CSE-234: Data Systems for Machine Learning, Winter 2025

9: Quantization

Lecturer: Hao Zhang

Scribe: Davit Abrahamyan, Junda Chen, Andrew Choi, Kayla Hom, Jai Malegaonkar, Sairam Mahadeva Ganapathy, Parsa Mirtaheri, Anikait Sunil Nair, Nived Neetha Sooraj, Sharanya Ranka

1 Multiple Choice Questions & Recap

1.1 Understanding Memory Consumption in Training

(**Answer: D**) This is a multiple-choice question to identify the main sources of memory consumption during model training. Training code consumes minimal memory compared to other elements: intermediate activation values, model weights, and optimizer states.

1.2 Gradient Checkpointing

(**Answer: B**) Gradient checkpointing is a technique for reducing memory usage by selectively storing activations and recomputing them during the backward pass. The false statement was *"Gradient checkpointing applies to both model weights and activations."* Gradient checkpointing primarily saves memory for **activations** rather than weights. The effectiveness of checkpointing depends on the placement of checkpoints, which influences the trade-off between computation and memory savings. Other statements are all true.

1.3 Activation Size Calculation for GPT-3 2.7B

(**Answer: C**) Given a GPT-3 2.7B model with fp16 precision and gradient checkpointing at the transformer layer boundary, activation size is computed as:

- **Activation memory per layer** = (Batch Size × Sequence Length) × Hidden Dimension = Number of Tokens × d_{model} = Activation per Layer
- **Total activation memory** = Activation per Layer × Number of Layers.
- **Final conversion** to GB.

Using values from the provided table:

- **Batch Size** = 1M tokens
- **Hidden Dimension** = 2560
- **Layers** = 32
- **Precision (fp16)** = 2 bytes per value Therefore, the correct answer after the calculation is **152.59GB**.

2 How to Choose / Tune Memory Optimization

A structured approach to navigating out-of-memory (OOM) issues in training large models is as follows:

1. **Start with a desirable batch size** – Hyperparameter tuning helps determine an optimal batch size that balances accuracy and training efficiency.
2. **Check for OOM errors** – If training with the chosen batch size fits within memory, no further optimization is needed.
3. **Apply gradient accumulation first** – If OOM occurs, reduce the micro-batch size while maintaining global batch size via accumulation.
4. **Use gradient checkpointing** – If accumulation alone is insufficient, enable checkpointing to trade computation for memory.
5. **Resort to swapping** – As a last option, swap data to CPU memory, though this significantly slows training.
6. **Parallelization** – If none of the above strategies work, distributing computation across multiple GPUs is necessary.

These strategies ensure efficient training of large models while balancing memory and computational efficiency.

3 Floating-Point Representation

For floating point numbers, the intuition is behind using a few bits to represent the exponent, 1 bit for the sign, and the remaining bits for the fraction. By varying how many bits should be designated to the exponent and the fractional parts will determine how much precision the number contains as well as the range of possible numbers. For example, the standard FP32 numbers contain 8 bit in the exponent part and 23 bits in the fraction part.

The formula for converting from binary to decimal for fp32 is the following: $(-1)^{sign} * (1 + Fraction) * 2^{Exponent-127}$. Here, the 127 represents the exponent bias ($2^{8-1} - 1$, where 8 represents the number of bits in the exponent part).

However, there is a clear problem with this representation, as we cannot represent the number 0.

3.1 Normal vs Subnormal Numbers

In order to represent the number 0, we will assume the following: if all the bits in the exponent part are equal to 0, then, we will change the formula that represents floating points. In particular, the formula becomes $(-1)^{sign} * (Fraction) * 2^{1-127}$. Thus, this allows us to represent the number 0 by having all 0's for the fraction part. When all of the exponent bits are 0, we call these numbers subnormal numbers, otherwise, they are called normal. Thus, for any floating point representation we have two ranges, one for subnormal numbers and one for the normal numbers.

3.2 Minimum Positive Value

Now, let us try to understand what is the minimum possible value that we can represent using the representation defined above, which will give us an idea of how precise we can be through this representation. For normal range, the minimum that we can get is through setting all fractional bits to 0 and all exponent bits to 0, except the last one (as otherwise it would become subnormal). Thus, the minimum number in the normal range would be $(1+0)*2^{1-127}=2^{-126}$. For the subnormal range, we have to set all exponent bits to 0, and for the fractional part, the minimum non-zero value would be setting all bits, except the last one, equal to 0. Thus, the smallest possible value in the subnormal range would be $2^{-23}*2^{-126}=2^{-149}$. Thus, subnormal range can represent even smaller values than the normal range.

3.3 Special Floating-Point Values

There are certain special values that we have to address. Firstly, in the case of normal numbers, if we set the sign bit equal to 0, all exponent bits to 1 and all fraction bits to 0, we call this positive infinity ($+\infty$). If we set the sign bit to 1, we would get negative infinity ($-\infty$). Another special case is NaN (Not a number), which is represented by setting all exponent bits to 1, while not using the sign bit. It results in a wasted memory space as we do not care about the fractional bits as well.

3.4 FP32 Summary

So summarize, we have two ranges to represent floating point values in fp32, the normal range and subnormal range. When the exponent is 0 (subnormal), we use the following formula to convert from binary to decimal: $(-1)^{sign}*(Fraction)*2^{1-127}$. In the case of normal range, we use the following formula: $(-1)^{sign}*(1+Fraction)*2^{Exponent-127}$. As discussed above, there are special values, in particular positive/negative infinity and NaN, which differ in their representation from all other numbers.

If we put the floating point numbers on a number line, we can see that they differ from fixed-point numbers in a way that smaller values can be represented using more precision (the subnormal values).

3.5 FP32 vs. FP16 vs. BF16

The 3 main floating-point representations used nowadays are fp32, fp16, and bf16 (brain float). Compared to fp32 representation, fp16 uses 5 bits for the exponent part and 10 bits for the fractional part, while bf16 uses 8 bits for the exponent part and 7 bits for the fraction part. This indicates that fp16 can represent smaller numbers with more precision than bf16, however, bf16 can represent larger values due to allocating more exponent bits.

Due to the fact that bf16 has a greater dynamic range (can represent larger numbers), it is used more widely in neural network training. The reason is that during training the gradients can explode and the initial iterations, and having more bits in the exponent will allow us to represent those gradients instead of having them explode to infinity. In contrast, fp16 is used more during inference, as we need more precision.

3.5.1 Floating-Point Exercises

Exercise 1: Represent 1 10001 1100000000 in decimal (1 sign bit, 5 exponent bits and 10 fraction bits).

The sign bit is 1, thus, the number is negative. The bias of the number is $2^(5-1)-1=15$ (5 represents the number of exponent bits). Consequently, the $(Exponent - 15)$ part will be 2. The fractional part would be $1/2+1/4=0.75$ (as only the first two bits are 1). Thus, the final number would be $-(1+0.75)*2^2=-7.1$. During the exam, we can expect to get a similar problem but in the subnormal range.

Exercise 2: Represent 2.5 decimal value in BF16.

We can represent 2.5 as $1.25*2^1$. From this, we can deduce that the sign bit is 1. The exponent bias is $2^7-1=127$. Thus, we have x-127=1, meaning that our x (or the exponent part in decimal) is 128, which is 10000000 in binary. As we allocate 7 bits for the fractional part in the case of BF16 and the fractional part is equal to 0.25, we can represent it as 0100000 in binary. Thus, the final number in BF16 would be 0 10000000 0100000.

3.6 FP8: The Latest Standard for AI Acceleration

FP8 is being adopted because it reduces computational costs while maintaining performance. The primary advantage of using FP8 over higher precision formats like FP16 and FP32 is its ability to store numbers more efficiently while still offering the flexibility needed for deep learning. Lower precision reduces memory requirements, speeds up matrix operations, and allows more AI models to fit into hardware with limited memory. However, using FP8 requires careful handling to balance the trade-off between numerical precision and dynamic range.

Floating-point formats always consist of an exponent and a mantissa. The exponent controls the range, while the mantissa determines precision. Traditional IEEE floating-point formats like FP32 use an 8-bit exponent and a 23-bit mantissa, offering high precision but at a significant computational cost. FP16, a half-precision format, reduces the exponent to 5 bits and the mantissa to 10 bits, which is often sufficient for many AI models. FP8 takes this further by splitting into two different configurations: E4M3, which uses 4 bits for the exponent and 3 for the mantissa, and E5M2, which uses 5 bits for the exponent and 2 for the mantissa.

Format	Exponent (bits)	Mantissa (bits)	Max Value	Infinity (INF)	Use Case
FP8 (E4M3)	4	3	448	No	Inference (more precision)
FP8 (E5M2)	5	2	57,344	Yes	Training (wider range)

Table 1: Comparison of FP8 formats.

The two formats serve different purposes. FP8 (E5M2) is used for training because it has a larger dynamic range, meaning it can handle larger values and avoid gradient overflow during backpropagation. This is crucial because, during training, numbers can become extremely large due to accumulated gradients, and a higher exponent range helps mitigate issues. FP8 (E4M3) is used for inference because it provides better precision, which ensures that predictions are more accurate when running trained models. Since inference typically does not require handling very large numbers, a smaller exponent range with a larger mantissa helps maintain numerical stability.

NVIDIA has played a key role in promoting FP8 adoption, as lower precision directly impacts AI hardware efficiency. The company has integrated FP8 support into its latest GPUs, such as the H100, which is optimized for deep learning workloads. As AI models continue to grow, the trend of using low-precision formats like FP8 will become even more critical, enabling faster and more power-efficient training and inference.

3.7 INT4 and FP4: Lower Precision for Greater Efficiency

The motivation behind these formats is to further minimize memory usage and increase computation speed, particularly for small-scale AI models running on edge devices, such as smartphones.

3.7.1 INT4: Simple 4-bit Integer Representation

INT4, or 4-bit integer, is a straightforward representation that uses 2's complement encoding to store signed values. This means it can represent values between -8 and 7, with each step between values being uniform. Unlike floating-point formats, INT4 has no exponent and simply represents fixed-interval values. This makes it computationally very efficient since integer arithmetic is much faster than floating-point arithmetic.

The primary benefit of INT4 is that it requires minimal memory and computational power, making it ideal for AI inference on mobile devices. Many AI models deployed on devices like iPhones already use INT4 for efficiency. However, the lack of fine granularity makes it unsuitable for AI training, where small changes in weights are crucial for optimization. Instead, floating-point formats like FP8 and FP4 are preferred for training, as they provide more flexibility in representing numbers.

3.7.2 FP4: Different Configurations for Flexibility

FP4, or 4-bit floating-point, introduces a compromise between range and precision by incorporating an exponent field. Since FP4 only has 4 bits total, the distribution of these bits between the exponent and mantissa is crucial in determining how numbers are represented. There are three main configurations of FP4: E1M2, E2M1, and E3M0, each of which provides a different trade-off between range and precision.

Format	Exponent (bits)	Mantissa (bits)	Characteristics
FP4 (E1M2)	1	2	Behaves similarly to INT4
FP4 (E2M1)	2	1	Larger dynamic range than INT4
FP4 (E3M0)	3	0	Represents only powers of two

Table 2: Comparison of FP4 formats.

The first configuration, FP4 (E1M2), allocates 1 bit to the exponent and 2 bits to the mantissa. However, this format is not very useful because it behaves almost identically to INT4, offering little benefit over integer representation. In fact, in most cases, E1M2 is not used at all, as INT4 is a better choice for fixed-point arithmetic.

The second configuration, FP4 (E2M1), gives 2 bits to the exponent and 1 bit to the mantissa. This provides a larger dynamic range than INT4, making it somewhat useful for AI workloads. Since the exponent has more bits, it allows representation of larger numbers while still keeping memory usage low.

The third and most interesting configuration is FP4 (E3M0), which uses all 3 bits for the exponent and has no mantissa. This means that all the values it can represent are powers of two, forming a logarithmic scale. The advantage of this format is that it efficiently represents a wide range of values with minimal precision loss. The lecture highlights that E3M0 is the most promising format for FP4, as it allows significant reductions in numerical precision while maintaining a broad range of values.

3.7.3 Future of FP4 in AI Computing

As of today, FP4 is not widely used in AI training, but research is being conducted to explore its potential. NVIDIA is actively looking into FP4 as a possible format for future GPUs, as lower precision could further optimize AI workloads. However, at this stage, no major deep learning models have been successfully trained using FP4, meaning it is still an area of active research rather than a deployed standard.

4 K-Means Quantization

4.1 Quantization Basics

Quantization is the process of constraining an input from a continuous or otherwise large set of values to a discrete set. There are two classic quantization techniques: K-Means Quantization, which is the most widely adopted method, and Linear Quantization, which is considered the most powerful method. Quantization reduces storage requirements by converting floating-point weights into low-precision representations. Additionally, it improves computational efficiency, since lower precision allows for higher floating-point operations per second (FLOPs).

4.2 Clustering

In K-Means quantization, we use K-Means clustering to determine a set of centroids that represent the weight values. Each weight is then mapped to its nearest centroid in a codebook, minimizing quantization error—the difference between the original weight and its mapped value.

Consider applying N-bit quantization to M weights of 32-bit floating-point type. Before quantization, storing these weights requires $32M$ bits. After quantization, the storage requirement consists of MN bits to store the mapped weights and $32 \times 2^N = 2^{N+5}$ bits to store the floating point centroids.

For example, with $M = 16$ and $N = 2$:

$$\text{Original Storage: } 32 \times 16 = 512 \text{ bits} = 64 \text{ bytes}$$

$$\text{Quantized Storage: } (2 \times 16 = 32 \text{ bits} = 4 \text{ bytes}) + (32 \times 4 = 128 \text{ bits} = 16 \text{ bytes}) = 20 \text{ bytes}$$

This results in a 3.2× reduction in storage. More generally, assuming $M \gg 2^N$, the compression ratio is approximately:

$$\frac{32M}{NM} = \frac{32}{N}.$$

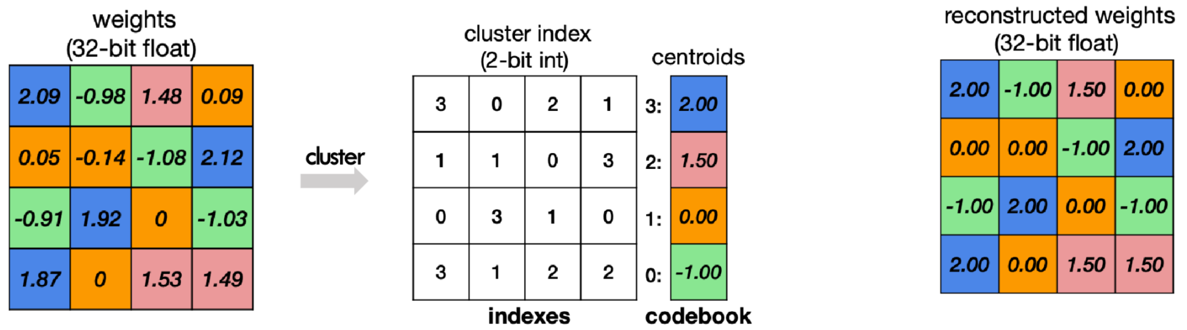


Figure 1: K-Means Quantization: Original Weights, Cluster Indices, and Reconstructed Weights.

4.3 Backward Pass

During training, the weights are updated by gradient descent. To handle updates in the backward pass for quantized weights, the gradients of weights assigned to the same centroid (i.e., in the same cluster) are aggregated and reduced to a single representative gradient. This aggregated gradient is then multiplied by the learning rate and used to update the centroid's value in the codebook.

In this way, all weights mapped to the same centroid are updated indirectly by modifying their shared centroid, ensuring that the quantization remains consistent throughout training.

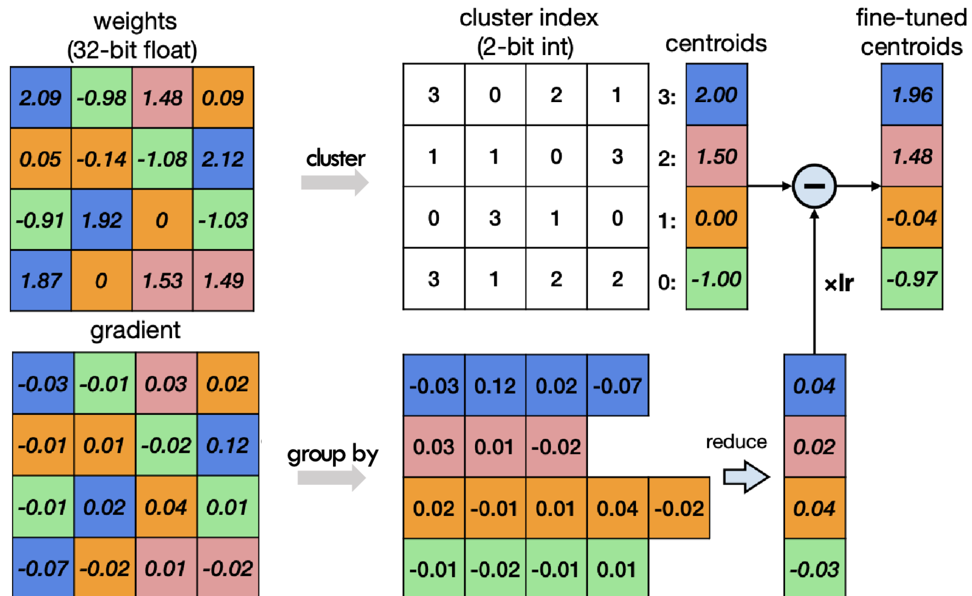


Figure 2: Backward Pass: Gradient Aggregation and Centroid Update.

4.4 Weight Distribution Effects

We can analyze how quantization affects the weights in a model. In the example shown, the original (unquantized) CNN model, we have weights distributed in a bimodal fashion, with one mode having negative values, while the other having positive values. The weights are distributed continuously between $-0.1 \rightarrow 0.1$. KMeans Quantization decreases the number of unique weights we can represent, but with the benefit of (highly) reduced storage space.

4.5 Finetuning after Quantization

The model can be further finetuned to retain some accuracy that might have been lost due to quantization. As in the previous slides, the weights in a cluster (represented by 1 value) are updated with the same gradient to maintain the number of unique parameters. This can result in a slight shift in weights (before vs after finetuning training).

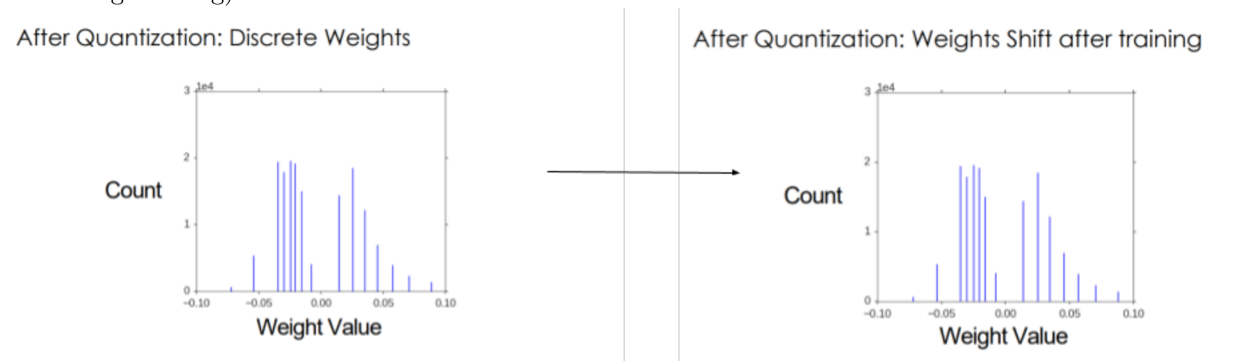


Figure 3: Before and After Finetuning

4.6 Hyperparameter: Number of Clusters

The number of clusters determines the number of bits required to represent each one ($\text{num_bits} = \log_2(\text{num_clusters})$). The lower the number of clusters, lower the number of bits to represent them, and consequently smaller size for the model. However, this can come at the cost of accuracy of the model. It is thus useful to experiment with the number of bits suitable for a particular model architecture and task. In the case of CNNs, experiments show that convolutional layers need higher resolution (4 *bits*), while fully connected layers are more robust, and only seriously degrade model performance below 2 *bits*

How Many Bits do We Need?

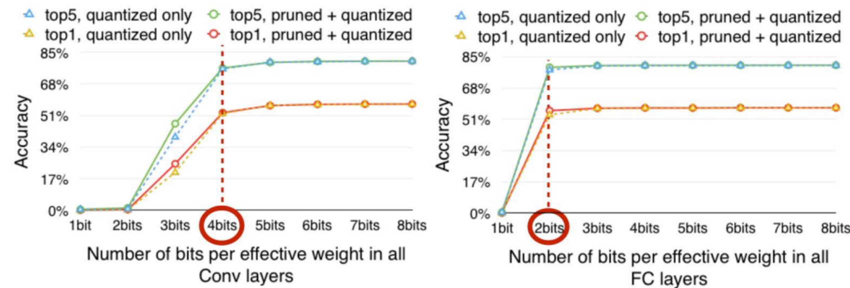


Figure 4: Number of Bits per Effective Weight in All Conv and FC Layers

4.7 Runtime

At runtime (inference / further training), the code book is used to decompress the weights into the floating point numbers they represent. Inference then proceeds as usual. The runtime still takes place in the (unquantized) floating point space, so there is no compute speedup. KMeans Quantization provides a storage compression compared to unquantized models, but takes the same or slightly longer time for compute (due to the decompression step).

5 Linear Quantization

5.1 Linear Quantization Basics

Moving towards linear quantization, we start with a 4x4 matrix represented using 32 bits. So the intuition behind linear quantization is that we try to quantize our original weights, which are in 32 bits, to integer values. Our target output values are integers but linear quantization is also pretty general and we can choose different target values. For example, we can quantize from 32 bits to 8 or 4 bits integer or we can quantize it in a manner of FP 32 to FP 16 or FP 32 to FP 8. It depends on how we choose our target. But here, we look at the most aggressive one, i.e., we want to quantize it into 2-bit integers. So what we basically try to do is to scale the original floating point value into a new range, and this new range is determined by our chosen target range. And in order to do this scaling, the equation that we try to follow is basically a linear mapping. We try to minus it by some value and we call this value, the zero point. Then we scale it by a floating point value. So basically, we are scaling from our original range to a new range and in the new range, numbers are represented through fewer choices.

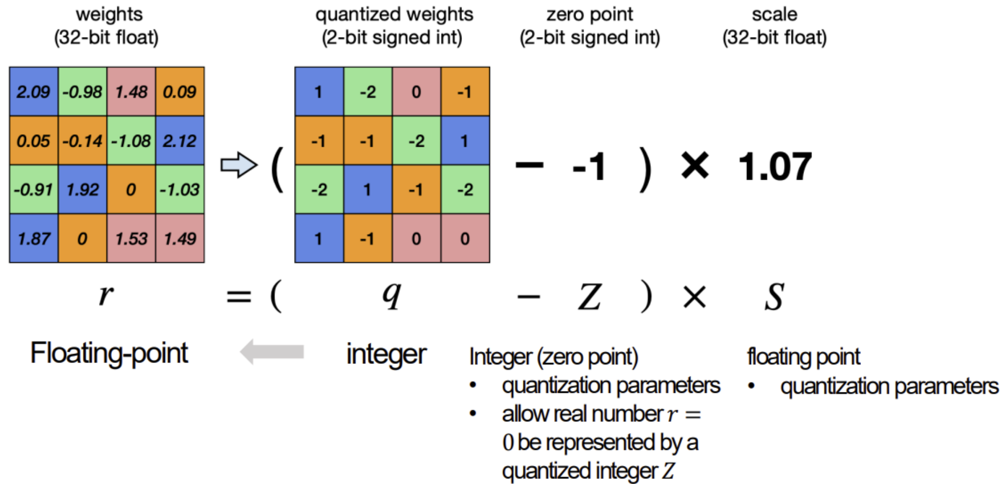


Figure 5: Linear Mapping of Integers to Real Numbers

5.2 Linear Quantization Parameters

So, we can express the scaling process through the above shown equation. We hold our original value which is r, and r is at high precision. We have our target value or the quantized value q, where q stands for quantized. Our intuition is that if we quantize our original value into q, we can still recover the quantized value back into the original value and preserve accuracy. The way we recover is basically by following the linear transformation, which will first minus a bias term Z, which basically represents zero point in the new range, and then scale it back to our original range. Linear quantization, therefore always carries these two critical parameters. So if we go into HuggingFace and look for language models, especially the quantized versions, and their configs, in addition to all those hidden dimensions and so on, we will observe these two values. These two values are also crucial in reproducing results.

So q is our target value which is an integer, and our original point which is r, is a floating point. Z is an integer because it is zero point over our target range and also, it is a quantization parameter. The meaning of Z is basically that we allow the real number r=0 to be represented by a quantized number Z in the target range. We then scale it back with S and S is basically a floating point quantization parameter.

Looking at this geometrically, we understand that we are doing some sort of normalization. We try to normalize a value that is in our original range into a smaller target range. We also try to align things. For example, we try to align the min value and the max value in our original range into our target range. We also try to align the zero point of these two ranges. If we do integer quantization and choose different bits of integers, we will have different q_{max} in the target range.

5.3 Parameter Determination

So now the next question that arises is how to determine S and Z. To answer this, we initially have our original value, which is a matrix and we try to find the maximum value and minimum value in this matrix, which is r_{max} and r_{min} and we follow the linear transformation and try to quantize it into the new integer range and we also try to use the new quantized value to recover the original value. So this gives us the two

equations.
r_{max} = S(q_{max} - Z), where we try to align the max value and
r_{min} = S(q_{min} - Z), where we try to align the min value.
Here we have two unknowns and two equations and therefore, this can be solved. We minus the two equations and we do some transformations and get the equation for S, which is basically normalization.

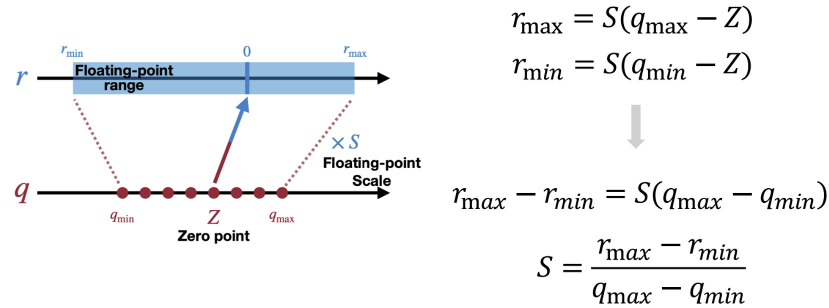


Figure 6: $r = S(q - Z)$: Determining S and Z

5.3.1 Parameter Determination Example

So let's do a practice case. We have a 4x4 matrix and our target range is a 2-bit integer, which is basically 4 values, which have been represented in the figure. -2 is the min value and 1 is the max value. We have to use linear quantization to quantize this matrix into the range we need. From the matrix, r_{max} is 2.12, and r_{min} is -1.08. putting the values into the equation for S, we get
S = (2.12 - (-1.08)) / (1 - (-2))
= 1.07, which is the scaling factor.

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

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

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

$$S = \frac{2.12 - (-1.08)}{1 - (-2)} = 1.07$$

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

$$= \text{round}\left(-2 - \frac{-1.08}{1.07}\right) = -1$$

Figure 7: Practice Solving for Determining S and Z

And once we have the S value, we can figure out the Z value, which is the zero point. Because Z is the zero point in the target range, and in our example, we choose an integer as our target range, therefore, we have to

take a round() in the equation of Z.

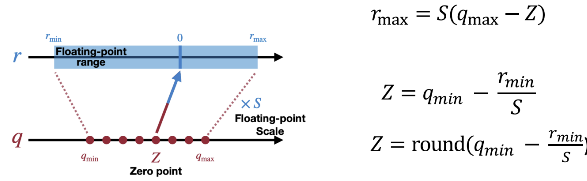


Figure 8: Equations for r_{max} and Z

With the same previous example, we can fill in the value for our equation for Z and we get the value as -1. So therefore, when we try to linearly quantize this matrix into our target range, we use a scaling factor, S of value 1.07, and a zero point, Z of value -1.

5.4 Apply Linear Quantization to Matrix Multiplication

A student asks a question - the professor answers that's called post-training quantization, which will be covered in the next lecture.

You have a few choices; you can choose a different S and Z for each tensor. You can also choose one constant S and Z for the entire model. This is called calibrating quantization. You want to calibrate the values of S and Z based on how much accuracy you lose in quantization. So apparently, if you do post-training quantization, you will have higher accuracy, but if you use a single S and Z for all weights and activations in the entire neural network and input data, you are going to lose more accuracy, right? So that is a tradeoff.

People use post-training quantization or entire model quantization, and they can figure out a calibration dataset. They quantize depending on the calibration dataset, or they can use some aggressive ones, for example, per-channel quantization. If I have a matrix where I am meeting the channel, I just try to figure out S and Z for each channel. I can even use per-group quantization. I determine that this group of values is pretty close, so I should designate S and Z for the group, but for another group where values are quite different, I designate a different S and Z. Does that make sense? So, when you move to more fine-grained quantization, it becomes very complex.

Apply Linear Quantization into Matmul

$$Y = WX$$

$$S_Y(q_Y - Z_Y) = S_W(q_W - Z_W)S_X(q_X - Z_X)$$

$$q_Y = \frac{S_W S_X}{S_Y}(q_W - Z_W)(q_X - Z_X) + Z_Y$$

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_W q_X - Z_X q_W + Z_W Z_X) + Z_Y$$

Figure 9: Equations for Applying Linear Quantization into Matmul

Here we have $Y = WX$ and we want to apply our quantization. Remember, at the beginning of the quantization lecture, we had to figure out two things: one is how you store data, which is pretty clear. How

you store data is by storing target range values, which are integers, and also storing a bit of S and Z based on the scheme you use.

The question is what kind of arithmetic course we use to compute after quantization. In the original equation $Y = WX$, we use floating-point operations. Now, after quantization, assuming we do some potential quantization, we introduce S_W, S_X, Z_W, and Z_X. We substitute them into the equation, perform a transformation, and find that our computation transforms into a new form.

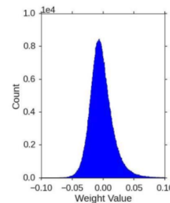
Here, q_w and q_x are the quantized weight and input X, and q_y is the quantized output. The reason we want to quantize the output is that this is a layer of the neural network, and then it proceeds to the next layer. After some manipulation, we get a new equation. Now, let's try to inspect this equation a little bit.

The first part we notice is that we can precompute it because the weight is fixed. So we can precompute how we quantize the weight. For X, if we calibrate the dataset, we can take a dataset and assume we are going to use the same scale and zero point for all the data in the dataset. Then, using the calibration dataset, we can determine a value for Z_X.

Apply Linear Quantization into Matmul

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_W q_X - Z_X q_W + Z_W Z_X) + Z_Y$$

- precomputed;
- N-bit integer multiplication
- 32-bit integer addition/subtraction



Empirically: $Z_W = 0$

$$q_Y = \frac{S_W S_X}{S_Y}(q_W q_X - Z_X q_W) + Z_Y$$

- Heavy lifting part
- integer multiplication

Figure 10: Equations for q_y

Using the calibration dataset, we identify that all the values in the orange box are constant and do not depend on the input, so we precompute them. They are precomputed, and we can ask: what does it take to compute these values? Here, Z is the zero value in a target range, so it's an integer, and q_w is the quantized value, which is also an integer. So, to perform these computations, we only need integer cores. Integer cores have a much higher FLOP efficiency compared to floating-point cores. Meanwhile, as we accumulate these values, we perform addition and subtraction, which also happen in the integer range. This means we can still use integer cores, which is advantageous.

In practice, we usually use an integer representation for values, but during addition and subtraction, we use 32-bit operations to avoid overflow.

Now, we examine the term $S_W \times S_X / S_Y$. Remember, S is a floating-point scale in the original range, which is problematic. If we compute this value directly, we must call floating-point cores, which defeats the purpose of quantization because, in some scenarios, quantization aims to avoid floating-point operations entirely and rely solely on integer cores.

How do we handle this?

Empirically, in many neural networks, this value lies in the range of 0 and 1 because it acts as a scaling

factor. So, a common trick is to avoid direct floating-point computation by using fixed-point multiplication plus bit-shifting. This trick represents the value as a power of 2. Of course, the power is a negative value multiplied by another fixed-point fraction, typically between 0.5 and 1. The reason for this approach is that fixed-point computation is much cheaper than floating-point computation. You don't have to touch floating-point cores, and bit shifting is much cheaper—it simply shifts bits, making fixed-point arithmetic easy.

Next, we look at the term $Z_W \times q_x$, where q_x depends on the input X and must be computed on the fly. Z_W is the zero point of the weight.

Observations show that many neural networks exhibit a Gaussian distribution with zero mean, particularly during training. Instead of explicitly calculating this value, we leverage this property. Since the mean is zero, we can quantize within a range that aligns the zero point, meaning that the zero point in the target range should also be zero in the original range.

Thus, we assume $D_W = 0$. Empirically, $Z_W = 0$. Let's simplify further. Of course, this is not entirely accurate—quantization is about approximation, and some accuracy loss is acceptable as long as it is minimal.

By substituting $D_W = 0$, we obtain a simplified equation. Now, the only heavy computation required is $q_w \times q_x$.

What does this mean?

Essentially, q_w and q_x are integer representations of the original W and X in the target range. Multiplying these two is equivalent to performing matrix multiplication using integer cores. This achieves our goal: reducing storage and simplifying arithmetic to integer arithmetic. Integer arithmetic is much faster, cheaper, and more energy-efficient than floating-point operations.

This provides a high-level intuition of what linear quantization does.

As mentioned at the beginning, different target ranges can be chosen. If the target range is LP8, then q_w and q_x are in LP8, reducing the original LP16 arithmetic to LP8 arithmetic. This is essentially what DeepSeek does, with the target range set to LP8.

The final slide provides practical performance insights. Linear quantization preserves much of the model's accuracy compared to K-means quantization. In K-means quantization, computation slows down, whereas in linear quantization, computation can sometimes be accelerated using lower-cost cores.

6 Contributions

- **Davit Abrahamyan:** Section 3.1-3.5
- **Andrew Choi:** Slide organization, Section 1, Section 2
- **Kayla Hom:** Slide assignment, editing to LaTeX, figures, Section 6
- **Jai Malegaonkar:** Section 3.6-3.73 and corresponding tables
- **Sairam Mahadeva Ganapathy:** Section 5.1- 5.3.1
- **Parsa Mirtaheri:** Section 4.1-4.3
- **Anikait Sunil Nair:** Section 5.1- 5.3.1
- **Nived Neetha Sooraj:** Section 5.4
- **Sharanya Ranka:** Section 4.4-4.7