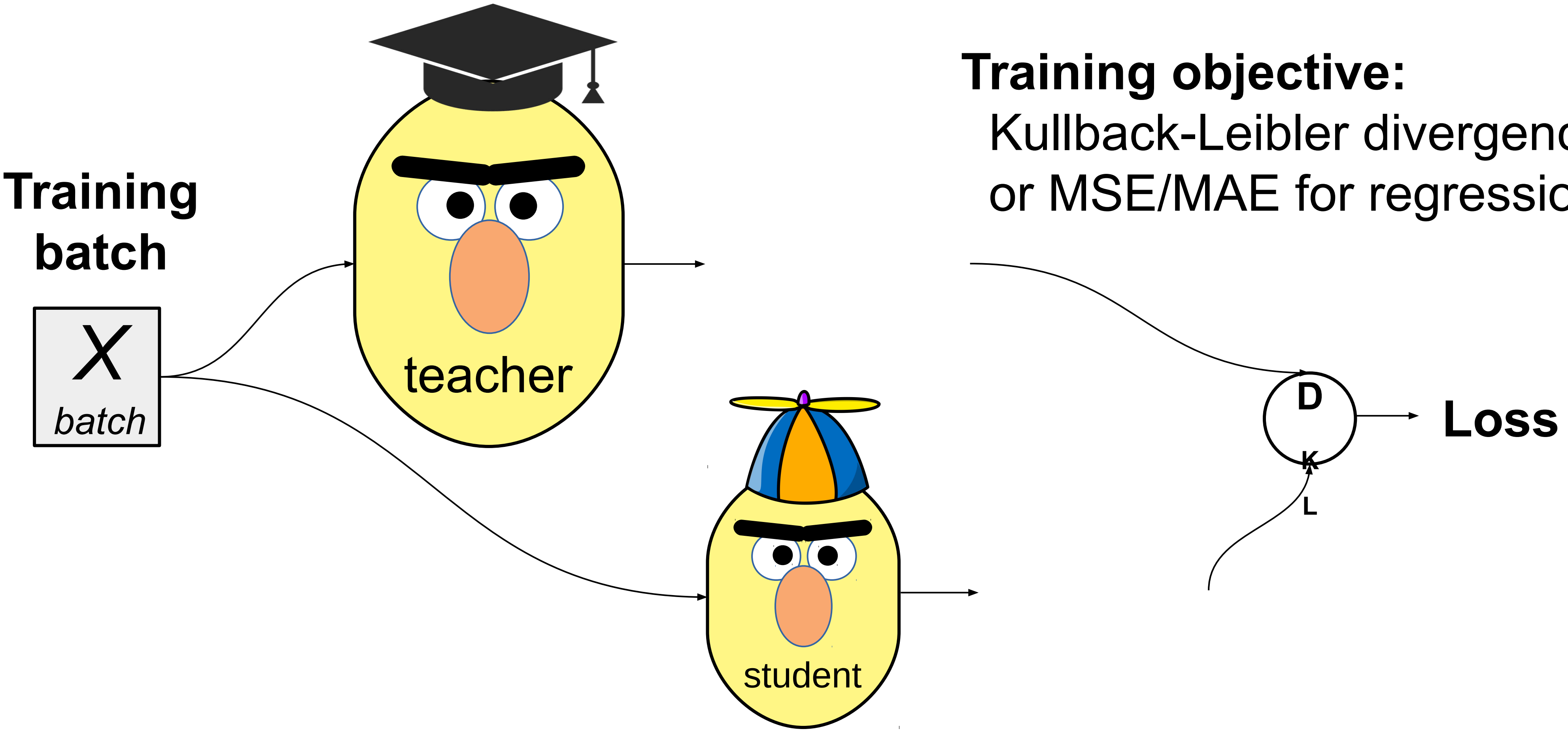
Then, train a more compact model to approximate it!

Compression by Distillation



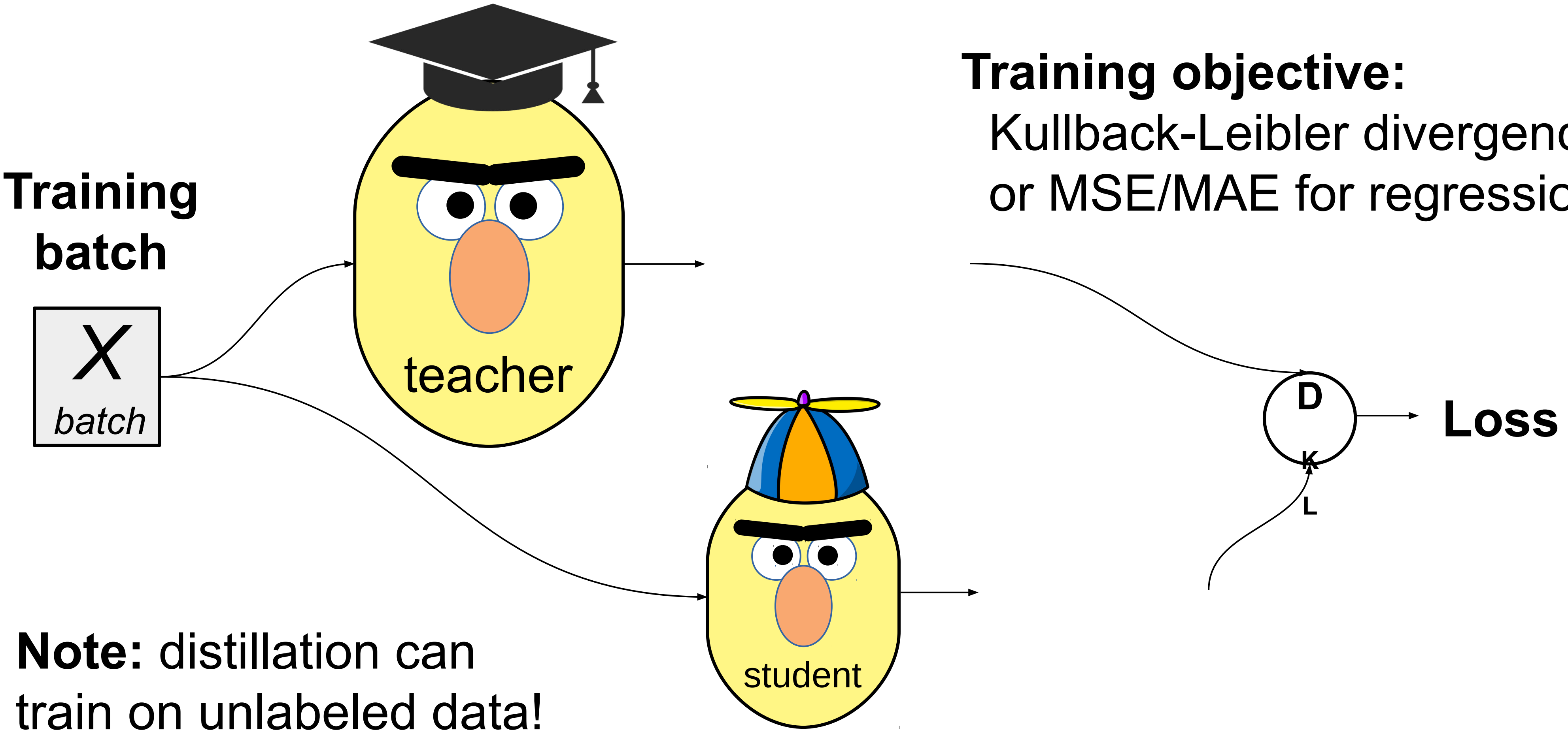
Compression by Distillation

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



Compression by Distillation

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



Note: distillation can
train on unlabeled data!

Compression by Distillation

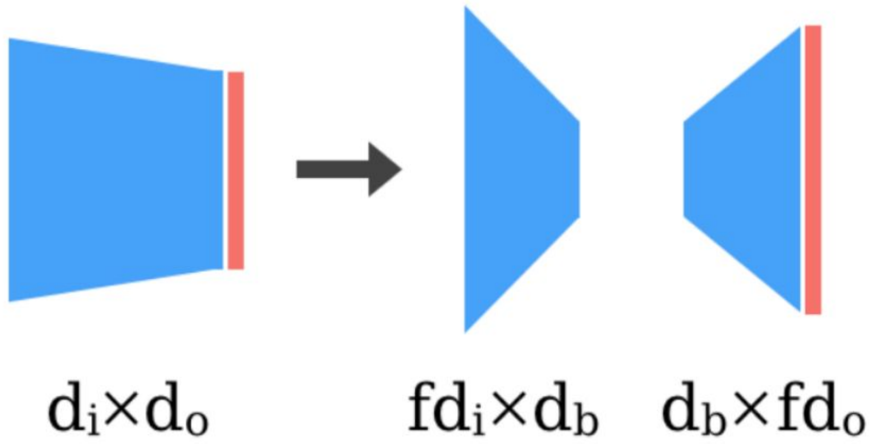
- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- e.g. DistillBERT: https://arxiv.org/pdf/1910.01108.pdf

  - 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

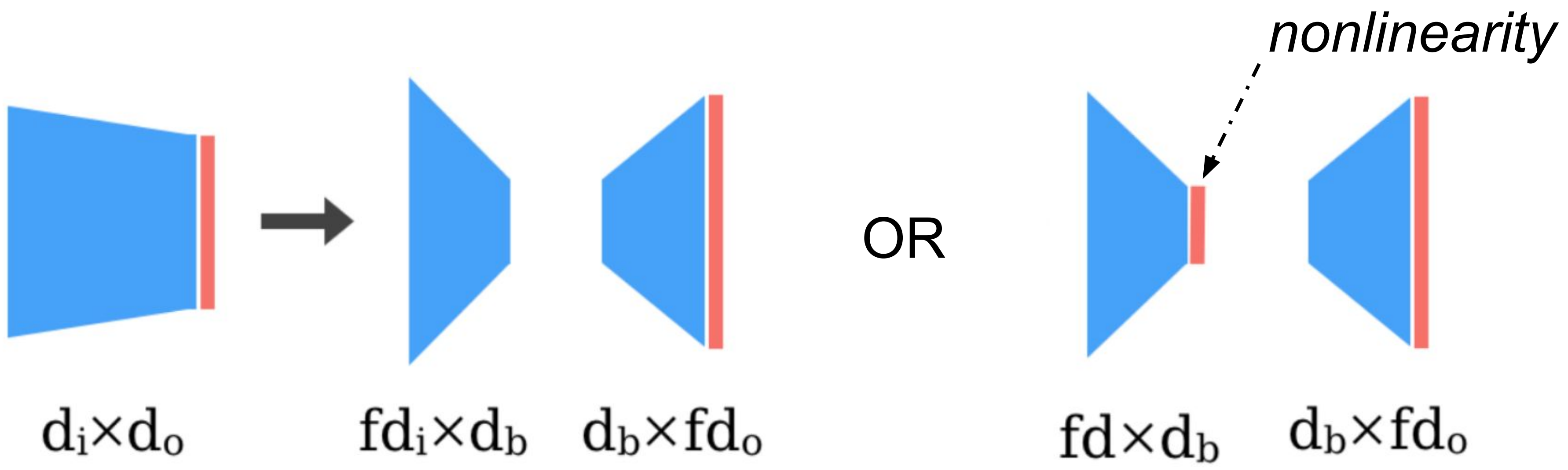
Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- **Factorized:** product of smaller matrices or tensors
- 

Images: https://openreview.net/pdf?id=_zx8Oka09eF

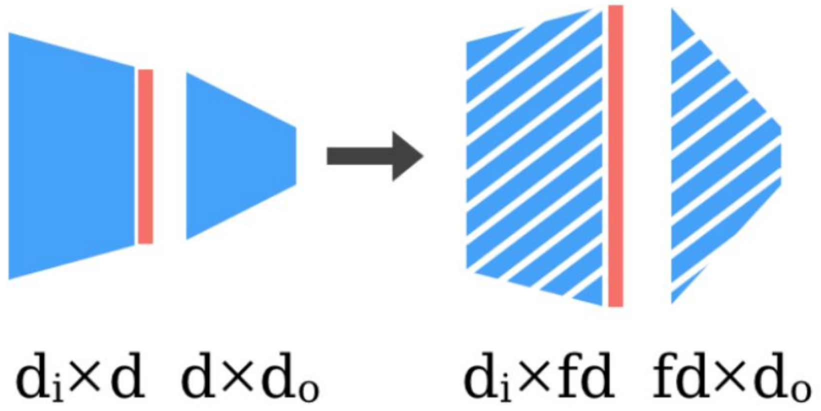
Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- **Factorized:** product of smaller matrices or tensors



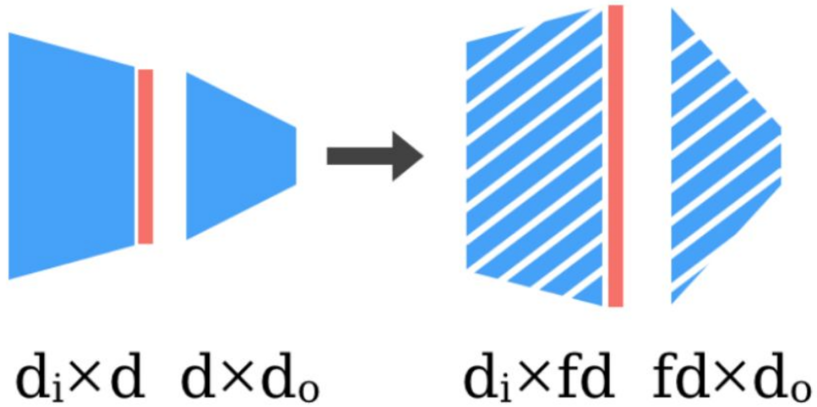
Images: https://openreview.net/pdf?id=_zx8Oka09eF

Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- **Factorized:** product of smaller matrices or tensors
- **Sparse:** only a small (random) subset of weights are nonzero
- 

Images: https://openreview.net/pdf?id=_zx8Oka09eF

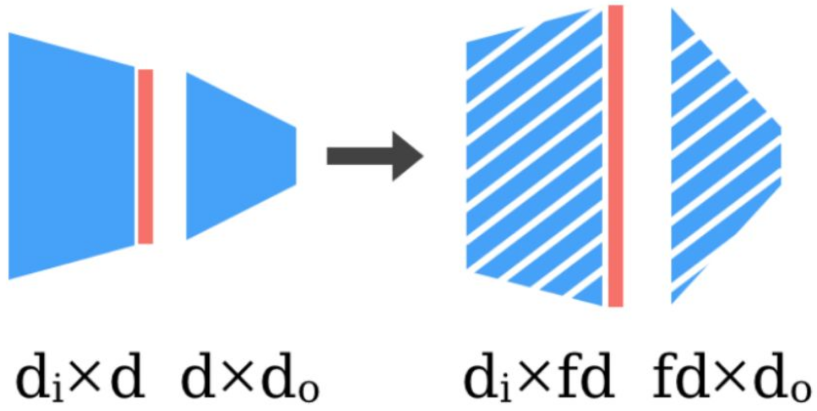
Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- **Factorized:** product of smaller matrices or tensors
- **Sparse:** only a small (random) subset of weights are nonzero
- 

Q: how to store sparse weights?

Images: https://openreview.net/pdf?id=_zx8Oka09eF

Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- **Factorized:** product of smaller matrices or tensors
- **Sparse:** only a small (random) subset of weights are nonzero
- 

Storage: only store random seed and nonzero weights.

Compute: sparse matrix multiply

Images: https://openreview.net/pdf?id=_zx8Oka09eF

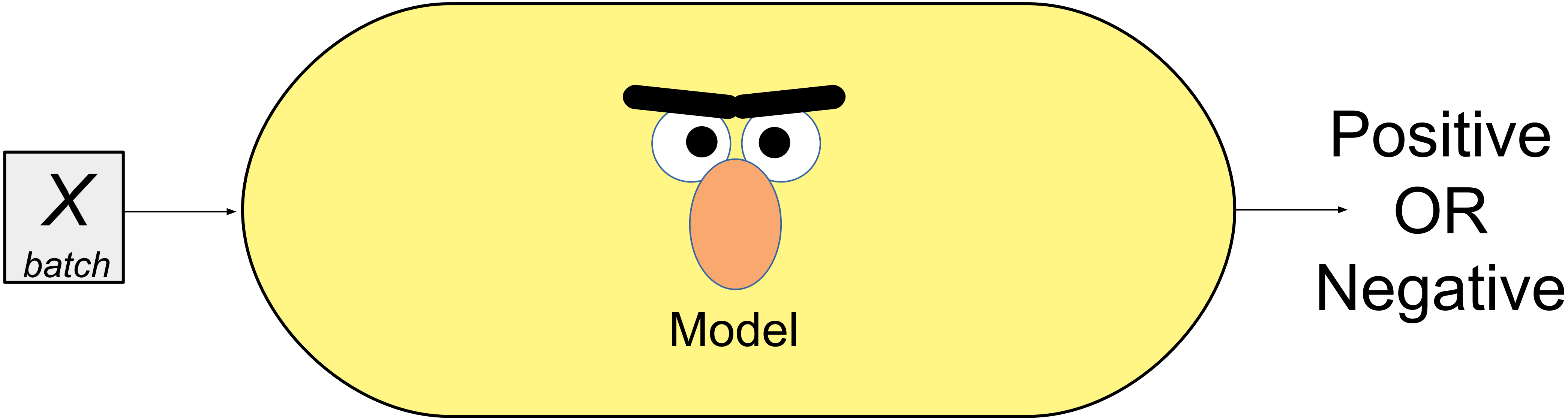
Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- **Factorized:** product of smaller matrices or tensors
- **Sparse:** only a small fraction of weights are nonzero
- **Read more:** https://openreview.net/pdf?id=_zx8Oka09eF
- **Also:** factorized embeddings https://arxiv.org/abs/1901.10787
- **Also also:** small-world sparse weights graphs for RNNs
  - https://tinyurl.com/openai-blocksparse
-

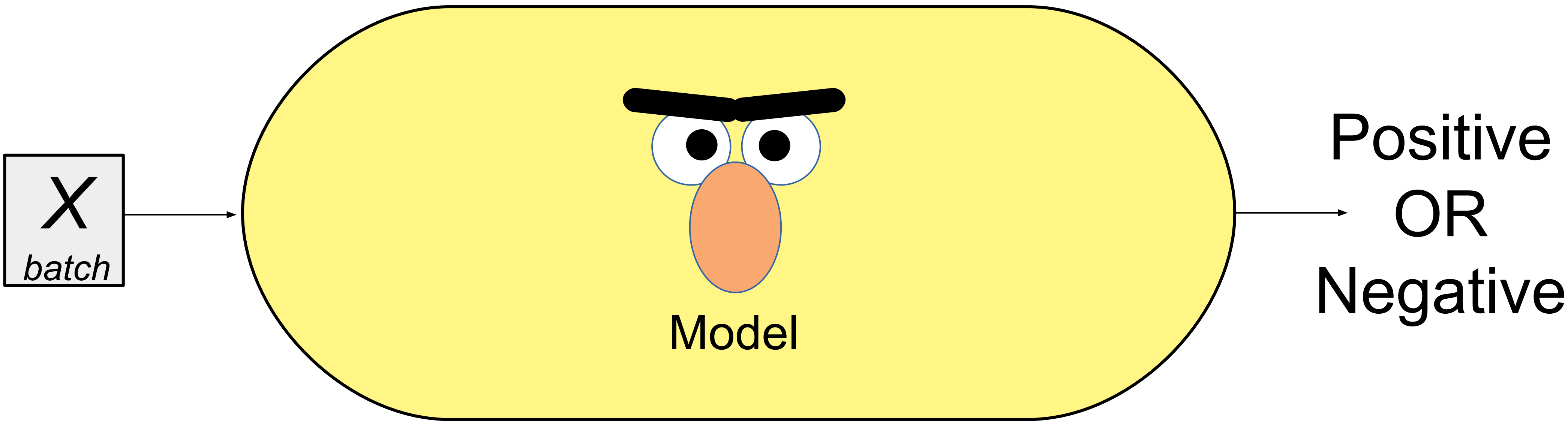
Compression by Distillation

- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
- **Factorized:** product of smaller matrices or tensors
- **Sparse:** only a small fraction of weights are nonzero
- **Read more:** https://openreview.net/pdf?id=_zx8Oka09eF
- **Also:** factorized embeddings https://arxiv.org/abs/1901.10787
  Also also: https://tinyurl.com/openai-blocksparse
- 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

Acceleration by cascading

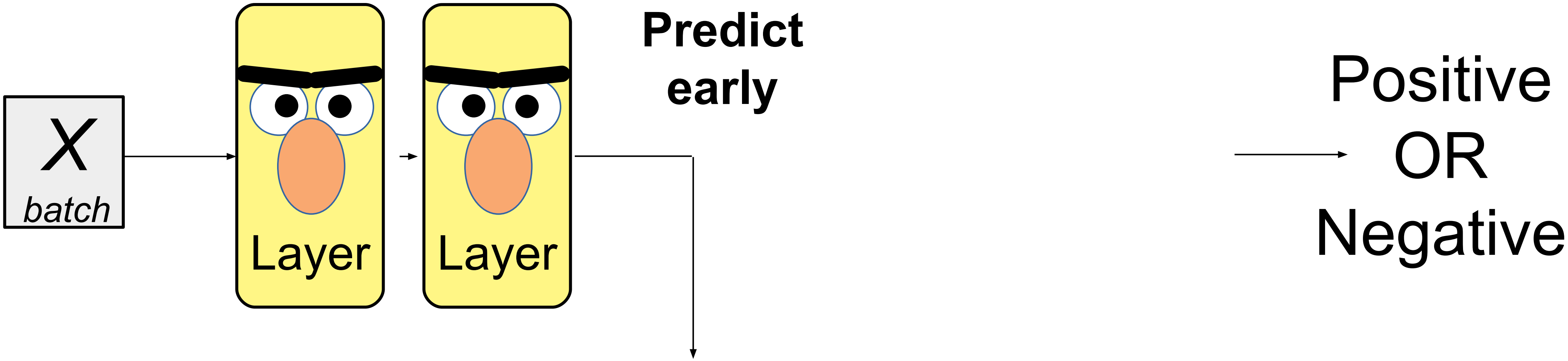


Acceleration by cascading



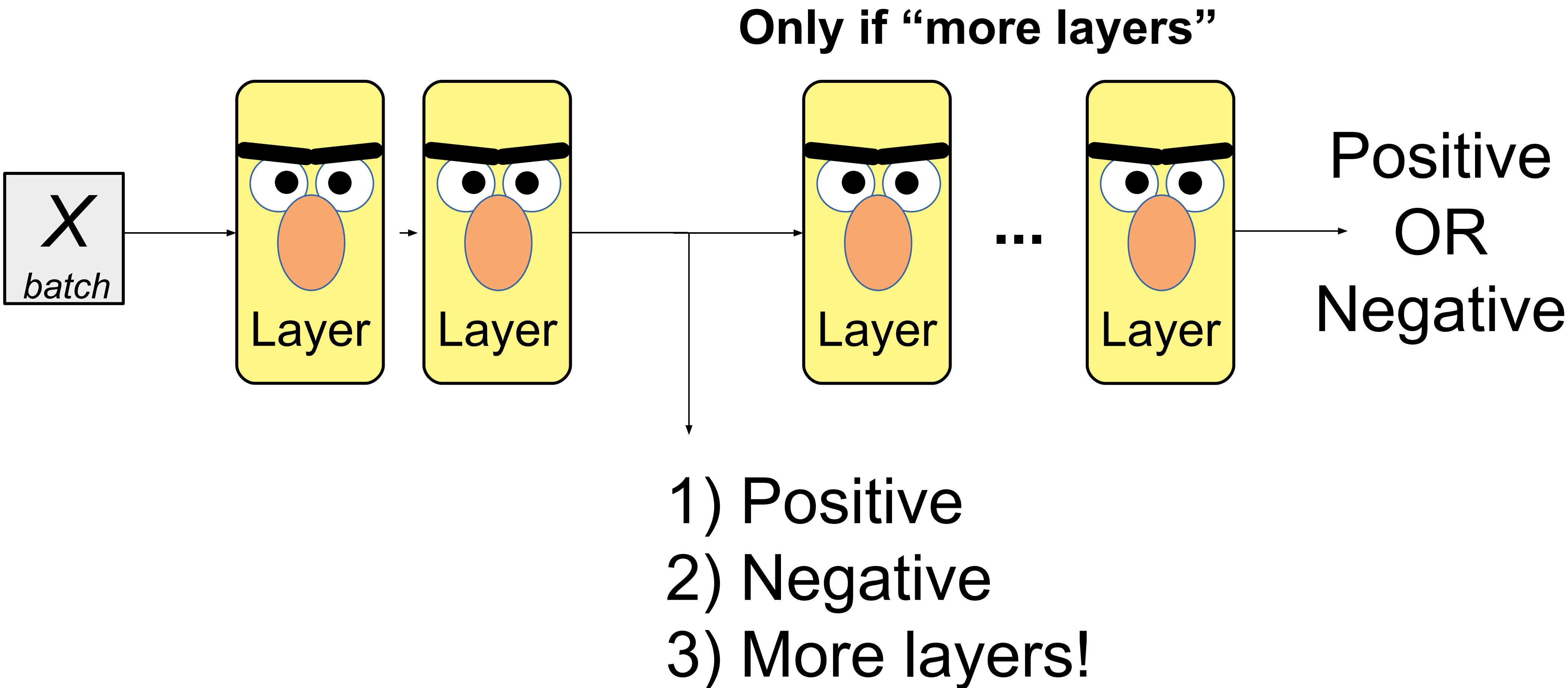
Do we really need every layer
all the time?

Acceleration by cascading

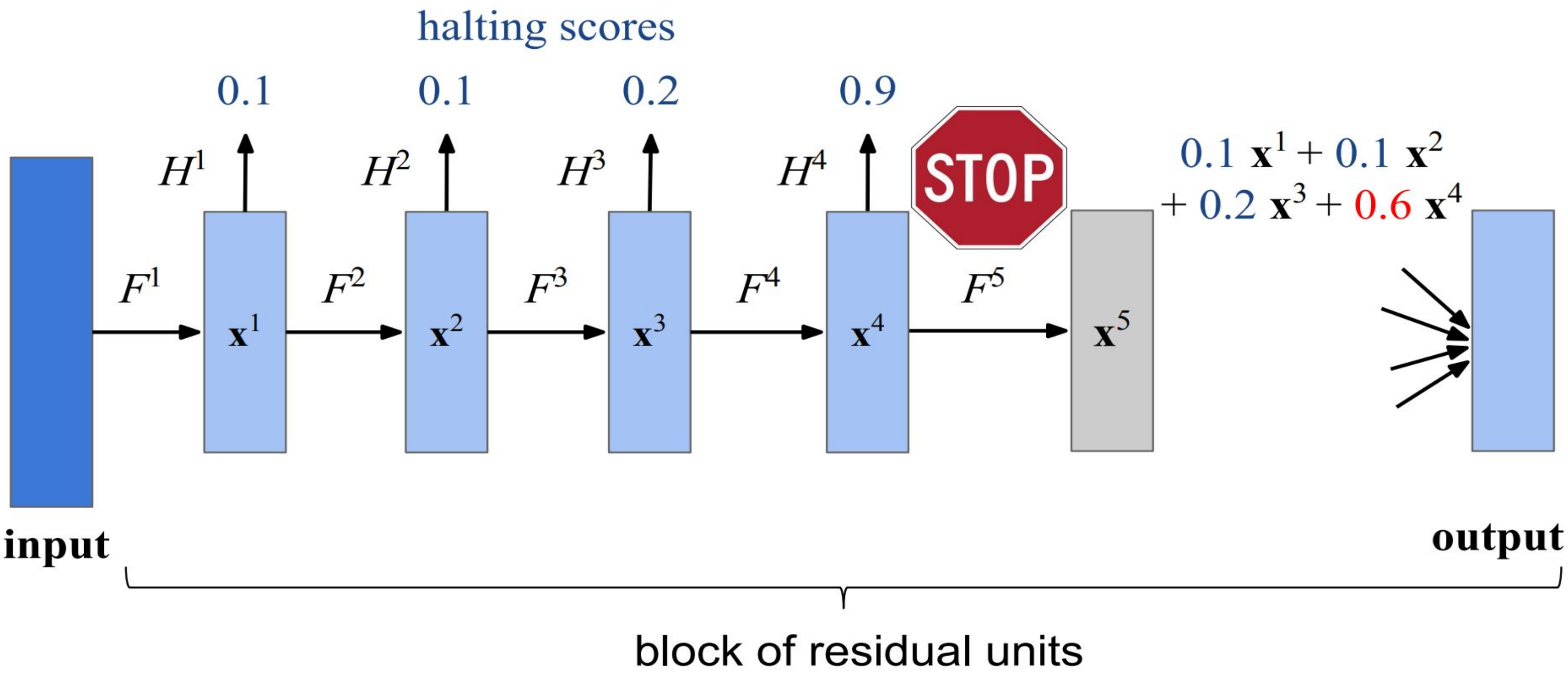


1) Positive
2) Negative
3) More layers!

Acceleration by cascading



Adaptive Computation Time



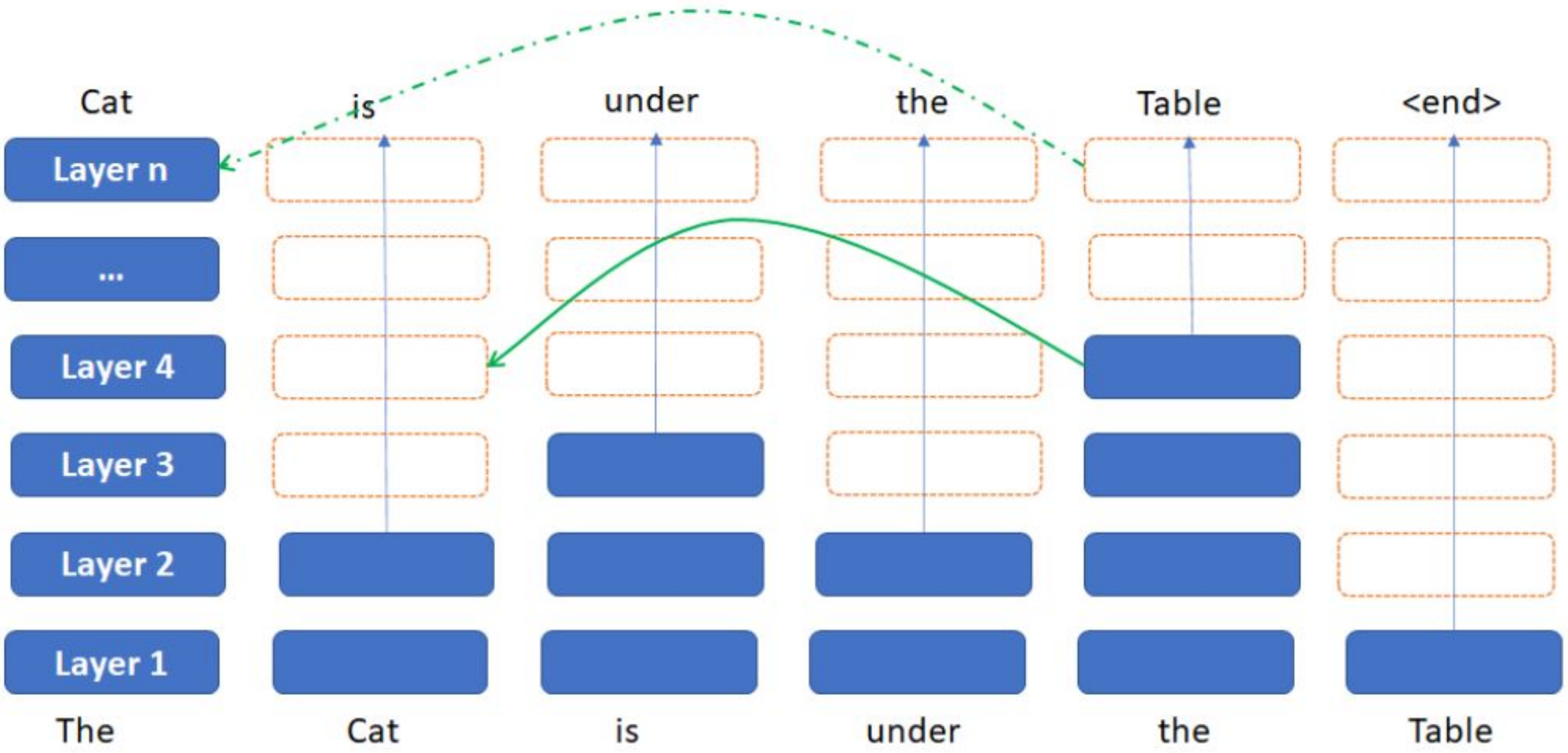
Origina ACTI (for RNN)
https://arxiv.org/abs/1603.08983

Spatial ACT (conv)
https://tinyurl.com/sact-pdf

ACT Transformers
https://arxiv.org/abs/1807.03819

ACT-like metods for LLMs

https://arxiv.org/abs/2207.07061
https://arxiv.org/abs/2307.02628



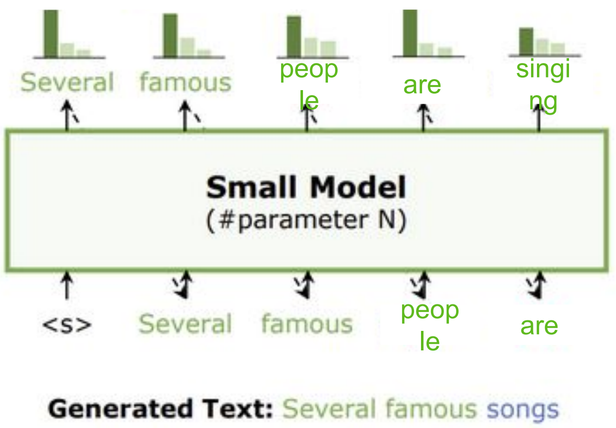
Speculative decoding

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)



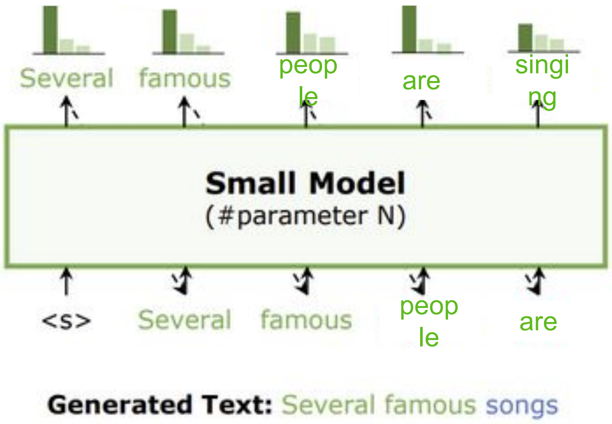
Speculative decoding

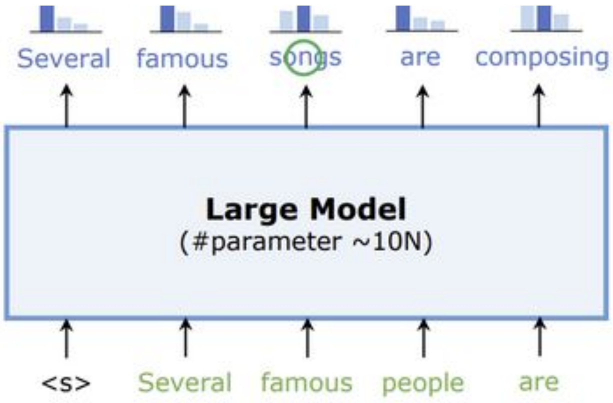
https://arxiv.org/abs/2211.17192

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

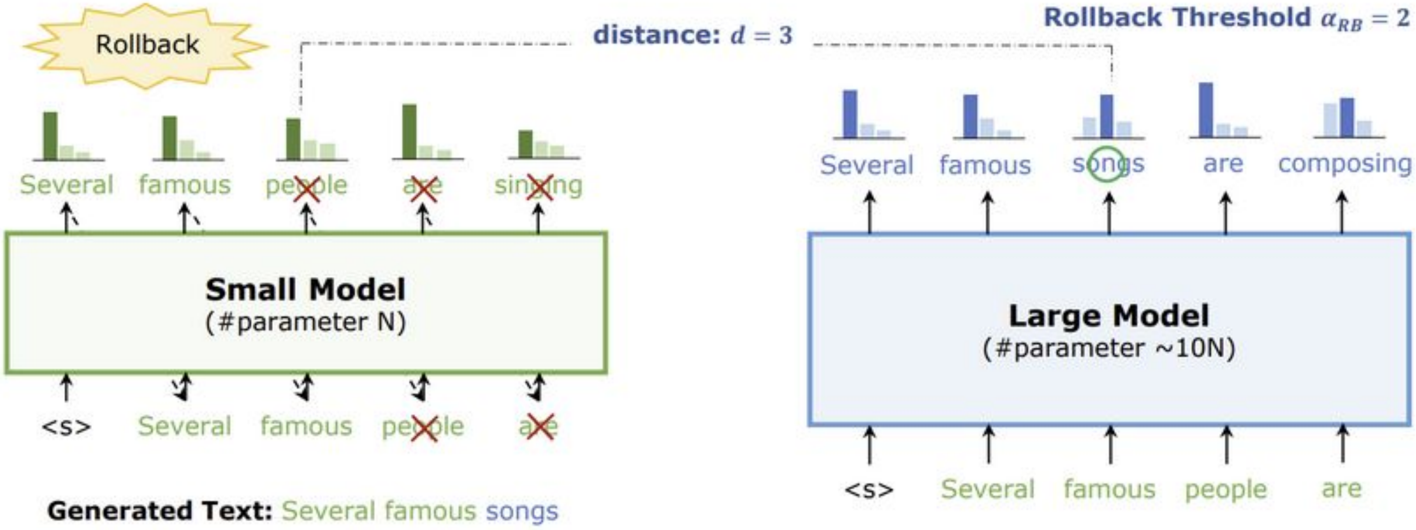
Two models: main model (LLaMA-70B) and draft model (LLaMA-7B)

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)

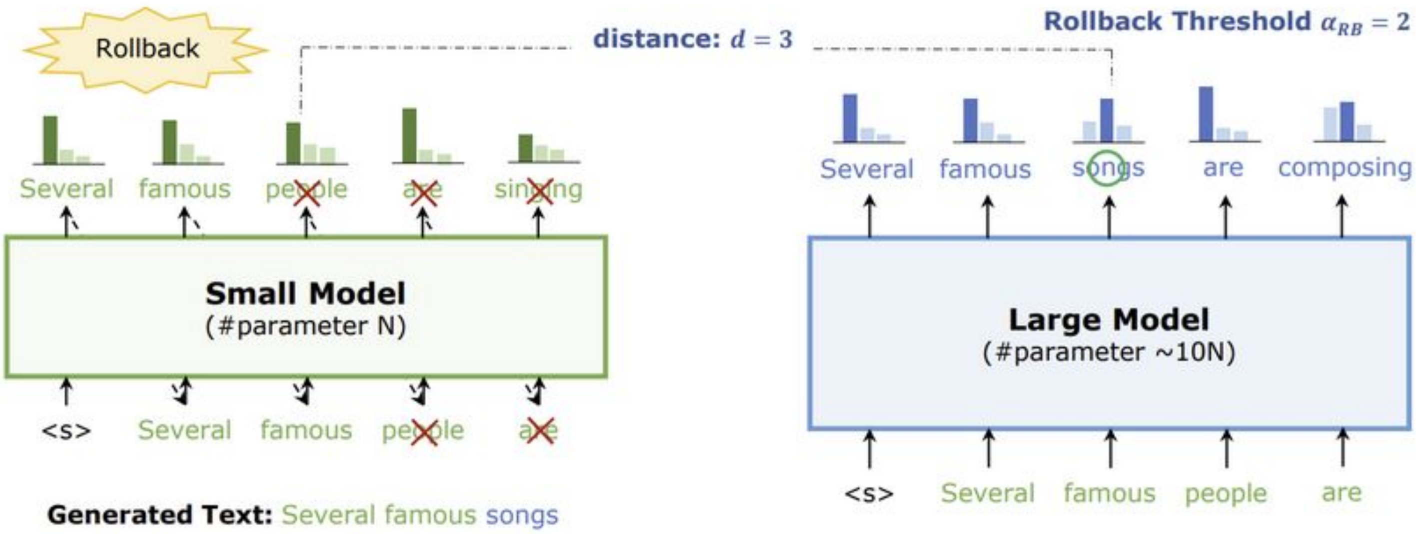
Greedy decoding:

Step 1: generate with draft model (sequential)

Step 2: verify with large model (parallel)

Accept multiple tokens

Repeat

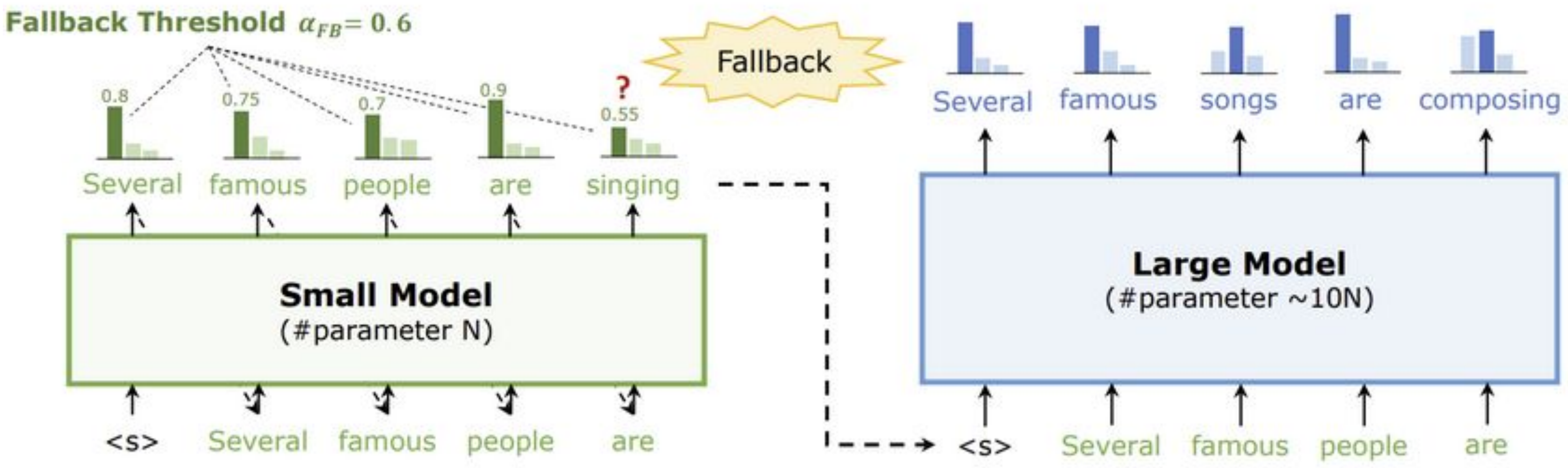


Speculative decoding

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



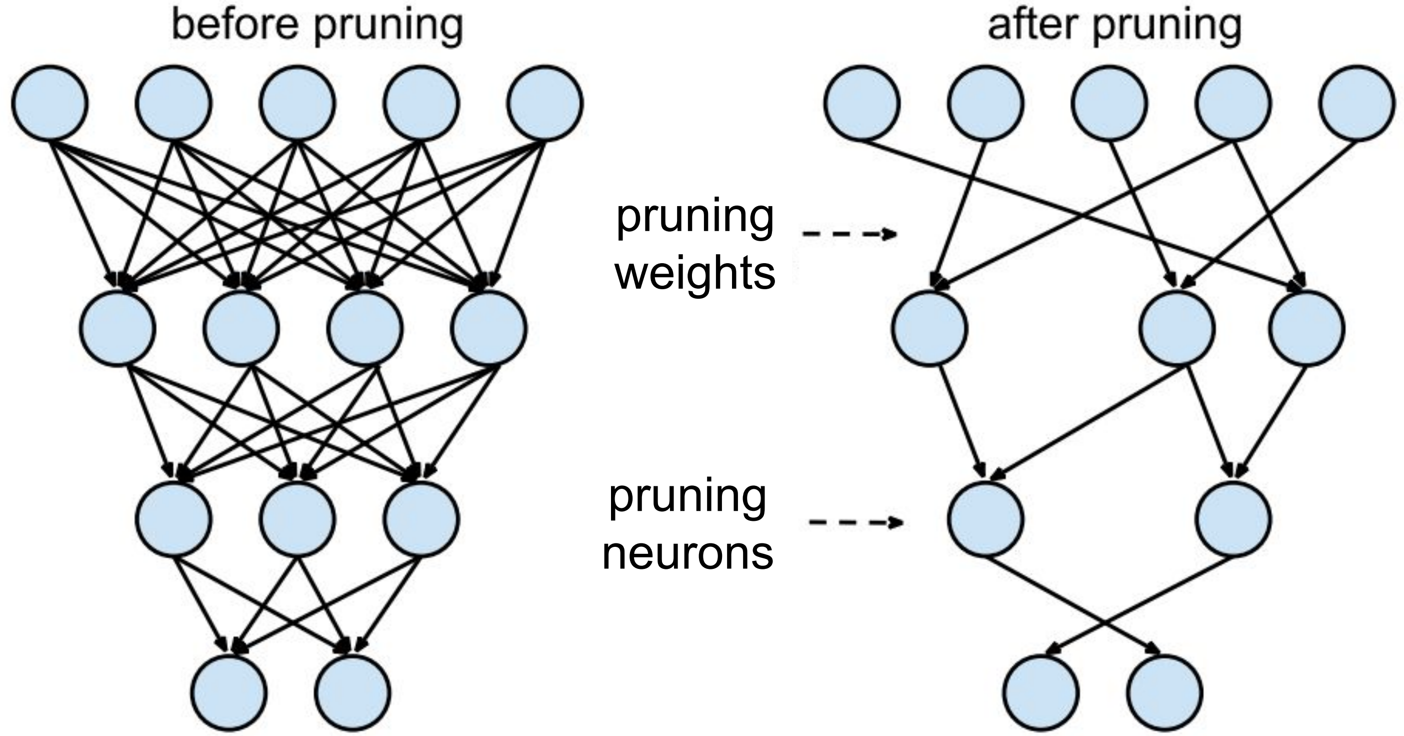
Going inside model layers

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)

Reminds you of something?

Magnitude pruning

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

Reminds you of something?

See ML course, Optimal Brain Damage

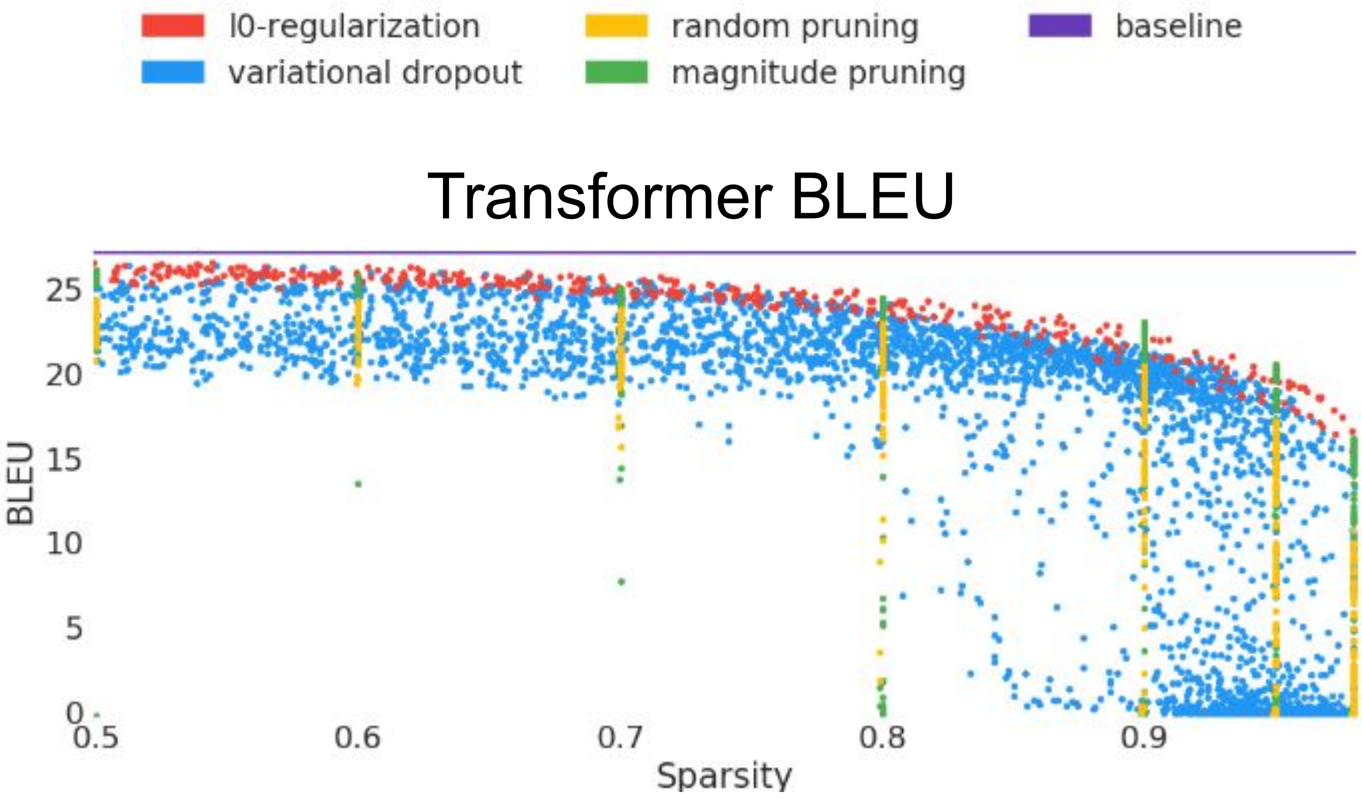
Pruning with L_0 regularization

Add a special regularizer that encourages dropping unnecessary weights

Whiteboard time!

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

Which one works best?



Source https://arxiv.org/abs/1902.09574

Pruning with L_0 regularization

Add a special regularizer that encourages dropping unnecessary weights

Whiteboard time!

Pruning with L_0 regularization

Add a special regularizer that encourages dropping unnecessary weights

- Can prune
  - individual weights
  - Individual neurons
  - attention heads
  - entire layers!

$\lambda = 0.01$

Pruning heads: https://lena-voita.github.io/posts/acl19_heads.html

Compression by sparsification

Unstructured sparsity = prune individual weights
(minimal model size)

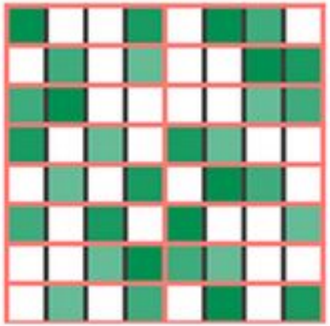
Structured sparsity= prune entire neurons/heads
(fastest inference)

Compression by sparsification

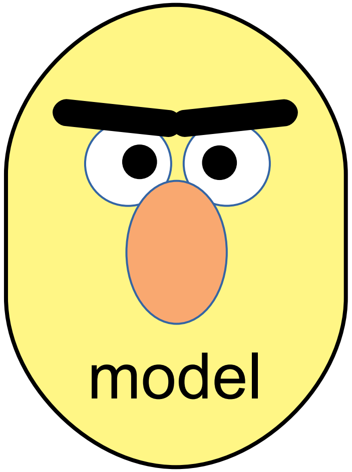
Unstructured sparsity = prune individual weights
(minimal model size)

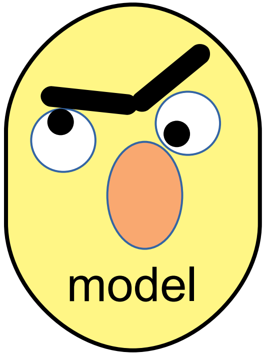
Structured sparsity= prune entire neurons/heads
(fastest inference)

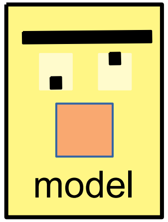
Note: some GPU/FPGAs also run fast with low-level structured sparsity, e.g. “Any 2 of 4 consecutive weights” (left)



Compression by quantization





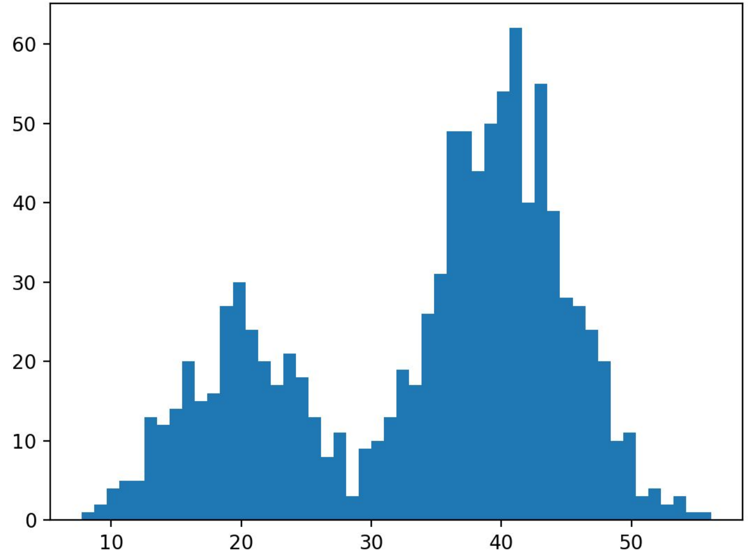


INT8

8 BITS

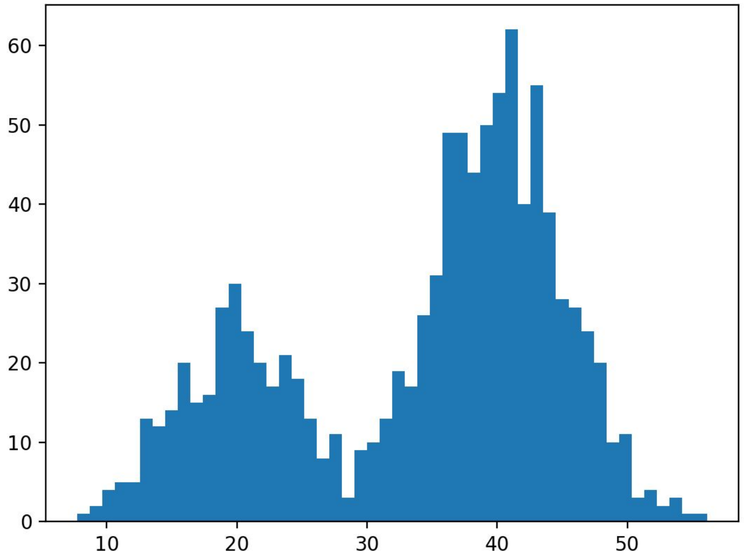
Quantization basics

Goal: encode data as int8 / int4



Quantization basics

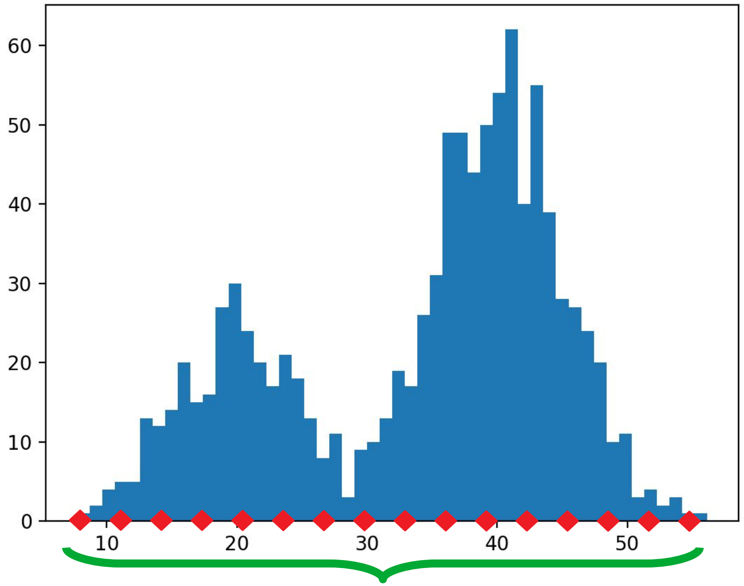
Goal: encode data as int8 / int4



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

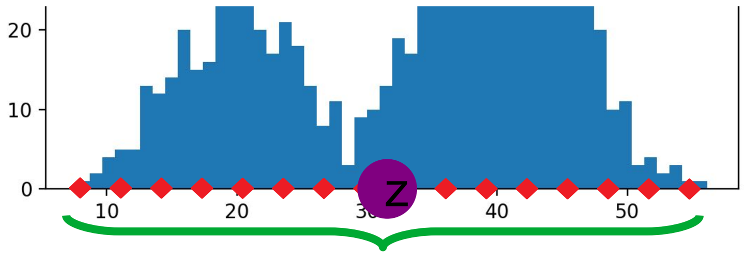
Linear quantization

Fit a linear range to data

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

uint4 range

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



scale = (max(w) – min(w)) / 2^4

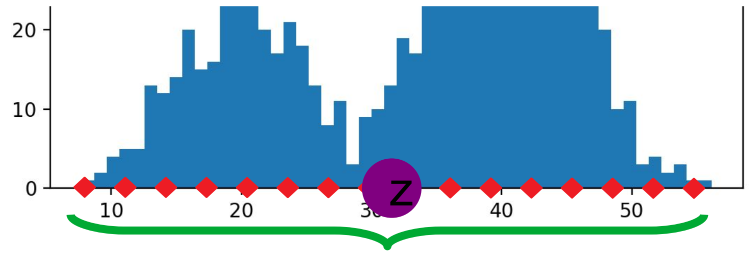
zero = -min(w) / scale

Linear quantization

Fit a linear range to data

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

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

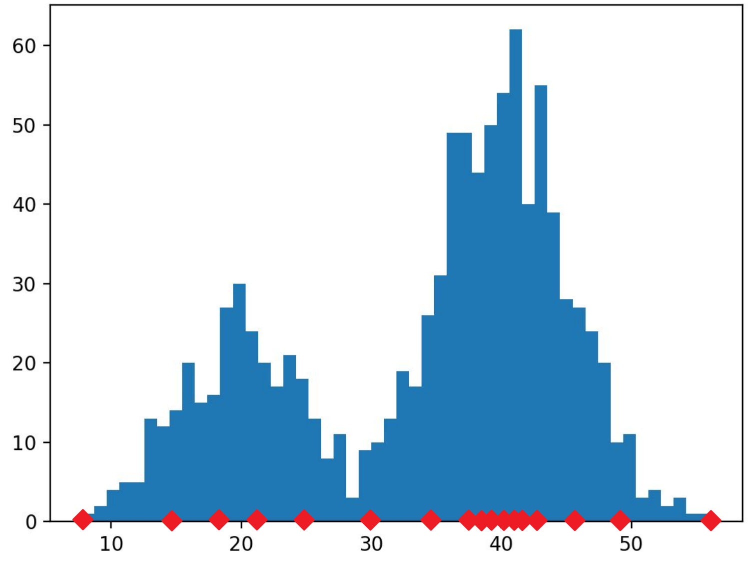


scale = (max(w) – min(w)) / 2^4

zero = -min(w) / scale

Nonlinear quantization

Fit codes to data

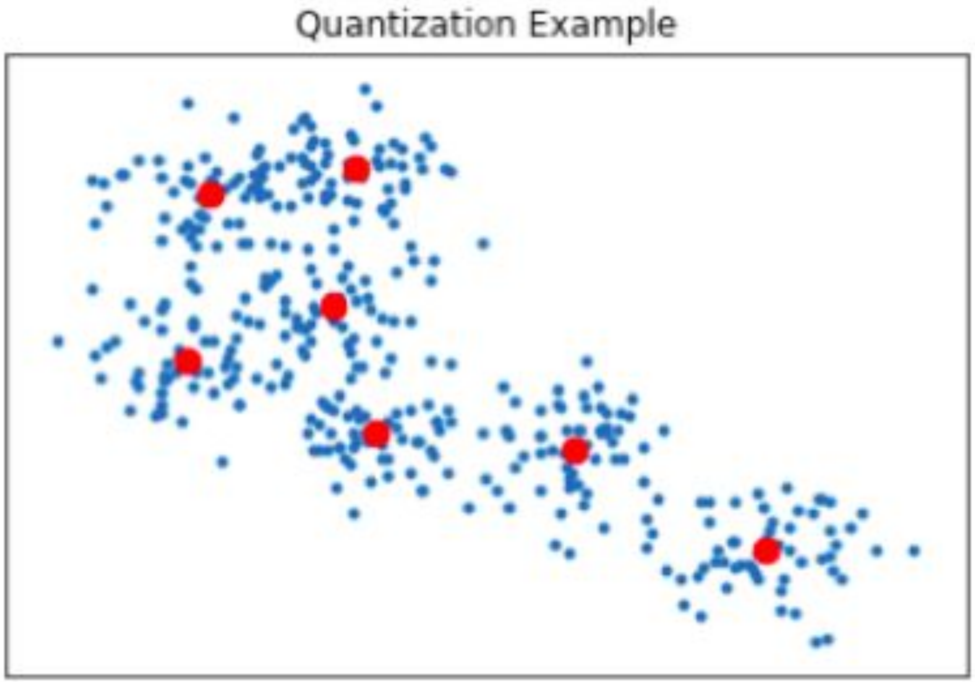


Compute a grid of percentiles or centroids (k-means 1d)

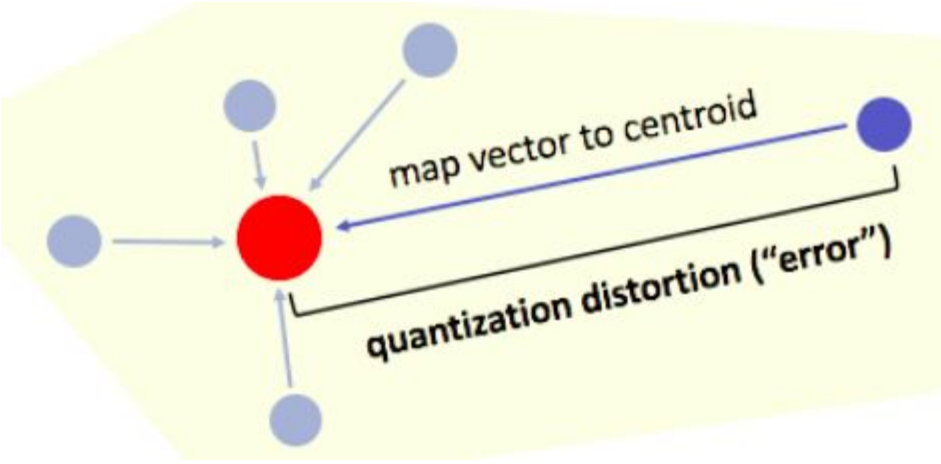
Store each weight as the index of nearest percentile/centroid

High-dimensional case

Quantize entire vectors as K-means



quantizer = KMeans(n_clusters=7).fit(X)



Images: [Jeremy Jordan]

OPQ, AQ, LSQ

Product **Q**uantization

Split vectors into chunks, quantize each chunk separately

Orthogonal **P**roduct **Q**uantization

First run orthogonal transform, then product quantization

http://kaiminghe.com/publications/cvpr13opq.pdf

More:

Additive **Q**uantization https://tinyurl.com/babenko-aq-pdf

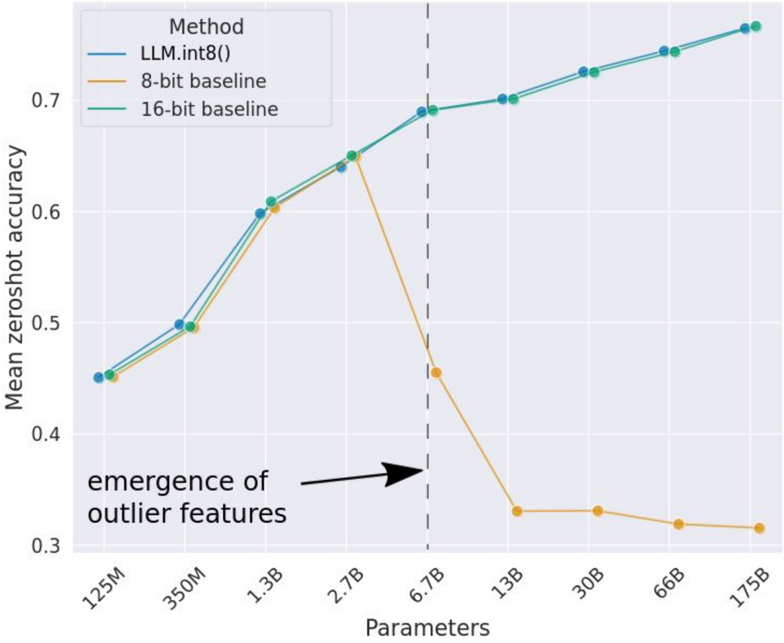
Local **S**earch **Q**uantization https://tinyurl.com/martinez-lsq-pdf

Images: Jeremy Jordan

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

https://arxiv.org/abs/2208.07339

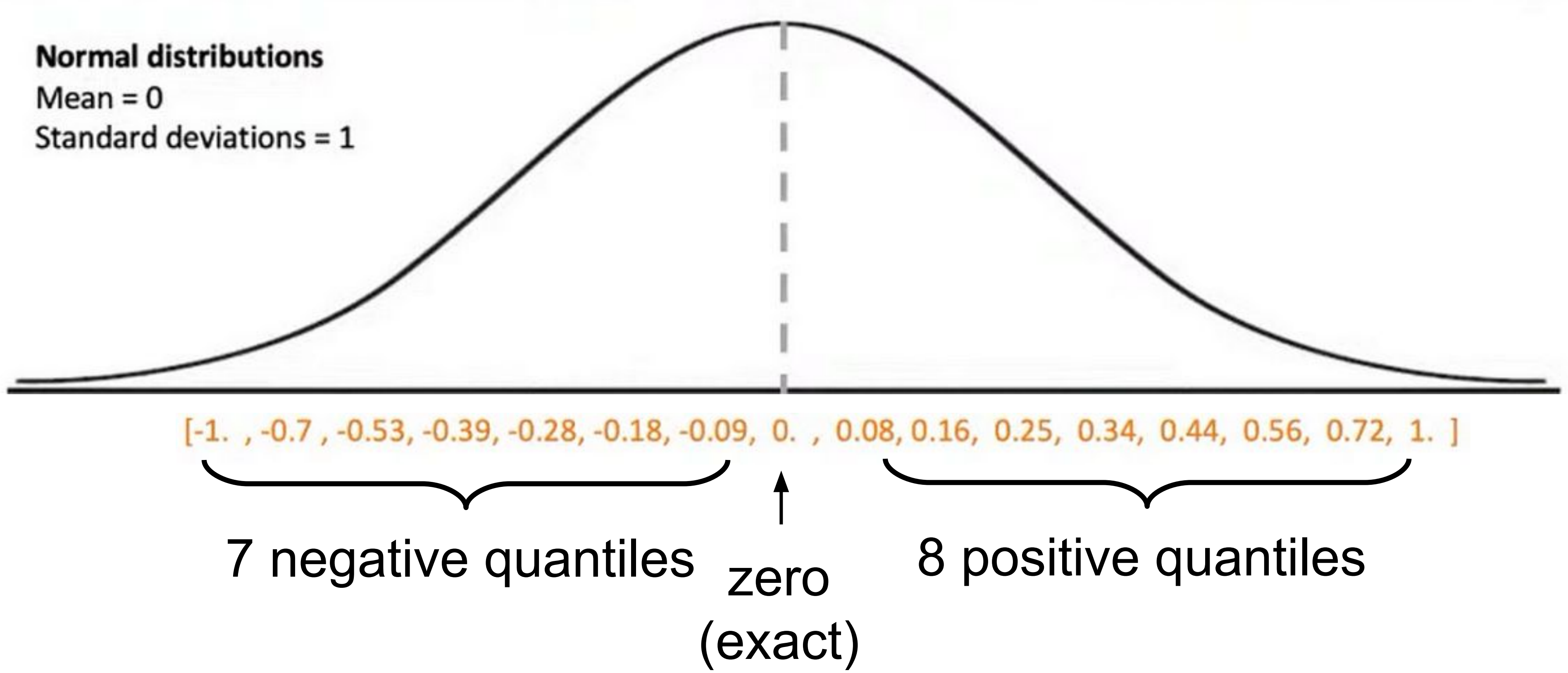
TL;DR in very LLM, some input features become outliers
Weights for those features are sensitive
KEEP <1% MOST SENSITIVE WEIGHTS IN 16-bit!



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)
```

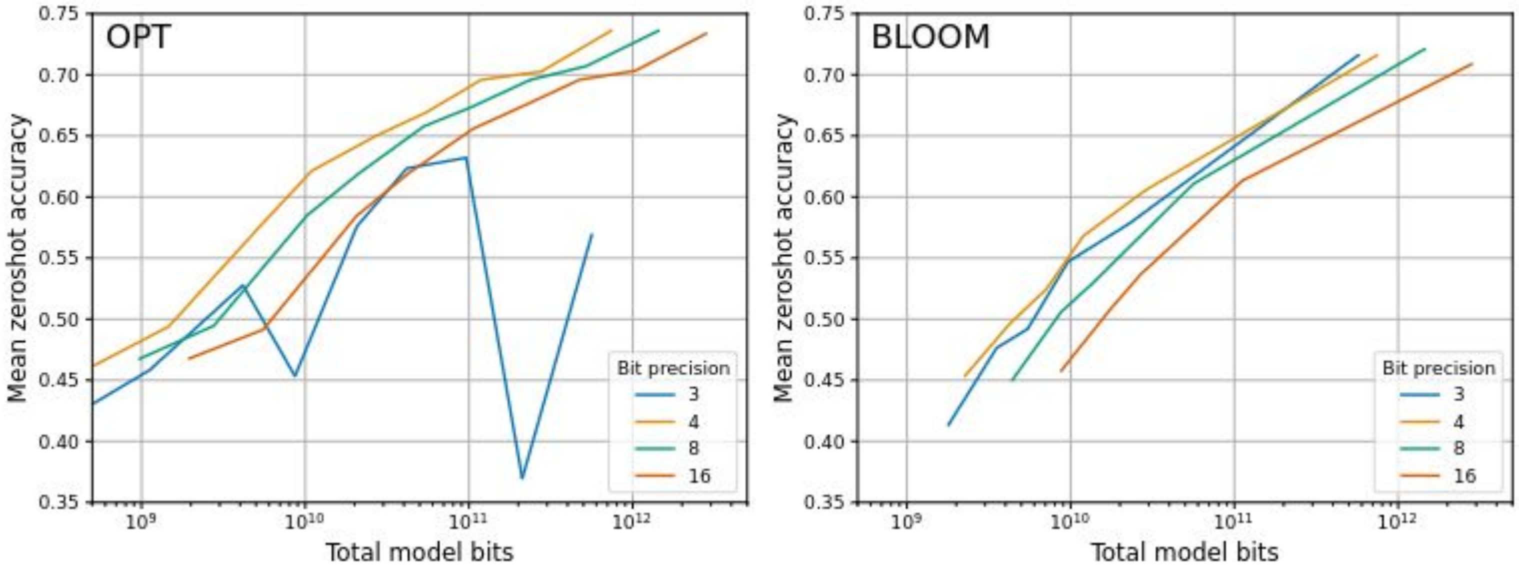
[-1. , -0.7 , -0.53, -0.39, -0.28, -0.18, -0.09, 0. , 0.08, 0.16, 0.25, 0.34, 0.44, 0.56, 0.72, 1.]

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: 100B in 2-bit often worse than 50B in 4-bit



Model compression landscape

Goal: faster / smaller / both

Compression: quantize / prune / factorize

Setup: no data, some data, training data

Model compression landscape

Goal: faster / smaller / both

Compression: quantize / prune / factorize

Setup: no data, some data, training data

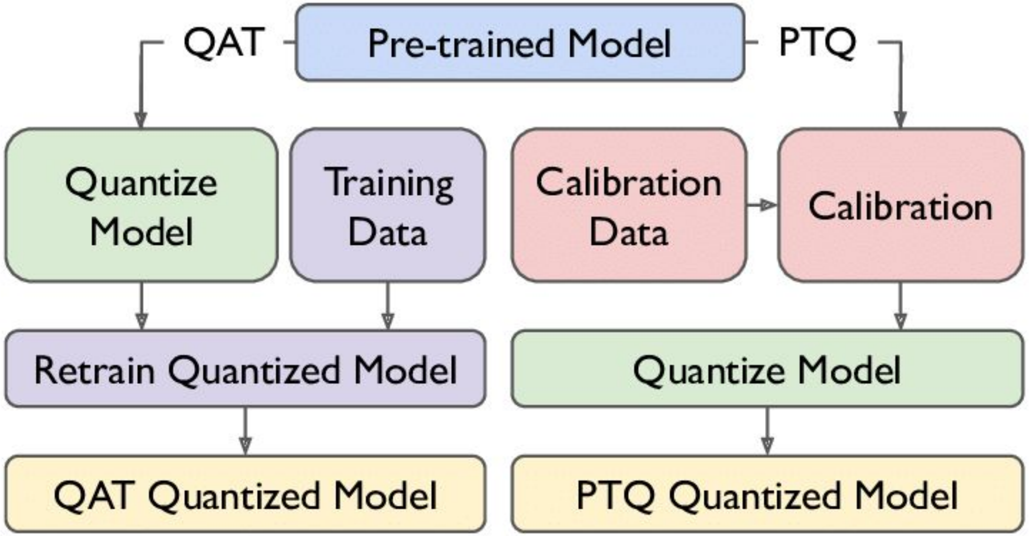
Q: can we take advantage of data
to improve quantization?

Model compression landscape

Goal: faster / smaller / both

Compression: quantize / prune / factorize

Setup: no data, some data, training data



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

Optimal Brain Damage

[Le Cun et al, 1989](#)

Approximate loss function with Taylor series up to squared term

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta \mathbf{w} + \frac{1}{2}\delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + O(\| \delta \mathbf{w} \|^3) \quad (1)$$

Note: this Taylor expansion assumes that model is trained to the exact minimum of loss function, i.e. initial gradients are zero

Optimal Brain Damage

Le Cun et al, 1989

Approximate loss function with Taylor series up to squared term

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta \mathbf{w} + \frac{1}{2} \delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + O(\| \delta \mathbf{w} \|^3) \quad (1)$$

increase in error

weight change

loss hessian

Optimal Brain Damage

[Le Cun et al, 1989]

Approximate loss function with Taylor series up to squared term

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta \mathbf{w} + \frac{1}{2}\delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + O(\| \delta \mathbf{w} \|^3) \quad (1)$$

increase in error

weight change

loss hessian

Find Δw that minimizes the error **AND** (w+Δw) is sparse or quantized

Optimal Brain Surgeon

Hassibi et al, 1993

Approximate loss function with Taylor series up to squared term

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta \mathbf{w} + \frac{1}{2}\delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + O(\| \delta \mathbf{w} \|^3) \quad (1)$$

Repeat steps 1-2 until model is fully quantized / pruned
Step 1: find Δw to sparsify or quantize a small portion of weights
Step 2: update the remaining (dense fp32) weights to compensate

Optimal Brain Surgeon

[Hassibi et al, 1993](#)

Approximate loss function with Taylor series up to squared term

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta \mathbf{w} + \frac{1}{2}\delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + O(\|\delta \mathbf{w}\|^3) \quad (1)$$

Repeat steps 1-2 until model is fully quantized / pruned
Step 1: find Δw to sparsify or quantize a small portion of weights
Step 2: update the remaining (dense fp32) weights to compensate

Doesn’t work for LLM: hessian is too large

Optimal Brain Surgeon

Frantar et al, 2022

Minimize layer-wise MSE loss $\text{argmin}_{\widehat{\mathbf{W}}_\ell} \quad ||\mathbf{W}_\ell \mathbf{X}_\ell - \widehat{\mathbf{W}}_\ell \mathbf{X}_\ell||_2^2$

Use the same math as Optimal Brain Surgeon

$$\delta E = \left(\frac{\partial E}{\partial \mathbf{w}}\right)^T \cdot \delta \mathbf{w} + \frac{1}{2} \delta \mathbf{w}^T \cdot \mathbf{H} \cdot \delta \mathbf{w} + O(\| \delta \mathbf{w} \|^3) \quad (1)$$

Why:

- hessian H is identical for each neuron: $\mathbf{H} = \mathbf{X} \times \mathbf{X}^T$
- even if model is not trained to convergence, MSE with itself is optimal (and equals zero) – used to simplify Taylor
- still **much** better than rounding

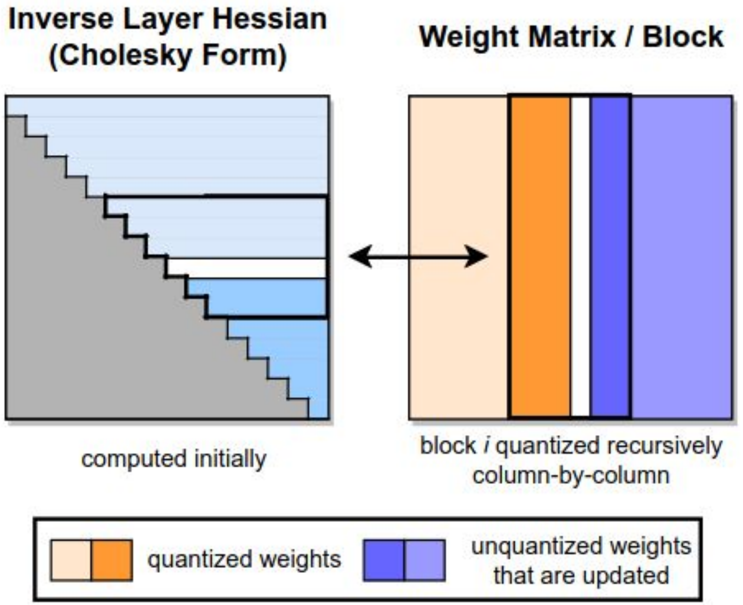
GPTQ

Frantar et al, 2022

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:

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



GPTQ

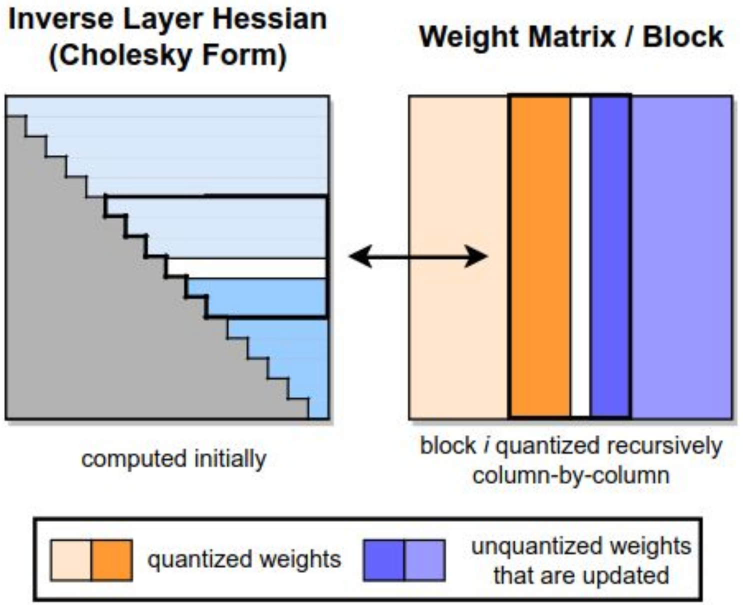
[Frantar et al, 2022](#)

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

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

For i = 1

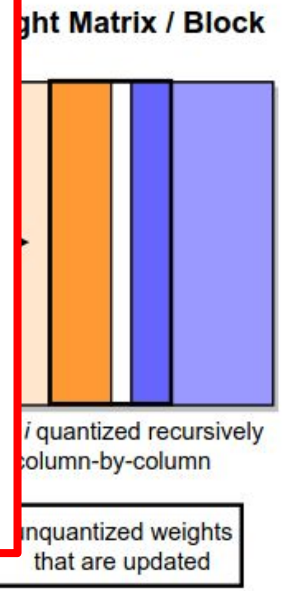
- **quantiz**

(from

- freeze t
- update

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

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



Use linear

zero per ea

Tricks: process weights in “hacky” order

SparseGPT

[Frantar et al, 2023]

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

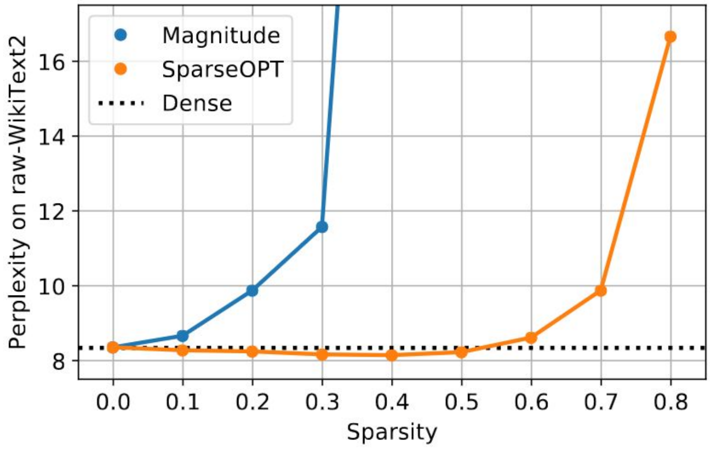
For i = 1 … in_features:

- **sparsify** i-th column of weight matrix
  (from one input feature and all outputs)
- freeze the quantized model forever
- update all remaining columns

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

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

OPT-175B



More quantization papers

SmoothQuant – quantize both weights and activations
https://arxiv.org/abs/2211.10438

SpQR – in gptq, keep some sensitive weights in 16-bit
https://arxiv.org/abs/2306.03078

AWQ – tune scale/zero to better fit sensitive weights
https://arxiv.org/abs/2306.00978

QUIK – sensitive outliers + quantize activations + black magic
https://arxiv.org/abs/2310.09259

Many more cool papers

What did we learn?

Wanna compress an LLM / BERT-like?

TL;DR try quantization first, pruning/factorization later

No time? load_in_4bit=True

Some time? GPTQ / AWQ / SpQR /SparseGPT+Q

A ton of time? Quantization-aware training

Combine with ACT (e.g. CALM) and/or speculative decoding