Numerical Precision for Deep Learning

Karl Stratos

1 Floats

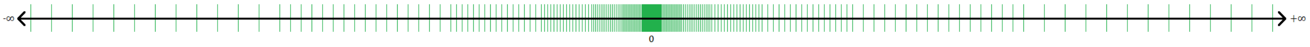
We approximate $\mathbb{R}$ with 2^B bitstrings $b \in \{0,1\}^B$. A "normal" **floating-point number**, in short **float**[1], assumes a partition of the bitstring $b = (b_1, z, f)$ where $z \in \{0,1\}^e \setminus \{0_e, 1_e\}$ encodes the exponent and a significand (or mantissa) $f \in \{0,1\}^p$ encodes the fractional part. Given a base (or radix) $\beta \in \{2, 10\}$, it computes

$$F_\beta(b) = (-1)^{b_1} \times \left(1 + f_1\beta^{-1} + \cdots + f_p\beta^{-p}\right) \times \beta^{E_\beta(z)} \qquad z \notin \{0_e, 1_e\} \tag{1}$$

where $E_\beta(z) \in \mathbb{Z}$ is a signed integer computed as

$$E_\beta(z) = \left(z_1\beta^{e-1} + \cdots + z_{e-1}\beta + z_e\right) - (\beta^{e-1} - 1) \qquad z \notin \{0_e, 1_e\} \tag{2}$$

The first term, for $z \notin \{0_e, 1_e\}$, ranges from 1 to $\frac{\beta^e - 1}{\beta - 1}$. Thus $E_\beta(z)$ ranges from $E_{\min} = 2 - \beta^{e-1}$ to $E_{\max} = \frac{\beta^{e-1} + \beta - 2}{\beta - 1}$. Thus (1) ranges from $N_{\min} = \beta^{E_{\min}}$ to $N_{\max} = \left(\frac{\beta^{p+1} - 1}{\beta^p(\beta - 1)}\right)\beta^{E_{\max}}$. Since (1) divides any $[\beta^i, \beta^{i+1}]$ into β^p uniformly spaced values, we have fewer floats away from zero (e.g., the next float after β^p is $\beta^p + \beta - 1$). Here is an illustration from Wikipedia:



Bitstrings with $z \in \{0_e, 1_e\}$ are used for special cases. Under the IEEE 754 standards, a complete float system is given by

$$F_\beta(b) = \begin{cases} (-1)^{b_1} \times \left(1 + f_1\beta^{-1} + \cdots + f_p\beta^{-p}\right) \times \beta^{E_\beta(z)} & \text{if } z \notin \{0_e, 1_e\} \ (\textbf{normal}) \\ (-1)^{b_1} \times \left(0 + f_1\beta^{-1} + \cdots + f_p\beta^{-p}\right) \times \beta^{E_{\min}} & \text{if } z = 0_e \text{ and } f \neq 0_p \ (\textbf{subnormal}) \\ (-1)^{b_1} \times 0 & \text{if } z = 0_e \text{ and } f = 0_p \ (\textbf{signed zeros}) \\ (-1)^{b_1} \times \infty & \text{if } z = 1_e \text{ and } f = 0_p \ (\textbf{signed infinities}) \\ \mathtt{NaN}(f) & \text{if } z = 1_e \text{ and } f \neq 0_p \ (\textbf{NaNs}) \end{cases} \tag{3}$$

The smallest subnormal magnitude is $S_{\min} = \beta^{E_{\min} - p}$ and the largest $S_{\max} = \left(\frac{\beta^{p+1} - 1}{\beta^p(\beta - 1)} - 1\right)\beta^{E_{\min}}$. 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. NaNs propagate: any operation involving a NaN generally outputs a NaN.

1.1 Rounding Errors

For simplicity, consider a positive real number $0 < x \leq N_{\max}$. Let $\hat{x}$ denote the float closest to x. Then $\hat{x} \in \{x_L, x_U\}$ where $x_L < x_U$ are consecutive floats. Since $x_U - x_L = \beta^{t-p}$ for some exponent $t \in [E_{\min}, E_{\max}]$, the absolute rounding error can be bounded as $|x - \hat{x}| \leq \frac{1}{2}\beta^{t-p}$. But this bound becomes loose for large t. A more popular measure is the **relative rounding error** $\left|\frac{x - \hat{x}}{x}\right|$ which can be bounded as

$$\left|\frac{x - \hat{x}}{x}\right| = \frac{|x - \hat{x}|}{x} \leq \frac{\beta^{t-p}}{2x} \leq \frac{\beta^{t-p}}{2x_L} \leq \frac{\beta^{t-p}}{2\beta^t} = \frac{1}{2}\beta^{-p} =: \epsilon_{\text{mach}} \tag{4}$$

The last term ϵ_{mach} is called **machine epsilon** representing the maximum error when rounding to 1.

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

1.1.1 Exact rounding

IEEE 754 mandates *exact rounding*. It means that the result of any float operation must be calculated exactly first, then rounded. Exact rounding can be achieved by extended precision. For instance, CPUs have a part dedicated to float operations (aka. float unit or FPU) that support extended precision. A popular format is the x86 extended precision format that uses $B = 80$ bits. Unlike CPUs, GPUs may not support a specific extended precision format, but they achieve IEEE 754 compliance through other means (e.g., performing intermediate calculations in higher precision).

2 Floats in Binary Base

While floats (3) can be defined using any integer base $\beta \geq 2$, the binary base $\beta = 2$ is the dominant choice for clear reasons like hardware efficiency, consistency with integer representation (which is binary), and better precision (e.g., (4) is minimized with $\beta = 2$). An exception is when precision with respect to a specific nonbinary base is paramount (e.g., $\beta = 10$ in finance). Using $\beta = 2$, we can simplify the constants as

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

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

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 B). 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 [8]. `E4M3` increases the range of `float8` by deviating from IEEE 754 (e.g., it has no infinities) [10]. 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 C).

2.1 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.

3 Quantization

Let $\mathcal{X}$ be a set of B'-bit floats in range $[X_{\min}, X_{\max}]$. Pick $B < B'$ and $\mathcal{Z}$ be a set of 2^B numbers in range $[Z_{\min}, Z_{\max}]$ representing a B-bit data type (i.e., all representable values). A **quantization** is a function $Q : \mathcal{X} \to \mathcal{Z}$. An associated **dequantization** is a function $D : \mathcal{Z} \to \mathcal{X}$ such that $x \approx D(Q(x))$ for all $x \in \mathcal{X}$.

If $\mathcal{Z}$ is another IEEE-compliant float format, quantization can be as simple as bit shifting (e.g., in mixed-precision training, Appendix A). For instance, to map `float32` to `bfloat16`, we can simply keep the exponent bits (since they both have $e = 8$) and truncate the significand to the right ("rounding toward zero"). For dequantization, we just pad the significand with zeros. More explicitly,

$$Q(x) = \text{Binary}(x)[: -16] \qquad\qquad D(z) = \text{cat}(\text{Decimal}(z), 0_{16})$$

In general, however, quantization scales and shifts the range of $\mathcal{X}$ to match the range of $\mathcal{Z}$ then finds nearest neighbors. Mathematically,

$$Q(x) = \text{nearest}_{\mathcal{Z}}\left(\frac{x}{s} + b\right) \tag{5}$$

$$D(z) = s(z - b) \tag{6}$$

$$s = \frac{X_{\max} - X_{\min}}{Z_{\max} - Z_{\min}} \qquad\qquad b = \text{nearest}_{\mathcal{Z}}\left(\frac{Z_{\min}X_{\max} - Z_{\max}X_{\min}}{X_{\max} - X_{\min}}\right) \tag{7}$$

where (6) is obtained by solving for x in (5) (ignoring the lossy operation); (7) is obtained by solving the linear system $X_{\min} = s(Z_{\min} - b)$ and $X_{\max} = s(Z_{\max} - b)$. The scale $s \in \mathbb{R}$ is represented as a B_1-bit float, where B_1 is the bit budget we specify for scales. The bias $b \in \mathcal{Z}$ is a B-bit number and called the "zero point" since $Q(0) = b$ and $D(b) = 0$. If we choose the ranges to be *symmetric*, namely $X_* = X_{\max} = -X_{\min}$ and $Z_* = Z_{\max} = -Z_{\min}$, then $b = 0$ and (5-7) simplify to **scale quantization**:

$$Q(x) = \text{nearest}_{\mathcal{Z}}\left(\frac{x}{s}\right) \qquad\qquad D(z) = sz \qquad\qquad s = \frac{X_*}{Z_*} \tag{8}$$

In addition to eliminating the bias term, (8) quantizes zero exactly (i.e., $Q(0) = D(0) = 0$), a useful property in deep learning. We can always take the absolute maximum $X_* = \text{absmax}(\mathcal{X})$ to achieve a symmetric input range. The target range depends on the data type: see the following table for examples.

$\mathcal{Z}$	$\min(\mathcal{Z})$	$\max(\mathcal{Z})$	scale quant	s	b
B-bit signed integers	-2^{B-1}	$2^{B-1} - 1$	yes	$\frac{\text{absmax}(\mathcal{X})}{2^{B-1}-1}$	0
B-bit unsigned integers	0	$2^B - 1$	no	$\frac{\max(\mathcal{X})-\min(\mathcal{X})}{2^B-1}$	$-\text{nearest}_{\texttt{uint}}(\frac{\min(\mathcal{X})}{s})$
B-bit NormalFloat (App. C.1)	-1	1	yes	$\text{absmax}(\mathcal{X})$	0

The B-bit signed integers have the asymmetric range $\mathcal{Z} = \left\{-2^{B-1} \ldots 2^{B-1} - 1\right\}$ under two's complement, so we choose $Z_* = 2^{B-1} - 1$ to have a symmetric target range, throwing away the lowest value. In the rest of the note, we will assume scale quantization for simplicity.

3.1 Precision-Memory Tradeoff

A single float s is used to scale all $x \in \mathcal{X}$. This implies a fundamental tradeoff between precision and memory in the nature of $\mathcal{X}$. Suppose $\mathcal{X}$ is the set of several billion floats representing the parameters of an LLM. If we quantize the whole $\mathcal{X}$, we only need to introduce one extra float but the precision will be poor. If we quantize each $x \in \mathcal{X}$ separately, we can achieve lossless quantization (e.g., use the scale $s_x = \frac{x}{z}$) but we introduce billions of extra floats, clearly defeating the purpose of quantization.

In practice, we define a partition $\mathcal{X} = \mathcal{X}_1 \cup \cdots \cup \mathcal{X}_n$ and quantize each $\mathcal{X}_i$ separately. Since $\mathcal{X}$ is typically a set of tensors T in deep learning, natural scaling options are

$$\mathcal{X}_i = \begin{cases} T & \text{(tensor-wise)} \\ T.\texttt{reshape}(-1, M)[j, :] & \text{(group-wise)} \\ T.\texttt{reshape}(-1)[k : k + M] & \text{(block-wise)} \end{cases} \tag{9}$$

where M is an integer that divides $|T|$ and j, k are some indices. Grouping always takes M groups from each tensor. Blocking keeps the block size constant $|\mathcal{X}_i| = M$, which is useful for measuring how much additional memory is allocated in quantization. Specifically, if we quantize the tensor T into a B-bit data type using group size M_1 and B_1-bit scales, the number of bits to store T is precisely

$$|T| \times \left(B + \frac{B_1}{M_1}\right) \tag{10}$$

where $\frac{B_1}{M_1}$ is the additional bits per parameter. To further reduce memory, we can quantize the scales again (aka. **double quantization**) [2]. If we quantize the scales into a B_2-bit data type (where $B_2 < B_1$) using group size M_2 and B_3-bit (meta-)scales, the number of bits to store T becomes

$$|T| \times \left(B + \frac{B_2}{M_1} + \frac{B_3}{M_1 M_2}\right) \tag{11}$$

3.2 Post-Training Quantization

Post-training quantization (PTQ) refers to quantizing the parameters of a trained model (under some quantization units (9)) to reduce the model size, enabling inference or finetuning with models too big to fit in available GPUs. Rather than quantizing all weights uniformly, we typically optimize the precision of *each layer*, most importantly the linear layer with a weight matrix $W \in \mathbb{R}^{d \times d'}$ (bias omitted). Let $R_\phi(W) = D_\phi(Q_\phi(W))$ denote the approximate reconstruction of W under a quantization parameter $\phi \in \Phi$. The PTQ optimization settings considered in the literature include:

$$\min_{\phi \in \Phi} \; ||W - R_\phi(W)||_F^2 \qquad \text{(dataless)} \tag{12}$$

$$\min_{\phi \in \Phi} \; ||XW - XR_\phi(W)||_F^2 \qquad \text{(output-calibrated)} \tag{13}$$

$$\min_{\phi \in \Phi} \; \left|\left|\widehat{F}(X, W)^{1/2} \odot (W - R_\phi(W))\right|\right|_F^2 \qquad \text{(sensitivity-calibrated)} \tag{14}$$

(12) just minimizes the reconstruction error. (13) minimizes the output error assuming an input $X \in \mathbb{R}^{N \times d}$ (aka. calibration set). (14) minimizes a weighted reconstruction error where $\widehat{F}_{j,k}(X, W) = \frac{1}{N}\sum_i (\frac{\partial L_i(W)}{\partial W_{j,k}})^2$ (Appendix D).

Once ϕ has been optimized (per layer), we quantize each W into $\overline{W}_\phi = Q_\phi(W)$ offline.[2] At inference time, we must compute $XD_\phi(\overline{W}_\phi)$ where $\overline{W}_\phi$ must be dequantized on the fly. To improve efficiency, existing methods write custom CUDA extensions [3] or precompile the operation [4].

PTQ can be combined with light-weight finetuning (**PTQ-FT**), popularly with the LoRA adapter [5]. The idea is that the performance loss due to quantization can be recovered by learning a small set of additional weights. We can approach this as a pipeline (i.e., do PTQ, then do LoRA while holding the quantized weights fixed [2]) or jointly optimize quantization and LoRA [4]. The latter corresponds to switching $R_\phi(W)$ with $R_\phi(W) + L_1 L_2$ in (12–14) where $L_1 \in \mathbb{R}^{d \times r}$ and $L_2 \in \mathbb{R}^{r \times d'}$ are the LoRA weights then optimizing ϕ, L_1, L_2 together.

Most PTQ methods can be seen as some combination of the above settings. See Appendix E for a discussion of specific methods.

References

[1] 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*.

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

[3] 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*.

[4] 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*.

[2] In practice, this is more complicated because the quantization data type is often not natively supported in the programming language. Thus $\overline{W}_\phi$, typically in a low-bit int (if `NF`, we store the bin numbers, e.g., like this), is first converted to a supported format (e.g., `uint8`) which is further packed into bytes for storage efficiency.

[5] 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*.

[6] 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*.

[7] 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**.

[8] 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.

[9] 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*.

[10] 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*.

[11] 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*.

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

A Mixed-Precision Training

Mixed-precision training performs only precision-sensitive operations in `float32` and the rest in `bfloat16`.[3] The conversion is achieved by simply truncating and padding the significand. It is typical to also 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. $L \leftarrow$ ComputeLossAMP($B$) # Autocast based on operation types (e.g., 16 bits for matmul, 32 for log).
  2. ($2^t \times L$).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 [11]. With such a small number of bits, much more care is needed in scaling (e.g., per-tensor instead of global scaling).

B 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 $g_\theta \leftarrow \textbf{Grad}(L, \theta)$ 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:

$$L'(\theta) := \lim_{\epsilon \to 0^+} \frac{L(\theta + \epsilon) - L(\theta)}{\epsilon} \approx \frac{L(\theta + \hat{\epsilon}) - L(\theta)}{\hat{\epsilon}} =: \hat{g}^{\text{one}}_{\theta, \hat{\epsilon}} \tag{15}$$

[3] 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`).

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

$$\hat{g}^{\text{two}}_{\theta,\hat{\epsilon}} := \frac{L(\theta + \hat{\epsilon}) - L(\theta - \hat{\epsilon})}{2\hat{\epsilon}} \tag{16}$$

The symmetry of the approximation will yield an improvement. Since

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

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 (16) is a much more accurate estimate of $L'(\theta)$ than (15) for a small $\hat{\epsilon}$. To account for different scales, the gradient check typically tests the relative error

$$\frac{\left|g_\theta - \hat{g}^{\text{two}}_{\theta,\hat{\epsilon}}\right|}{\max(|g_\theta|, |\hat{g}^{\text{two}}_{\theta,\hat{\epsilon}}|)} \leq \tau \tag{17}$$

where $\tau > 0$ is some tolerance. If $g_\theta = L'(\theta)$ (i.e., the algorithm is implemented correctly), then (17) is $O(\hat{\epsilon}^2)$ up to scaling. For instance, if $\hat{\epsilon} = 10^{-5}$ then (17) 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 (16) (or (15)) 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.

C 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,

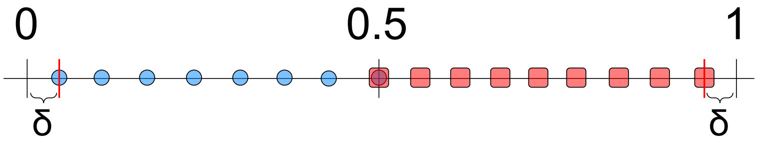
$$t_k = F^{-1}_{\mathbf{pop}}\left(\frac{k}{K}\right)$$

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:

$$q_k = \frac{F^{-1}_{\mathbf{pop}}\left(\frac{k}{K+1}\right) + F^{-1}_{\mathbf{pop}}\left(\frac{k+1}{K+1}\right)}{2} \qquad \forall k = 1 \dots K \tag{18}$$

C.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 **pop** $= \mathcal{N}(0,1)$ [2]. They propose a quantization scheme called 4-bit NormalFloat (`NF4`) based on the following 17 probabilities:



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`, $q^{\texttt{NF4}} \in \mathbb{R}^{18}$, are obtained as $q_i^{\texttt{NF4}} = \frac{c_i}{\max_j c_j}$. They are

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

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` [4], are given by

$$q^{\texttt{NF3}} = \begin{array}{cccccccccc} (& -1, & -0.4786, & -0.2171, & 0, & 0.1609, & 0.3379, & 0.5626, & 1 &) \end{array}$$

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 (8), namely

$$\bar{w}_i = q_{\mathbf{nn}(i)} \qquad\qquad \mathbf{nn}(i) = \arg\min_{k=1\ldots 2^B} \left| q_k^{\texttt{NF}} - \frac{w_i}{\text{absmax}(w)} \right| \tag{19}$$

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 [12].

D Sensitivity-Based Quantization

Kim *et al.* [6] consider the task of finding a B-bit quantization $\widehat{W}$ of weight $W \in \mathbb{R}^{d \times d'}$ (i.e., each float $W_{i,j}$ is clustered to one of the 2^B bins) 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 [7].) Further using a diagonal approximation of $\widehat{F}$, we can write the problem as

$$\arg\min_{\widehat{W}} \left|\left| \widehat{F}^{1/2} \odot (W - \widehat{W}) \right|\right|_F^2 \tag{20}$$

E PTQ Examples

E.1 LLM.int8()

`LLM.int8()` is a dataless PTQ method focused on quantized matrix multiplication (`matmul`) [1]. 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 (9) which defines $\overline{W} = \mathrm{round}(s_W^{-1} W)$ and $\bar{X} = \mathrm{round}(s_X^{-1} X)$ for some $s_W, s_X > 0$. Then

$$XW \approx (s_X \bar{X})(s_W \overline{W}) = \underbrace{s_X s_W}_{\text{float}} \underbrace{\bar{X}\overline{W}}_{\text{integer } \texttt{matmul}} \tag{21}$$

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 (21) can exploit integer arithmetic, it also incurs the overhead of quantizing X (in both inference speed and precision).

To improve precision, `LLM.int8()` proposes "vector-wise" scaling 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, (21) becomes

$$XW \approx (\mathrm{diag}\,(u_X)\,\bar{X})(\overline{W}\mathrm{diag}\,(u_W)) = \underbrace{u_X u_W^\top}_{\text{float}} \underbrace{\bar{X}\overline{W}}_{\text{integer } \texttt{matmul}} \tag{22}$$

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):

$$\begin{aligned} \mathcal{O} &= \{h = 1 \ldots d : X_{i,h} > \alpha \text{ for some } i \in [N]\} \qquad\qquad \mathcal{O}_\perp = \{1 \ldots d\} \setminus \mathcal{O} \\ XW &= X[:, \mathcal{O}]W[\mathcal{O}, :] + X[:, \mathcal{O}_\perp]W[\mathcal{O}_\perp, :] \end{aligned} \tag{23}$$

The first is computed in the original float format; only the second term is computed by (22). 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.

E.2 AWQ

AWQ is an output-calibrated PTQ method (13) that learns additional feature scales by a simple grid-search heuristic to minimize the output error [9]. Let $R(W) \approx W$ denote the approximate reconstruction of the linear weight $W \in \mathbb{R}^{d \times d'}$ after quantization and dequantization (under some quantization units (9), AWQ uses grouping). Given an input $X \in \mathbb{R}^{T \times d}$, AWQ introduces additional scaling parameters $\beta \in \mathbb{R}^d$ learned by

$$\min_{\beta \in \mathbb{R}^d:\ \beta_h \geq 1\ \forall h} \left|\left| XW - X\mathrm{diag}\,(\beta)^{-1}\, R(\mathrm{diag}\,(\beta)\, W) \right|\right|^2 \tag{24}$$

The main idea is that there exists some β such that it does not affect the *quantization error* of $R(\mathrm{diag}\,(\beta)\, W)$ compared to $R(W)$. This holds empirically for two reasons. First, the average rounding error (5) tends to be always uniformly distributed between 0 and 1/2 regardless of the argument. Second, the quantization parameters (7) 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 $X\mathrm{diag}\,(\beta)^{-1}$ 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 (23) for better precision.

Since R is not differentiable (though one can presumably consider straight-through estimation), AWQ crudely optimizes (24) by setting $\beta_h = \mathrm{absmax}(X(:, h))^\alpha$ where the optimal value $\alpha \in [0, 1]$ is selected over a grid size of 20. Once β is chosen, the downscaling operation $\mathrm{diag}\,(\beta)^{-1}$ can be absorbed into the weight of the previous layer and quantized offline.

E.3 QLoRA

QLoRA is a PTQ-FT pipeline [2]. 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 [2]:

$$Y^{\texttt{bfloat16}} = X^{\texttt{bfloat16}}\mathrm{Dequant}(\mathrm{Dequant}(c_1^{\texttt{float32}}, c_2^{\texttt{float8}}), W^{\texttt{NF4}}) + X^{\texttt{bfloat16}}\underbrace{L_1^{\texttt{bfloat16}} L_2^{\texttt{bfloat16}}}_{\text{finetuned}} \tag{25}$$

A similar approach has been taken by GPTQ-LoRA which uses GPTQ (an output-calibrated PTQ method for low-bit integer quantization [3]) for the first term.

QLoRA proposes NormalFloat (Appendix C.1) and double quantization (Section 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.

E.4 LQ-LoRA

LQ-LoRA performs QLoRA (25) with a better initialization [4]. 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 $W \approx \widehat{W} + L_1 L_2$. Under the sensitivity calibration loss (14), the joint optimization problem can be framed as

$$\min_{\widehat{W}\in\mathcal{Q}^{d\times d'},\ L_1\in\mathbb{R}^{d\times r},\ L_2\in\mathbb{R}^{r\times d'}} \left|\left|\widehat{F}(X,W)^{1/2}\odot(W-(\widehat{W}+L_1L_2)\right|\right|_F^2 \tag{26}$$

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 (26).

1. Holding $\widehat{W}$ fixed, the general weighted squared loss (26) 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 (26) becomes

   $$\min_{L_1\in\mathbb{R}^{d\times r},\ L_2\in\mathbb{R}^{r\times d'}} ||\mathrm{diag}\,(u)\,(W-L_1L_2)\mathrm{diag}\,(v)-\mathrm{diag}\,(u)\,L_1L_2\mathrm{diag}\,(v)||_F^2 \tag{27}$$

   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

   $$\min_{K_1\in\mathbb{R}^{d\times r},\ K_2\in\mathbb{R}^{d'\times r}} \left|\left|\mathrm{diag}\,(u)\,(W-L_1L_2)\mathrm{diag}\,(v)-K_1K_2^\top\right|\right|_F^2 \tag{28}$$

   then recover $L_1 = \mathrm{diag}\,(u)^{-1}\,K_1$ and $L_2 = K_2^\top\mathrm{diag}\,(v)^{-1}$. A solution of (28) 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, (26) becomes

   $$\min_{\widehat{W}\in\mathcal{Q}^{d\times d'}} \left|\left|\widehat{F}(X,W)^{1/2}\odot\Big((W-L_1L_2)-\widehat{W}\Big)\right|\right|_F^2 \tag{29}$$

   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 (11)) 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.