

LoRA and QLoRA notes:

QLoRA: Efficient Finetuning of Large Language Models on a Single GPU? LoRA & QLoRA paper review

LORA: LOW-RANK ADAPTATION OF LARGE LANGUAGE MODELS

Edward Hu*, Yelong Shen*, Phillip Wallis, Zeyuan Allen-Zhu
Yuanzhi Li Shean Wang Lu Wang Weizhu Chen
Microsoft Corporation
{edwardhu, yeshe, phwallis, zeyuana, yuanzhil, swang, luw, wzchen}@microsoft.com
(Version 2)

ABSTRACT

An important paradigm of natural language processing consists of large-scale pre-training on general domain data and adaptation to particular tasks or domains. As we pre-train larger models, full fine-tuning, which retrains all model parameters, becomes less feasible. Using GPT-3 175B as an example – deploying independent instances of fine-tuned models, each with 175B parameters, is prohibitively expensive. We propose Low-Rank Adaptation, or LoRA, which freezes the pre-trained model weights and injects trainable rank decomposition matrices into each layer of the Transformer architecture, greatly reducing the number of trainable parameters for downstream tasks. Compared to GPT-3 175B fine-tuned with Adam, LoRA can reduce the number of trainable parameters by 10,000 times and the GPU memory requirement by 3 times. LoRA performs on-par or better than fine-tuning in model quality on RoBERTa, DeBERTa, GPT-2, and GPT-3, despite having fewer trainable parameters, a higher training throughput, and, unlike adapters, no additional inference latency. We also provide an empirical investigation into rank-deficiency in language model adaptation, which sheds light on the efficacy of LoRA. We release a package that facilitates the integration of LoRA with PyTorch models and provide our implementations and model checkpoints for RoBERTa, DeBERTa, and GPT-2 at <https://github.com/mlscope/LoRA>.

MLExpert.io

85v2 [cs.CL] 16 Oct 2021

1:38 / 12:42 • LoRA >

QLoRA: Efficient Finetuning of Quantized LLMs

Tim Dettmers*, Artidoro Pagnoni*, Ari Holtzman
Luke Zettlemoyer
University of Washington
{dettmers, artidoro, ahai, lsz}@cs.washington.edu

Abstract

We present QLoRA, an efficient finetuning approach that reduces memory usage enough to finetune a 65B parameter model on a single 48GB GPU while preserving full 16-bit finetuning task performance. QLoRA backpropagates gradients through a frozen, 4-bit quantized pretrained language model into Low Rank Adapters (LoRA). Our best model family, which we name **Guanaco**, outperforms all previous openly released models on the Vicuna benchmark, reaching 99.3% of the performance level of ChatGPT while only requiring 24 hours of finetuning on a single GPU. QLoRA introduces a number of innovations to save memory without sacrificing performance: (a) 4-bit NormalFloat (NF4), a new data type that is information theoretically optimal for normally distributed weights (b) Double Quantization to reduce the average memory footprint by quantizing the quantization constants, and (c) Paged Optimizers to manage memory spikes. We use QLoRA to finetune more than 1,000 models, providing a detailed analysis of instruction following and chatbot performance across 8 instruction datasets, multiple model types (LLaMA, T5), and model scales that would be infeasible to run with regular finetuning (e.g. 33B and 65B parameter models). Our results show that QLoRA

MLExpert.io

14v1 [cs.LG] 23 May 2023

[LoRA and QLoRA Explanation | Parameterized Efficient Finetuning of Large Language Models | PEFT](#)



LoRA and QLoRA Explanation | Parameterized Efficient Finetuning of Large Language Models | PEFT

Home Insert Design Transitions Animation Slide Show Review View Tools

Format Painter From Current Slide+ New Layout+ Section+ Reset

Outline Slides <

1. Pretrained Efficient Finetuning

2. Issues with Full Fine-tuning

3. LoRA

4. QLoRA

5. Summary

6. Conclusion

Click to add notes

Slide 1 / 9

33°C Mostly sunny

0:13 / 44:42 • Intro >

Object Formatting Fill

Solid fill

Gradient fill

Picture or texture fill

Pattern fill

Hide background graphics

Color

Transparency 0 %

Apply to all Reset Background

LoRA and QLoRA Explanation | Parameterized Efficient Finetuning of Large Language Models | PEFT

Issues with Full Fine-tuning

- Training the complete network can be a computationally expensive task, particularly for the average user dealing with large language models like LLAMA 2 or GPT.
- Catastrophic forgetting (forget the previously finetuned knowledge)
- More Storage requirements: need to save entire model on disk
- Inference can be expensive since it requires loading the entire fine-tuned model, including all of its weights, which leads to higher latency.

2:41 / 44:42 • Issue with Full Finetuning >

Full Rank Matrices

If a matrix of order (d,k) and its
rank = $\min(d,k)$
then it is said to be full rank matrix.



Rank Deficient Matrices

If a matrix of order (d,k) and its
rank < $\min(d,k)$
then it is said to be rank deficient matrix.



Rank of matrix is total number of independent rows and columns

LoRA and QLoRA Explanation | Parameterized Efficient Finetuning of Large Language Models | PEFT

Full Rank Matrices

If a matrix of order (d,k) and its
rank = $\min(d,k)$
then it is said to be full rank matrix.

$d=5$

$K=3$

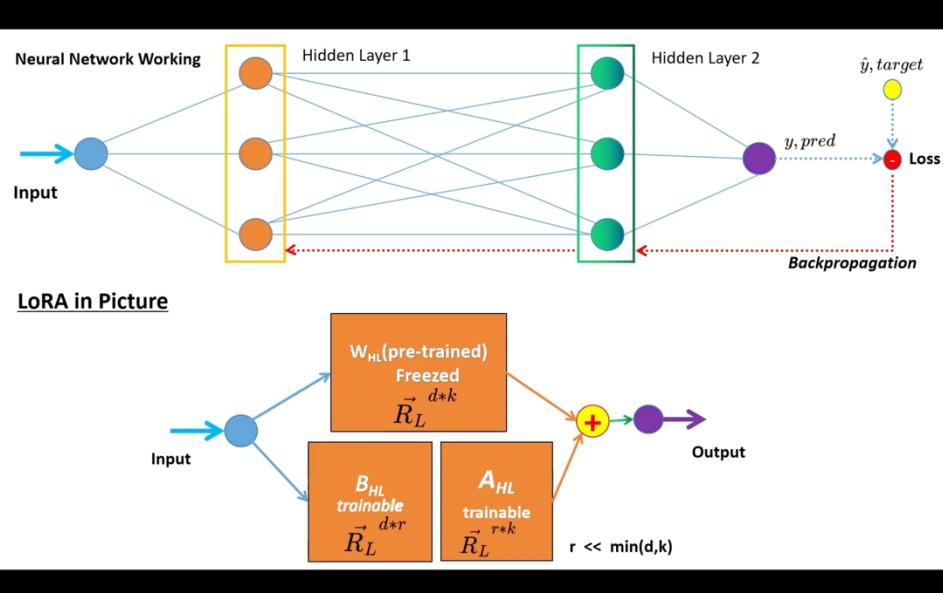
1, 2

Rank Deficient Matrices

If a matrix of order (d,k) and its
rank < $\min(d,k)$
then it is said to be rank deficient matrix.

Note:
Rank-deficient matrices play a significant role in LoRA because they allow the model to learn these changes with fewer trainable parameters. This is because a rank-deficient matrix can represent a high-dimensional space with fewer dimensions.

5:04 / 44:42 • Low Rank or Rank Deficient Matrix >



LoRA and QLoRA Explanation | Parameterized Efficient Finetuning of Large Language Models | PEFT

Benefits of using LoRA

- In our example,

original parameters of $W_{HL} = d \cdot k = 8000 \cdot 2000 = 16M$
total parameters= **16M(freezed)**

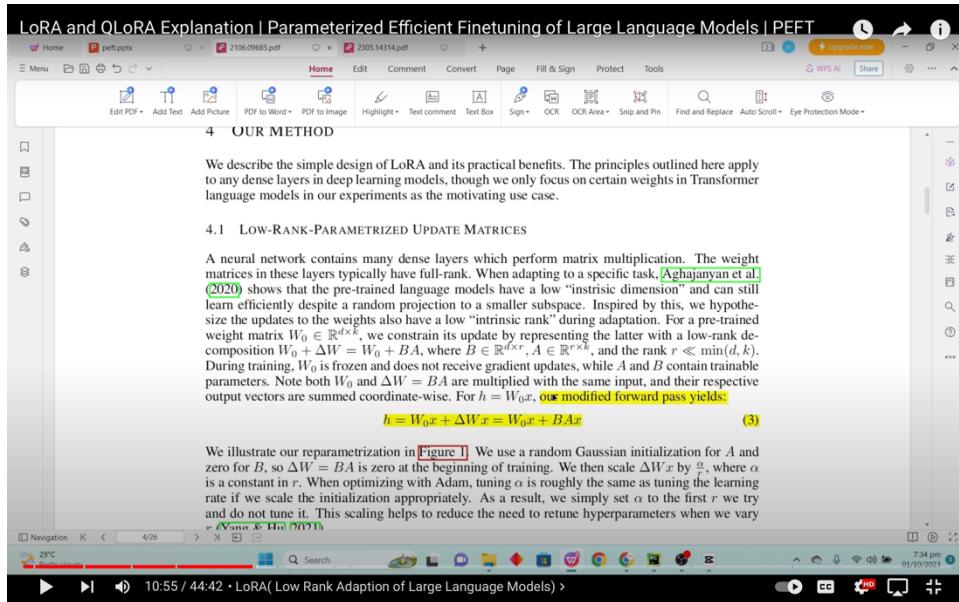
By using LoRA:

assume $r=10$ then, we have $B = d \cdot r = 8000 \cdot 10 = 80K$, $A = r \cdot k = 10 \cdot 2000 = 20K$
Total LoRA parameters= $(80K + 20K) = 100K$ (trainable)
which is ~ 160 times less than original parameters.

So less parameters,

- less computation, faster training**
- less storage required**
- Less catastrophic forgetting**
- task oriented approach**

10:15 / 44:42 • LoRA(Low Rank Adaption of Large Language Models) >



Rank Decomposition Methods

- SVD
- Randomized SVD
- Alternating least squares (ALS)
- Nuclear norm minimization (NNM)

Significance of Rank Decomposition/Factorization

we obtain a resultant matrices B and A that repeats with high accuracy the characteristics of the initial W matrix, but has a smaller dimension, because it does not take into account redundant and unnecessary information.

LoRA and QLoRA Explanation | Parameterized Efficient Finetuning of Large Language Models | PEFT

```

File Edit View Navigate Code Refactor Run Tools VCS Window Help pydantic_use_cases - a_pydantic.py
Run SVD.py
Project Structure
Run SVD.py
C:\Users\forci\pydantic_use_cases\Scripts\python.exe C: 4 X = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
5 S = [1.68481034e+01 1.06836951e+00 4.41842475e-16]
6 # Compute the low rank factorization of X with rank 2
7 U, S, Vh = np.linalg.svd(X, full_matrices=False)
8 print("S= ", S)
9 print("U= ", U[:, :2])
10 A = U[:, :2] @ np.diag(S[:2])
11 B = Vh[:, :2] @ S[:2]
12
13 # Print the low rank matrices
14 print("A= ", A)
15 print("B= ", B)
16
17 # Reconstruct the original matrix from the low rank factorization
18 X_approx = A @ B
19
20 # Print the reconstructed matrix
21 print("X_approx= ", X_approx)

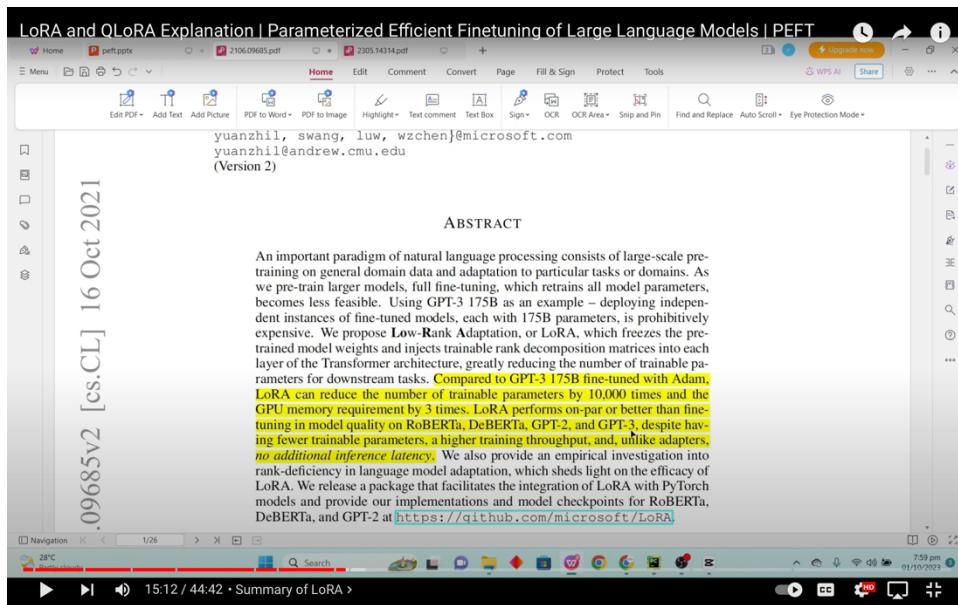
Process finished with exit code 0

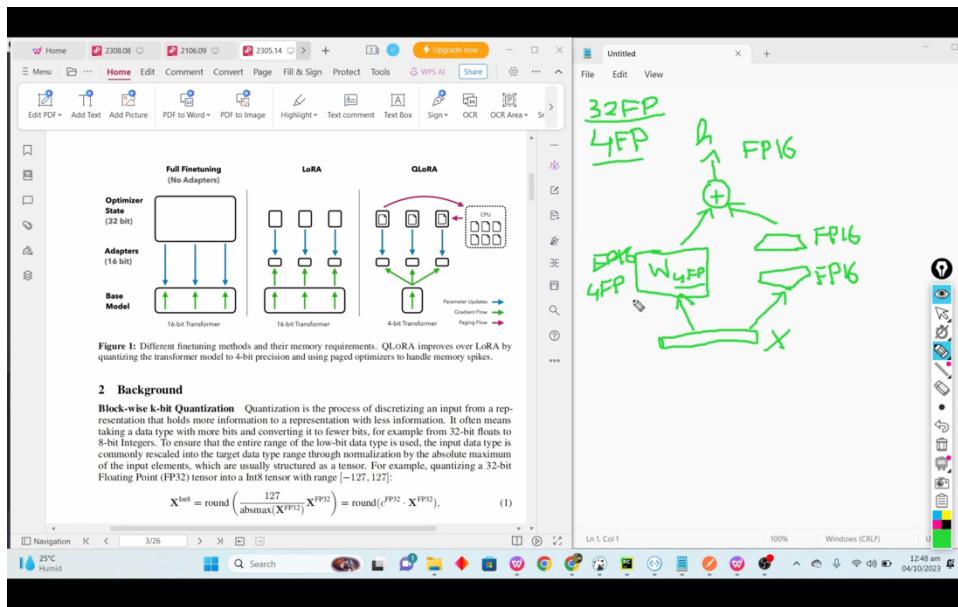
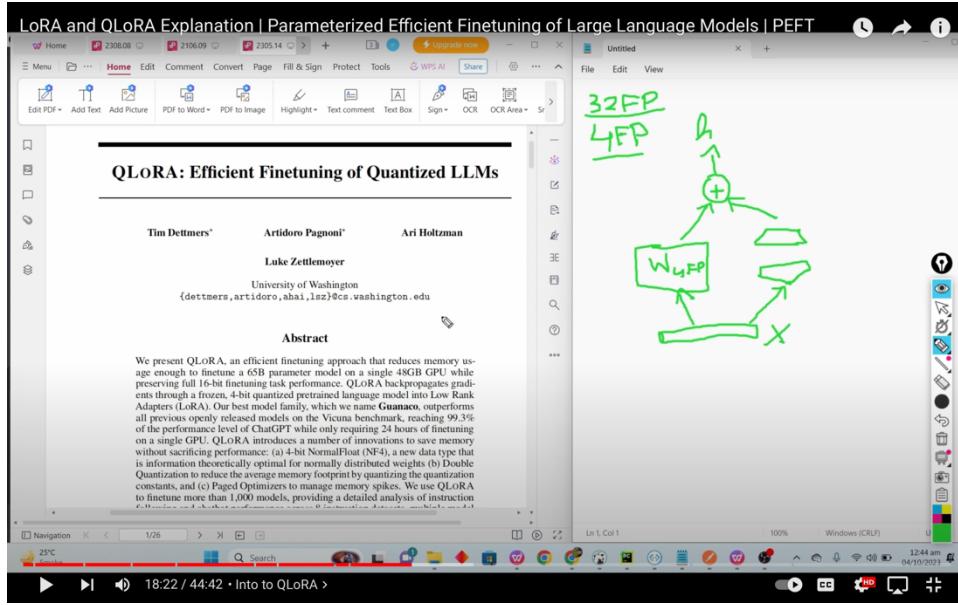
```

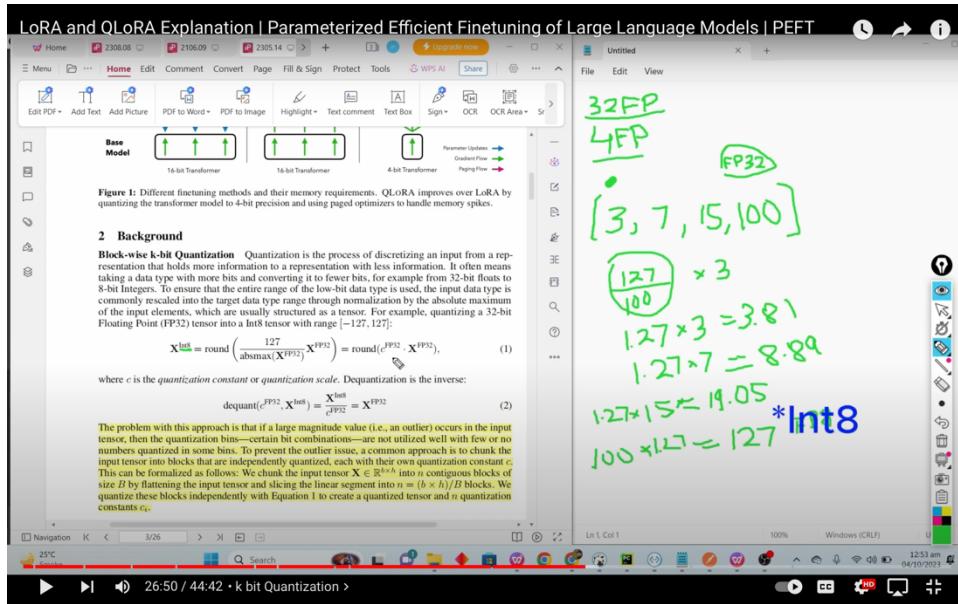
Requirement already satisfied: annotated-types>=0.4.0 in c:\users\forci\pydantic_use_cases\lib\site-packages (from pydantic) (0.5.0)

(pydantic_use_cases) PS C:\Users\forci\PycharmProjects\pydantic_use_cases>

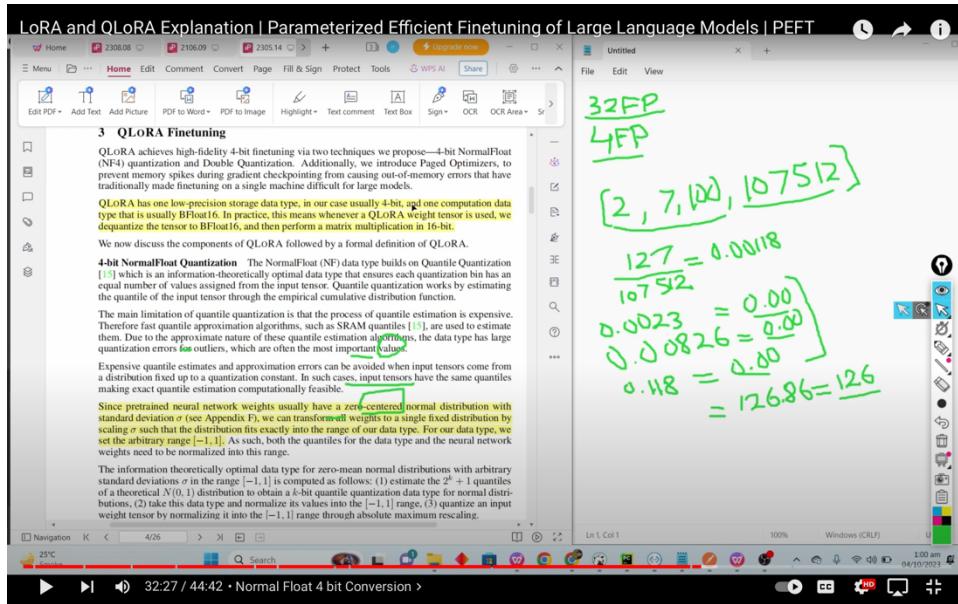
28°C 13:48 / 44:42 • Code of Singular Value Decomposition >







Outlier problem tackled by normal 4 bit precision, and breaking in chunks above.



3 QLoRA Finetuning

QLoRA achieves high-quality 4-bit finetuning via two techniques we propose: 4-bit NormalFloat (NF4) quantization and Double Quantization. Additionally, we introduce Paged Optimizers, to prevent memory spikes during gradient checkpointing from causing out-of-memory errors that have traditionally made finetuning on a single machine difficult for large models.

QLoRA has one low-precision storage data type, in our case usually 4-bit, and one computation data type that is usually BFloat16. In practice, this means whenever a QLoRA weight tensor is used, we dequantize the tensor to BFloat16, and then perform a matrix multiplication in 16-bit.

We now discuss the components of QLoRA followed by a formal definition of QLoRA.

4-bit NormalFloat Quantization The NormalFloat (NF) data type builds on Quantile Quantization [13] which is an information-theoretically optimal data type that ensures each quantization bin has an equal number of values assigned from the input tensor. Quantile quantizations work by estimating the quantile of the input tensor through the empirical cumulative distribution function.

The main limitation of quantile quantization is that the process of quantile estimation is expensive. Therefore fast quantile approximation algorithms, such as SRAM quantiles [15], are used to estimate them. Due to the approximate nature of these quantile estimation algorithms, the data type has large quantization errors [~~low~~ outliers], which are often the most important [~~value~~ outliers].

Expensive quantile estimates and approximation errors can be avoided when input tensors come from a distribution fixed up to a quantization constant. In such cases, input tensors have the same quantiles making exact quantile estimation computationally feasible.

Since previous neural network weight tensors are drawn from zero-mean normal distributions with standard deviation σ (see Appendix F), we can transform weights to a single fixed distribution by scaling σ such that the distribution fits exactly into the range of our data type. For our data type, we set the arbitrary range $[-1, 1]$. As such, both the quantiles for the data type and the neural network weights need to be normalized into this range.

The information theoretically optimal data type for zero-mean normal distributions with arbitrary standard deviations σ in the range $[-1, 1]$ is computed as follows: (1) estimate the $2^k + 1$ quantiles of a theoretical $N(0, 1)$ distribution to obtain a k -bit quantile quantization data type for normal distributions, (2) take this data type and normalize its values into the $[-1, 1]$ range, (3) quantize an input weight tensor by normalizing it into the $[-1, 1]$ range through absolute maximum rescaling.

35:18 / 44:42 • Normal Float 4 bit Conversion >

ensure a discrete mapping of 0 and to use all 2^k bits for a k -bit datatype, we create an asymmetric data type by estimating the quantiles q_i of two ranges $q_1 \dots q^{k-1}$ for the negative part and $q^{k-1} + 1$ for the positive part and then we unify these sets of q_i and remove one of the two zeros that occurs in both sets. We term the resulting data type that has equal expected number of values in each quantization bin *k-bit NormalFloat* (NF k), since the data type is information-theoretically optimal for zero-centered normally distributed data. The exact values of this data type can be found in Appendix E.

Double Quantization We introduce *Double Quantization* (DQ), the process of quantizing constants for additional memory savings. While a small blocksize is required for precise 4-bit quantization [13], it also incurs considerable memory overhead. For example, using 32-bit constants and a blocksize of 64 for W , quantization constants add $32/64 = 0.5$ bits per parameter on average. Double Quantization helps reduce the memory footprint of quantized constants.

More specifically, Double Quantization introduces a constant c_{DQ} after the first quantization as inputs to a second quantization. This second step yields the quantized quantization constants $c_{\text{DQ}}^{\text{FP}}$ and the second level of quantization constants $c_{\text{DQ}}^{\text{BFP}}$. We use 8-bit Floats with a blocksize of 256 for double quantization as no performance degradation is observed for 8-bit quantization, in line with results from Zenemeyer and Zenemeyer [17]. Since the c_{DQ} are positive, we subtract the mean from c_{DQ} before quantization to center the values around zero and make use of symmetric quantization. On average, for a blocksize of 64, this quantization reduces the memory footprint per parameter from $32/64 = 0.5$ bits, to $8/64 + 32/(64 \cdot 256) = 0.127$ bits, a reduction of 0.373 bits per parameter.

Paged Optimizers use the NVIDIA unified memory³ feature which does automatic page-to-page

42:01 / 44:42 • Double Quantization and Paged Optimizers >

