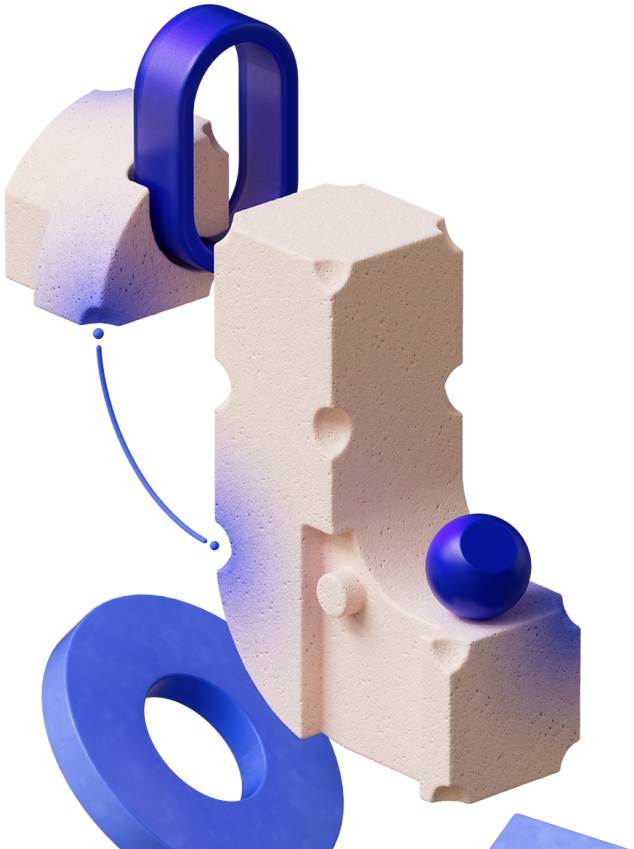
Google DeepMind



Efficiency in LLMs

Nikita Saxena
Research Engineer

Agenda

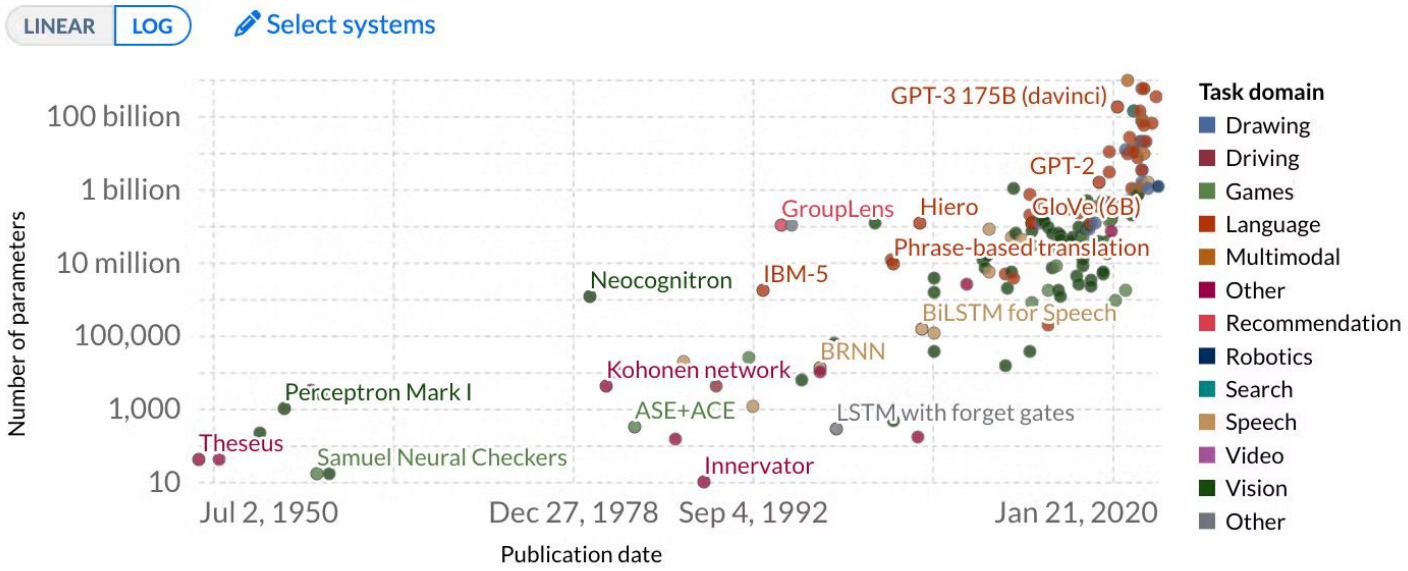
The Need for Efficiency

1

Parameters in notable artificial intelligence systems

Our World in Data

Parameters are variables in an AI system whose values are adjusted during training to establish how input data gets transformed into the desired output; for example, the connection weights in an artificial neural network.



Model	Number of parameters
GPT-4	1.5 Trillion
BARD	1.6 Trillion
LLaMA	1.2 Trillion
BLOOM	176 billion
LaMDA	137 billion
Flan-UL2	20 billion
Mistral 7B	7 billion

Source: Epoch (2023)

OurWorldInData.org/artificial-intelligence • CC BY

Note: Parameters are estimated based on published results in the AI literature and come with some uncertainty. The authors expect the estimates to be correct within a factor of 10.

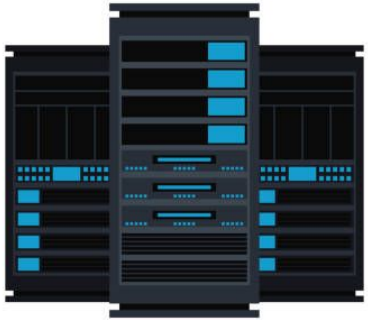
The Hidden Cost of LLM Intelligence

Cost

Training: GPT-3 cost $4.6M+ for a single run.

Inference: The ongoing, dominant operational cost.

Hardware Wall



Memory: Llama 2 70B needs ~140GB of VRAM (FP16).

Accessibility: Out of reach for consumer hardware & edge devices.

Need for Speed

Latency: Users expect instant responses.

Throughput: Serving millions of users requires high efficiency.

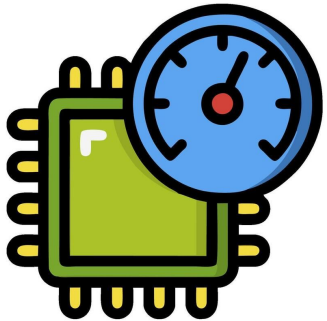
The Performance Bottlenecks

Memory

Can it even fit?

The model's weights and its dynamic run-time state (like the KV Cache) must fit into the limited VRAM of a GPU.

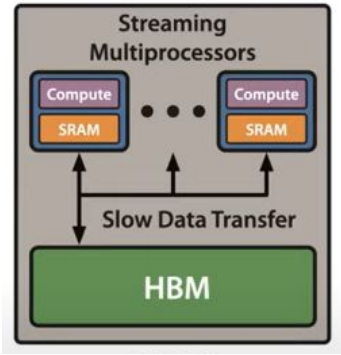
Compute



Can we calculate fast enough?

The speed is limited by the raw mathematical throughput of the hardware—its ability to perform FLOPs (Floating-Point Operations Per Second).

Memory Bandwidth



Can we feed the processor fast enough?

The processor is idle, waiting for data to travel from slow, large memory to fast, small on-chip memory.

What We Will Cover

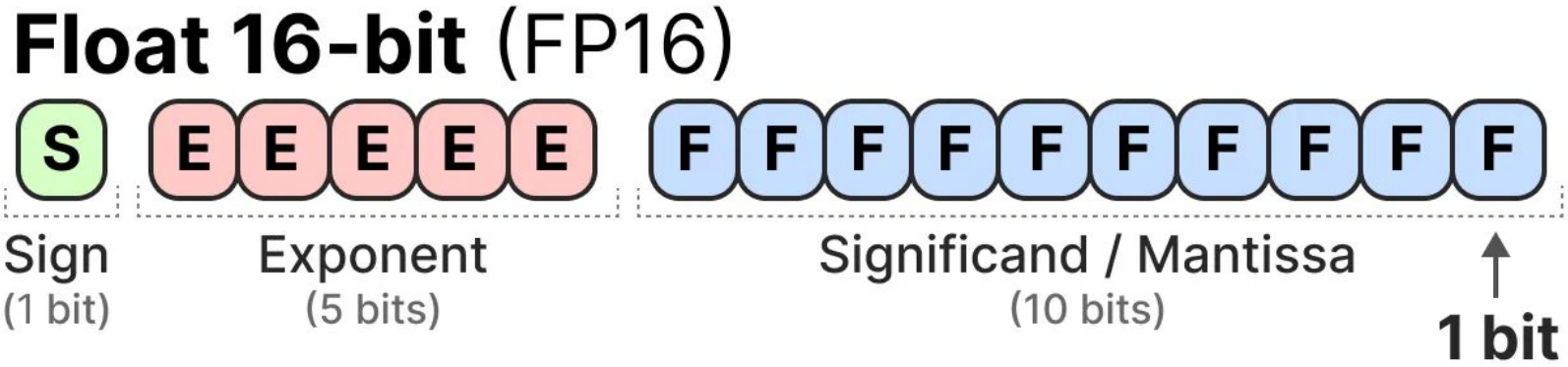
1. Model Compression
    a. Quantization
    b. Pruning
2. Efficient Attention
    a. Paged Attention
    b. Flash Attention
3. Training Methods
    a. Distillation
4. Advanced
    a. Matryoksha Embeddings
    b. Speculative Decoding

Quantization

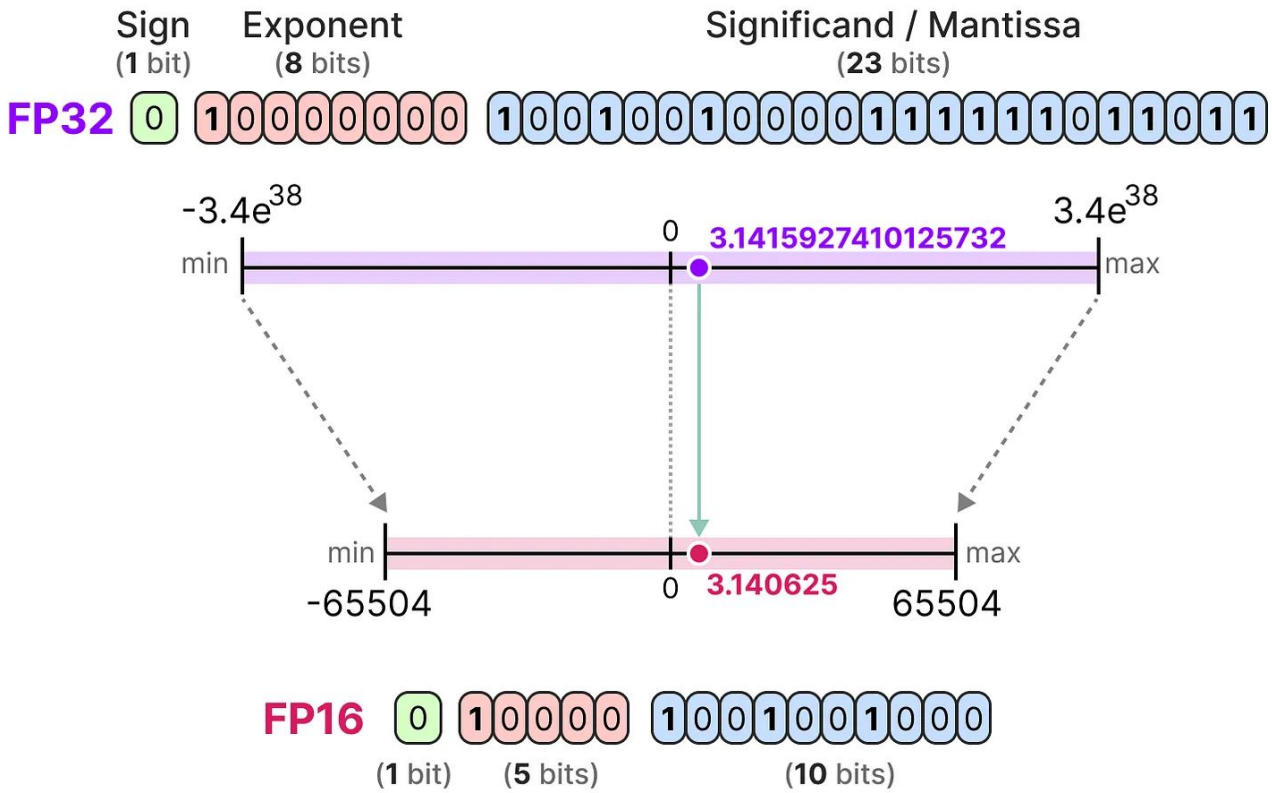
01

Floating Point Numbers

Representation

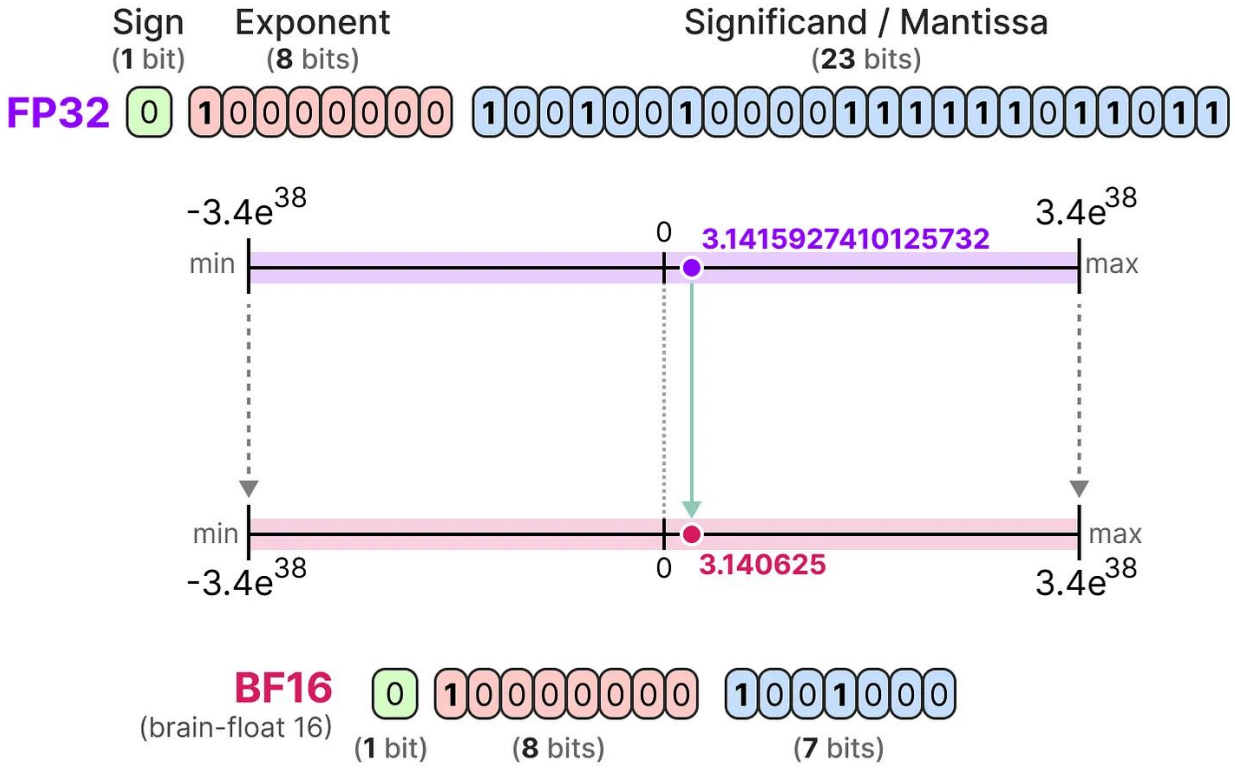


Half-Precision: FP16



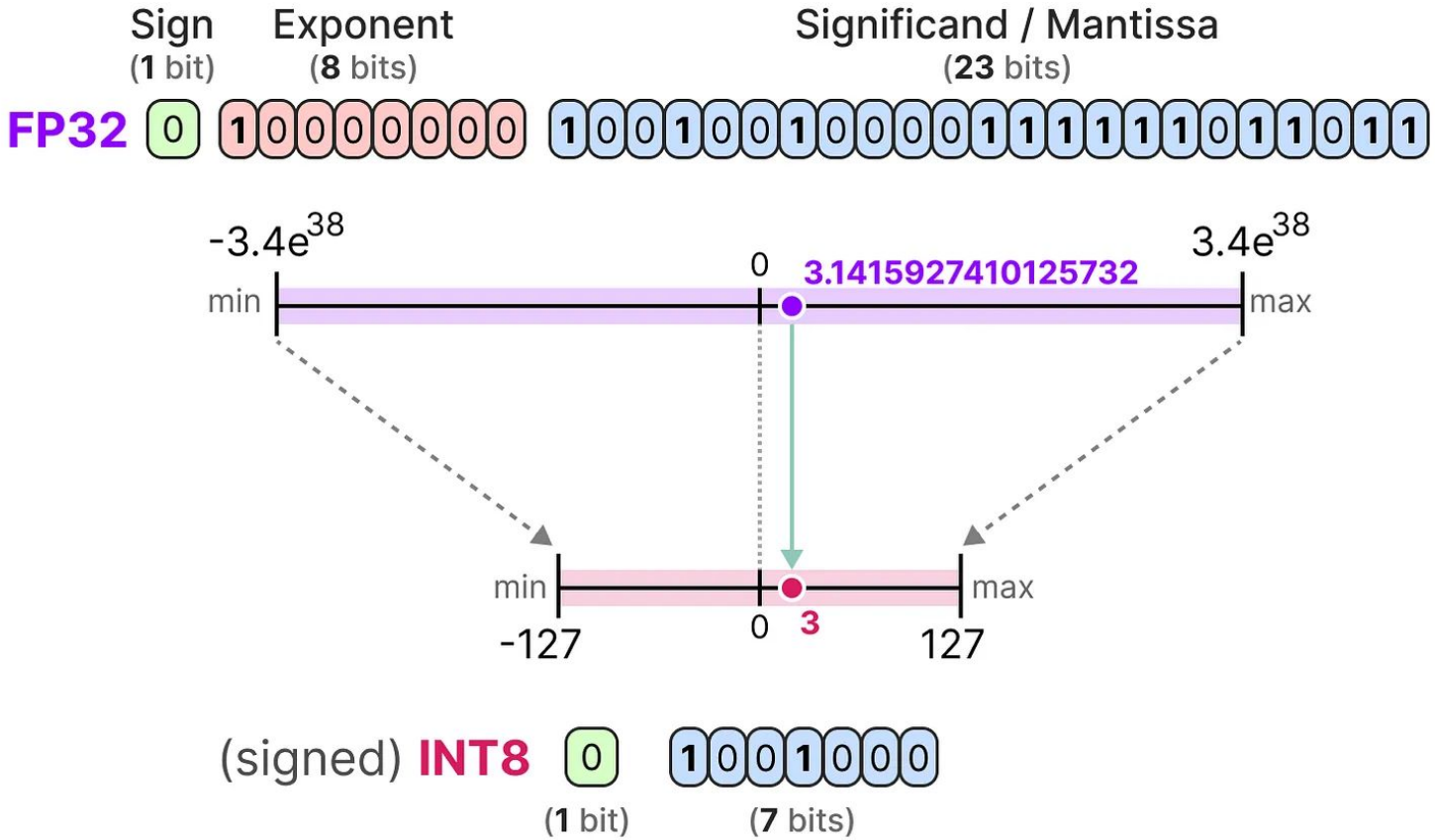
Notice how the range of values FP16 can take is quite a bit smaller than FP32.

Truncated FP32: BF16



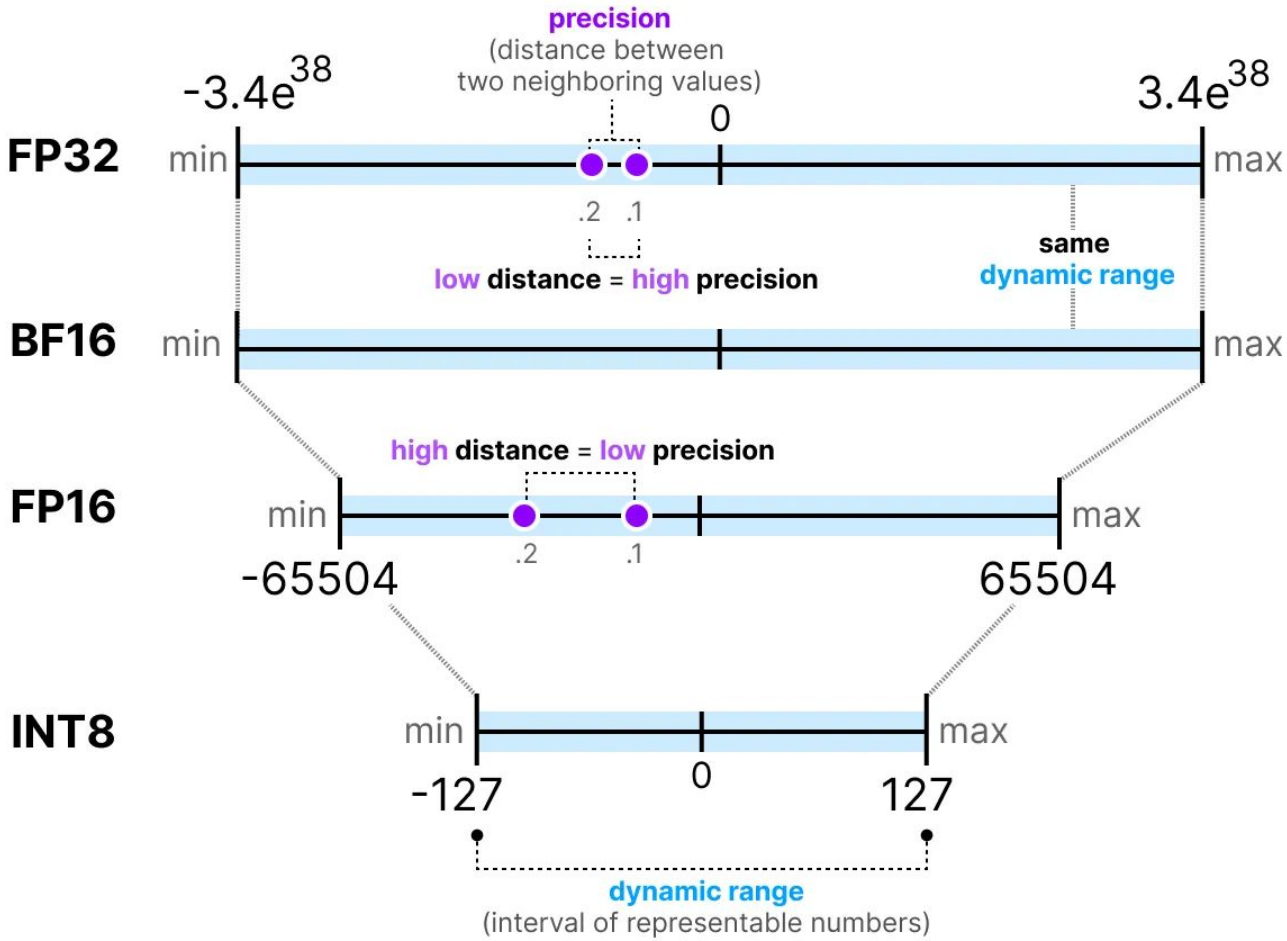
Note: BF16 uses the same amount of bits as FP16 but can take a wider range of values and is often used in deep learning applications.

Integer-based: INT8



Precision and Range

The interval of representable numbers a given representation can take is called the dynamic range whereas the distance between two neighboring values is called precision.



02

The Need for Quantization

Memory of Floating Point Representations

Since there are 8 bits in a byte of memory, we can create a basic formula for most forms of floating point representation.

$$\text{memory} = \frac{\text{nr\_bits}}{8} \times \text{nr\_params}$$

Memory of LLMs

Let's assume that we have a model with 70 billion parameters.

Most models are natively represented with float 32-bit (often called full-precision), which would require 280GB of memory just to load the model

$$\textbf{64-bits} = \frac{64}{8} \times 70B \approx \textbf{560 GB}$$

$$\textbf{32-bits} = \frac{32}{8} \times 70B \approx \textbf{280 GB}$$

$$\textbf{16-bits} = \frac{16}{8} \times 70B \approx \textbf{140 GB}$$

As such, it is very compelling to minimize the number of bits to represent the parameters of your model (as well as during training!).

However, as the precision decreases the accuracy of the models generally does as well.

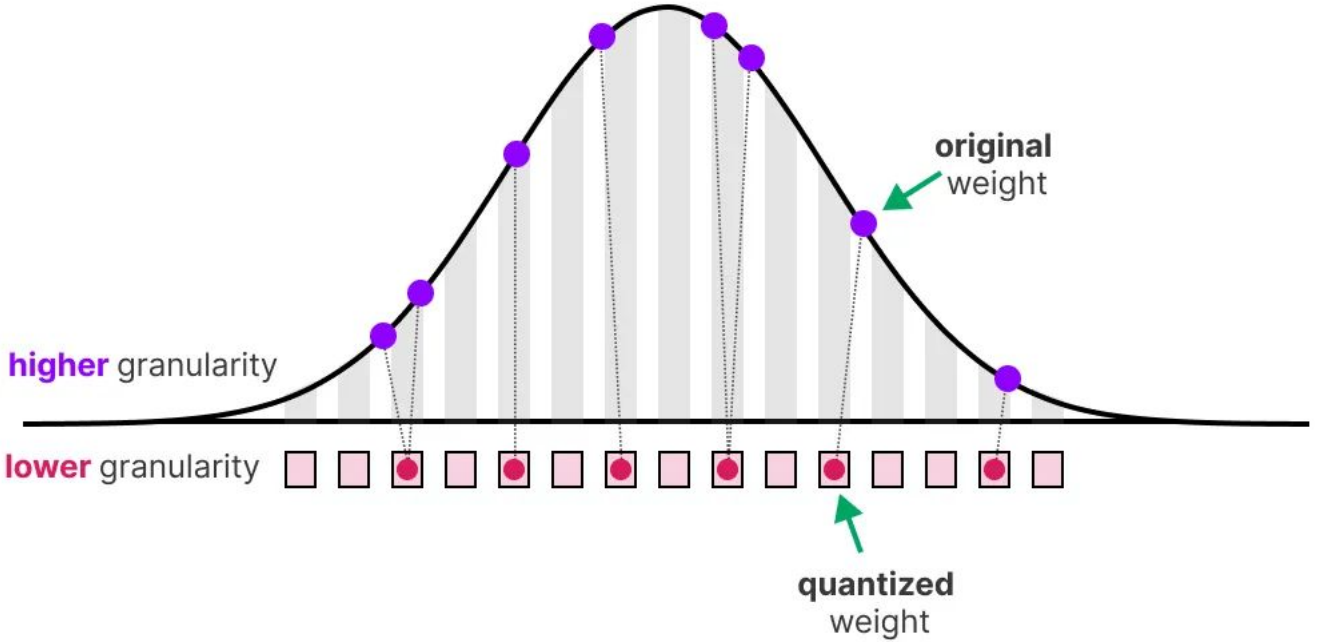
We want to reduce the number of bits representing values while maintaining accuracy... This is where quantization comes in!

03

What is Quantization

Introduction

Quantization aims to reduce the precision of a model's parameter from higher bit-widths (like 32-bit floating point) to lower bit-widths (like 8-bit integers).



Loss of Precision

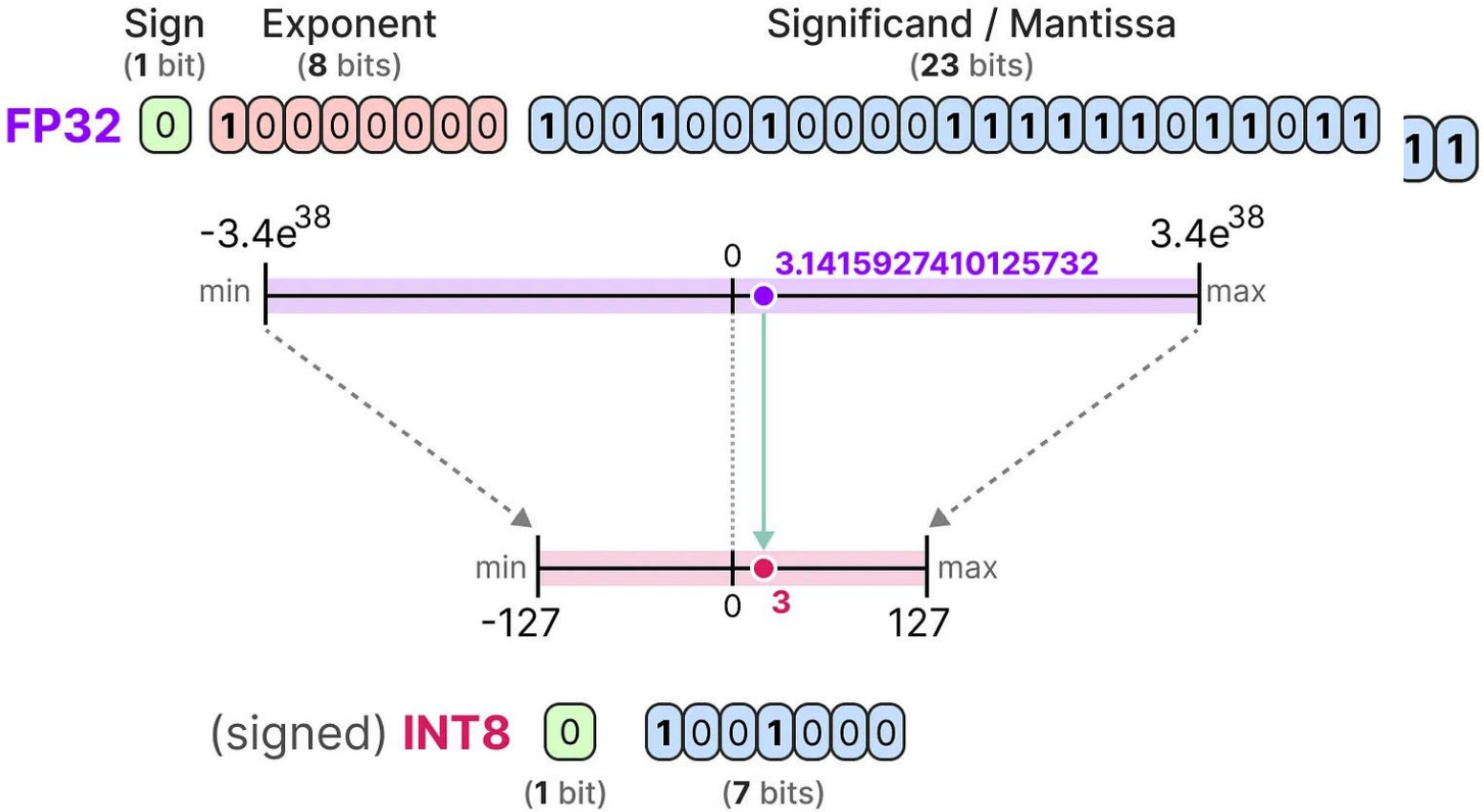
Original Image

“Quantized” Image

03

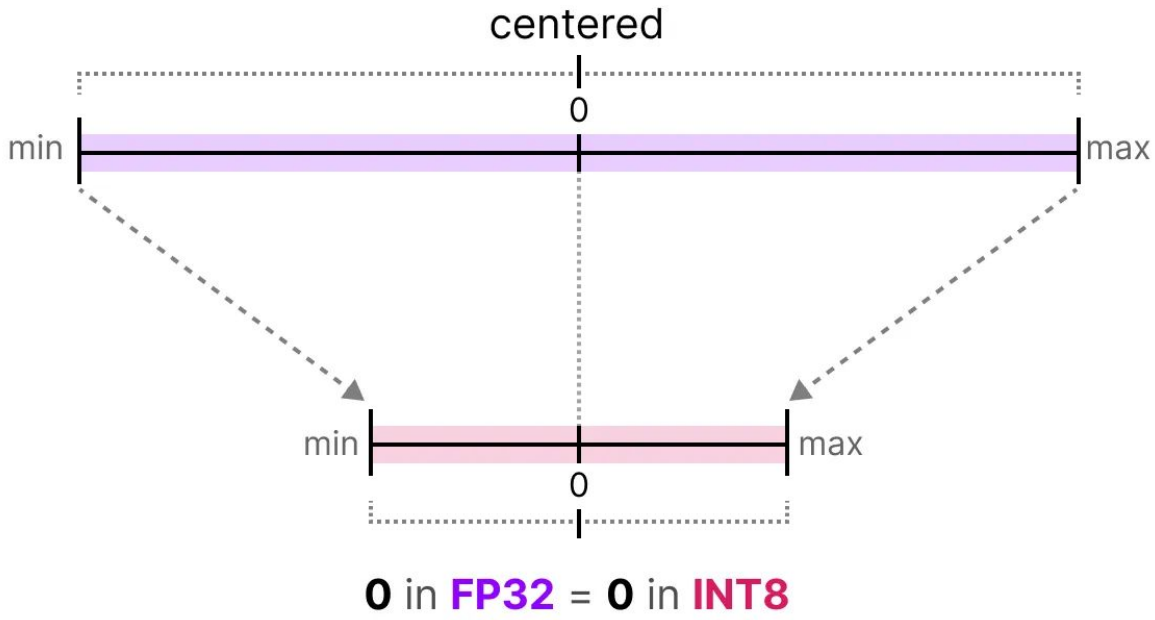
Squeezing/Mapping Methods

For each reduction in bits, a mapping is performed to “squeeze” the initial FP32 representations into lower bits.



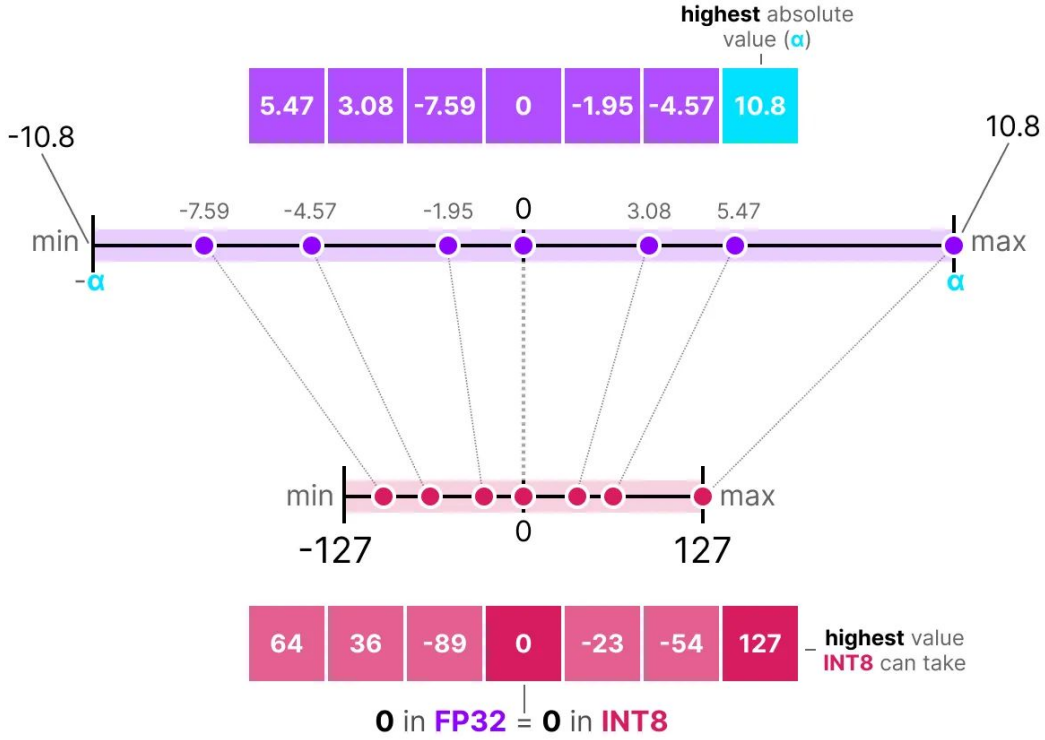
Symmetric Quantization

the range of the original floating-point values is mapped to a symmetric range around zero in the quantized space.



Symmetric Quantization: *absmax*

Given a list of values, we take the highest absolute value (α) as the range to perform the linear mapping.



Symmetric Quantization: *absmax*

Since it is a linear mapping centered around zero, the formula is straightforward.

We first calculate a scale factor (s) using:

- b is the number of bytes that we want to quantize to (8),
- α is the highest absolute value,

Then, we use the s to quantize the input x

$$s = \frac{2^{b-1} - 1}{\alpha}$$

(scale factor)

$$x_{quantized} = \text{round}\left(s \cdot x\right)$$

(quantization)

Symmetric Quantization: *absmax*

Filling in values.

$$s = \frac{127}{10.8} = 11.76 \qquad \text{(scale factor)}$$

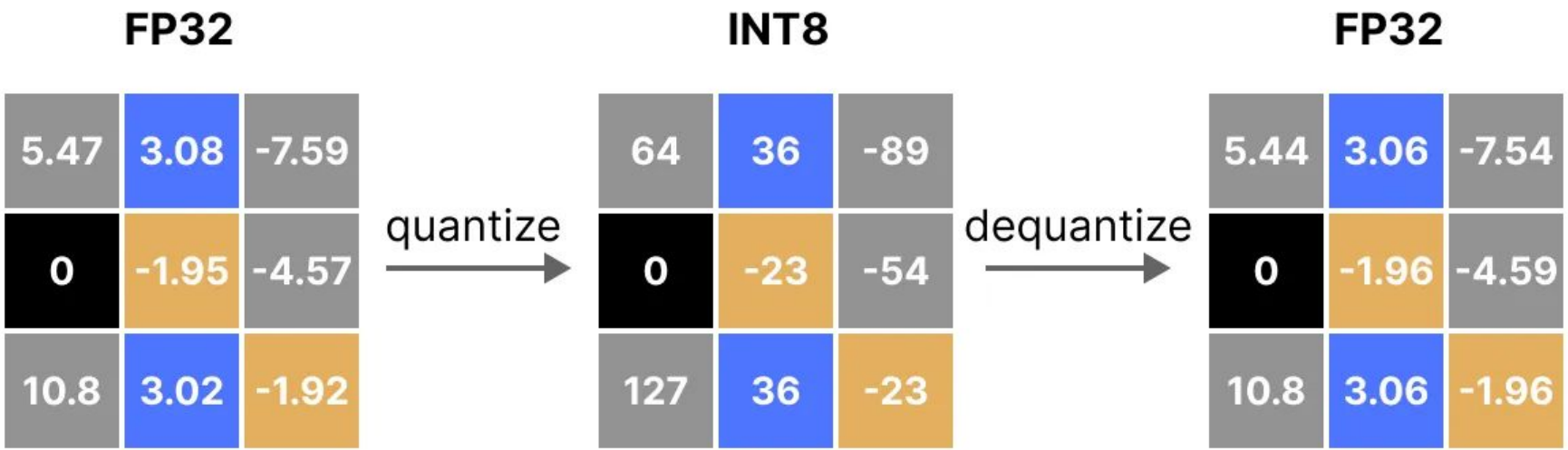
$$x_{quantized} = \text{round}\left(11.76 \cdot \blacksquare\right) \qquad \text{(quantization)}$$

To retrieve the original FP32 values, we can use the previously calculated scaling factor (s) to dequantize the quantized values.

$$x_{dequantized} = \frac{\blacksquare}{s} \qquad \text{(dequantize)}$$

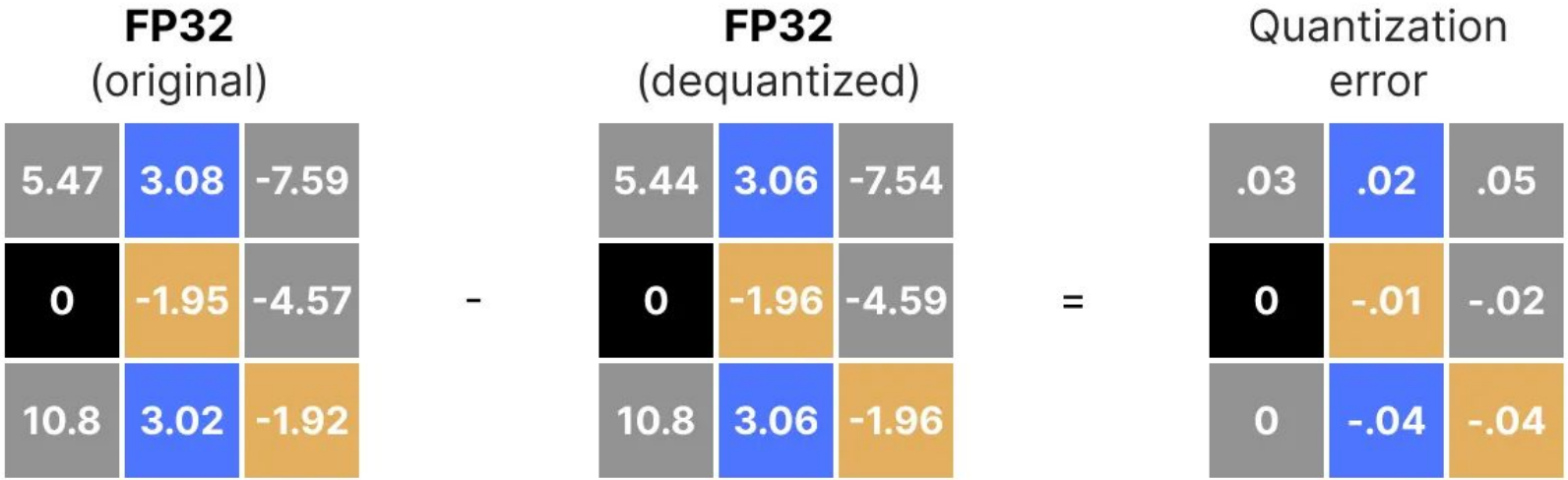
Quantization Error

Applying the quantization and then dequantization process to retrieve the original looks as follows



You can see certain values, such as 3.08 and 3.02 being assigned to the INT8, namely 36. When you dequantize the values to return to FP32, they lose some precision and are not distinguishable anymore.

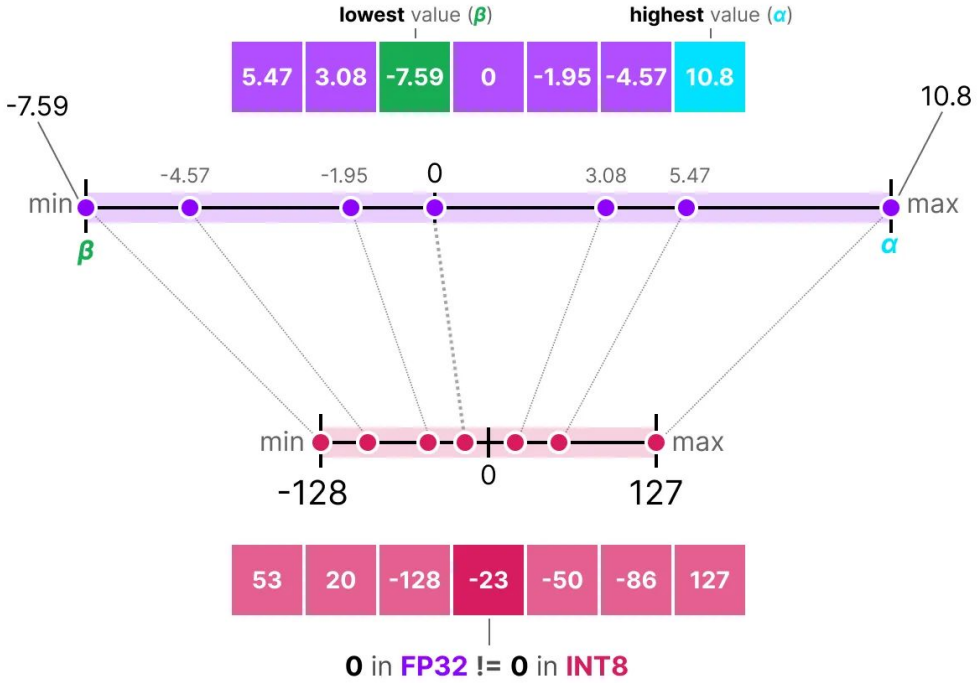
Quantization Error



Generally, the lower the number of bits, the more quantization error we tend to have.

Asymmetric Quantization

Maps the minimum (β) and maximum (α) values from the float range to the minimum and maximum values of the quantized range.



Notice how the 0 has shifted positions?
That's why it's called asymmetric quantization.

The min/max values have different distances to 0 in the range [-7.59, 10.8].

Asymmetric Quantization: *zero-point*

Due to its shifted position, we have to calculate the zero-point for the INT8 range to perform the linear mapping. As before, we also have to calculate a scale factor (s) but use the difference of INT8's range instead [-128, 127]

$$s = \frac{128 - -127}{\alpha - \beta} \quad \text{(scale factor)}$$

$$z = \text{round}\left(-s \cdot \beta\right) - 2^{b-1} \quad \text{(zeropoint)}$$

$$x_{quantized} = \text{round}\left(s \cdot x + z\right) \quad \text{(quantization)}$$

Asymmetric Quantization: *zero-point*

We need to calculate the zeropoint (z) in the INT8 range to shift the weights

$$s = \frac{255}{10.8 - -7.59} = 13.86 \quad \text{(scale factor)}$$

$$z = \text{round}\left(-13.86 \cdot -7.59\right) - 128 = -23 \quad \text{(zeropoint)}$$

$$x_{quantized} = \text{round}\left(13.86 \cdot \blacksquare\blacksquare\blacksquare\blacksquare\blacksquare + -23\right) \quad \text{(quantization)}$$

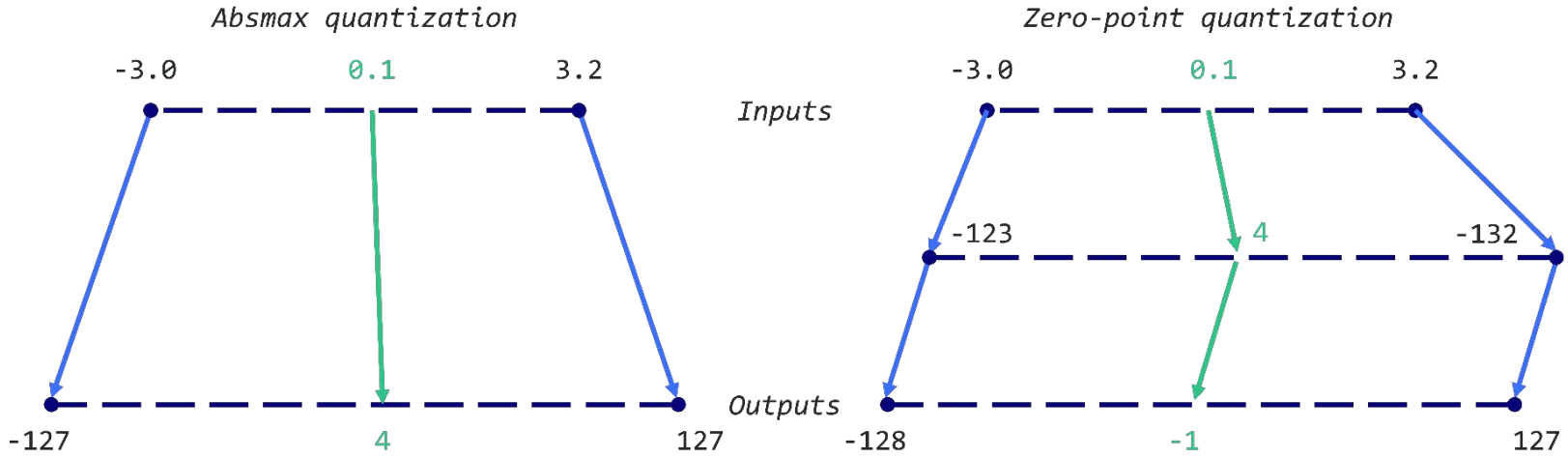
Asymmetric Quantization: *zero-point*

To dequantize the quantized from INT8 back to FP32, we will need to use the previously calculated scale factor (s) and zeropoint (z).

$$X_{dequantized} = \frac{\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare\blacksquare - Z}{S} \qquad \text{(dequantize)}$$

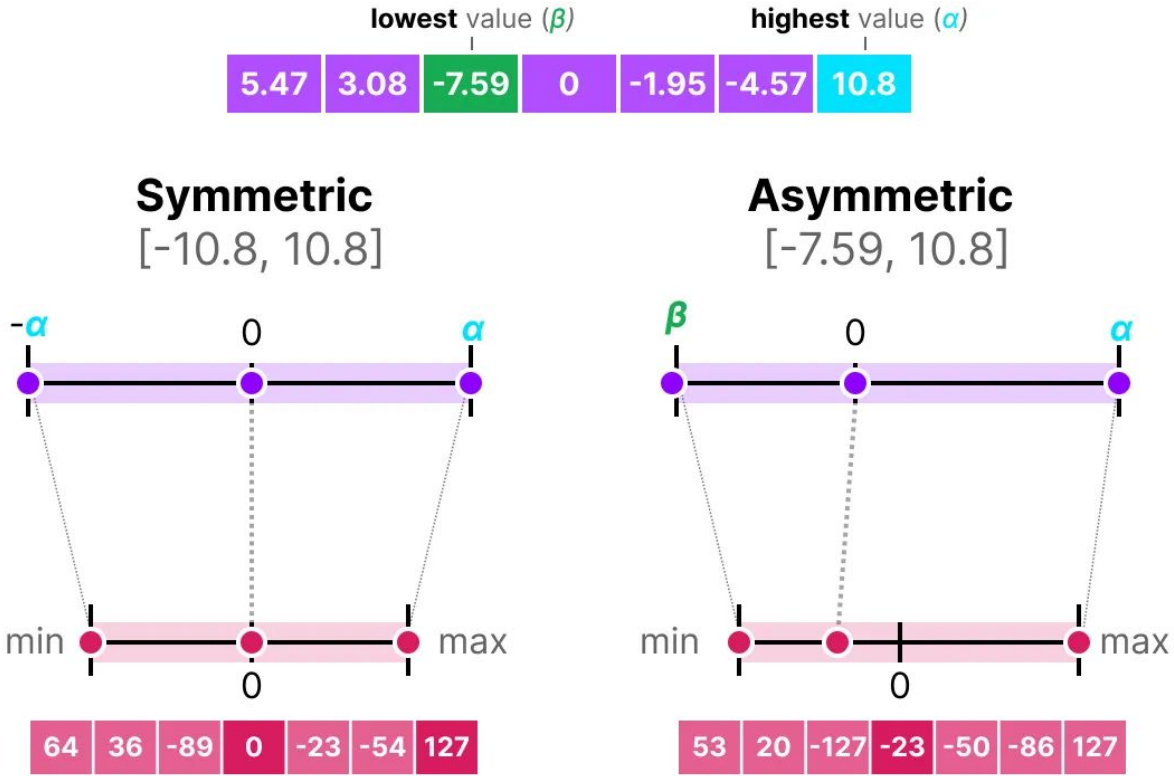
Symmetric vs Asymmetric

For example, we want to quantize 0.1.
We have a maximum value of 3.2 and a minimum value of -3.0 .



We have an absolution maximum value of 3.2. A weight of 0.1 would be quantized to $round(0.1 \times 127/3.2) = 4$

The scale is $255/(3.2 + 3.0) = 41.13$ and the zero-point $-round(41.13 \times -3.0) - 128 = 123 - 128 = -5$. A weight of 0.1 would be quantized to $round(41.13 \times 0.1 - 5) = -1$

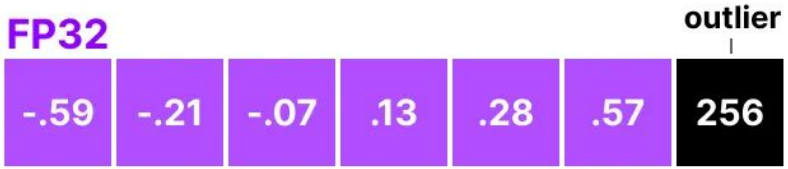


Note the zero-centered nature of symmetric quantization versus the offset of asymmetric quantization.

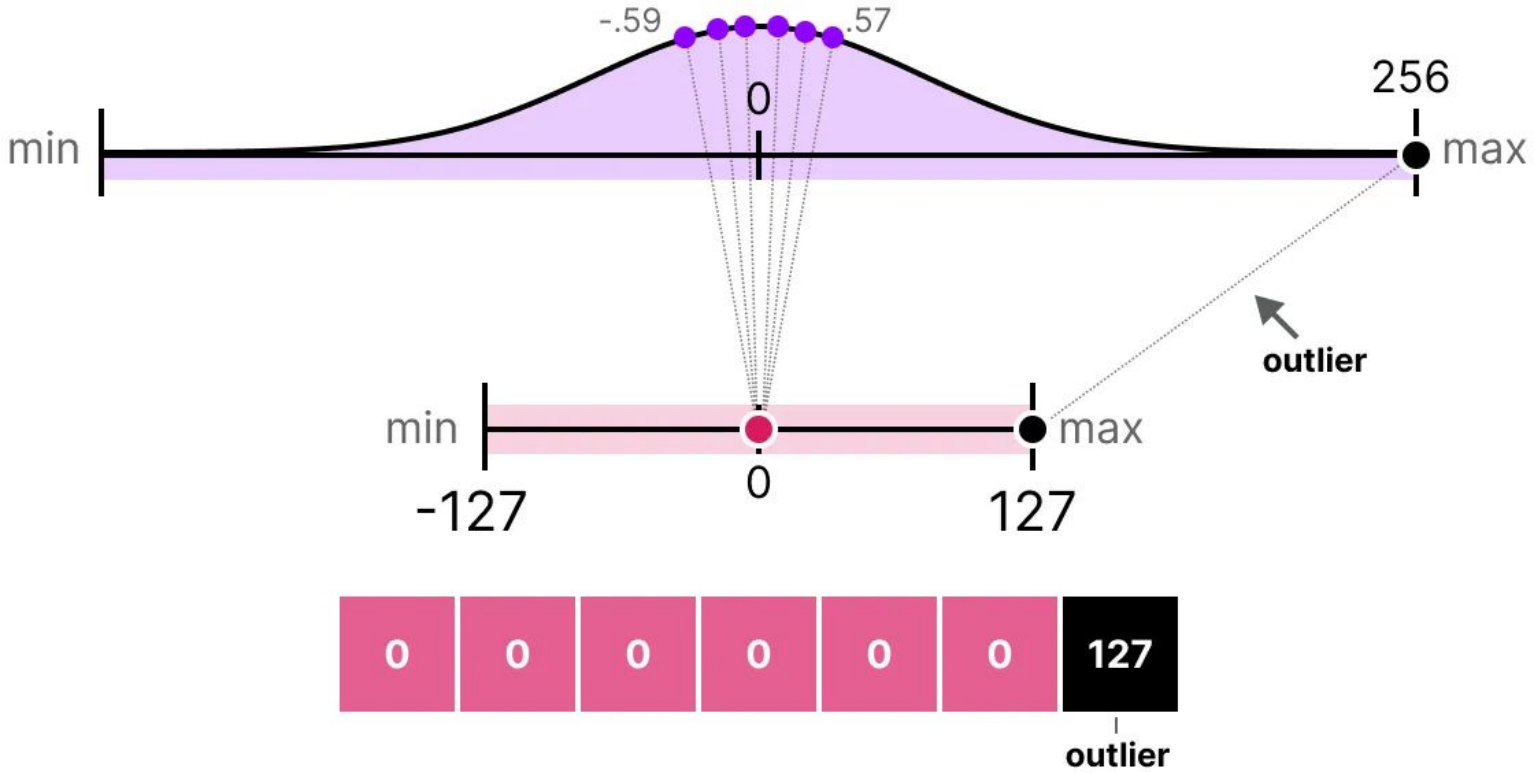
04

Range Mapping and Clipping

Imagine that you have a vector with the following values

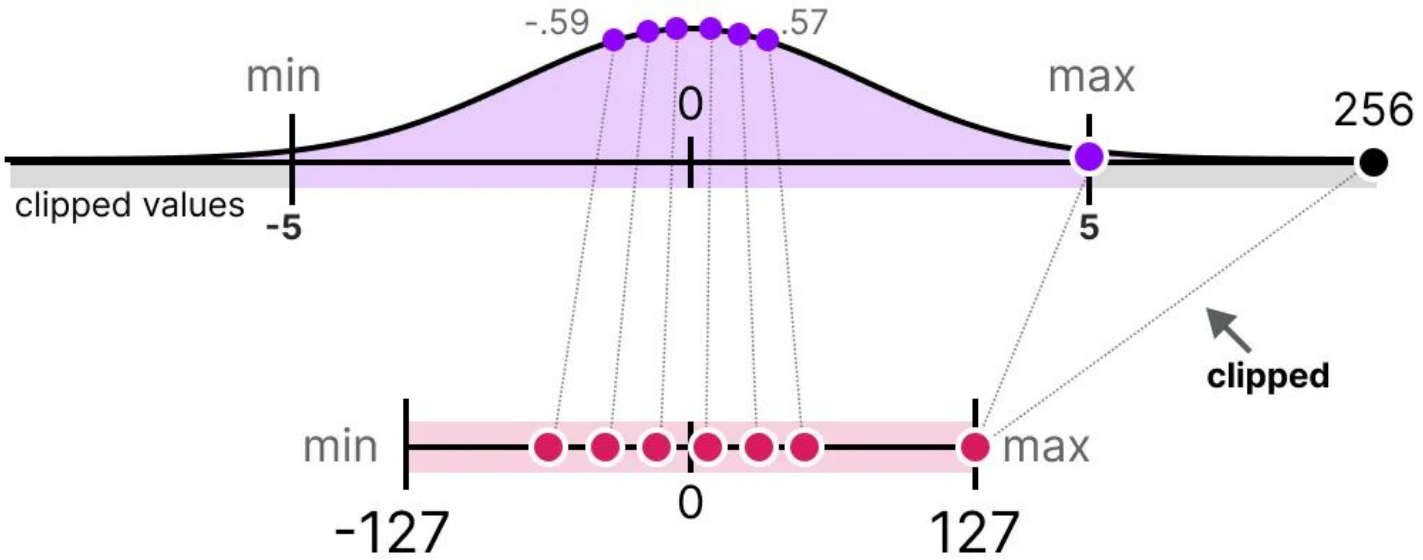


If we were to map the full range of this vector, all small values would get mapped to the same lower-bit representation and lose their differentiating factor:



Clipping

Clipping involves setting a different dynamic range of the original values such that all outliers get the same value.



We set the dynamic range to [-5, 5] all values outside that will either be mapped to -127 or to 127 regardless of their value

The major advantage is that the quantization error of the non-outliers is reduced significantly. However, the quantization error of outliers increases.

04

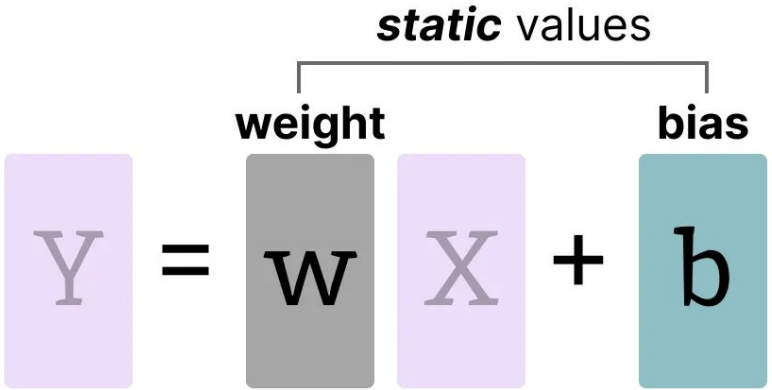
Quantizing the Model

Weights (and Biases)

Since there are significantly fewer biases (millions) than weights (billions), the biases are often kept in higher precision (such as INT16), and the main effort of quantization is put towards the weights.

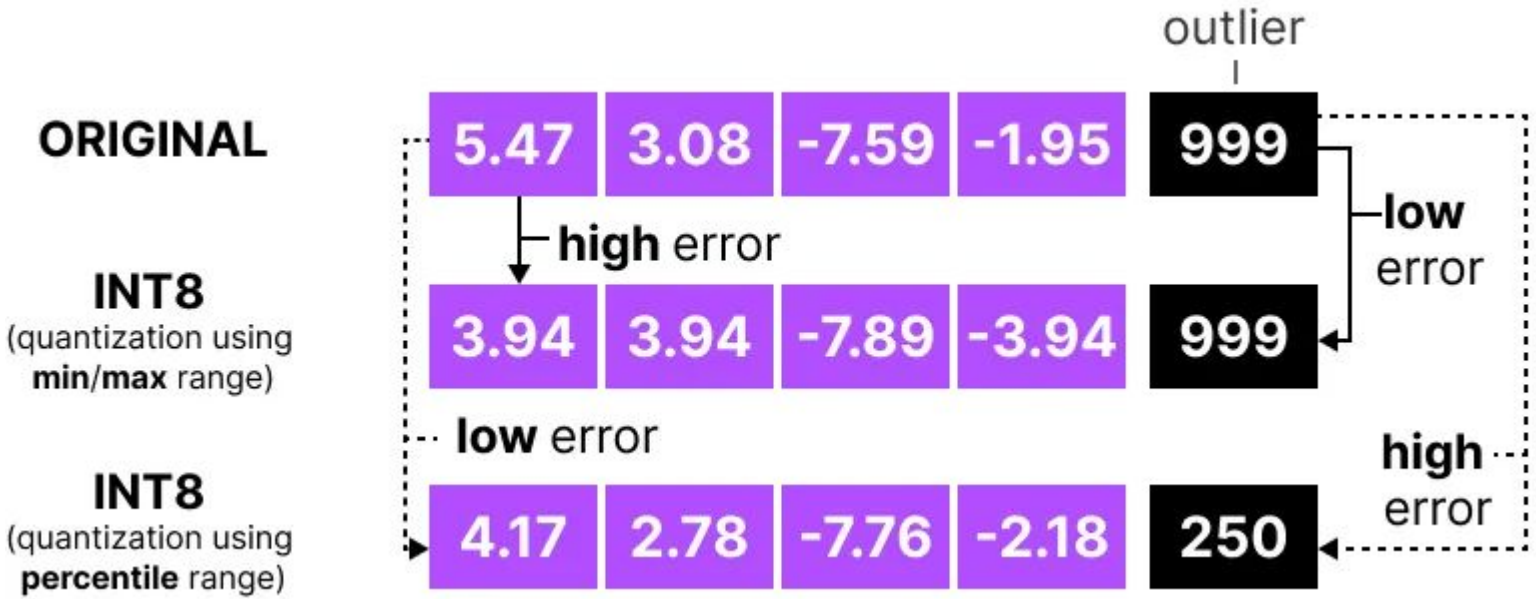
For weights, which are static and known, calibration techniques for choosing the range include:

- Manually choosing a *percentile* of the input range
- Optimize the *mean squared error* (MSE) between the original and quantized weights.
- Minimizing *entropy* (KL-divergence) between the original and quantized values



Quantization using Percentile Range

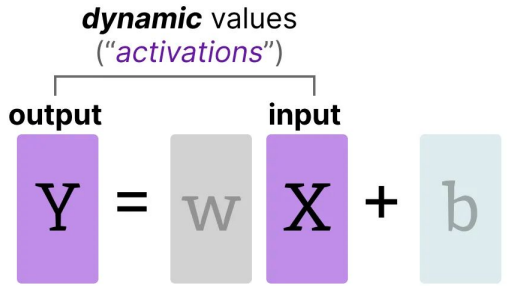
Choosing a percentile, for instance, would lead to similar clipping behavior as we have seen before.

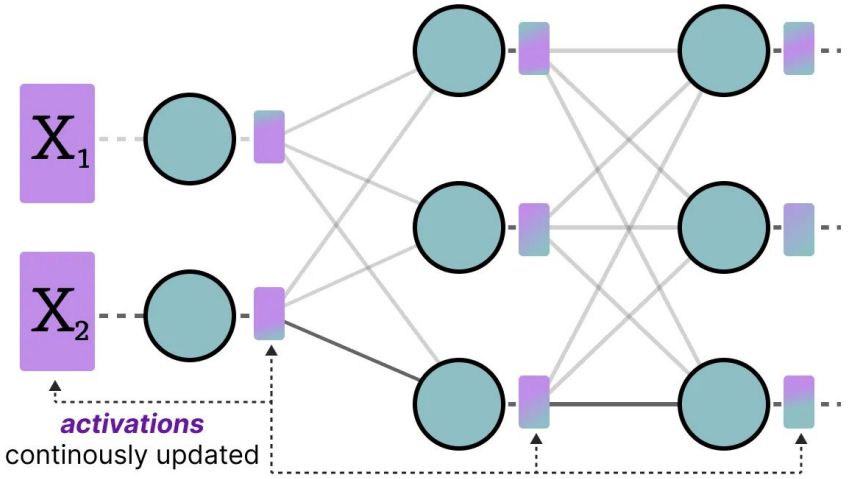


Activations

Unlike weights, activations vary with each input data fed into the model during inference, making it challenging to quantize them accurately.

Since these values are updated after each hidden layer, we only know what they will be during inference as the input data passes through the model





Broadly, there are two methods for calibrating the quantization method of the weights and activations:

- Post-Training Quantization (PTQ)
    - Quantization ***after*** training
- Quantization Aware Training (QAT)
    - Quantization ***during*** training/fine-tuning

Post-training quantization (PTQ) is more popular. It involves quantizing a model's parameters (both weights and activations) **after** training the model.

Quantization of the *weights* is performed using either symmetric or asymmetric quantization.

Quantization of the *activations*, however, requires inference of the model to get their potential distribution since we do not know their range.

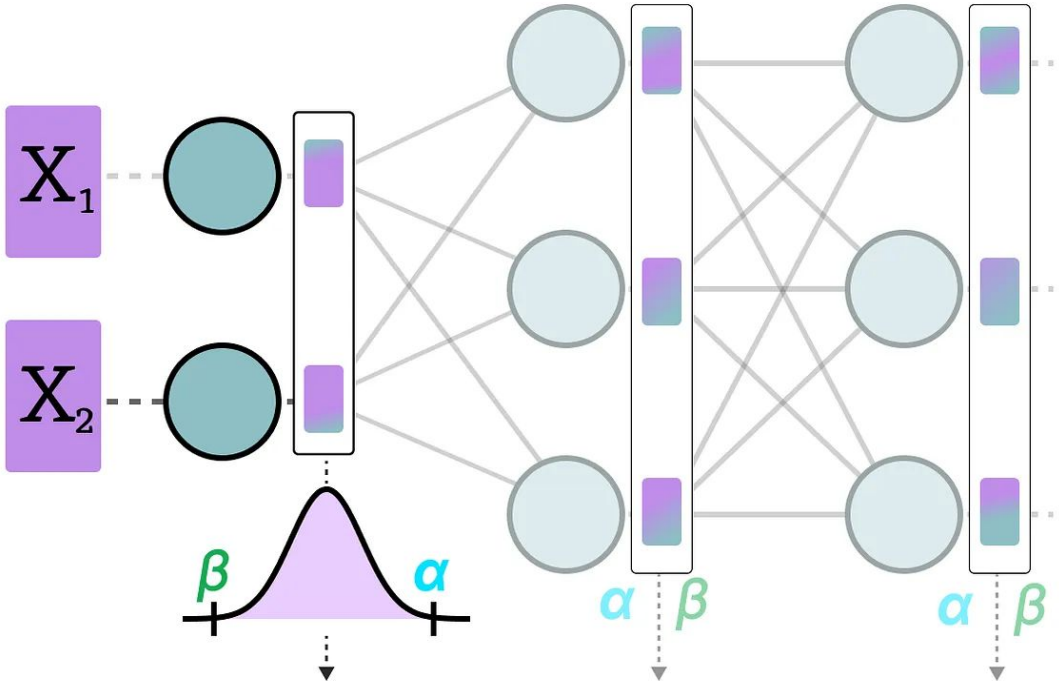
06

Post-Training Quantization

Dynamic Quantization
Static Quantization

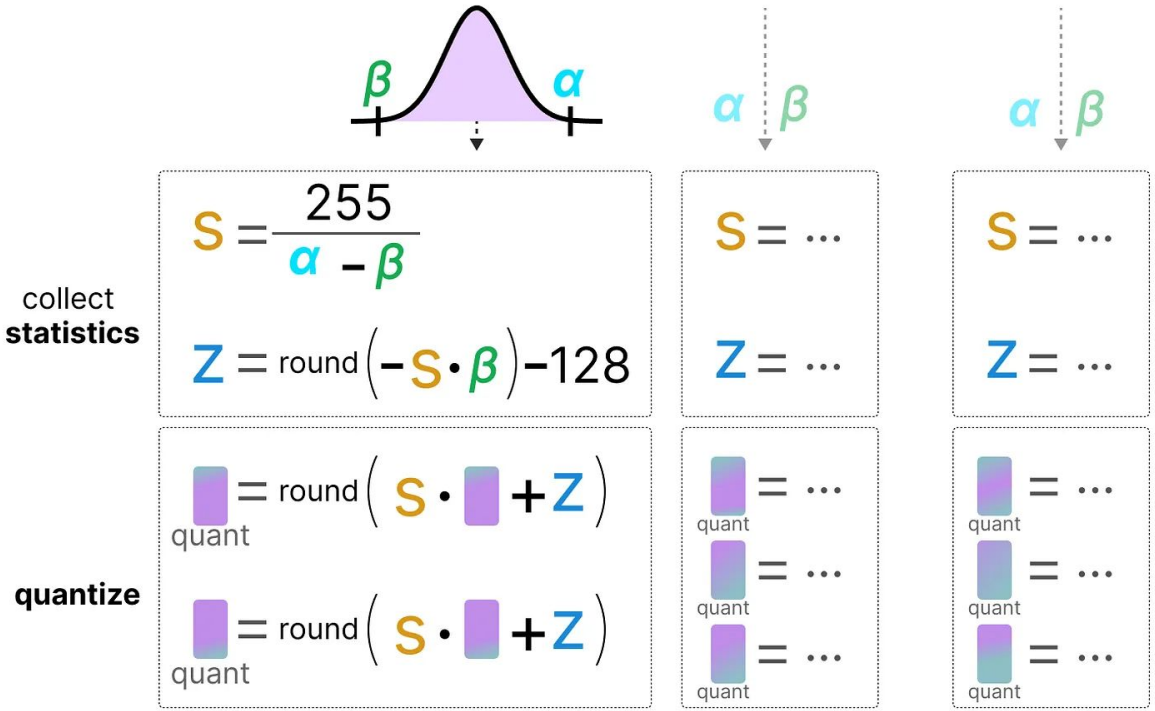
Dynamic Quantization

After data passes a hidden layer, its activations are collected



Dynamic Quantization

This distribution of activations is then used to calculate the zero point (z) and scale factor (s) values needed to quantize the output:

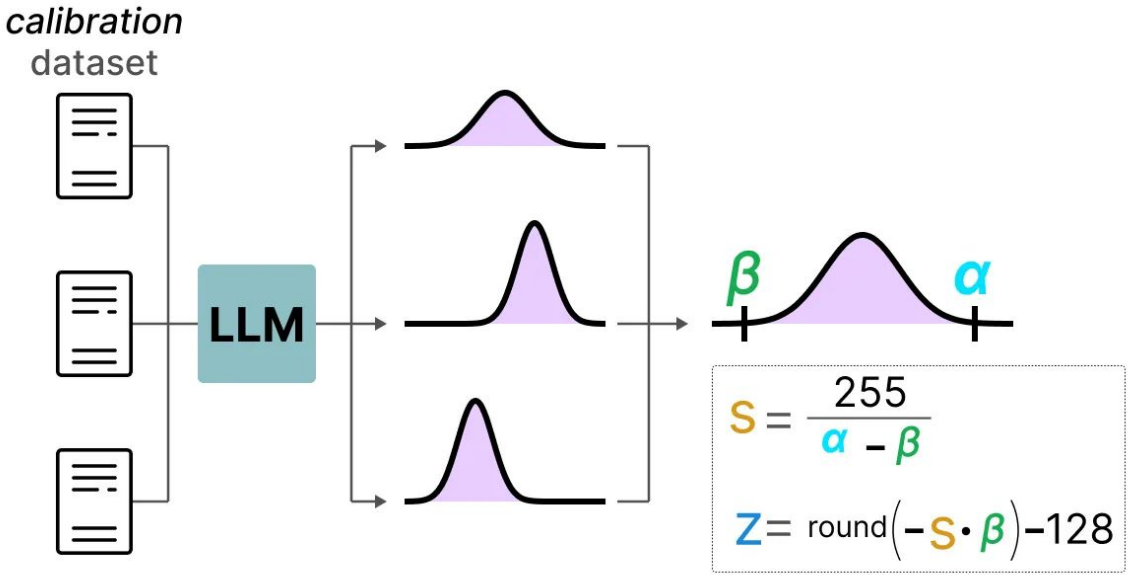


The process is repeated each time data passes through a new layer. Therefore, each layer has its own separate ***z*** and ***s*** values and therefore different quantization schemes.

Static Quantization

Static quantization calculates the zeropoint (z) and scale factor (s) beforehand (instead of during inference) with the help of a calibration dataset.

The calibration dataset is used by the model to collect these potential distributions of the activations



When you are performing actual inference, the s and z values are not recalculated but are used globally over all activations to quantize them

Dynamic Quantization

How it Works:

- Model weights are converted to INT8 beforehand.
- Activations (the outputs of each layer) are converted to INT8 during inference, as they are generated.

Key Characteristic: The scaling factors for activations are calculated dynamically for each input.

✅ Robust: Less sensitive to unexpected inputs since it adapts on the fly.

❌ Performance Overhead: Calculating scaling factors at runtime adds a small amount of latency.

Static Quantization

How it Works:

- The model is "calibrated" by feeding it a small, representative dataset - O(K) samples).
- The range (min/max values) of both weights and activations is observed.
- Fixed scaling factors are calculated and "fused" into the model.

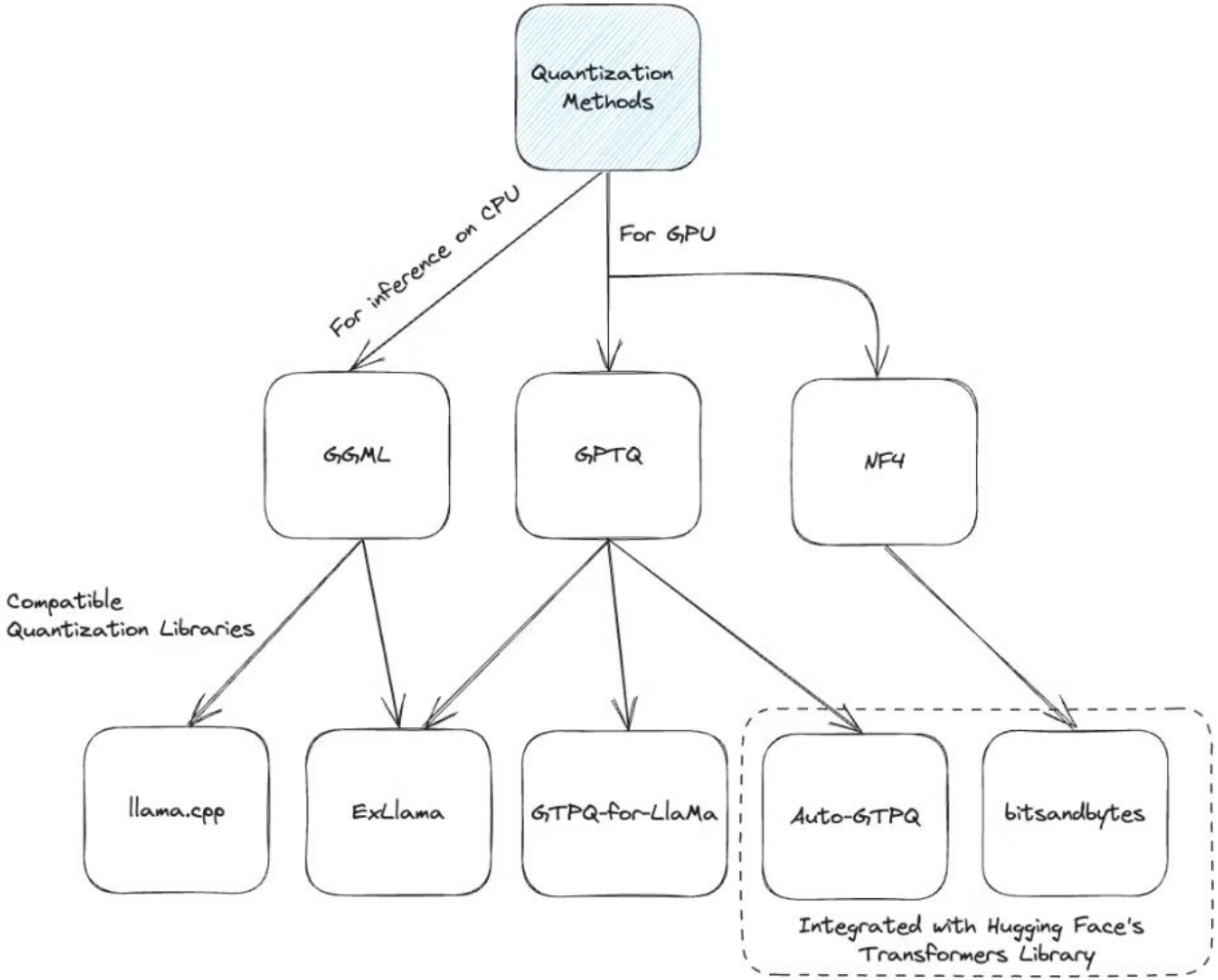
Key Characteristic: All quantization parameters are pre-computed and fixed.

✅ Maximum Speed: No runtime calculations needed.

❌ Potentially Fragile: Accuracy can drop if real-world data differs significantly from calibration data.

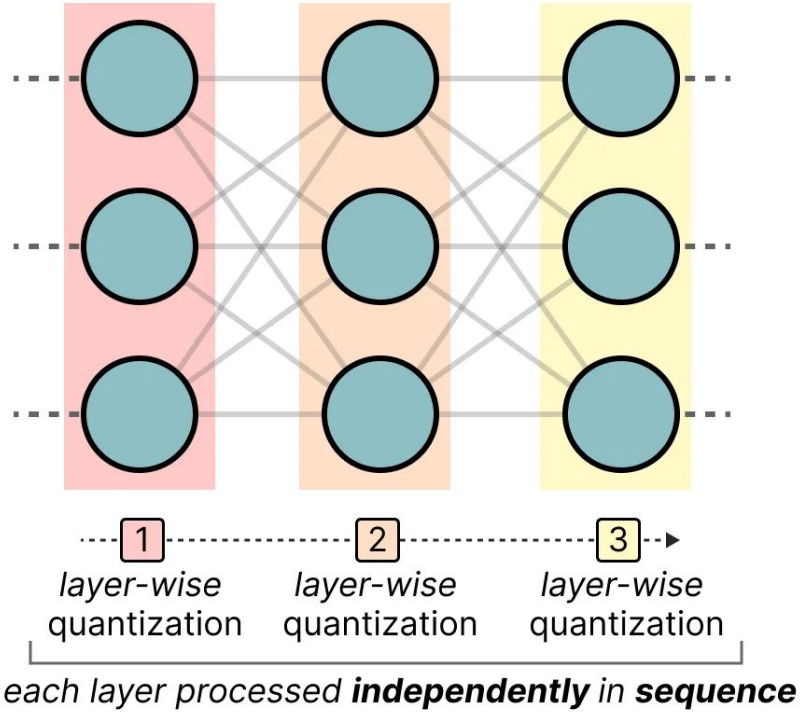
07

Post-Training Static Quantization Method: GPTQ



GPT-Quantization (GPT-Q) Overview

- Quantization to 4-bits
- Asymmetric quantization
- Quantizes layer by layer such that each layer is processed independently before continuing to the next.



“

Imagine you are in one layer with thousands of unquantized weights.

Step A: It picks one weight to quantize.
Step B: It quantizes that weight (e.g., it changes 0.831 to the nearest 4-bit value, let's say 0.8). This introduces a small error (an error of 0.031).
Step C (The Crucial Part): It immediately updates all the other unquantized weights in that same layer to compensate for the error it just introduced. It's like saying, "Okay, I had to round this one number down, so I'll nudge all the other numbers up a tiny bit to balance things out."

“

How does it know how to compensate?

It uses something called the Hessian matrix (or an approximation of it).

You don't need to understand the math, just the concept: The Hessian tells the algorithm how sensitive the model's output is to changes in each weight.

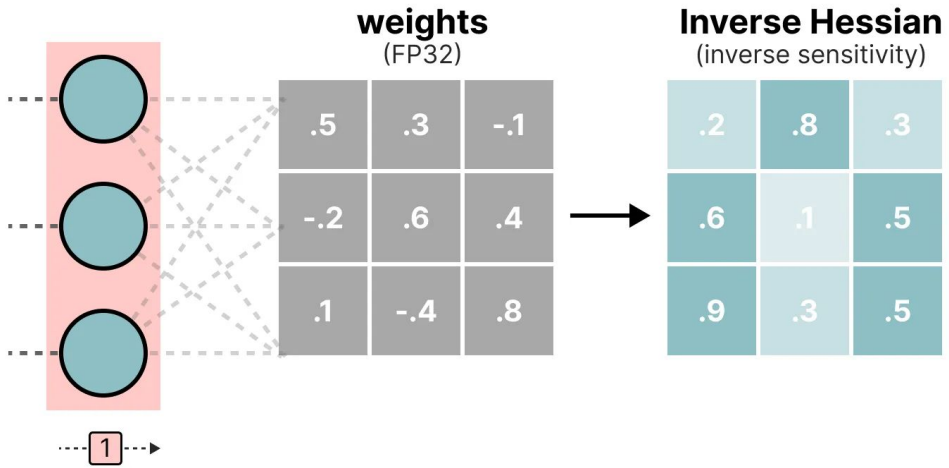
In other words, it identifies the most important weights.
By using this information, GPT-Q makes sure that the compensation process minimizes the overall damage to the layer's output. It protects the integrity of the most critical parts of the model while quantizing.

GPT-Quantization (GPT-Q) Method

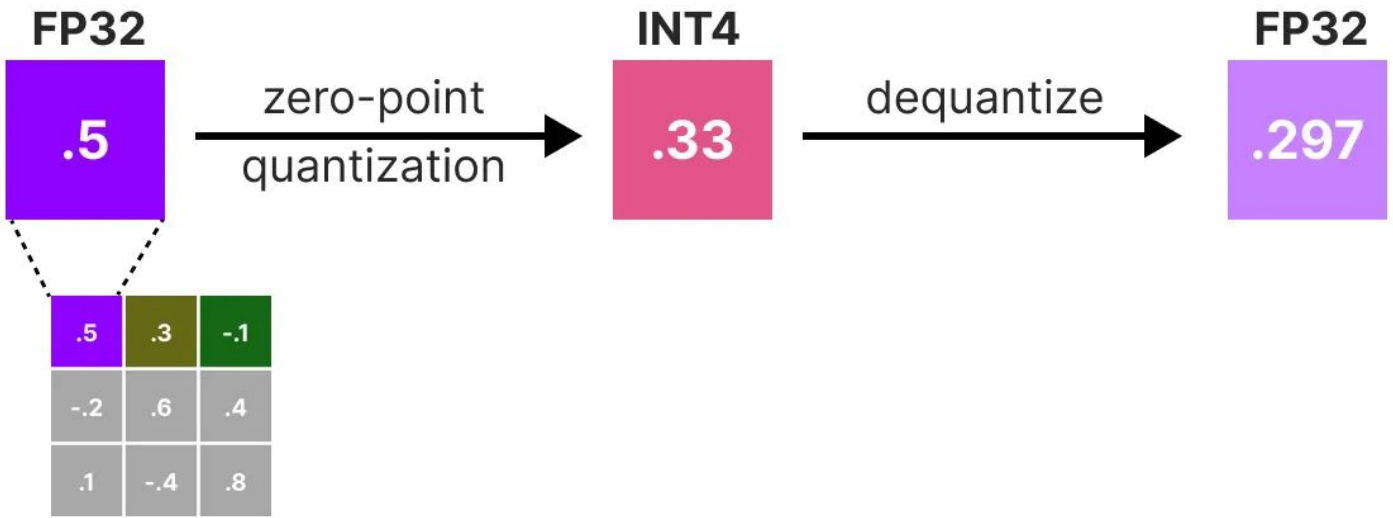
During this layer-wise quantization process, it first converts the layer's weights into the inverse-**Hessian**. It is a second-order derivative of the model's loss function and tells us how sensitive the model's output is to changes in each weight.

It essentially demonstrates the (*inverse*) **importance of each weight** in a layer.

Weights associated with smaller values in the Hessian matrix are more crucial because small changes in these weights can lead to significant changes in the model's performance.



Next, we quantize and then dequantize the weight of the first row in our weight matrix



This process allows us to calculate the quantization error (q)

The quantization error (q) is weighed using the inverse-Hessian (h_1) that we calculated beforehand.

Essentially, we are creating a weighted-quantization error based on the importance of the weight.

We redistribute this weighted quantization error over the other weights in the row. This allows for maintaining the overall function and output of the network.

$$q = \frac{x_1 - x_1}{h_1}$$

(hessian-weighted quantization error)

$$q = \frac{.5 - .297}{.2} = .203$$

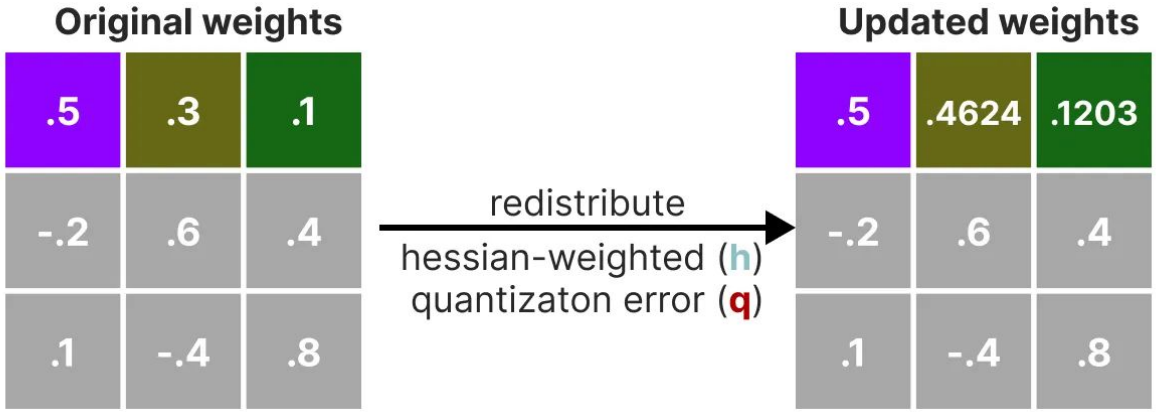
For example, if we were redistribute this weighted quantization error over the second weight, namely .3 (x_2), we would add the quantization error (q) multiplied by the inverse-Hessian of the second weight (h_2)

$$x_2 = x_2 + q \cdot h_2$$

(update weight)

$$x_2 = .3 + .203 \cdot .8$$

We can do the same process over the third weight in the given row



We iterate over this process of redistributing the weighted quantization error until all values are quantized.

This works so well because weights are typically related to one another. So when one weight has a quantization error, related weights are updated accordingly (through the inverse-Hessian).

GPTQ in Practice

The GPTQ quantization technique can be applied to many models to transform them into 3, 4 or 8-bit representations in a few simple steps.

The most commonly used library for quantization related to GPTQ is AutoGPTQ due to its integration with the transformers library.

An example of how to quantize and use already quantized models with AutoGPTQ is shown on the right:

```
from transformers import AutoModelForCausalLM,
AutoTokenizer, GPTQConfig

model_id = "facebook/opt-125m"

tokenizer = AutoTokenizer.from_pretrained(model_id)

quantization_config = GPTQConfig(
            bits=4,
            dataset = "c4",
            tokenizer=tokenizer
      )

model = AutoModelForCausalLM.from_pretrained(
            model_id,
            device_map="auto",
            quantization_config=quantization_config
      )
```

GPTQ Benefits

GPTQ can only quantize models into INT-based data types, being most commonly used to convert to 4INT.

1. **Scalability**: GPTQ has the capacity to compress large networks such as the GPT models with 175 billion parameters in about 4 GPU hours, cutting the bit width down to 3 or 4 bits per weight with very minimal degradation in accuracy
2. **Performance**: This technique makes it feasible to run inference on a 175 billion-parameter model using a single GPU.
3. **Inference Speed**: GPTQ models offer 3.25x speed-ups on high-end GPUs like NVIDIA A100 and a 4.5x speed increase on cost-effective ones like NVIDIA A6000, compared to FP16 models.

07

Quantization File Formats

GGML File Format

- ➔ GGML is a C library for machine learning (ML), where the "GG" refers to the initials of its originator (Georgi Gerganov).
- ➔ The is the file format behind llama.cpp that brought large models to your laptop.
- ➔ A noteworthy progression is the transition from the GGML format to GGUF, which supports the use of non-llama models.

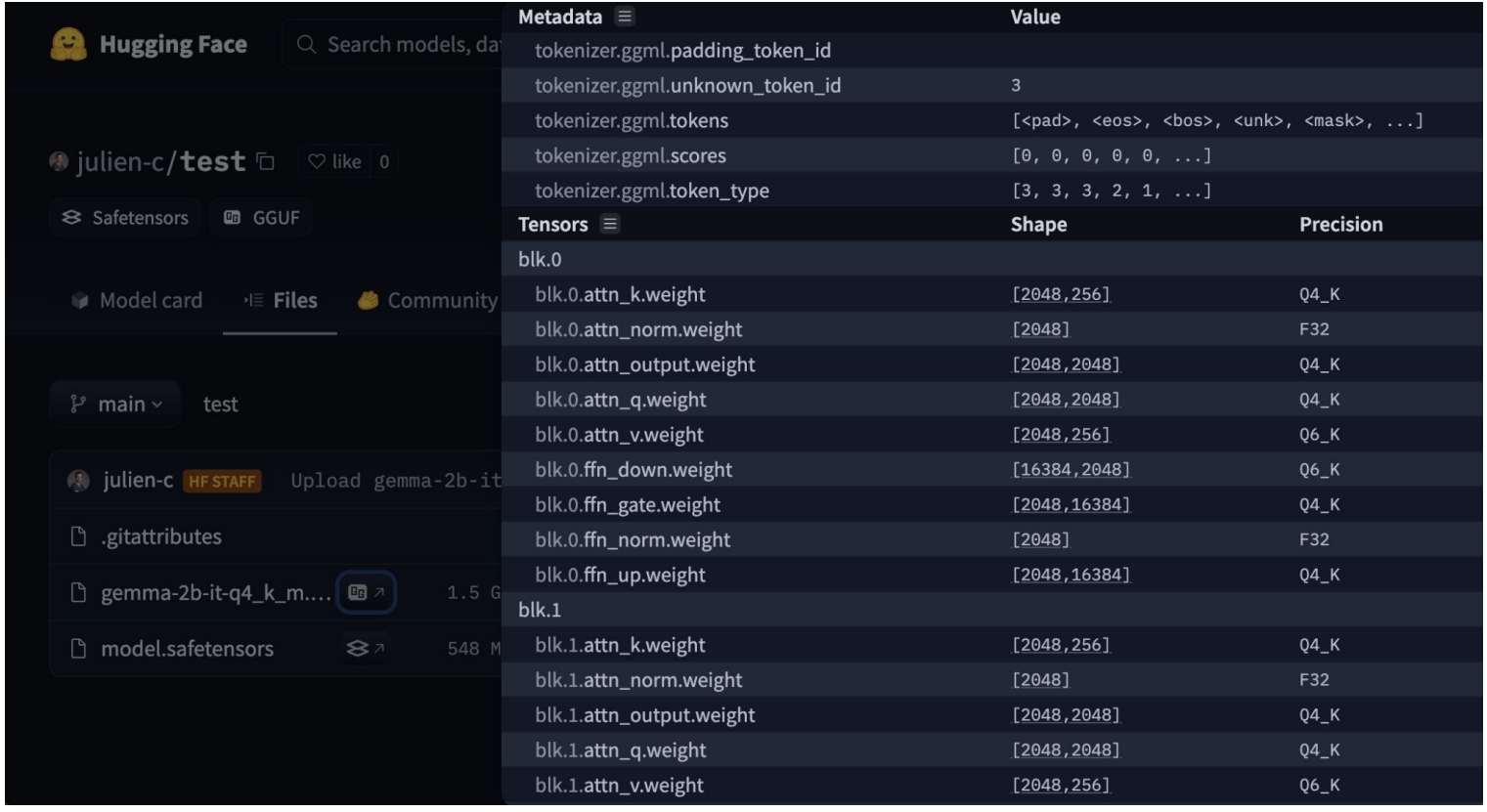
What is GGUF?
It's a container that bundles model weights, architecture info, and per-layer quantization data into one portable file.

Key Idea: Not all layers are equal. Some are more sensitive to quantization than others.

As we can see in this graph, GGUF encodes both the tensors and a standardized set of metadata.

Mixed Precision

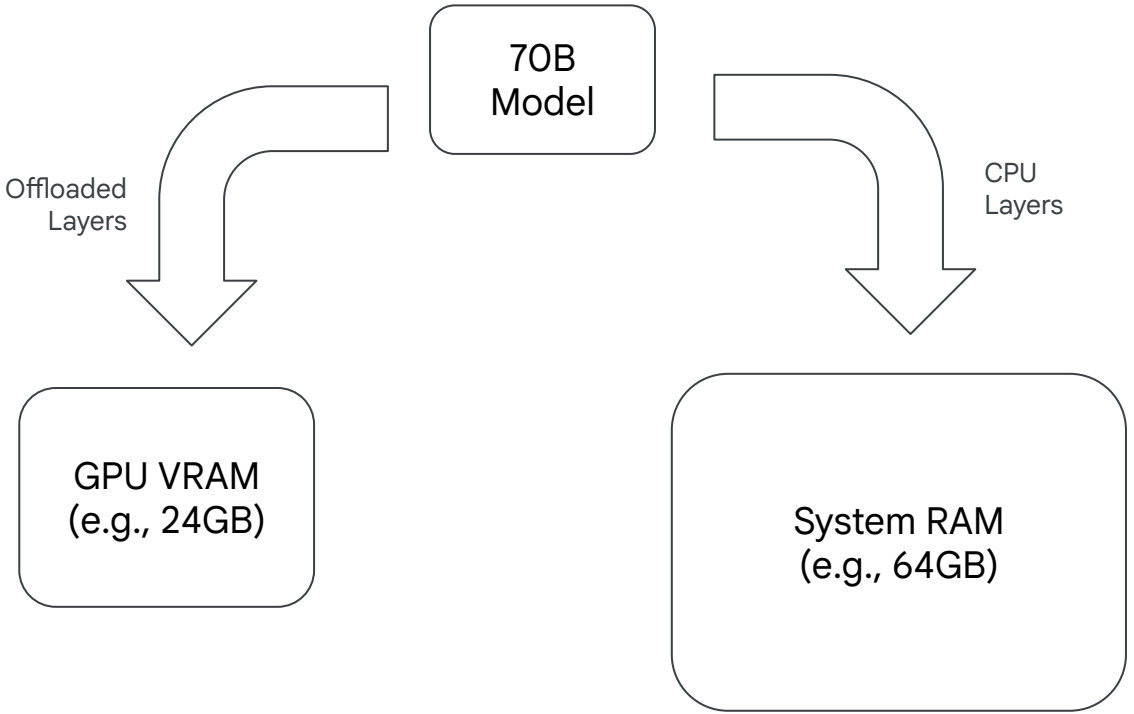
Allows different layers to use different bit-depths (e.g., FP16 for sensitive layers, INT4 for others). This is a smart accuracy-vs-size trade-off.



Metadata	Value	
tokenizer.ggml.padding_token_id		
tokenizer.ggml.unknown_token_id	3	
tokenizer.ggml.tokens	[<pad>, <eos>, <bos>, <unk>, <mask>, ...]	
tokenizer.ggml.scores	[0, 0, 0, 0, 0, ...]	
tokenizer.ggml.token_type	[3, 3, 3, 2, 1, ...]	
Tensors	**Shape**	**Precision**
blk.0		
blk.0.attn_k.weight	[2048,256]	Q4_K
blk.0.attn_norm.weight	[2048]	F32
blk.0.attn_output.weight	[2048,2048]	Q4_K
blk.0.attn_q.weight	[2048,2048]	Q4_K
blk.0.attn_v.weight	[2048,256]	Q6_K
blk.0.ffn_down.weight	[16384,2048]	Q6_K
blk.0.ffn_gate.weight	[2048,16384]	Q4_K
blk.0.ffn_norm.weight	[2048]	F32
blk.0.ffn_up.weight	[2048,16384]	Q4_K
blk.1		
blk.1.attn_k.weight	[2048,256]	Q4_K
blk.1.attn_norm.weight	[2048]	F32
blk.1.attn_output.weight	[2048,2048]	Q4_K
blk.1.attn_q.weight	[2048,2048]	Q4_K
blk.1.attn_v.weight	[2048,256]	Q6_K

Offloading to CPU

The file format contains all the metadata needed for an engine like `llama.cpp` to intelligently split the model between CPU and any available GPU VRAM



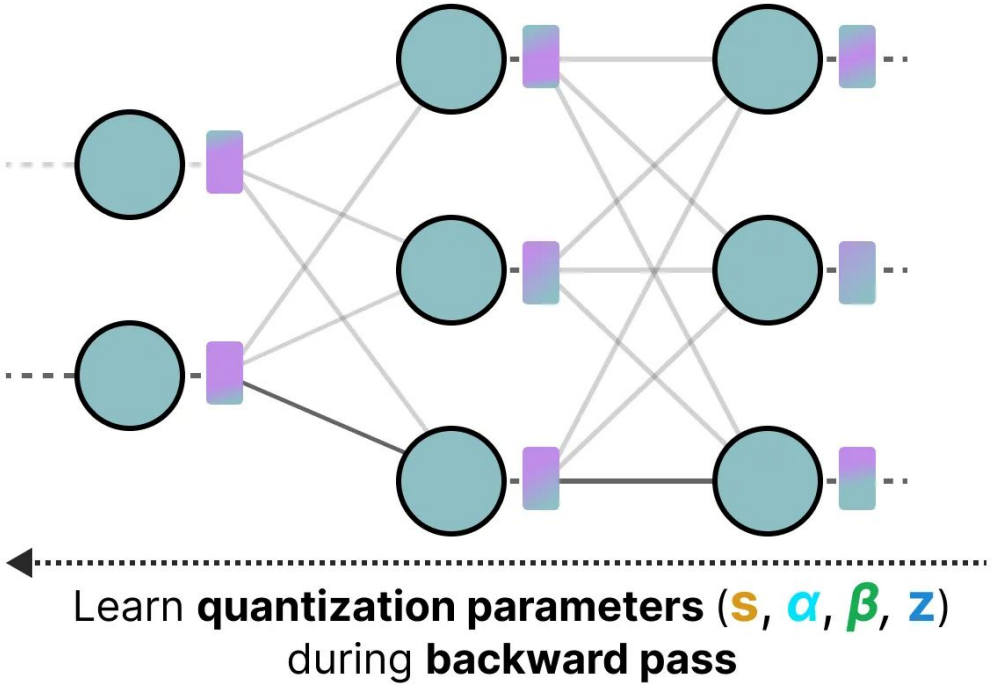
The first chunk of the model block (e.g., 25 layers) is placed inside the GPU VRAM box

The rest of the model block (e.g., 55 layers) is placed inside the System RAM box.

08 Quantization Aware Training

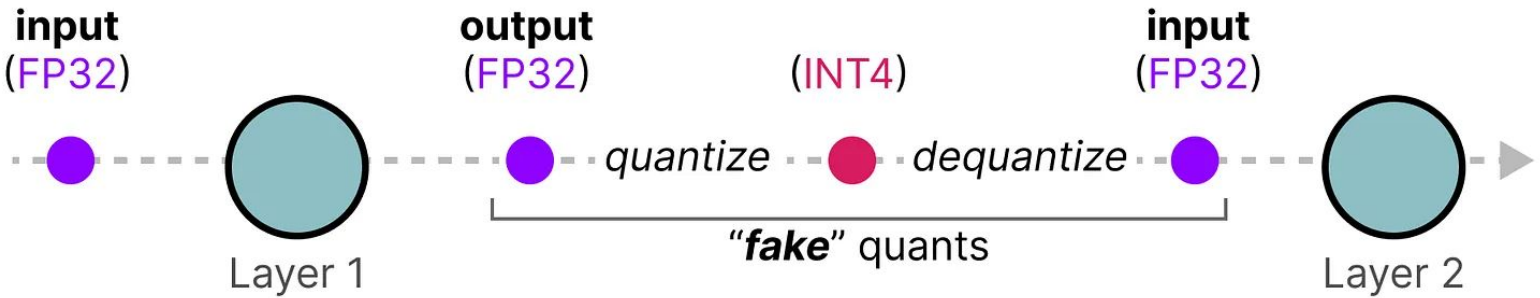
Overview

Instead of quantizing a model ***after*** it was trained with post-training quantization (PTQ), QAT aims to learn the quantization procedure ***during*** training.



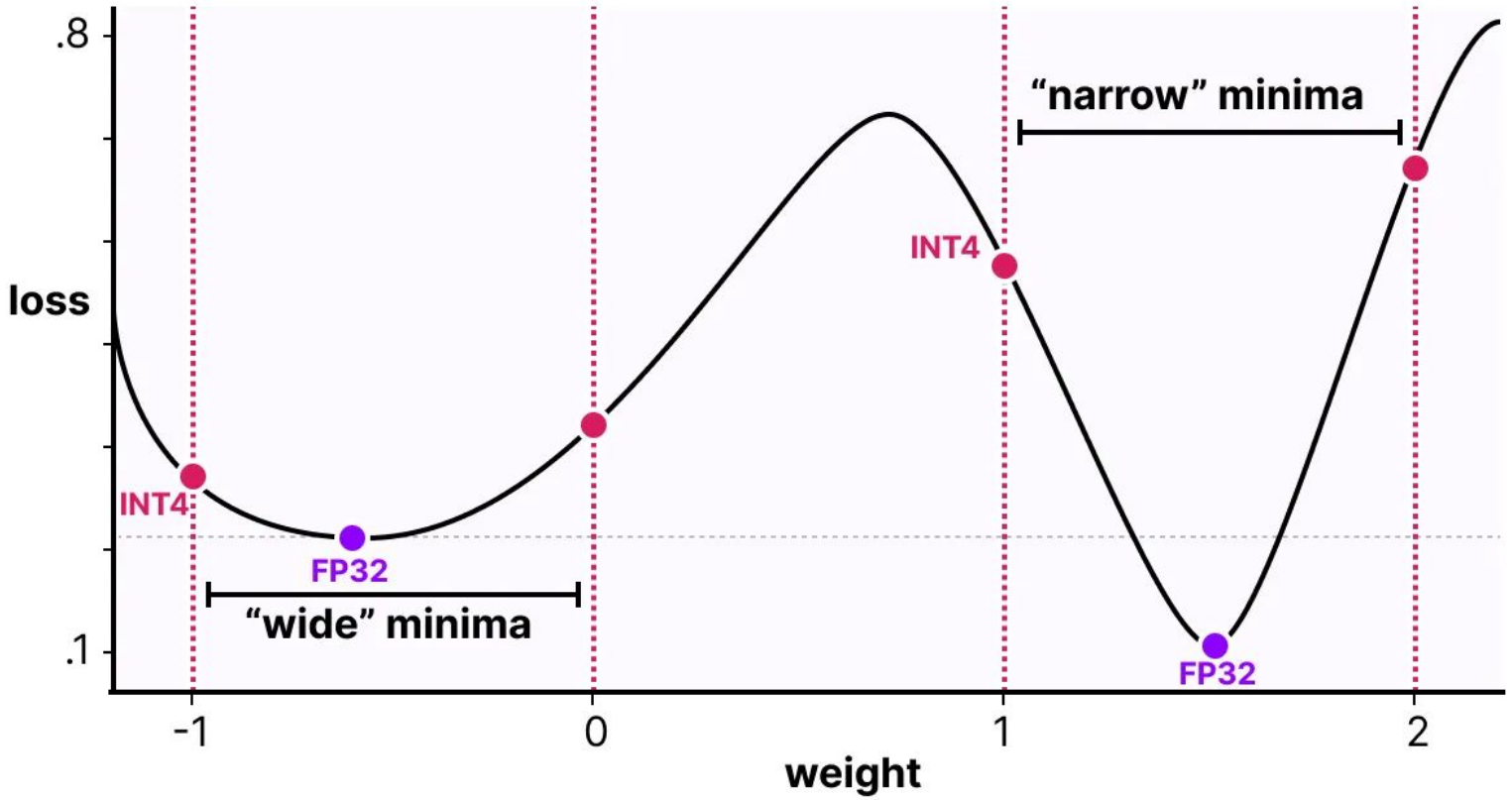
Method

During training, so-called "*fake*" quants are introduced. This is the process of first quantizing the weights to, for example, INT4 and then dequantizing back to FP32:



This process allows the model to consider the quantization process during training, the calculation of loss, and weight updates.

QAT attempts to explore the loss landscape for “wide” minima to minimize the quantization errors as “narrow” minima tend to result in larger quantization errors.



Google DeepMind

Practical: Weight Quantization

Colab Notebook

tme: 5 mins

Model Pruning

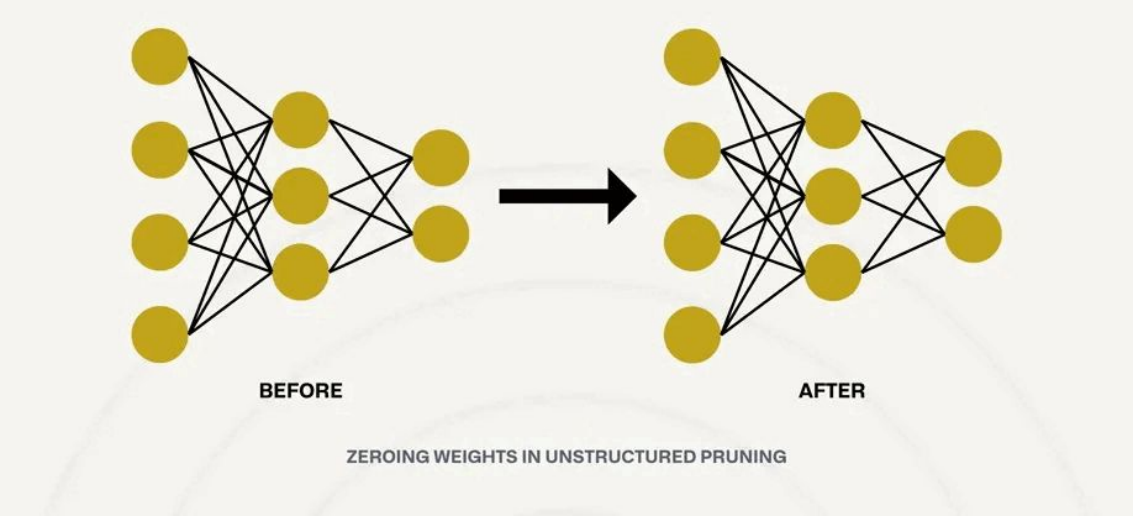
01 Introduction

Overview

Model pruning refers to the act of removing unimportant parameters from a deep learning neural network model.

Generally, only the weights of the parameters are pruned, leaving the biases untouched. The pruning of biases tends to have much more significant downsides.

As these parameters are being removed, there may be resultant degradation of the model's inference performance, hence it should be performed with care.

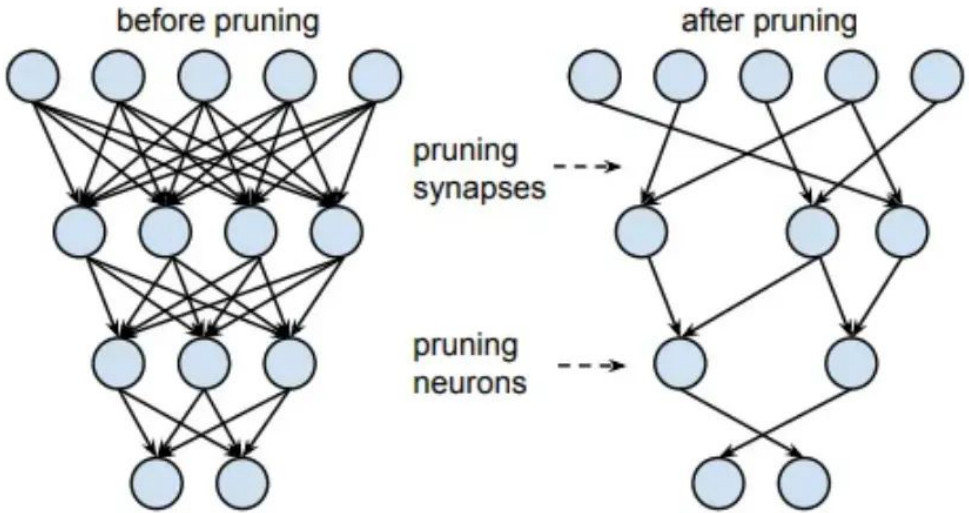


Why is it important?

Neural networks have an excess of parameters needed to generalize well and make accurate predictions.

"The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks" (Frankle and Carbin, 2019) demonstrates that neural networks tend to have a specific subset of parameters that are essential for prediction.

Pruning generally is more surgical in compressing than other methods such as quantization, which is just bluntly removing precision from model weights.

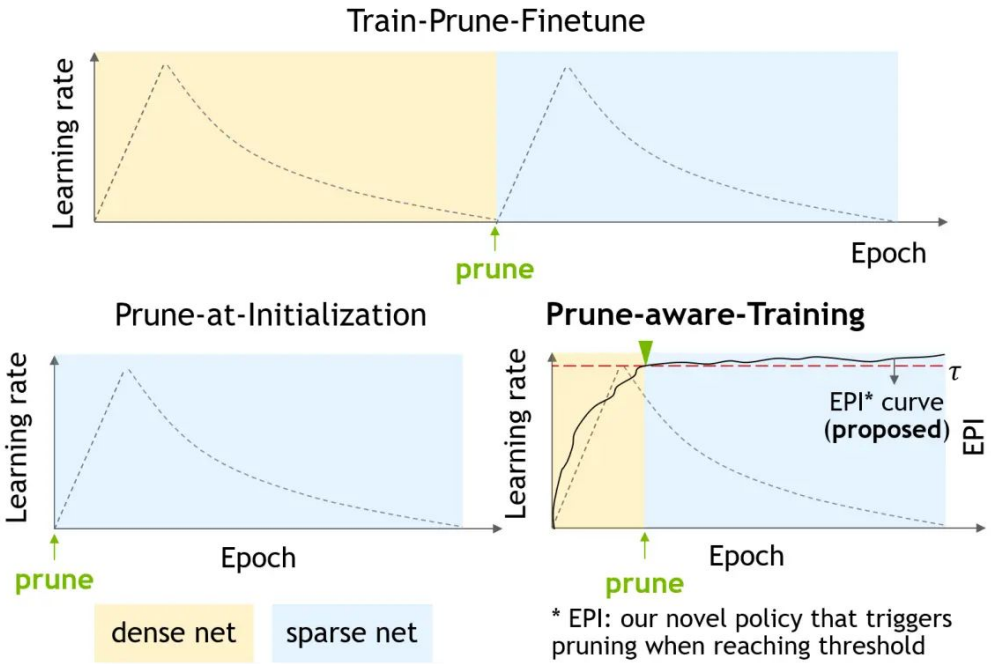


02

Approaches

Overview

Based on when the pruning decisions are made relative to the training process, there are two main approaches: train-time pruning and post-training pruning



A Comparison of Different Pruning Approaches. Source: When to Prune? A Policy towards Early Structural Pruning

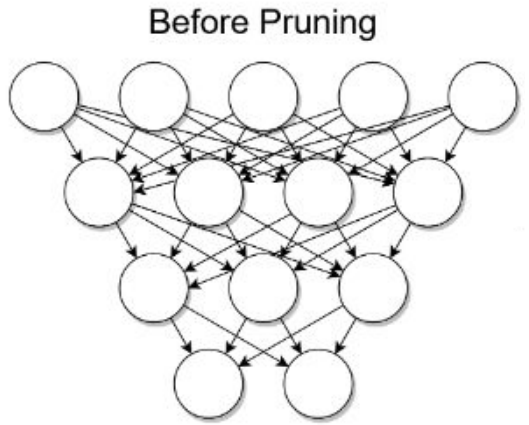
Post-training Tuning

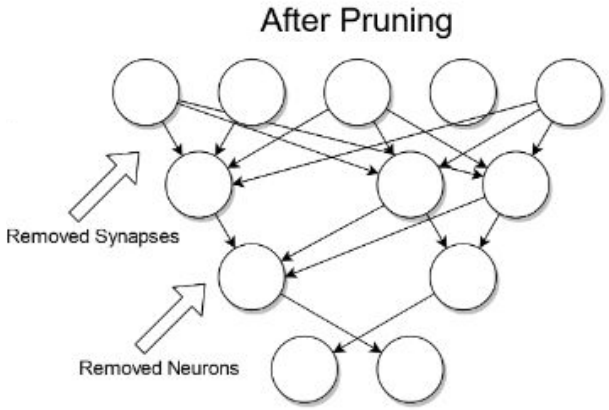
Train First, Prune Second

The Core Hypothesis
Weights with a small magnitude (close to zero) are less important to the model's output.

The Process:

1. Take your fully trained, dense model.
2. Define a target sparsity (e.g., "prune 50% of the weights").
3. Find the magnitude threshold that corresponds to this sparsity level.
4. Set all weights with a magnitude below this threshold to zero.
5. (Optional but Recommended): Briefly fine-tune the pruned model on the data to help it recover from the "shock" of losing weights.





Train-Time Pruning

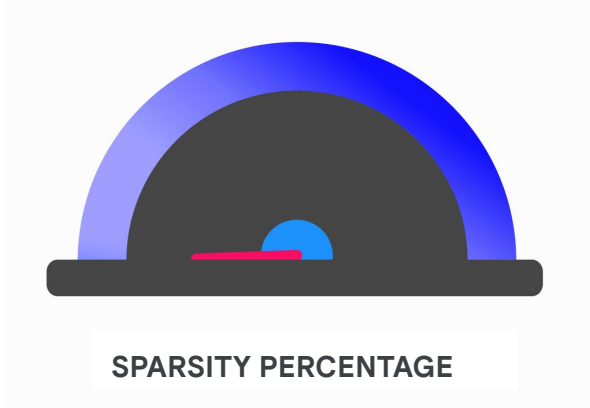
Learn while Pruning.

Sparsity via L1/L2 Regularization

We add a penalty to the loss function that is proportional to the magnitude of the weights (e.g., L1 Regularization).

```
Total Loss = Task Loss + λ * Σ|weight|
```

The model is incentivized to shrink less important weights to exactly zero to minimize the total loss



Gradual Magnitude Pruning (GMP)

Instead of one big prune at the end, sparsity is increased gradually over the course of training.

Example Schedule:

1. Start training at 0% sparsity.
2. Gradually increase to 50% sparsity over the first 40% of training.
3. Train at a fixed 50% sparsity for the remaining 60%.

PRUNING APPROACH	TRAIN-TIME PRUNING	POST-TRAINING PRUNING
PROS	More efficient models since they are trained with the objective of sparsity in mind	Simpler to implement
	Pruning decisions are made alongside the consideration of model parameter optimization	Pruning parameters can be easily adjusted based on inference requirements
CONS	Makes training process more complex	May require fine-tuning to regain performance in the case of accuracy degradation
	Change in pruning parameters may require retraining of the whole model	Pruning decisions are user-defined and may not be optimal

03

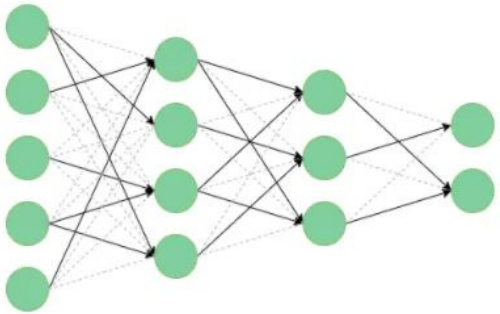
Post-Training Pruning Types

Unstructured Pruning

Remove individual, scattered weights anywhere in the model.

How it Works

1. Applies a simple threshold test to individual weights.
2. If abs(weight) < threshold, then weight = 0.
3. This creates a sparse matrix with zero-values scattered randomly.



✅ **Model Size Reduction:** The resulting sparse matrix can be compressed effectively, leading to a much smaller model file on disk.
✅ **Denoising**: Can act as a regularizer, potentially improving model robustness.
❌ **No Automatic Speed-up**
Why? Standard GPUs and CPUs are optimized for dense matrix multiplication. They don't have a way to "skip" individual zero-value calculations. The hardware still performs X * 0, Y + 0, etc. The full matrix operation is executed.
Analogy: A spreadsheet with some empty cells. You still have to look at every cell to see if it's empty. You can't just skip rows.

Structured Pruning

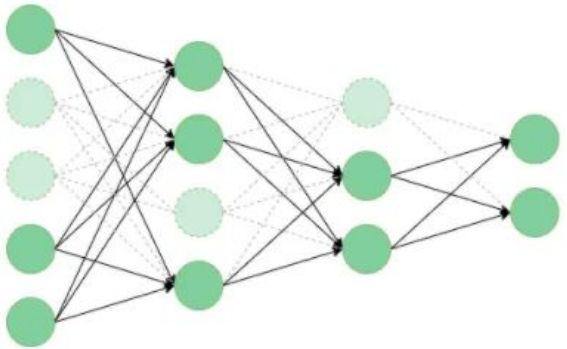
Remove entire, contiguous blocks of weights (neurons, channels, or attention heads).

How it Works

Instead of scoring individual weights, we score the importance of groups of weights.

- e.g., The sum of magnitudes of all weights in a channel.
- e.g., The average activation value of a neuron.

Entire structures (rows, columns, filter channels) are then zeroed out and removed.



✅ Real Latency Improvement: Removing an entire row/column shrinks the matrix dimensions allowing our hardware can process the same efficiently.

❌ Complex : This is not a simple threshold test. It's a high-stakes architectural change.
❌ Catastrophic Forgetting: Removing an entire neuron or channel can have a massive, cascading impact, as it removes all relationships associated with that structure.
❌ Requires Precision: Need to understand dependencies and prune intelligently without crippling the model.

03

Post-Training Pruning Scopes

When we say "prune 50% of the weights," does that mean 50% from each layer, or 50% from the entire model?

Local Pruning

The Rule:
Apply a fixed sparsity target (e.g., 50%) independently to each layer.

➔ "Prune the 50% weakest weights within Layer 1."
➔ "Prune the 50% weakest weights within Layer 2."
➔ ...and so on.

✅ **Simple & Predictable**: Very easy to implement. You iterate through layers and apply the same logic. It guarantees a uniform sparsity level across the model.
✅ **Maintains Layer Structure**: Doesn't risk accidentally deleting an entire layer if all its weights happen to be small.
❌ **Sub-optimal**: This is a "one-size-fits-all" approach. It assumes all layers have the same capacity for redundancy. What if Layer 5 is critical and can't handle 50% pruning, while Layer 20 is over-parameterized and could easily handle 80%?

Global Pruning

The Rule :
Treat all weights in the model as a single, giant pool. Find the weakest weights across the entire network.
"Find the 50% weakest weights, wherever they may be, and prune them."

✅ **Higher Accuracy & More Efficient:** This is a "market-based" approach. It automatically discovers which layers are over-parameterized and prunes them more aggressively, while being gentle with sensitive, critical layers.
✅ **Optimal Distribution:** Achieves the target sparsity with the minimum possible impact on model performance.
❌ **More Complex:** Requires handling all model weights at once, which can be memory-intensive.
❌ **(Potential) Risk**: In extreme cases, could theoretically remove an entire layer if all its weights are globally insignificant (though this is rare in practice).

PRUNING SCOPE	LOCAL	GLOBAL
EXECUTION	Focuses on individual weights at a more fine-grained level	Considers entire network with a more big-picture approach
PROS	Simpler to implement Typically faster to execute More measured approach	Accounts for more context during pruning Potentially higher levels of compression
CONS	More likely to result in degraded model performance	Requires more computational resources

TRADEOFFS OF DIFFERENT PRUNING SCOPES

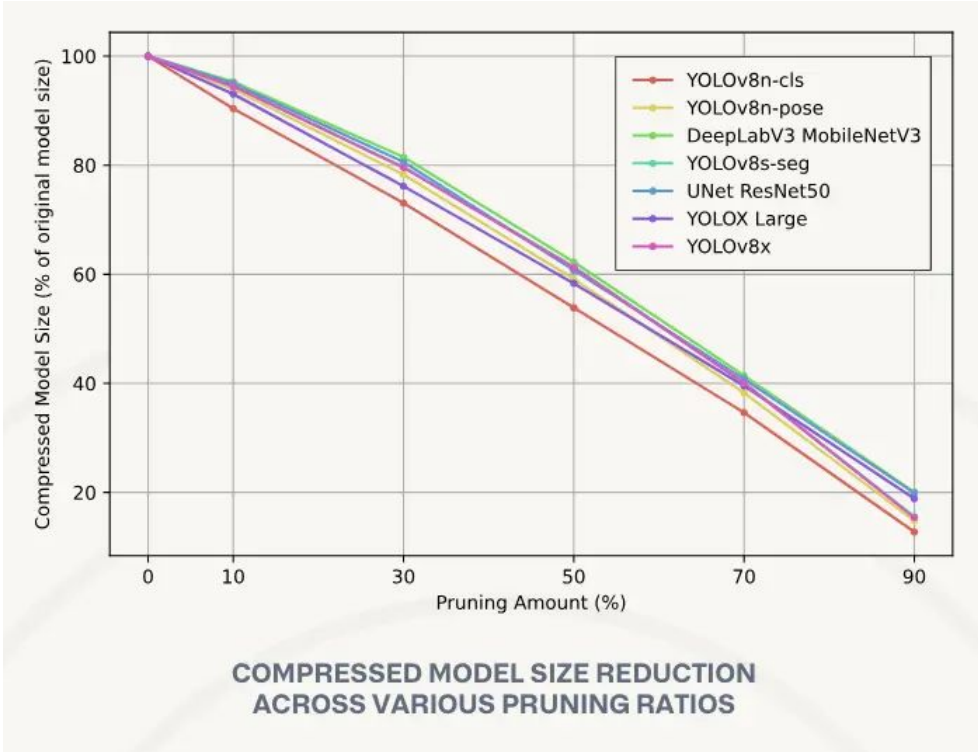
04 Effects of Pruning

Model Compression

Model pruning results in significant reductions in the model file size.

One thing to note is that the pruning is performed in a best-effort manner.

For example, a pruning percentage of 90% means that 90% of weights that are eligible for pruning will be zeroed. Since nodes like activations and biases are excluded from this process, the percentage of model compression may not directly correlate to 90% of the original model size, as evidenced by models such as UNet ResNet50 and YOLOX Large in the graph.



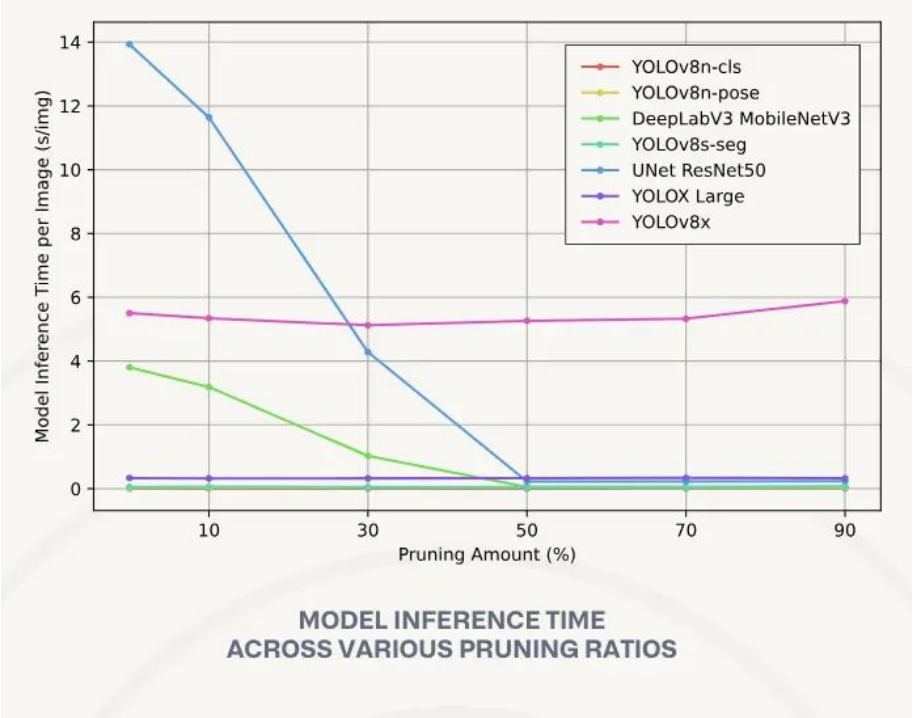
Graph Showing the Effects of Various Pruning Ratios on Compressed Model Size Across Different Model Architectures

Inference Speed

Model pruning can also reduce the inference time since zeroed weights are a simple pass-through and do not contribute to the computational complexity of the model.

In the graph, the time taken for the model to perform inference on each image generally decreases as the pruning ratio increases.

While this may not be the case for all models, most seem to follow this trend.



Graph Showing the Effects of Various Pruning Ratios on Model Inference Time Across Different Model Architectures

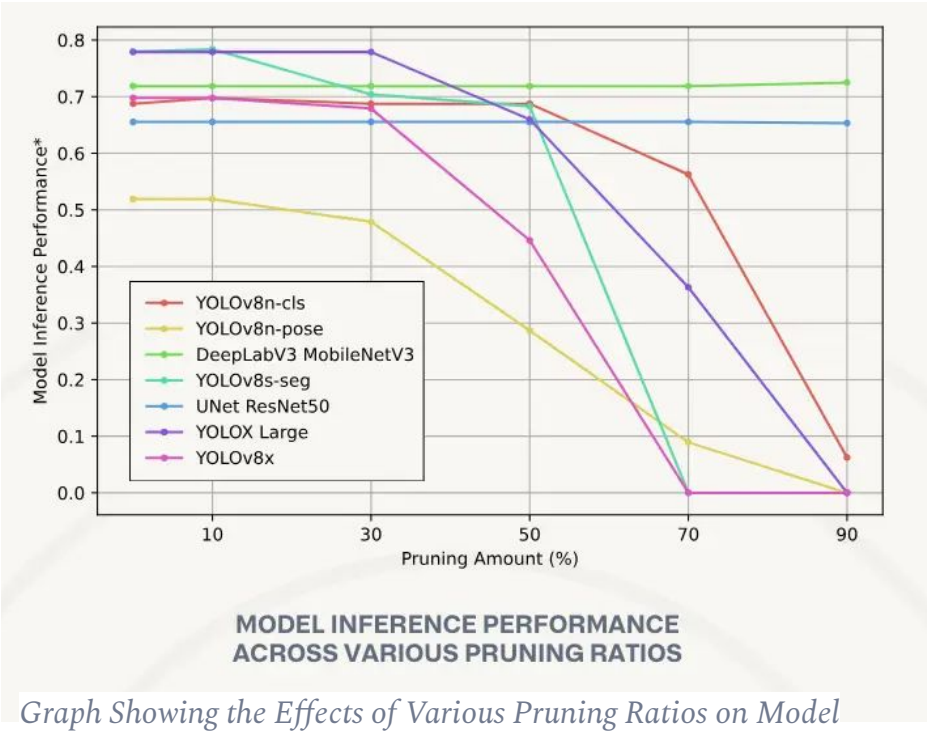
Inference Performance

Though pruning aims to zero out unimportant weights, these weights may still contribute slightly to the decision-making process of the model.

Higher pruning ratios may also inadvertently prune important weights. This may result in the accuracy degradation of the model.

Based on the graph, some models managed to retain their high performance despite a majority of their weights being zeroed out (e.g. Semantic Segmentation Models like DeepLabV3 MobileNetV3 and UNet ResNet50).

However, there are still other models that can be greatly affected by high amounts of pruning (e.g. YOLOv8x, YOLOv8s-seg).



Graph Showing the Effects of Various Pruning Ratios on Model Inference Performance Across Different Model Architectures

** Model inference performance is measured using mAP@0.5IOU for object detection, keypoint detection and instance segmentation models, while Accuracy is used for semantic segmentation and classification models. Both values range from 0 to 1, where a higher value generally indicates a better model performance.*

05

Pruning in Practice

Applications

Edge Devices

Deploying models on edge devices such as smartphones, IoT devices, or embedded systems often requires lightweight models due to limited computational resources, memory, and power constraints.

Real Time Applications

In applications where low latency is crucial, such as real-time video analysis, autonomous vehicles, or speech recognition, the pruned model requires fewer computations, leading to faster inference without compromising accuracy

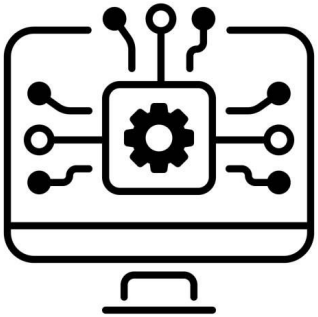
Cloud Services

Smaller models require fewer resources to deploy and maintain, leading to reduced infrastructure costs and improved scalability, especially in scenarios with high demand or elastic workloads.

Applications

Mobile Apps

Mobile applications often have limited storage space and processing power, making it challenging to deploy large models. Pruning allows developers to create more lightweight models



Embedded Systems

In industrial automation or robotics, pruning helps optimize resource utilization and improve energy efficiency. This helps prolong battery life of devices and reducing energy consumption in resource-constrained environments.

Bandwidth-Constrained Environment

In remote locations or IoT deployments with intermittent connectivity, pruned models use less data transmission during deployment and inference, leading to more reliable communication.

How Much to Prune

From the graphs shown previously, we can observe that model inference performance steeply degrades outside the “safe-zone” of 30% - 50% of parameters pruned. As such, a suggested starting point could be an initial pruned ratio of 30%.

Use-Case	**Pruning %**
High Performance Batch Jobs	For batched tasks requiring more precise predictions, you may reduce the pruning percentage so as to increase the accuracy of the performance. It is important to find the precise level that balances no loss in validation metrics while trimming as many weights as possible.
High Speed Inference	If your model must fit within a certain fixed size (e.g. 25 MB) so that it can satisfy certain requirements for deployment on an edge device, or if you want to potentially reduce the time taken for a single inference, you can opt for a certain minimum level of pruning to achieve that size.

Is Pruning a Solved Problem?

The "Gold Standard" Idea:

- Don't just remove the smallest weights.
- Remove weights that cause the least increase in the loss function.

This requires understanding the curvature of the loss landscape.

The Mathematical Tool for Curvature: The Hessian Matrix

The Hessian matrix contains all the second-order partial derivatives of the loss with respect to the weights.
It tells you how a change in one weight will affect the gradients of all other weights.
In short: It's the most precise way to measure a weight's true importance.

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \, \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \, \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \, \partial x_1} & \frac{\partial^2 f}{\partial x_n \, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

The problem is scale.

The size of the Hessian matrix grows with the square of the number of parameters.

Let's just take ResNet-50, a standard vision model with 25 million parameters. That's tiny compared to an LLM. Just to store its Hessian matrix would require 2.5 petabytes of memory. That's thousands of high-end servers. And we haven't even talked about the insane computational cost of actually calculating it.

So, the theoretically 'best' methods for pruning are physically impossible for today's Large Language Models.

We have hit a hard 'Hessian Wall

Active Research: How We Prune Today

Magnitude Pruning as a "Zeroth-Order" Approximation

Assume the Hessian is the identity matrix. This simplification mathematically reduces the problem to just pruning the weights with the lowest magnitude.

Iterative Pruning & Retraining

Since we can't get it perfect in one shot, we do it gradually

Core Loop:

1. [Prune] - Remove the p% of weights with the lowest magnitude.
2. [Fine-tune] - Retrain the model for a while to let it recover from the shock.
3. [Repeat] - Go back to step 1 until the desired sparsity is reached.

Dynamic Sparsity

What if the "best" sparse mask isn't static? What if different neurons should be active for different inputs?

Research Areas:

1. Mixture of Experts (MoE): A form of dynamic structured pruning where only a few "expert" sub-networks are activated per token.
2. Dynamic Masks: Research into learning pruning masks that change on-the-fly based on the input data.

Google DeepMind

Practical: Pruning

Colab Notebook

tme: 5 mins

References

1. [2308.06767] A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations
2. A Visual Guide to Quantization - by Maarten Grootendorst
3. What are Quantized LLMs?
4. Introduction to Weight Quantization | Towards Data Science
5. Fitting AI models in your pocket with quantization - Stack Overflow
6. A Comprehensive Guide to Neural Network Model Pruning | Datature Blog
7. Model Pruning, Distillation, and Quantization, Part 1 | Deepgram
8. Techniques for Efficient Inference of LLMs (II/IV) | by Andrei Apostol | MantisNLP | Medium
9. GGUF
10. Model Quantization 1: Basic Concepts | by Florian June | Medium

Thank you.

Nikita Saxena
Research Engineer

Google DeepMind