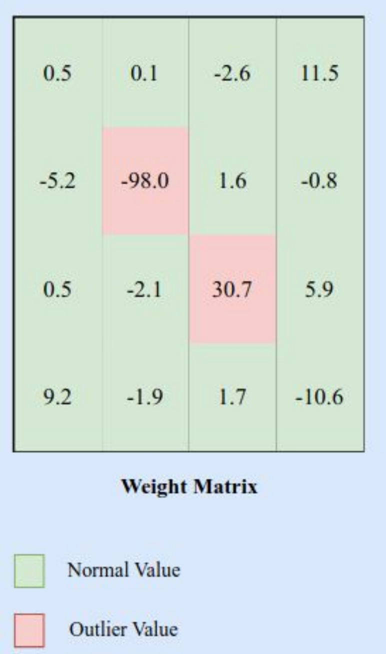
Quantization

LLM-QAT

- "Quantization Aware Training"
- Goal: improve **efficiency** and **performance** at quantization levels as low as **4 bits**
- Utilizes **data-free distillation** technique
  - Generates training data with pre-trained model
  - Helps quantization of **weights**, **activations** and **key-value-cache**
- **Directly quantize** weight matrix

1. Symmetric **MinMax-quantization** to retain outliers
2. **Student-Teacher-Model-Framework** for keeping performance of the full-precision model
3. Generation of **Next-Token-Data**
4. Use **generated data as input** for **fine-tuning** quantized model



LLM-QAT

8	8	7	8
7	0	8	7
8	7	10	8
8	7	8	7

PEQA (L4Q)

- Combines **quantization** and **parameter-efficient fine-tuning**
  - Post-Training Quantization is **efficient** but **error-prone**
  - QAT is **accurate** but **resource-intensive**
  - L4Q achieves integration **high precision** and **low memory usage**

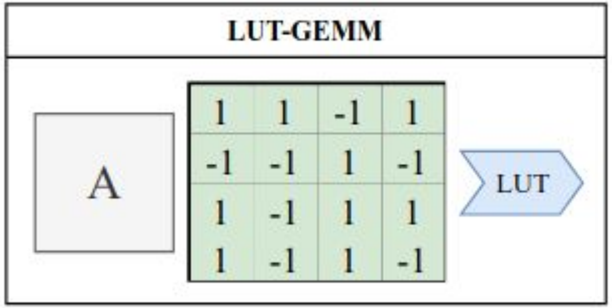
1. **Merging** weights and **LoRA**-parameters into **new weight matrix**
2. **Quantizing** New Weight Matrix (see below)
   - R: **rounding function**
   - clamp: **limits values** within quantization range
3. Optimizing Quantization Parameters with **LSQ-Method**
4. **Gradient Calculation** for LoRA Parameters

$$W_q = R(\mathrm{clamp}(\frac{W' - b}{s}, Q_n, Q_p)) \times s + b$$

LUT-GEMM

- "**L**ook**u**p **T**able-based **GEMM**"
- Enhances inference efficiency
  - **Quantizes weights** while maintaining **full precision for activations**
  - Eliminates **dequantization step**

1. Construction Lookup Tables
   - **Precompute** all possible combinations of **activation values** and **binary patterns**
2. **Table Retrieval** of precomputed partial **dot products**
   - Replaces original calculations
3. Reducing Computational Complexity
4. GPU Parallel Implementation
   - Assign as many threads as possible
   - Performs independent LUT accesses



ZeroQuant

- **Eliminates need for retraining**
- Proposal: **fine-grained, hardware-friendly** quantization strategy
  - **Layer-wise Knowledge Distillation** (LKD)
  - Allows maintenance of high model accuracy **even under extreme low-bit-width quant.**
- **Group-wise Quantization**:
  - Divide weight **matrix into multiple groups** and **quantizing each group separately**
  - Reduces **quantization errors** and improves **hardware efficiency**
- **Token-wise Quantization:**
  - Issue: significant variance in activation ranges
  - Dynamically **calculate quantization range** for each token
  - Reduces quantization errors

ZeroQuant

0.043	0.008	0.226	1.000
0.053	1.000	-0.016	0.008
0.016	0.068	1.000	0.192
-0.868	0.179	-0.160	1.000

OliVe (Outlier-Victim Pair Quantization)

- Employs **hardware-friendly** method to **handle outliers**

1. **Pair-wise Analysis**
    - Analyse **tensor values** in model and **classifies into three types of pairs**
        - Normal-normal, outlier-normal, outlier-outlier
    - Set normal values to 0 in outlier-normal pairs (**victim**) => space to handle outliers
2. **Outlier Quantization**
    - **abfloat** datatype to quantize outliers
    - Adds **suitable bias** to adjust range of floating-point values
3. Hardware-Friendly **Memory Alignment**
    - Position victims adjacent to outliers
    - Efficient memory access, low hardware overhead
    - **Avoids complexity** of **sparse indexing hardware**

More Strategies

Model	Weight		
	Feature	Consider Outliers	Consider Importance
LLM-QAT [1]	Symmetrical MinMax quantization		✓
PEQA(L4Q) [9]	LoRA		
QLORA [7]	4-bit NormalFloat (NF4) quantization	✓	✓
LUT-GEMM [10]	BCQ		
ZeroQuant [8]	fine-grained+Group-wise		
SmoothQuant [2]	diag(s)W	✓	✓
SpQR [3]	Low bit width quantization + high precision 16-bit weight storage	✓	✓
OliVe [4]	OVP+abfloat	✓	✓
GPTQ [6]	MinMax quantization of approximate second-order information	✓	✓
AWQ [11]	Determine the key parts by activating the values		
ACIQ [12]	Per-channel bit allocation		
LowbitQ [13]	kernel-wisely		
DFQ [14]	equalizing ranges		
PWLQ [15]	divide range into two regions		
Easyquant [5]	leaving outliers unchanged	✓	
BRECQ [17]	Hessian matrix		

Results of models: LLM-QAT

- Out-performs traditional PTQ
- Average zero-shot accuracy of 69.7% in 8-8-4 setting
  - Compared to 50.7 with SmootQuant
- 69.9% in the 4-8-4 setting, 1.5% less than the full model
- 4-8-8 setting beats the best PTQ(RTN) method
  - Weights are quantized to 4-bit precision
  - Activations are quantized to 8-bit precision
  - Output values are quantized to 8-bit precision
- Nothing detailed about speedup

Results of models: L4Q

- Better performance by 2% across the board for 3-bit quantization

- Out-performs QLoRA and QA-LoRA in the MMLU benchmark

Results of models: ZeroQuant

- Reduced the weights for BERT and GPT-3 without retraining, achieving a speedup of 5.19x and 4.16x with only little loss in accuracy

- Up to 3x reduction of memory footprint compared to FP16

- Successfully used in open-source models like GPT-J6B and GPT-NeoX20B

Results of models: OliVe

- Only 1% accuracy loss on GLUE benchmark with BERT

- Almost preserved GPT2-XL, BOOM-7B1, and OPT-6.7B original performance

Results of Models

Model	Feature	Activation Consider Outliers	Activation Consider Importance
LLM-QAT [1]	Activation quantization of each token	✓	✓
QLORA [7]	Brain Floating Point 16 (BFloat16)	✓	✓
LUT-GEMM [10]	Full precision		
ZeroQuant [8]	Fine-grained+Token-wise		
SmoothQuant [2]	$X diag(s)^{-1}$		
AWQ [11]	determine critical weights	✓	✓
ACIQ [12]	clip the range		✓
LowbitQ [13]	quantizing the residual		
DFQ [14]	absorbs high biases		
PWLQ [15]	same as weight		
Easyquant [5]	0: weight only		
BRECQ [17]	same as weight		

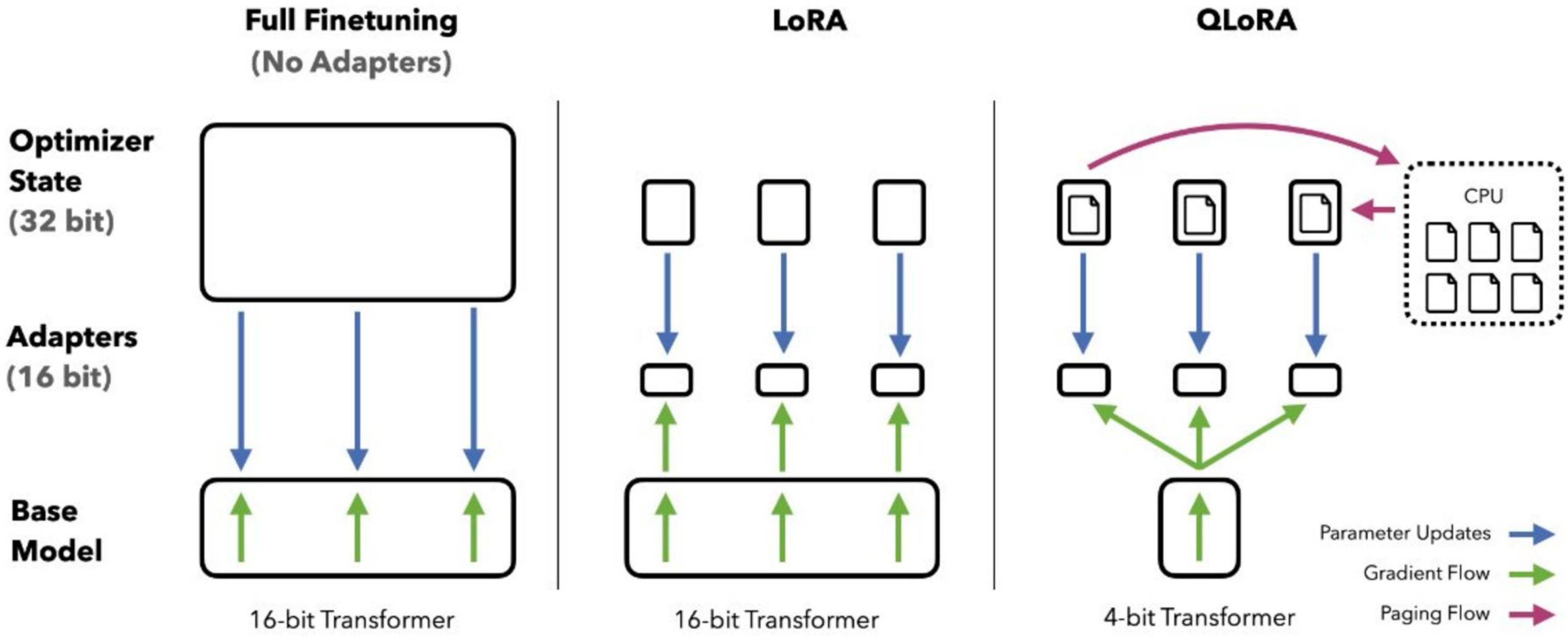
TABLE III: Comparison of Activation Quantization Across Different Large Model Quantization Algorithms. It displays the features and considerations of various algorithms in Activation quantization. The "✓" indicates that the respective algorithm considers outliers or the importance of activation during the quantization process.

Model	Memory Aligned	Trained	Knowledge Distillation Feature	Bias Correction	Calibration Set	Mixed-precision	PTQ	QAT
LLM-QAT [1]		✓	Logits Distillation		✓			✓
PEQA(L4Q) [9]		✓		✓	✓			✓
QLORA [7]	✓	✓						✓
LUT-GEMM [10]				✓		✓	✓	
ZeroQuant [8]		✓	LKD				✓	
SmoothQuant [2]							✓	
SpQR [3]	✓				✓		✓	
OliVe [4]	✓			✓			✓	
GPTQ [6]	✓				✓		✓	
AWQ [11]				✓		✓	✓	
ACIQ [12]				✓		✓	✓	
LowbitQ [13]		✓			✓		✓	
DFQ [14]				✓			✓	
PWLQ [15]				✓			✓	
SPARQ [16]							✓	
Easyquant [5]							✓	
BRECQ [17]						✓	✓	
PTQD [18]				✓	✓	✓	✓	
Zeroq [19]						✓	✓	

TABLE IV: Comparison of Different Algorithms for Quantizing Large-Scale Models. The "✓" symbol indicates that the specified feature or attribute is implemented or considered by the algorithm. This symbol helps to quickly identify which algorithms include certain functionalities, such as training, use of calibration sets, and implementation of quantization-aware training (QAT), among others.

QLora

- Quantized model weights + Low-Rank Adapters
- Quantization through bitsandbytes library
    - CUDA Wrapper
    - 8-bit & 4-bit operations
    - Apple Metal, AMD, CPU coming soon
- Low-rank adaptors at every network layer, which together still make up just 0.2% of the original model’s weight memory footprint
- “Matches” 16-bit LoRA finetuneing with 90% less memory requirements


Full Finetuning
(No Adapters)
LoRA
QLoRA
Optimizer State (32 bit)
Adapters (16 bit)
Base Model
CPU
16-bit Transformer
16-bit Transformer
4-bit Transformer
Parameter Updates
Gradient Flow
Paging Flow

Comparision

Model: Llama-2-7B

(per parameter)	16-bit finetuning	QLoRA (4bit)
weight	2 bytes	0,5 bytes
gradient	2 bytes	2 bytes
Optimizer State (Adam)	4+8 bytes	4+8 bytes
Total	112 GB without intermediate states	4,5GB (0,29% trainable parameters), 7GB with intermediate states

```python
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
from peft import LoraConfig, TaskType, get_peft_model,
prepare_model_for_kbit_training
# Create quantization config
quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_quant_type="nf4"
)

# Base model
model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf",
                                             quantization_config=quantization_config)

# Prepare quantized model for peft training
model = prepare_model_for_kbit_training(model)

# Create peft config
lora_config = LoraConfig(
    r=8,
    target_modules=["q_proj", "o_proj", "k_proj", "v_proj",
                    "gate_proj", "up_proj", "down_proj"],
    bias="none",
    task_type=TaskType.CAUSAL_LM,
)

# Create PeftModel which insert LoRA adapters using above config
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()

# Train the model using your training loop/ Huggingface Trainer

# Save the model
model.save_pretrained("lora_model")
```