1 Why Compress Large Language Models?

1.1 The computational problem

Large language models contain enormous numbers of parameters.

For example, consider a model with 70 billion parameters.

If each parameter is stored using a 32-bit floating-point number, the memory required for the weights alone is:

$$70 \times 10^9 \times 4 \text{ bytes} = 280 \text{ GB}$$

This is only the model weights.

It does not include:

- Intermediate activations.
- Attention-related memory.
- Key-value (KV) caches.
- Temporary computational buffers.
- Memory required for training gradients and optimizer states.

This creates several problems.

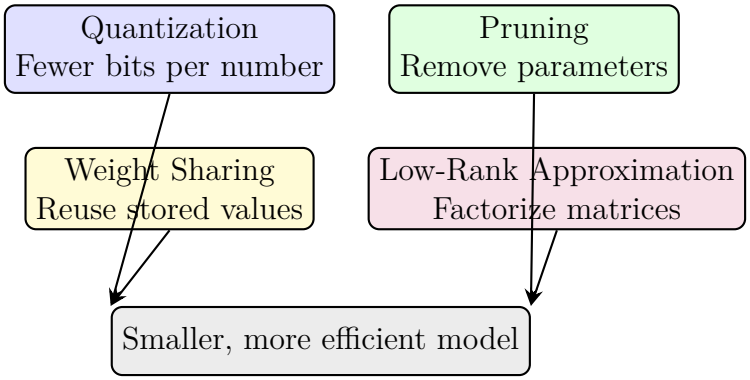
1. Large models require expensive hardware.
2. Moving parameters between memory and processors can become a bottleneck.
3. Large memory requirements limit the number of simultaneous users.
4. High computational requirements increase latency and energy consumption.
5. Training and fine-tuning become more expensive.

An important observation is that a model does not necessarily need to store every parameter with maximum numerical precision, or maintain every original connection independently.

We can often find a more economical representation.

1.2 Four approaches to compression

We will study four major techniques.



These techniques make different assumptions about the information stored in a neural network.

- Quantization assumes that many numbers can be represented approximately using fewer bits.
- Pruning assumes that some parameters or connections can be removed without substantially affecting performance.
- Weight sharing assumes that many parameters can use a common numerical value.
- Low-rank approximation assumes that a matrix can be approximated using a smaller number of independent directions.

The central question for each technique is:

> How much of the model's computational cost can we eliminate while preserving its useful behavior?

2 Quantization: Representing Numbers with Fewer Bits

2.1 What is quantization?

Neural network parameters are numerical values.

Traditionally, they are represented using floating-point numbers.

A common representation is FP32, which uses 32 bits per number.

However, many applications do not require 32-bit precision for every parameter.

We can represent the same approximate values using fewer bits.

This process is called quantization.

Common numerical formats include:

Format	Bits per value	Approximate storage
FP32	32	4 bytes
FP16	16	2 bytes
BF16	16	2 bytes
FP8	8	1 byte
INT8	8	1 byte
INT4	4	0.5 bytes

FP16 and BF16 are floating-point formats.

INT8 and INT4 are integer representations.

The exact numerical behavior differs between formats.

For example, BF16 has fewer mantissa bits than FP16, but a larger exponent range.

2.2 Analogy: Measuring length

Imagine measuring the length of a table.

We could represent its length as:

1.837492 meters

But if we are building a wooden table, we may only need:

1.84 meters

The additional decimal places do not provide meaningful benefit for our particular application.

We have reduced the precision of the representation without materially changing the result.

Quantization applies the same principle to model parameters.

Instead of storing every parameter with high precision, we use a smaller numerical representation.

2.3 Basic numerical example

Suppose a model contains the following parameters:

$$W = \begin{bmatrix} 0.12 & -0.84 & 0.53 \\ 0.91 & -0.31 & 0.07 \end{bmatrix}$$

With FP32, every value occupies 32 bits.

We could approximate the values using a smaller integer representation.

A simple symmetric quantization scheme is:

$$q = \text{round}\left(\frac{x}{s}\right)$$

and reconstruction is:

$$\hat{x} = sq$$

where:

- x is the original floating-point value.
- s is a scaling factor.

- q is the quantized integer.
- $\hat{x}$ is the reconstructed approximation.

For example, let:

$$s = 0.01$$

For:

$$x = 0.537$$

we obtain:

$$q = \text{round}(0.537/0.01) = 54$$

The reconstructed value is:

$$\hat{x} = 54(0.01) = 0.54$$

The quantization error is:

$$e = x - \hat{x} = -0.003$$

The original value is not preserved exactly, but the approximation is reasonably close.

2.4 Quantization error

The simplest way to measure quantization error is the difference between the original and reconstructed values.

For a single parameter:

$$e_i = w_i - \hat{w}_i$$

For an entire matrix, we might measure the squared error:

$$E = \sum_i (w_i - \hat{w}_i)^2$$

or the Frobenius norm:

$$E = \|W - \hat{W}\|_F^2$$

The goal is to reduce the number of bits while keeping the error sufficiently small.

However, an important distinction must be made:

$$\text{Small numerical error} \not\Rightarrow \text{Small model performance loss}$$

Some parameters are more important than others.

A small change in a particularly sensitive parameter may have a larger effect on the final output than a larger change elsewhere.

2.5 Uniform versus non-uniform quantization

Uniform quantization divides the numerical range into equally spaced intervals.

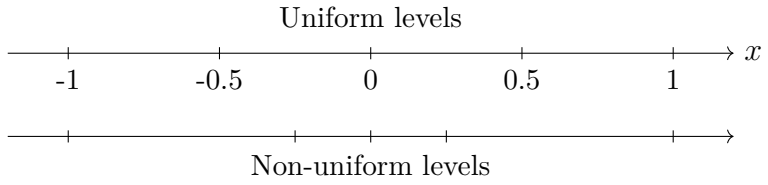
For example:

$$-1.0, -0.5, 0, 0.5, 1.0$$

Each value is represented using the nearest available quantization level.

Non-uniform quantization allows different intervals or levels.

For example, we may want more precision near zero if most model weights are concentrated around zero.



Non-uniform quantization can allocate more representational capacity to values that occur more frequently or are more important.

2.6 Symmetric and asymmetric quantization

In symmetric quantization, the range is centered around zero.

For signed integers:

$$q = \text{clip}\left(\text{round}\left(\frac{x}{s}\right), -q_{\max}, q_{\max}\right)$$

In asymmetric quantization, we also introduce a zero point:

$$q = \text{clip}\left(\text{round}\left(\frac{x}{s}\right) + z, q_{\min}, q_{\max}\right)$$

Reconstruction is:

$$\hat{x} = s(q - z)$$

Asymmetric quantization can better represent data whose range is not centered around zero.

2.7 Weight-only versus activation quantization

Quantization can be applied to different parts of a neural network.

Weight quantization

The model parameters are stored using fewer bits.

For example:

- FP16 weights.
- INT8 weights.
- INT4 weights.

Weight-only quantization is particularly attractive for LLM inference because the weights are repeatedly read from memory.

Activation quantization

Intermediate values generated during computation are also represented using fewer bits.

For example, instead of computing and storing activations in FP16, we may use INT8.

This can reduce:

- Activation memory.
- Memory bandwidth.
- Arithmetic costs, when supported by hardware.

However, activations can have unusual distributions and large outliers.

These outliers make quantization more difficult.

KV cache quantization

Transformers maintain key and value vectors for previously processed tokens during autoregressive generation.

These vectors are stored in the KV cache.

For long sequences and many concurrent requests, the KV cache can consume substantial memory.

Quantizing the KV cache can therefore improve the number of concurrent sequences and the maximum context length that fits in memory.

This is a different optimization from quantizing the model weights.

2.8 Why quantization is especially useful for LLMs

Consider a 70-billion-parameter model.

The approximate weight storage becomes:

Precision	Bits per parameter	Weight storage
FP32	32	280 GB
FP16	16	140 GB
INT8	8	70 GB
INT4	4	35 GB

These are idealized storage figures. Actual memory requirements are higher because of metadata, alignment, scales, zero points, buffers, and runtime overhead.

Nevertheless, the savings can be substantial.

A model that previously required multiple GPUs may become deployable on a much smaller hardware configuration.

2.9 Why does quantization improve computational speed?

There are two important reasons.

1. Less memory movement

A major bottleneck in modern neural networks is moving data between memory and the processor.

This is often called the memory bandwidth problem.

If we reduce the weight size from 16 bits to 4 bits, we reduce the amount of data that must be transferred, assuming comparable packing and memory access.

This can improve inference speed, especially when the workload is memory-bandwidth limited.

2. Faster arithmetic

Modern GPUs and specialized accelerators support low-precision arithmetic.

For supported formats, matrix multiplication can be faster than full-precision computation.

However, the actual speedup depends on:

- Hardware support.
- Matrix dimensions.
- Kernel implementation.
- Dequantization overhead.
- Batch size.
- Memory bandwidth.

Reducing precision does not automatically make every operation faster.

2.10 Quantization-aware training versus post-training quantization

There are two major approaches.

Post-training quantization (PTQ)

The model is trained normally.

After training, its parameters are converted to a lower-precision representation.

Some methods use a small calibration dataset to estimate appropriate scaling factors and identify sensitive layers.

Advantages:

- No need to repeat the original training process.
- Relatively inexpensive.
- Convenient for existing pretrained models.

Disadvantages:

- Quantization errors can damage model quality.
- Some layers are particularly sensitive.
- Very low bit widths may be difficult to achieve.

Quantization-aware training (QAT)

During training, we simulate the effects of quantization.

The model learns to compensate for the reduced numerical precision.

Conceptually:

$$W \to Q(W) \to \text{Forward pass}$$

The quantization operation is incorporated into the training process.

Because rounding is not differentiable in the usual sense, practical QAT methods often use a straight-through estimator (STE) to approximate the gradient.

Advantages:

- The model can adapt to quantization errors.
- Often better quality at aggressive precision.

Disadvantages:

- Requires additional training computation.
- More complicated training infrastructure.

2.11 Important LLM quantization methods

Several techniques are worth recognizing.

- **GPTQ:** A post-training quantization method using approximate second-order information to minimize quantization error in a layer-wise process.

- **AWQ (Activation-aware Weight Quantization):** Uses activation statistics to identify weights or channels that require special treatment.
- **SmoothQuant:** Transfers quantization difficulty between activations and weights through mathematical rescaling, making activation quantization easier.
- **bitsandbytes:** Provides practical low-bit quantization implementations, including 8-bit and 4-bit methods.
- **GGUF and related formats:** Commonly used for quantized model storage and local inference. They support multiple quantization representations.

2.12 Quantization limitations and long-tail effects

Quantization usually preserves common behavior well when it is carefully designed.

However, rare or unusual computations may be particularly sensitive to numerical precision.

Consider a model that normally produces activation values in the range:

$$[-1, 1]$$

But occasionally produces a value of:

$$100$$

If we use one uniform quantization scale for the entire range, most ordinary values may be represented with very little precision.

The outlier consumes much of the available numerical range.

This is why activation outliers are an important problem in LLM quantization.

Another problem involves rare capabilities.

A model may contain parameter directions that contribute little to average performance but are important for specialized tasks.

Quantization can distort those directions.

Analogy:

Imagine compressing a photograph.

The large objects and general scene remain recognizable.

However, small details, such as a distant street sign or a tiny object in the background, become unreadable.

The overall picture looks almost identical, but information has been lost.

Similarly, an LLM may retain excellent average performance while losing precision on unusual reasoning tasks.

The long-tail problem can therefore appear as:

- Reduced performance on rare languages.
- Errors involving unusual numerical values.

- Reduced reliability on specialized technical tasks.
- Degradation in uncommon token combinations.
- Increased sensitivity to unusual input distributions.

The effect is not inevitable. It depends on the model, quantization scheme, bit width, calibration data, and evaluation task.

3 Pruning: Removing Parameters and Connections

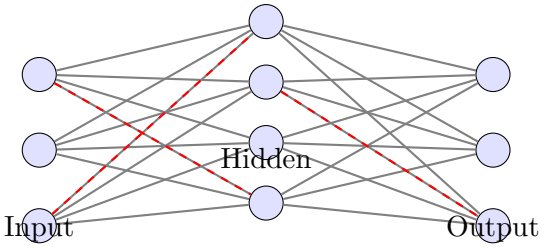
3.1 What is pruning?

A neural network contains many connections between neurons.

Each connection has an associated weight.

Pruning attempts to remove weights or structural components that contribute relatively little to the model's behavior.

Consider a simple neural network:



The red dashed connections represent connections that we might consider removing.

The idea is that not every connection is equally important.

3.2 Analogy: A road network

Imagine a city with thousands of roads.

Some roads carry enormous traffic.

Others are rarely used.

If we want to simplify the road network, we might remove some roads that contribute very little to transportation.

However, there is a danger.

A road that is rarely used might be the only road connecting a remote neighborhood to the rest of the city.

Removing it could have a disproportionate effect.

This is analogous to pruning neural networks.

A connection that contributes little to average behavior may still be important for unusual inputs.

3.3 Unstructured pruning

The simplest approach is to remove individual weights.

A common strategy is magnitude pruning.

The assumption is that weights with small absolute values are less important.

We remove weights satisfying:

$$|w_i| < \tau$$

where τ is a threshold.

Alternatively, we can remove a fixed percentage of the smallest weights.

For example:

$$W = \begin{bmatrix} 0.91 & 0.02 & -0.04 \\ 0.70 & -0.01 & 0.83 \end{bmatrix}$$

If we prune weights with absolute values below 0.05:

$$W_{\text{pruned}} = \begin{bmatrix} 0.91 & 0 & 0 \\ 0.70 & 0 & 0.83 \end{bmatrix}$$

We have removed three of the six weights.

The matrix is now 50% sparse.

3.4 Sparsity

Sparsity is the fraction of weights that are zero.

$$\text{Sparsity} = \frac{\text{Number of zero weights}}{\text{Total number of weights}}$$

For example, a matrix with 1,000 weights, of which 700 are zero, has:

$$\text{Sparsity} = 70\%$$

The density is the complementary quantity:

$$\text{Density} = 1 - \text{Sparsity}$$

A model with 70% sparsity contains 30% nonzero weights.

3.5 Why does pruning work?

Neural networks often contain redundancy.

Different parameters can contribute to similar computations.

A network may have enough capacity that removing some parameters does not substantially change its output.

There are several reasons for this.

1. Some weights have relatively little influence on the output.
2. Multiple neurons may learn overlapping representations.
3. Neural networks are often overparameterized.
4. Training can distribute information across many different connections.

Pruning attempts to identify and eliminate some of this redundancy.

3.6 Structured versus unstructured pruning

This distinction is especially important for practical LLM deployment.

Unstructured pruning

Removes individual weights.

For example:

$$W = \begin{bmatrix} 1 & 0 & 0.5 \\ 0 & 0.8 & 0 \\ 0.2 & 0 & 0.9 \end{bmatrix}$$

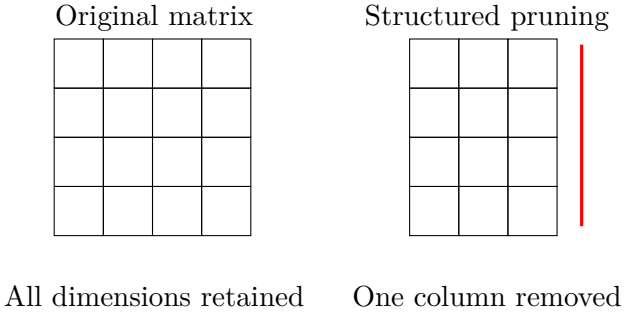
The resulting matrix contains many zeros, but its original dimensions remain unchanged.

Structured pruning

Removes entire structures, such as:

- Neurons.
- Attention heads.
- Channels.
- Transformer blocks.
- Rows or columns of matrices.

For example, removing an entire attention head can reduce the dimensions of subsequent computations.



Structured pruning is often easier to accelerate because the resulting matrices are physically smaller.

3.7 The hardware problem with unstructured pruning

Suppose we have a matrix with one million entries.

After pruning 90% of its weights, only 100,000 entries remain nonzero.

It may appear that we should obtain a tenfold speedup.

However, there is a complication.

The processor still needs to know:

- Where the nonzero values are.
- Which input values they multiply.
- How to navigate the sparse representation.

Sparse computation introduces indexing and memory access overhead.

A conventional dense matrix multiplication kernel is highly optimized and can operate very efficiently on GPUs.

A sparse operation is not necessarily faster.

Therefore:

$$90\% \text{ sparsity} \not\Rightarrow 10\text{x speedup}$$

This is a particularly important practical lesson.

3.8 N:M structured sparsity

One compromise is N:M sparsity.

For example, in 2:4 sparsity, every group of four weights contains exactly two nonzero weights.

The hardware can exploit this regular structure.

Instead of arbitrary zeros, the sparsity pattern is constrained.

This can enable more efficient sparse matrix multiplication on supported accelerators.

However, the model must be trained or adjusted to work with the imposed sparsity pattern.

3.9 Pruning and the lottery ticket hypothesis

An interesting research idea is the lottery ticket hypothesis.

The basic intuition is that a large, randomly initialized neural network may contain smaller subnetworks that, when trained appropriately, can achieve performance comparable to the original network.

Imagine buying thousands of lottery tickets.

A few tickets may contain a winning combination.

The hypothesis suggests that a sufficiently large network contains subnetworks that are particularly well suited to learning the task.

This idea helped motivate research into pruning, sparse training, and network structure.

It does not mean that arbitrary pruning will preserve performance.

3.10 Pruning limitations and long-tail effects

Pruning can preserve common capabilities while damaging specialized computations.

For example, imagine a model that answers millions of ordinary questions.

A particular group of connections may be activated primarily when processing a rare technical concept.

Those connections may have low average importance across the general training distribution.

A pruning algorithm may remove them.

The model continues to perform well on common tasks but loses some specialized ability.

This is similar to removing a rarely used road that is nevertheless essential for reaching a remote destination.

Other limitations include:

- Importance scores may be misleading.
- Removing one weight can change the importance of other weights.
- Sparse representations require specialized hardware or kernels.
- Aggressive pruning may require additional fine-tuning.
- Some model architectures are more sensitive to pruning than others.

A useful strategy is to prune iteratively:

1. Identify candidate weights or structures.
2. Remove a small percentage.
3. Evaluate performance.
4. Fine-tune if necessary.
5. Repeat.

This is generally safer than removing a large fraction of the model in one step.

4 Weight Sharing: Reusing Numerical Values

4.1 What is weight sharing?

In an ordinary neural network, each connection can have its own independently stored parameter.

Weight sharing allows multiple connections to use the same stored value.

For example, consider:

$$W = \begin{bmatrix} 0.11 & 0.12 & 0.13 \\ 0.10 & 0.11 & 0.12 \\ 0.13 & 0.10 & 0.11 \end{bmatrix}$$

Instead of storing nine separate values, we could use a smaller collection of representative values:

$$C = \{0.10, 0.11, 0.12, 0.13\}$$

Each original weight is represented by an index into this collection.

4.2 Analogy: A paint palette

Imagine painting a large picture.

You could purchase a separate container of paint for every pixel.

That would be extremely wasteful.

Instead, you purchase a small palette of colors and reuse those colors throughout the picture.

Weight sharing applies the same principle.

Rather than storing every numerical value independently, we maintain a collection of representative values and allow many weights to refer to the same value.

4.3 Clustering weights

A common approach is to cluster weights.

Suppose we have:

$$W = [0.11, 0.12, 0.13, 0.10, 0.11, 0.12]$$

We choose two representative values:

$$C = \{0.105, 0.125\}$$

Each weight is assigned to its nearest representative.

The approximate result might be:

$$\hat{W} = [0.105, 0.125, 0.125, 0.105, 0.105, 0.125]$$

Instead of storing six independent values, we store:

- Two representative values.
- Six cluster indices.

This is closely related to vector quantization and clustering algorithms such as k-means.

4.4 The mathematical formulation

Suppose a matrix contains N weights.
We want to represent each weight using one of K shared values.
Let:

$$C = \{c_1, c_2, \ldots, c_K\}$$

Each weight is assigned to a cluster:

$$z_i \in \{1, 2, \ldots, K\}$$

The reconstructed weight is:

$$\hat{w}_i = c_{z_i}$$

We can choose the cluster centers to minimize:

$$\min_{C,z} \sum_{i=1}^{N} (w_i - c_{z_i})^2$$

This is the familiar k-means objective.

4.5 How much memory can we save?

Suppose we have one million weights.
With FP16, each weight requires 16 bits.
The original storage is:

$$1,000,000 \times 16 = 16,000,000 \text{ bits}$$

Now suppose we use 256 shared values.
Each weight needs an 8-bit index because:

$$\log_2(256) = 8$$

The 256 representative values require:

$$256 \times 16 = 4096 \text{ bits}$$

The indices require:

$$1,000,000 \times 8 = 8,000,000 \text{ bits}$$

Total:

$$8,004,096 \text{ bits}$$

This is approximately half the original storage, before accounting for additional metadata and alignment.

4.6 Why does weight sharing work?

Neural networks often contain numerical redundancy.

A large collection of independently learned weights may have a relatively concentrated distribution.

Many values may be close to one another.

Instead of storing each value independently, we can approximate them using a smaller codebook.

The model sacrifices some numerical precision in exchange for a more compact representation.

4.7 A subtle distinction: Weight sharing versus quantization

These techniques are related but not identical.

Quantization generally restricts the precision of individual values.

Weight sharing restricts the number of distinct values used across the model.

For example:

- Quantization: Each weight is stored using 4 bits.
- Weight sharing: Each weight refers to one of 256 shared representative values.

In fact, weight sharing can be implemented using quantized indices.

The two methods can therefore be combined.

4.8 Weight sharing in neural architectures

Weight sharing also occurs naturally in neural network architectures.

For example, convolutional neural networks reuse the same filter across different spatial locations.

A convolutional filter is not independently learned for every pixel.

Similarly, transformer architectures may reuse parameters in certain designs, such as recurrent or shared-layer transformers.

This is conceptually different from clustering individual weight values, but both approaches reduce the number of independently stored parameters.

4.9 Limitations and long-tail effects

The major disadvantage is that multiple weights are forced to share a value.

Two weights may be close numerically but play different functional roles.

For example, a weight used in a common computation and another used in a rare computation may be assigned to the same cluster.

The rare computation may be more sensitive to the resulting approximation.

This can cause:

- Loss of fine-grained distinctions.
- Reduced accuracy in sensitive layers.
- Reduced performance on rare input patterns.
- Difficulty representing outlier weights.

Another issue is computational overhead.

The system must retrieve the appropriate shared value using its index.

Depending on hardware and implementation, this lookup can introduce additional work.

Weight sharing is therefore not guaranteed to improve inference speed, even when it reduces storage.

5 Low-Rank Approximation: Compressing Matrices

5.1 What is a matrix rank?

Low-rank approximation is based on an important idea from linear algebra.

The rank of a matrix is the number of linearly independent rows or columns.

Consider:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}$$

The second row is simply twice the first row.

The rows are linearly dependent.

Therefore:

$$\text{rank}(A) = 1$$

Although the matrix has four entries, it contains only one independent direction of information.

This observation is the foundation of low-rank approximation.

5.2 Analogy: Describing a photograph

Imagine describing a photograph of a landscape.

A photograph may contain millions of pixels.

But much of its structure may be explained by a relatively small number of patterns:

- Sky.
- Mountains.
- Ground.
- Buildings.
- Shadows.

We may not need to store every pixel independently if we can reconstruct most of the image using a smaller collection of underlying patterns.

Low-rank approximation attempts to find similar redundancy in matrices.

5.3 Matrix factorization

Suppose we have a weight matrix:

$$W \in \mathbb{R}^{m \times n}$$

Instead of storing the entire matrix, we approximate it as:

$$W \approx AB$$

where:

$$A \in \mathbb{R}^{m \times r}$$

and:

$$B \in \mathbb{R}^{r \times n}$$

The integer r is the chosen rank.

If:

$$r \ll \min(m, n)$$

then we may save a substantial amount of memory.

5.4 Numerical example

Suppose:

$$W \in \mathbb{R}^{1000\times 1000}$$

The original matrix contains:

$$1000 \times 1000 = 1,000,000$$

parameters.
Now approximate it using:

$$A \in \mathbb{R}^{1000\times 50}$$

and:

$$B \in \mathbb{R}^{50\times 1000}$$

The total number of parameters becomes:

$$1000 \times 50 + 50 \times 1000 = 100,000$$

We have reduced the parameter count by 90%.
The original matrix multiplication:

$$y = Wx$$

requires approximately:

$$mn$$

multiply-accumulate operations per vector.
The factorized version:

$$y = A(Bx)$$

requires approximately:

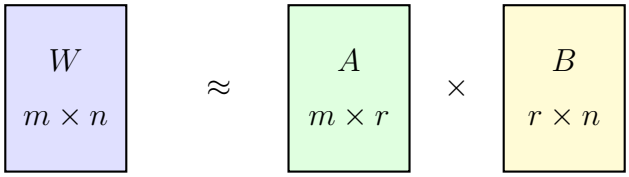
$$rn + mr$$

operations.
For the example:

$$1,000,000 \quad \longrightarrow \quad 100,000$$

This can significantly reduce computation.

5.5 Visualizing the factorization



Instead of storing one large matrix, we store two smaller matrices.

5.6 Singular Value Decomposition (SVD)

One important mathematical method for obtaining low-rank approximations is Singular Value Decomposition.

Any real matrix W can be decomposed as:

$$W = U\Sigma V^T$$

where:

- U contains orthogonal left singular vectors.
- V contains orthogonal right singular vectors.
- Σ contains singular values.

The singular values are ordered:

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$$

They describe the strength of the corresponding independent directions.

If most singular values are small, we can discard some of them.

For example:

$$\Sigma = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 0.2 & 0 \\ 0 & 0 & 0 & 0.01 \end{bmatrix}$$

The first two singular values dominate.

We might approximate the matrix using only those two directions.

This gives a rank-2 approximation:

$$W_2 = U_2\Sigma_2 V_2^T$$

The Eckart–Young theorem tells us that truncating the SVD to the largest k singular values produces the best rank-k approximation in both the Frobenius norm and the spectral norm.

5.7 Why does low-rank approximation work?

Many neural network weight matrices contain redundancy.

Their rows or columns may not represent completely independent transformations.

A large matrix may be approximated using a smaller number of important directions.

This is related to:

- Principal Component Analysis (PCA).
- Dimensionality reduction.
- Latent factor models.
- Matrix completion.
- Feature extraction.

The central idea is that a high-dimensional representation may have a lower-dimensional underlying structure.

5.8 Low-rank approximation in transformers

Transformers contain several large matrices.

For example, attention uses projections:

$$Q = XW_Q$$

$$K = XW_K$$

$$V = XW_V$$

and the attention output is projected through another matrix.

The feed-forward network also contains large linear transformations.

These matrices can potentially be approximated using low-rank factorization.

For example:

$$W_Q \approx A_Q B_Q$$

Then:

$$Q = X(A_Q B_Q)$$

This may reduce both storage and multiplication costs.

However, the rank must be chosen carefully.

A rank that is too small may remove important representational directions.

5.9 Low-rank approximation versus LoRA

These two concepts are related, but they are not the same.

Low-rank approximation

Attempts to represent an existing matrix using a lower-rank factorization.

For example:

$$W \approx AB$$

The goal is usually to reduce the size of an existing transformation.

LoRA (Low-Rank Adaptation)

LoRA uses low-rank matrices to represent a parameter update during fine-tuning.

Instead of directly modifying a large pretrained weight matrix:

$$W' = W + \Delta W$$

we represent the update as:

$$\Delta W = BA$$

where the rank is small.

Thus:

$$W' = W + BA$$

The original model weights can remain frozen.

Only the small matrices A and B need to be trained.

This makes fine-tuning substantially cheaper.

An important distinction:

LoRA primarily reduces the number of trainable parameters and optimizer states.

It does not automatically reduce the inference cost of the original model.

The low-rank update may need to be merged into the original weights or otherwise handled at inference time.

5.10 Limitations and long-tail effects

Low-rank approximation assumes that most of the important information can be represented by a relatively small number of directions.

This may work very well for the dominant structure of the matrix.

However, small singular directions can still be important.

Consider a model that performs well on common mathematical problems.

A rare mathematical operation may depend on a direction represented by a relatively small singular value.

Removing that direction may have little effect on average benchmark performance but cause failure on that particular task.

Analogy:

Imagine reducing a large orchestra to a small group of musicians.

The main melody can still be reproduced.

However, a subtle instrument that appears only briefly in the original composition may disappear.

The overall composition remains recognizable, but a particular detail is lost.

Low-rank approximation can therefore preserve dominant behavior while removing weaker but specialized capabilities.

Other limitations include:

- Selecting the appropriate rank is difficult.
- Some matrices do not have a strongly concentrated singular-value spectrum.
- Factorization introduces additional operations.
- Computing SVD for very large matrices can be expensive.
- The best approximation under matrix error does not necessarily preserve downstream task accuracy.

6 The Long-Tail Problem Across Compression Methods

6.1 A common underlying problem

All four techniques make some form of approximation.

They attempt to preserve the most important information while eliminating what appears to be redundant or less significant.

But importance is often measured statistically.

A parameter may contribute little to average behavior while being essential for a rare capability.

This is the fundamental long-tail concern.

6.2 An analogy: Compressing a dictionary

Imagine a dictionary containing one million words.

Most words are used frequently.

Others are extremely rare.

Suppose we want to create a smaller dictionary.

We could remove uncommon words, approximate their spellings, or group similar words together.

The resulting dictionary might be perfectly adequate for ordinary conversations.

But a specialist trying to read an old scientific manuscript may encounter words that no longer exist in the compressed version.

The dictionary is still useful, but its coverage has become narrower.

LLM compression can produce a similar effect.

6.3 Four ways the long tail can be damaged

Technique	**What is discarded or approximated?**	**Possible long-tail failure**
Quantization	Fine numerical distinctions.	Sensitive numerical or specialized computations become less reliable.
Pruning	Individual weights or structural components.	Rarely activated connections supporting unusual tasks are removed.
Weight sharing	Differences between individual parameter values.	Specialized weights are forced to share values with unrelated parameters.
Low-rank approximation	Weak or less dominant matrix directions.	Unusual capabilities represented in those directions are lost.

6.4 Why average benchmarks can be misleading

Suppose we evaluate a model on 10,000 questions.

The questions are distributed as follows:

- 9,000 common questions.
- 900 uncommon questions.
- 100 rare questions.

Suppose the original model and compressed model achieve:

Category	Original	Compressed
Common	95%	94%
Uncommon	90%	80%
Rare	85%	40%

The original model's overall accuracy is:

$$\frac{9000(0.95) + 900(0.90) + 100(0.85)}{10000} = 94.3\%$$

The compressed model's overall accuracy is:

$$\frac{9000(0.94) + 900(0.80) + 100(0.40)}{10000} = 92.6\%$$

The average performance difference is only 1.7 percentage points.

However, performance on rare questions has dropped from 85% to 40%.

An average score hides this substantial degradation.

6.5 A useful evaluation strategy

When evaluating compressed LLMs, we should not rely exclusively on average benchmark scores.

We should also evaluate:

1. Common task performance.
2. Rare task performance.
3. Specialized technical domains.
4. Long-context reasoning.
5. Mathematical and numerical precision.
6. Multilingual performance.
7. Robustness to unusual inputs.
8. Calibration and uncertainty.

It is also useful to evaluate performance at different levels of compression.

For example:

Compression level	Performance
0%	100%
25%	99%
50%	97%
75%	91%
90%	78%

The relationship is not necessarily linear.

A model may tolerate moderate compression very well and then experience a sharp performance decline beyond some threshold.

6.6 Importance is not the same as frequency

A particularly important observation:

> A parameter or computational pathway can be rarely used but still be essential.

For example, an emergency braking system may be used very rarely.

That does not make it unimportant.

Similarly, an LLM may rarely encounter a particular combination of mathematical concepts, programming constructs, or technical terminology.

The associated capabilities should not necessarily be discarded simply because they contribute little to average activity.

This distinction suggests that compression algorithms should consider not only average parameter magnitude or frequency, but also sensitivity and task importance.

7 Combining Compression Techniques

7.1 Why combine them?

The four methods attack different sources of computational cost.

They are not mutually exclusive.

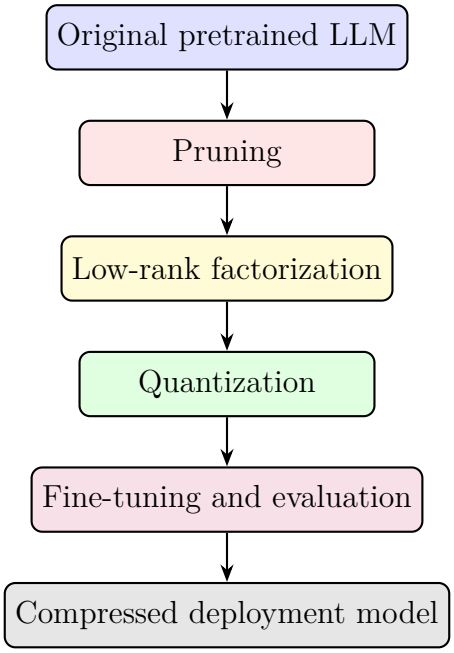
For example, a model could be:

1. Pruned to remove unnecessary connections.
2. Quantized to reduce the precision of remaining weights.
3. Factorized to reduce large matrix dimensions.
4. Stored using shared numerical values.

In practice, combinations must be designed carefully because one compression operation can change the effectiveness of another.

For example, quantizing a matrix before pruning it may produce different results from pruning first and then quantizing.

7.2 An example compression pipeline



This is an illustrative pipeline, not a universally optimal ordering.
The best order depends on the architecture, hardware, training data, and target performance.

7.3 Compression versus distillation

These techniques are often used alongside knowledge distillation.
Recall that distillation transfers behavior from a teacher model to a student model.
Compression modifies how a model is represented or computed.
For example:

1. A large teacher produces training examples.
2. A smaller student learns from the teacher.
3. The student is quantized.
4. Unimportant structures are pruned.
5. The compressed student is evaluated and possibly fine-tuned.

Distillation and compression solve related but different problems.
Distillation transfers behavior.
Compression reduces the cost of representing or executing a model.

8 Comparing the Four Techniques

Method	Primary savings	Main assumption	Main limitation
Quantization	Memory bandwidth and numerical storage	Lower precision is sufficient	Numerical sensitivity and outliers
Pruning	Parameter count and potentially computation	Some parameters are unnecessary	Sparse execution overhead
Weight sharing	Number of distinct stored values	Many weights can share values	Loss of individual distinctions
Low-rank approximation	Matrix storage and multiplication	Important information lies in fewer directions	Loss of weaker but important directions

8.1 Which method should we choose?

The answer depends on the bottleneck.

- If model weights do not fit in memory, quantization is often an attractive first step.
- If a model contains redundant structures and hardware supports sparse computation, pruning may be useful.
- If weights have substantial numerical redundancy, weight sharing can reduce storage.
- If large matrices have concentrated singular-value spectra, low-rank approximation may reduce both storage and computation.
- If fine-tuning is expensive, LoRA can reduce the number of trainable parameters.

There is no universally best compression technique.

The correct choice depends on the target hardware, the workload, and the quality requirements.

9 Conclusion

9.1 The central ideas

We studied four ways to make large language models more computationally efficient.

Quantization

Store numerical values using fewer bits.

Pruning

Remove parameters or structures that contribute relatively little.

Weight sharing

Allow multiple parameters to reuse a smaller collection of numerical values.

Low-rank approximation

Represent large matrices using smaller factorized matrices.

All four techniques exploit redundancy or reduce the amount of information that must be stored and processed.

9.2 The most important trade-off

The central challenge is deciding what information can safely be discarded or approximated.

A model's average behavior may survive substantial compression.

However, rare capabilities can be much more sensitive.

This is why compression should be evaluated not only by model size and average accuracy, but also by its effect on the full distribution of tasks.

9.3 Takeaway

The important lesson is that a large neural network does not necessarily need to be represented and executed in its original form.

Much of its computational cost may come from redundant precision, redundant connections, repeated numerical values, or correlated matrix directions.

Quantization, pruning, weight sharing, and low-rank approximation exploit these properties in different ways.

But compression is fundamentally an approximation problem.

The goal is not simply to make the model smaller.

The goal is to preserve the capabilities that matter while reducing the computational resources required to deliver them.