Natural Language Processing

Episode 7 ‘2021

Model compression & acceleration

Yandex
Research

British Hedgehog
Preservation Society

Chapter 1: why should you care?

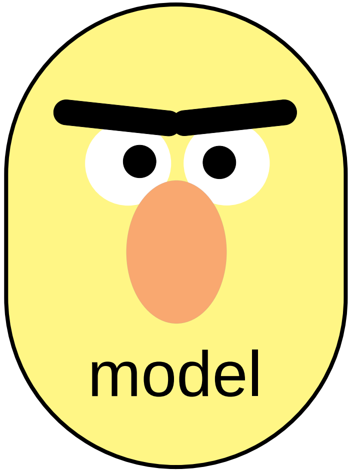
Case study: text classification

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

▶ prepare batch ▶

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

▶



▶

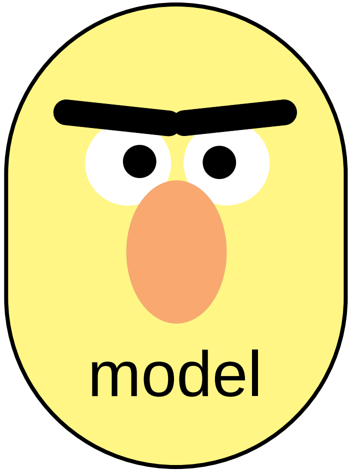
(0) Normal tweet
(1) Should ban

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 Size
(mega)bytes

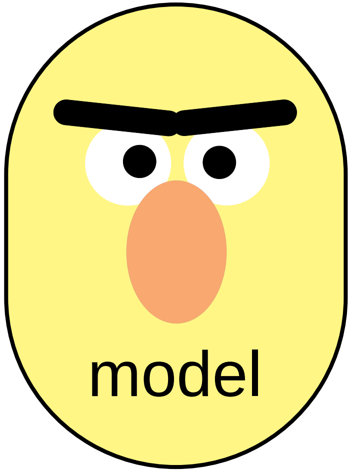
Case study: text classification

"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 Size
(mega)bytes

Throughput
samples/second

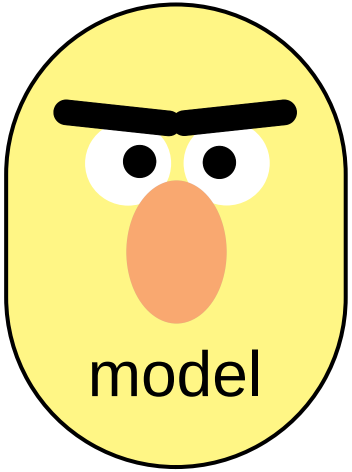
Case study: text classification

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

▶ prepare batch ▶

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

▶



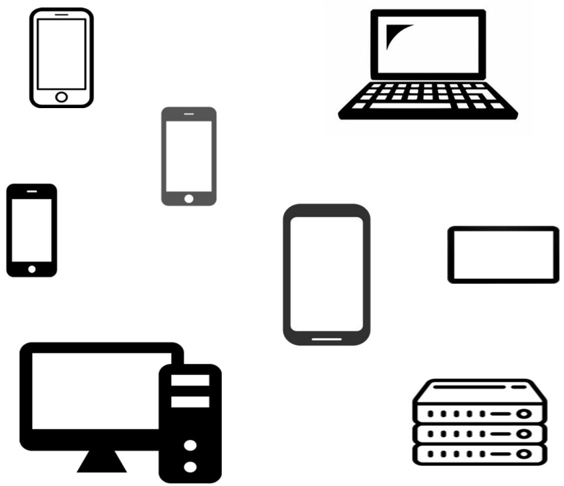
▶

(0) Normal tweet
(1) Should ban

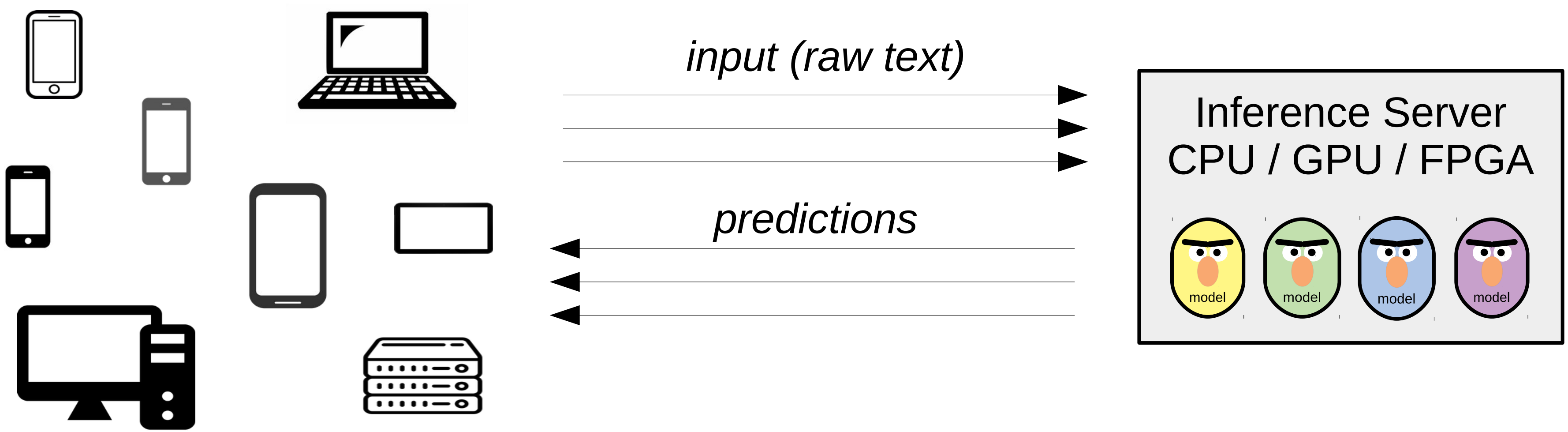
Three kinds of efficiency:

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

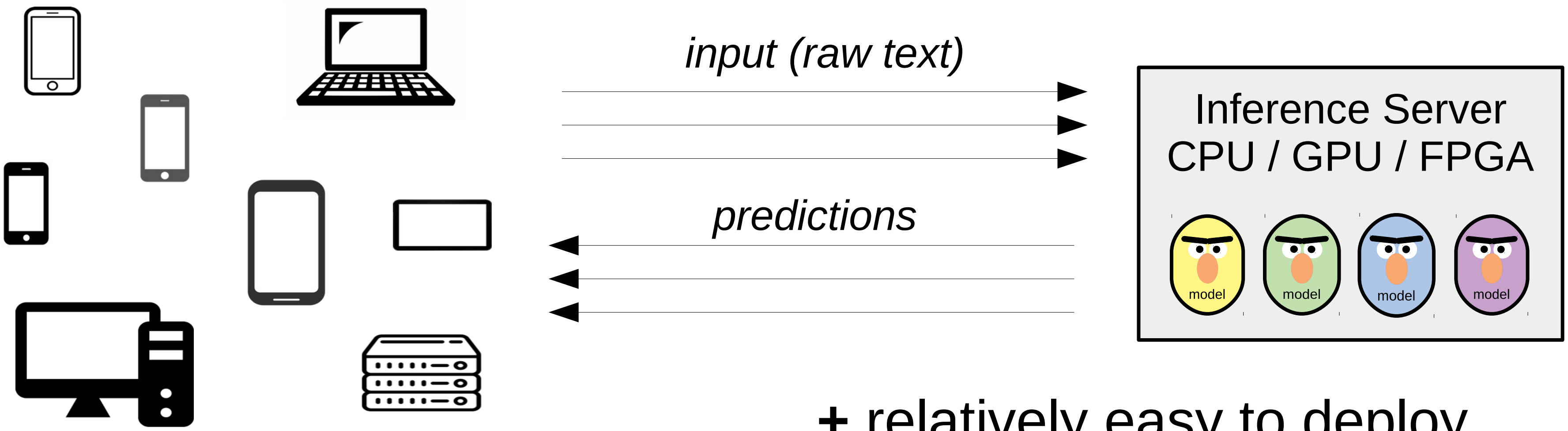
Scenario 1: inference server



Scenario 1: inference server

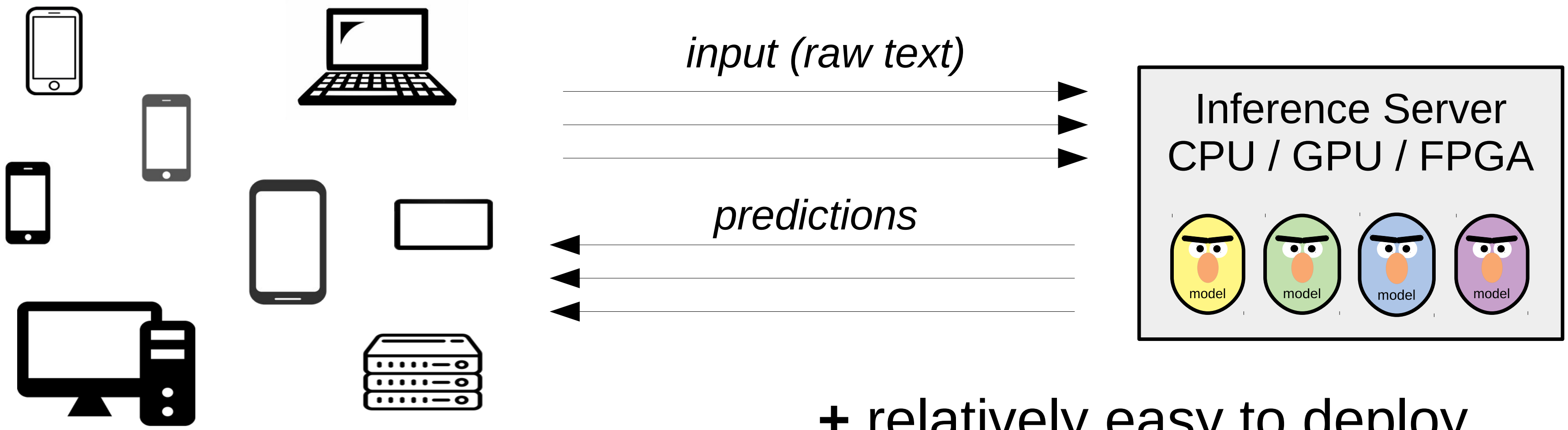


Scenario 1: inference server



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

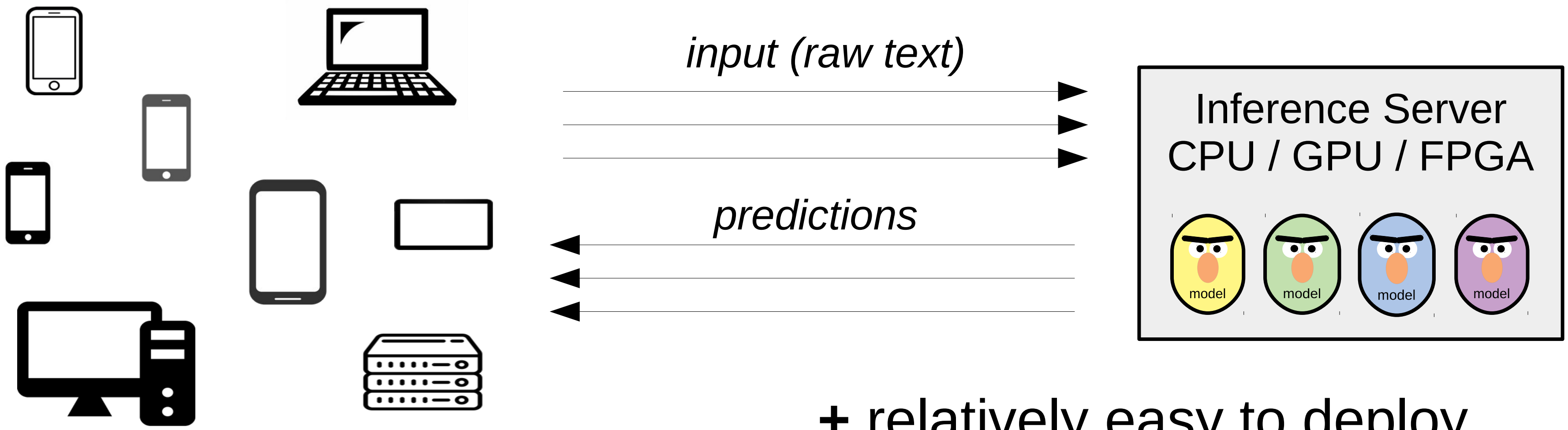
Scenario 1: inference server



+ 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



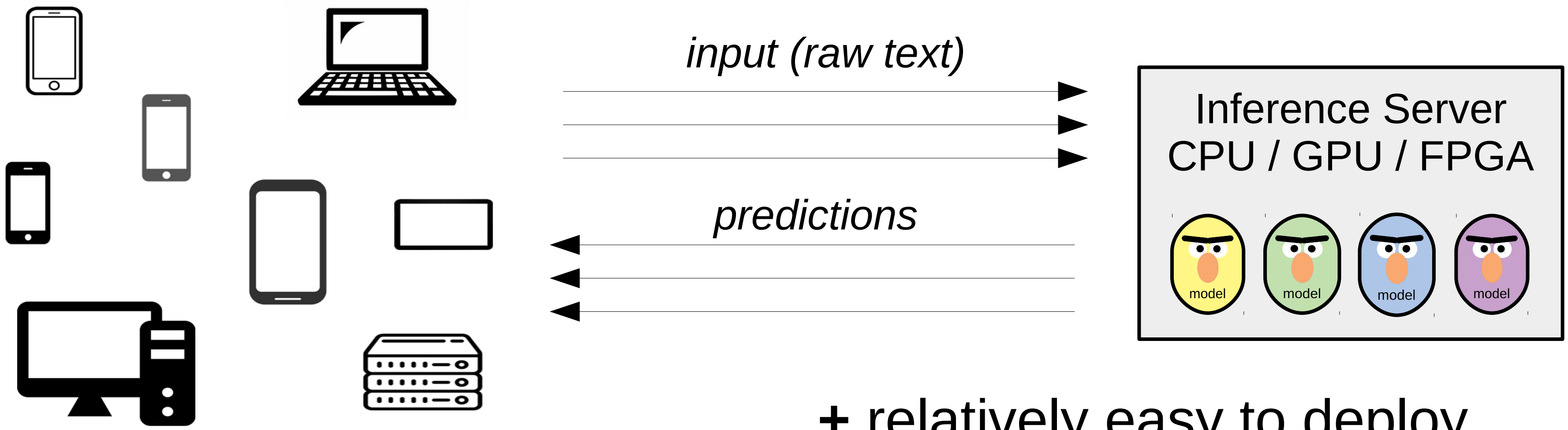
Which is the most important?

 ?  ?  ?

+ 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

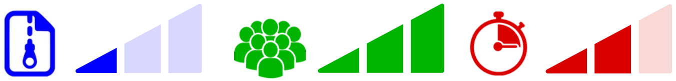
Scenario 1: inference server



+ 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
  improves throughput at the cost of your budget :)

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:

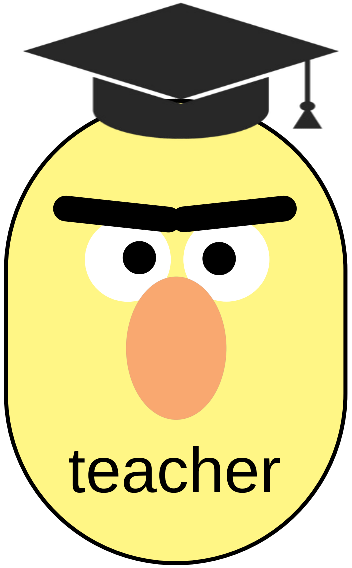
	Platform
TensorFlow.js	All modern browsers
CoreML	iOS devices
NNAPI	Android devices

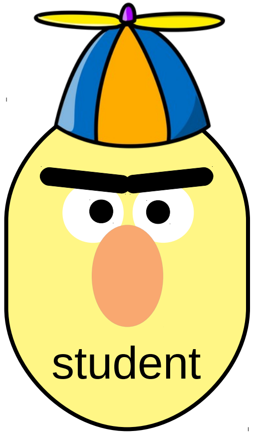
Chapter 2: how do I improve my model?

Compression by Distillation

Distillation…
Heard that word before?

Compression by Distillation

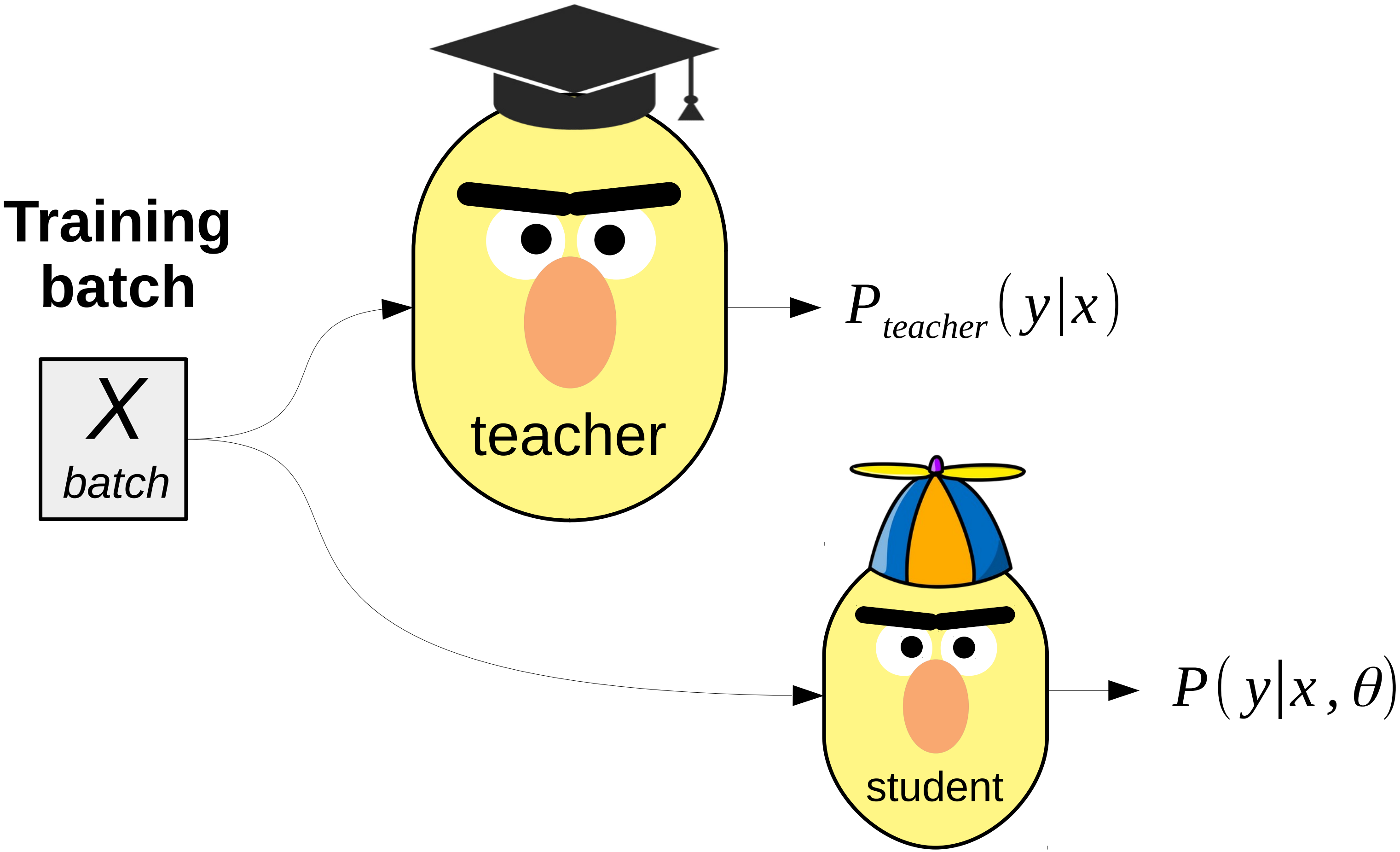




First, get the best performing model regardless of size

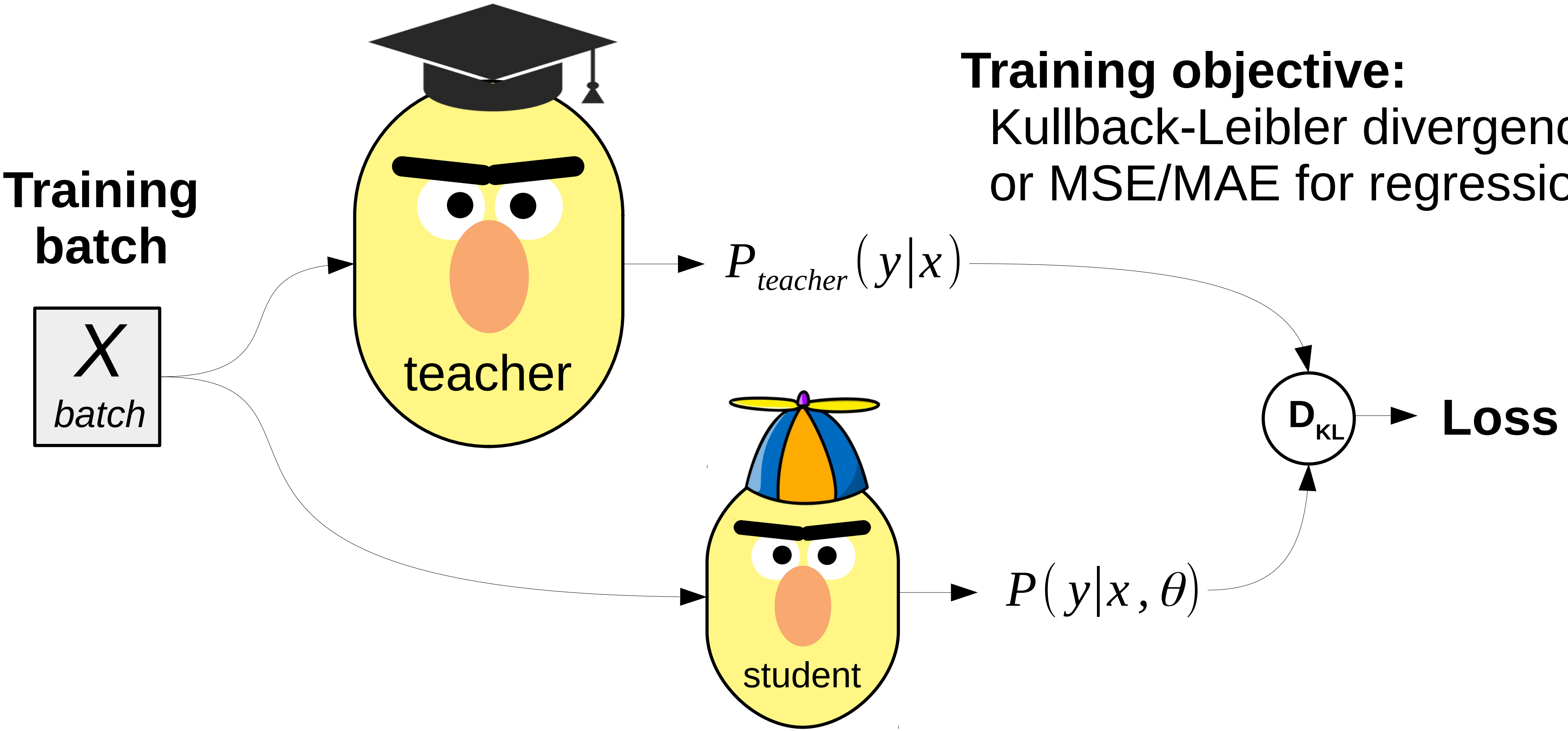
Then, train a more compact model to approximate it!

Compression by Distillation

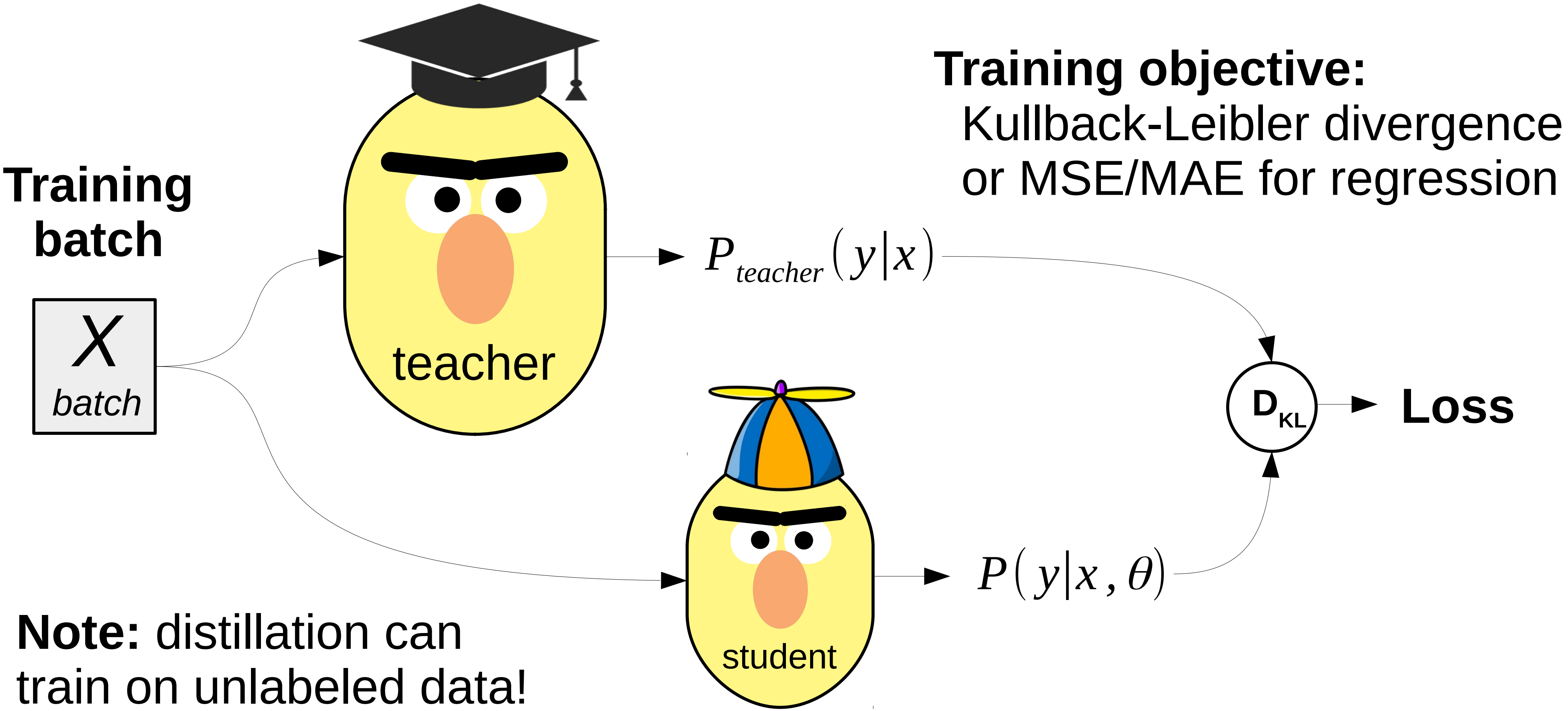


Compression by Distillation

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



Compression by Distillation



Compression by Distillation

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

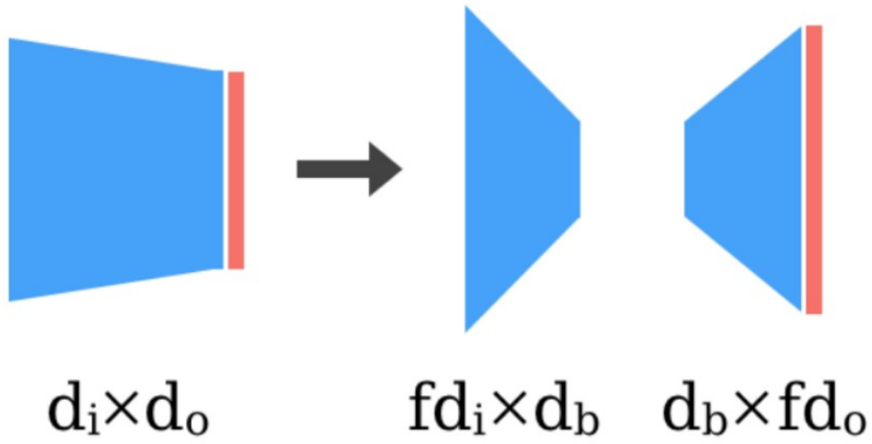
Same as BERT-base, but
with *half as many layers*
(and ≈1.5 times faster)

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

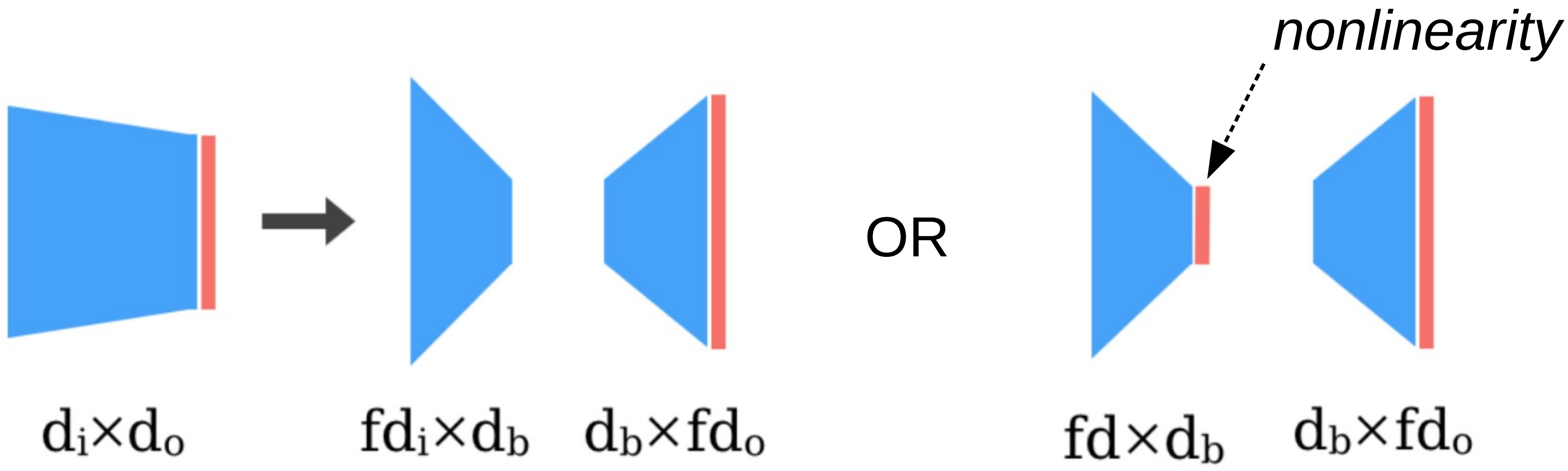
- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
  Factorized: product of smaller matrices or tensors



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

Compression by Distillation

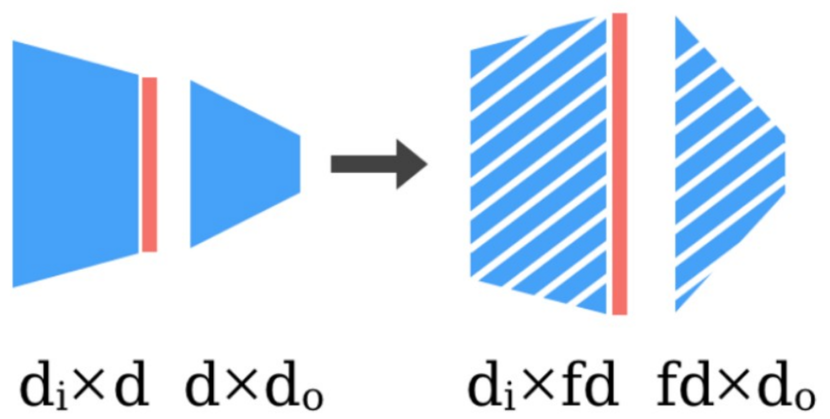
- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
  Factorized: product of smaller matrices or tensors



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

Compression by Distillation

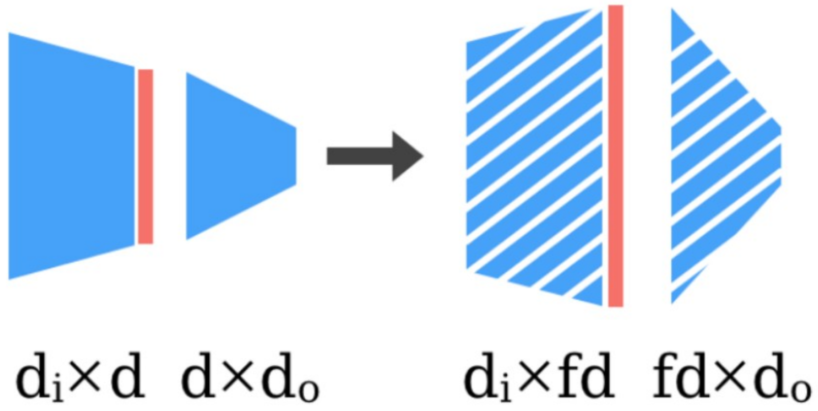
- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
  Factorized: product of smaller matrices or tensors
  Sparse: only a small (random) subset of weights are nonzero



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

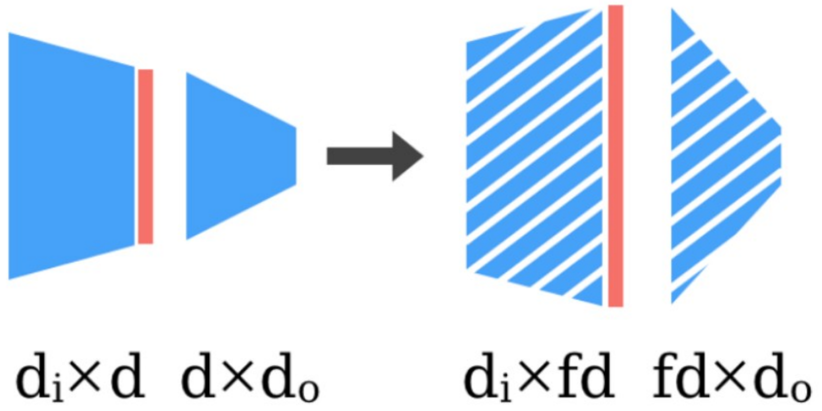


Q: how to store sparse weights?

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

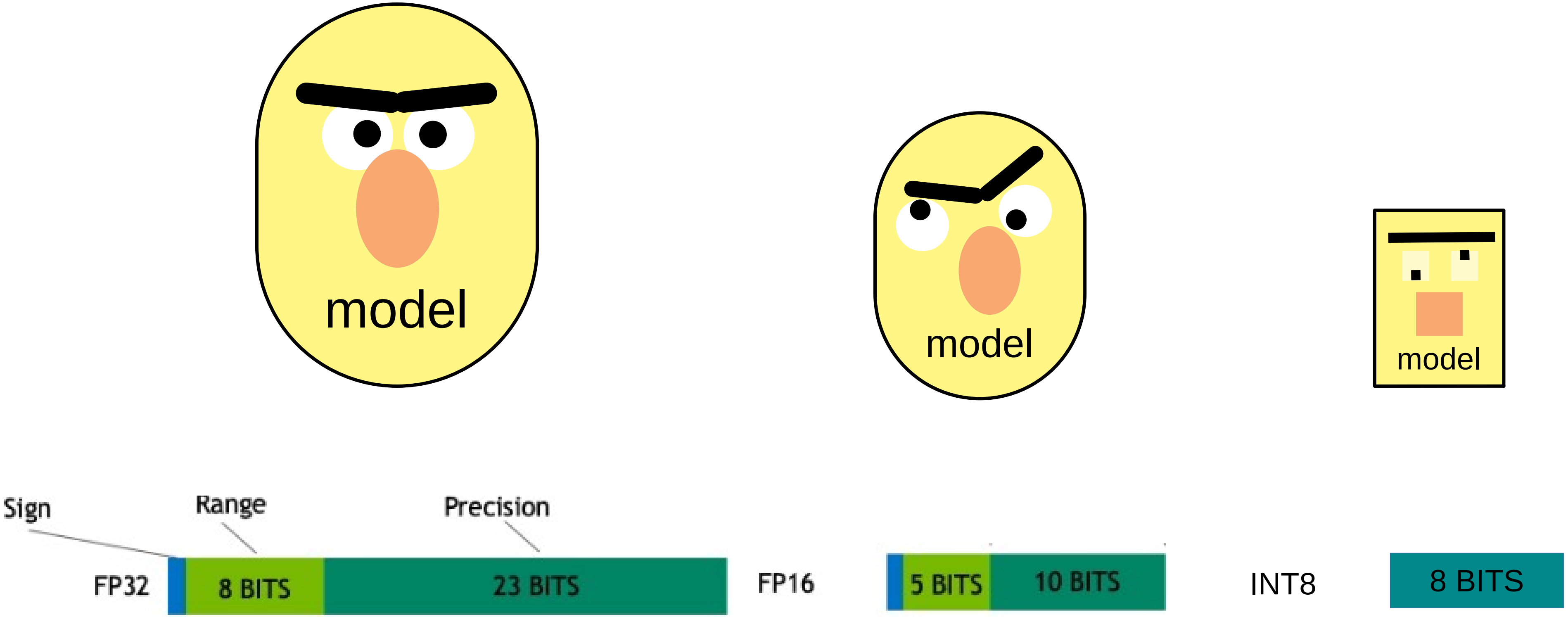
- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
  Factorized: product of smaller matrices or tensors
  Sparse: only a small fraction of weights are nonzero
  Read more: https://openreview.net/pdf?id=_zx8Oka09eF
  Also: factorized embeddings https://arxiv.org/abs/1901.10787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

Compression by Distillation

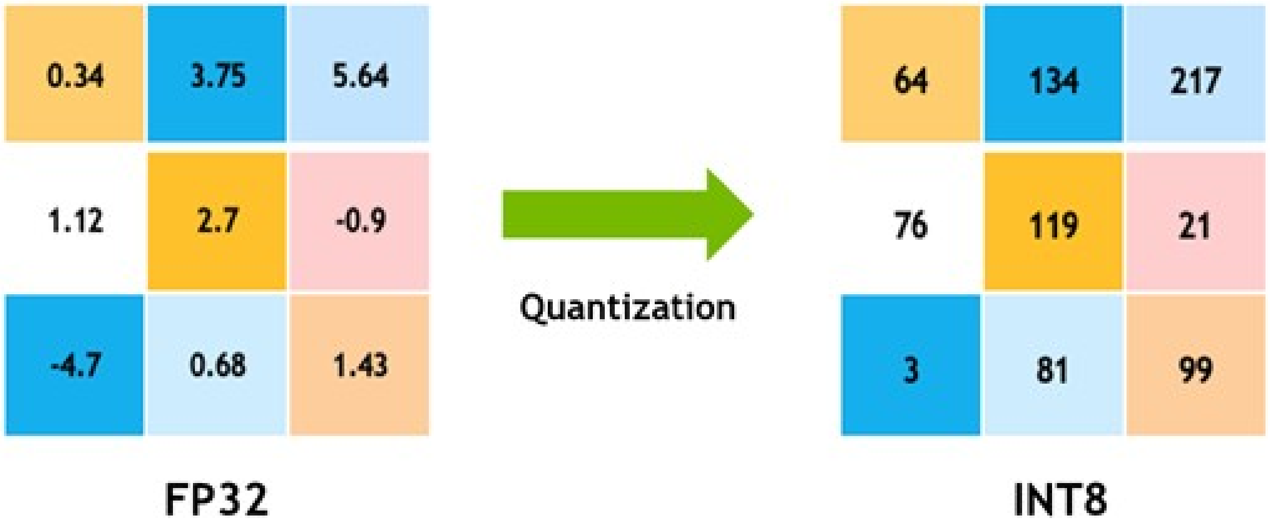
- Student architecture choices:
  Naïve: same but smaller, less layers / hidden units
  Factorized: product of smaller matrices or tensors
  Sparse: only a small fraction of weights are nonzero
  Read more: https://openreview.net/pdf?id=_zx8Oka09eF
  Also: factorized embeddings https://arxiv.org/abs/1901.10787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

Compression by quantization

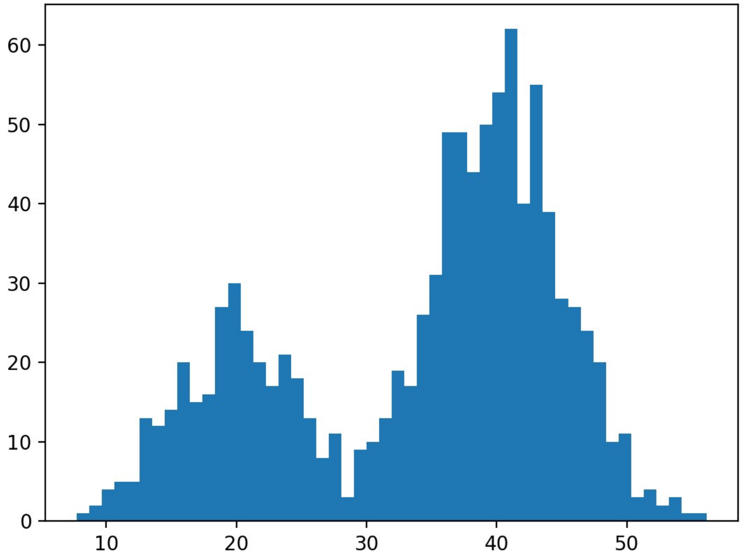


Linear quantization



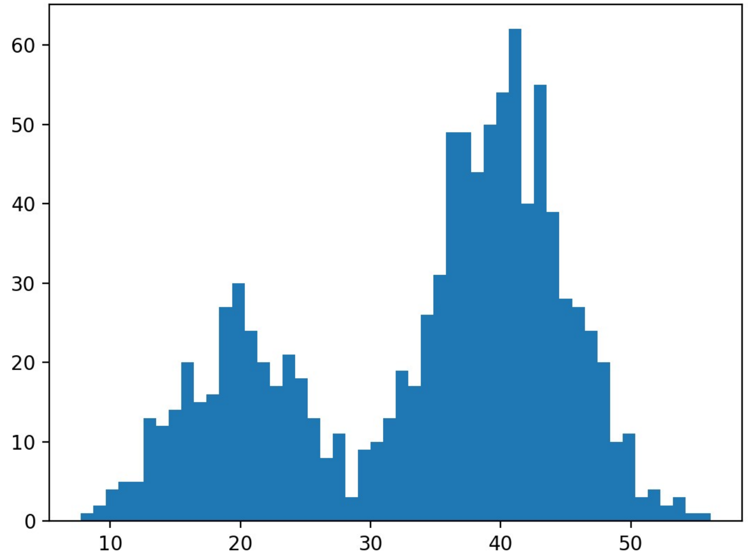
1) Scale inputs and parameters into uint8 range
2) Multiply in uint8, accumulate to int32
3) Un-scale multiplication results in float32

Non-linear quantization



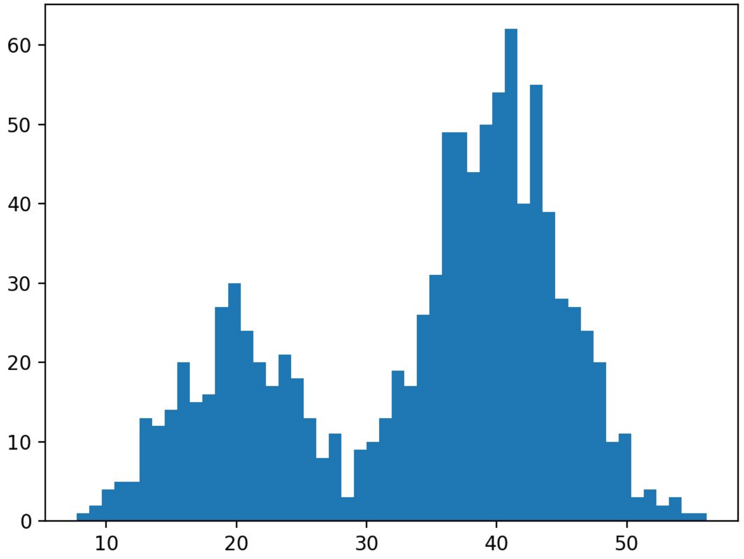
Consider weights as a distribution

Non-linear quantization



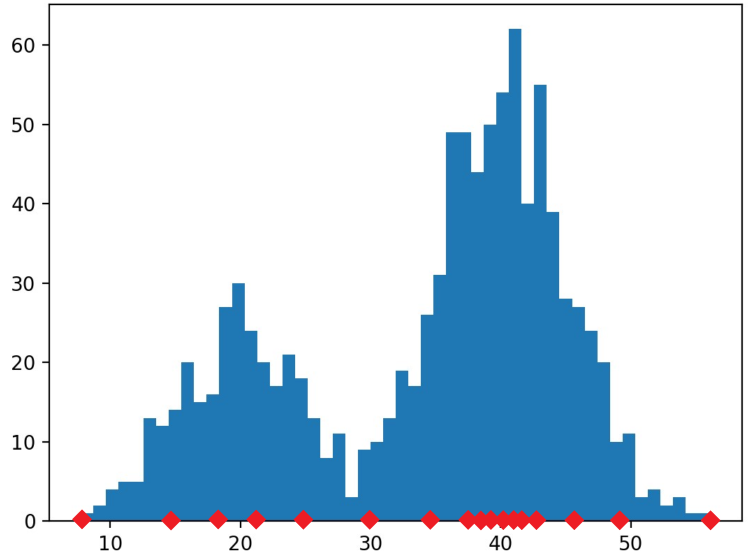
Consider weights as a distribution

Non-linear quantization



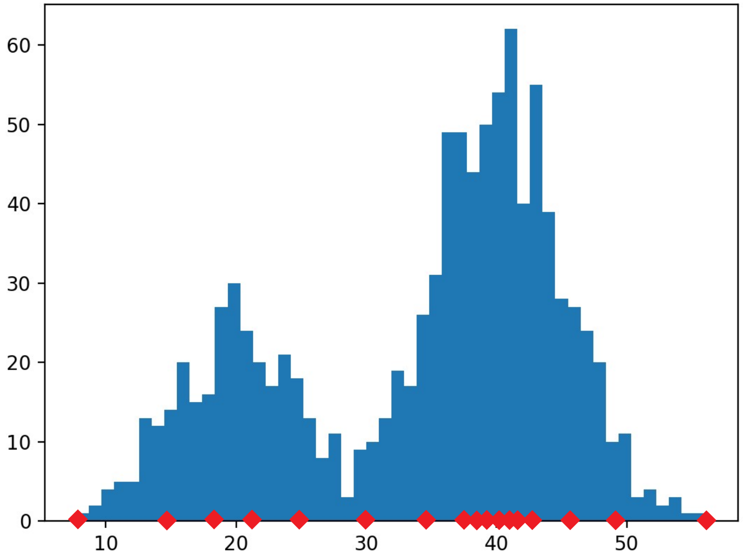
Consider weights as a distribution

Non-linear quantization



Compute a grid of percentiles

Non-linear quantization



percentiles (32-bit)

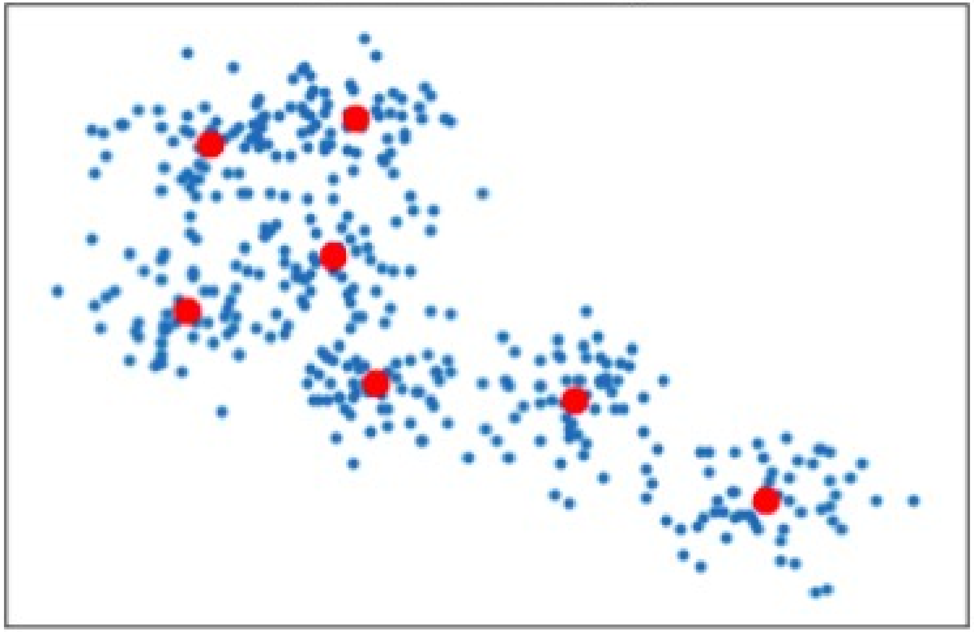
Index (4- or 8-bit) of nearest percentile for each weight

Store each weight as its nearest percentile

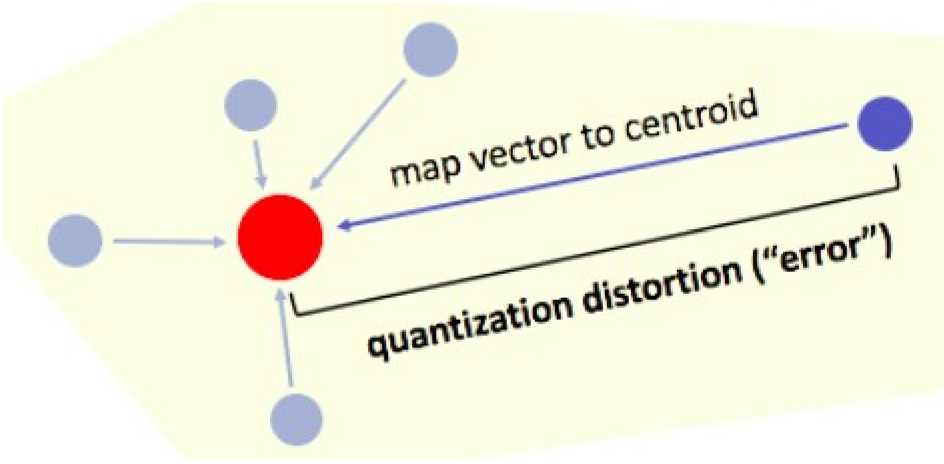
High-dimensional case

Quantize entire vectors as K-means

Quantization Example



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

Low-precision training!

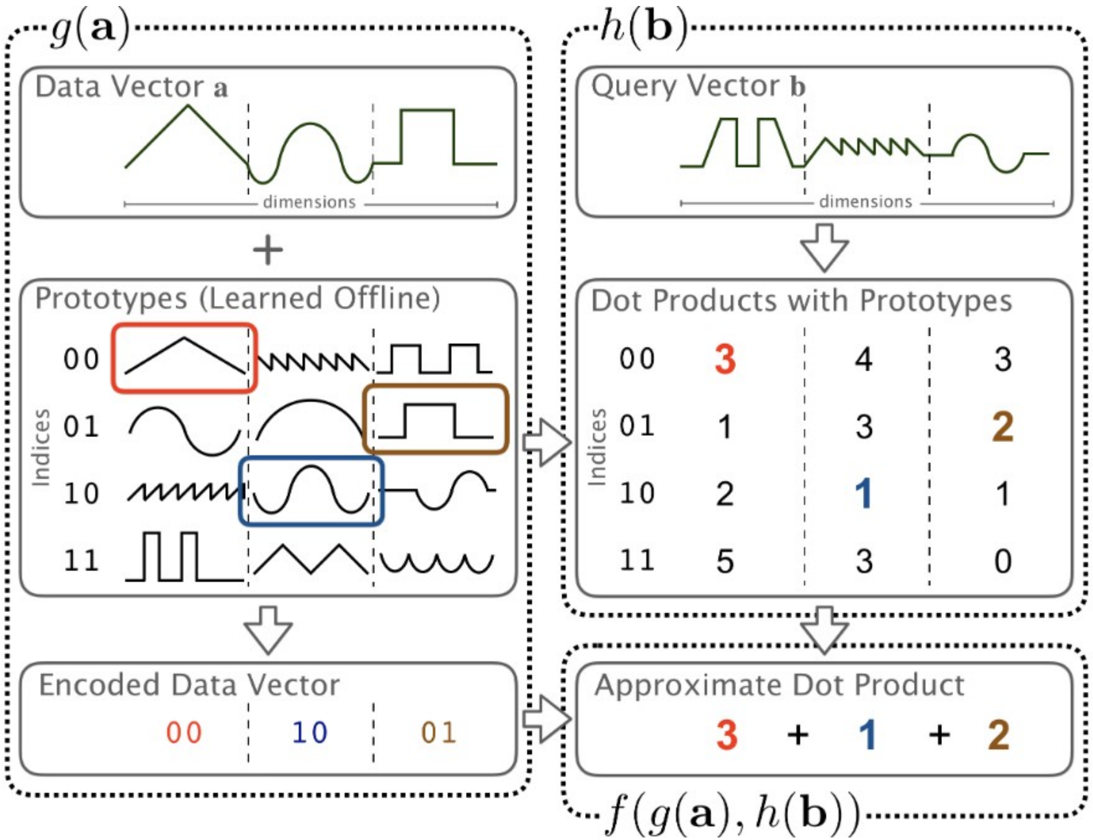
Training with 8-bit params:
https://arxiv.org/abs/1812.08011
https://arxiv.org/abs/1805.11046

Compressing gradients to 1 bit and beyond
https://arxiv.org/abs/2102.02888
https://arxiv.org/abs/1905.13727

Training with 8-bit optimizers:
https://www.youtube.com/watch?v=IxrlHAJtqKE
https://arxiv.org/abs/2110.02861

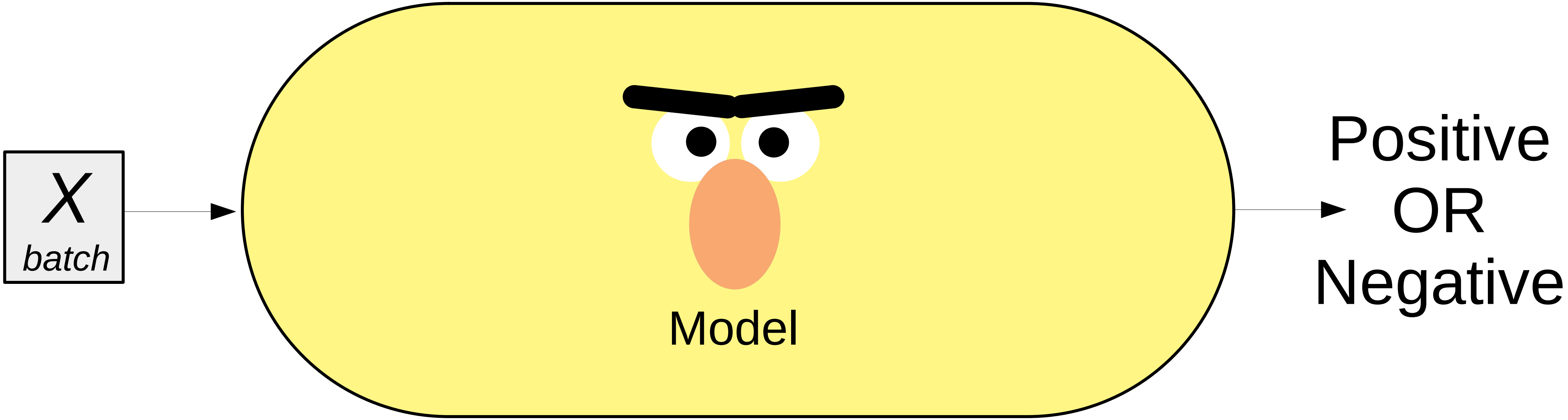
Quantization + pre-computation

Source: https://arxiv.org/abs/2106.10860

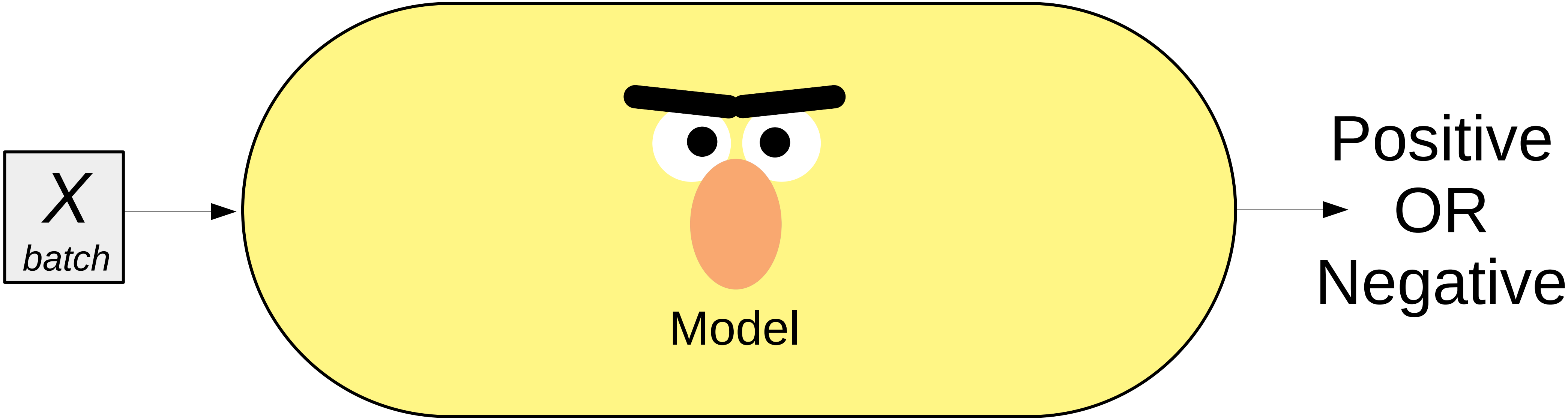


- product-quantize weights
- “quantize” inputs via LSH
- pre-compute dot products all inputs x all codes
- “multiply” by **looking up** pre-computed products

Acceleration by cascading

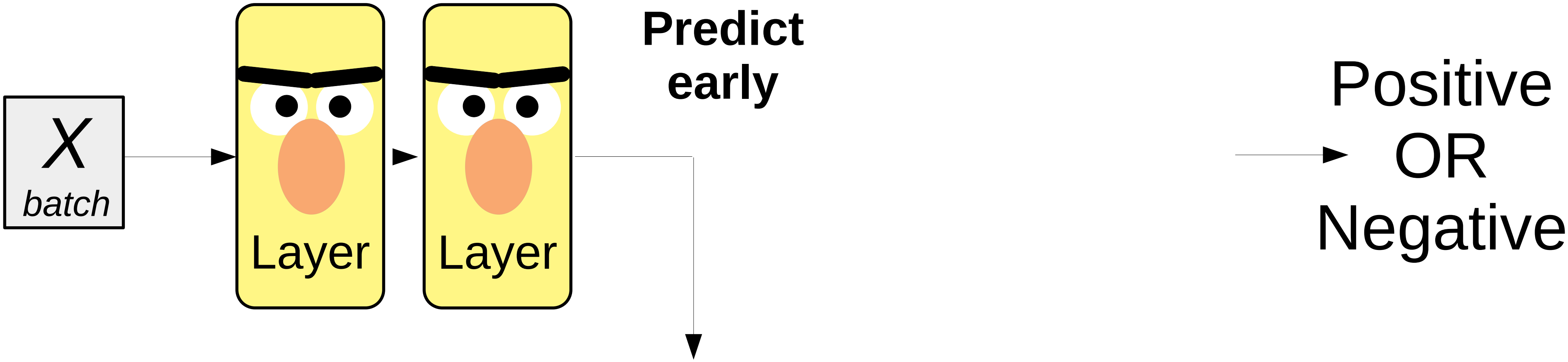


Acceleration by cascading



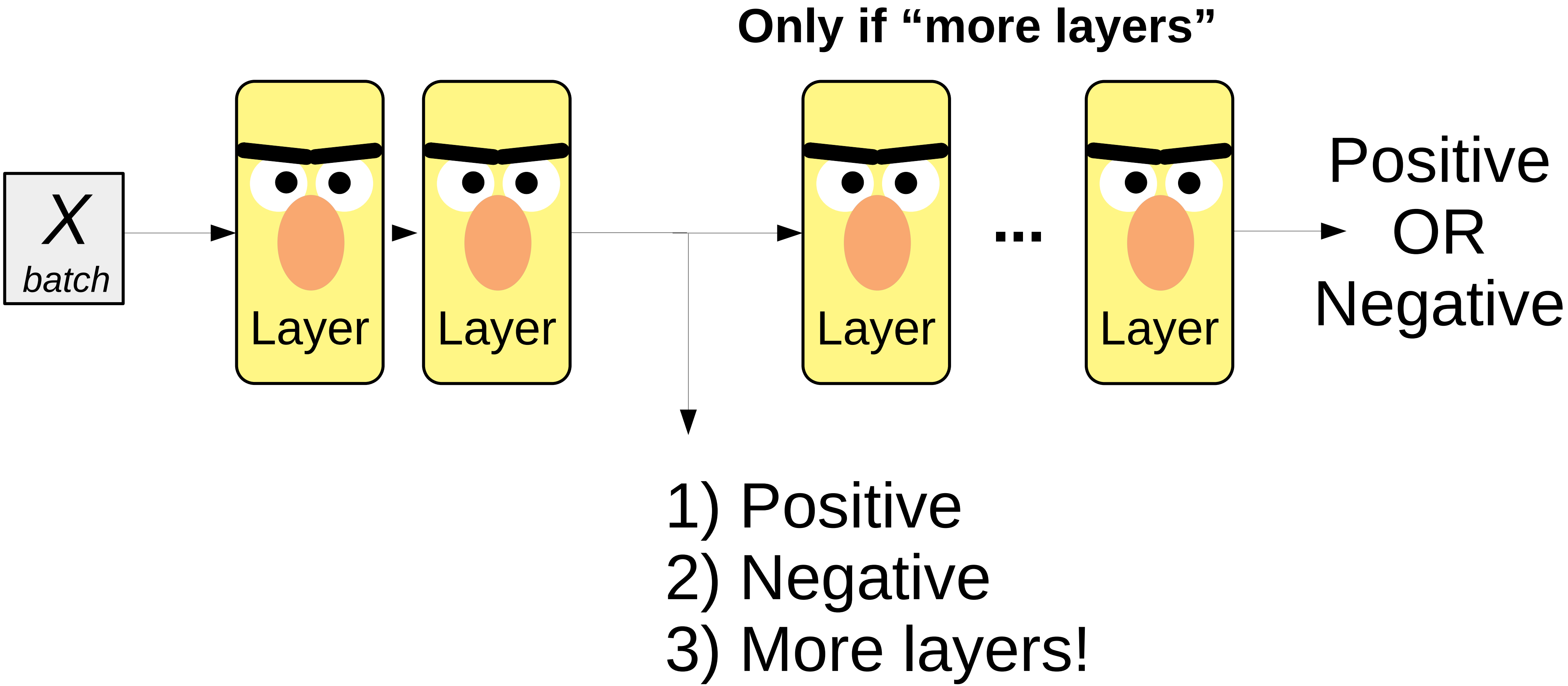
Do we really need every layer
all the time?

Acceleration by cascading

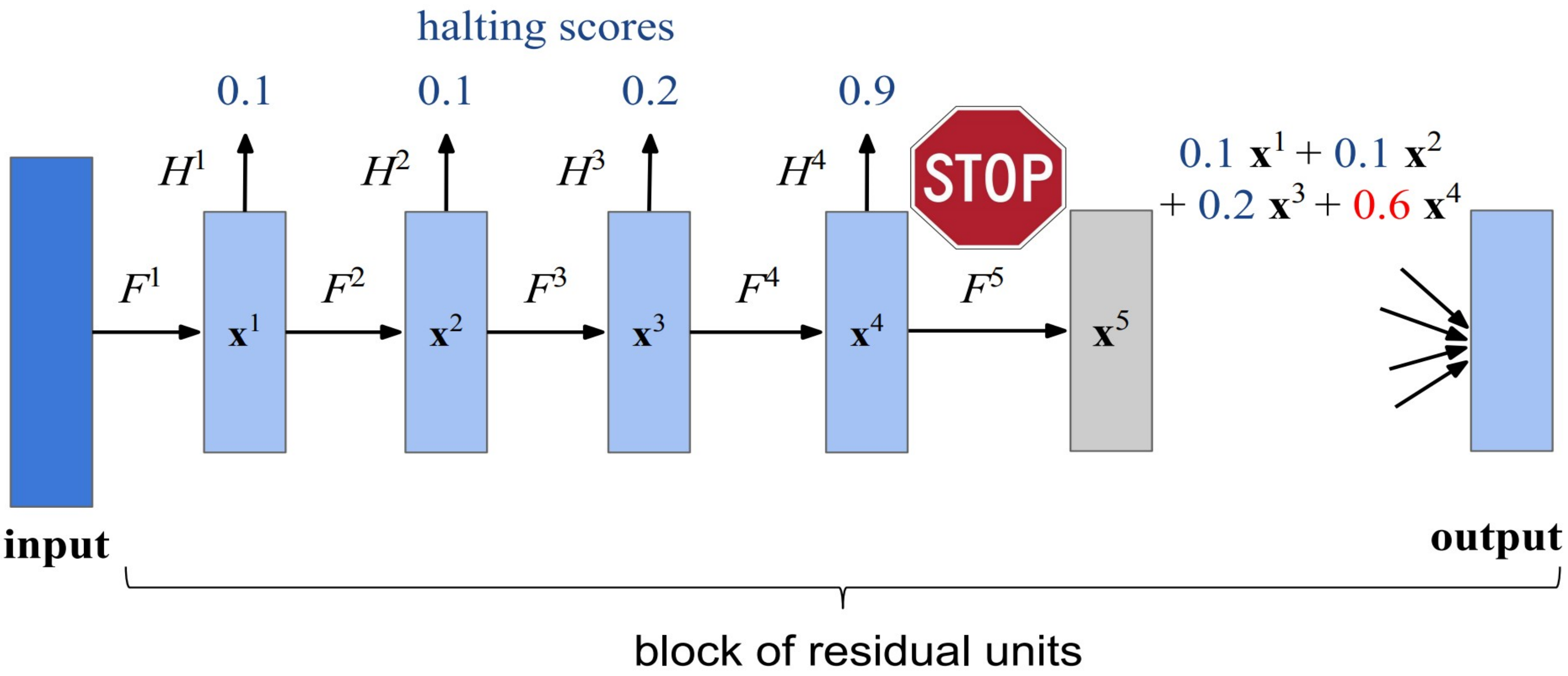


1) Positive
2) Negative
3) More layers!

Acceleration by cascading



Adaptive Computation Time



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

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

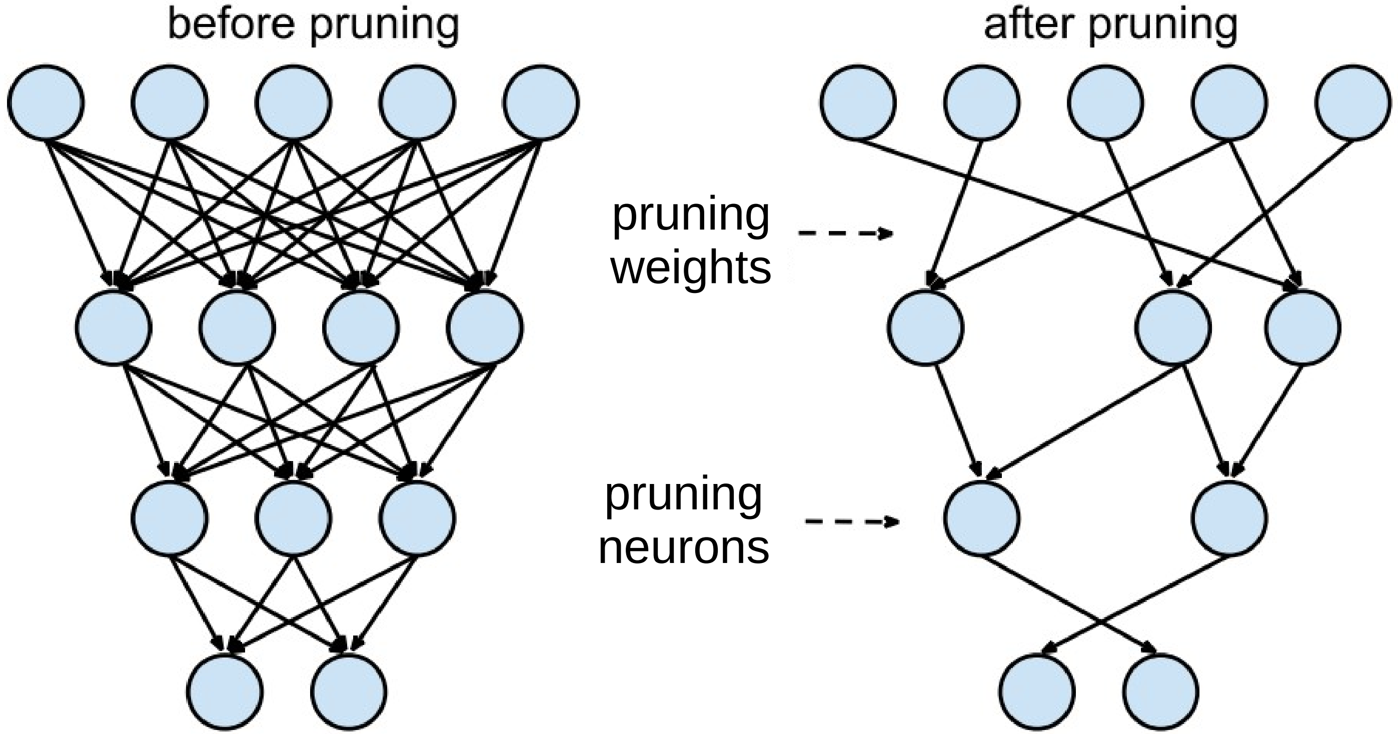
ACT Transformers
https://arxiv.org/abs/1807.03819

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

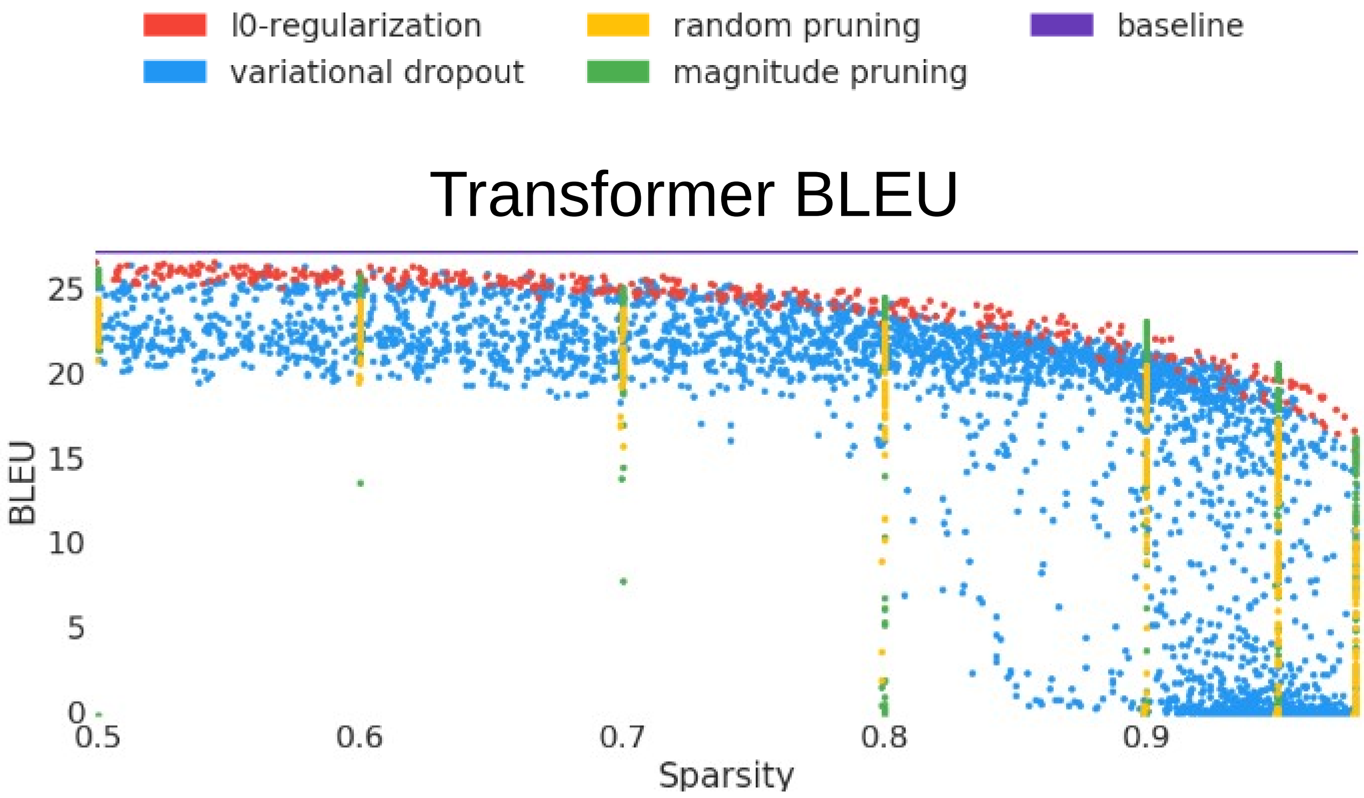
Pruning with L_0 regularization

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

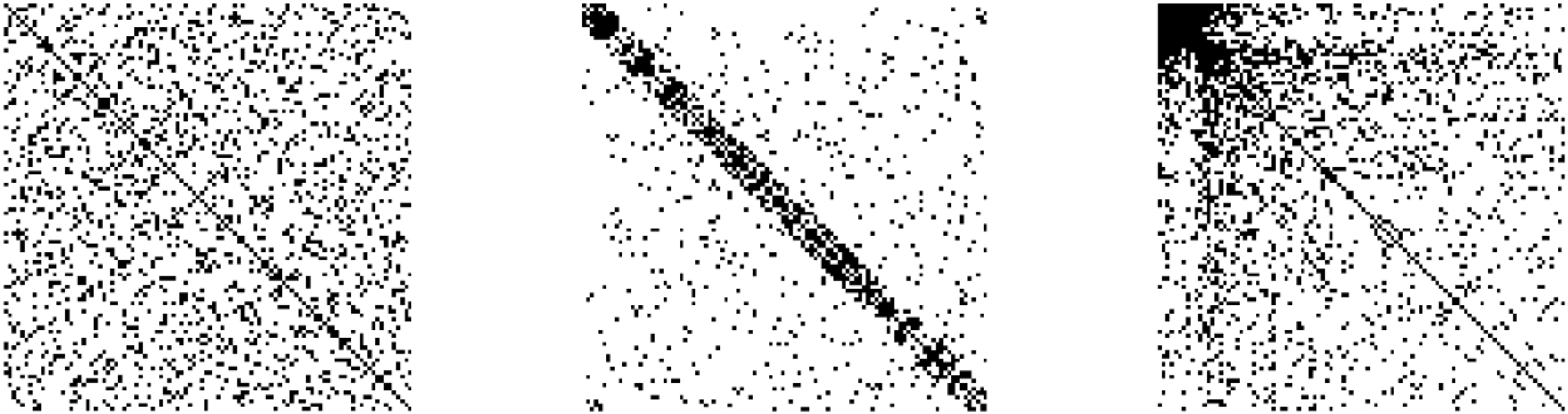
Add a special regularizer that encourages dropping unnecessary weights

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

$$\lambda = 0.01$$

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

Sparse training



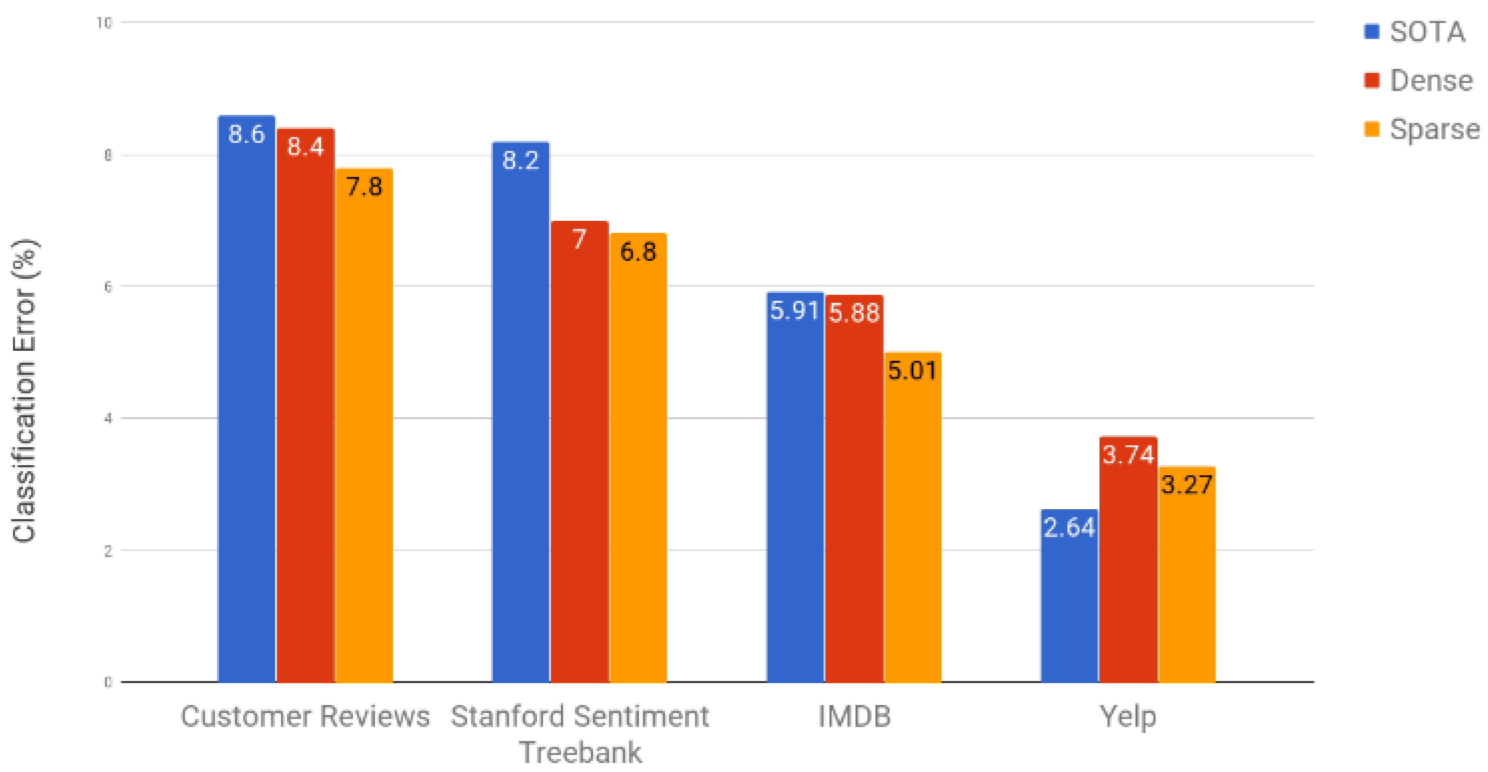
Random

1-Dimensional Watts-Strogatz

Barabasi-Albert

https://cdn.openai.com/blocksparse/blocksparsepaper.pdf

Sparse training



https://cdn.openai.com/blocksparse/blocksparsepaper.pdf

What did we learn?