Quantization for Deep Learning

Karl Stratos
me@karlstratos.com

Last updated: August, 2025

Abstract

Model weights are usually encoded as 16-bit floats (Appendix A). We trade accuracy for compression by mapping weights to lower-bit codebooks and dequantizing them on the fly at inference. The Pareto frontier depends on codebook design, quantization granularity, and quantization-aware training.

Contents

1 Quantization

Let $\mathcal{X} \subseteq [X_{\min}, X_{\max}] \subset \mathbb{R}$ denote B_{base}-bit floats. Given a codebook $\mathcal{Z} \subseteq [Z_{\min}, Z_{\max}] \subset \mathbb{R}$ that is representable with $B < B_{\text{base}}$ bits (thus up to 2^B distinct codes), **quantization** $Q : \mathcal{X} \to \mathcal{Z}$ maps each $x \in \mathcal{X}$ to a code $z \in \mathcal{Z}$. The (approximate) inverse $D : \mathcal{Z} \to \mathcal{X}$ is **dequantization**. This framework underlies several use cases:

What	$\mathcal{X}$	When	Why	How
Post-training quant	Model weights	Inference	Cut memory/IO	Store codes, dequant on the fly
Low-bit training	Model weights/acts	Training	Faster matmuls	INT8/FP8 matmul (dynamic scale)
Optim state quant	Optim states	Training	Cut optim memory	Dequant-update-requant each step
Gradient quant	Gradients	Training (DDP)	Cut comm	Communicate quantized gradients

1.1 Affine Quantization

In general, we quantize and dequantize by[1]

$$Q(x) = \text{nearest}_{\mathcal{Z}}\left(\frac{x}{s} + b\right) \qquad \text{(scale and shift range, find nearest code)} \tag{1}$$

$$D(z) = s(z - b) \qquad \text{(reverse (1))} \tag{2}$$

Some formulations shift and scale instead (e.g., $\frac{x+b}{s}$), which is mathematically equivalent. The scale and bias terms (typically stored as B_1-bit floats for some $B_1 \geq 1$) are obtained by matching the endpoints. Solving the linear system $X_{\max} = s(Z_{\max} - b)$ and $X_{\min} = s(Z_{\min} - b)$ yields

$$s = \frac{X_{\max} - X_{\min}}{Z_{\max} - Z_{\min}} \qquad b = \frac{Z_{\min}X_{\max} - Z_{\max}X_{\min}}{X_{\max} - X_{\min}} \left(= Z_{\min} - \frac{X_{\min}}{s}\right) \tag{3}$$

The bias term is also called the **zero point**. Note that if $b \in \mathcal{Z}$, zero is quantized exactly: $D(Q(0)) = 0$.

1.1.1 Scale quantization

If $b = 0$, affine quantization simplifies to **scale quantization**:

$$Q(x) = \text{nearest}_{\mathcal{Z}}\left(\frac{x}{s}\right) \qquad \text{(scale range, find nearest code)} \tag{4}$$

$$D(z) = sz \qquad \text{(reverse (4))} \tag{5}$$

where $s = \frac{X_{\max}}{Z_{\max}}$. If $0 \in \mathcal{Z}$, then $Q(0) = 0$ and $D(0) = 0$, thus zero maintains its special identity (in addition to being quantized exactly)—a useful feature in deep learning where zero is pervasive (e.g., ReLU activations, masking, dropout). We naturally have scale quantization if the input/code ranges are either

- (Symmetric endpoints) $X_* = X_{\max} = -X_{\min}$ and $Z_* = Z_{\max} = -Z_{\min}$, or
- (Nonnegative) $X_{\min} = Z_{\min} = 0$

If $\mathcal{Z}$ is a nonuniformly spaced lookup table (LUT), we especially prefer to use symmetric endpoints to avoid shifting by $b \neq 0$ in the code space (2), which can distort the LUT's intended density near zero (e.g., resulting in a sign-flip).

1.2 Codebook

Intuitively, we choose $\mathcal{Z}$ to match the nature of $\mathcal{X}$. Some common cases (all resulting in scale quantization):

- (General $\mathcal{X}$) Use signed integers—one extreme often dropped to keep symmetry under two's complement.
- (Nonnegative $\mathcal{X}$) Use unsigned integers.
- (Distributional $\mathcal{X}$) Use quantiles (Appendix E).

$\mathcal{Z}$	$\lvert\mathcal{Z}\rvert$	$Z_{\min}$	$Z_{\max}$	spacing	scale quant	s	b
signed int	$2^B - 1$	$-2^{B-1} + 1$	$2^{B-1} - 1$	uniform grid	✓	$\frac{\text{absmax}(\mathcal{X})}{2^{B-1}-1}$	0
unsigned int	2^B	0	$2^B - 1$	uniform grid	✓ w/ $X_{\min} = 0$	$\frac{X_{\max}}{2^B-1}$	0
NormalFloat	2^B	-1	1	nonuniform LUT	✓	$\text{absmax}(\mathcal{X})$	0

Liu *et al.* [12] highlight the importance of choosing the right codebook for the given bit width, with a key trade-off between symmetric levels and zero inclusion. For $B \leq 2$, they favor balanced symmetric grids (no zero level; e.g., $-1.5, -0.5, 0.5, 1.5$); for $B \geq 3$, they favor zero-including grids.

[1] In the special case $\mathcal{X}$ and $\mathcal{Z}$ are both IEEE-compliant float formats, quantization can be achieved by bit shifting (e.g., Appendix D).

Figure 1: Deterministic vs stochastic rounding.

1.2.1 Deterministic rounding

Since $\text{nearest}_{\mathcal{Z}}(y) \in \mathcal{Z}$ expects that the input $y = \frac{x}{s} + b$ is already normalized to match the range of $\mathcal{Z}$, it can be computed in sublinear time. For instance, if $\mathcal{Z} = \{Z_{\min} \dots Z_{\max}\}$ is an interval of integers, we can just "round and clamp" in $O(1)$ time:

$$\text{nearest}_{\mathcal{Z}}(y) = y.\text{round}().\text{clamp}(Z_{\min}, Z_{\max}) \tag{6}$$

More generally, if $\mathcal{Z} = (l_1 \dots l_K)$ is an LUT of K levels $l_1 < \dots < l_K$, we can bucketize with midpoints $m_i = \frac{l_i + l_{i+1}}{2}$ in $O(\log K)$ time (i.e., binary search):

$$\begin{aligned} k &= 1 + \text{bucketize}(y.\text{clamp}(l_1, l_K), [m_1 \dots m_{K-1}]) \\ \text{nearest}_{\mathcal{Z}}(y) &= l_k \end{aligned} \tag{7}$$

1.2.2 Stochastic rounding

Deterministic rounding implies a biased estimator $\mathbf{E}[D(Q(x))|x] \neq x$. This is fine in a one-off setting (e.g., PTQ), but may be limiting in a training setting where we repeatedly quantize (e.g., gradients). **Stochastic rounding** yields a consistent estimator by allowing an input value to take the non-nearest code with probability proportional to normalized distance (Figure 1). If $c \leq y < c'$ is a normalized input between two codes c, c', it returns $z \in \{c, c'\}$ where

$$z = \begin{cases} c & \text{with probability } \frac{c'-y}{c'-c} \\ c' & \text{otherwise} \end{cases}$$

We can easily check that $\mathbf{E}[D(Q(x))|x] = x$.

1.2.3 Bit packing

If $\mathcal{Z}$ is `int8` or `uint8`, the low-bit storage of $z \in \mathcal{Z}$ is automatic with type casting (e.g., the stored int8 code actually occupies $B = 8$ bits). This freebie goes away if $\mathcal{Z}$ is a general LUT. For instance, for 2-bit quantization we may use any of the following for $\mathcal{Z} = (l_1, l_2, l_3, l_4)$:

$$\mathcal{Z} = \begin{cases} (-1.5, -0.5, 0.5, 1.5) & \text{(balanced LUT)} \\ (-2, -1, 0, 1) & \text{(unbalanced LUT)} \\ (-1.0, -0.3, 0.3, 1.0) & \text{(NF2)} \end{cases}$$

Since the codes themselves are general floats that may require > 2 bits, we store their indices `inds`. Unfortunately, most languages do not support a native data type for < 8-bit unsigned integers, so we typically first represent the indices in `uint8` (which only use the lowest bits and waste higher bits), then "pack" 4 of these values into a single byte `byte` (aka. **bit packing**). E.g., 2-bit quantization with $\mathcal{Z} = (-1.5, -0.5, 0.5, 1.5)$ gives

$x = (-4, 6, 0, -2)$	(`bfloat16`)	0<<0 ≡ 0 0 0 0 0 0 0 0
$z = (-1.5, 1.5, -0.5, -0.5)$	(implicit, *not* stored)	3<<2 ≡ 0 0 0 0 1 1 0 0
`inds` $= (0, 3, 1, 1)$	(`uint8`)	1<<4 ≡ 0 0 1 0 0 0 0 0
`byte` $= (92)$	(`uint8`, stored)	1<<6 ≡ 0 1 0 0 0 0 0 0
		92 ≡ 0 1 0 1 1 1 0 0

where 92 corresponds to `OR`-ing the shifted `uint8` representations of $(0, 3, 1, 1)$. At dequantization, the byte 92 can be unpacked into the original 4 indices $(0, 3, 1, 1)$ which are then used to look up $z = (-1.5, 1.5, -0.5, -0.5)$. Similar bit-packing strategies work for other bits (e.g., two 4-bit indices are packed into 1 byte, eight 3-bit indices are packed into 3 bytes).

1.3 Partitioned Quantization

In practice, we partition $\mathcal{X} = \mathcal{X}_1 \cup \cdots \cup \mathcal{X}_n$ into $n \geq 1$ clusters and quantize each cluster $\mathcal{X}_i$ independently to reduce quantization error. But this introduces n scales to store in memory. We choose granularity so each unit has a small dynamic range (e.g., per-channel or fixed-size blocks), which reduces uniform-quantization error. Since $\mathcal{X}$ is typically already grouped into semantically meaningful tensors corresponding to different layers (e.g., a 2D matrix in a linear layer, a 3D filter in a convolutional layer), natural scaling schemes include:

- **Tensor-wise**: Each tensor is quantized independently.
- **Group-wise**: A tensor is further split into semantically coherent groups (e.g., row/column-wise, channel-wise, head-wise [15]), each of which is quantized independently.
- **Block-wise**: A tensor is split into blocks of equal size M (a hyperparameter), each of which is quantized independently.
- **Hybrid**: A tensor is first split into groups (e.g., along a specified axis), then each group is split into blocks of a specified size [16].

Block-wise scaling makes all quantization units have the same size M, and is convenient for calculating how much total memory is needed. If we quantize T values in a tensor into a B-bit data type in blocks of size M_1 using B_1-bit scales, we need

$$\left(B + \frac{B_1}{M_1}\right) T \tag{8}$$

bits to store the quantized values. For example, if we quantize 32-bit to 8-bit with block size $M_1 = 64$ and $B_1 = 32$ scale bits, each parameter requires $8 + \frac{32}{64} = 8.5$ bits; the compression factor is $\frac{32}{8.5} \approx 3.76$.

1.3.1 Double quantization

Suppose we quantize the scales again [3]. That is, we first quantize T values to B bits in blocks of size M_1. We quantize the resulting $\frac{T}{M_1}$ scales to B_2 bits in blocks of size M_2. The resulting $\frac{T}{M_1 M_2}$ "meta-scales" are stored as B_3-bit floats. Note that we never store the first-level scales; they are dequantized from B_2-bit codes and B_3-bit meta-scales. We need

$$\left(B + \frac{B_2}{M_1} + \frac{B_3}{M_1 M_2}\right) T \tag{9}$$

bits to store the doubly quantized values. The main idea is that scales are "easier" to quantize. All blocks have similar statistics (since they are in the same layer), thus the scales $s \propto X_{\max}$ vary slowly and can be quantized coarsely. We can capture their variance with high-bit meta-scales, but the number of meta-scales is tiny. If we quantize 32-bit to 8-bit with block size $M_1 = 64$ as before, but instead of storing the scales as 32-bit floats we quantize them to 8-bit with block size $M_2 = 256$ and store the meta-scales as 32-bit floats, each parameter requires $8 + \frac{8}{64} + \frac{32}{64 \cdot 256} \approx 8.13$ bits; the compression factor is 3.94. The final quantization error is likely no worse.

2 Post-Training Quantization (PTQ)

In PTQ, we quantize trained model weights and store only the resulting codes in $B < B_{\text{base}}$ bits. We typically quantize each linear layer $W \in \mathbb{R}^{d \times d'}$ independently.[2] During inference, we dequantize the codes on the fly. We can use custom [5] (e.g., see Puzzle 12 by Sasha Rush) or precompiled [6] GPU kernels to enhance dequantization efficiency.

2.1 Post-Training Calibration

We can achieve PTQ "instantly" by calibrating a quantization configuration $\phi \in \Phi$ (e.g., codebook, block size, some additional parameters), manually or using grid search. Let $R_\phi(W) = D_\phi(Q_\phi(W))$ denote the quantized

[2] Bias vectors are typically kept in higher precision since their memory share is negligible. Modern LLMs often do not even have bias vectors.

reconstruction of W under ϕ. Various metrics have been proposed:

$$\min_{\phi \in \Phi} ||W - R_\phi(W)||_F^2 \qquad \text{(dataless)} \tag{10}$$

$$\min_{\phi \in \Phi} ||XW - XR_\phi(W)||_F^2 \qquad \text{(output-calibrated)} \tag{11}$$

$$\min_{\phi \in \Phi} \left|\left|\widehat{F}(X,W)^{1/2} \odot (W - R_\phi(W))\right|\right|_F^2 \qquad \text{(loss-calibrated)} \tag{12}$$

We write (10) to broadly refer to early non-data-driven approaches. For instance, `LLM.int8()` identifies the issue of outlier features by manual inspection and avoids quantizing them.[3] (11) calibrates the layer output (e.g., AWQ). (12) calibrates the target loss; $\widehat{F}(X,W)^{1/2}$ is the usual Fisher approximation of the Hessian (e.g., SqueezeLLM). However, these metrics are at best rough guiding principles; final decisions are made based on downstream performance.

2.2 Quantization-Aware Training (QAT)

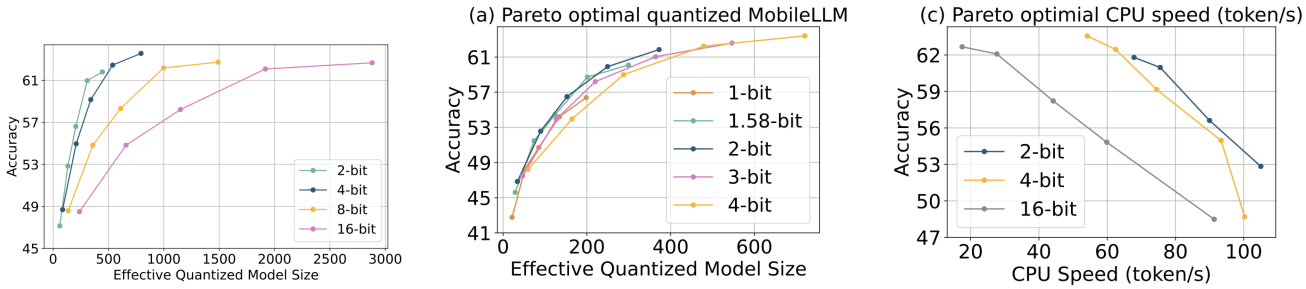
We can dramatically enhance the performance of a quantized model by QAT, though we lose some flexibility (e.g., we would need another QAT session if we want to change the codebook or vary the compression ratio). Specifically, we treat the quantized reconstruction as an additional training-only layer ("fake-quant"):

$$R(W) = s\left(\text{nearest}_{\mathcal{Z}}\left(\frac{W}{s} + b\right) - b\right) \tag{13}$$

where s, b are derived deterministically (3) (we assume tensor-wise scaling for simplicity, but the same concept holds for block-wise). The non-differentiability of (13) is handled by the straight-through estimator. After training, we remove the layer and obtain $R(W)$ from dequantization. (13) typically makes the scale term learnable as well ("learned ranges") [4]. For instance, if we assume 2-bit scale quantization with $\mathcal{Z} = (-1.5, -0.5, 0.5, 1.5)$, we may use

$$R(W, f) = \frac{f \times \text{absmax}(W)}{1.5}\left(\text{nearest}_{\mathcal{Z}}\left(\frac{(1.5)W}{f \times \text{absmax}(W)}\right)\right) \tag{14}$$

where the additional parameter $f \in \mathbb{R}$ is learned (with careful initialization, see [18]). Liu *et al.* [12] find that keeping the first 90% of training QAT-free and introducing QAT in the last 10% gives the best performance. Importantly, they find that while 4-bit QAT is "optimal" in the sense that it achieves almost lossless compression, the Pareto frontier in the accuracy-compression tradeoff is actually achieved by sub-4-bit QAT (ternary, 2-bit, 3-bit). See the plots from the paper:



2.3 Adapter Fine-Tuning

We can further compensate for quality degradation by fine-tuning an adaptor, typically LoRA. The linear layer transforms the input $X \mapsto Y$ as

$$Y = XD(W_{\text{quantized}}) + \frac{\alpha}{r}(XL_1)L_2$$

where the low-rank adapter weights $L_1 \in \mathbb{R}^{d \times r}$ and $L_2 \in \mathbb{R}^{r \times d'}$ are fine-tuned in base precision. In post-training calibration, L_1, L_2 are fine-tuned for downstream tasks. QLoRA [3] treats this as a pure pipeline, holding `NF4` double quantization fixed. LQ-LoRA [6] jointly optimizes quantization and LoRA with a gradient-free algorithm

[3] Outlier features are an issue in their setting because they quantize *activations* to `int8` with no training. It is less relevant in other methods which do not quantize activations and/or use training to fix it (QAT).

(minimizing (12) by alternating between SVD and quantization). Adapters are also used after QAT: the quantized weights are frozen and only LoRA parameters are trained to recover quality [18]. Although one could fold LoRA into the base weights by re-quantizing $W_{\text{quantized}} + \frac{\alpha}{r} L_1 L_2$, doing so will lose the quality recovery from keeping LoRA in base precision (as well as the modularity of adapters), so it is not desirable in practice.

References

[1] Dettmers, T. and Zettlemoyer, L. (2023). The case for 4-bit precision: k-bit inference scaling laws. In *International Conference on Machine Learning*, pages 7750–7774. PMLR.

[2] Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. (2022). Llm. int8 (): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*.

[3] Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. *arXiv preprint arXiv:2305.14314*.

[4] Esser, S. K., McKinstry, J. L., Bablani, D., Appuswamy, R., and Modha, D. S. (2020). Learned step size quantization. In *International Conference on Learning Representations*.

[5] Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. (2022). Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*.

[6] Guo, H., Greengard, P., Xing, E. P., and Kim, Y. (2023). Lq-lora: Low-rank plus quantized matrix decomposition for efficient language model finetuning. *arXiv preprint arXiv:2311.12023*.

[7] Hu, E. J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2022). LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

[8] Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M. W., and Keutzer, K. (2023). Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*.

[9] Kunstner, F., Hennig, P., and Balles, L. (2019). Limitations of the empirical fisher approximation for natural gradient descent. *Advances in neural information processing systems*, **32**.

[10] Kuzmin, A., Van Baalen, M., Ren, Y., Nagel, M., Peters, J., and Blankevoort, T. (2022). Fp8 quantization: The power of the exponent. *Advances in Neural Information Processing Systems*, **35**, 14651–14662.

[11] Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. (2023). Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*.

[12] Liu, Z., Zhao, C., Huang, H., Chen, S., Zhang, J., Zhao, J., Roy, S., Jin, L., Xiong, Y., Shi, Y., *et al.* (2025). Paretoq: Scaling laws in extremely low-bit llm quantization. *arXiv preprint arXiv:2502.02631*.

[13] Micikevicius, P., Stosic, D., Burgess, N., Cornea, M., Dubey, P., Grisenthwaite, R., Ha, S., Heinecke, A., Judd, P., Kamalu, J., *et al.* (2022). Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433*.

[14] Peng, H., Wu, K., Wei, Y., Zhao, G., Yang, Y., Liu, Z., Xiong, Y., Yang, Z., Ni, B., Hu, J., *et al.* (2023). Fp8-lm: Training fp8 large language models. *arXiv preprint arXiv:2310.18313*.

[15] Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. (2020). Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821.

[16] Sheng, Y., Zheng, L., Yuan, B., Li, Z., Ryabinin, M., Chen, B., Liang, P., Ré, C., Stoica, I., and Zhang, C. (2023). Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR.

[17] Yoshida, D. (2023). Nf4 isn't information theoretically optimal (and that's good). *arXiv preprint arXiv:2306.06965*.

[18] Zhou, H., Hornberger, E., Guo, P., Zhou, X., Wang, S., Wang, X., He, Y., Chang, X., Rauch, R., D'hauwe, L., *et al.* (2025). Apple intelligence foundation language models: Tech report 2025. *arXiv preprint arXiv:2507.13575*.

A IEEE-Compliant Floats

Pick a whole number $\beta \geq 2$ called the **base** or **radix**. Pick integers $p, e \geq 1$ for precision and exponent levels. A normal **floating-point number**, in short **float**[4], approximates a real value $x \in \mathbb{R}$ using

- 1 sign bit: $s \in \{0, 1\}$
- $p+1$ **significand** (or **mantissa**) digits: $f = (f_0, f_1 \dots f_p) \in \{0 \dots \beta - 1\}^{p+1}$ where $f_0 \geq 1$ is a leading digit
- e exponent digits: $z = (z_1 \dots z_e) \in \{0 \dots \beta - 1\}^e$

The exponent value is computed by

$$E = \underbrace{\left(z_1 \beta^{e-1} + \cdots + z_{e-1}\beta + z_e\right)}_{\{0,1,\dots,\beta^e-1\}} - \underbrace{\left\lfloor \frac{\beta^e - 1}{2} \right\rfloor}_{B} \in \begin{cases} \{-B, \dots, B\} \cup \{B+1\} & \text{if } \beta \text{ is even (asymmetric)} \\ \{-B, \dots, B\} & \text{if } \beta \text{ is odd} \end{cases} \tag{15}$$

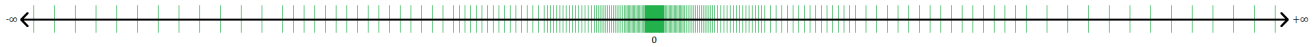
where the bias B is chosen to center the range around zero.[5] Thus E ranges over β^e integers ($2B+2$ if β is even, $2B+1$ if odd). The fractional value is computed by

$$F = \underbrace{f_0}_{\{1 \dots \beta-1\}} + \underbrace{f_1\beta^{-1} + \cdots + f_p\beta^{-p}}_{\left\{\frac{0}{\beta^p}, \frac{1}{\beta^p}, \dots, \frac{\beta^p-1}{\beta^p}\right\}} = f_0 + \frac{k}{\beta^p} \qquad k := \sum_{i=1}^{p} f_i \beta^{p-i} \in \{0, 1, \dots, \beta^p - 1\} \tag{16}$$

which ranges over the $(\beta-1)\beta^p$ evenly spaced values in $[1, \beta)$ with spacing $\Delta = \beta^{-p}$. For instance with $\beta = 2$ and $p = 2$, we have 4 values $\{1, 1.25, 1.5, 1.75\}$; with $\beta = 10$ and $p = 3$, we have 9000 values $\{1, 1.001, \dots, 9.998, 9.999\}$. We compute the final float by $\hat{x} = (-1)^s \times F \times \beta^E$. Assuming $s = 0$ WLOG since it simply changes the sign, for a fixed exponent E the set of positive floats are

$$\hat{x} = F \times \beta^E = \left(f_0 + \frac{k}{\beta^p}\right)\beta^E = f_0\beta^E + k\beta^{E-p} \qquad k \in \{0, 1, \dots, \beta^p - 1\} \tag{17}$$

They divide $[\beta^E, \beta^{E+1})$ uniformly into $(\beta-1)\beta^p$ parts with spacing $\Delta = \beta^{E-p}$, regardless of the value of E. This has the consequence of having fewer floats available as we move further away from zero (e.g., with $\beta = 10$ and $p = 2$, we have 900 floats for both $[1, 10)$ and $[100, 1000)$). Here is an illustration from Wikipedia:



This nonuniform spacing allows us to express extremes more effectively than uniform spacing, especially when there is no prior assumption on the range of values (e.g., it could be molecular or cosmic scale).

A.1 Subnormal Floats and Signed Zeros

Note that (17) cannot be zero: the smallest positive normal float is $\beta^{-B} \leq 1$. We can go "deeper" into the origin by lifting the constraint $f_0 \geq 1$. Setting $f_0 = 0$ and $E = -B$, we define the set of (positive) **subnormal floats**

$$\hat{x} = k\beta^{-B-p} \qquad k \in \{0, 1, \dots, \beta^p - 1\} \tag{18}$$

which further divide $[0, \beta^{-B})$ uniformly into β^p parts with spacing $\Delta = \beta^{-B-p}$. When $k = 0$ (i.e., $f_i = 0$ for all i), (18) finally becomes zero. Since the model must use the sign bit $s \in \{0, 1\}$, it defines *signed* zeros $\hat{x} = \pm 0$.

A.2 Relative Rounding Error

Any positive real value $\beta^{-B} < x < \beta^B$ is between the normal floats $\hat{x}_L \geq \beta^E$ and $\hat{x}_R = \hat{x}_L + \beta^{E-p}$ for some $-B \leq E \leq B$. Rounding x to the nearest $\hat{x} \in \{\hat{x}_L, \hat{x}_R\}$, we have the absolute error bounded as $|x - \hat{x}| \leq \frac{1}{2}\beta^{E-p}$. As the bound becomes looser away from zero, a more popular metric is the **relative rounding error**

$$1 \geq \left|1 - \frac{\hat{x}}{x}\right| = \left|\frac{x - \hat{x}}{x}\right| = \frac{|x - \hat{x}|}{x} \leq \frac{\beta^{E-p}}{2x} \leq \frac{\beta^{E-p}}{2\hat{x}_L} \leq \frac{\beta^{E-p}}{2\beta^E} = \frac{1}{2}\beta^{-p} =: \epsilon_{\text{mach}} \tag{19}$$

The last term ϵ_{mach} is called **machine epsilon** or **unit roundoff** representing the maximum error when rounding to 1.[6]

[4] Not to be confused with the `float` data type in C which specifically refers to the 32-bit floating-point format in binary base

[5] The first term takes the max value of $\beta^e - 1$ when $z_1\beta^{e-1} + \cdots + z_{e-1}\beta + z_e = (\beta-1)\left(\beta^{e-1} + \cdots + \beta + 1\right) = (\beta-1)\frac{\beta^e-1}{\beta-1} = \beta^e - 1$.

[6] For $0 < x < \beta^{-B}$ (subnormal region), the absolute error is $|x - \hat{x}| \leq \frac{1}{2}\beta^{-B-p}$. The relative error is not uniformly small, and it equals 1 for $0 < x \leq \frac{1}{2}\beta^{-B-p}$ (rounds to 0).

Correct rounding enforcement IEEE 754 mandates correct rounding: each basic arithmetic operation must produce the same result as if computed exactly and then rounded to the target format. Implementations typically ensure this by using extra internal precision or algorithms that can guarantee correct rounding. On x86, the legacy x87 FPU uses an 80-bit extended format internally; on GPUs there is usually no exposed extended format, but the required operations are still correctly rounded (e.g., via FMA and sufficient internal precision). Transcendental functions are not universally required to be correctly rounded.

A.3 Binary Base

While IEEE-compliant floats can be defined using any integer base $\beta \geq 2$, the binary base $\beta = 2$ is the dominant choice for hardware efficiency and consistency with integer representation (which is binary).[7] Plugging in $\beta = 2$ substantially simplifies the equations. First, the leading significand bit is now deterministically $f_0 = 1$ for normal, so we may assume only p precision bits $f = \{0, 1\}^p$ rather than $p + 1$. Under the IEEE 754 standards, $z \in \{0^e, 1^e\}$ is reserved for special values, yielding the following classification of binary floats

$$\hat{x} = \begin{cases} (-1)^s \times \left(2^E + k2^{E-p}\right) & \text{if } z \notin \{0_e, 1_e\} \textbf{ (normal)} \\ (-1)^s \times k2^{E_{\min}-p} & \text{if } z = 0_e \text{ and } f \neq 0_p \textbf{ (subnormal)} \\ (-1)^s \times 0 & \text{if } z = 0_e \text{ and } f = 0_p \textbf{ (signed zeros)} \\ (-1)^s \times \infty & \text{if } z = 1_e \text{ and } f = 0_p \textbf{ (signed infinities)} \\ \texttt{NaN}(f) & \text{if } z = 1_e \text{ and } f \neq 0_p \textbf{ (NaNs)} \end{cases} \tag{20}$$

where $k = \sum_{i=1}^{p} f_i 2^{p-i} \in \{0 \dots 2^p - 1\}$. We can standardize relevant constants as

$$\begin{aligned} &\text{(machine epsilon)} && \epsilon_{\text{mach}} = 2^{-(p+1)} \\ &\text{(exponent range)} && E_{\min} = 1 - E_{\max} && E_{\max} = 2^{e-1} - 1 \\ &\text{(normal range)} && N_{\min} = 2^{E_{\min}} && N_{\max} = (1 - \epsilon_{\text{mach}})2^{E_{\max}+1} && (21) \\ &\text{(subnormal range)} && S_{\min} = 2^{E_{\min}-p} && S_{\max} = (1 - 2\epsilon_{\text{mach}})2^{E_{\min}} && (22) \end{aligned}$$

The signed zeros work as expected in most cases (e.g., $0 = -0$), but there are certain corner cases such as $\frac{1}{0} \neq \frac{1}{-0}$ (the former evaluates to ∞ while the latter to $-\infty$). NaNs occur as outputs of illegal operations (e.g., $\frac{0}{0}$, $\log(-1)$, $\infty \times 0$) and are categorized into either "signaling" (i.e., throw an exception) or "quiet" types based on the significand f. NaNs propagate: any operation involving a NaN generally outputs a NaN.

A.3.1 Example formats

We summarize some binary formats below. For readability, we approximate small or large values by powers of *ten* (e.g., for `float16` we have $S_{\min} = 2^{-24} \approx 5.96 \times 10^{-8}$).

Name	B	e	p	$E_{\min}$	$E_{\max}$	$S_{\min}$	$N_{\min}$	$N_{\max}$	ϵ_{mach}
`float4`	4	2	1	0	1	0.5	1	3	0.25
`float8`	8	4	3	−6	7	≈ 0.002	≈ 0.02	240	0.0625
`E4M3` (non-compliant*)	8	4	3	−6	7	≈ 0.002	≈ 0.02	448*	0.0625
`E5M2`	8	5	2	−14	15	≈ 0.00002	≈ 0.00006	57344	0.125
`float16` (half precision)	16	5	10	−14	15	$\approx 10^{-8}$	≈ 0.00006	65504	≈ 0.0005
`bfloat16`	16	8	7	−126	127	$\approx 10^{-45}$	$\approx 10^{-38}$	$\approx 10^{38}$	≈ 0.004
`float32` (single precision)	32	8	23	−126	127	$\approx 10^{-45}$	$\approx 10^{-38}$	$\approx 10^{38}$	$\approx 10^{-8}$
`float64` (double precision)	64	11	52	−1022	1023	$\approx 10^{-324}$	$\approx 10^{-308}$	$\approx 10^{308}$	$\approx 10^{-16}$

Double precision (`float64`) can express extreme values and is useful for precision-critical tasks such as gradient checks (Appendix C). Single precision (`float32`) is often the default format in deep learning (e.g., PyTorch FloatTensors). Half precision (`float16`) halves the memory requirement, but its limited range is often ill-suited for model training. In response, `bfloat16` allocates more bits to the exponent to match the range of single precision. The 8-bit formats have only $2^8 = 256$ numbers to represent $\mathbb{R}$ (e.g., this table). How to allocate the precious bits (i.e., $8 = e + p$) is task-dependent [10]. `E4M3` increases the range of `float8` by deviating from IEEE 754 (e.g., it has no infinities) [13]. An even more extreme situation is 4-bit formats which have only $2^4 = 16$ numbers. `float4` is the lowest-bit format that satisfies all IEEE 754 standards, but is pitifully limited. Recent works explore more useful definitions of 4-bit or even lower-bit floats based on quantile quantization (Appendix E).

[7] An exception is when precision with respect to a specific nonbinary base is paramount (e.g., $\beta = 10$ in finance).

A.3.2 Quirky Examples

We compile a few examples in Python (64-bit floats) to illustrate the quirky behavior of float arithmetic.

```
format(0.1, '.25')               # 0.1000000000000000055511151 (64-bit)
format(np.float32(0.1), '.25')   # 0.1000000014901161193847656
(0.1 + 0.2) + 0.3 == 0.1 + (0.2 + 0.3)  # False
262144 + 0.01 == 262144  # False (64-bit)
np.float32(262144) + np.float32(0.01) == np.float32(262144)  # True
np.zeros(1) == -np.zeros(1)  # True
np.ones(1) / np.zeros(1) ==  np.ones(1) / -np.zeros(1)  # False (inf vs -inf)
np.sqrt((3 + 4 + 1 * 3) / 2) + np.nan  + 7 * 3 + 1  # nan
```

The examples demonstrate that (1) decimal fractions are not precisely represented in binary base; (2) additions (and multiplications) are not associative due to rounding errors; (3) adding large and small numbers is more susceptible to rounding errors than numbers in a similar range; (4) NaNs propagate.

A.3.3 Non-numerical data types

Numerical data types, such as floats for real numbers, allow us to directly control memory usage by choosing a specific precision level for storing values. It is interesting to contrast them with non-numerical data types, such as characters, which do not allow for such control. See Appendix B for an overview of how (Unicode) characters are stored in memory.

B Unicode Characters

The Unicode Standard defines a set $\mathcal{U}$ ("codespace") of $1,114,112$ characters ("code points"), meant to capture all of the world's major writing systems. The characters are enumerated in hexadecimal, starting from U+0000 to U+10FFFF. By convention, the prefix "U+" signals a Unicode character and that at least four digits are written; leading zeros beyond four digits may be omitted. The characters are partitioned into 17 "planes" $\mathcal{U} = \mathcal{U}_1 \cup \cdots \cup \mathcal{U}_{17}$ identified by the first two digits $\{00 \ldots 10\}$ (so $|\mathcal{U}_k| = 16^4$). $\mathcal{U}_1$ contains characters for almost all modern languages and is called the **Basic Multilingual Plane (BMP)** (other planes are called supplementary). For historical reasons, some characters are disallowed, leaving $\mathcal{U}_{\text{valid}} \subset \mathcal{U}$ of 1,112,064 valid characters.

Naively, a Unicode character can be stored as an integer using 3 bytes (to express all > 1 million values). Instead, we use a variable-length scheme called **UTF-8**. The encoding algorithm is given below (source: Wikipedia):[8]

Code point ↔ UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+010000	[b]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

BMP	1-byte (128)	= ASCII
	2-byte (1,920)	⊃ Latin
	3-byte (61,440)	⊃ CJK
Supp.	4-byte (1,048,576)	⊃ Emojis

The 256 possible values of a byte are typically expressed as 2 hexadecimal digits $\{00 \ldots FF\}$ rather than an 8-bit vector. Thus the final encoding of a Unicode character is a sequence of $\{2, 4, 6, 8\}$ hexadecimal digits, as in

Unicode point	Expression	UTF-8 byte sequence (binary)	UTF-8 byte sequence (hexadecimal)
U+0041	A	01000001	41
U+03B8	θ	11001110 10111000	CE B8
U+C0B0	산	11101100 10000010 10110000	EC 82 B0
U+1F617	😗	11110000 10011111 10011000 10010111	F0 9F 98 97

Another quirk in UTF-8 encoding is that some code points function as combining marks. For instance, the expression é can be achieved by either U+00E9 or [U+0065, U+0301] (corresponding to [e, ´]).

[8] Note that the encoding uses the prefixes 0, 110, 1110, 11110 in the first byte to signal the number of bytes, and the prefix 10 in other bytes to signal the continuing byte. Any bit vector that does not conform to this format cannot be decoded to a Unicode character. Also note that this encoding is not maximally space-efficient. For instance, the third digit in the 2-byte characters ranges between 8 and F and can be encoded using 3 bits, but it is encoded using 4 bits.

B.1 Unicode Tokenization

Like quantizing and dequantizing real values, we can similarly define compression methods for Unicode strings in the context of tokenization (but exact rather than lossy). A Unicode **tokenizer** is a tuple $(\mathcal{V}, \text{Enc}, \text{Dec})$ where $\mathcal{V} = \{1 \dots V\}$ is the encoding space ("vocabulary"), $\text{Enc} : \mathcal{U}^+ \to \mathcal{V}^+$ and $\text{Dec} : \mathcal{V}^+ \to \mathcal{U}^+$ satisfy

$$x = \text{Dec}(\text{Enc}(x)) \qquad \forall x \in \mathcal{U}^+ \tag{23}$$

We want a small V and a short $\text{Enc}(x) \in \mathcal{V}^+$ (on average over x) while satisfying (23). This is because in the context of language models, V is the size of the softmax and $|\text{Enc}(x)|$ is the length of the tokenized input sequence, both of which are critical performance bottlenecks. Example tokenizers include

	$\text{Enc}(x)$	$\text{Dec}(z)$	V	$\lvert\text{Enc}(x)\rvert$
Character-level	$z = x$	z	1,114,112	$\lvert x\rvert$
Byte-level	$z = \text{bytes}(x)$	z.decode('utf-8')	256	$\lvert\text{bytes}(x)\rvert$
Rule-based	$z = \text{rule}_e(x)$	$\text{rule}_d(z)$	custom	custom
BPE	$z = \textbf{EncodeBPE}(\mathcal{V}, x)$	$\mathcal{V}^{-1}(z)$.decode('utf-8')	User specified	$\lvert\textbf{EncodeBPE}(\mathcal{V}, x)\rvert$

(e.g., a rule-based tokenizer may split Unicode strings by the whitespace and language-specific grammar). The dominant choice in practice is the **byte-pair encoding (BPE)** tokenizer, which allows for a customizable vocab size and efficient encoding.

B.1.1 BPE tokenizer

Vocabulary. A BPE tokenizer assumes that $\mathcal{V}$ is a hierarchy of *ranked* byte sequences such that

1. $\{0 \dots 255\} \subset \mathcal{V}$ (i.e., all byte values are in the vocabulary)
2. If $\tau \in \{0 \dots 255\}^n$ with $n \geq 2$ has rank r in $\mathcal{V}$, then $\tau = \tau_1\tau_2$ for some $\tau_1, \tau_2 \in \mathcal{V}$ with ranks $r_1, r_2 < r$.

Encoder. Given any Unicode character sequence, the encoder decides how to partition the input sequence, then maps each byte subsequence to the corresponding ID in $\mathcal{V}$.

EncodeBPE
Input: $\mathcal{V}$, $x \in \mathcal{U}^+$
Output: $z \in \mathcal{V}^{|\text{bytes}(x)|-k}$ where $k \geq 0$ is the number of merges

1. $z \leftarrow \text{bytes}(x)$
2. While there is a bigram in z that belongs to $\mathcal{V}$, merge the one with the *smallest* rank (all its occurrences).
3. Return z.

Decoder. Return the concatenation of the byte sequences in $z \in \mathcal{V}^+$. Optionally decode as UTF-8; this succeeds for $z = \text{Enc}(x)$ with valid x, but may fail for arbitrary z (e.g., a lone continuation byte A9). In practice this is rare; one can enforce UTF-8 validity during decoding or use a reversible byte-Unicode mapping (as in GPT-2's byte-level BPE).

Training. Given a budget $V \geq 256$, we can "train" a BPE tokenizer from a corpus $x \in \mathcal{U}^+$ by solving

$$\mathcal{V}^\star = \underset{\mathcal{V}:\ |\mathcal{V}| \leq V}{\arg\min}\ |\textbf{EncodeBPE}(\mathcal{V}, x)| \tag{24}$$

That is, find a hierarchy of byte sequences that yield the shortest encoding length for the training corpus. From the vocabulary constraints, a standard greedy procedure (iteratively merging the most frequent bigram) gives an effective approximation to (24).[9]

TrainBPE
Input: $V \geq 256$, $x \in \mathcal{U}^+$ with $|\text{bytes}(x)| \geq V$
Output: approximation $\widehat{\mathcal{V}} \approx \mathcal{V}^\star$ in (24)

1. $z \leftarrow \text{bytes}(x)$, $\mathcal{V} \leftarrow (0, \dots, 255)$
2. While $|\mathcal{V}| < V$, merge the *most frequent* bigram in z (replacing all occurrences) and add it as the next element in $\mathcal{V}$. Recompute counts.
3. Return $\widehat{\mathcal{V}} \leftarrow \mathcal{V}$.

[9] Greedy merging fails on the example $z = [0, 0, 0, 1, 0, 0, 1]$.

Pre-tokenization. One important detail not shown above is a "pre-tokenization" step. Rather than allowing merges anywhere in the input $x \in \mathcal{U}^+$, we first split it into segments or "words" $w = (w_1 \dots w_T) \in (\mathcal{U}^+)^T$. This is done to reflect natural boundaries including but not limited to space (e.g., using the regex pattern popularized by GPT-2). For instance:

$$x = \text{"I'll buy 3 apples..."}$$
$$w = [\text{"I"}, \text{"'ll"}, \text{" buy"}, \text{" 3"}, \text{" apples"}, \text{"..."}]$$

Note that the space is preserved so that we recover x simply by concatenating w. Then, we only consider merging the byte sequences within a word. This can be viewed as exercising our prior of semantic boundaries. For instance, we will have "apple", " apple", and " apples" in $\mathcal{V}$, but not "3 apples" which is intuitively less reusable.

Pre-tokenization is also important to make merging efficient. The above algorithms have the runtime $O(|x|\, m)$ where m is the number of merges (a naive merge requires scanning the entire input x to update the bigram statistics). However, with pre-tokenization, we can (1) first create a set of distinct word types by pre-tokenizing x once, and (2) at each merge only scan through this set instead of the entire input. Thus the runtime is $O(|x| + Mm)$ where $M \ll |x|$ is the number of distinct word types. Since each word type is independent, we can parallelize the merges across word types.

C Gradient Checks

Let $L : \mathbb{R} \to \mathbb{R}$ be a loss viewed as a function of a single parameter $\theta \in \mathbb{R}$. Let $g_\theta \leftarrow \mathbf{Grad}(L, \theta)$ denote the output of an algorithm expected to compute $L'(\theta) \in \mathbb{R}$, the analytic gradient of L at θ (e.g., backpropagation). A gradient check compares g_θ to a *numerical* estimate of $L'(\theta)$, which can be obtained from the definition of a derivative:

$$L'(\theta) := \lim_{\epsilon \to 0^+} \frac{L(\theta+\epsilon) - L(\theta)}{\epsilon} \approx \frac{L(\theta+\hat{\epsilon}) - L(\theta)}{\hat{\epsilon}} =: \hat{g}^{\text{one}}_{\theta,\hat{\epsilon}} \tag{25}$$

where $\hat{\epsilon} > 0$ is some tiny value. Using the Taylor expansion $L(\theta+\hat{\epsilon}) = L(\theta) + \hat{\epsilon}L'(\theta) + \frac{1}{2}\hat{\epsilon}^2 L''(c)$ where $c \in [\theta, \theta+\hat{\epsilon}]$ is some constant, we can calculate the numerical error

$$\hat{g}^{\text{one}}_{\theta,\hat{\epsilon}} = \frac{L(\theta+\hat{\epsilon}) - L(\theta)}{\hat{\epsilon}} = \frac{\hat{\epsilon}L'(\theta) + \frac{1}{2}\hat{\epsilon}^2 L''(c)}{\hat{\epsilon}} = L'(\theta) + \frac{1}{2}\hat{\epsilon}L''(c) \qquad \Rightarrow \qquad \left|L'(\theta) - \hat{g}^{\text{one}}_{\theta,\hat{\epsilon}}\right| = O(\hat{\epsilon})$$

A better estimate is given by the two-sided version

$$\hat{g}^{\text{two}}_{\theta,\hat{\epsilon}} := \frac{L(\theta+\hat{\epsilon}) - L(\theta-\hat{\epsilon})}{2\hat{\epsilon}} \tag{26}$$

The symmetry of the approximation will yield an improvement. Since

$$L(\theta+\hat{\epsilon}) = L(\theta) + \hat{\epsilon}L'(\theta) + \frac{1}{2}\hat{\epsilon}^2 L''(\theta) + \frac{1}{6}\hat{\epsilon}^3 L'''(c')$$
$$L(\theta-\hat{\epsilon}) = L(\theta) - \hat{\epsilon}L'(\theta) + \frac{1}{2}\hat{\epsilon}^2 L''(\theta) - \frac{1}{6}\hat{\epsilon}^3 L'''(c'')$$

for some $c', c'' \in \mathbb{R}$, defining $C = L'''(c') + L'''(c'')$ we have

$$\hat{g}^{\text{two}}_{\theta,\hat{\epsilon}} = \frac{L(\theta+\hat{\epsilon}) - L(\theta-\hat{\epsilon})}{2\hat{\epsilon}} = \frac{2\hat{\epsilon}L'(\theta) + \frac{1}{3}\hat{\epsilon}^3 C}{2\hat{\epsilon}} = L'(\theta) + \frac{1}{6}\hat{\epsilon}^2 C \qquad \Rightarrow \qquad \left|L'(\theta) - \hat{g}^{\text{two}}_{\theta,\hat{\epsilon}}\right| = O(\hat{\epsilon}^2)$$

which shows that (26) is a much more accurate estimate of $L'(\theta)$ than (25) for a small $\hat{\epsilon}$. To account for different scales, the gradient check typically tests the relative error

$$\frac{\left|g_\theta - \hat{g}^{\text{two}}_{\theta,\hat{\epsilon}}\right|}{\max(|g_\theta|, |\hat{g}^{\text{two}}_{\theta,\hat{\epsilon}}|)} \le \tau \tag{27}$$

where $\tau > 0$ is some tolerance. If $g_\theta = L'(\theta)$ (i.e., the algorithm is implemented correctly), then (27) is $O(\hat{\epsilon}^2)$ up to scaling. For instance, if $\hat{\epsilon} = 10^{-5}$ then (27) can be as small as 10^{-10} in an idealized scenario where the magnitude of the gradient is about 1. In practice, 10^{-7} is deemed safe: see the note here for details.

A gradient check is an interesting unit test because even the reference answer (i.e., numerical gradient estimate) is not perfect. Clearly we wish to use an $\hat{\epsilon}$ as small as possible since that determines the accuracy of the numerical estimate, but it will cause numerical instability in (26) (or (25)) when it is *too* small. Similarly, if $L'(\theta)$ is too small then $\hat{g}^{\text{two}}_{\theta,\hat{\epsilon}}$ may underflow to zero, thus we may want to scale the loss $\alpha L(\theta)$ so that $|\hat{g}^{\text{two}}_{\theta,\hat{\epsilon}}| \approx 1$. A great way to combat these issues is to always use double precision, which is capable of expressing extreme values.

D Mixed-Precision Training

Mixed-precision training performs only precision-sensitive operations in `float32` and the rest in `bfloat16`.[10] The conversion is achieved by simply truncating and padding the significand:

$$Q(x) = x \texttt{ >> } 16 \quad (\texttt{bfloat16}) \qquad\qquad D(z) = z \texttt{ << } 16 \quad (\texttt{float32})$$

where `>>` and `<<` are the bit-shift operators. This works because both formats have the same number of exponent bits $e = 8$ (Section A.3.1). Quantization truncates the significand to the right ("rounding toward zero"). Dequantization pads the significand with zeros (no change in value).

Loss scaling. It is also typical to dynamically scale precision-critical values so that they are more representable in fewer bits. In training, gradients are precision-critical, and they can be scaled by scaling the loss immediately before backpropagation. A pseudocode of (automatic) mixed-precision training with adaptive gradient scaling is given below, following the torch.amp library.

Input: Initial shift $t = 16$
For each batch B in the training data iterator:

1. $L \leftarrow \text{ComputeLossAMP}(B)$ # Autocast based on operation types (e.g., 16 bits for matmul, 32 for log).
2. $(2^t \times L).\text{backward}()$ # Compute the gradients of a scaled loss.
3. For each gradient g: $g \leftarrow 2^{-t} \times g$ # Unscale the gradients.
4. If no inf/NaN appears in the gradients:
   (a) Update the parameters.
   (b) If no inf/NaN has appeared in any gradient for the past 2000 consecutive updates, set $t \leftarrow t + 1$.
5. Otherwise: set $t \leftarrow t - 1$.

Recent work has proposed 8-bit formats for mixed-precision model training [14]. With such a small number of bits, much more care is needed in scaling (e.g., per-tensor instead of global scaling).

E Quantile Quantization

An "information-theoretically optimal" quantization scheme with respect to a distribution **pop** over $x \in \mathbb{R}$ (in our case, x represents a model weight) using B bits is one that partitions $\mathbb{R}$ so that each of $K = 2^B$ bins contains an equal probability mass. Each bin is assigned some representative value (e.g., midpoint). Recall that if t_k is the k-th K-quantile, it means

$$F_{\mathbf{pop}}(t_k) := \Pr_{x \sim \mathbf{pop}}(x \leq t_k) = \frac{k}{K}$$

where $F_{\mathbf{pop}}$ is the cumulative distribution function (e.g., the median is the first 2-quantile). If $F_{\mathbf{pop}}$ is invertible,

$$t_k = F^{-1}_{\mathbf{pop}}\left(\frac{k}{K}\right)$$

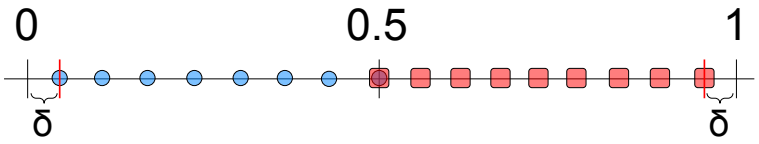
Instead of using the raw K-quantiles as quantization values, we may use the K midpoints of the $(K+1)$-quantiles as a more faithful approximation of the midpoints of the K-partition:

$$q_k = \frac{F^{-1}_{\mathbf{pop}}\left(\frac{k}{K+1}\right) + F^{-1}_{\mathbf{pop}}\left(\frac{k+1}{K+1}\right)}{2} \qquad \forall k = 1 \ldots K \tag{28}$$

E.1 NormalFloat (NF)

Estimating the quantiles of an unknown distribution from samples (of weights) is susceptible to large errors for outliers. The authors of QLoRA instead assume that $\mathbf{pop} = \mathcal{N}(0, 1)$ [3]. They propose a quantization scheme called 4-bit NormalFloat (`NF4`) based on the following 17 probabilities:

[10] While either `float16` or `bfloat16` can be used for half precision, the latter seems clearly better suited since precision is not an issue (i.e., it is assumed to be handled in `float32`).



The offset is chosen as $\delta = \frac{1}{2}(\frac{1}{32} + \frac{1}{30})$ (i.e., average half length of 15-th and 16-th segment lengths). Then 8 evenly spaced values between $[\delta, \frac{1}{2}]$ (blue points) and 9 evenly spaced values between $[\frac{1}{2}, 1-\delta]$ (red points) are chosen. These probabilities are mapped to $a \in \mathbb{R}^8$ and $b \in \mathbb{R}^9$ through $F^{-1}_{\mathcal{N}(0,1)} : [0,1] \to \mathbb{R}$ where $a_1 = F^{-1}_{\mathcal{N}(0,1)}(\delta) = -b_9$ and $a_8 = b_1 = 0$. Discarding the duplicate zeros, we have $c = (a, b(2:)) \in \mathbb{R}^{16}$. The final values of `NF4`, $q^{\texttt{NF4}} \in \mathbb{R}^{16}$, are obtained as $q_i^{\texttt{NF4}} = \frac{c_i}{\max_j c_j}$. They are

$$\begin{aligned} q^{\texttt{NF4}} = \quad (\quad & -1, \quad -0.6962, \quad -0.5251, \quad -0.3949, \quad -0.2844, \quad -0.1848, \quad -0.0910, \quad 0, \\ & 0.0796, \quad 0.1609, \quad 0.2461, \quad 0.3379, \quad 0.4407, \quad 0.5626, \quad 0.7230, \quad 1 \quad) \end{aligned}$$

More generally, given B bits and an offset δ, NF considers an even partition of $[\delta, \frac{1}{2}]$ and $[\frac{1}{2}, 1-\delta]$ into 2^{B-1} and $2^{B-1}+1$ probabilities, which are then converted by $F^{-1}_{\mathcal{N}(0,1)}$ and normalized to final values in $[-1, 1]$. For instance, the 3-bit NormalFloat (`NF3`) values, using the same offset as `NF4` [6], are given by

$$q^{\texttt{NF3}} = \quad (\quad -1, \quad -0.4786, \quad -0.2171, \quad 0, \quad 0.1609, \quad 0.3379, \quad 0.5626, \quad 1 \quad)$$

NormalFloat is motivated by the finding that the weight of an LLM is empirically distributed as a Gaussian $w \sim \mathcal{N}(0, \omega^2)$. Thus if we scale $w' = \frac{1}{\omega} w$ we have $w' \sim \mathcal{N}(0,1)$ and `NF` can indeed bin the weights optimally. However, in practice we partition the model parameters as blocks of M values, and quantize each block $w \in \mathbb{R}^M$ into $\bar{w}$ by scale quantization (4), namely

$$\bar{w}_i = q_{\mathbf{nn}(i)} \qquad \mathbf{nn}(i) = \arg\min_{k=1\ldots 2^B} \left| q_k^{\texttt{NF}} - \frac{w_i}{\text{absmax}(w)} \right| \tag{29}$$

The dequantization from $\bar{w}$ is given by $\hat{w} = \text{absmax}(w)\bar{w}$. Because the scaling uses the absolute maximum of the block, not a constant, the distribution is not Gaussian and depends on the block size M. It is possible to use the correct quantiles [17].

F PTQ Examples

F.1 LLM.int8()

`LLM.int8()` is a dataless PTQ method focused on quantized matmul [2]. It does not require training; it simply loads a trained transformer-based model, quantizes the 32- or 16-bit weight $W \in \mathbb{R}^{d \times d'}$ of every linear layer to 8-bit integers $\overline{W} \in \mathbb{Z}^{d \times d'}$, then estimates the original linear operation XW where $X \in \mathbb{R}^{N \times d}$ is the input matrix. To estimate XW, it chooses to quantize X to $\bar{X} \in \mathbb{Z}^{d \times d'}$ and compute matmul in integer, rather than dequantizing $\overline{W}$ and computing matmul in float. To see how this is done, consider tensor-wise scaling which defines $\overline{W} = \text{round}(s_W^{-1} W)$ and $\bar{X} = \text{round}(s_X^{-1} X)$ for some $s_W, s_X > 0$. Then

$$XW \approx (s_X \bar{X})(s_W \overline{W}) = \underbrace{s_X s_W}_{\text{float}} \underbrace{\bar{X}\overline{W}}_{\text{integer matmul}} \tag{30}$$

Typically $\bar{X}\overline{W}$ is computed in a higher-bit integer format to avoid rounding errors (e.g., accumulate `int8` values in `int32`). While (30) can exploit integer arithmetic, it also incurs the overhead of quantizing X (in both inference speed and precision).

To improve precision, `LLM.int8()` proposes a form of group-wise scaling called "vector-wise" which scales each row of X and each column of W separately (i.e., treating matrix multiplication as Nd' dot products). Under vector-wise scaling, (30) becomes

$$XW \approx (\text{diag}\,(u_X)\,\bar{X})(\overline{W}\,\text{diag}\,(u_W)) = \underbrace{u_X u_W^\top}_{\text{float}} \underbrace{\bar{X}\overline{W}}_{\text{integer matmul}} \tag{31}$$

for some $u_X \in \mathbb{R}^N$ and $u_W \in \mathbb{R}^{d'}$. `LLM.int8()` is also one of the first works that report the "outlier" feature phenomenon in LLMs, namely that when language models become sufficiently large (starting around 6B parameters)

some feature dimensions (i.e., columns of X) have large magnitude dominating the behavior of the model. The outlier features are excluded from quantization as follows, using some threshold α (e.g., 6):

$$\begin{aligned} \mathcal{O} &= \{h = 1 \dots d : X_{i,h} > \alpha \text{ for some } i \in [N]\} \qquad & \mathcal{O}_\perp &= \{1 \dots d\} \setminus \mathcal{O} \\ XW &= X[:, \mathcal{O}]W[\mathcal{O}, :] + X[:, \mathcal{O}_\perp]W[\mathcal{O}_\perp, :] \end{aligned} \tag{32}$$

The first is computed in the original float format; only the second term is computed by (31). This means that we have to keep $W[\mathcal{O}, :] \in \mathbb{R}^{|\mathcal{O}| \times d}$ in full float, but outlier features remain rare (e.g., $|\mathcal{O}| \leq 7$ up to OPT-13B) so the decomposition is relatively cheap while significantly improving precision.

F.2 AWQ

AWQ is an output-calibrated PTQ method (11) that learns additional feature scales by a simple grid-search heuristic to minimize the output error [11]. Let $R(W) \approx W$ denote the approximate reconstruction of the linear weight $W \in \mathbb{R}^{d \times d'}$ after quantization and dequantization (AWQ uses group-wise scaling). Given an input $X \in \mathbb{R}^{T \times d}$, AWQ introduces additional scaling parameters $\beta \in \mathbb{R}^d$ learned by

$$\min_{\beta \in \mathbb{R}^d:\ \beta_h \geq 1\ \forall h} \left|\left| XW - X \operatorname{diag}(\beta)^{-1} R(\operatorname{diag}(\beta) W) \right|\right|^2 \tag{33}$$

The main idea is that there exists some β such that it does not affect the *quantization error* of $R(\operatorname{diag}(\beta) W)$ compared to $R(W)$. This holds empirically for two reasons. First, the average rounding error (1) tends to be always uniformly distributed between 0 and 1/2 regardless of the argument. Second, the quantization parameters (3) are only affected by extremal values in a quantization group and may remain unchanged, particularly when the rows of W are sparsely scaled and with clipping. But the error is now amplified by $X \operatorname{diag}(\beta)^{-1}$ instead of X, resulting in a β-fold reduction in relative error. The column scaling has the effect of eliminating the outlier features, which would otherwise have to be computed separately as in (32) for better precision.

Since R is not differentiable (though one can presumably consider straight-through estimation), AWQ crudely optimizes (33) by setting $\beta_h = \operatorname{absmax}(X(:, h))^\alpha$ where the optimal value $\alpha \in [0, 1]$ is selected over a grid size of 20. Once β is chosen, the downscaling operation $\operatorname{diag}(\beta)^{-1}$ can be absorbed into the weight of the previous layer and quantized offline.

F.3 SqueezeLLM

SqueezeLLM [8] finds a B-bit quantization $\widehat{W}$ of weight $W \in \mathbb{R}^{d \times d'}$ so that the training loss $L(\widehat{W})$ is minimized. Taking the vectorized views $\widehat{w}, w$, we seek to minimize

$$L(\widehat{w}) \approx L(w) + \nabla L(w)(\widehat{w} - w) + \frac{1}{2}(\widehat{w} - w)^\top \nabla^2 L(w)(\widehat{w} - w)$$

where $\nabla L(w) \approx 0$ in PTQ. Since the Hessian matrix is not typically computed in a standard deep learning framework, we estimate $\nabla^2 L(w) \approx \frac{1}{N} \sum_{i=1}^N (\nabla L_i(w))(\nabla L_i(w))^\top = \widehat{F}$ where $L_i(w)$ is the loss on the i-th of N samples. ($\widehat{F}$ is unfortunately known as the empirical Fisher matrix even though it is not a consistent estimator of the true Fisher information matrix, and it is often used for approximating the Hessian even though the relationship between Hessian and Fisher is only vaguely established under certain conditions [9].) Further using a diagonal approximation of $\widehat{F}$, we can write the problem as

$$\arg\min_{\widehat{W}} \left|\left| \widehat{F}^{1/2} \odot (W - \widehat{W}) \right|\right|_F^2$$

SqueezeLLM optimizes this over codebook levels (clustering) and outlier detection thresholds.

F.4 QLoRA

QLoRA is a PTQ-FT pipeline [3]. The model weights are quantized to `NF4` offline (with block-wise scaling) and the computation happens in `bfloat16`. More specifically, QLoRA computes for each linear layer [3]:

$$Y^{\texttt{bfloat16}} = X^{\texttt{bfloat16}} \text{Dequant}(\text{Dequant}(c_1^{\texttt{float32}}, c_2^{\texttt{float8}}), W^{\texttt{NF4}}) + X^{\texttt{bfloat16}} \underbrace{L_1^{\texttt{bfloat16}} L_2^{\texttt{bfloat16}}}_{\text{fine-tuned}} \tag{34}$$

A similar approach has been taken by GPTQ-LoRA which uses GPTQ (an output-calibrated PTQ method for low-bit integer quantization [5]) for the first term.

QLoRA proposes NormalFloat (Appendix E.1) and double quantization (Section 1.3.1). It also proposes paged optimizers that allocate paged memory for optimizer states which are automatically moved to CPU RAM when GPU runs out of memory (e.g., due to a long sequence length), then paged back to GPU memory when the memory is needed in the update step.

F.5 LQ-LoRA

LQ-LoRA performs QLoRA (34) with a better initialization [6]. Instead of quantizing $W \in \mathbb{R}^{d\times d'}$ to $\widehat{W}$ independently of the LoRA weights L_1, L_2, it proposes to use a LoRA-aware initialization such that $W \approx \widehat{W} + L_1L_2$. Under the sensitivity calibration loss (12), the joint optimization problem can be framed as

$$\min_{\widehat{W}\in\mathcal{Q}^{d\times d'},\ L_1\in\mathbb{R}^{d\times r},\ L_2\in\mathbb{R}^{r\times d'}} \left|\left|\widehat{F}(X,W)^{1/2}\odot(W-(\widehat{W}+L_1L_2))\right|\right|_F^2 \tag{35}$$

where $\mathcal{Q}^{d\times d'}$ is the space of all matrices that are losslessly quantizable to B-bit NF. (We may set $\widehat{F}(X,W) = 1_{d\times d'}$ if we have no calibration set X.) LQ-LoRA uses an alternating minimization algorithm to approximately minimize (35).

1. Holding $\widehat{W}$ fixed, the general weighted squared loss (35) is still (NP-)hard. Instead of doing a local search, LQ-LoRA assumes that $\widehat{F}(X,W)^{1/2} = uv^\top$ for some $u \in \mathbb{R}^d$ and $v \in \mathbb{R}^{d'}$. Then (35) becomes

$$\min_{L_1\in\mathbb{R}^{d\times r},\ L_2\in\mathbb{R}^{r\times d'}} ||\mathrm{diag}\,(u)\,(W-L_1L_2)\mathrm{diag}\,(v)-\mathrm{diag}\,(u)\,L_1L_2\mathrm{diag}\,(v)||_F^2 \tag{36}$$

   Since this is unconstrained, letting $K_1 = \mathrm{diag}\,(u)\,L_1$ and $K_2 = \mathrm{diag}\,(v)\,L_2^\top$, we can instead solve

$$\min_{K_1\in\mathbb{R}^{d\times r},\ K_2\in\mathbb{R}^{d'\times r}} \left|\left|\mathrm{diag}\,(u)\,(W-L_1L_2)\mathrm{diag}\,(v)-K_1K_2^\top\right|\right|_F^2 \tag{37}$$

   then recover $L_1 = \mathrm{diag}\,(u)^{-1}\,K_1$ and $L_2 = K_2^\top\mathrm{diag}\,(v)^{-1}$. A solution of (37) is given by $K_1 = U_r\Sigma_r^{1/2}$ and $K_2 = V_r\Sigma_r^{1/2}$ where $U_r\Sigma_rV_r$ is the rank-r SVD of $\mathrm{diag}\,(u)\,(W-L_1L_2)\mathrm{diag}\,(v)$. For the approximation step, LQ-LORA uses the row/column means of $\widehat{F}(X,W)$ as u, v (instead of the optimal rank-1 SVD).

2. Holding L_1, L_2 fixed, (35) becomes

$$\min_{\widehat{W}\in\mathcal{Q}^{d\times d'}} \left|\left|\widehat{F}(X,W)^{1/2}\odot\left((W-L_1L_2)-\widehat{W}\right)\right|\right|_F^2 \tag{38}$$

   This is approximately minimized by $\widehat{W} = D(Q(W-L_1L_2))$.

Additionally, LQ-LoRA optimizes the double quantization configuration (B, B_2, B_3, M_1, M_2) (B, B_2, B_3 are the target bitwidths, M_1, M_2 are the block sizes (9)) for each layer to minimize the quantization errors $\left|\left|W-(\widehat{W}+L_1L_2)\right|\right|_F^2$ while satisfying the bit budget. This can be done with an off-the-shelf integer linear programming solver.