Addressing the Scale of Large Language Models: A Deep Dive into Quantization

1. Introduction

Large Language Models (LLMs) like **BERT** and **LLaMA** have revolutionized Natural Language Processing. However, their massive size presents significant challenges:

- **Memory Constraints**: A model like LLaMA-70B requires ~140GB of VRAM just to load in FP16.
- **Inference Latency**: Large models require more memory bandwidth, slowing down token generation.
- **Deployment Costs**: High-end GPUs (A100/H100) are expensive and often unavailable.

Quantization is a key technique to address these issues by reducing the precision of the model's weights and activations, effectively shrinking the model size with minimal impact on performance.

2. Theoretical Foundation of Quantization

Quantization maps high-precision values (typically **FP32** or **FP16**) to lower-precision discrete values (e.g., **INT8**, **INT4**).

2.1 Linear Quantization

The most common form is linear quantization, which can be expressed as:

$$Q(x) = \text{round}\left(\frac{x}{S} + Z\right)$$

Where:

- x: The original floating-point value.
- S: The **Scale** factor (a positive floating-point number).
- Z: The **Zero-point** (an integer ensuring that the floating-point zero maps exactly to a quantized value).

To recover the approximate floating-point value (dequantization):

$$\hat{x} = S \cdot (Q(x) - Z)$$

2.2 Symmetric vs. Asymmetric Quantization

Feature	Symmetric	Asymmetric

Zero-point (Z)	Always 0	Non-zero integer
Range	$[-r, r]$	$[min, max]$
Efficiency	Faster (simpler math)	Better utilization of bit-range

3. Advanced Quantization Techniques for LLMs

Standard INT8 quantization often causes significant accuracy drops in LLMs due to "outlier features." Modern techniques address this:

1. **GPTQ (Post-Training Quantization)**: Uses second-order information (Hessian matrix) to quantize weights layer-by-layer, minimizing the reconstruction error.
2. **AWQ (Activation-aware Weight Quantization)**: Protects important weights by observing activation magnitudes. It scales weights instead of just rounding them.
3. **bitsandbytes (NF4)**: Introduced with QLoRA, it uses a **NormalFloat 4-bit** data type, which is optimal for normally distributed weights.

4. Coding Examples

4.1 4-bit Quantization with bitsandbytes

Python

```python
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfi
import torch

model_id = "facebook/opt-125m"

# 4-bit configuration
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True,
)

# Load model
model = AutoModelForCausalLM.from_pretrained(model_id, quantization_config=bnb_
```

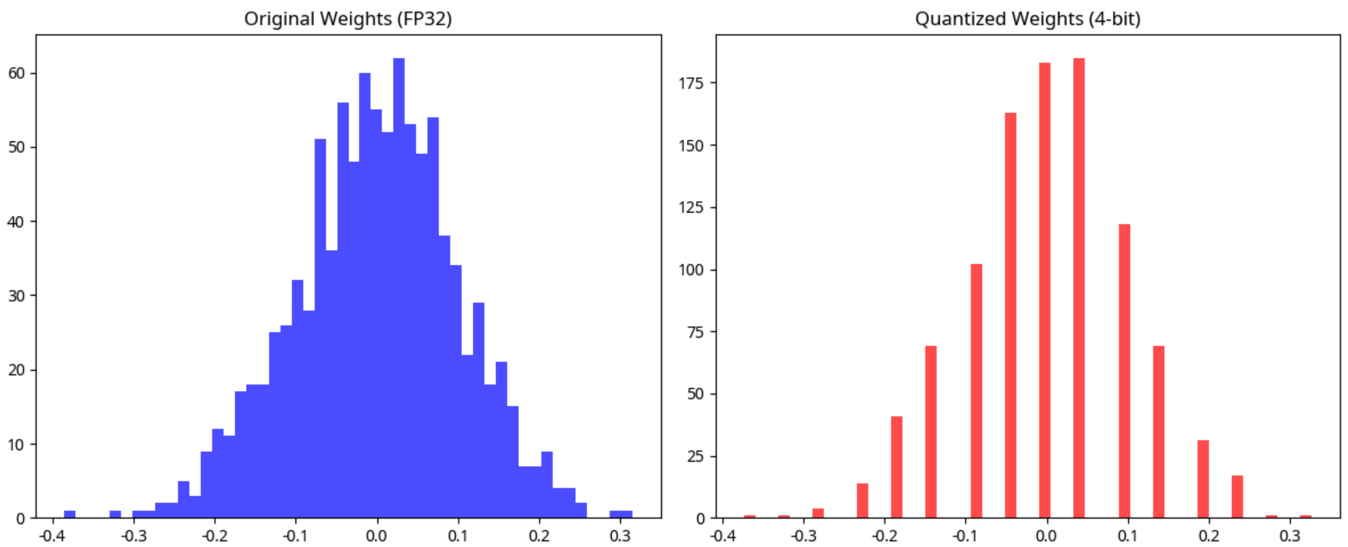
4.2 Manual Quantization Simulation

Python

```python
import numpy as np

def quantize(x, bits=4):
    qmin = 0
    qmax = 2**bits - 1
    scale = (x.max() - x.min()) / (qmax - qmin)
    zero_point = qmin - np.round(x.min() / scale)

    q_x = np.round(x / scale + zero_point)
    q_x = np.clip(q_x, qmin, qmax)

    # Dequantize
    dq_x = scale * (q_x - zero_point)
    return dq_x
```

5. Visualizing the Impact

The following graph shows how 4-bit quantization discretizes the weight distribution of a model.



6. Comparison Table

Model Size	Precision	Memory (Approx)	Accuracy Loss
LLaMA-7B	FP16	14 GB	0% (Baseline)
LLaMA-7B	INT8	7 GB	< 1%

LLaMA-7B	4-bit (GPTQ)	3.5 GB	~1-2%
LLaMA-7B	4-bit (NF4)	3.5 GB	~1%