Today's Agenda:

1. Quantization methods in training / inference
2. A little bit on distillation
3. A little more on MoEs (content that Trevor didn't get to)

Recall from last time

	Exponent bits	Fraction bits	Total
IEEE 754 FP32	8	23	32
IEEE 754 FP16	5	10	16
Google Brain Float 16	8	7	16
Nvidia FP8 EFM4	4	3	8
Nvidia FP8 E5M2	5	2	8

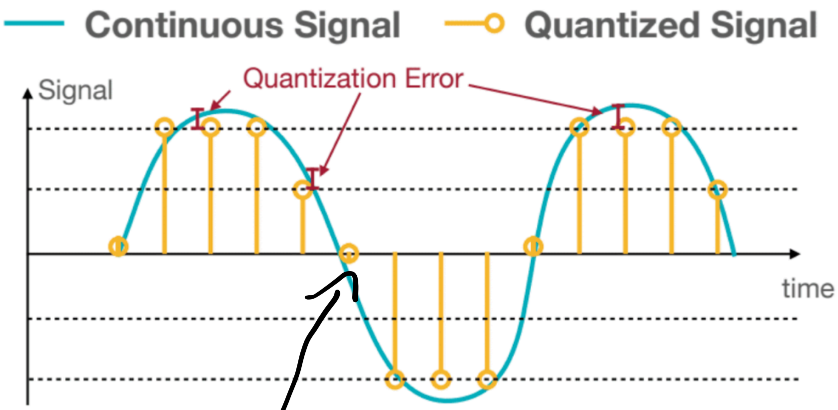
↳ used for gradient in BW pass.

x

- #exponent bits defines range / dif. windows that can exist
- fraction bits define precision (within any bucket)
  - * 2nd fp8 type can represent larger normal values

What is quantization?

" process of constraining an input from a continuous or otherwise large set of values to a discrete set"



dif. btwn input and quantized value = quantization error

Credits: MIT 6.5940: TinyML and Efficient Deep Learning Computing

K-means Quantization for ML

- use INTEGER WEIGHTS, floating - point computation

K-means quantization (1st for inference

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

2 is cluster center for all blue values

→ idea: take original fp32 weight matrix, cluster values

→ instead of storing raw weights, store index to cluster centroid

→ cluster centroids themselves are stored w/ fp32

*use k-means to find cluster centroids

Song Han et al., *Deep Compression*, 2016 ICLR (Best Conference Pape

↳ so what we actually store is:

$$\begin{bmatrix} 3 & 0 & 2 & 1 \\ 1 & 1 & 0 & 3 \\ 0 & 3 & 1 & 0 \\ 3 & 1 & 2 & 2 \end{bmatrix}$$

+ cluster centroids themselves

WHAT ARE STORAGE SAVINGS?

(for inference case)

→ original: $D \times D$ matrix in fp32

$32D^2$ bits (or $4D^2$ bytes)

→ compressed: $D \times D$ matrix of indices to cluster centroids; suppose S = size of index

* # of bits needed to represent a centroid indice =

$\log_2(S)$ bits → $\log_2(S) \cdot D^2$ bits

→ also storing centroids themselves in fp32 → $32S$ bits or $8S$ bytes

original:

$32D^2$ BITS

or $4D^2$ Bytes

compressed:

$(\log_2(S) \cdot D^2 + 32S)$ bits or

$\log_2 S \cdot D^2/8 + 4S$ bytes

→ consider $D=4$, $S=4$:

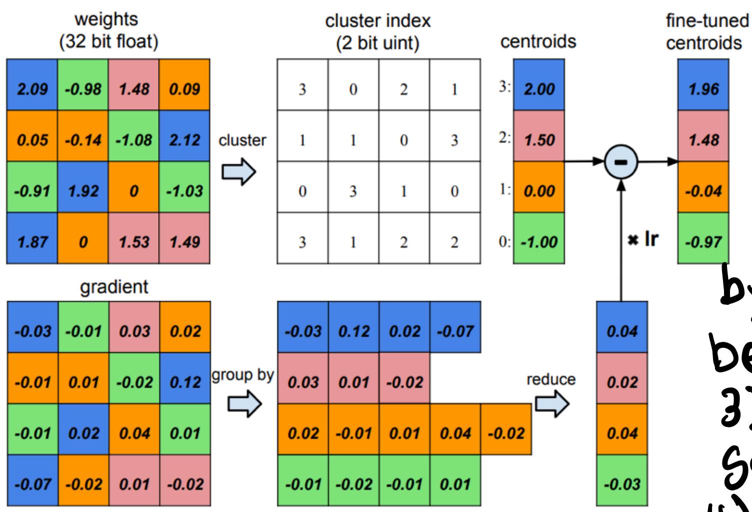
→ original = 64 Bytes

→ compressed = 4 + 16 = 20 Bytes

3.2x smaller!

- WHAT HAPPENS FOR TRAINING, when we have to update weights?

Idea: update cluster centroids (in fp32)



1) for each weight, calculate its gradient w.r.t the loss (this uses centroid value)

2) group weight updates by which centroid it belongs to

3) sum all gradients for a centroid

4) make update

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

→ steps 1 → 3:

$$\partial L / \partial C_k = \sum_{i,j} \partial L / \partial W_{i,j} \text{ for } W_{i,j} \text{ in centroid } c$$

→ step 4:

$$C^{(s+1)} = C^{(s)} - l.r. \cdot \frac{\partial L}{\partial C^{(s)}}$$

+ general compression rate for a set of weights w, represented w/ s centroids, each using b bits:

$$r = \frac{w \cdot b}{w \cdot \log_2(s) + s \cdot b}$$

→ in prev example, $b = 32$, $s = 4$

→ can we do even better? what if weights are distributed non-uniformly?

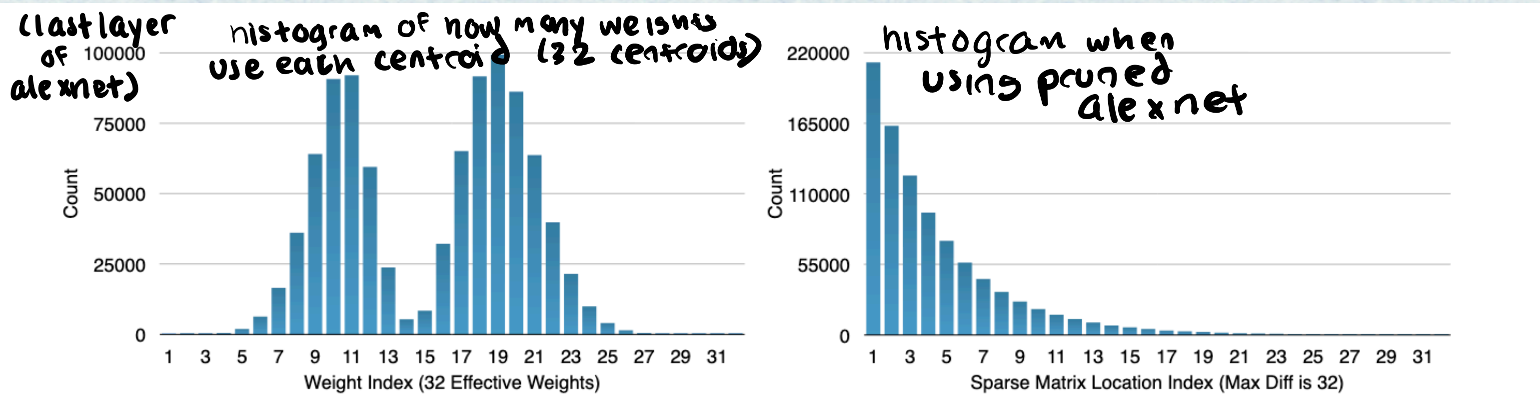


Figure 5: Distribution for weight (Left) and index (Right). The distribution is biased.

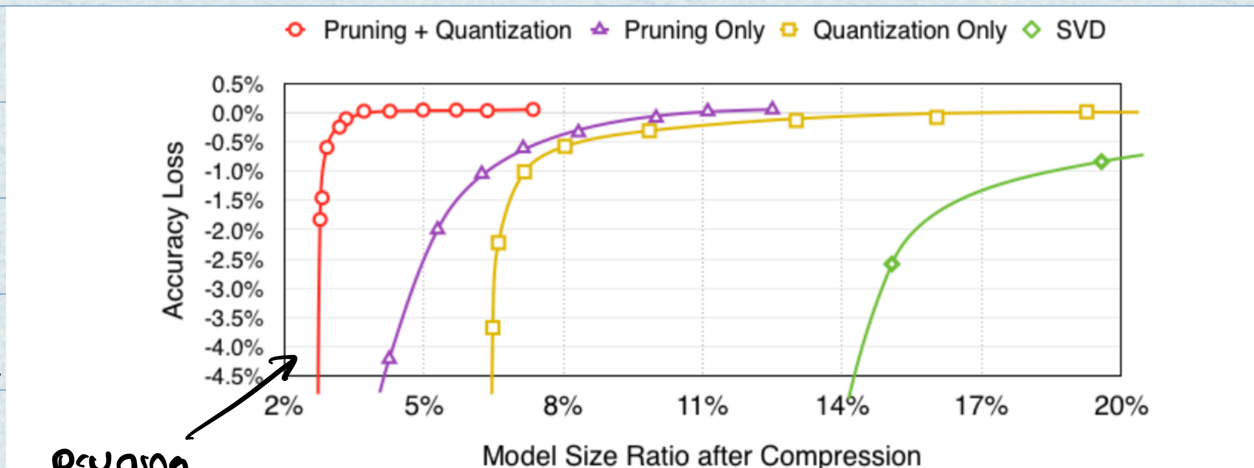
Song Han et al., Deep Compression, 2016 ICLR (Best Conference Paper)

Observation: use HUFFMAN codes

- instead of using a constant # of bits for each centroid (codeword) - use variable-length codewords
- intuition: can use FEWER bits for more frequent values; results in 20% storage savings w/ NO accuracy loss

Results

- within a method, as we increase compression, accuracy↓



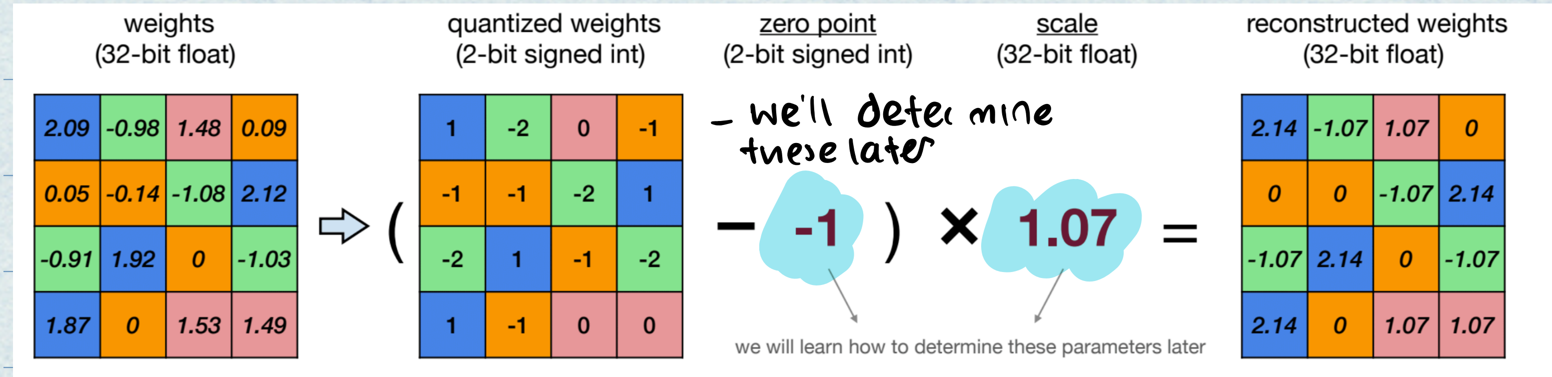
pruning AND quantization is best

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

LINEAR QUANTIZATION

→ integer weights + integer arithmetic

→ "affine" transformation of FPs to integers – involves a scaling + offset

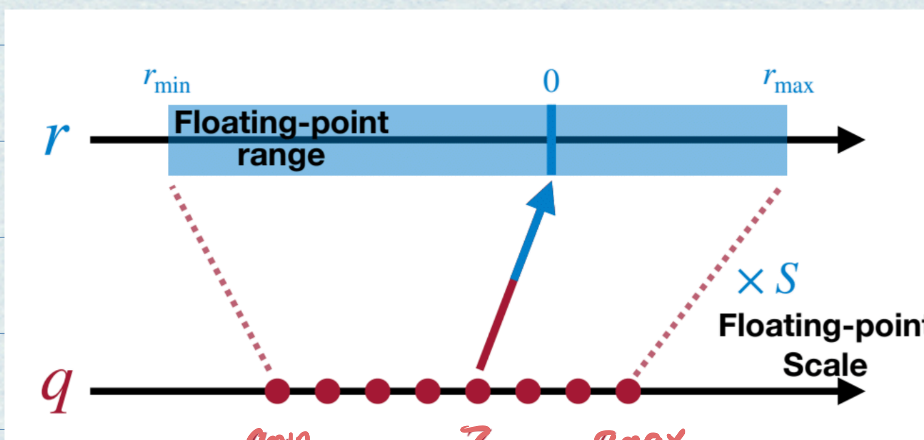


MIT 6.5940 Lecture: TinyML and Efficient Deep Learning Computing (Song Han)

idea: $r \approx S(q - Z)$

→ so only need to store q (quantized integer version), Z (offset int) and S (scaling factor)

How do we determine offset and scaling factor?



q_{min} Z (zero-point) q_{max}

→ we know r_{min} and r_{max} from original weights

→ we know q_{min} and q_{max}:

· If we are quantizing to N bits → range is $-2^{N-1} \ldots 2^{N-1}$

→ we can just solve for S

$r_{max} = S(q_{max} - Z)$ → $r_{max} - r_{min} = S(q_{max} - q_{min})$

$r_{min} = S(q_{min} - Z)$

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

* now calc Z:

$$Z = q_{min} - \frac{r_{min}}{S} \approx \text{round}\left(q_{min} - \frac{r_{min}}{S}\right)$$

CONSIDER a MATRIX MULTIPLICATION

$$Y = WX$$

$$S_y(q_y - Z_y) = S_w(q_w - Z_w) \cdot S_x(q_x - W_x)$$

$$q_y = \frac{S_w S_x}{S_y} \underbrace{(q_w - Z_w)(q_x - Z_x)}_{\text{all integer math!}} + Z_y$$

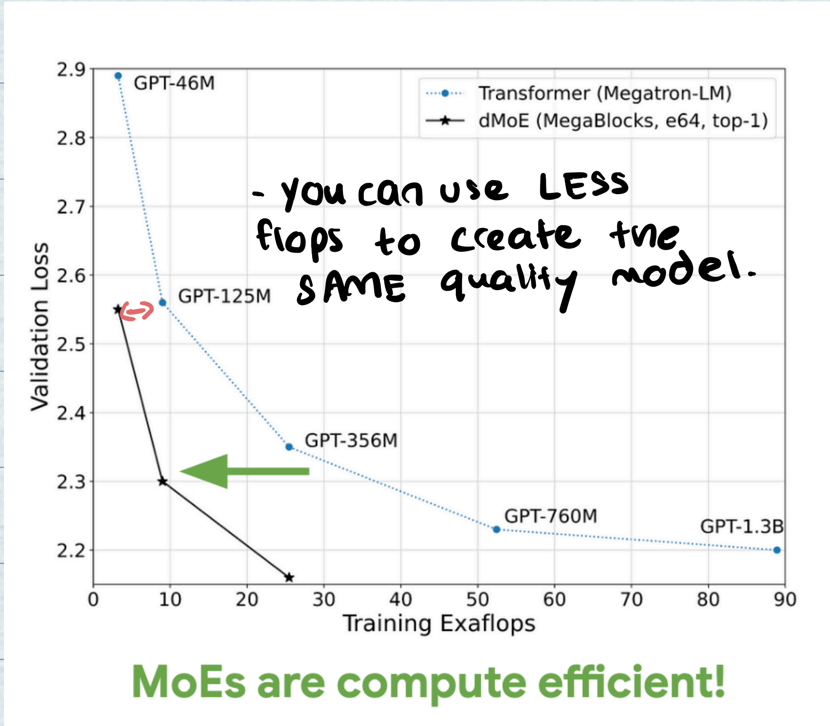
↳ this can be rescaled to a n-bit integer

QUICK OVERVIEW OF KNOWLEDGE DISTILLATION

- general idea: guide large "teacher-model" to teach training of "student" model
  - can do this at many dif. layers, including:
    - output logits
    - intermediate weights
    - intermediate features
    - gradients
    - sparsity patterns

More Mixture of Experts (what trevor didn't get to)

* Efficiency of MOEs in Inference vs. Training



- you can use LESS flops to create the SAME quality model.

* MOEs will be compute efficient - use less compute for same quality

- you can Actually translate theoretical compute gain into Runtime wins - large batches

RECAP on AI of Dense model vs. AI of MoE Expert:

$$ai_{dense} = \frac{ops}{bytes} = \frac{N_{tokens} \cdot d_{model} \cdot d_{ff} \cdot 2}{N_{tokens} \cdot d_{model} + d_{model} \cdot d_{ff} + N_{tokens} \cdot d_{ff}}$$

assumes you write out intermediate result

2 projections: $((N_t \times d_{model})(d_{model} \times d_{ff})) \cdot (d_{ff} \cdot d_{model})$

$$ai_{moe} \approx \frac{top_k}{num_{experts}} \cdot ai_{dense}$$

→ we do FFN compute topk times, but load numexperts times as many weights

* so this means moes have LOW AI compared to dense (will use HW ineffectively)

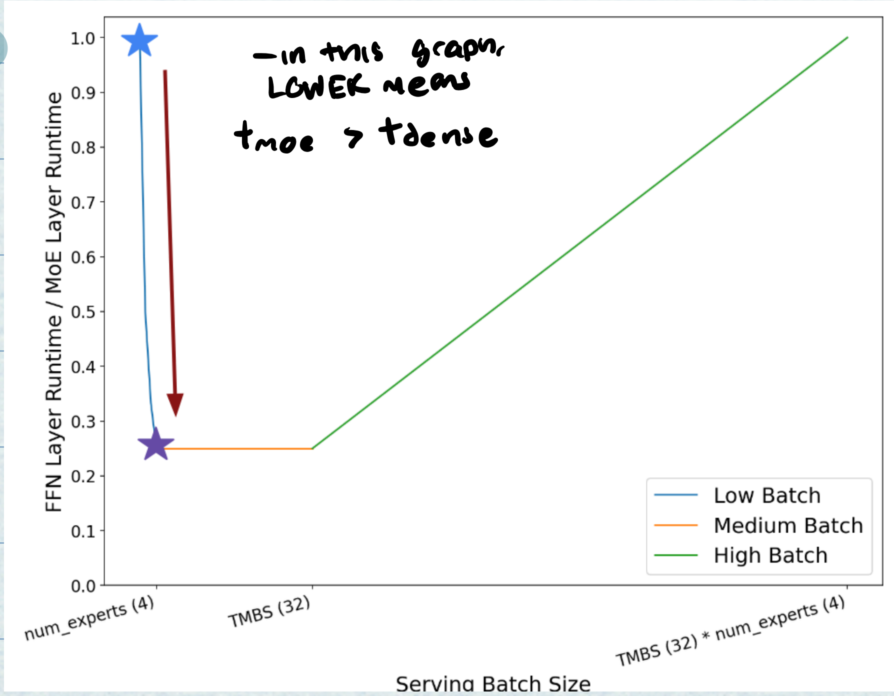
↳ as trevor said, expert-level parallelism can help w/ this!

* At serving time – batch size can be small which CHANGES speedup calculus

Region 1 = low batch serving

↳ perf limiter will be MEMORY to load weights

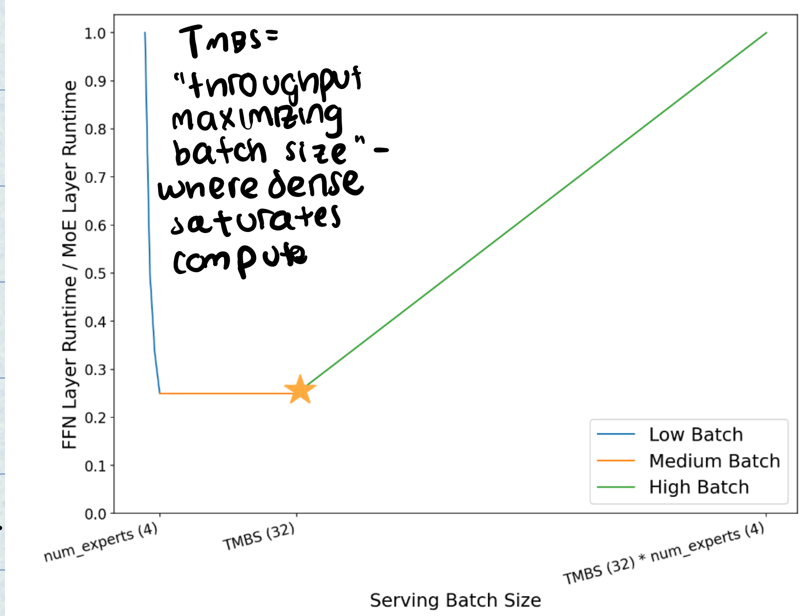
B=1 → MoE / dense touch same # of weights (1 expert's weights are loaded)



B = num_experts → all experts are touched, so MoE loads topk as many weight

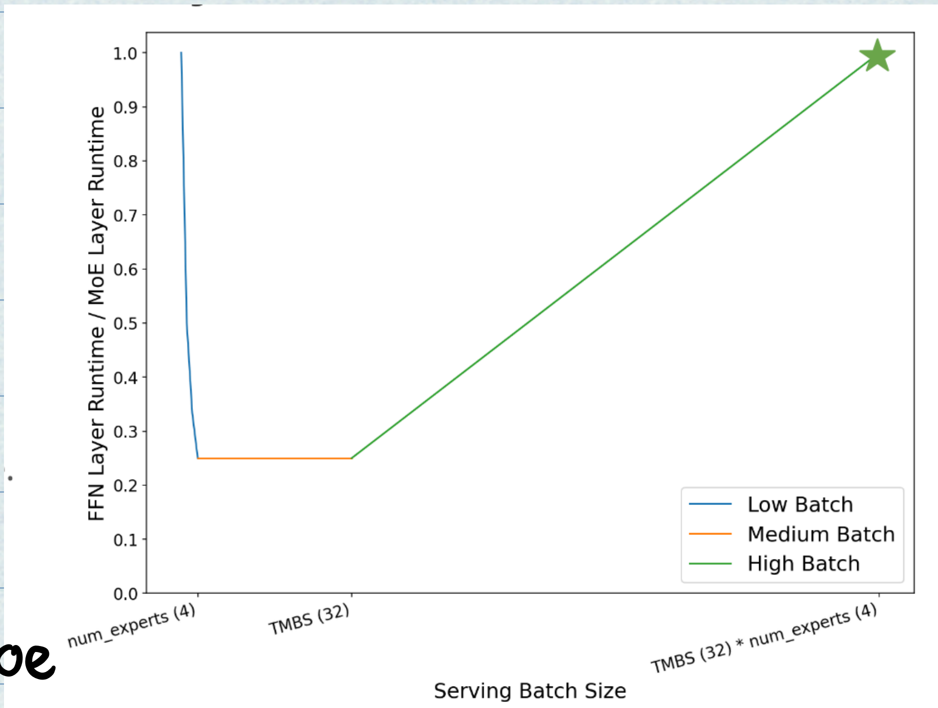
Region 2 – medium batch serving

– here $\frac{t_{dense}}{t_{moe}}$ is constant
– as all experts are loaded, but compute utilization for dense ramps up
– recall – moe has LOWER AI (flops/byte) – so not yet saturating compute



Region3: High batch

- when batch=TMBS·num_experts- moe will saturate compute and $t_{dense} = t_{moe}$



* remember this MoE has overall many params - so is more accurate - so we may be happy w/ tradeoff even if we're in "inefficient" region in the middle