CSE 291A/DSC 291: Data Systems for Machine Learning, Spring 2026

9: Quantization

Lecturer: Hao Zhang
Scribes: Wei-Ting Chen, Chih-Yun Lin, ChaoYuan Lin, Yuxin Xiong, Hongrui Zhang, Yinuo Zhou, Xuanbin Peng, Emma Zhang, Sifan Li

1 Motivation: Reducing Memory by Reducing Element Size

This lecture continues the discussion of memory optimization. The previous discussion focused on techniques that manage memory through scheduling: the computation can be arranged so that peak memory usage does not exceed the available memory. This lecture shifts to a different question: instead of only changing the schedule, can the system reduce the amount of memory needed to store each tensor value?

The main observation is that memory usage depends on the number of stored elements and the size of each element. Model weights, activations, and optimizer states are all stored as collections of elements. Their memory usage is therefore related to: memory usage $\approx$ number of elements $\times$ sizeof(element).

This suggests a direct way to reduce memory usage: reduce `sizeof(element)`. For example, if a value is stored using 8 bits instead of 32 bits, the storage needed for that value is reduced by a factor of four, ignoring implementation overhead.

This idea leads to quantization. From the memory perspective, quantization reduces the number of bits used to represent each value. The goal is to reduce memory usage while still keeping enough numerical information for the model to work well.

1.1 Definition of Quantization

Quantization is the process of constraining an input from a continuous, or otherwise large, set of values to a discrete set. Equivalently, a value that could originally take many possible values is mapped to one of a smaller number of representable values.

This process introduces quantization error. The quantization error is the difference between the original value and the value after quantization. For example, if a continuous signal is mapped to discrete levels, the quantized signal no longer exactly matches the original signal.

The lecture used an image example to explain why quantization can still work in machine learning. An image can be represented with fewer colors or lower precision and still be recognizable. The image may become less detailed, but a person can still identify that it is a cat. This illustrates an important intuition: machine learning models can often tolerate a small amount of numerical error. Therefore, it may be possible to use lower-precision values without losing too much task performance.

1.2 Quantization in Machine Learning

In machine learning, the high-level idea of quantization is to use a lower-precision representation for data while almost preserving model performance. For example, in language modeling, the goal is to preserve the language modeling capability. In image classification, the goal is to preserve classification accuracy.

Quantization provides several benefits.

First, it reduces memory usage. If values that previously used four bytes or two bytes are represented using one byte or half a byte, then the memory required to store those values decreases.

Second, it can accelerate computation. Modern hardware often provides higher arithmetic throughput for

lower-precision operations. If computation can be performed at lower precision, the hardware can execute more operations per second, which can make the model faster.

Third, it can reduce energy usage. The lecture noted that chips are designed so that computation with fewer bits can save energy. This matters especially for power-constrained settings, such as running machine learning models on mobile devices.

Thus, in this part of the lecture, quantization is introduced as a technique that connects memory usage, computation speed, and energy efficiency.

1.3 Digital Representation of Data

Before discussing quantization in more detail, the lecture reviews how numbers are represented in memory. Computers represent data using bits, and each bit can be either 0 or 1. Different numerical formats assign different meanings to these bits.

The lecture first discusses integer representations, including unsigned integers, signed integers, and two's complement. It then introduces fixed-point numbers as a simple way to represent fractional values.

Unsigned Integer Representation. An unsigned integer represents only nonnegative values. For an n-bit unsigned integer, each bit position is assigned a power of two. If the bits are written as $b_{n-1}b_{n-2}\cdots b_1b_0$, then the represented value is $\sum_{i=0}^{n-1} b_i \times 2^i$. The range of an n-bit unsigned integer is $[0, 2^n - 1]$.

For example, the 8-bit unsigned integer 00110001 represents $0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 49$.

This representation is simple, but it cannot represent negative numbers.

Sign-Magnitude Representation. To represent negative numbers, one simple method is sign-magnitude representation. In this representation, the most significant bit is used as the sign bit, and the remaining $n-1$ bits represent the magnitude.

A sign bit of 0 represents a nonnegative number. A sign bit of 1 represents a negative number. The remaining bits are interpreted as the absolute value.

With n bits, the standard range of sign-magnitude representation is $[-(2^{n-1} - 1), 2^{n-1} - 1]$.

For example, with 8 bits, there is one sign bit and seven magnitude bits. The largest magnitude is $2^7 - 1 = 127$, so the range is $[-127, 127]$.

The main problem with sign-magnitude representation is that it has two representations for zero:

$000\ldots000$ represents $+0$ and $100\ldots000$ represents -0.

This wastes one bit pattern because both patterns represent the same numerical value.

Two's Complement Representation. Modern computers usually use two's complement representation for signed integers. In two's complement, the most significant bit is not only a sign indicator. Instead, it contributes a negative value, while the remaining bits contribute nonnegative powers of two.

For an n-bit two's complement integer, $b_{n-1}b_{n-2}\cdots b_1b_0$, the represented value is $-b_{n-1} \times 2^{n-1} + \sum_{i=0}^{n-2} b_i \times 2^i$.

The standard range of an n-bit two's complement integer is $[-2^{n-1}, 2^{n-1} - 1]$.

For example, an 8-bit two's complement integer has range $[-128, 127]$.

Two's complement avoids the duplicate-zero problem. There is only one representation of zero: $000\ldots000$.

The bit pattern that would represent negative zero in sign-magnitude representation is instead used to represent the most negative value. This is why two's complement can represent one more negative value than positive value.

Fixed-Point Numbers. Integer representations are not enough for machine learning because model parameters and activations are often fractional values. The lecture therefore introduces fixed-point numbers

as a straightforward way to represent fractions.

In fixed-point representation, the position of the binary point is fixed in advance. Some bits represent the integer part, and some bits represent the fractional part. Once the binary point position is chosen, the bit pattern can be interpreted as a scaled value.

For example, consider the bit pattern 00110001.

If this bit pattern is interpreted as an integer, it represents 49. If it is interpreted as a fixed-point number with four fractional bits, then it is scaled by 2^{-4}: $49 \times 2^{-4} = 49 \times 0.0625 = 3.0625$.

The same value can also be written by assigning powers of two around the fixed binary point: $0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 3.0625$.

This example shows that the same bit pattern can represent different numerical values depending on the representation rule. Fixed-point representation provides a simple way to represent fractional values using a fixed interpretation of the bits.

2 Floating-point Representation

Fixed-point number suffers from a limited range because the position of the binary point is fixed. To address this issue, modern computers adopt floating-point representation, where the binary point can "float" according to the exponent.

In FP32, a 32-bit number is divided into three parts: a 1-bit sign, an 8-bit exponent, and a 23-bit fraction field. For normal numbers (i.e., when the exponent is not all zeros), the value is defined as $(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent}-127}$.

In this formula, $(1 + \text{fraction})$ is the significand. Note that the lecture slides use the term *significant*, but the standard term is *significand*.

There are two design ideas:

- The significand always has a leading 1 in binary scientific notation (e.g., $1.xxx$), so this leading 1 is not explicitly stored. Instead, it is implicitly assumed to provide one extra bit of precision.
- The exponent controls the scale of the number. When the exponent increases, the value becomes larger; when it decreases, the value becomes smaller. Intuitively, this is like moving the binary point to the right or left. This is why the representation is called "floating-point."

However, under the normal representation, zero cannot be represented. This is because the significand is always of the form $1 + \text{fraction}$, which is never equal to zero. To solve this problem, the IEEE standard introduces a special representation called *subnormal numbers*.

3 Subnormal numbers

In the IEEE standard, we put all zeros on the exponent part to represent subnormal number. When we try to represent the subnomal number, we use a different equation to represent floating points: $(-1)^{\text{sign}} \times \text{fraction} \times 2^{-126}$.

Compared to normal numbers, two changes are made:

- The implicit leading 1 is removed. The significand becomes just the fraction.
- The exponent is fixed at -126 (instead of using the biased exponent).

This representation allows numbers closer to zero to be expressed. In particular, zero can be represented by setting both the exponent and fraction fields to all zeros.

By using the equation above, our calculation will be $1 \times 0 \times 2^{-126} = 0$, or $(-1) \times 0 \times 2^{-126} = 0$.

4 Infinity and NaN

There are a few special cases in the numerical representation of the floating point and they are ∞, $-\infty$ and NaN. In order to represent ∞, we set all the exponent bit to 1 and all the fraction bit to 0. When we want to represent NaN(Not a Number), we set the exponent bit to all 1 and the fractoin bit is anything other than 0.

5 Summary of FP32

An FP32 number has 32 bits, divided into one sign bit, eight exponent bits, and 23 fraction bits. The sign bit determines whether the value is positive or negative, while the exponent and fraction bits determine the magnitude.

The interpretation of an FP32 bit pattern depends on the exponent field. There are three main cases.

First, when the exponent bits are all zeros, the number is either zero or subnormal. If the fraction field is also zero, the value represents signed zero, either $+0$ or -0 depending on the sign bit. If the fraction field is nonzero, the value is a subnormal number: $(-1)^{\text{sign}} \times \text{fraction} \times 2^{1-127}$. Unlike normal numbers, subnormal numbers do not have an implicit leading one in the significand. They are used to represent values very close to zero. In FP32, the smallest positive subnormal value is 2^{-149}.

Second, when the exponent field is between 1 and 254, the number is a normal floating-point value. In this case, FP32 uses an implicit leading one, so the value is $(-1)^{\text{sign}} \times (1 + \text{fraction}) \times 2^{\text{exponent}-127}$. The bias is 127 because FP32 has eight exponent bits. This biased exponent lets the format represent both very small and very large values.

Third, when the exponent bits are all ones, the value is a special value. If the fraction field is zero, the value is either $+\infty$ or $-\infty$. If the fraction field is nonzero, the value is NaN, which stands for "Not a Number." NaN is used for undefined or invalid numerical results.

Overall, FP32 covers zero, subnormal numbers, normal numbers, infinities, and NaN. The exponent controls the scale of the number, while the fraction controls the precision within that scale.

6 Comparing FP32, FP16, and BF16

FP32, IEEE FP16, and BF16 differ in how they divide bits between the exponent and fraction fields. FP32 uses one sign bit, eight exponent bits, and 23 fraction bits. FP16 uses one sign bit, five exponent bits, and ten fraction bits. BF16, short for Brain Floating Point, uses one sign bit, eight exponent bits, and seven fraction bits.

Format	Exponent bits	Fraction bits	Total bits
FP32	8	23	32
FP16	5	10	16
BF16	8	7	16

The key tradeoff is: exponent width controls range, while fraction width controls precision. A larger exponent field allows the format to represent a wider range of magnitudes. A larger fraction field allows the format to represent values more precisely within that range.

FP16 has fewer exponent bits than FP32, so its representable range is much smaller. However, among 16-bit formats, FP16 keeps more fraction bits, which gives it better precision than BF16 when the value is within range.

BF16 takes a different design choice. It keeps the same eight exponent bits as FP32, so it has a similar dynamic range to FP32. However, it only has seven fraction bits, so it has lower precision. This makes BF16 useful in machine learning because neural networks often need a wide numerical range, while they can tolerate some loss of precision.

7 Exercise: Decoding an FP16 Number

Consider the following FP16 bit pattern: 1 | 10001 | 1100000000. The first bit is the sign bit, the next five bits are the exponent, and the last ten bits are the fraction.

The sign bit is 1, so the number is negative: $(-1)^1 = -1$.

For FP16, the exponent has five bits, so the bias is $2^{5-1} - 1 = 15$. The stored exponent is $10001_2 = 17_{10}$. Therefore, the unbiased exponent is $17 - 15 = 2$.

The fraction bits are 1100000000_2. As a binary fraction, this equals $0.1100000000_2 = 2^{-1} + 2^{-2} = 0.5 + 0.25 = 0.75$. Since this is a normal FP16 number, we include the implicit leading one. Thus the significand is $1 + 0.75 = 1.75$.

Putting the sign, significand, and exponent together, the decoded value is $(-1)^1 \times (1+0.75) \times 2^2 = -1.75 \times 4 = -7$. Therefore, the FP16 bit pattern represents -7_{10}.

8 IEEE Normal Numbers and Examples

A normal IEEE binary float encodes $(-1)^s \times (1.f) \times 2^{E-\text{bias}}$, with implicit leading one, biased exponent E, and special cases when the exponent bits are all zeros or all ones. The bias for FP16 uses five exponent bits, so bias $= 2^{5-1} - 1 = 15$.

8.1 Decoding an FP16 pattern

Split the FP16 word into sign, five exponent bits, and ten fraction bits: $\underbrace{1}_{\text{sign}} \; \underbrace{10001}_{E} \; \underbrace{1100000000}_{f}$. The exponent field is $10001_2 = 17$, so the power of two is $2^{17-15} = 2^2$.

The fraction bits are the binary tail after the implicit one. Here $.1100000000_2 = \frac{1}{2} + \frac{1}{4} = 0.75$, so the full significand is $1 + 0.75 = 1.75$. (Some writeups record only the tail as 0.75; the stored value uses the full $1.f$.)

The decoded value is $(-1) \times (1 + 0.75) \times 2^2 = -7$. Equivalently, multiply 1.1100000000_2 by 2^2 by shifting the binary point two places right to obtain $111.0_2 = 7_{10}$, then apply the negative sign.

8.2 Encoding decimal 2.5 in BF16

Write the value in scientific form, $2.5 = 1.25 \times 2^1$. The sign is positive, so the sign bit is 0. BF16 uses the FP32 bias bias $= 2^7 - 1 = 127$. The true exponent is 1, so the stored exponent satisfies $E - 127 = 1$, hence $E = 128_{10} = 10000000_2$.

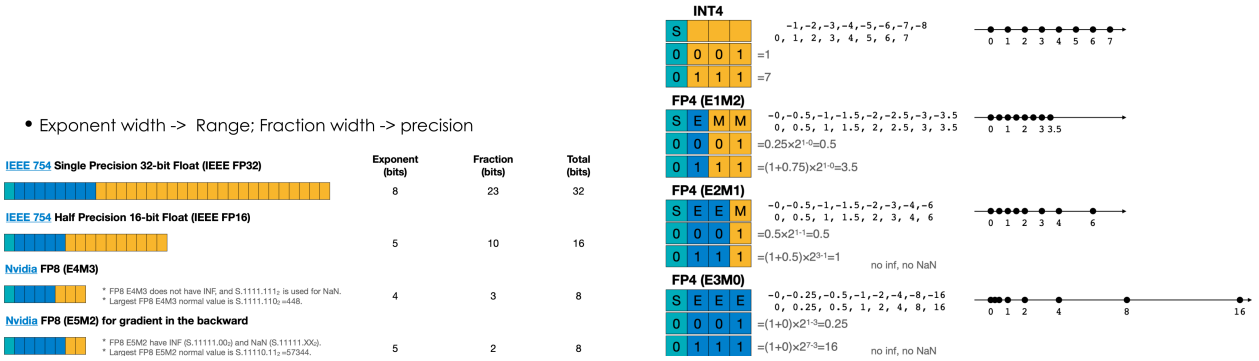
In binary, $2.5_{10} = 10.1_2 = 1.01_2 \times 2^1$ after normalizing the binary point once. Thus $1.25_{10} = 1.01_2$, the tail after the implicit one is $.01_2 = \frac{1}{4} = 0.25$, and the seven fraction bits are 0100000_2. The full word is `0 | 10000000 | 0100000`.

9 FP8, INT4, and Narrow Floats

For even narrower types, the same range-versus-precision trade-off applies. FP8 formats used in hardware include NVIDIA E4M3 (four exponent, three fraction bits) and E5M2 (five exponent, two fraction bits). INT4 and ultra-low-bit floating encodings also appear in research stacks: integer grids are evenly spaced for a fixed scale, while low-bit floats devote bits to exponents so representable values are *not* uniformly spaced.

10 Quantization: Overview

Quantization replaces many distinct stored values by a smaller discrete set so tensors need fewer bits while models remain accurate enough. Figure 2 gives a compact illustration. A complete treatment spans *post-training quantization*, *quantization-aware training*, and *mixed precision*; those topics are developed in later



(a) Narrow floating-point layouts.

(b) FP8-style encodings and related low-bit designs.

Figure 1: Narrow floating-point and FP8-style low-bit formats.

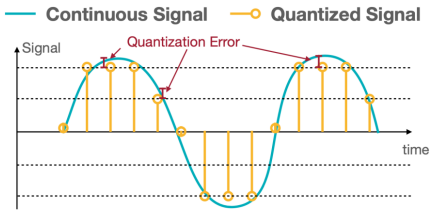


Figure 2: Quantization maps many values to a small discrete set.

notes.

K-means (vector) quantization clusters tensor entries (often weights), keeps a small floating-point *codebook* of centroids, and stores only short *indices* per entry. The idea is to pay once for the table and then ship compact codes through memory; runtime typically looks up a float from the codebook before arithmetic. The left two panels of Figure 3 give a compact k-means style example. Clustering objectives, bit budgets, backward behavior, and accuracy–compression trade-offs are left to a dedicated section later.

Linear (affine) quantization stores integer codes together with a real *scale* and integer *zero point* so each code maps affinely back to the original numeric range. This layout matches integer matrix-multiply hardware and underpins common INT8 paths. The right panel of Figure 3 illustrates the linear mapping; deriving parameters, folding quantization into matmul, and measuring speedups are deferred to later material.

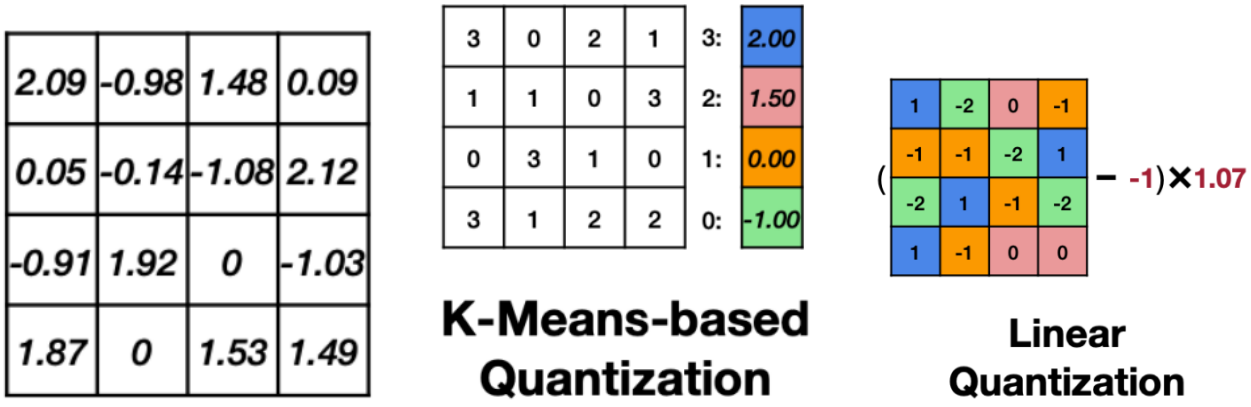


Figure 3: K-means example (left two panels); linear affine quantization (right panel).

11 Evolution of Floating-Point Representations in ML Community

Overtime, in order to reduce compute and storage cost of ML model training and inferencing, the ML professionals attempted to reduce the number of bits in model parameters. The bit reduction process started with FP32, and then reduced to FP16 to avoid represent large numbers.

3 years ago, the ML community shifted towards FP8 representations, where the number of bits becomes very limited, and the trade-off between precision (mantissa) and range (exponent) is an essential consideration. For the two standards of FP8, including E4M3 (4 bits for exponents, 3 bits for mantissa) and E5M2 (5 bits for exponents, 2 bits for mantissa), E4M3 can represent narrower range of numbers but is better at precision, due to 1 extra bit for mantissa and 1 less bit for exponent. However, with 3 mantissa bits in E4M3, the precision loss is already very bad.

Nowadays, Nvidia tries to sell chips in FP4, with almost no difference with integer. The impact of FP4 chip is that it demonstrates reasonable performance for matmul operations. However, the effort to perform attention computations with FP4 is still in progress. In addition, for model application, it is possible for a model to approach the quality of ChatGPT by freezing the weights and quantizing them on FP4 for inference.

12 Quantization: Motivation and Basics

The intuition of quantization is to constrain the continuous values to discrete set, while maintaining the distribution of values. As shown, the basic representation has floating point weights for storage, and floating point arithmetics for compute. The motivation of K-means quantization and linear quantization is to reduce at least one of weights or arithmetics, in context of training or inferencing, to discrete representation and therefore improve efficiency.

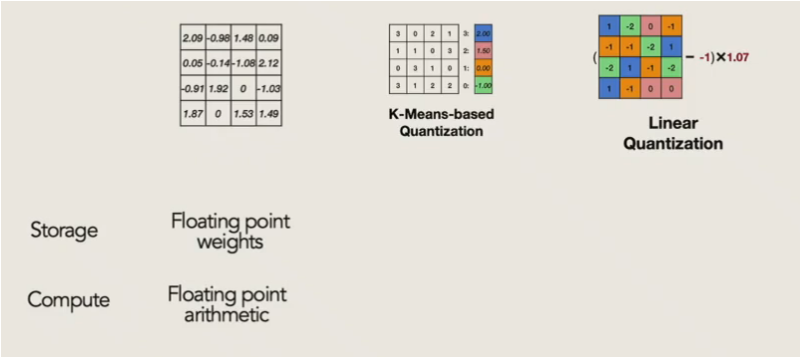


Figure 4: Quantization Basics

13 K-means Quantization: Introduction and Clustering

The core idea of K-means quantization is to reduce a set of floating point number weights to integer representation by performing clustering over the weight matrix to target a K-means number, a value that costs limited bit representation demand and is around the center of the weight value distribution. This approach considers the tradeoff between efficient value representation from floating-point to integer and the tolerance of quantization error (the differentiation between the K-mean number and the floating-point real weight). Hence, a direction of optimization is to pick the K-mean number that minimizes the quantization error.

The motivation of performing the clustering is to reduce the storage cost of weight matrix, from floating-point representation to integer representation with a smaller floating-point codebook. The process of clustering conversion is to perform a clustering operation to confirm a set of K-mean number i.e. centroids (that prefers fewer bits to represent) that individual weight can map to (one of the centroids) with minimized quantization errors.

In implementation, after the centroids are located, the original weight matrix is converted to a matrix of (integer) indices, where each index map to a centroid value closest to original weight, and the mapping is stored in a smaller codebook(hashmap format). This conversion effective reduced storage as the matrix of cluster index shares the same shape as weight matrix while replaced element type with integer of much smaller size (in example, from 32bit float to 2bit int), and the codebook has much smaller size (in example, from 16 entries of weight matrix to 4 entries of codebook) despite requiring limited floating-point representation. Hence, from the example, we observe that the original storage (64 bytes) is reduced to 16+4=20 bytes. Again, the core tradeoff now concerns the comparison between the benefit of smaller storage and the reduced model accuracy due to quantization error.

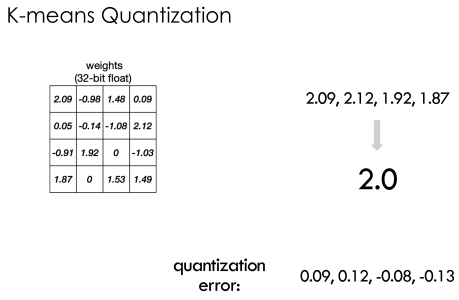


Figure 5: Used by Xuanbin and Yinuo; K-means: Introduction, replacing similar weights by a single representative value.

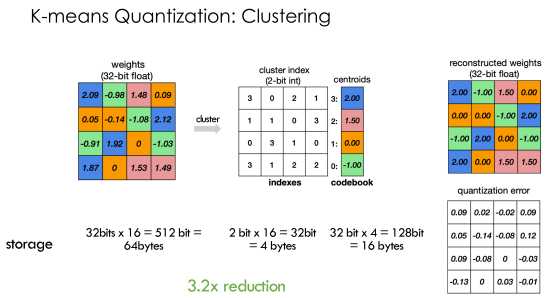


Figure 6: Used by Xuanbin and Yinuo; Clustering example, clustering with $K = 4$ centroids: indices, codebook, reconstruction, and per-element error.

14 K-means Quantization

Building on Yinuo's introduction above, this section follows the K-means quantization slides through the quantization error example, the general storage accounting, the backward pass, the weight-distribution evidence, the empirical bit budget, and the runtime picture. The running 4×4 FP32 example shown earlier (Figures 5–6) is reused throughout.

14.1 Idea and Quantization Error

Consider again the 4×4 FP32 weight matrix in Figure 5. The four entries $\{2.09, 2.12, 1.92, 1.87\}$ are all close to 2, so storing the single representative value 2.0 in their place introduces only small per-entry residuals $\{2.09 - 2.0,\ 2.12 - 2.0,\ 1.92 - 2.0,\ 1.87 - 2.0\} = \{0.09,\ 0.12,\ -0.08,\ -0.13\}$, which match the quantization-error numbers shown on the slide. These residuals are the *quantization errors*: K-means picks the representative values to keep them small, using the standard one-dimensional k-means clustering on the tensor entries.

14.2 Clustering, Codebook, and Storage

Running k-means with $K = 2^N$ clusters yields the two artifacts already shown in Figure 6: a *codebook* of 2^N FP32 centroids, and a *cluster-index matrix* of the same shape as the original tensor whose entries are N-bit integers pointing into the codebook. The reconstructed weight at any position is the centroid looked up by that position's index. For the 4×4, $N = 2$ instance on the slide, the storage breakdown is 32 bits $\times$ 16 = 512 bits = 64 bytes for the original weights, 2 bits $\times$ 16 = 32 bits = 4 bytes for the indices, and 32 bits $\times$ 4 = 128 bits = 16 bytes for the codebook — a 3.2$\times$ reduction.

The general accounting on the next slide writes the same expressions as a function of tensor size M and bit budget N, under the assumption $M \gg 2^N$: original $= 32M$ bits, quantized $= NM + 32 \cdot 2^N$ bits $= NM + 2^{N+5}$ bits. Since the codebook term 2^{N+5} is small relative to NM, the compression ratio simplifies to $\frac{32M}{NM} = \frac{32}{N}$. So $N = 4$ gives roughly 8$\times$ compression and $N = 2$ gives roughly 16$\times$.

14.3 Backward Pass: Fine-Tuning the Codebook

Plain nearest-centroid rounding loses accuracy, but the codebook is tiny and can be fine-tuned to absorb most of that error. Standard backpropagation produces one gradient per FP32 weight; we then *group* those gradients by cluster index and *reduce* (sum) within each cluster to obtain a single gradient per centroid, which is then multiplied by the learning rate and subtracted from the centroid to produce a fine-tuned codebook (Figure 7).

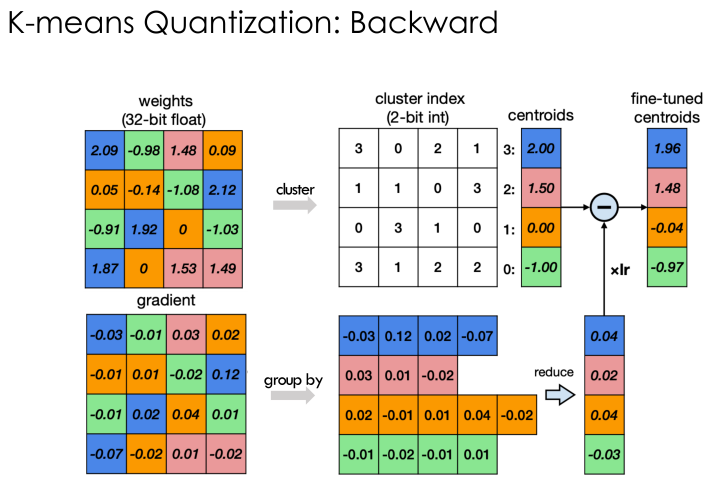
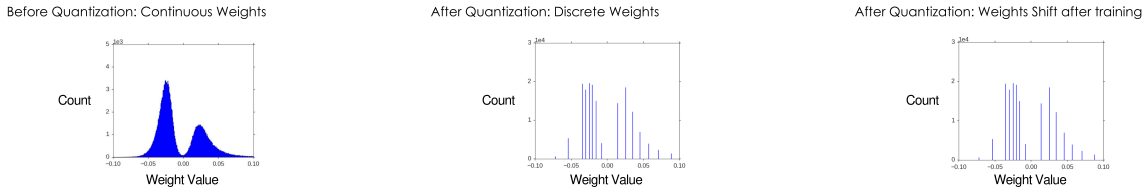


Figure 7: Backward pass: gradients are grouped by cluster index and reduced to one per centroid.

Formally, for each cluster k the centroid is updated by $c_k \leftarrow c_k - \eta \sum_{i\,:\,\text{cluster}(w_i)=k} \frac{\partial \mathcal{L}}{\partial w_i}$. Only the 2^N FP32 centroids are trained; the cluster assignments are integers and stay fixed.

14.4 Why It Works: Weight Distributions

The reason this aggressive compression works on neural network weights is visible in their empirical distribution (Figure 8). Trained weights form a smooth, roughly bimodal distribution concentrated near zero, so a handful of representative values already cover most of the mass. After clustering, all entries collapse onto the centroid locations; after fine-tuning, those centroids shift slightly along the axis to better match the loss surface.

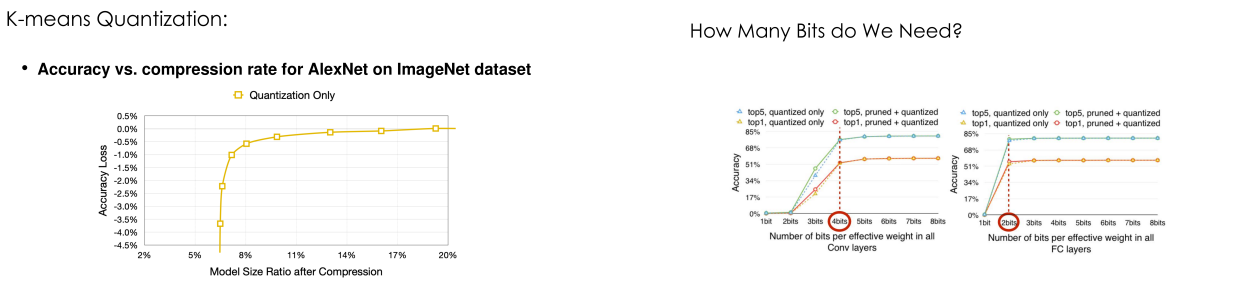


(a) Before quantization. (b) After quantization. (c) After fine-tuning.

Figure 8: Weight histograms: continuous distribution → discrete centroids → shifted after fine-tuning.

14.5 How Many Bits Do We Need?

The slides answer this empirically on AlexNet/ImageNet (Figure 9). The accuracy-vs-compression curve on the left stays essentially flat (within a fraction of a percent) until the compressed model size drops to roughly 8% of the original, and falls off sharply below that. The per-layer-type chart on the right separates the two layer families and circles the smallest bit-width that still preserves accuracy: about 4 bits per weight for convolutional layers, and about 2 bits per weight for fully connected layers.



(a) Accuracy loss vs. compressed size. (b) Conv ≈ 4 bits, FC ≈ 2 bits.

Figure 9: Bit budget for K-means quantization on AlexNet/ImageNet.

14.6 Runtime: Decode-Then-Compute

The runtime slide states the same point in three bullets: weights are decompressed using a lookup table (the codebook) at inference, *storage* is quantized, and *compute* is still floating-point arithmetic. As Figure 10 shows, each layer's cluster-index tensor is decoded back to FP32 via a codebook lookup just before the matmul, and the matmul itself uses standard floating-point arithmetic. K-means quantization therefore reduces what is held in storage, but does not change the arithmetic format at compute time.

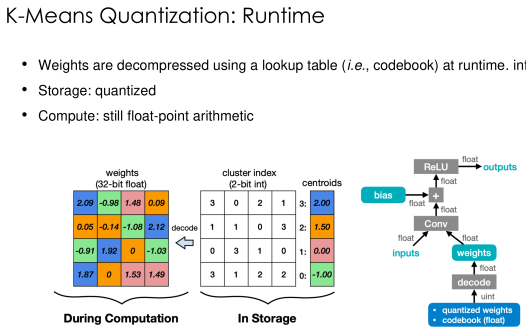


Figure 10: Inference: indices and codebook are stored, decoded to FP32, then used in a standard matmul.

15 Linear Quantization

Linear quantization is a strict linear mapping between a low-precision integer representation and a high-precision real-valued representation. In our running example, the original weights are 32-bit floats and we encode them as 2-bit signed integers; in practice, N-bit signed integers (typically $N \in \{4, 8\}$) are used and the analysis is identical.

The defining equation is $r = S(q - Z)$,

where r is the original real-valued weight in the same format as the unquantized model (FP32 here), q is the quantized integer weight that is actually stored, Z is the zero point, an integer in the same representation as q — and S is the scale, a floating-point scalar in the same representation as r. The fact that Z is itself an integer is what allows the real number $r = 0$ to be represented exactly by the integer code $q = Z$.

Read in this direction, the equation describes *de-quantization*: given a stored integer q, subtract the integer zero point Z to centre it, then multiply by the floating-point scale S to recover an approximation $\hat{r}$ of the original weight. In a deployed quantized model, the file stores only the integer weights q together with the two scalars S and Z for each quantized tensor or tensor group; if you open the `config.json` of any quantized

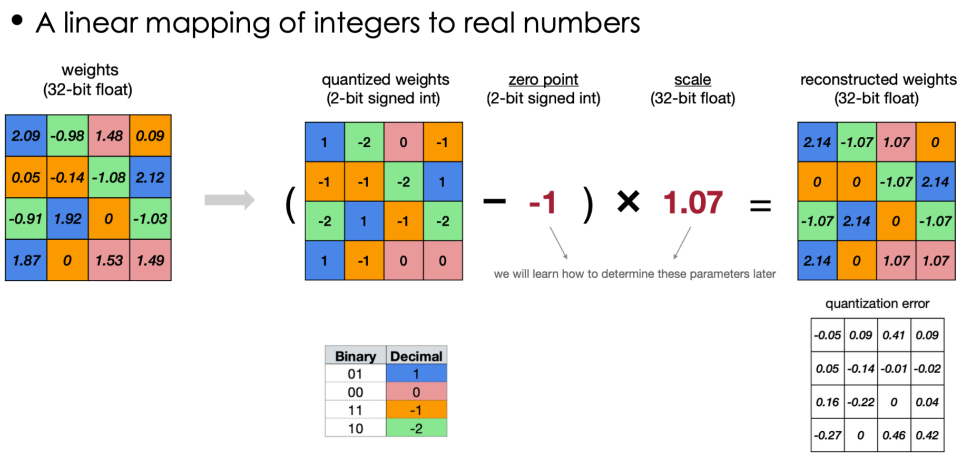


Figure 11: Linear quantization on a 4×4 FP32 weight matrix. The original weights (left) are encoded by a 2-bit signed integer matrix together with a single zero point $Z = -1$ and a single scale $S = 1.07$. Multiplying $(q - Z)$ by S recovers the reconstructed FP32 matrix on the right; element-wise differences with the original form the quantization error matrix.

large language model on Hugging Face DeepSeek, LLaMA, and so on you will find these scale and zero-point fields recorded per tensor.

16 Determining the Scale and Zero Point

The central design problem of linear quantization is to choose S and Z so that quantization error is small. The simplest reasonable choice, often called *min–max* calibration, is to require the mapping to send the extreme integer codes onto the extreme observed weights.

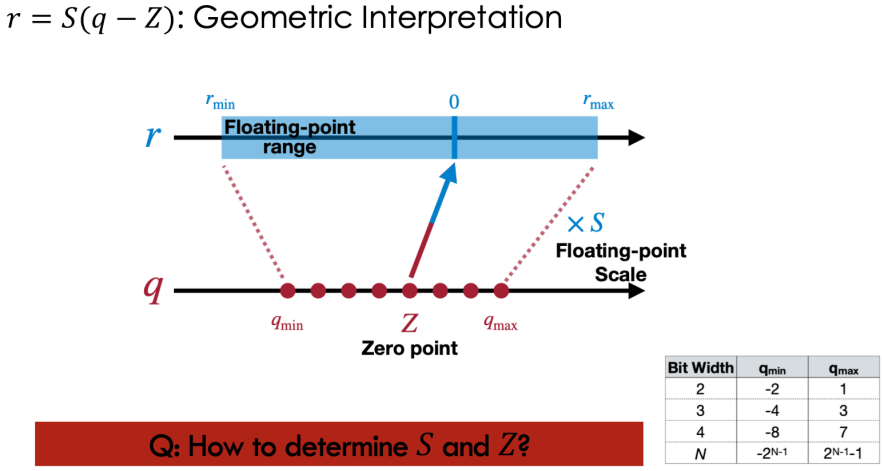


Figure 12: Geometric interpretation of $r = S(q - Z)$. The integer grid is first translated so that the integer zero point Z aligns with the real value 0, then scaled by the floating-point factor S so that $[q_{\min}, q_{\max}]$ covers $[r_{\min}, r_{\max}]$. The bit-width table gives the integer extremes for N-bit signed representations.

Geometrically, the mapping does two things to the integer line. First, it translates the line by $-Z$, so that the integer code $q = Z$ lands on the real value 0. Second, it stretches (or shrinks) the translated grid by the floating-point factor S, so that the integer extremes $q_{\min}$ and $q_{\max}$ land on the floating-point extremes $r_{\min}$ and $r_{\max}$. The integer range is fixed by the bit width N of the chosen signed-integer representation: $q_{\min} = -2^{N-1}$ and $q_{\max} = 2^{N-1} - 1$. For our running 2-bit example this gives $q_{\min} = -2$ and $q_{\max} = 1$.

The min–max constraint gives one equation at each endpoint: $r_{\max} = S(q_{\max} - Z), \qquad r_{\min} = S(q_{\min} - Z).$

These are two equations in the two unknowns S and Z. The integer extremes are fixed by the bit width; the

floating-point extremes $r_{\min}$ and $r_{\max}$ are read directly off the weight tensor we wish to quantize. Subtracting the second equation from the first eliminates Z and gives a closed-form expression for the scale: $S = \frac{r_{\max} - r_{\min}}{q_{\max} - q_{\min}}$.

Applying this to the 4×4 weight matrix in Figure 11 gives $r_{\max} = 2.12$ and $r_{\min} = -1.08$. With $q_{\max} = 1$ and $q_{\min} = -2$: $S = \frac{2.12 - (-1.08)}{1 - (-2)} = \frac{3.20}{3} \approx 1.07$,

which matches the scale shown on the slide. The zero point Z is obtained by substituting this scale back into either endpoint equation and rounding to the nearest integer; that derivation is taken up in the next part of the lecture.

Once the scale S has been determined, we can solve for the zero point Z by substituting S back into one of the endpoint equations. Using the lower endpoint constraint, $r_{\min} = S(q_{\min} - Z)$,

we can isolate Z as follows: $q_{\min} - Z = \frac{r_{\min}}{S}$,

and therefore $Z = q_{\min} - \frac{r_{\min}}{S}$.

Since the zero point must be an integer code, we round the result to the nearest integer: $Z = \text{round}\left(q_{\min} - \frac{r_{\min}}{S}\right)$.

This formula has a simple geometric meaning. The quantity tells us how many integer-scale steps the lower floating-point endpoint $r_{\min}$ is away from real zero. Subtracting this quantity from $q_{\min}$ tells us which integer code should align with the real value 0. Applying this formula to the same 4×4 weight matrix, we have $r_{\min} = -1.08, \qquad q_{\min} = -2, \qquad S \approx 1.07.$

Therefore, $Z = \text{round}\left(-2 - \frac{-1.08}{1.07}\right)$.

Since $\frac{-1.08}{1.07} \approx -1.01$,

we obtain $Z = \text{round}(-2 + 1.01) = \text{round}(-0.99) = -1$.