Agenda:

1. Mixture of Expert models
2. (Preview of) Efficiently Computing mixture of Experts
3. memory Efficient ML (as time permits)
   - ↳ Pruning / Sparsity
   - ↳ Distillation
   - ↳ Quantization

] Background for Thursday's guest lecture!

- Overview behind MoEs

- Large models take a lot of compute to train
- But we know the larger the model, the better the accuracy
- Can we have a large model (e.g., many params) but have less compute needed for training or pre-training?

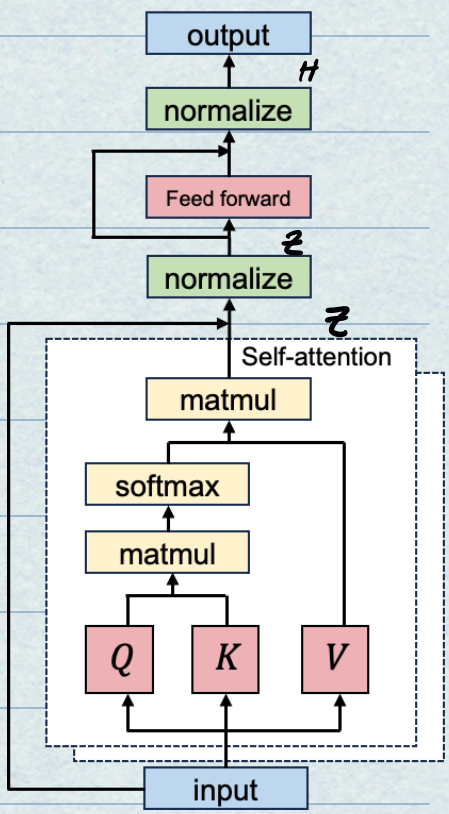
Recall Transformer Block:

$$z = \text{selfattention}(XW_Q, XW_K, XW_V)$$

$$z = \text{LayerNorm}(X + z)$$

$$H = \text{LayerNorm}(\text{Relu}(zW_1)W_2 + z)$$

→ recall output of self attn is: (z)



z: $1 \times N \times d$ → seqlen

batch size

↳ attention dimension

in proj 2, we used $m = 4$

- $W_1 \rightarrow d \times md$
- $W_2 \rightarrow md \times d$

$\longrightarrow (zW_1)(W_2)$

$N \times d \quad d \times md \quad md \times d$

$N \times md$ $\quad$ $N \times d$

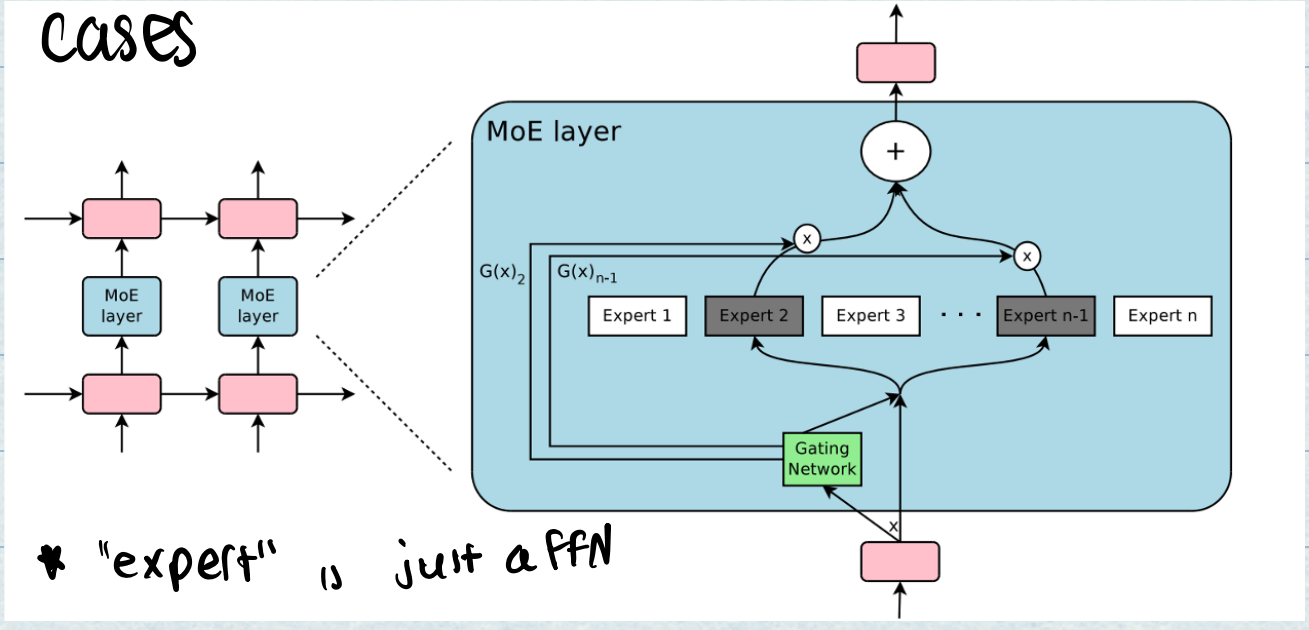
- what is flops for $(zW_1)(W_2)$?

$2Nd^2m + 2Nd^2m$

for $m = 4$ in proj 2: this is $16Nd^2$

→ point: when we increase d (a common way to make model "larger") → computation for feed-forward grows quadratically

Key idea in MoEs: make each expert focus on predicting answer for a subset of cases



* "expert" is just a FFN

How does this actually work?

1) Gating function: decide which experts to activate

- $G = \text{Softmax}(W_g \cdot X)$ — W_g = separate set of trained weights which gating network uses to choose expert.
- Expert indices: $I = \{i_0, i_1\} = \text{TopK}(G, K=2)$
- weight experts:

$$s_0 = \frac{g_{i_0}}{g_{i_0} + g_{i_1}} \qquad s_1 = \frac{g_{i_1}}{g_{i_0} + g_{i_1}}$$

- output: (each expert also has FFN weights)

$$Y = s_0 \text{FFN}_{i_0}(X) + s_1 \text{FFN}_{i_1}(X)$$

Why does this make training or inference less computationally intense?

→ suppose we have an 8×7B MoE model

→ we can run w/ effectively less compute:

↳ for each training example,

K experts activated

- 14B params for K=2

↳ in practice, self-attn weights are shared, so it is more like computation of 12B dense model

*Trevor's guest lecture will cover efficient MoEs!

→ also only need to load k expert FFN weights per example

PART 2 of LECTURE: Pruning/Sparsity, Distillation, Quantization

- we want models that take up less space
  - for mobile devices, small embedded devices

	Cloud AI (NVIDIA V100)		Mobile AI (iPhone 11)		Tiny AI (STM32F746)		ResNet-50	MobileNetV2	MobileNetV2 (int8)
Memory	16 GB	4× →	4 GB	3100× →	**320 kB**	← gap →	7.2 MB	6.8 MB	1.7 MB
Storage	TB~PB	1000× →	>64 GB	64000× →	**1 MB**	← gap →	102MB	13.6 MB	3.4 MB

- microcontrollers have 3x less memory than mobile phones
- widely used vision models are too large by 100x!

MLPerf Training of BERT, maintaining 99% accuracy

Scenario	Closed Division	Open Division	Speedup
Offline samples/sec	1029	4609	4.5x
Server samples/sec	899	4265	4.7x
Single Stream p90 Latency (ms)	2.58	0.82	3.1x

Table 1. BERT Large performance metrics for both closed division and open division

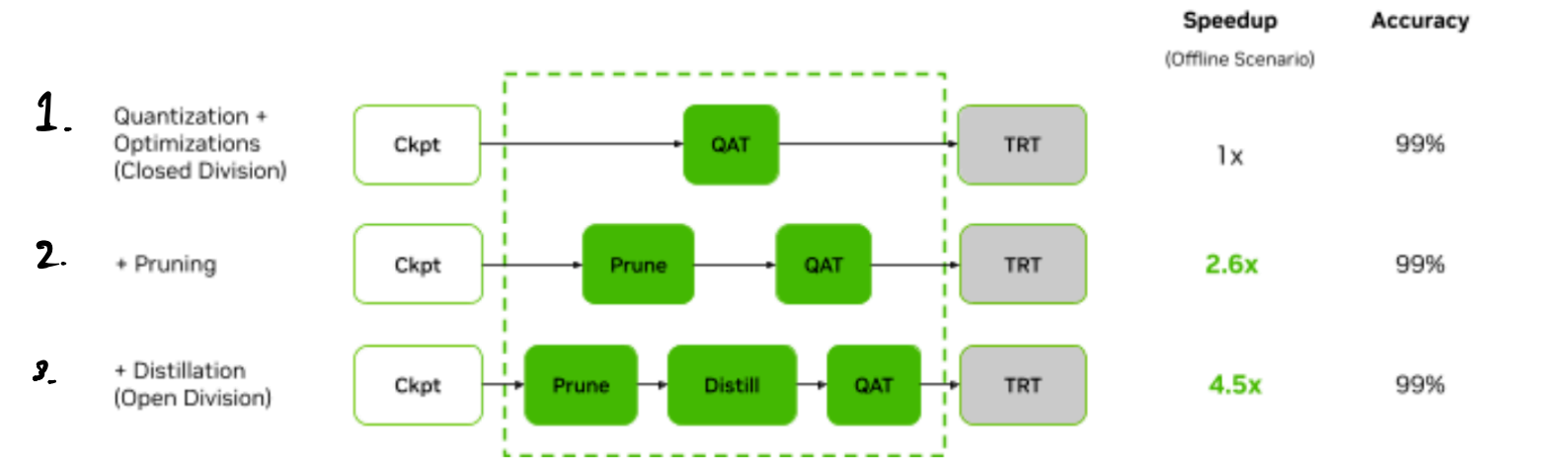
https://developer.nvidia.com/blog/leading-mlperf-inference-v3-1-results-gh200-grace-hopper-superchip-debut/#:~:text=edge%20deployment%20scenarios.-,MLPerf%20Inference%20v3.,a%20form%20of%20generative%20AI

- closed division - cannot change network, but only change HW
- open division - can change anything in network

(quantization, compression, pruning)

↳ latency AND throughput is much faster!

.3 main ideas



https://developer.nvidia.com/blog/leading-mlperf-inference-v3-1-results-gh200-grace-hopper-superchip-debut/#:~:text=edge%20deployment%20scenarios.-,MLPerf%20Inference%20v3.,a%20form%20of%20generative%20AI

* when we discussed FlashAttn and vllm - these were ways to reduce memory bottleneck for ML training and inference

CAN we reduce memory requirement all-together?

- Pruning / Sparsity - after training, reduce size of network by discarding a subset of parameters / neurons
- Quantization - use subset of real #s in model, rather than full set
- Knowledge Distillation - train smaller model that mimics quality of larger model

PRUNING / SPARSITY

- intuition: NN is typically overparametrized, w/ lots of

redundancy in what is being represented - can turn some weights to 0

- W: original weights
- W_p: pruned weights → $||W_p||$ = # of non-zero vals
- T = target # of params after pruning
- L = loss, x = input

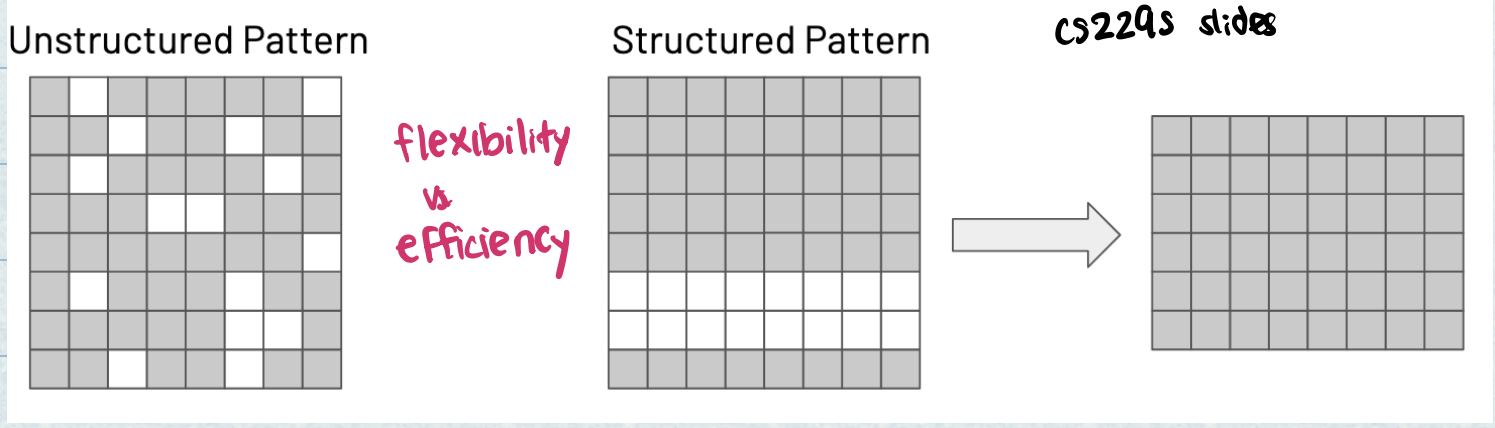
$$\min_{W_p} L(x; W_p) \text{ s.t. } ||W_p|| < T$$

- ideally, PRUNING PROCESS CAN: / IS

1) accelerate training /and/or inf. on HW
2) retains accuracy
3) generalizable: not over-engineered to any specific architecture

Pruning Weights Patterns

↳ note could also prune entire neurons or activations; weights are popular



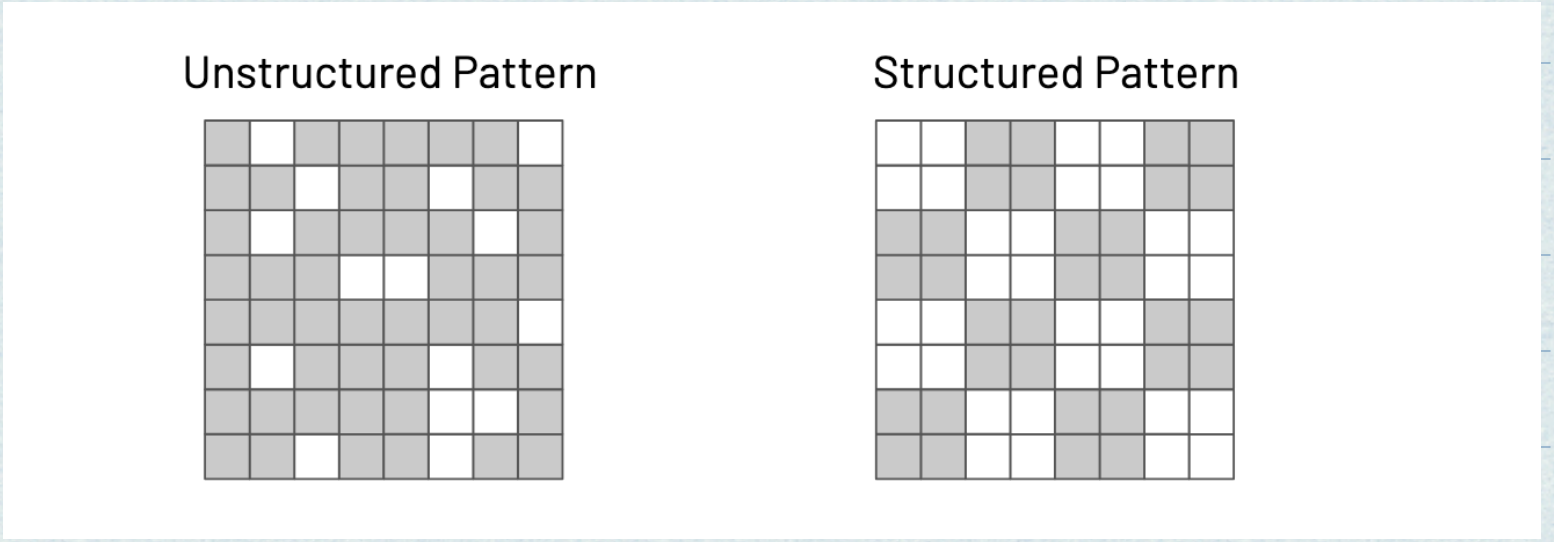
- more flexible, could result in higher accuracy
- very randomly sparse

- removing entire rows is HW efficient, but could hurt accuracy as it is so coarse
- when doing computation, compress matrix into dense representation

models may not be HW efficient

and also store which actual row IDs are non-zero

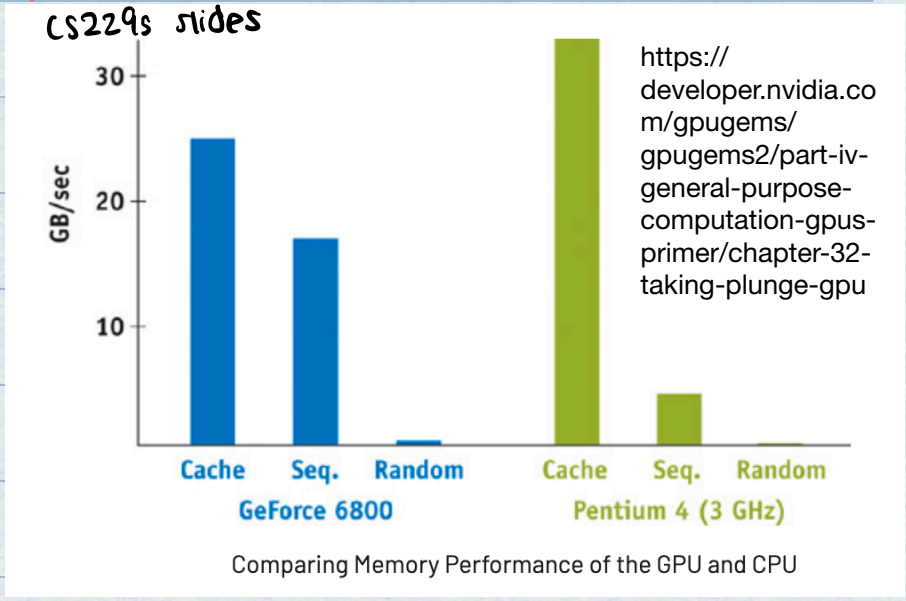
Another Pattern, optimized for Tensor Cores



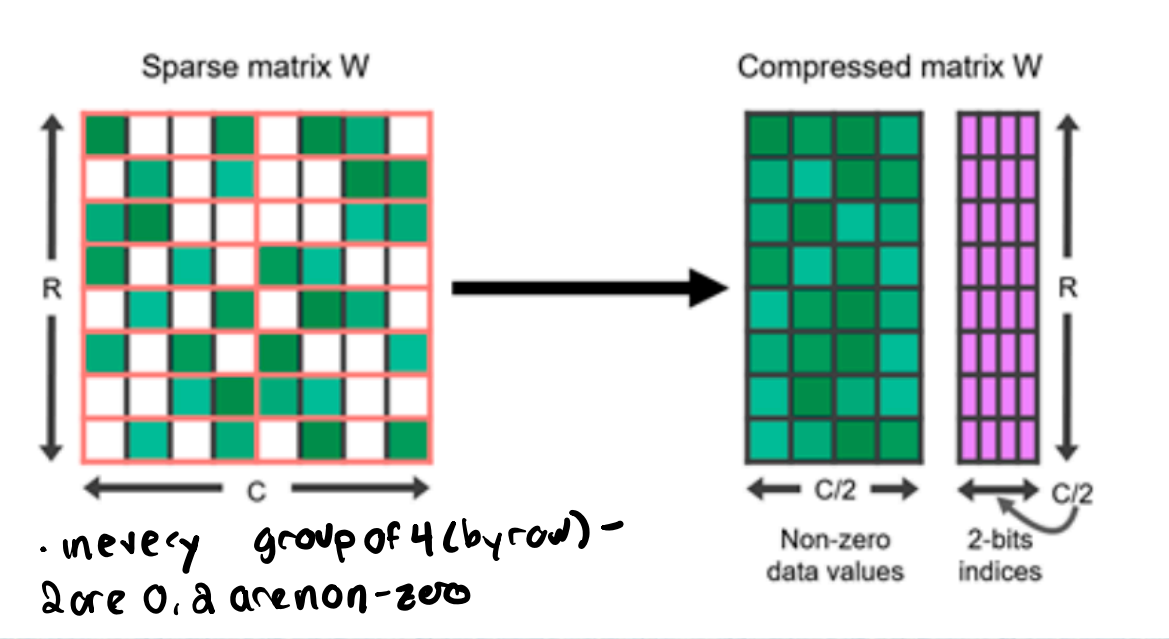
- Tensor cores can perform mat mul's on 16x16 or 8x32, etc. sized chunks
  - ↳ if we know sparsity pattern of weights, we could implement a special kernel that restructures entire matmul to avoid doing anything w/ 0-blocks

Structured Access is very important for HW efficiency

* cached - read same location repeatedly
* sequential - GPU at this really good
* random - both CPU and GPU are bad



sparse tensor cores-



. in every group of 4 (by row) - 2 are 0, 2 are non-zero

https://developer.nvidia.com/blog/accelerating-inference-with-sparsity-using-ampere-and-tensorrt/

* there are in betweens of pruning entire rows/columns and complete unstructured sparsity

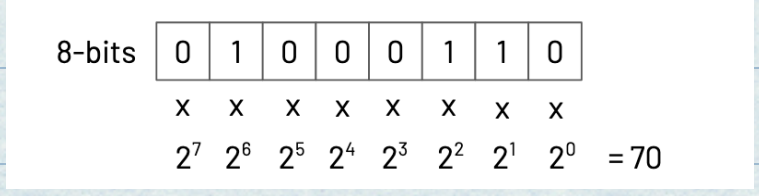
- above is called 2:4 sparsity pattern — in each contiguous block of 4 values, 2 are 0
  - ↳ 1) can store in dense representation which is $\frac{2}{4}$ or 50% of the size
- also need to store, for each block of 4, which 2 are 0:
  - store 2 2-bit #s of index of non-zero elts for each block of 4
- sparse tensor cores can do mat-muls directly w/ this representation
  - can do same calc in half the time

- there are many ways to discuss WHICH weights to remove - we won't discuss in detail here
  - core idea: iterative pruning (prune + retrain/finetune), remove weights w/ smaller absolute value

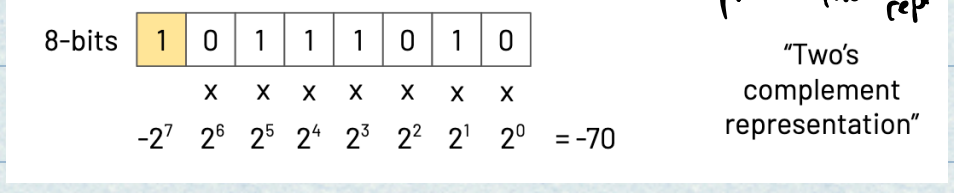
- idea 2: Quantization
  - reduce # of bits per weight

- recap: how do computers represent #s?

- unsigned integers:

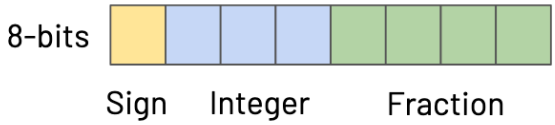


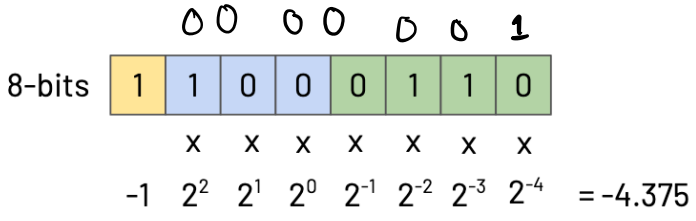
signed integer



note there's also "signed magnitude repr"

Fixed Point #s





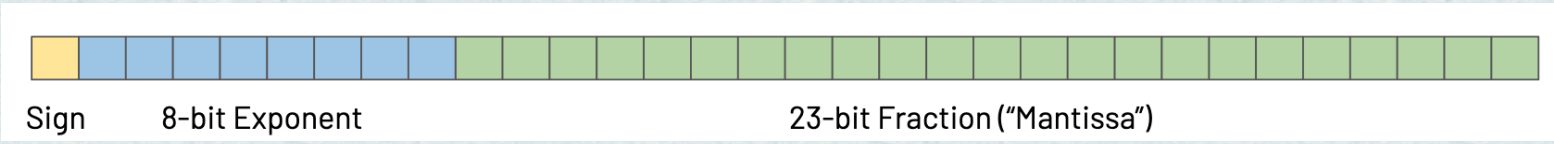
- what is smallest non-zero value? (pos. or neg.)

→ 2^{-4} or -2^{-4}

.0625 or -.0625

we miss all #s in middle!

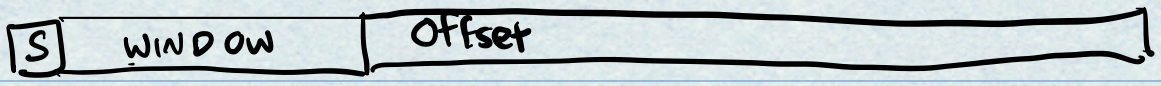
Floating-Point Numbers



$$\text{Value} = (-1)^{\text{sign}} \times (1 + \text{Fraction}) \times 2^{\text{Exponent - BIAS}}$$

- Bias $= 2^{(\#\text{exp bits}-1)} - 1 = 2^7 - 1 = 127$

what does this mean?



→ consider $(1 + \text{fraction}) \cdot 2^{\text{Exp - BIAS}}$ part

window: tells us which two consecutive powers of 2 the # will be in:

note these are exclusive on right side

$[2^{-127}, 2^{-126}) \; [0.5, 1) \; [1, 2) \; [2, 4) \; [4, 8) \ldots [2^{127}, 2^{128})$

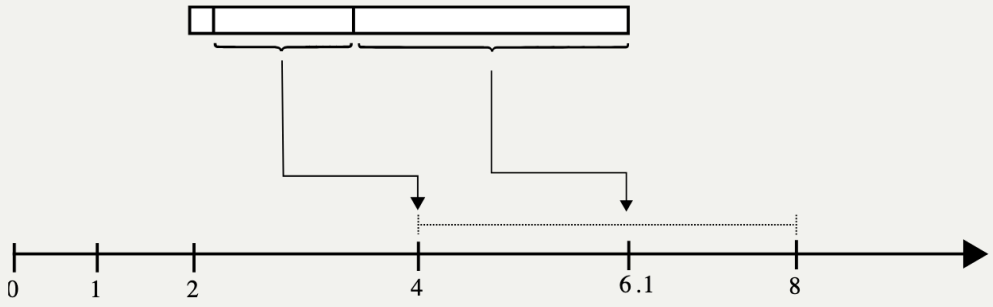
$2^{-1} \; 2^0 \quad 2^0 \; 2^1 \quad 2^1 \; 2^2 \quad 2^2 \; 2^4$

* exp can range from
* so $2^{\text{exp-bias}}$ will range from:

$2^{0-127} \ldots 2^{255-127}$ or $2^{-127} \ldots 2^{128}$

- offset divides window into 2^{23} (8388608) buckets
- we can use window and offset to approx a #:

* consider 6.1:
  - → between 4 and 8 (2^2 and 2^3)
  - → offset is a little more than halfway down window



Value 6.1 approximated with floating point.

https://fabiensanglard.net/floating_point_visually_explained/

Precision of fp32:

- depends on which window you are in!
- for window $[1, 2]$: $\frac{(2-1)}{2^{23}}$ = some very small #
- for window $[2048, 4096]$: $\frac{2048}{2^{23}} = .0002$ ↓ not as small!

* what is smallest # we can represent?

→ consider smallest window:

$[2^{-127} \ldots 2^{-126}]$

→ f is all 0's:

so $(2^{-127}) \cdot (1.0) = 2^{-127}$

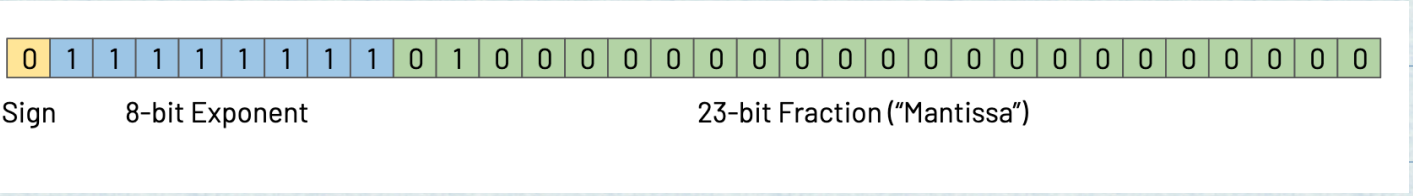
* encoded as all 0's in exp and all 0s in Fractio

*where does NaN come from?

→ exp is all 1 → largest bucket:

$[2^{128} \ldots]$

↳ but nothing after this is valid!
(any non-0 fraction)



Other floating point reprs vary buckets and offsets within buckets

	Exponent bits	Fraction bits	Total
IEEE 754 FP32	8	23	32
IEEE 754 FP16	5	10	16
Google Brain Float 16	8	7	16

BACK TO QUANTIZATION:

- we could use lower precision type (fp8, fp16, bf16)
- or:

- K-means quantization
  ↳ use floating pt computations, integer weights
- Linear Quantization
  ↳ integer weights and arithmetic

K-means quantization (for inference)

- idea: take original weights, cluster them
- store cluster centroids
- store which cluster centroid each weight corresponds to

CS2291 slides

Original FP32 Weights

2.09	-0.98	1.48	0.09
0.05	-0.14	-1.08	2.12
-0.91	1.92	0	-1.03
1.87	0	1.53	1.49

Centroids

3:	2.00
2:	1.50
1:	0.00
0:	-1.00

Cluster Index (2-bits)

3	0	2	1
1	1	0	3
0	3	1	0
3	1	2	2

Song Han et al., Deep Compression, 2016 ICLR (Best Conference Paper)

S = sizeof (index)

if orginal dxd matrix is in fp 32:

what are storage savings?

bits
$32 \cdot D^2$ vs. $(\log_2(S) \cdot D^2 + 5.32)$ ← store cluster centroids in fp32

→ for D = 4

we have 64B vs. 20B (44B saved or 3.2x compression)

→ as S ↓↓↓, larger savings)

* we will continue 2 lectures from now