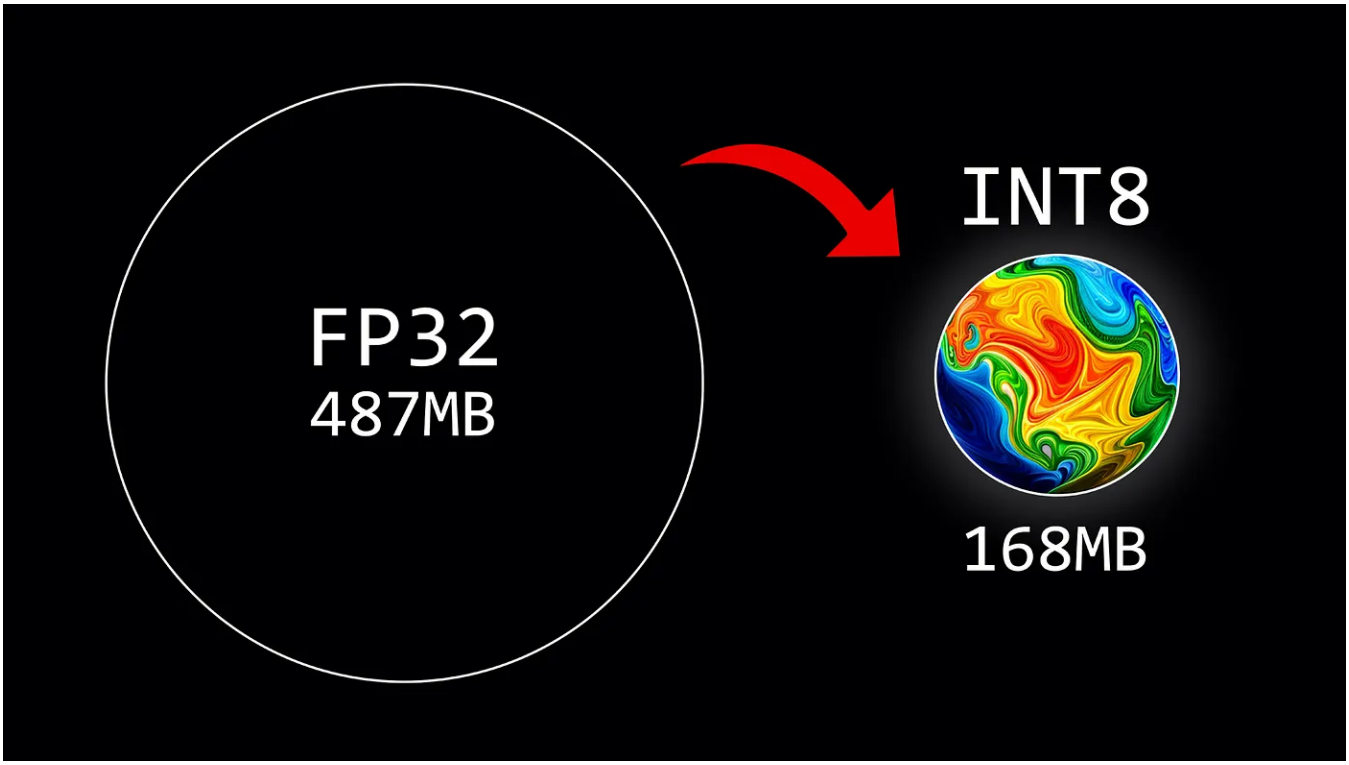
Introduction to Weight Quantization

Reducing the size of Large Language Models with 8-bit quantization

Maxime Labonne · Follow
Published in Towards Data Science
14 min read · Jul 7



Large Language Models (LLMs) are known for their extensive computational requirements. Typically, the size of a model is calculated by multiplying the number of parameters (**size**) by the precision of these values (**data type**). However, to save memory, weights can be stored using lower-precision data types through a process known as quantization.

We distinguish two main families of weight quantization techniques in the literature:

- **Post-Training Quantization** (PTQ) is a straightforward technique where the weights of an already trained model are converted to lower precision without necessitating any retraining. Although easy to implement, PTQ is associated with potential performance degradation.
- **Quantization-Aware Training** (QAT) incorporates the weight conversion process during the pre-training or fine-tuning stage, resulting in enhanced model performance. However, QAT is computationally expensive and demands representative training data.

In this article, we focus on PTQ to reduce the precision of our parameters. To get a good intuition, we will apply both naïve and more sophisticated techniques to a toy example using a GPT-2 model.

The entire code is freely available on Google Colab and GitHub.

📚 Background on Floating Point Representation

The choice of data type dictates the quantity of computational resources required, affecting the speed and efficiency of the model. In deep learning applications, balancing precision and computational performance becomes a vital exercise as higher precision often implies greater computational demands.

Among various data types, floating point numbers are predominantly employed in deep learning due to their ability to represent a wide range of values with high precision. Typically, a floating point number uses n bits to store a numerical value. These n bits are further partitioned into three distinct components:

1. **Sign**: The sign bit indicates the positive or negative nature of the number. It uses one bit where 0 indicates a positive number and 1 signals a negative number.
2. **Exponent**: The exponent is a segment of bits that represents the power to which the base (usually 2 in binary representation) is raised. The exponent can also be positive or negative, allowing the number to represent very large or very small values.
3. **Significand/Mantissa**: The remaining bits are used to store the significand, also referred to as the mantissa. This represents the significant digits of the number.

The precision of the number heavily depends on the length of the significand.

This design allows floating point numbers to cover a wide range of values with varying levels of precision. The formula used for this representation is:

$$(-1)^{\text{sign}} \times \text{base}^{\text{exponent}} \times \text{significand}$$

To understand this better, let's delve into some of the most commonly used data types in deep learning: float32 (FP32), float16 (FP16), and bfloat16 (BF16):

- **FP32** uses 32 bits to represent a number: one bit for the sign, eight for the exponent, and the remaining 23 for the significand. While it provides a high degree of precision, the downside of FP32 is its high computational and memory footprint.
- **FP16** uses 16 bits to store a number: one is used for the sign, five for the exponent, and ten for the significand. Although this makes it more memory-efficient and accelerates computations, the reduced range and precision can introduce numerical instability, potentially impacting model accuracy.
- **BF16** is also a 16-bit format but with one bit for the sign, *eight* for the exponent, and *seven* for the significand. BF16 expands the representable range compared to FP16, thus decreasing underflow and overflow risks. Despite a reduction in precision due to fewer significand bits, BF16 typically does not significantly impact model performance and is a useful compromise for deep learning tasks.

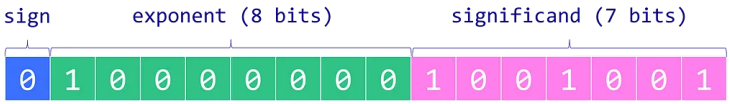
32-bit float (FP32)

$$(-1)^0 \times 2^{128-127} \times 1.5707964 = 3.1415927$$

16-bit float (FP16)

$$(-1)^0 \times 2^{128-127} \times 1.571 = 3.141$$

bfloat16 (BF16)



$$(-1)^0 \times 2^{128-127} \times 1.5703125 = 3.140625$$

Image by author

In ML jargon, FP32 is often termed “full precision” (4 bytes), while BF16 and FP16 are “half-precision” (2 bytes). But could we do even better and store weights using a single byte? The answer is the INT8 data type, which consists of an 8-bit representation capable of storing $2^8 = 256$ different values. In the next section, we’ll see how to convert FP32 weights into an INT8 format.

🔰 Naïve 8-bit Quantization

In this section, we will implement two quantization techniques: a symmetric one with **absolute maximum (absmax) quantization** and an asymmetric one with **zero-point quantization**. In both cases, the goal is to map an FP32 tensor **X** (original weights) to an INT8 tensor **X_quant** (quantized weights).

With **absmax quantization**, the original number is divided by the absolute maximum value of the tensor and multiplied by a scaling factor (127) to map inputs into the range [-127, 127]. To retrieve the original FP16 values, the INT8 number is divided by the quantization factor, acknowledging some loss of precision due to rounding.

$$\mathbf{X}_{\text{quant}} = \text{round}\left(\frac{127}{\max|\mathbf{X}|} \cdot \mathbf{X}\right)$$

$$\mathbf{X}_{\text{dequant}} = \frac{\max|\mathbf{X}|}{127} \cdot \mathbf{X}_{\text{quant}}$$

For instance, let's say we have an absolution maximum value of 3.2. A weight of 0.1 would be quantized to *round(0.1 × 127/3.2) = 4*. If we want to dequantize it, we would get *4 × 3.2/127 = 0.1008*, which implies an error of 0.008. Here's the corresponding Python implementation:

```
import torch

def absmax_quantize(X):
    # Calculate scale
    scale = 127 / torch.max(torch.abs(X))

    # Quantize
    X_quant = (scale * X).round()

    # Dequantize
    X_dequant = X_quant / scale

    return X_quant.to(torch.int8), X_dequant
```

With **zero-point quantization**, we can consider asymmetric input distributions, which is useful when you consider the output of a ReLU function (only positive values), for example. The input values are first scaled by the total range of values (255) divided by the difference between the maximum and minimum values. This distribution is then shifted by the zero-point to map it into the range [-128, 127] (notice the extra value compared to absmax). First, we calculate the scale factor and the zero-point value:

$$\text{scale} = \frac{255}{\max(\mathbf{X}) - \min(\mathbf{X})}$$

$$\text{zeropoint} = -\text{round}(\text{scale} \cdot \min(\mathbf{X})) - 128$$

Then, we can use these variables to quantize or dequantize our weights:

$$\mathbf{X}_{\text{quant}} = \text{round}\left(\text{scale} \cdot \mathbf{X} + \text{zeropoint}\right)$$

$$\mathbf{X}_{\text{dequant}} = \frac{\mathbf{X}_{\text{quant}} - \text{zeropoint}}{\text{scale}}$$

Let's take an example: we have a maximum value of 3.2 and a minimum value of -3.0. We can calculate the scale is *255/(3.2 + 3.0) = 41.13* and the zero-point - *round(41.13 × -3.0) - 128 = 123 -128 = -5*, so our previous weight of 0.1 would be quantized to *round(41.13 × 0.1 -5) = -1*. This is very different from the previous value obtained using absmax (4 vs. -1).

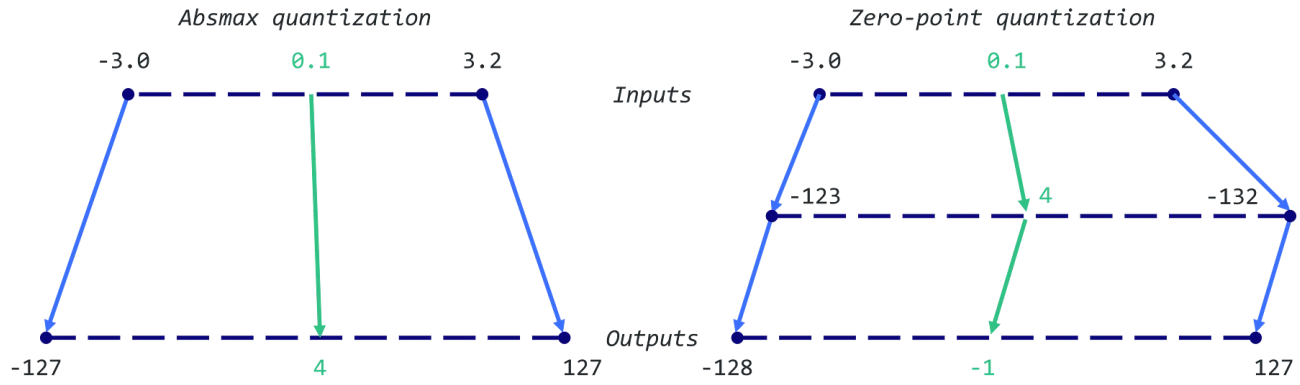


Image by author

The Python implementation is quite straightforward:

```
def zeropoint_quantize(X):
    # Calculate value range (denominator)
    x_range = torch.max(X) - torch.min(X)
    x_range = 1 if x_range == 0 else x_range

    # Calculate scale
    scale = 255 / x_range

    # Shift by zero-point
    zeropoint = (-scale * torch.min(X) - 128).round()

    # Scale and round the inputs
    X_quant = torch.clip((X * scale + zeropoint).round(), -128, 127)

    # Dequantize
    X_dequant = (X_quant - zeropoint) / scale
```

```
    return X_quant.to(torch.int8), X_dequant
```

Instead of relying on complete toy examples, we can use these two functions on a real model thanks to the `transformers` library.

We start by loading the model and tokenizer for GPT-2. This is a very small model we probably don't want to quantize, but it will be good enough for this tutorial. First, we want to observe the model's size so we can compare it later and evaluate the **memory savings** due to 8-bit quantization.

```
!pip install -q bitsandbytes>=0.39.0
!pip install -q git+https://github.com/huggingface/accelerate.git
!pip install -q git+https://github.com/huggingface/transformers.git
```

```
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch
torch.manual_seed(0)

# Set device to CPU for now
device = 'cpu'

# Load model and tokenizer
model_id = 'gpt2'
model = AutoModelForCausalLM.from_pretrained(model_id).to(device)
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Print model size
print(f"Model size: {model.get_memory_footprint():,} bytes")
```

```
Model size: 510,342,192 bytes
```

The size of the GPT-2 model is approximately 487MB in FP32. The next step consists of quantizing the weights using zero-point and absmax quantization. In the following example, we apply these techniques to the first attention layer of GPT-2 to see the results.

```
# Extract weights of the first layer
weights = model.transformer.h[0].attn.c_attn.weight.data
print("Original weights:")
print(weights)

# Quantize layer using absmax quantization
weights_abs_quant, _ = absmax_quantize(weights)
print("\nAbsmax quantized weights:")
print(weights_abs_quant)

# Quantize layer using absmax quantization
weights_zp_quant, _ = zeropoint_quantize(weights)
print("\nZero-point quantized weights:")
print(weights_zp_quant)
```

```
Original weights:
tensor([[-0.4738, -0.2614, -0.0978,  ...,  0.0513, -0.0584,  0.0250],
        [ 0.0874,  0.1473,  0.2387,  ..., -0.0525, -0.0113, -0.0156],
        [ 0.0039,  0.0695,  0.3668,  ...,  0.1143,  0.0363, -0.0318],
        ...,
        [-0.2592, -0.0164,  0.1991,  ...,  0.0095, -0.0516,  0.0319],
        [ 0.1517,  0.2170,  0.1043,  ...,  0.0293, -0.0429, -0.0475],
        [-0.4100, -0.1924, -0.2400,  ..., -0.0046,  0.0070,  0.0198]])

Absmax quantized weights:
tensor([[-21, -12,  -4,  ...,   2,  -3,   1],
        [  4,   7,  11,  ...,  -2,  -1,  -1],
        [  0,   3,  16,  ...,   5,   2,  -1],
        ...,
        [-12,  -1,   9,  ...,   0,  -2,   1],
        [  7,  10,   5,  ...,   1,  -2,  -2],
        [-18,  -9, -11,  ...,   0,   0,   1]], dtype=torch.int8)

Zero-point quantized weights:
tensor([[-20, -11,  -3,  ...,   3,  -2,   2],
        [  5,   8,  12,  ...,  -1,   0,   0],
        [  1,   4,  18,  ...,   6,   3,   0],
        ...,
        [-11,   0,  10,  ...,   1,  -1,   2],
        [  8,  11,   6,  ...,   2,  -1,  -1],
        [-18,  -8, -10,  ...,   1,   1,   2]], dtype=torch.int8)
```

The difference between the original (FP32) and quantized values (INT8) is clear, but the difference between absmax and zero-point weights is more subtle. In this case,

the inputs look shifted by a value of -1. This suggests that the weight distribution in this layer is quite symmetric.

We can compare these techniques by quantizing every layer in GPT-2 (linear layers, attention layers, etc.) and create two new models: `model_abs` and `model_zp`. To be precise, we will actually replace the original weights with *de*-quantized ones. This has two benefits: it allows us to 1/ compare the distribution of our weights (same scale) and 2/ actually run the models.

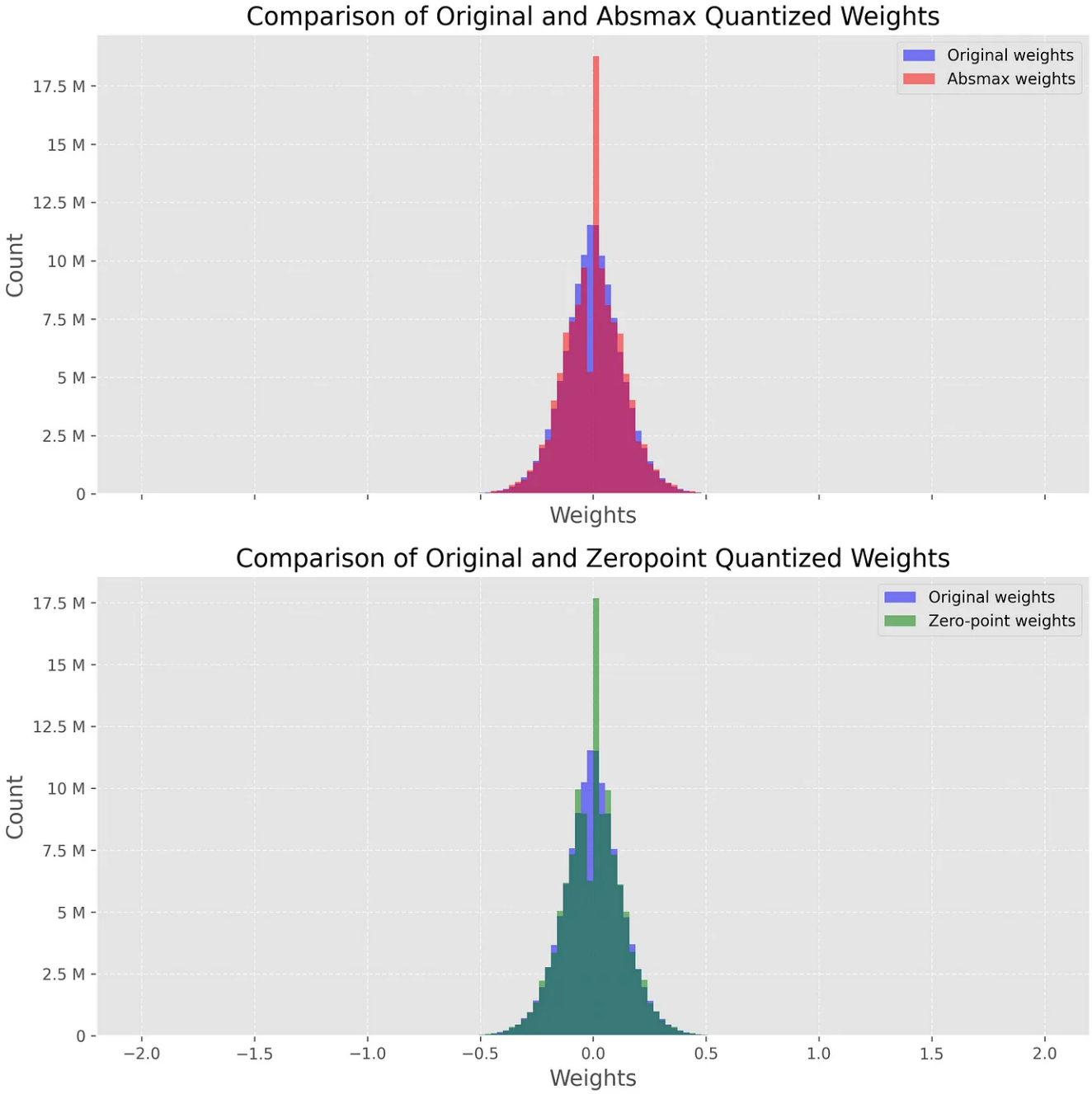
Indeed, PyTorch doesn't allow INT8 matrix multiplication by default. In a real scenario, we would dequantize them to run the model (in FP16 for example) but store them as INT8. In the next section, we will use the `bitsandbytes` library to solve this issue.

```
import numpy as np
from copy import deepcopy

# Store original weights
weights = [param.data.clone() for param in model.parameters()]

# Create model to quantize
model_abs = deepcopy(model)

# Quantize all model weights
weights_abs = []
for param in model_abs.parameters():
    _, dequantized = absmax_quantize(param.data)
    param.data = dequantized
    weights_abs.append(dequantized)

# Create model to quantize
model_zp = deepcopy(model)

# Quantize all model weights
weights_zp = []
for param in model_zp.parameters():
    _, dequantized = zeropoint_quantize(param.data)
    param.data = dequantized
    weights_zp.append(dequantized)
```

Now that our models have been quantized, we want to check the impact of this process. Intuitively, we want to make sure that the quantized weights are **close to the original ones**. A visual way to check it is to plot the distribution of the dequantized

and original weights. If the quantization is lossy, it would drastically change the weight distribution.

The following figure shows this comparison, where the blue histogram represents the original (FP32) weights, and the red one represents the dequantized (from INT8) weights. Note that we only display this plot between -2 and 2 because of outliers with very high absolute values (more on that later).



Both plots are quite similar, with a surprising spike around 0. This spike shows that our quantization is quite lossy since reversing the process doesn't output the original values. This is particularly true for the absmax model, which displays both a lower valley and a higher spike around 0.

Let's compare the performance of the original and quantized models. For this purpose, we define a `generate_text()` function to generate 50 tokens with top-k sampling.

```
def generate_text(model, input_text, max_length=50):
    input_ids = tokenizer.encode(input_text, return_tensors='pt').to(device)
    output = model.generate(inputs=input_ids,
                            max_length=max_length,
                            do_sample=True,
                            top_k=30,
                            pad_token_id=tokenizer.eos_token_id,
                            attention_mask=input_ids.new_ones(input_ids.shape))
    return tokenizer.decode(output[0], skip_special_tokens=True)

# Generate text with original and quantized models
original_text = generate_text(model, "I have a dream")
absmax_text   = generate_text(model_abs, "I have a dream")
zp_text       = generate_text(model_zp, "I have a dream")

print(f"Original model:\n{original_text}")
print("-" * 50)
print(f"Absmax model:\n{absmax_text}")
print("-" * 50)
print(f"Zeropoint model:\n{zp_text}")
```

```
Original model:
I have a dream, and it is a dream I believe I would get to live in my future. I
--------------------------------------------------
Absmax model:
I have a dream to find out the origin of her hair. She loves it. But there's nc

We found a photo of the hairstyle posted on
--------------------------------------------------
Zeropoint model:
I have a dream of creating two full-time jobs in America—one for people with me
```

Instead of trying to see if one output makes more sense than the others, we can quantify it by calculating the **perplexity** of each output. This is a common metric used to evaluate language models, which measures the uncertainty of a model in

predicting the next token in a sequence. In this comparison, we make the common assumption that the lower the score, the better the model is. In practice, a sentence with a high perplexity could also be correct.

We implement it using a minimal function since it doesn't need to consider details like the length of the context window since our sentences are short.

```
def calculate_perplexity(model, text):
    # Encode the text
    encodings = tokenizer(text, return_tensors='pt').to(device)

    # Define input_ids and target_ids
    input_ids = encodings.input_ids
    target_ids = input_ids.clone()

    with torch.no_grad():
        outputs = model(input_ids, labels=target_ids)

    # Loss calculation
    neg_log_likelihood = outputs.loss

    # Perplexity calculation
    ppl = torch.exp(neg_log_likelihood)

    return ppl

ppl     = calculate_perplexity(model, original_text)
ppl_abs = calculate_perplexity(model_abs, absmax_text)
ppl_zp  = calculate_perplexity(model_zp, absmax_text)

print(f"Original perplexity:  {ppl.item():.2f}")
print(f"Absmax perplexity:    {ppl_abs.item():.2f}")
print(f"Zeropoint perplexity: {ppl_zp.item():.2f}")
```

```
Original perplexity:  15.53
Absmax perplexity:    17.92
Zeropoint perplexity: 17.97
```

We see that the perplexity of the original model is **slightly lower** than the two others. A single experiment is not very reliable, but we could repeat this process multiple times to see the difference between each model. In theory, zero-point

quantization should be slightly better than absmax, but is also more costly to compute.

In this example, we applied quantization techniques to entire layers (per-tensor basis). However, we could apply it at different granularity levels: from the entire model to individual values. Quantizing the entire model in one pass would seriously degrade the performance, while quantizing individual values would create a big overhead. In practice, we often prefer the **vector-wise quantization**, which considers the variability of values in rows and columns inside of the same tensor.

However, even vector-wise quantization doesn't solve the problem of outlier features. Outlier features are extreme values (negative or positive) that appear in all transformer layers when the model reach a certain scale (>6.7B parameters). This is an issue since a single outlier can reduce the precision for all other values. But discarding these outlier features is not an option since it would **greatly degrade** the model's performance.

🔢 8-bit Quantization with LLM.int8()

Introduced by Dettmers et al. (2022), LLM.int8() is a solution to the outlier problem. It relies on a vector-wise (absmax) quantization scheme and introduces mixed-precision quantization. This means that outlier features are processed in a FP16 format to retain their precision, while the other values are processed in an INT8 format. As outliers represent about 0.1% of values, this effectively reduces the memory footprint of the LLM by almost 2x.

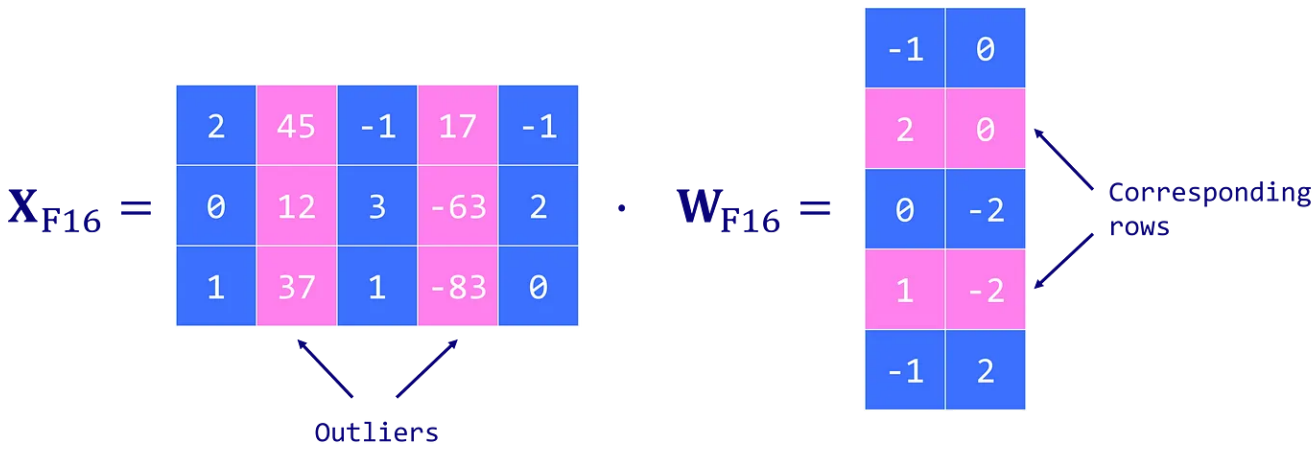


Image by author

LLM.int8() works by conducting matrix multiplication computation in three key steps:

1. Extract columns from the input hidden states **X** containing outlier features using a custom threshold.

2. Perform the matrix multiplication of the outliers using FP16 and the non-outliers using INT8 with vector-wise quantization (row-wise for the hidden state **X** and column-wise for the weight matrix **W**).

3. Dequantize the non-outlier results (INT8 to FP16) and add them to the outlier results to get the full result in FP16.

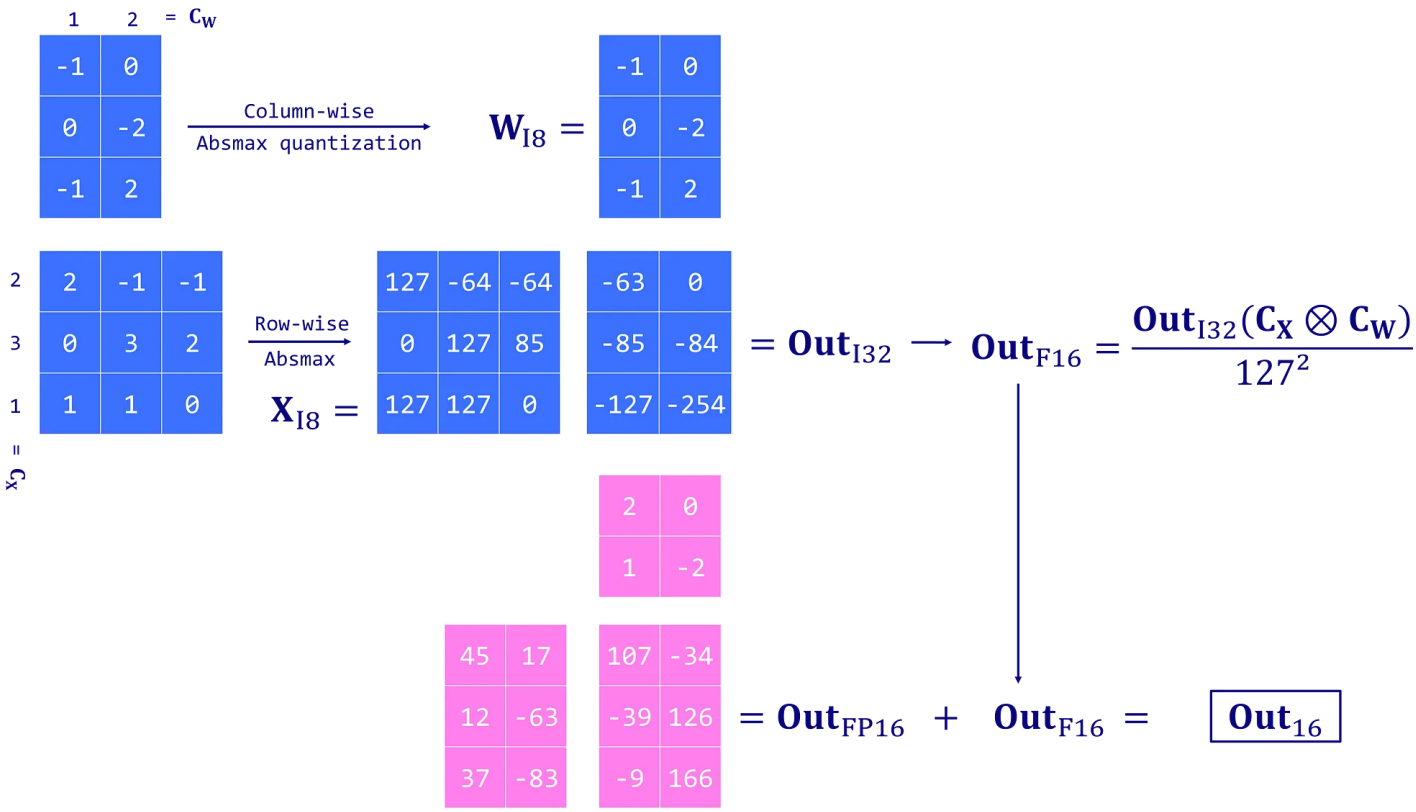


Image by author

This approach is necessary because 8-bit precision is limited and can lead to substantial errors when quantizing a vector with large values. These errors also tend to amplify as they propagate through multiple layers.

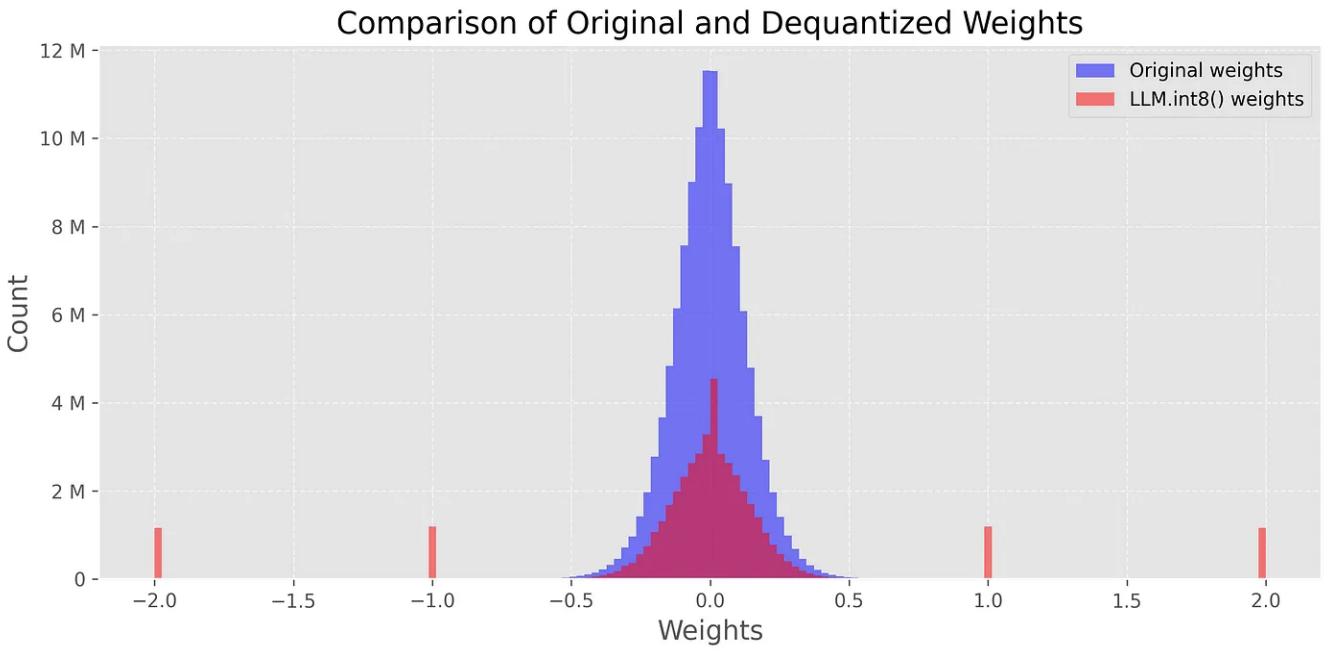
We can easily use this technique thanks to the integration of the `bitsandbytes` library into the Hugging Face ecosystem. We just need to specify `load_in_8bit=True` when loading the model (it also requires a GPU).

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model_int8 = AutoModelForCausalLM.from_pretrained(model_id,
```

```
                                                    device_map='auto',
                                                    load_in_8bit=True,
                                                    )
print(f"Model size: {model_int8.get_memory_footprint():,} bytes")
```

```
Model size: 176,527,896 bytes
```

With this extra line of code, the model is now almost three times smaller (168MB vs. 487MB). We can even compare the distribution of the original and quantized weights as we did earlier:



In this case, we see spikes around -2, -1, 0, 1, 2, etc. These values correspond to the parameters stored in the INT8 format (non-outliers). You can verify it by printing the model's weights using `model_int8.parameters()`.

We can also generate text with this quantized model and compare it to the original model.

```
# Generate text with quantized model
text_int8 = generate_text(model_int8, "I have a dream")

print(f"Original model:\n{original_text}")
```

```
print("-" * 50)
print(f"LLM.int8() model:\n{text_int8}")
```

```
Original model:
I have a dream, and it is a dream I believe I would get to live in my future. I
--------------------------------------------------
LLM.int8() model:
I have a dream. I don't know what will come of it, but I am going to have to lc
```

Once again, it is difficult to judge what is the best output, but we can rely on the perplexity metric to give us an (approximate) answer.

```
print(f"Perplexity (original):   {ppl.item():.2f}")

ppl = calculate_perplexity(model_int8, text_int8)
print(f"Perplexity (LLM.int8()): {ppl.item():.2f}")
```

```
Perplexity (original):   15.53
Perplexity (LLM.int8()): 7.93
```

In this case, the perplexity of the quantized model is twice as low as the original one. In general, this is not the case, but it shows that this quantization technique is very competitive. In fact, the authors of LLM.int8() show that the performance degradation is so low it's negligible (<1%). However, it has an additional cost in terms of computation: LLM.int8() is roughly about 20% slower for large models.

Conclusion

This article provided an overview of the most popular weight quantization techniques. We started by gaining an understanding of floating point representation, before introducing two techniques for 8-bit quantization: **absmax** and **zero-point quantization**. However, their limitations, particularly when it comes to handling outliers, led to **LLM.int8()**, a technique that also preserves the model's

performance. This approach underlines the progress being made in the field of weight quantization, revealing the importance of properly addressing outliers.

Looking forward, our next article will explore the GPTQ weight quantization technique in depth. This technique, introduced by Frantar et al., only utilizes 4 bits and represents a significant advancement in the field of weight quantization. We will provide a comprehensive guide on how to implement GPTQ using the AutoGPTQ library.

If you're interested in more technical content around LLMs, follow me on Twitter @maximelabonne.

References

- T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. 2022.
- Y. Beldaka, and T. Dettmers, A Gentle Introduction to 8-bit Matrix Multiplication, Hugging Face Blog (2022).
- A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, A Survey of Quantization Methods for Efficient Neural Network Inference. 2021.
- H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. 2020.
- Lilian Weng, Large Transformer Model Inference Optimization, Lil'Log (2023).
- Kamil Czarnogorski, Local Large Language Models, Int8 (2023).

Large Language Models | Quantization | Machine Learning | Data Science | Hands On Tutorials

Follow

Written by Maxime Labonne

3.6K Followers · Writer for Towards Data Science

Ph.D., Sr. Machine Learning Scientist @ JPMorgan • Author of "Hands-On Graph Neural Networks" • twitter.com/maximelabonne

More from Maxime Labonne and Towards Data Science

 Maxime Labonne in Towards Data Science

Graph Convolutional Networks: Introduction to GNNs

A step-by-step guide using PyTorch Geometric

16 min read · Aug 14

738 5

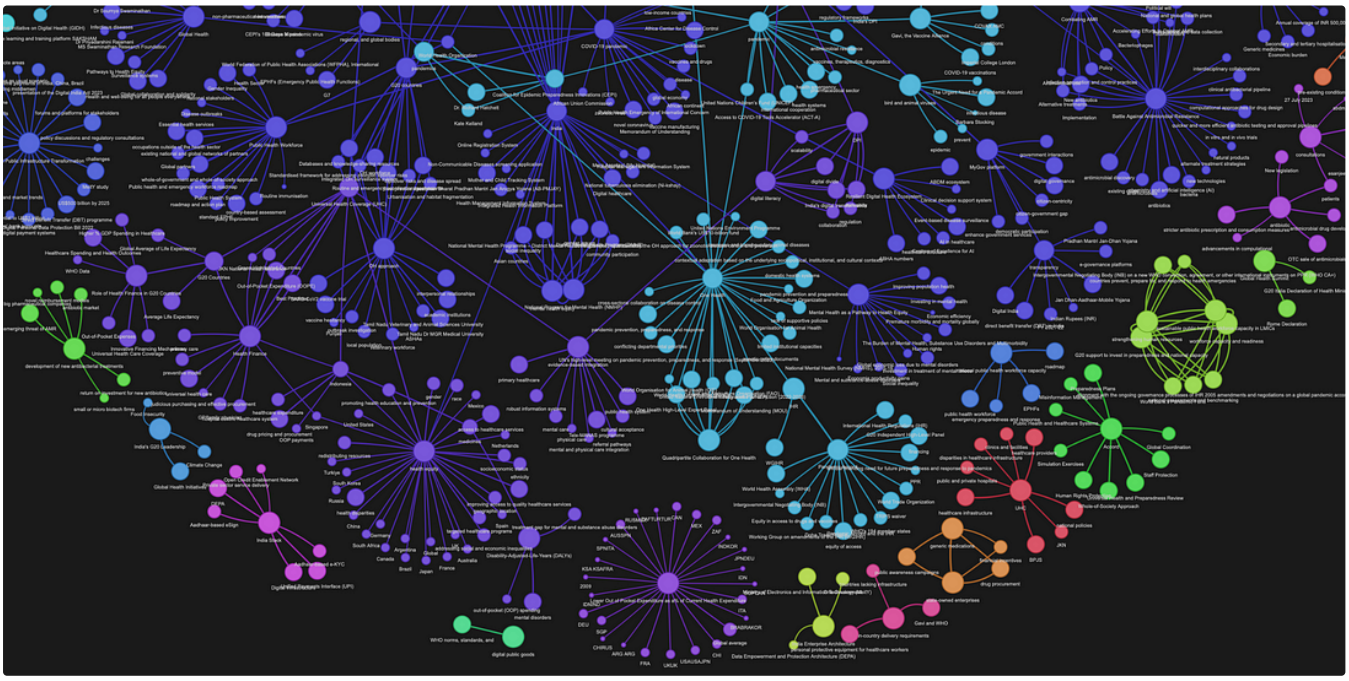
Marco Peixeiro in Towards Data Science

TimeGPT: The First Foundation Model for Time Series Forecasting

Explore the first generative pre-trained forecasting model and apply it in a project with Python

· 12 min read · Oct 24

2.2K 20



Rahul Nayak in Towards Data Science

How to Convert Any Text Into a Graph of Concepts

A method to convert any text corpus into a Knowledge Graph using Mistral 7B.

12 min read · Nov 10

1.1K 20

Maxime Labonne in Towards Data Science

Quantize Llama models with GGML and llama.cpp

GGML vs. GPTQ vs. NF4

9 min read · Sep 4

394 3

See all from Maxime Labonne

See all from Towards Data Science

Recommended from Medium

Maxime Labonne in Towards Data Science

4-bit Quantization with GPTQ

Quantize your own LLMs using AutoGPTQ

10 min read · Jul 31

413 3

Eduardo Alvarez

Llama2 Fine-Tuning with Low-Rank Adaptations (LoRA) on Gaudi 2 Processors

Learn to fine-tune Llama2 more efficiently with recently enabled Low-Rank Adaptations (LoRA) on Gaudi2 processors

9 min read · Oct 18

Phillip Gimmi

What is GGUF and GGML?

GGUF and GGML are file formats used for storing models for inference, particularly in the context of language models like GPT (Generative...

2 min read · Sep 8

74

Sergei Savvov in Better Programming

7 Ways to Speed Up Inference of Your Hosted LLMs

TLDR; techniques to speed up inference of LLMs to increase token generation speed and reduce memory consumption

14 min read · Jun 27

1K 1

Astarag Mohapatra

QLoRA: Quantized Low-Rank Adaptation paper explained

Continuing my fine-tuning journey, you can find the first article on LoRA, let's get into the QLoRA paper which was released on May 2023.

· 6 min read · Aug 7

22

Eduardo Muñoz in Towards AI

GPTQ Quantization on a Llama 2 7B Fine-Tuned Model With HuggingFace

A how-to easy-following guide on quantizing an LLM

9 min read · Sep 7