Accessible Foundation Models: Systems, Algorithms, and Science

Tim Dettmers

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington
2024

Reading Committee:
Luke Zettlemoyer, Chair
Yejin Choi
Pang Wei Koh

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Accessible Foundation Models: Systems, Algorithms, and Science

Tim Dettmers

Chair of the Supervisory Committee:

Luke Zettlemoyer
Computer Science and Engineering

The ever-increasing scale of foundation models, such as ChatGPT and AlphaFold, has revolutionized AI and science more generally. However, increasing scale also steadily raises computational barriers, blocking almost everyone from studying, adapting, or otherwise using these models for anything beyond static API queries. In this thesis, I will present research that significantly lowers these barriers for a wide range of use cases, including inference algorithms that are used to make predictions after training, finetuning approaches that adapt a trained model to new data, and finally, full training of foundation models from scratch. For inference, I will describe our LLM.int8() algorithm, which showed how to enable high-precision 8-bit matrix multiplication that is both fast and memory efficient. LLM.int8() is based on the discovery and characterization of sparse outlier sub-networks that only emerge at large model scales but are crucial for effective Int8 quantization. To empirically maximize inference efficiency for devices with limited memory footprint, I will present k-bit inference scaling laws, which empirically determine the quantization procedure to get the highest performance density per bit large language models (LLMs). The main finding of k-bit inference scaling laws is that, in almost all cases, 4-bit quantization is the most effective way of getting the best LLM performance for devices with limited memory. I will also discuss follow-up work, SpQR, which combines the insights about outlier structures from LLM.int8() and the performance density maximizing approach from k-bit inference scaling laws to achieve a quantization that replicates 16-bit performance with an average 4.6 bits per parameter. For finetuning, I will introduce the QLoRA algorithm, which pushes such

quantization much further to unlock the finetuning of very large models on a single GPU by only updating a small set of the parameters while keeping most of the network in a new information-theoretically optimal 4-bit representation. For full training, I will present SWARM parallelism, which allows collaborative training of foundation models across continents on standard internet infrastructure while still being 80% as effective as the prohibitively expensive supercomputers that are currently used. Finally, I will close by outlining my plans to make future foundation models more accessible, which will be needed to maintain truly open AI-based scientific innovation as models continue to scale.

Acknowledgements

I want to thank Luke Zettlemoyer for being a great adviser, mentor and for being a fantastic human being – you are an inspiration! I hope I can one day be like you. I was so glad that I got to work with you during my PhD. Your gentle, caring, and supportive nature bundled with the sheer freedom that you gave me made all of this work possible.

I want to thank Ludwig Schmidt, Yejin Choi, Noah Smith, Pang Wei, Luis Ceze, and the rest of the UW faculty, in particular the UW NLP faculty, for their mentorship and support. I want to thank my letter writers, Luke, Yejin, Chris Re, Colin Raffel, and Kyunghyun Cho, for their support – your help and advice made it possible to be so successful on the job market. I want to thank my collaborators, in particular JustHeuristic, Colin Raffel, and Dan Alistarh, for their steadfast enthusiasm and support. I want to thank HuggingFace, in particular Younes Belkada, for help and support with integration of bitsandbytes and QLoRA into HF transformers and PEFT.

I want to thank Ari Holtzman and Gabriel Ilharco for being such awesome friends and colleagues and for the many wonderful hours walking around Green Lake. Your support and friendship is so important and made it so enjoyable to do a PhD.

I want to thank my other friends and colleagues at UW, in particular all of CSE318, Ofir Press, Sam Ainsworth, Akari Asai, Aditya Kusupati, and the rest of UW NLP. I want to thank Katelyn Mei, Stella Li, Rulin Shao, and Jacqueline He for making the past year so fun and enjoyable.

I want to thank my friend Titus von Koeller for everything. You mean the world to me. I would not be the person I am without you. Thank you for your faith in me, your friendship through good and bad, and your continuous support, advice, and friendship.

I want to thank Margaret Li for pulling me out of my pit of apathy – you gave me appetite for life, which was so important in my development. I want to thank Zoey Chen for the first years of my PhD – I had a wonderful time with you. I want to thank Jacquelyn Swaoger for helping me stand on my own feet. I want to thank Karsten Vandamme, Thomas Graebe, Bo Biene, and Soeren Rose for their encouragement and training in finding my path.

I want to thank my parents Detlef and Heike Dettmers, and my brothers Sven and Bjorn Dettmers, and my grandma Margret Sander for their support, encouragement, and love.

DEDICATION

To my past self, who decided to live and see it through – it was worth it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large foundation models, such as AlphaFold [Jumper et al., 2021] and ChatGPT [OpenAI, 2023] revolutionized computer science and biology by making it possible to transform simple input data into highly complex outputs, such as protein structure or code generations, which are difficult to obtain with any other tool. Because foundation models can generate appropriate outputs from simple-to-understand inputs, they are highly useful for various tasks and applications. These models mainly derive their utility through the unsupervised pretraining of neural networks with large amounts of parameters on large amounts of data [Brown et al., 2020a; Kaplan et al., 2020; Hoffmann et al., 2022b]. However, such models also become increasingly inaccessible for researchers and practitioners due to their increasing computational and memory requirements.

In particular, we get the following numbers if we compare the memory capacity of consumer GPUs with the largest model that can be generated from or finetuned. In the past years, memory on consumer GPUs has been limited to 11 to 24GB of memory. Generating from or finetuning a large language model (LLM) the maximum model size usable with such GPUs would be 5-10B parameters for generation and 0.75B to 1.5B parameters for finetuning. However, the best open source models currently available have 70B parameters [Touvron et al., 2023] – almost 10x too large for generation and 100x too large for finetuning – and as such, without methods that improve accessibility, these models cannot be run on consumer setups and only on expensive GPU servers that start at $150,000.

Why is accessibility important? If more researchers and practitioners can access the best models, this enables research into understanding how these models work, when they work well, when they fail, if they

harbor privacy or copyright problems, how to shape interactions to improve user experience, and how to advance the state of the art of foundation model performance. Accessibility of large-scale models is also important because smaller models often yield much lower predictive performance and generation quality [Brown et al., 2020a; OpenAI, 2023; Kaplan et al., 2020; Hoffmann et al., 2022b]. As such, research on the accessibility of large foundation models is critical to enable the progress of research outside of well-resourced institutions, such as academic institutions.

This thesis presents research that shows that with the right scientific insights into the workings of foundation models, we can derive new algorithms and systems that make these large foundation models much more accessible and thus democratize access to large-scale models.

There are three main ways in which foundation models are used. These are: (1) inference of foundation models, (2) finetuning of foundation models, and (3) pretraining of foundation models. Each of these comes with its own challenges and particular user groups that face problems with access. For example, inference of foundation models, producing predictions and generations from input data, is useful for students and practitioners to run custom models on local devices such as laptops. By making large foundation models accessible on laptops, students can explore the advantages and disadvantages of working with foundation models, which is critical to developing the skills necessary to work with AI models when students join the workforce. Researchers, particularly PhD students, mainly use foundation model finetuning to adapt models to new research problems that require finetuning on task—or domain-relevant data. Since pretrained models are difficult to use without the finetuning step, this is a particularly important step to make foundation models useful for particular use cases. Pretraining foundation models are particularly useful for scientists who want to train for new scientific disciplines. Models such as AlphaFold have been very useful for the biological sciences. Still, most sciences do not yet have their own foundation model and might benefit significantly if new models are pretrained on high-quality domain-relevant data.

This thesis develops contributions in these three areas to help these user groups access foundation models. The following sections outline these three major areas, why each area is important, what the major challenges in each area are, and it outlines the contributions in this thesis that tackle those challenges. The structure of this thesis follows the outline of the following sections.

1.1 Accessible inference of foundation models

Model inference is the process where a foundation model is used to produce new predictions and generation by processing input data not seen during training. For example, the ESM-2[Lin et al., 2022] protein model can predict the structure of a protein from an input amino acid sequence, or GPT-4 [OpenAI, 2023] can take Python code input and an instruction to generate a new function that uses the existing Python code in the desired way.

There are two main ways that models are used during inference: single-batch and multi-batch inference. In single-batch inference, the model receives one input query at a time and generates one output response at a time. In multi-batch inference, the foundation model is queried with multiple inputs and generates multiple outputs simultaneously. Single batch inference is most common for personal use, for example, when one uses a large language model (LLM) on a laptop. Multi-batch inference is most commonly used by companies with large-scale deployments where achieving high hardware utilization is critical to minimize the total cost per generation or prediction.

Multi-batch inference. The main accessibility challenges for **multi-batch inference** are the GPU memory required for large models as well as achieving a high GPU utilization as measured in Floating-point Operations Per Second (FLOPS). To accomplish both memory reductions and high FLOPS utilization, one can compress the model weights and input hidden states of each layer through quantization. For example, using 8-bit Integers for the model weights instead of the typical 16-bit Float halves the memory requirements and thus improves accessibility. Furthermore, to achieve higher FLOPS throughput, the input hidden states can be quantized so that 8-bit matrix multiplication can be used, requiring both input matrices to have the same data type. Modern hardware often enables about 2x higher computational throughput when using 8-bit matrix multiplication compared to the standard 16-bit matrix multiplication. As such, the quantization of both the inputs and the weights leads to an effective and accessible solution for multi-batch inference.

Chapter 2 presents our work LLM.int8() [Dettmers et al., 2022a], where we develop the first Int8 matrix multiplication that works for LLMs and which makes **multi-batch inference** much more accessible. The key challenge to producing a stable 8-bit matrix multiplication is that large outliers are present in large-scale models, which disrupt the quantization precision so that the LLM is no longer able to produce high-quality

generations. We characterize these outliers and show that for large-scale models, they follow a highly structured pattern.

To solve this issue, we develop a simple algorithm that overcomes the quantization challenges and thus makes 8-bit matrix multiplication possible. While LLM.int8() was the first technique that made multi-batch inference of LLMs possible, it also laid the fundamental insight into emergent outliers, which has been widely built upon in future work [Lin et al., 2023; Xiao et al., 2022; Dettmers et al., 2023].

Single-batch inference. The main accessibility challenges for **singe-batch inference** are mostly GPU memory since the most common use-case is personal use on memory-limited devices such as laptops or consumer GPUs. Since FLOPS throughput is not a major requirement for single-batch inference, only the weights, and not the inputs, need to be compressed to receive memory and, with that, accessibility benefits. Again, a very effective technique for compressing weights is quantization, which can reduce the 16-bit weights to a lower precision.

However, the more a foundation model is compressed with quantization, the more the generation quality degrades. This compression-quality tradeoff raises the question of which quantization techniques and bit-precision yield the best quality predictions and generations per bit used for the foundation model. For example, an 8-bit model with 1B parameters and a 4-bit model with 2B parameters have equal amounts of total bits, 8B bits, but one of the models will have better predictive performance and thus higher performance per bit or higher performance density. How can we maximize this performance density empirically?

Chapter 3 presents the research of a very large-scale empirical study of the compression-quality tradeoff as measured as performance density per bit. We study all available open-source LLMs and study the performance density for a variety of quantization methods and for 3 to 16-bit precision models. Through this study of over 35,000 experiments, we find, empirically, that 4-bit precision and a particular set of quantization techniques are almost universally optimal. Based on these findings, we develop a list of best practices that now have been widely adopted for quantizing LLMs to make them accessible on low-grade GPUs and consumer devices. Thus, this work has been critical for making LLMs and other foundation models accessible for personal use.

Chapter 4 presents SpQR, a quantization algorithm and sparse representation that combines the previous two approaches of low-bit quantization and outlier detection and isolation to develop even stronger compres-

sion methods that preserve the performance of the LLM. Our method, SpQR, can compress an LLM to an average of 4.6 bits per parameter while still maintaining the same quality as a 16-bit LLM. With SpQR, the highest performance density of LLMs is achieved at an average of 3.4 bits per parameter, making LLMs significantly more accessible on low-grade hardware and consumer devices.

1.2 Accessible finetuning of foundation models

Finetuning is the process of taking an already trained foundation model and adapting it with a small dataset to improve its predictive and generative performance for new tasks or new domains. For example, one can finetuning a protein model on a small database of cancer-related proteins to improve the accuracy of predicting the 3D structure of novel cancer-related proteins, or a model can be finetuned on Python code to make it better at generating correct and functional Python code.

The main accessibility challenge of finetuning is the enormous GPU memory that is required. For the most common case of finetuning in 16-bit with the Adam optimizer [Kingma and Ba, 2015] one needs 12 bytes per parameter that is being finetuned. For example, a 70B parameter model will use 840GB of GPU memory.

Chapter 5 introduces QLoRA, a technique that enables 4-bit finetuning of LLMs and reduces the memory required during finetuning by a factor of 18x while still preserving the quality of the finetuning relative to 16-bit finetuning. QLoRA applies the best practices from Chapter 3 for maximum performance density and then couples this quantized 4-bit base model with low-rank adaptation (LoRA) [Hu et al., 2021] for finetuning. While the base model is run in 4-bit, the LoRA adapters remain in 16-bit, and as such, backpropagation of errors [Rumelhart et al., 1986] is used with 4-bit weights in the base model that produce 4-bit gradients, which are used to update the 16-bit LoRA weights. This is a critical innovation since 4-bit gradients and 4-bit weights yield unstable weight optimization, but 4-bit gradients and 16-bit weights yield stable optimization. QLoRA also introduces the NormalFloat 4-bit (NF4) data type, which is information-theoretically optimal for normally distributed neural network weights. We show that with this data type, QLoRA finetuning achieves the same performance as regular 16-bit finetuning. Thus, QLoRA makes finetuning dramatically more accessible while having no drawbacks in finetuning quality.

1.3 Accessible training of foundation models

The training of the foundation model usually requires supercomputers that combine hundreds or thousands of GPUs with a high-performance network, such as Infiniband with 400 Gbit/s bandwidth and nanosecond latency. Due to the cost of supercomputers, large-scale training is usually only affordable by large companies. As such, the main accessibility problem for large-scale training is the requirement of supercomputers.

To overcome this accessibility challenge, a solution could be to split up the training over several locations so that the computational requirements can be met by combining the computational power of multiple GPU clusters at multiple institutions. For example, if ten universities would sponsor 100 GPUs each, then the computational power would be sufficient to train large models equivalent to Llama 70B [Touvron et al., 2023] within a week. The main challenge of this approach is that the networking bandwidth across the internet is rarely better than 1 Gbit/s, which is about 400x slower than the networking provided by the supercomputer. Additionally, latency between GPUs is nanoseconds in a supercomputer but about 70 milliseconds for intercontinental communication. An additional challenge is that, unlike supercomputers, distributed computation might include different hardware, in particular GPUs, and the training speed is usually determined by the slowest GPU in the training system.

Chapter 6 introduces the SWARM parallelism system, which overcomes these challenges by combining several parallelization strategies and workload scheduling algorithms. In particular, SWARM parallelism uses delayed weight updates to overlap computation and communication within each geographic location, while communication between locations is facilitated through stochastic routing of micro-batches in pipeline parallelism. Stochastic pipeline parallelism allows for the fine-grained scheduling of individual data points to individual GPUs, which makes it possible to balance the computational load so that no GPU is overutilized. For training large neural networks between continents, SWARM parallelism achieves an efficiency of about 75% of regular supercomputer training that is located in a single location. As such, SWARM parallelism makes large-scale training accessible to institutions that can collaborate with others to meet the computational requirements to train large models.

Chapter 2

`LLM.int8()`: 8-bit Matrix Multiplication for Transformers at Scale

2.1 Introduction

Large pretrained language models are widely adopted in NLP [Vaswani et al., 2017b; Radford et al., 2019; Brown et al., 2020b; Zhang et al., 2022] but require significant memory for inference. For large transformer language models at and beyond 6.7B parameters, the feed-forward and attention projection layers and their matrix multiplication operations are responsible for 95%[1] of consumed parameters and 65-85% of all computation [Ilharco et al., 2020]. One way to reduce the size of the parameters is to quantize them to less bits and use low-bit-precision matrix multiplication. With this goal in mind, 8-bit quantization methods for transformers have been developed [Chen et al., 2020; Lin et al., 2020b; Zafrir et al., 2019; Shen et al., 2020]. While these methods reduce memory use, they degrade performance, usually require tuning quantization further after training, and have only been studied for models with less than 350M parameters. Degradation-free quantization up to 350M parameters is poorly understood, and multi-billion parameter quantization remains an open challenge.

In this paper, we present the first multi-billion-scale Int8 quantization procedure for transformers that does not incur any performance degradation. Our procedure makes it possible to load a 175B parameter

[1] Other parameters come mostly from the embedding layer. A tiny amount comes from norms and biases.

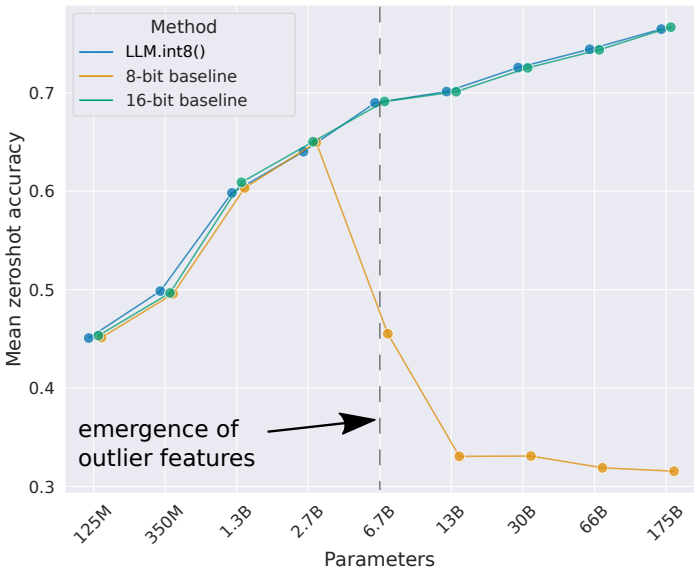


Figure 2.1: OPT model mean zeroshot accuracy for WinoGrande, HellaSwag, PIQA, and LAMBADA datasets. Shown is the 16-bit baseline, the most precise previous 8-bit quantization method as a baseline, and our new 8-bit quantization method, LLM.int8(). We can see once systematic outliers occur at a scale of 6.7B parameters, regular quantization methods fail, while LLM.int8() maintains 16-bit accuracy.

transformer with 16 or 32-bit weights, convert the feed-forward and attention projection layers to 8-bit, and use the resulting model immediately for inference without any performance degradation. We achieve this result by solving two key challenges: the need for higher quantization precision at scales beyond 1B parameters and the need to explicitly represent the sparse but systematic large magnitude outlier features that ruin quantization precision once they emerge in *all* transformer layers starting at scales of 6.7B parameters. This loss of precision is reflected in C4 evaluation perplexity (subsection 2.2) as well as zeroshot accuracy as soon as these outlier features emerge, as shown in Figure 2.1.

We show that with the first part of our method, vector-wise quantization, it is possible to retain performance at scales up to 2.7B parameters. For vector-wise quantization, matrix multiplication can be seen as a sequence of independent inner products of row and column vectors. As such, we can use a separate quantization normalization constant for each inner product to improve quantization precision. We can recover the output of the matrix multiplication by denormalizing by the outer product of column and row normalization constants before we perform the next operation.

To scale beyond 6.7B parameters without performance degradation, it is critical to understand the emergence of extreme outliers in the feature dimensions of the hidden states during inference. To this end, we

provide a new descriptive analysis which shows that large features with magnitudes up to 20x larger than in other dimensions first appear in about 25% of all transformer layers and then gradually spread to other layers as we scale transformers to 6B parameters. At around 6.7B parameters, a phase shift occurs, and *all* transformer layers and 75% of all sequence dimensions are affected by extreme magnitude features. These outliers are highly systematic: at the 6.7B scale, 150,000 outliers occur per sequence, but they are concentrated in only 6 feature dimensions across the entire transformer. Setting these outlier feature dimensions to zero decreases top-1 attention softmax probability mass by more than 20% and degrades validation perplexity by 600-1000% despite them only making up about 0.1% of all input features. In contrast, removing the same amount of random features decreases the probability by a maximum of 0.3% and degrades perplexity by about 0.1%.

To support effective quantization with such extreme outliers, we develop mixed-precision decomposition, the second part of our method. We perform 16-bit matrix multiplication for the outlier feature dimensions and 8-bit matrix multiplication for the other 99.9% of the dimensions. We name the combination of vector-wise quantization and mixed precision decomposition, **LLM.int8()**. We show that by using LLM.int8(), we can perform inference in LLMs with up to 175B parameters without any performance degradation. Our method not only provides new insights into the effects of these outliers on model performance but also makes it possible for the first time to use very large models, for example, OPT-175B/BLOOM, on a single server with consumer GPUs. While our work focuses on making large language models accessible without degradation, we also find that we maintain end-to-end inference runtime performance for large models, such as BLOOM-176B and provide modest matrix multiplication speedups for GPT-3 models of size 6.7B parameters or larger. We open-source our software[2] and release a Hugging Face Transformers [Wolf et al., 2019] integration making our method available to all hosted Hugging Face Models that have linear layers.

2.2 Int8 Matrix Multiplication at Scale

The main challenge with quantization methods that use a single scaling constant per tensor is that a single outlier can reduce the quantization precision of all other values. As such, it is desirable to have multiple

[2] `https://github.com/TimDettmers/bitsandbytes`

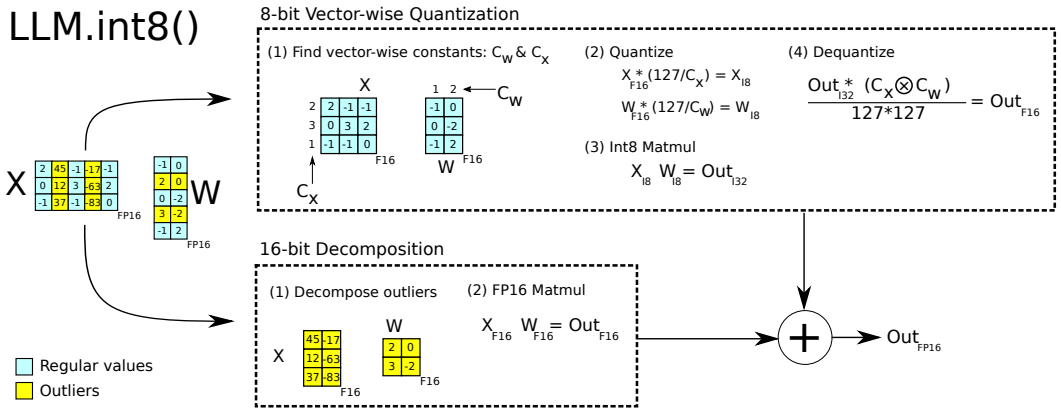


Figure 2.2: Schematic of LLM.int8(). Given 16-bit floating-point inputs $\mathbf{X}_{f16}$ and weights $\mathbf{W}_{f16}$, the features and weights are decomposed into sub-matrices of large magnitude features and other values. The outlier feature matrices are multiplied in 16-bit. All other values are multiplied in 8-bit. We perform 8-bit vector-wise multiplication by scaling by row and column-wise absolute maximum of $\mathbf{C}_x$ and $\mathbf{C}_w$ and then quantizing the outputs to Int8. The Int32 matrix multiplication outputs $\mathbf{Out}_{i32}$ are dequantization by the outer product of the normalization constants $\mathbf{C}_x \otimes \mathbf{C}_w$. Finally, both outlier and regular outputs are accumulated in 16-bit floating point outputs.

scaling constants per tensor, such as block-wise constants [Dettmers et al., 2022b], so that the effect of that outliers is confined to each block. We improve upon one of the most common ways of blocking quantization, row-wise quantization [Khudia et al., 2021], by using vector-wise quantization, as described in more detail below.

To handle the large magnitude outlier features that occur in all transformer layers beyond the 6.7B scale, vector-wise quantization is no longer sufficient. For this purpose, we develop mixed-precision decomposition, where the small number of large magnitude feature dimensions ($\approx$0.1%) are represented in 16-bit precision while the other 99.9% of values are multiplied in 8-bit. Since most entries are still represented in low-precision, we retain about 50% memory reduction compared to 16-bit. For example, for BLOOM-176B, we reduce the memory footprint of the model by 1.96x.

Vector-wise quantization and mixed-precision decomposition are shown in Figure 2.2. The **LLM.int8()** method is the combination of absmax vector-wise quantization and mixed precision decomposition.

2.2.1 Vector-wise Quantization

One way to increase the number of scaling constants for matrix multiplication is to view matrix multiplication as a sequence of independent inner products. Given the hidden states $\mathbf{X}_{f16} \in \mathbb{R}^{b \times h}$ and weight matrix

$\mathbf{W}_{f16} \in \mathbb{R}^{h \times o}$, we can assign a different scaling constant $c_{x_{f16}}$to each row of $\mathbf{X}_{f16}$ and c_w to each column of $\mathbf{W}_{f16}$. To dequantize, we denormalize each inner product result by $1/(c_{x_{f16}} c_{w_{f16}})$. For the whole matrix multiplication this is equivalent to denormalization by the outer product $\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}$, where $\mathbf{c}_x \in \mathbb{R}^s$ and $\mathbf{c}_w \in \mathbb{R}^o$. As such the full equation for matrix multiplication with row and column constants is given by:

$$\mathbf{C}_{f_{16}} \approx \frac{1}{\mathbf{c}_{x_{f16}} \otimes \mathbf{c}_{w_{f16}}} \mathbf{C}_{i32} = S \cdot \mathbf{C}_{i32} = \mathbf{S} \cdot \mathbf{A}_{i8} \mathbf{B}_{i8} = \mathbf{S} \cdot Q(\mathbf{A}_{f16})\, Q(\mathbf{B}_{f16}), \tag{2.1}$$

which we term *vector-wise quantization* for matrix multiplication.

2.2.2 The Core of LLM.int8(): Mixed-precision Decomposition

In our analysis, we demonstrate that a significant problem for billion-scale 8-bit transformers is that they have large magnitude features (*columns*), which are important for transformer performance and require high precision quantization. However, vector-wise quantization, our best quantization technique, quantizes each *row* for the hidden state, which is ineffective for outlier features. Luckily, we see that these outlier features are both incredibly sparse and systematic in practice, making up only about 0.1% of all feature dimensions, thus allowing us to develop a new decomposition technique that focuses on high precision multiplication for these particular dimensions.

We find that given input matrix $\mathbf{X}_{f16} \in \mathbb{R}^{s \times h}$, these outliers occur systematically for almost all sequence dimensions s but are limited to specific feature/hidden dimensions h. As such, we propose *mixed-precision decomposition* for matrix multiplication where we separate outlier feature dimensions into the set $O = \{i | i \in \mathbb{Z}, 0 \leq i \leq h\}$, which contains all dimensions of h which have at least one outlier with a magnitude larger than the threshold α. In our work, we find that $\alpha = 6.0$ is sufficient to reduce transformer performance degradation close to zero. Using Einstein notation where all indices are superscripts, given the weight matrix $\mathbf{W}_{f16} \in \mathbb{R}^{h \times o}$, mixed-precision decomposition for matrix multiplication is defined as follows:

$$\mathbf{C}_{f16} \approx \sum_{h \in O} \mathbf{X}^h_{f16} \mathbf{W}^h_{f16} + \mathbf{S}_{f16} \cdot \sum_{h \notin O} \mathbf{X}^h_{i8} \mathbf{W}^h_{i8} \tag{2.2}$$

where $\mathbf{S}_{f16}$ is the denormalization term for the Int8 inputs and weight matrices $\mathbf{X}_{i8}$ and $\mathbf{W}_{i8}$.

This separation into 8-bit and 16-bit allows for high-precision multiplication of outliers while using

memory-efficient matrix multiplication with 8-bit weights of more than 99.9% of values. Since the number of outlier feature dimensions is not larger than 7 ($|O| \leq 7$) for transformers up to 13B parameters, this decomposition operation only consumes about 0.1% additional memory.

2.2.3 Experimental Setup

We measure the robustness of quantization methods as we scale the size of several publicly available pretrained language models up to 175B parameters. The key question is not how well a quantization method performs for a particular model but the trend of how such a method performs as we scale.

We use two setups for our experiments. One is based on language modeling perplexity, which we find to be a highly robust measure that is very sensitive to quantization degradation. We use this setup to compare different quantization baselines. Additionally, we evaluate zeroshot accuracy degradation on OPT models for a range of different end tasks, where we compare our methods with a 16-bit baseline.

For the language modeling setup, we use dense autoregressive transformers pretrained in fairseq [Ott et al., 2019] ranging between 125M and 13B parameters. These transformers have been pretrained on Books [Zhu et al., 2015], English Wikipedia, CC-News [Nagel, 2016], OpenWebText [Gokaslan and Cohen, 2019], CC-Stories [Trinh and Le, 2018], and English CC100 [Wenzek et al., 2020]. For more information on how these pretrained models are trained, see Artetxe et al. [2021].

To evaluate the language modeling degradation after Int8 quantization, we evaluate the perplexity of the 8-bit transformer on validation data of the C4 corpus [Raffel et al., 2019] which is a subset of the Common Crawl corpus.[3] We use NVIDIA A40 GPUs for this evaluation.

To measure degradation in zeroshot performance, we use OPT models [Zhang et al., 2022], and we evaluate these models on the EleutherAI language model evaluation harness [Gao et al., 2021].

2.2.4 Main Results

The main language modeling perplexity results on the 125M to 13B Int8 models evaluated on the C4 corpus can be seen in Table 2.1. We see that absmax, row-wise, and zeropoint quantization fail as we scale, where models after 2.7B parameters perform worse than smaller models. Zeropoint quantization fails instead

[3] `https://commoncrawl.org/`

Table 2.1: C4 validation perplexities of quantization methods for different transformer sizes ranging from 125M to 13B parameters. We see that absmax, row-wise, zeropoint, and vector-wise quantization leads to significant performance degradation as we scale, particularly at the 13B mark where 8-bit 13B perplexity is worse than 8-bit 6.7B perplexity. If we use LLM.int8(), we recover full perplexity as we scale. Zeropoint quantization shows an advantage due to asymmetric quantization but is no longer advantageous when used with mixed-precision decomposition.

Parameters	125M	1.3B	2.7B	6.7B	13B
32-bit Float	25.65	15.91	14.43	13.30	12.45
Int8 absmax	87.76	16.55	15.11	14.59	19.08
Int8 zeropoint	56.66	16.24	14.76	13.49	13.94
Int8 absmax row-wise	30.93	17.08	15.24	14.13	16.49
Int8 absmax vector-wise	35.84	16.82	14.98	14.13	16.48
Int8 zeropoint vector-wise	25.72	15.94	14.36	13.38	13.47
Int8 absmax row-wise + decomposition	30.76	16.19	14.65	13.25	12.46
Absmax LLM.int8() (vector-wise + decomp)	25.83	15.93	14.44	**13.24**	**12.45**
Zeropoint LLM.int8() (vector-wise + decomp)	**25.69**	**15.92**	**14.43**	**13.24**	**12.45**

beyond 6.7B parameters. Our method, LLM.int8(), is the only method that preserves perplexity. As such, LLM.int8() is the only method with a favorable scaling trend.

When we look at the scaling trends of zeroshot performance of OPT models on the EleutherAI language model evaluation harness in Figure 2.1, we see that LLM.int8() maintains full 16-bit performance as we scale from 125M to 175B parameters. On the other hand, the baseline, 8-bit absmax vector-wise quantization, scales poorly and degenerates into random performance.

Although our primary focus is on saving memory, we also measured the run time of LLM.int8(). The quantization overhead can slow inference for models with less than 6.7B parameters, as compared to a FP16 baseline. However, models of 6.7B parameters or less fit on most GPUs and quantization is less needed in practice. LLM.int8() run times is about two times faster for large matrix multiplications equivalent to those in 175B models.

2.3 Emergent Large Magnitude Features in Transformers at Scale

As we scale transformers, outlier features with large magnitudes emerge and strongly affect *all* layers and their quantization. Given a hidden state $\mathbf{X} \in \mathbb{R}^{s \times h}$ where s is the sequence/token dimension and h the

hidden/feature dimension, we define a feature to be a particular dimension h_i. Our analysis looks at a particular feature dimension h_i across all layers of a given transformer.

We find that outlier features strongly affect attention and the overall predictive performance of transformers. While up to 150k outliers exist per 2048 token sequence for a 13B model, these outlier features are highly systematic and only representing at most 7 unique feature dimensions h_i. Insights from this analysis were critical to developing mixed-precision decomposition. Our analysis explains the advantages of zeropoint quantization and why they disappear with the use of mixed-precision decomposition and the quantization performance of small vs. large models.

2.3.1 Finding Outlier Features

The difficulty with the quantitative analysis of emergent phenomena is two-fold. We aim to select a small subset of features for analysis such that the results are intelligible and not to complex while also capturing important probabilistic and structured patterns. We use an empirical approach to find these constraints. We define outliers according to the following criteria: the magnitude of the feature is at least 6.0, affects at least 25% of layers, and affects at least 6% of the sequence dimensions.

More formally, given a transformer with L layers and hidden state $\mathbf{X}_l \in \mathbb{R}^{s \times h}, l = 0...L$ where s is the sequence dimension and h the feature dimension, we define a feature to be a particular dimension h_i in any of the hidden states $\mathbf{X}_{l_i}$. We track dimensions $h_i, 0 \leq i \leq h$, which have at least one value with a magnitude of $\alpha \geq 6$ and we only collect statistics if these outliers occur in the *same* feature dimension h_i in at least 25% of transformer layers $0...L$ and appear in at least 6% of all sequence dimensions s across all hidden states $\mathbf{X}_l$. Since feature outliers only occur in attention projection (key/query/value/output) and the feedforward network expansion layer (first sub-layer), we ignore the attention function and the FFN contraction layer (second sub-layer) for this analysis.

Our reasoning for these thresholds is as follows. We find that using mixed-precision decomposition, perplexity degradation stops if we treat any feature with a magnitude 6 or larger as an outlier feature. For the number of layers affected by outliers, we find that outlier features are *systematic* in large models: they either occur in most layers or not at all. On the other hand, they are *probabilistic* in small models: they occur *sometimes* in *some* layers for each sequence. As such, we set our threshold for how many layers need to be affected to detect an outlier feature in such a way as to limit detection to a *single* outlier in our

smallest model with 125M parameters. This threshold corresponds to that at least 25% of transformer layers are affected by an outlier in the same feature dimension. The second most common outlier occurs in only a single layer (2% of layers), indicating that this is a reasonable threshold. We use the same procedure to find the threshold for how many sequence dimensions are affected by outlier features in our 125M model: outliers occur in at least 6% of sequence dimensions.

We test models up to a scale of 13B parameters. To make sure that the observed phenomena are not due to bugs in software, we evaluate transformers that were trained in three different software frameworks. We evaluate four GPT-2 models which use OpenAI software, five Meta AI models that use Fairseq [Ott et al., 2019], and one EleutherAI model GPT-J that uses Tensorflow-Mesh [Shazeer et al., 2018b]. We also perform our analysis in two different inference software frameworks: Fairseq and Hugging Face Transformers [Wolf et al., 2019].

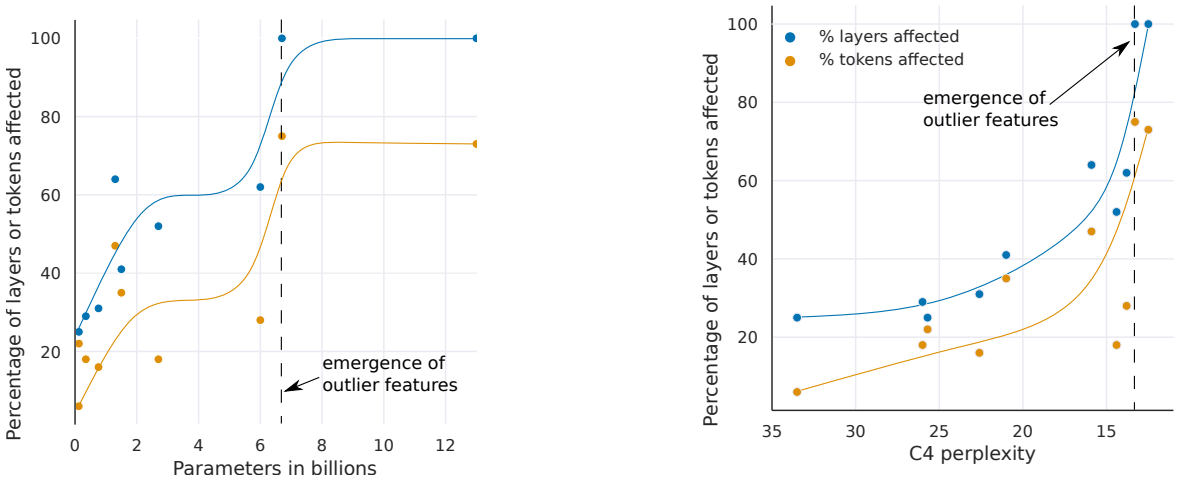
2.3.2 Measuring the Effect of Outlier Features

To demonstrate that the outlier features are essential for attention and predictive performance, we set the outlier features to zero before feeding the hidden states $\mathbf{X}_l$ into the attention projection layers and then compare the top-1 softmax probability with the regular softmax probability with outliers. We do this for all layers independently, meaning we forward the regular softmax probabilities values to avoid cascading errors and isolate the effects due to the outlier features. We also report the perplexity degradation if we remove the outlier feature dimension (setting them to zero) and propagate these altered, hidden states through the transformer. As a control, we apply the same procedure for random non-outlier feature dimensions and note attention and perplexity degradation.

Our main quantitative results can be summarized as four main points.

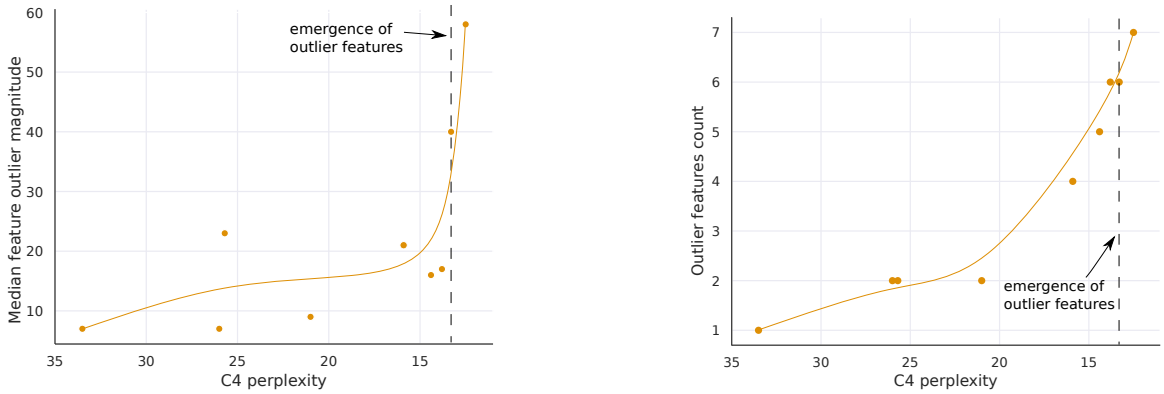
(1) When measured by the number of parameters, the emergence of large magnitude features across *all* layers of a transformer occurs suddenly between 6B and 6.7B parameters as shown in Figure 2.3a as the percentage of layers affected increases from 65% to 100%. The number of sequence dimensions affected increases rapidly from 35% to 75%. This sudden shift co-occurs with the point where quantization begins to fail.

(2) Alternatively, when measured by perplexity, the emergence of large magnitude features across all



(a) **(b)**

Figure 2.3: Percentage of layers and all sequence dimensions affected by large magnitude outlier features across the transformer by (a) model size or (b) C4 perplexity. Lines are B-spline interpolations of 4 and 9 linear segments for (a) and (b). Once the phase shift occurs, outliers are present in all layers and in about 75% of all sequence dimensions. While (a) suggest a sudden phase shift in parameter size, (b) suggests a gradual exponential phase shift as perplexity decreases. The stark shift in (a) co-occurs with the sudden degradation of performance in quantization methods.



(a) **(b)**

Figure 2.4: The median magnitude of the largest outlier feature in (a) indicates a sudden shift in outlier size. This appears to be the prime reason why quantization methods fail after emergence. While the number of outlier feature dimensions is only roughly proportional to model size, (b) shows that the number of outliers is *strictly monotonic* with respect to perplexity across all models analyzed. Lines are B-spline interpolations of 9 linear segments.

layers of the transformer can be seen as emerging smoothly according to an exponential function of decreasing perplexity, as seen in Figure 2.3b. This indicates that there is nothing sudden about emergence and that we might be able to detect emergent features before a phase shift occurs by studying exponential trends in smaller models. This also suggests that emergence is not only about model size but about perplexity, which is related to multiple additional factors such as the amount of training data used, and data quality [Hoffmann et al., 2022a; Henighan et al., 2020].

(3) Median outlier feature magnitude rapidly increases once outlier features occur in all layers of the transformer, as shown in Figure 2.4a. The large magnitude of outliers features and their asymmetric distribution disrupts Int8 quantization precision. This is the core reason why quantization methods fail starting at the 6.7B scale – the range of the quantization distribution is too large so that most quantization bins are empty and small quantization values are quantized to zero, essentially extinguishing information. We hypothesize that besides Int8 inference, regular 16-bit floating point training becomes unstable due to outliers beyond the 6.7B scale – it is easy to exceed the maximum 16-bit value 65535 by chance if you multiply by vectors filled with values of magnitude 60.

(4) The number of outliers features increases strictly monotonically with respect to decreasing C4 perplexity as shown in Figure 2.4b, while a relationship with model size is non-monotonic. This indicates that model perplexity rather than mere model size determines the phase shift. We hypothesize that model size is only one important covariate among many that are required to reach emergence.

These outliers features are highly systematic after the phase shift occurred. For example, for a 6.7B transformer with a sequence length of 2048, we find about 150k outlier features per sequence for the entire transformer, but these features are concentrated in only 6 different hidden dimensions.

These outliers are critical for transformer performance. If the outliers are removed, the mean top-1 softmax probability is reduced from about 40% to about 20%, and validation perplexity increases by 600-1000% even though there are at most 7 outlier feature dimensions. When we remove 7 random feature dimensions instead, the top-1 probability decreases only between 0.02-0.3%, and perplexity increases by 0.1%. This highlights the critical nature of these feature dimensions. Quantization precision for these outlier features is paramount as even tiny errors greatly impact model performance.

2.3.3 Interpretation of Quantization Performance

Our analysis shows that outliers in particular feature dimensions are ubiquitous in large transformers, and these feature dimensions are critical for transformer performance. Since row-wise and vector-wise quantization scale each hidden state sequence dimension s (rows) and because outliers occur in the feature dimension h (columns), both methods cannot deal with these outliers effectively. This is why absmax quantization methods fail quickly after emergence.

However, almost all outliers have a strict asymmetric distribution: they are either solely positive or negative. This makes zeropoint quantization particularly effective for these outliers, as zeropoint quantization is an asymmetric quantization method that scales these outliers into the full $[-127, 127]$ range. This explains the strong performance in our quantization scaling benchmark in Table 2.1. However, at the 13B scale, even zeropoint quantization fails due to accumulated quantization errors and the quick growth of outlier magnitudes, as seen in Figure 2.4a.

If we use our full LLM.int8() method with mixed-precision decomposition, the advantage of zeropoint quantization disappears indicating that the remaining decomposed features are symmetric. However, vector-wise still has an advantage over row-wise quantization, indicating that the enhanced quantization precision of the model weights is needed to retain full precision predictive performance.

2.4 Conclusion

We have demonstrated for the first time that multi-billion parameter transformers can be quantized to Int8 and used immediately for inference without performance degradation. We achieve this by using our insights from analyzing emergent large magnitude features at scale to develop mixed-precision decomposition to isolate outlier features in a separate 16-bit matrix multiplication. In conjunction with vector-wise quantization that yields our method, LLM.int8(), which we show empirically can recover the full inference performance of models with up to 175B parameters.

Table 2.2: Different hardware setups and which methods can be run in 16-bit vs. 8-bit precision. We can see that our 8-bit method makes many models accessible that were not accessible before, in particular, OPT-175B/BLOOM.

			Largest Model that can be run	
Class	Hardware	GPU Memory	8-bit	16-bit
Enterprise	8x A100	80 GB	**OPT-175B / BLOOM**	**OPT-175B / BLOOM**
Enterprise	8x A100	40 GB	**OPT-175B / BLOOM**	OPT-66B
Academic server	8x RTX 3090	24 GB	**OPT-175B / BLOOM**	OPT-66B
Academic desktop	4x RTX 3090	24 GB	**OPT-66B**	OPT-30B
Paid Cloud	Colab Pro	15 GB	**OPT-13B**	GPT-J-6B
Free Cloud	Colab	12 GB	**T0/T5-11B**	GPT-2 1.3B

Chapter 3

The case for 4-bit precision: k-bit Inference Scaling Laws

3.1 Introduction

Large Language Models (LLMs) are widely adopted for zero/few-shot inference Zhang et al. [2022]; Black et al. [2022]; Zeng et al. [2022]; Scao et al. [2022], but they can be challenging to use both due to their large memory footprints – up to 352 GB of GPU memory for 175B models – and high latency. However, both the memory and latency are primarily determined by the total number of bits in the parameters. Therefore, if we reduce the model bits through quantization, we can expect the latency of the model to reduce proportionally, potentially at the expense of end task accuracy [Frantar et al., 2022a; Park et al., 2022; Yao et al., 2022].

Since we can quantize the parameters of a trained model to an arbitrary bit-precision, this raises the question of how many bits should be used to optimally trade off accuracy and total model bits, given our current base methods for model quantization. For example, **if we have a 60B LLM in 4-bit precision and a 30B LLM in 8-bit precision, which will achieve higher accuracy?** To study such trade-offs, it is helpful to take the perspective of scaling laws [Kaplan et al., 2020; Henighan et al., 2020], which evaluate the underlying trends of variables to make generalizations beyond individual data points.

In this paper, we study bit-level inference scaling laws for zero-shot quantization to determine the precision that maximizes zero-shot accuracy given a certain number of total bits in the model. **Our main finding**

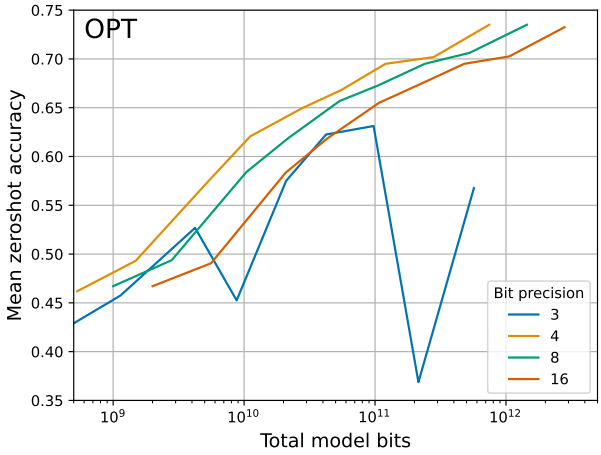


Figure 3.1: Bit-level scaling laws for mean zero-shot performance across four datasets for 125M to 176B parameter OPT models. Zero-shot performance increases steadily for fixed model bits as we reduce the quantization precision from 16 to 4 bits. At 3-bits, this relationship reverses, making 4-bit precision optimal.

is that 4-bit parameters yield optimal performance for a fixed number of model bits across all model scales and model families tested.

We study five different model families, OPT, Pythia/NeoX, GPT-2, BLOOM, and BLOOMZ [Zhang et al., 2022; Black et al., 2022; Radford et al., 2019; Scao et al., 2022], with 19M to 176B parameters, and for 3 to 16-bit precision. In addition, we run more than 35,000 zero-shot experiments to vary many of the recently studied advances in quantization precision, including the underlying data types and quantization block size. We find that reducing the precision steadily from 16 to 4 bits increases the zero-shot performance for a fixed number of model bits, while at 3-bit, the zero-shot performance decreases. This relationship holds across all models studied and did not change when scaling from 19M to 176B parameters, making 4-bit quantization universally optimal across all cases tested.

Beyond this, we analyze which quantization methods improve and which degrade bit-level scaling. We find that none of the quantization methods we test improve scaling behavior for 6 to 8-bit precision. For 4-bit precision, data types and a small quantization block size are the best ways to enhance bit-level scaling trends. We find that quantile quantization and floating point data types are the most effective, while integer and dynamic exponent data types generally yield worse scaling trends. For most 4-bit models, a block size between 64 to 128 is optimal in our study.

From our results, we can make straightforward **recommendations for using zero-shot quantized models for inference**: always use 4-bit models with a small block size and with a float data type. If trade-offs

in total model bits or predictive performance are desired, keep the precision in 4-bit but vary the number of parameters of the model instead.

While earlier work has shown that it is possible to significantly improve the predictive performance of quantized models by using outlier-dependent quantization [Dettmers et al., 2022a; Xiao et al., 2022], we show that this is not effective for bit-level scaling. In particular, we analyze instabilities in 3-bit OPT and Pythia models and show that while these models can be stabilized through an outlier-dependent quantization which we term *proxy quantization*, this does not improve bit-level scaling compared to 4-bit precision. On the other hand, we highlight that one-shot quantization methods, that is, methods that optimize the quantization through a single mini-batch of data, potentially could be scaled to be bit-efficient below the 4-bit level. Overall, these findings suggest that the most promising directions to improve zero-shot bit-level scaling laws are to develop new data types and techniques that quantize outliers with high precision without requiring a significant amount of additional bits. We also highlight the potential for one-shot quantization methods as a way towards low-bit transformers if combined with our insights.

3.2 Background

It might be unintuitive that reducing the number of bits of a model is directly related to inference latency for LLMs. The following section gives the background to understand this relationship. Afterward, we provide a background on quantization data types and methods.

3.2.1 Relationship Between Inference Latency and Total Model Bits

While the main goal of our work is to find the best trade-offs between model bits and zero-shot accuracy for LLMs, the total model bits are also strongly related to inference latency. The overall computation latency – the time it takes from start to finish of a computation – is mainly determined by two factors: (1) how long does it take to load the data from main memory into caches and registers, (2) how long does it take to perform the computation. For example, for modern hardware like GPUs, it usually takes more than 100 times longer to load a number than to do an arithmetic operation with that number [Jia et al., 2019a; Dongarra, 2022]. Therefore, reducing the time spent loading data from main memory is often the best way to accelerate overall computation latency. Such reductions can be achieved mainly through caching and

lower precision numbers.

Caching can reduce the overall latency of matrix multiplication by a factor of 10x or more [Jia et al., 2019a], given that neither matrix entirely fits into the L1 cache of the device. In matrix multiplication, $\mathbf{bW} = \mathbf{h}$, with the batch $\mathbf{b}$ and parameter matrix $\mathbf{W}$, we load each row of $\mathbf{b}$ and each column of $\mathbf{W}$ and multiply them – thus multiple loads from global memory occur for each row/column which can be cached. If $\mathbf{b}$ entirely fits into the L1 cache, no reuse is possible because neither $\mathbf{W}$ nor $\mathbf{b}$ is loaded more than once from global memory. This occurs if the inference batch size is below 60 or 200 for an RTX 3090 or RTX 4090 GPU. As such, caching is ineffective below these batch sizes, and the inference latency is solely determined by the memory loaded from $\mathbf{W}$.

Thus in the case where the mini-batch fits into the L1 cache, inference latency can be reduced by using a smaller model with smaller $\mathbf{W}$ or by compressing an existing model to use a lower bit-precision per parameter in $\mathbf{W}$. For example, beyond their algorithmic innovation of improved rounding for quantization, Frantar et al. [2022a] also developed inference CUDA kernels for 16-bit inputs and 3-bit integer weights, which yields inference latency improvements of up to 4.46x compared to 16-bit inputs and weights for OPT-175B – close to the 5.33x reduction in model bits. As such, reduction in the total model bits is strongly correlated with inference latency for small inference batch sizes.

3.2.2 Data types

Here we provide a brief overview of the data types that we study.

We use four different data types. For **Integer** and **Float** data types, use IEEE standards. Our Float data type has an exponent bias of 2^{E-1}, where E is the number of exponent bits. We also use **quantile quantization**, a lossy maximum entropy quantization data type [Dettmers et al., 2022b], where each quantization bin holds an equal number of values. This ensures that each bit pattern occurs equally often. Finally, we use **dynamic exponent quantization** [Dettmers, 2016], which uses an indicator bit to separate an exponent bit region and a linear quantization region. The exponent can vary from value to value by shifting the indicator bit. This data type has low quantization error for tensors which have numbers that vary by many orders of magnitude.

3.2.3 Blocking / Grouping

Quantization precision is, in part, determined by whether all quantization bins are used equally. For example, a 4-bit data type has 16 bins, but if, on average, only 8 bins are used, it is equivalent to a 3-bit data type. As such, methods that help to increase the average use of all quantization bins in the data type can increase quantization precision. In this subsection, we introduce blocking. Blocking/grouping methods chunk the tensor into smaller pieces and quantize each block independently. This helps to confine outliers to particular blocks, which increases the average number of bins used across other blocks and thus increases the average quantization precision.

Blocking/Grouping. Blocking and grouping are similar concepts. In grouping, we sub-divide a tensor along a certain dimension into n parts and assign each sub-tensor, called group, its own normalization constant c, which is usually the absolute maximum of the group. In blocking, we view the tensor as a one-dimensional sequence of values and divide this sequence into parts of size n called blocks.

In our work, we use blocking because, unlike grouping, it provides a measure of additional bits per parameter independent of the hidden dimension. For example, using 16-bit normalization constants and a block size of 64 means, we have an extra 16 bits every 64 parameters or 16/64=0.25 bit-per-parameter additional cost for using block-wise quantization – this is true for every model regardless of hidden dimension. For grouping, the exact cost would depend on the size of the hidden dimension of each model.

For block-wise quantization, we use the notation of Dettmers et al. [2022b], which defines the block-wise quantization with k bits, block-size B, input tensor $\mathbf{T}$ with n elements, n/B blocks as follows. If $\mathbf{Q}_k^{\text{map}}(\cdot)$ maps the integer representation of a data type to the representative floating point value, for example, the bit representation of a 32-bit float to its real value, and if we define the index of each block in $0..n/B$ by index b, and we compute the normalization constant as $m_b = \max(|\mathbf{T}_b|)$, then block-wise quantization can be defined by finding the minimum distance to the value of the quantization map as follows:

$$\mathbf{T}_{bi}^{Q_k^{\text{map}}} = \underset{j=0}{\overset{n=2^k}{\arg\min}} |\mathbf{Q}_k^{\text{map}}(j) - \frac{\mathbf{T}_{bi}}{m_b}| \Big|_{0<i<B}, \tag{3.1}$$

3.3 Outlier-dependent Quantization Through Proxy Quantization

Outlier features that emerge in large language models [Gao et al., 2019; Timkey and van Schijndel, 2021; Bondarenko et al., 2021; Wei et al., 2022; Luo et al., 2021; Kovaleva et al., 2021; Puccetti et al., 2022] can cause large quantization errors and severe performance degradation [Dettmers et al., 2022a; Zeng et al., 2022; Xiao et al., 2022]. While it has been shown that it is sufficient to use 16-bit inputs and 8-bit weights to avoid this disruption [Zeng et al., 2022], it is unclear if outlier features can cause degradation if we use 16-bit inputs and weights below 8-bit precision.

To this end, we develop outlier-dependent quantization through proxy quantization, where we quantize weights to a higher precision for the corresponding outlier feature dimensions to test how much precision is needed for the weights. A significant challenge is that each model has a different number of outlier features, and outliers partially depend on inputs that are different for each zero-shot task. As such, we seek a model-independent model that has a constant memory footprint across all models and tasks.

In initial experiments, we noted that the criterion developed by Dettmers et al. [2022a], which thresholds the hidden states to detect outlier features, is unreliable as it depends on the standard deviation of the hidden state. This causes problems because, for models such as OPT, the standard deviation of the hidden states increases in later layers, which causes too many outliers to be detected. This also has been noted by Zeng et al. [2022]. By inspecting this anomaly in OPT models, we find that a better measure of detecting outlier dimensions is the standard deviation of the weights of each hidden unit of the previous layer. The standard deviation of hidden unit weights that produce outliers are up to 20x larger than the standard deviation of other dimensions.

With this insight, we develop what we call **proxy quantization**. Proxy quantization is input-independent and, therefore task-independent, as it uses the standard deviation of each layer's hidden unit weights as a proxy for which dimensions have outlier features. For example, given a transformer with n linear layers (FFN and attention projection layers) with weight matrices $\mathbf{W}_i \in \mathbb{R}^{h \times o}$ where h is the input dimension and o the output dimension (thus o hidden units), we define the set of indices J to be quantized in higher precision by:

$$J_{i+1} = \arg\max_{j=0}^{k} \text{std}(\mathbf{W}_i)\big|_{i..n} \tag{3.2}$$

where std$(\cdot)$ is the standard deviation of the output dimension o. We then quantize the input dimensions of the weight of the next layer in 16-bit if it is in set J and k-bit otherwise.

3.4 Experimental Setup

In our experiments, we use 16-bit inputs and k-bit quantized parameters for $3 \geq k \geq 8$. Attention matrices are not quantized since they do not contain parameters. We also use a 16-bit baseline that does not use any quantization (16-bit floats).

To measure inference performance for k-bit quantization methods, we use perplexity on the CommonCrawl subset of The Pile [Gao et al., 2020] and mean zero-shot performance on the EleutherAI LM Evaluation harness [Gao et al., 2021]. In particular, for the zero-shot setting, we use the EleutherAI LM eval harness [Gao et al., 2021] in the GPT-2 setting on the tasks LAMBADA [Paperno et al., 2016], Winogrande [Sakaguchi et al., 2021], HellaSwag [Zellers et al., 2019], and PiQA [Bisk et al., 2020].

The choice of these particular zero-shot tasks was mainly motivated by previous work [Dettmers et al., 2022a; Yao et al., 2022; Xiao et al., 2022]. However, in our evaluation, we find that perplexity is a superior metric since its continuous value per sample leads to less noisy evaluations. This has also been noted by Frantar et al. [2022a]. For example, when using perplexity to evaluate data types, quantile quantization is the best data type. Still, when we use zero-shot accuracy as an evaluation metric, the float data type is sometimes better because zero-shot accuracy is noisier. Furthermore, we find that across more than 35,000 zero-shot experiments, the **Pearson correlation coefficient between The Pile Common Crawl perplexity and zero-shot performance is -0.94**.

This highlights that perplexity is sufficient and preferable for evaluation purposes. A serious drawback is that perplexity is challenging to interpret. As such, we use zero-shot accuracies in the main paper for clarity but encourage the reader to use perplexity measures. Since perplexity evaluations are so reliable, it is possible to replicate our work by evaluating a small number of samples using the perplexity metric, which makes the construction of scaling laws less computationally intensive.

Scaling laws. We try to fit power laws to our data, but we find that bivariate power functions with respect to the number of parameters and the bit-precision provide a poor fit. However, when we fit linear interpolations to represent scaling curves, we find that different bit-precisions are almost parallel, indicating

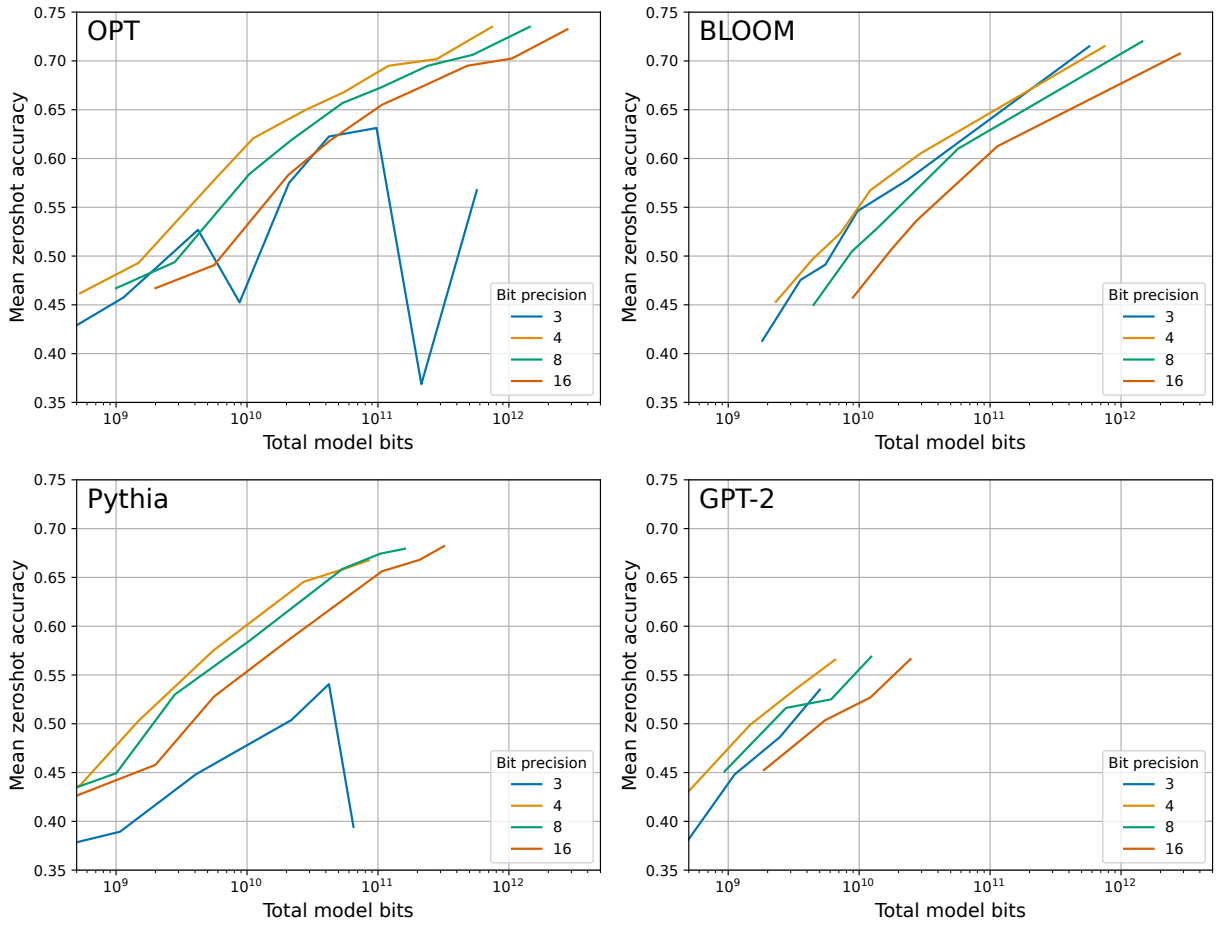


Figure 3.2: Bit-level scaling laws for mean zero-shot performance across Lambada, PiQA, Winogrande, and Hellaswag. 4-bit precision is optimal for almost all models at all scales, with few exceptions. While lowering the bit precision generally improves scaling, this trend stops across all models at 3-bit precision, where performance degrades. OPT and Pythia are unstable at 3-bit precision, while GPT-2 and BLOOM remain stable.

that the scaling trends of different precisions can be represented faithfully by a base function and an offset for each bit-precision. As such, we choose to use linear interpolations to represent scaling trends.

3.5 Results & Analysis

3.5.1 Bit-level Inference Scaling Laws

The main results are shown in Figure 3.2, which depicts the mean zero-shot accuracy over Lambada, PiQA, HellaSwag, and Windogrande, given the total number of bits for OPT, BLOOM, Pythia, and GPT-2 for 3 to

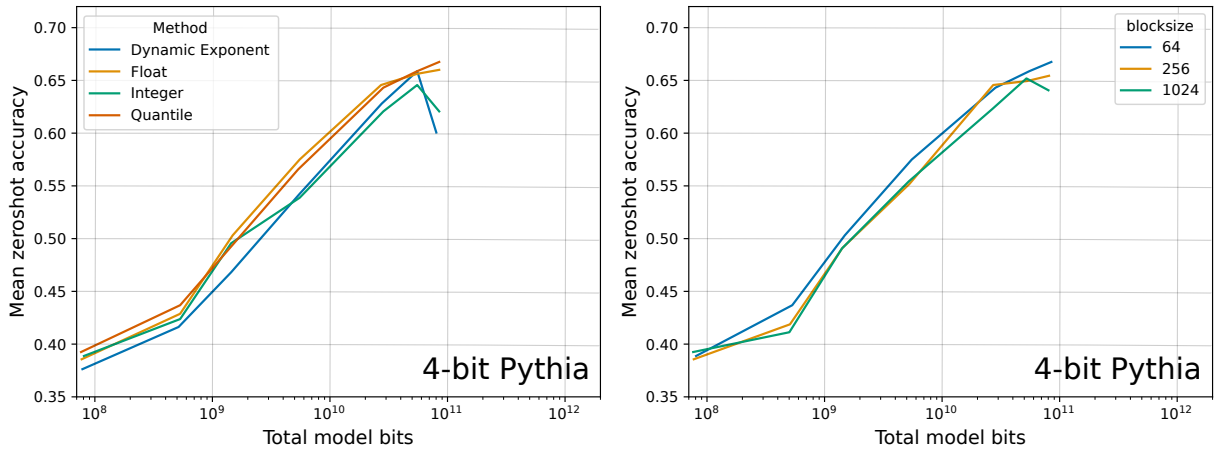


Figure 3.3: Bit-level mean zero-shot scaling laws for 4-bit Pythia models for different data types and block sizes. Block size and data types yield the largest improvements in bit-level scaling. For Pythia, a small block size adds only 0.25 bits per parameter but provides an improvement similar to going from 4-bit to 5-bit precision. In general, across all models, the float and quantile data types yield better scaling than int and dynamic exponent quantization.

16-bit parameters.

We make the following observations:

- For a given zero-shot performance, 4-bit precision yields optimal scaling for almost all model families and model scales. The only exception is BLOOM-176B where 3-bit is slightly but not significantly better.

- Scaling curves are almost parallel, which indicates that bit-level scaling is mostly independent of scale. An exception to this is 3-bit quantization.

- Pythia and OPT are unstable for 3-bit inference where performance is close to random (35%) for the largest Pythia/OPT models.

3.5.2 Improving Scaling Laws

Given the main scaling results in Figure 3.2, an important follow-up question is how we can improve the scaling trends further. To this end, we run more than 35,000 zero-shot experiments to vary many of the recently studied advances in quantization precision, including the underlying data types, quantization block size, and outlier-dependent quantization.

These methods usually improve the quantization error at a small cost of additional bits. For example,

a block size of 64 with 16-bit quantization constants uses 16 extra bits for every 64 parameters, or 16/64 = 0.25 additional bits per parameter. Outlier-dependent quantization stores the top $p\%$ of weight vectors in 16-bit precision and increases the bits per parameter by $p(16 - k)$, where k is the precision of the regular weights. For example, for $p = 0.02$ and $k = 4$, the additional memory footprint is 0.24 bits per parameter.

No scaling improvements for 6 to 8-bit precision. We combine all possible combinations of quantization methods (data types, blocking) with 6 to 8-bit quantization, and we find that none of these methods improve bit-level scaling. It appears that for 6 to 8-bit precision, the model parameters have enough precision to not cause any significant performance degradation compared to 16-bit weights. As such, scaling behavior can only be improved by using less than 6-bit precision rather than enhancing the quantization precision through other means.

Small block size improves scaling. For 3 to 5-bit precision, we do see improvements in scaling for various quantization methods. Figure 3.3 shows that for 4-bit Pythia models, considerable improvements in bit-level scaling can be achieved by using a small block size. To put this improvement into perspective: Going from a block size of 1024 to 64 adds 0.24 bits per parameter but improves zero-shot accuracy almost as much as going from 4 to 5-bit precision. As such, using a small block size adds a few extra bits compared to improving zero-shot accuracy for 4-bit precision. Besides Pythia, GPT-2 models improve by a large degree. BLOOM, BLOOMZ, and OPT models improve significantly, but less in magnitude compared to Pythia and GPT-2 – this relationship likely arises from emergent outlier features. For 5-bit models, the improvement of using small block sizes is minor but still significant. Small block sizes improve 3-bit scaling considerably but still do not make it competitive with 4-bit precision scaling.

Data types improve scaling. From Figure 3.3, we see that data types improve scaling trends for 4-bit Pythia. In particular, the quantile quantization and float data types provide better scaling than integer and dynamic exponent quantization. We generally find that quantile quantization is the best data type across all models, scales, and precisions. The float data type seems to be superior to Integer quantization with a few exceptions: Integer quantization is better than float quantization for 5-bit – it appears since the float data type is quite dependent on the balance between exponent and fraction bits, the float data type can be better or worse depending on the particular bit precision.

Outlier-dependent quantization improves stability, but not scaling. Finally, we noticed the 3-bit

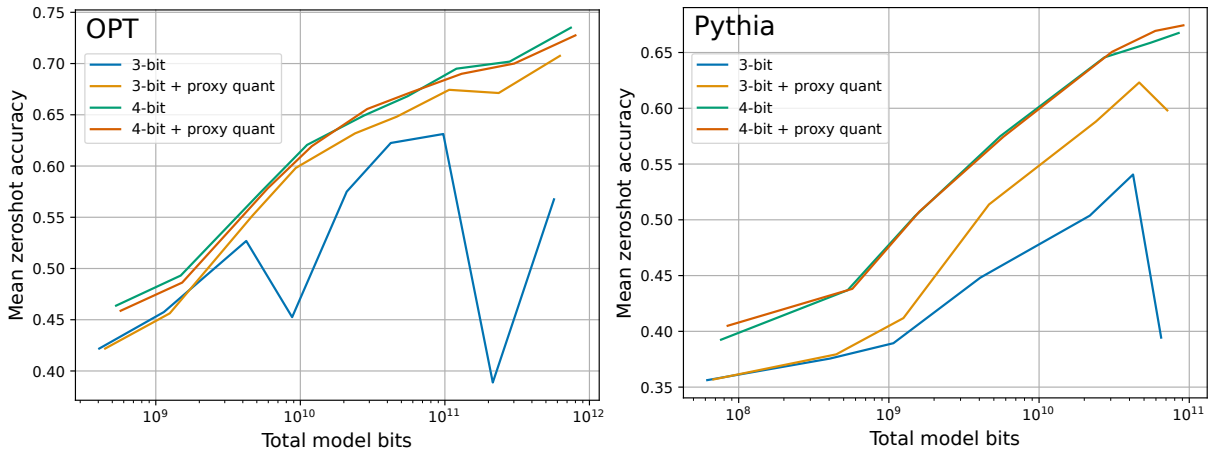


Figure 3.4: Bit-level scaling laws for outlier dependent quantization for OPT and Pythia. While proxy quantization removes instabilities and improves the 3-bit precision scaling of OPT and Pythia, it still scales worse than 4-bit precision. Proxy quantization is only useful for 3-bit precision weights. Proxy quantization is not useful for models that are relatively stable such as 3-bit BLOOM and GPT-2.

instabilities in OPT and Pythia, and we relate them to emergent outlier features [Dettmers et al., 2022a; Xiao et al., 2022]. If we use proxy quantization to quantize the 2% most significant outlier dimensions to 16-bit instead of 3-bit, we can increase stability and overall scaling for 3-bit Pythia and OPT models. This is shown for OPT in Figure 3.4 (left). However, despite this improvement, 4-bit precision still provides better scaling. For 4-bit precision, outlier-dependent quantization has no scaling benefit, as shown in Figure 3.4 (right), which means 4-bit precision is optimal despite the considerable improvements for 3-bit precision when using proxy quantization. As such, it appears the outlier features do not require more than 4-bit precision weights to provide optimal bit-level scaling.

3.6 Related Work

Large language model quantization. The most closely related work is on large language model (LLM) quantization for models with more than a billion parameters. Compared to smaller models, LLM quantization poses some unique challenges, such as emergent outliers [Dettmers et al., 2022a; Zeng et al., 2022; Xiao et al., 2022] and optimized low-bit inference for LLMs Frantar et al. [2022a]; Park et al. [2022]; Yao et al. [2022]. One major defining factor between approaches is zero-shot quantization methods that directly quantize a model without any additional information and one-shot quantization methods that need a mini-

batch of data for quantization. While one-shot methods are more accurate, such as GPTQ, which optimizes the rounding during quantization via a mini-batch of data [Frantar et al., 2022a], they are also more complex and may require hours of optimization before a model can be used. On the other hand, the advantage of zero-shot methods is that they can be used immediately, which makes them easy to use, but zero-shot quantization methods often fail at lower precisions.

Quantization methods. Another aspect related to our work are quantization methods which can be grouped into specific categories. For example, there are methods associated with blocking and grouping Park et al. [2022]; Wu et al. [2020]; Jain et al. [2020]; Nagel et al. [2019]; Krishnamoorthi [2018]; Rusci et al. [2020], centering [Krishnamoorthi, 2018; Jacob et al., 2017], learned data types that are found through clustering [Gong et al., 2014; Han et al., 2015; Choi et al., 2016; Park et al., 2017], or direct codebook optimization [Rastegari et al., 2016; Hou et al., 2016; Leng et al., 2018; Zhang et al., 2018]. While our work studies grouping and blocking, we only study one data type that groups similar weights through their quantiles of the entire input tensor [Dettmers et al., 2022b]. While we do not study learned data types in depth, we are the first work that shows that these are critical for improving bit-level scaling for LLMs.

Scaling Laws for Inference. Early work in scaling laws highlighted the importance of studying how variables change with scale since scale is one of the best predictors of model performance [Kaplan et al., 2020; Rosenfeld et al., 2019; Hestness et al., 2017]. Particularly, for inference, there has been work that studies scaling trends of zero-shot performance for 4-bit vs. 16-bit models [Zeng et al., 2022]. We study precisions from 3 to 16-bit and disentangle the factors that improve scaling. Work by Pope et al. [2022] looks at scaling inference in a production setting where large batch sizes are common. While they only study quantization rudimentary, they disentangle factors that lead to better model FLOPS utilization (MFU). Since reducing the bit-precision of bits loaded leads to higher MFU, it is similar to our approach to studying bit-level scaling. The main difference is that we vary the bit-width of models and study small batch sizes that are common for consumers and small organizations.

Table 3.1: WikiText-2 perplexity for 2-bit GPTQ and 3-bit Float with blocking. We can see that GPTQ is superior to 3-bit Float if blocking is used. As such, methods like GPTQ are a promising way to improve low-bit scaling. However, GPTQ requires blocking to provide good scaling (see Figure 3.5).

	WikiText-2 Perplexity	
Blocksize	2-bit GPTQ	3-bit Float
1024	**11.84**	13.26
256	**10.00**	10.38
64	**9.18**	9.99

3.7 Recommendations & Future Work

We make the following **recommendations**:

- By default, use 4-bit quantization for LLM inference as it offers the total model bits and zero-shot accuracy trade-offs.
- Use a block size of 128 or lower to stabilize 4-bit quantization and improve zero-shot performance.
- Use a floating point or quantile quantization data type. In some cases, integer data types might be preferable to improve inference latency depending on the implementation and hardware support.

A case where a higher than 4-bit precision is desirable is when one works with a GPU with enough memory to hold a higher bit precision but not a larger model. For example, a 48 GB GPU has enough memory to use a 66B model in 5-bit precision but cannot fit a 175B model in 4-bit. Therefore, if maximal zero-shot accuracy is desired, 5-bit precision and a 66B model is preferable for this scenario.

Promising directions for future work. Our results highlight that 4-bit precision is currently bit-by-bit the most efficient precision, but we also show that 3-bit scaling can be significantly improved. As such, a promising research direction is to focus on low-bit precisions below 4-bit and improve their scaling trends. It has been shown that one-shot quantization methods, like GPTQ, which use an input sample for optimizing the quantization are more effective at low-bit precisions [Frantar et al., 2022a]. Table 3.1 shows that 2-bit GPTQ with blocking yields better performance than zero-shot 3-bit Float. This highlights that one-shot methods are very promising for low-bit precisions.

On the other hand, Figure 3.5 also shows that 3-bit GPTQ without blocking scales worse compared to 3-bit Float with a blocksize of 64 and 4-bit GPTQ without blocking yields similar scaling compared to 4-bit Float with a blocksize of 64. From these results, it appears that the insights gained from our zero-shot

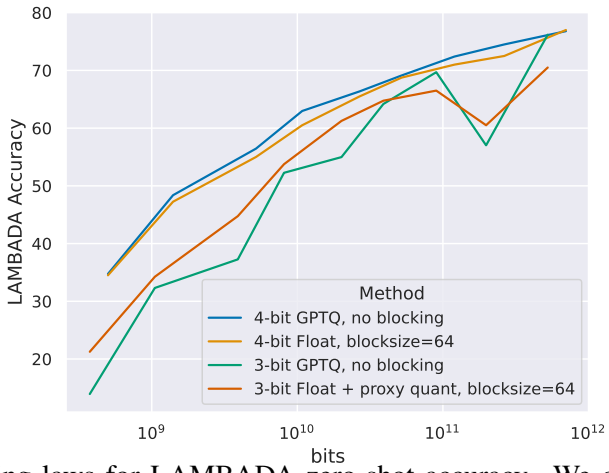


Figure 3.5: Bit-level scaling laws for LAMBADA zero-shot accuracy. We can see that GPTQ without blocking scales poorly at 3-bit. While one-shot quantization methods like GPTQ are superior to zero-shot methods like proxy quantization, other methods like blocking are required to make them bit-level efficient (see Table 3.1).

scaling laws translate to one-shot quantization methods. This shows that zero-shot quantization research is well suited for disentangling and understanding individual factors in quantization scaling, while one-shot methods maximize performance. In contrast, current one-shot methods are expensive to study. For example, repeating our scaling experiments for GPTQ only for the OPT-175B and BLOOM-176B models would consume an estimated 5,120 GPU days of compute. Therefore, the combination of zero-shot quantization scaling insights and one-shot quantization methods may yield the best future methods.

One challenge unaddressed in our work is that the data type should also be able to be used efficiently on a hardware level. For example, while quantile quantization is the data type that shows the best scaling trends, it requires a small lookup table that is difficult to implement in a highly parallel setup where parallel memory access often leads to poor performance due to the serialization. This problem can be solved by designing new data types that are both bit-level scaling efficient and hardware efficient. A different approach to this problem would be hardware design: Can we design hardware to use data types such as quantile quantization efficiently?

Both block-size and outlier-dependent quantization improve the quantization precision of outliers. While outlier-dependent quantization does not offer improvements in scaling, it is reasonable that there are unknown quantization methods that help with outliers and improve scaling trends simultaneously. As such, another primary focus of future research should center around preserving outlier information while mini-

mizing the use of additional bits.

3.8 Discussion & Limitations

While we ran more than 35,000 experiments, a main limitation is that we did not consider certain classes of quantization methods. For example, there are quantization methods where a data type is optimized with additional input data [Rastegari et al., 2016; Frantar et al., 2022a] or from the weights of the model alone [Gong et al., 2014]. Optimization from the weights alone is similar to quantile quantization which was the most effective data type in our study. As such, this hints that such quantization methods could improve scaling for inference and present a missed opportunity to be included in our study. However, our study is an essential step towards recognizing the importance of these methods for optimizing the model-bits-accuracy trade-off – a perspective that did not exist before.

Another limitation is the lack of optimized GPU implementations. It is unclear if other data types that rely on lookup tables can achieve significant speedups. However, efficient implementations for our Int/Float data types would be possible, and our results for other data types are still useful for the development of future data types, which yield strong scaling and efficient implementations.

While we only study the latency-optimal perspective indirectly through studying the model-bits-accuracy trade-off, a practical limitation of the latency-optimal perspective is that low-bit models with 16-bit inputs might be less latency efficient if such a model is deployed to be used by many users [Pope et al., 2022]. Given a busy deployed API with thousands of requests per second, the large mini-batches would no longer fit into the cache. This means bit-level scaling laws would be increasingly unrelated to inference latency in this case. For such high throughput systems, scaling laws that model both low-bit weights *and* low-bit inputs are required to study optimal inference latency scaling. In short, our scaling laws are only valid for cases where the mini-batch does not fit into the L1 cache of the device, and beyond this, a new set of scaling laws is required.

A final limitation is that loading the weight matrix is only one part of inference latency which needs to be optimized to achieve fast inference. For example, without optimizations for the attention operations, multi-head attention can be a large chunk of the inference latency footprint [Jaszczur et al., 2021; Pope et al., 2022]. However, the overall memory footprint of the model is still reduced, making large language models

more easily usable when GPU memory is limited.

3.9 Discussion & Limitations

While we ran more than 35,000 experiments, a main limitation is that we did not consider certain classes of quantization methods. For example, there are quantization methods where a data type is optimized with additional input data [Rastegari et al., 2016; Frantar et al., 2022a] or from the weights of the model alone [Gong et al., 2014]. Optimization from the weights alone is similar to quantile quantization which was the most effective data type in our study. As such, this hints that such quantization methods could improve scaling for inference and present a missed opportunity to be included in our study. However, our study is an essential step towards recognizing the importance of these methods for optimizing the model-bits-accuracy trade-off – a perspective that did not exist before.

Another limitation is the lack of optimized GPU implementations. It is unclear if other data types that rely on lookup tables can achieve significant speedups. However, efficient implementations for our Int/Float data types would be possible, and our results for other data types are still useful for the development of future data types, which yield strong scaling and efficient implementations.

While we only study the latency-optimal perspective indirectly through studying the model-bits-accuracy trade-off, a practical limitation of the latency-optimal perspective is that low-bit models with 16-bit inputs might be less latency efficient if such a model is deployed to be used by many users [Pope et al., 2022]. Given a busy deployed API with thousands of requests per second, the large mini-batches would no longer fit into the cache. This means bit-level scaling laws would be increasingly unrelated to inference latency in this case. For such high throughput systems, scaling laws that model both low-bit weights *and* low-bit inputs are required to study optimal inference latency scaling. In short, our scaling laws are only valid for cases where the mini-batch does not fit into the L1 cache of the device, and beyond this, a new set of scaling laws is required.

A final limitation is that loading the weight matrix is only one part of inference latency which needs to be optimized to achieve fast inference. For example, without optimizations for the attention operations, multi-head attention can be a large chunk of the inference latency footprint [Jaszczur et al., 2021; Pope et al., 2022]. However, the overall memory footprint of the model is still reduced, making large language models

more easily usable when GPU memory is limited.

Chapter 4

SpQR: A Sparse-Quantized Representation for Near-Lossless LLM Weight Compression

4.1 Introduction

Pretrained large language models (LLMs) improved rapidly from task-specific performance [Wang et al., 2018; Devlin et al., 2018; Radford et al., 2019], to performing well on general tasks if prompted with instructions [Brown et al., 2020b; Wei et al., 2021; OpenAI, 2023]. While the improved performance can be attributed to scaling in training data and parameters [Kaplan et al., 2020; Chowdhery et al., 2022b] recent trends focused on smaller models trained on more data, that are easier to use at inference time [Hoffmann et al., 2022a; Biderman et al., 2023; Touvron et al., 2023]. For example, the 7B parameter LLaMA model trained on 1 trillion tokens achieved an average performance only slightly lower than GPT-3 [Brown et al., 2020b] despite being 25x smaller. Current techniques for LLM compression can shrink these models further by a factor of about 4x, while preserving their performance [Dettmers et al., 2022a; Xiao et al., 2022; Frantar et al., 2022a; Dettmers and Zettlemoyer, 2022]. This yields performance levels comparable to the largest GPT-3 model, with major reductions in terms of memory requirements. With such improvements, well-performing models could be efficiently served on end-user devices, such as laptops.

The main challenge is to compress models enough to fit into such devices while also preserving generative quality. Specifically, studies show that, although accurate, existing techniques for 3 to 4-bit quantization still lead to significant accuracy degradation [Dettmers and Zettlemoyer, 2022; Frantar et al., 2022a]. Since LLM generation is sequential, depending on previously-generated tokens, small errors can accumulate and lead to severely diverging outputs. To ensure reliable quality, it is critical to design quantization that does not degrade predictive performance compared to the 16-bit model.

In this work, we introduce Sparse-Quantized Representations (SpQR), a hybrid sparse-quantized format which can be used to compress accurate LLMs to 3-4 bits per parameter while staying *near-lossless*: specifically, SpQR is the first weight quantization method which is able to reach high compression ratios while inducing end-to-end error of less than 1% relative to the dense baseline. This 1% relative threshold follows the MLCommons standards [Reddi et al., 2020]; since LLM accuracy is usually measured in the stringent *perplexity* metric, this is very challenging to achieve.

Our compression approach is motivated by a new analysis showing that LLM weight quantization errors exhibit both vertical and horizontal group correlations, corresponding to systematic large errors corresponding to input and output dimensions. While outlier input features have been observed before [Dettmers et al., 2022a; Xiao et al., 2022], our work is the first to demonstrate that similar outliers occur *in the weights, for particular output hidden dimensions*. Unlike input feature outliers, the output hidden dimension outliers occur only in small segments for a particular output hidden dimension.

In SpQR, we resolve the difficulties posed by these correlations by combining two ideas. First, we isolate *outlier weights*, whose quantization induces disproportionately high errors: these weights are kept in high precision, while the other weights are stored in lower bitwidth. Second, we implement a variant of grouped quantization with very small group size, e.g., 16 contiguous elements, but we show that one can quantize the quantization scales themselves to a 3-bit representation.

To convert a given pretrained LLM into SpQR format, we extend the post-training quantization (PTQ) approach recently introduced by GPTQ [Frantar et al., 2022a]. Specifically, this method passes calibration data through the uncompressed model; to compress each layer, it applies a layer-wise solver with respect to the L2 error between the outputs of the uncompressed model, and those of the quantized weights. Our technique splits this process into two steps: an "outlier detection" step, in which we isolate weights whose

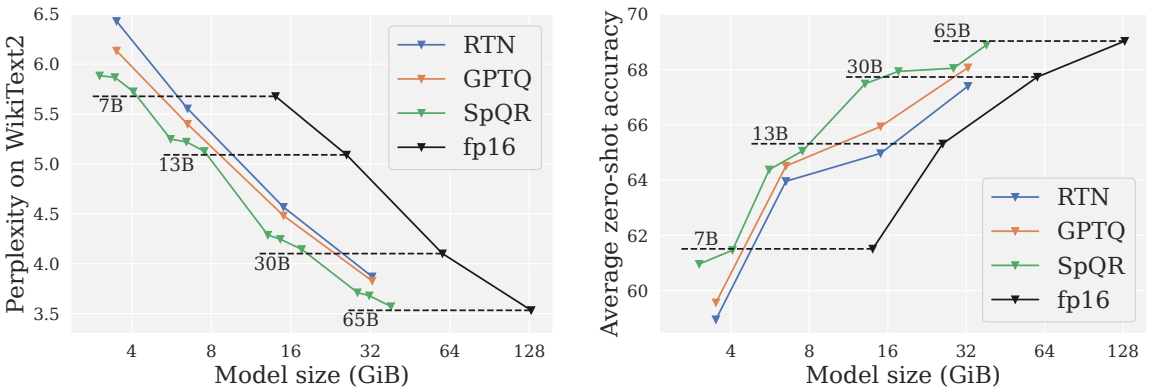


Figure 4.1: Compressed LLM performance for LLaMA models. (**left**) LM loss on WikiText2 vs model size. (**right**) Average performance on zero-shot tasks vs model size.

direct quantization has outsize impact on layer output behavior, and a compression step, in which $\geq 99\%$ of weights are compressed to low-bitwidth. Furthermore, the whole representation is rendered more efficient by compressing the quantization metadata. Our quantization algorithm isolates such outliers and efficiently encodes a given model in SpQR format.

One key challenge in exploiting the SpQR format is computational efficiency, as sparse representations generally have poor GPU support. We circumvent this issue by developing a sparse-matrix multiplication algorithm that is specialized to our format. To use SpQR for generative inference, we combine sparse-matrix multiplication for the outliers with dense-quantized matrix multiplication for the 3-4 bit "base" weights. Our experimental evaluation, across LLMs from the LLaMA [Touvron et al., 2023], Falcon [TII UAE, 2023a], and OPT [Zhang et al., 2022] families, shows that SpQR provides state-of-the-art trade-offs in terms of accuracy versus model size (see Figure 4.1 for an illustration). For instance, SpQR can provide 3.4x compression without any degradation in perplexity, while also being 20-30% faster for LLM generation compared to 16-bit inference.

4.2 Related Work

We focus our discussion on related *post-training quantization (PTQ) methods* [Nagel et al., 2020], referring the reader to the recent survey of Gholami et al. [2021] for full background on quantization. PTQ is popular for *one-shot compression* of models with various sizes, based on a limited amount of calibration data, using accurate solvers. Most PTQ methods, such as AdaRound [Nagel et al., 2020], BitSplit [Wang et al., 2020], AdaQuant [Hubara et al., 2021], BRECQ [Li et al., 2021], or OBQ [Frantar et al., 2022b] were designed for

vision or small-scale language models. These recent approaches tend to use accurate solvers, which would not scale to LLMs in terms of computational or memory cost, as they are 10-1000x larger in size.

Recently, there has been significant interest in obtaining accurate post-training methods that scale to such massive models. Due to computational constraints, early work such as ZeroQuant [Yao et al., 2022], LLM.int8() [Dettmers et al., 2022a], and nuQmm [Park et al., 2022] used direct rounding of weights to the nearest quantization level, while customizing the quantization granularity (i.e., group size) to trade off space for increased accuracy. LLM.int8() [Dettmers et al., 2022a] suggested isolating "outlier features" which would be quantized separately to higher bit-width. These approaches are able to induce relatively low error, e.g., 5.5% relative LM Loss increase for LLaMA-7B at 4-bit weight quantization, provided that the quantization granularity is low enough. GPTQ [Frantar et al., 2022a] proposed a higher-accuracy approach (e.g., 4% LM Loss increase in the above setting), which works via an approximate large-scale solver for the problem of minimizing the layer-wise squared error.

Dettmers and Zettlemoyer [2022] provided an in-depth overview of the accuracy-compression trade-offs underlying these methods, establishing that 4-bit quantization is an optimal point for round-to-nearest-based methods, whereas higher compression can be achieved via data-aware methods such as GPTQ. SparseGPT [Frantar and Alistarh, 2023] presented an approach to jointly sparsify LLM weights to medium sparsities, together with quantization of the remaining weights to a fixed given bit-width. One common drawback of existing methods is that the accuracy loss relative to the original model is still significant (see Section 4.5). This is especially relevant to relatively small but easily deployable models, e.g., in the 7-13B parameter range, where existing methods show drastic accuracy drops. We investigate this question here, and provide a new compression format which can lead to near-lossless 3-4 bits compression in this regime.

A related question is performing both activation and weight quantization. Dettmers et al. [2022a]; Xiao et al. [2022]; Yao et al. [2022] showed that both activations and weights can be quantized to 8-bits.These investigations yield insights into the sources of quantization error, in particular, the former two observe "outlier features" with significantly higher magnitudes in the input/output of large LLMs, which induce higher quantization error, and propose different mitigation strategies.

We analyze this phenomenon from the point of view of weight quantization that goes beyond outlier features in the hidden state. While we find that input feature outliers of the current layer are correlated

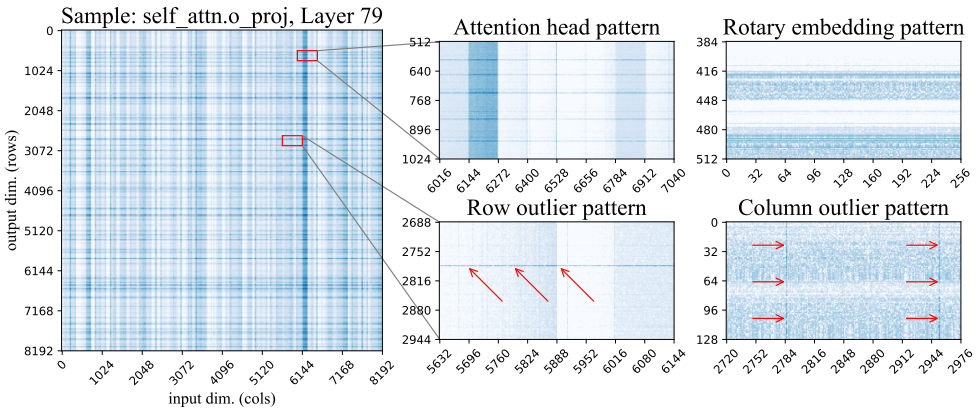


Figure 4.2: Weight log-sensitivities from the last attention layer of LLaMA-65B. Dark-blue shades indicate higher sensitivity. The image on the left is a high-level view, resized to 1:32 scale with max-pooling. The two images in the middle are zoomed in from the main figure. The two images are zoomed in sensitivities of weights from to other weight matrices.

to hidden unit outliers weight in the previous layer there is not a strict correspondence. Such partially-structured outlier patterns necessitate a fine-grained hybrid compression format that goes beyond algorithms that exploit the column structure of outlier features found in previous work.

4.3 Quantization sensitivity of LLM weights

4.3.1 Parameter sensitivity under quantization

Not all parameters in a neural network are equally important. Intuitively, a weight could be seen as sensitive to quantization if its rounding error is large, i.e., it is not close to a quantization point, and/or the inputs it is usually multiplied with are large, amplifying even a small rounding error. These simple notions of sensitivity however disregard the fact that LLMs operate on very large vectors with significant correlations: a weight w_a may have a large rounding error while being strongly correlated to another weight w_b, meaning that the error of rounding up w_a can be well compensated by rounding down w_b. This idea is exploited by modern quantization algorithms [Frantar et al., 2022a; Yao et al., 2022] and can lead to major improvements over vanilla rounding, especially a low bitwidths. Properly capturing this aspect of sensitivity requires a more robust definition.

For computational tractability, we assess sensitivity on a per-layer level using a small set of *calibration inputs* X, collected by running them through the model up to the particular layer. We define the sensitivity s_{ij} of some weight w_{ij} in the layer's weight matrix W as the minimum squared difference between the

original predictions on X and those of any weight matrix W' where this weight is quantized, i.e., $w'_{ij} = \text{quant}(w_{ij})$: $s_{ij} = \min_{W'} ||WX - W'X||_2^2 \quad \text{s.t.} \quad w'_{ij} = \text{quant}(w_{ij})$

Crucially, all weights of W' except for w'_{ij} may take on arbitrary, not necessarily quantized, values in order to compensate for the quantization error incurred by rounding w_{ij}, thus capturing the correlation aspect discussed above. Further, as we allow continuous values, this problem admits a closed-form solution: $s_{ij} = \frac{(w_{ij} - \text{quant}(w_{ij}))^2}{2(XX^\top)^{-1}}$, as per the Optimal Brain Surgeon framework [Frantar et al., 2022b].

This saliency measure can be approximated efficiently by quantization solvers, such as GPTQ [Frantar et al., 2022a]. In more detail, GPTQ quantizes weight matrices column-by-column while in each step adjusting the not-yet-quantized part to compensate for the quantization error in a similar sense as defined above. Consequentially, instead of statically deciding all sensitivities in advance, they can be computed dynamically as the algorithm processes each column, by using the inverse of the Hessian subselection corresponding to all not yet quantized weights. This matrix is already efficiently computed by GPTQ and thus does not impose any additional overheads. The main advantage of this approach is that s_{ij} is always determined based on the most current value of w_{ij} and thus accounts for adjustments due to previously quantized weights as well.

4.3.2 Exploring parameter sensitivity

Before we define out main method, SpQR, we provide a motivating analysis of parameter sensitivity which uncovers that the location of sensitive weights in the weight matrix are not random but have particular structures. To highlight these structural elements during the quantization process, we calculate the the per-weight sensitivities and visualize them for the popular and highly-accurate LLaMA-65B model [Touvron et al., 2023]. As the quantization method, we use GPTQ quantization to 3-bit, without weight grouping, following [Frantar et al., 2022a]. We use C4 [Raffel et al., 2020a] as the calibration dataset, and we estimate the error on 128 sequences of 2048 tokens each. Figure 4.2 depicts the output projection of the last self-attention layer of LLaMA-65B.

Using the sensitivity analysis, we observe several patterns in the weight matrix, often in a single row or column. Since the large weight matrices in LLaMA-65B have too many rows/columuns to be respresentable in a compact image (default: 8k $\times$ 32k pixels) we perform max pooling to visualize the matrices, that is we

take the maximum sensitivity in each square of 32×32 rows and columns. This max pooling only affects the leftmost image. Using this visualization, we observe that the quantization error patterns vary both by layer type, for example attention vs multilayer perceptron (MLP), and layer depth. In particular, we find that more sensitive outliers are present for deeper layers. To categorize outlier structures, we will use the attention weight matrix as a reference. We make the following observations:

- **Row outliers** are shown in Figure 4.2 bottom-center as regions of high sensitivity within one output unit. Some of these patterns span the entire row, while others are partial. In attention layers, some of the partial row outliers correspond to some subset of attention heads. **Column outliers** appear in Figure 4.2, bottom-right, showing high sensitivity in select input dimensions (columns) across all rows. The latter is related to the "outlier feature" phenomenon reported in Dettmers et al. [2022a].

- **Sensitive attention heads.** (Figure 4.2, top-center) – regular stripes of width 128 highlight all weights corresponding to one attention head. This could be related to some attention heads having more important functions [Voita et al., 2019; Vig, 2019; Olsson et al., 2022]. The corresponding "stripes" are horizontal for attention Q & K projections, vertical in output projection, and absent from value projections and any MLP weights. Of note, there is significant variation in individual weight sensitivity even within the sensitive heads.

- **The Rotary embedding pattern**, a repeating vertical pattern of sensitivity with a period of 64 units. We attribute this to the use of rotary embeddings [Su et al., 2021]: each attention head (dim = 128) is split into two halves: the first 64 are "rotated" with cosine, and the other 64 use sine. Both sine and cosine rotation use the same set of frequencies. Typically, the weights that correspond to low-frequency sines and cosines are more sensitive than their high-frequency counterparts, as shown in Figure 4.2 (top-right). This pattern is absent from any layer not using rotary embeddings.

- **Unstructured outliers.** Besides the above, each layer has a number of individual sensitivity weights that do not fit into any of the above patterns. These unstructured outliers occur more frequently for columns with largest input index (i.e., on the right side of the images). We believe is probably an artifact of the GPTQ algorithm, which compresses one by one, using yet-uncompressed weights to compensate the error. Thus, the rightmost batch of weights accumulates the most error.

Next, we will leverage these findings to propose a compressed representation which can support all these different outlier types.

4.4 SpQR: A Sensitivity-aware compressed representation

4.4.1 Overview

Existing LLM quantization algorithms treat low- and high-sensitivity weights equally; however, our above discussion suggests that this may lead to sub-optimal compression. Ideally, we would want the representation to assign more of its "size budget" to sensitive weights. However, these weights are scattered in the weight matrix as either individual weights or small groups, for example, partial rows or attention head. To capture this structure, we are introducing two changes to the quantization procedure: one for capturing small sensitive groups, and another for capturing individual outliers.

Capturing small groups of weights with bilevel quantization. In the previous section, we observed several cases where weights behave similarly in small consecutive groups, with abrupt changes between groups, for example for some attention head and partial row outliers (see Figure 4.2). When applying a standard approach, there will be many cases where these weights will be grouped together, sharing the same quantization statistics. To reduce the number of such cases, we use groupwise quantization with extremely small groups, typically of 8–32 weights. That is, for every β_1 consecutive weights, there is a separate quantization scale and zero-point. This choice runs contrary to current intuition: for instance, Yao et al. [2023] recommends against small groups, arguing that the overhead for storing quantization statistics would outweigh the precision advantages.

To circumvent this issue, we quantize the groupwise statistics themselves using the same quantization algorithm as for weights — asymmetric (min-max) quantization. In other words, we group groupwise statistics from $\beta_2 = 16$ consecutive values and quantize them together in the same number of bits, such that groups with atypical quantization parameters end up using more of the "quantization budget". Finally, both first and second-level quantization happens directly within the quantization process, allowing the algorithm to compensate the second-level quantization error where possible.

High-sensitivity outliers. Our analysis shows the existence of cases where a small percentage of sen-

Algorithm 1 SpQR quantization algorithm: the left snippet describes the full procedure, the right side contains subroutines for bilevel quantization and finding outliers.

`func` SPQRQUANTIZE($W, X, b, \beta_1, \beta_2, \tau, \lambda$)

Input: $W \in \mathcal{R}^{m \times n}$ — weight matrix,
$X \in \mathcal{R}^{n \times d}$ — calibration data,
b — the base number of quantization bits,
β_1, β_2 — quantization group sizes,
τ — sensitivity outlier threshold
λ — hessian regularizer,

$E :=$ float_matrix(m, n) // L2 error
$H := 2XX^T$ // L2 error hessian, $\mathcal{R}^{n \times n}$
$H^{\text{ic}} :=$ Cholesky$((H + \lambda \mathbf{I})^{-1})$
$Q :=$ int_matrix(m, n) // quantized weight
$\mathcal{O} := \emptyset$ // a set of all outliers
$\mathcal{S} := \emptyset$ // a set of quantization statistics **for** $i = 1, \beta_1, 2\beta_1, \dots n$ **do**
    end
    $W_{:,i:i+\beta_1}, \mathcal{O} :=$ outliers$(W_{:,i:i+\beta_1}, H^{\text{ic}}_{i:(i+\beta_1),i:(i+\beta_1)}\mathcal{O})$
$\hat{s}, \hat{z}, \mathcal{S} :=$ fit_statistics$(W_{:,i:i+\beta_1}, \mathcal{S}, \mathcal{O})$ **for** $j = i, \dots, i + \beta_1$ **do**
    end
    $Q_{:,j} :=$ quantize$(W_{:,j}, \hat{s}, \hat{z})$
$\vec{w}_q :=$ dequantize$(Q_{:,j}, \hat{s}, \hat{z})$
$E_{:,j} := (W_{:,j} - \vec{w}_q)/H^{\text{ic}}_{j,j} \cdot (1 -$ is_outlier$(W_{:,j}, \mathcal{O}))$
$W_{:,j:(i+\beta_1)} := W_{:,j:(i+\beta_1)} - E \cdot H^{\text{ic}}_{j,j:(i+\beta_1)}$

$W_{:,(i+\beta_1):n} := W_{:,(i+\beta_1):n} - E \cdot H^{\text{ic}}_{i:(i+\beta_1),i:(i+\beta_1)}$

$S_q, Z_q, S_s, Z_s, S_z, Z_z :=$ gather_statistics$(\mathcal{S})$
$W_{sparse} =$ gather_outlier_matrix$(W, \mathcal{O})$
return $Q, S_q, Z_q, S_s, Z_s, S_z, Z_z, W_{sparse}$

`func` **error**(W, H^{ic})

$\vec{s}, \vec{z} :=$ fit_quantizer(W, β_1)
$W_q :=$ quantize$(W, \vec{s}, \vec{z})$
$E := (W - W_q)/H^{\text{ic}}$
return E^2

`func` **outliers**$(W, H^{\text{ic}}, \mathcal{O})$

$E_{\text{base}} =$ error(W, H^{ic}) **for** $i = 1, \dots, \beta_1$ **do**
    end
    $loo := \{1, 2, ..., \beta_1\}/\{i\}$
$E_{\text{ol}} =$ error$(W_{:,\text{loo}}, H^{\text{ic}}_{\text{loo,loo}})$
$I_o =$ select$(E_{\text{base}} - E_{\text{ol}} > \tau)$
$\mathcal{O} := \mathcal{O} \cup I_o$

return $W, \mathcal{O}$

`func` **fit_statistics**$(W, \mathcal{S}, \mathcal{O})$

$W := W \cdot (1 -$ is_outlier$(W, O))$
$\vec{s}, \vec{z} :=$ fit_quantizer(W, β_1)
// $\vec{s}$ for scales, $\vec{z}$ for zero points
$\vec{s}_s, \vec{z}_s :=$ fit_quantizer$(\vec{s}, \beta_2)$
$\vec{s}_z, \vec{z}_z :=$ fit_quantizer$(\vec{z}, \beta_2)$
$\vec{s}_q :=$ quantize$(\vec{s}, \vec{s}_s, \vec{z}_s)$
$\vec{z}_q :=$ quantize$(\vec{z}, \vec{s}_z, \vec{z}_z)$
$\mathcal{S} := \mathcal{S} \cup \{s_q, s_s, s_z, z_q, s_z, z_z\}$
$\hat{s} :=$ dequantize(s_q, s_s, s_z)
$\hat{z} :=$ dequantize(z_q, z_s, z_z)
return $\hat{s}, \hat{z}, \mathcal{S}$

sitive weights come in small groups (in the self-attention) or individual "outliers" (in the MLP). In some cases, 1% of the weights account for over 75% of the total quantization error. Since these weights appear to lead to high, irreducible error, we choose to keep these outliers in 16-bit precision.

The procedure for detecting the outliers is described in detail in Alg. 1. It relies on **quantize**, **dequantize** and **fit_quantizer** functions that perform standard min-max quantization. The algorithm follows a rough two-step procedure: (1) find and isolate outliers as 16-bit weights, (2) quantize the non-outlier "base" weights into 3-4 bit and transfer the remaining quantization into the the 16-bit outliers weights. For the outlier isolation step, the algorithm implements a filtering technique based on the sensitivity criterion in defined in Section 4.3.1, which is used to isolate and separate outliers from base weights. We pick a model-dependent sensitivity threshold τ to obtain the desired number of outliers across the whole model. Usually,

the percentage of outliers as a share of total parameters has good memory/performance trade-off at around 0.3% to 1% of outlier weights which translates to $\tau \in [0.1, 0.40]$.

Following this first outlier detection step, we quantize the base weights ignoring all outliers that occur in the same quantization group. As such, the quantization statistics (e.g., scales) are computed by excluding outliers. This results in significant improvements in terms of error, since e.g. the min-max scales will be significantly reduced. Interestingly, unlike Dettmers et al. [2022a], a weight can be chosen to be an outlier not only if it causes error by itself, but also if the GPTQ algorithm can employ this weight to compensate errors from many other weights. Thus, the resulting 16-bit value will contain not the original weight, but a weight that was adjusted to minimize the output error. As such, SpQR goes beyond mere detection of outliers towards the more general notion of isolating and treating outliers that occur *during* the quantization process. Finally, the algorithm gathers and compresses sparse outlier matrix as well as the final quantization statistics with bilevel quantization and returns the compressed weights and their metadata.

4.4.2 Storing and Inferencing on SpQR Models

Our algorithm converts homogeneous weights into different data structures of various sizes and precisions, as depicted in Figure 4.3. Overall, the representation consists of four parts: (1) individual weight codes, (2) first level quantized quantization statistics, (3) second level quantization statistics, and (4) the sparse outlier indices and values. The first three parts are regular dense tensors that are easy to store and inference. In turn, the sparse outliers present a bigger challenge.

A naive way to encode a sparse outlier matrix would be to use compressed sparse row (CSR) format. This format groups outliers by their row index. In each row, CSR stores pairs of outlier value (e.g. float16) and it's column index (e.g. int16). In this format, at least half of the bits are spent on representing outlier locations, not their values. To reduce this overhead, we replace outlier indices with **relative index shifts between the adjacent outliers**. Instead storing a 16-bit index I_k, we could store 8-bit difference $\Delta_k = I_k - I_{k-1}$, but only if adjacent outliers are at most 255 columns apart.

In practice, not all outliers are within this range: about $3{-}7\%$ of those outliers would not fit into 8-bit Δ_k. Whenever adjacent outliers are too far away, *we introduce an additional "virtual" outlier with zero value*. Virtual outliers have no effect on model behavior, but they allow storing all Δ_k shifts as 8-bit integers. This results in an average of $8{-}9$ bits per *real* outlier index, compared to 16 bits in CSR. When

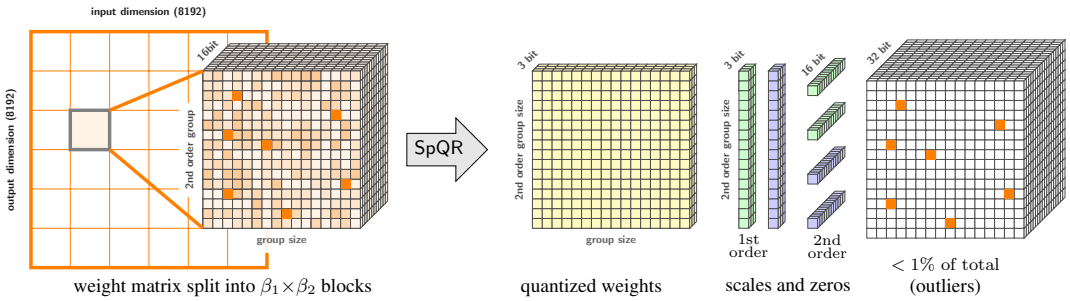


Figure 4.3: A high-level overview of the SpQR representation for a single weight tensor. The right side of the image depicts all stored data types and their dimensions.

running GPU inference in this format, we can recover the outlier indices on the fly using the parallel prefix sum (scan) algorithm [Harris et al., 2007]. The resulting format allows for a more memory-efficient outlier representation with negligible effect on GPU inference speed. This format starts being more effective than absolute indices at around 0.1% of the total number of weights.

Inference with SpQR. To illustrate the practicality of our approach, we design an efficient GPU-based decoding implementation for the SpQR format, focused on the popular token-by-token LLM generation as a use-case. We leverage the fact that autoregressive inference on GPUs is memory-bound, so high compression rates can hide decoding overheads, to a significant extent. At a high level, our algorithm loads group statistics and the quantized weights into shared memory (SRAM), dequantizes to 16-bits, and then performs matrix multiplication with 16-bit inputs. For handling outliers, we design a sparse matrix algorithm that takes advantage of outliers that occur in rows. Roughly, the algorithm works as follows. First, (1) we divide the matrix into equally sized blocks. Then, each GPU core (thread block) (2) loads a large slice of outliers into shared memory (SRAM), and each GPU core (3) determines if outliers are part of the segment or not. The corresponding weights are (4) loaded from main memory; finally, the matrix multiplication is performed.

This algorithm essentially performs load balancing through steps (1-3), while step (4) tends to have contiguous memory access due to the row-like patterns for the outliers. We will show in Section 4.5 that this custom approach is faster than the sparse matrix algorithms in PyTorch.

4.5 Experimental Validation

Experimental setup. We focus on three main goals: 1) evaluating most compact representation with which SpQR can replicate the performance of a 16-bit model within 1% perplexity, 2) controlling for the

average number of bits per parameter across methods and compare to round-to-nearest (RTN) and GPTQ baselines, 3) finding the best trade-off in terms of model size and performance. For these settings, we evaluate the full SpQR algorithm on publicly-available LLMs. We focus on the LLaMA-$\{7, 13, 30, 65\}$B model family [Touvron et al., 2023] and Falcon-$\{7, 40, 180\}$B model family [TII UAE, 2023a]. We quantize LLaMA models using the RedPajama dataset and Falcon models on RefinedWeb datasets [TII UAE, 2023b], both of which represent the original training data or open replications of the training data.We compare SpQR against two other post-training quantization schemes: GPTQ [Frantar et al., 2022a] and simple rounding-to-nearest (RTN) quantization, which is used by most other LLM compression methods [Dettmers et al., 2022a; Yao et al., 2022]. Both baselines use 4-bit quantization since it provides the best quality to size trade-off [Dettmers and Zettlemoyer, 2022]. For SpQR, we consider both 3-bit and 4-bit base quantization, though the resulting model size is slightly larger due to outliers.

We evaluate quantized model performance by two metrics. Firstly, we measure *perplexity*, measured on the WikiText2 [Merity et al., 2016], Penn Treebank [Marcus et al., 1994] and C4 [Raffel et al., 2020a] datasets. Secondly, we measure zero-shot accuracy on five tasks: WinoGrande [Sakaguchi et al., 2021], PiQA [Tata and Patel, 2003], HellaSwag, ARC-easy and ARC-challenge [Clark et al., 2018]. We use the LM Evaluation Harness [Gao et al., 2021] with recommended parameters. Our implementation takes around 4.5 hours on the largest model size (65B) on an NVIDIA A100 (80 GB). Our emory efficient implementations take 12 hours on a small 24 GB GPU.

To control for model size, we evaluate RTN and GPTQ with 4-bit base quantization. For SpQR we use 3-bit base quantization, a group size of 8 with 3-bit for the first quantization, a group size of 64 for the second quantization, and as many outliers as possible to still reach less than 4-bits per parameter on average. We aim to achieve *near-lossless* compression, for which we adopt the definition of the MLCommons benchmark [Reddi et al., 2020]: *1% error relative to the uncompressed baseline*. In all SpQR evaluations, we choose τ such that the proportion of outliers is under 1%.

Main Results. Figure 4.1 measures model size versus perplexity on LLaMA models on WikiText2, and accuracy on zero-shot tasks. We observe that SpQR outperforms GPTQ (and correspondingly RTN) at similar model size by a significant margin, especially on smaller models. This improvement comes from both SpQR achieving more compression, while also reducing loss degradation. In addition, if we measure

Table 4.1: Perplexity on WikiText2 [Merity et al., 2016], C4 [Raffel et al., 2020a] and Penn Treebank [Marcus et al., 1994] for SpQR and round-to-nearest (RTN) and GPTQ baselines on LLaMA and Falcon models. We can see that SpQR reaches performances within 1% of the perplexity with less than 4.5 bits per parameter. We also see that for 4-bits per parameter SpQR significantly improves on GPTQ with an improvement as large as the improvement from RTN to GPTQ.

Falcon

Size	Method	Avg bits	Wiki2	C4	PTB
7B	–	16.00	6.59	9.50	9.90
	SpQR	4.44	6.64	9.58	9.97
	RTN	4	8.73	12.56	13.76
	GPTQ	4	6.91	9.93	10.33
	SpQR	3.92	6.74	9.70	19.114
40B	–	16.00	5.23	7.76	7.83
	SpQR	4.46	5.26	7.79	7.86
	RTN	4	6.52	9.76	10.63
	GPTQ	4	5.36	7.95	8.01
	SpQR	3.90	5.29	7.85	7.91
180B	–	16.00	3.30	6.37	6.65
	GPTQ	4	3.66	6.83	6.58
	SpQR	3.83	3.43	6.41	6.71

LLaMA

Size	Method	Avg bits	Wiki2	C4	PTB
7B	–	16.00	5.68	7.08	8.80
	SpQR	4.63	5.73	7.13	8.88
	RTN	4	6.43	7.93	10.30
	GPTQ	4	6.13	7.43	9.27
	SpQR	3.94	5.87	7.28	9.07
30B	–	16.00	4.10	5.98	7.30
	SpQR	4.69	4.14	6.01	7.33
	RTN	4	4.57	6.34	7.75
	GPTQ	4	4.48	6.20	7.54
	SpQR	3.89	4.25	6.08	7.38
65B	–	16.00	3.53	5.62	6.91
	SpQR	4.71	3.57	5.64	6.93
	RTN	4	3.87	5.85	7.17
	GPTQ	4	3.83	5.80	7.07
	SpQR	3.90	3.68	5.70	6.99

the bits per parameter needed to come within 1% of the 16-bit performance in terms of perplexity, Figure 4.1 shows that SpQR with 4.6 to 4.71 bits per parameter approaches the non-quantized models with at most 1% margin of error for all models (see Table 4.1 for exact values).

Additional results where we control overall model size and compare quantization methods at an average of 4-bits per parameter are presented in Table 4.1. We see that SpQR improves over previous methods, with the gap between SpQR and the next best method GPTQ being as large as the improvement of GPTQ over naive RTN. For 4-bit, SpQR *halves the error* relative to the 16-bit baseline compared to GPTQ.

Ablations. The SpQR representation differs from standard quantization methods in two main ways: bilevel quantization with small quantization group size and unstructured outliers. To understand the effect of small group sizes, we compare 3-bit SpQR with group size 16, compressed using 3-bit bilevel quantization, versus a setup with group size 48, keeping quantization statistics in 16-bit. Both configurations result in approximately 3.6 average bits per parameter. For simplicity, neither uses outliers. We report both in Table 4.2,

Name	Wiki2	C4	PTB	Avg bits
Uncompressed	3.53	5.62	6.91	16
GPTQ (4 bit)	3.83	5.80	7.07	4
3-bit statistics	3.74	5.73	7.02	3.63
16-bit statistics	3.84	5.83	7.12	3.67
Round zero	3.75	5.76	7.01	3.63
w/o act order	3.74	5.76	7.05	3.63

Table 4.2: Perplexity for LLaMA-65B model.

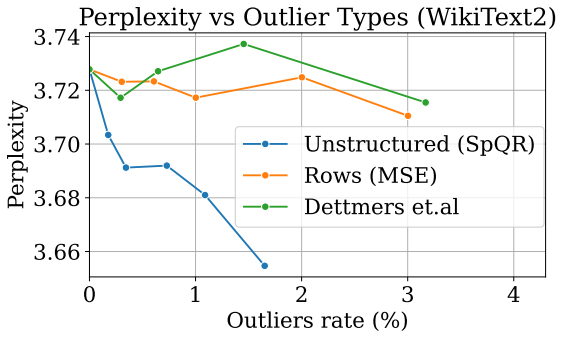


Figure 4.4: Different outlier types, LLaMA-65B.

the "3-bit statistics" entry corresponds to group size 16 with 3-bit statistics and "16-bit statistics" stands for group size 16 with 16-bit statistics. Given the same (slightly smaller) memory footprint, using quantized statistics significantly improves language modeling loss.

Next, we ask whether it is necessary to use unstructured outliers, considering two outlier types. First, we use the criterion of Dettmers and Zettlemoyer [2022] to find column outliers and quantize them in higher precision. The alternative is to treat the entire rows (output units / hidden units / neurons) as outliers: we run SpQR without outliers, then select k output units that have the highest quantization error (i.e., MSE between layer predictions) and treat the entire rows as 16-bit outliers. We compare the three outlier types on top of 3-bit SpQR and report the results in Figure 4.4. Overall, unstructured outliers reduce perplexity significantly faster than their row counterpart and the criterion of Dettmers and Zettlemoyer [2022], even after accounting for the different memory footprint.

Finally, we analyze the impact of the minor hyperparameter changes that we introduced at the end of Section 4.4. In Table 4.2 (bottom), we evaluate quantization errors without these changes. The "Round zero" entry corresponds to a version of SpQR where the zero-point is a 3-bit integer. This reduces the memory footprint of SpQR, but results in a moderate increase in perplexity. Similarly, we evaluate SpQR without the "act order" flag. This option re-orders the input dimensions by the diagonal of the inverse hessian, which was introduced as a part of the GPTQ algorithm. Using this heuristic slightly improves loss, though not as much as from quantized groups.

Inference Time. Finally, we evaluate LLM inference speed of SpQR for batch size 1 on a single A100 GPU. We measure inference speed in two setups: i) generating 100 tokens from scratch and ii) adding 100 tokens on top of a 1024-token prefix (prompt). We compare our sparse matrix multiplication algorithm with

the PyTorch default (cuSPARSE). We also compare against a 16-bit baseline. We measure the end-to-end latency as inference steps per second for the full SpQR algorithm, that is for both the dense and sparse multiplication part together. Results are shown in Table 4.3, where we see that our specialized sparse matrix multiplication algorithm yields speedups of about 20-30% and is 2x faster than a PyTorch implementation.

Table 4.3: Inference speed comparison (tokens/s), OOM means the model did not fit in an A100 GPU. We see that our optimized SpQR algorithm is faster than the 16-bit baseline and almost 2.0x faster than quantized matrix multiplication + standard PyTorch sparse matrix multiplication baseline.

Method	fp16 (baseline)				SpQR (PyTorch)				SpQR (optimized)			
LLaMA	7B	13B	30B	65B	7B	13B	30B	65B	7B	13B	30B	65B
scratch	47 ± 2.3	37 ± 0.8	19 ± 1.1	OOM	30 ± 2.2	24 ± 1.2	8.8 ± 0.4	OOM	**57** ± 2.4	**44** ± 0.5	**22** ± 0.9	**12** ± 0.6
prefix 1024	46 ± 2.4	31 ± 0.9	17 ± 0.8	OOM	27 ± 1.6	21 ± 1.1	6.5 ± 0.7	OOM	**55** ± 2.1	**37** ± 0.8	**22** ± 1.3	**11** ± 0.6

4.6 Discussion & Limitations

We have presented SpQR, a compression approach which quantizes sensitive outliers in higher precision, to achieve near-lossless 16-bit accuracy with less than 4.75 bits per parameter on average. We achieve even better quality-size-tradeoffs when compressing to as little as 3.5 bits. This makes SpQR an ideal method for compressing models for memory-limited devices. Despite our promising results, there are a few limitations. The main limitation is that we do not evaluate the generative quality of quantized LLMs, but only the predictive performance in terms of zero-shot accuracy and perplexity. While we believe that perplexity measurements and generation quality are strongly related, this is a hypothesis we aim to investigate in future work. While we devise a sparse matrix multiplication algorithm to accelerate the computation with outliers, another limitation is that we do not fuse sparse matrix multiplication with regular quantized matrix multiplication. Such an approach would yield even better inference time performance. We plan to investigate such practical kernel extensions more generally in future work.

Chapter 5

QLoRA: Efficient Finetuning of Quantized LLMs

5.1 Introduction

Finetuning large language models (LLMs) is a highly effective way to improve their performance, [Min et al., 2021; Wei et al., 2021; Ouyang et al., 2022; Wang et al., 2022c,b; Liu et al., 2022] and to add desirable or remove undesirable behaviors [Ouyang et al., 2022; Askell et al., 2021; Bai et al., 2022]. However, finetuning very large models is prohibitively expensive; regular 16-bit finetuning of a LLaMA 65B parameter model [Touvron et al., 2023] requires more than 780 GB of GPU memory. While recent quantization methods can reduce the memory footprint of LLMs [Dettmers et al., 2022a; Dettmers and Zettlemoyer, 2022; Frantar et al., 2022a; Xiao et al., 2022], such techniques only work for inference and break down during training [Wortsman et al., 2023].

We demonstrate for the first time that it is possible to finetune a quantized 4-bit model without any performance degradation. Our method, QLoRA, uses a novel high-precision technique to quantize a pretrained model to 4-bit, then adds a small set of learnable Low-rank Adapter weights [Hu et al., 2021] that are tuned by backpropagating gradients through the quantized weights.

QLoRA reduces the average memory requirements of finetuning a 65B parameter model from >780GB of GPU memory to <48GB without degrading the runtime or predictive performance compared to a 16-bit

Table 5.1: Elo ratings for a competition between models, averaged for 10,000 random initial orderings. The winner of a match is determined by GPT-4 which declares which response is better for a given prompt of the the Vicuna benchmark. 95% confidence intervals are shown (±). After GPT-4, Guanaco 33B and 65B win the most matches, while Guanaco 13B scores better than Bard.

Model	Size	Elo
GPT-4	-	1348 ± 1
Guanaco 65B	41 GB	1022 ± 1
Guanaco 33B	21 GB	992 ± 1
Vicuna 13B	26 GB	974 ± 1
ChatGPT	-	966 ± 1
Guanaco 13B	10 GB	916 ± 1
Bard	-	902 ± 1
Guanaco 7B	6 GB	879 ± 1

fully finetuned baseline. This marks a significant shift in accessibility of LLM finetuning: now the largest publicly available models to date finetunable on a single GPU. Using QLoRA, we train the **Guanaco** family of models, with the second best model reaching 97.8% of the performance level of ChatGPT on the Vicuna [Chiang et al., 2023] benchmark, while being trainable in less than 12 hours on a single consumer GPU; using a single professional GPU over 24 hours we achieve 99.3% with our largest model, essentially closing the gap to ChatGPT on the Vicuna benchmark. When deployed, our smallest **Guanaco** model (7B parameters) requires just 5 GB of memory and outperforms a 26 GB Alpaca model by more than 20 percentage points on the Vicuna benchmark (Table 5.4).

QLoRA introduces multiple innovations designed to reduce memory use without sacrificing performance: (1) **4-bit NormalFloat**, an information theoretically optimal quantization data type for normally distributed data that yields better empirical results than 4-bit Integers and 4-bit Floats. (2) **Double Quantization**, a method that quantizes the quantization constants, saving an average of about 0.37 bits per parameter (approximately 3 GB for a 65B model). (3) **Paged Optimizers**, using NVIDIA unified memory to avoid the gradient checkpointing memory spikes that occur when processing a mini-batch with a long sequence length. We combine these contributions into a better tuned LoRA approach that includes adapters at every network layer and thereby avoids almost all of the accuracy tradeoffs seen in prior work.

QLoRA's efficiency enables us to perform an in-depth study of instruction finetuning and chatbot performance on model scales that would be impossible using regular finetuning due to memory overhead. Therefore, we train more than 1,000 models across several instruction tuning datasets, model architectures,

and sizes between 80M to 65B parameters. In addition to showing that QLoRA recovers 16-bit performance (§5.3) and training a state-of-the-art chatbot, **Guanaco**, (§5.4), we also analyze trends in the trained models. First, we find that data quality is far more important than dataset size, e.g., a 9k sample dataset (OASST1) outperformed a 450k sample dataset (FLAN v2, subsampled) on chatbot performance, even when both are meant to support instruction following generalization. Second, we show that strong Massive Multitask Language Understanding (MMLU) benchmark performance does not imply strong Vicuna chatbot benchmark performance and vice versa—in other words, dataset suitability matters more than size for a given task.

Furthermore, we also provide a extensive analysis of chatbot performance that uses both human raters and GPT-4 for evaluation. We use tournament-style benchmarking where models compete against each other in matches to produce the best response for a given prompt. The winner of a match is judged by either GPT-4 or human annotators. The tournament results are aggregated into Elo scores [Elo, 1967, 1978] which determine the ranking of chatbot performance. We find that GPT-4 and human evaluations largely agree on the rank of model performance in the tournaments, but we also find there are instances of strong disagreement. As such, we highlight that model-based evaluation while providing a cheap alternative to human-annotation also has its uncertainties.

We augment our chatbot benchmark results with a qualitative analysis of **Guanaco** models. Our analysis highlights success and failure cases that were not captured by the quantitative benchmarks.

We release all model generations with human and GPT-4 annotations to facilitate further study. We open-source our codebase and CUDA kernels and integrate our methods into the Hugging Face transformers stack [Wolf et al., 2019], making them easily accessible to all. We release a collection of adapters for 7/13/33/65B size models, trained on 8 different instruction following datasets, for a total of 32 different open sourced, finetuned models.

5.2 QLoRA Finetuning

QLoRA achieves high-fidelity 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.

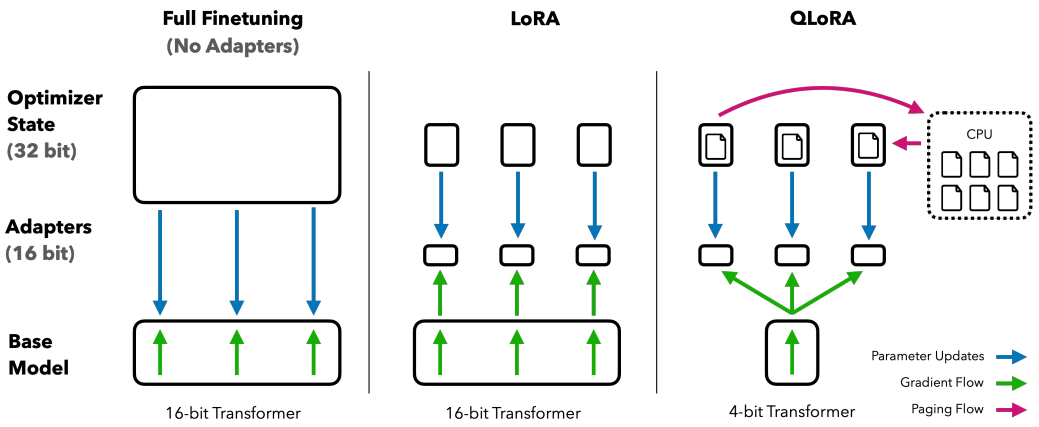


Figure 5.1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

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 [Dettmers et al., 2022b; Petersen and Sutter, 2023] 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 quantization works 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 [Dettmers et al., 2022b], are used to estimate them. Due to the approximate nature of these quantile estimation algorithms, the data type has large quantization errors for outliers, which are often the most important values.

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 pretrained neural network weights usually have a zero-centered normal distribution with standard deviation σ, we can transform all 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.

Once the weight range and data type range match, we can quantize as usual. Step (3) is equivalent to rescaling the standard deviation of the weight tensor to match the standard deviation of the k-bit data type. More formally, we estimate the 2^k values q_i of the data type as follows:

$$q_i = \frac{1}{2}\left(Q_X\left(\frac{i}{2^k+1}\right) + Q_X\left(\frac{i+1}{2^k+1}\right)\right), \tag{5.1}$$

where $Q_X(\cdot)$ is the quantile function of the standard normal distribution $N(0, 1)$. A problem for a symmetric k-bit quantization is that this approach does not have an exact representation of zero, which is an important property to quantize padding and other zero-valued elements with no error. To ensure a discrete zeropoint 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_i: 2^{k-1} for the negative part and $2^{k-1} + 1$ 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* (NFk), since the data type is information-theoretically optimal for zero-centered normally distributed data.

Double Quantization We introduce *Double Quantization* (DQ), the process of quantizing the quantization constants for additional memory savings. While a small blocksize is required for precise 4-bit quantization [Dettmers and Zettlemoyer, 2022], it also has a considerable memory overhead. For example, using 32-bit constants and a blocksize of 64 for $\mathbf{W}$, quantization constants add $32/64 = 0.5$ bits per parameter on average. Double Quantization helps reduce the memory footprint of quantization constants.

More specifically, Double Quantization treats quantization constants c_2^{FP32} of the first quantization as inputs to a second quantization. This second step yields the quantized quantization constants c_2^{FP8} and the second level of quantization constants c_1^{FP32}. We use 8-bit Floats with a blocksize of 256 for the sec-

ond quantization as no performance degradation is observed for 8-bit quantization, in line with results from Dettmers and Zettlemoyer [2022]. Since the c_2^{FP32} are positive, we subtract the mean from c_2 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 memory [1] feature which does automatic page-to-page transfers between the CPU and GPU for error-free GPU processing in the scenario where the GPU occasionally runs out-of-memory. The feature works like regular memory paging between CPU RAM and the disk. We use this feature to allocate paged memory for the optimizer states which are then automatically evicted to CPU RAM when the GPU runs out-of-memory and paged back into GPU memory when the memory is needed in the optimizer update step.

QLoRA. Using the components described above, we define QLoRA for a single linear layer in the quantized base model with a single LoRA adapter as follows:

$$\mathbf{Y}^{\text{BF16}} = \mathbf{X}^{\text{BF16}}\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{NF4}}) + \mathbf{X}^{\text{BF16}}\mathbf{L}_1^{\text{BF16}}\mathbf{L}_2^{\text{BF16}}, \tag{5.2}$$

$$\text{doubleDequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}, \mathbf{W}^{\text{k-bit}}) = \text{dequant}(\text{dequant}(c_1^{\text{FP32}}, c_2^{\text{k-bit}}), \mathbf{W}^{\text{4bit}}) = \mathbf{W}^{\text{BF16}}, \tag{5.3}$$

We use NF4 for $\mathbf{W}$ and FP8 for c_2. We use a blocksize of 64 for $\mathbf{W}$ for higher quantization precision and a blocksize of 256 for c_2 to conserve memory.

For parameter updates only the gradient with respect to the error for the adapters weights $\frac{\partial E}{\partial \mathbf{L}_i}$ are needed, and not for 4-bit weights $\frac{\partial E}{\partial \mathbf{W}}$. However, the calculation of $\frac{\partial E}{\partial \mathbf{L}_i}$ entails the calculation of $\frac{\partial \mathbf{X}}{\partial \mathbf{W}}$ which proceeds via equation (5) with dequantization from storage $\mathbf{W}^{\text{NF4}}$ to computation data type $\mathbf{W}^{\text{BF16}}$ to calculate the derivative $\frac{\partial \mathbf{X}}{\partial \mathbf{W}}$ in BFloat16 precision.

To summarize, QLoRA has one storage data type (usually 4-bit NormalFloat) and a computation data type (16-bit BrainFloat). We dequantize the storage data type to the computation data type to perform the forward and backward pass, but we only compute weight gradients for the LoRA parameters which use

[1] https://docs.nvidia.com/cuda/cuda-c-programming-guide

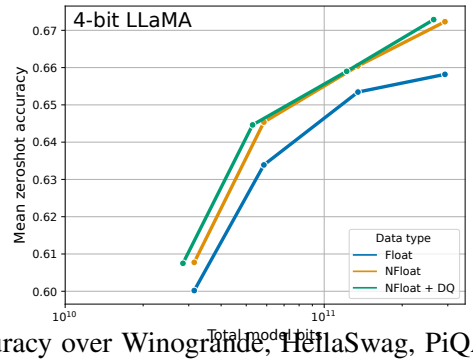


Figure 5.2: Mean zero-shot accuracy over Winogrande, HellaSwag, PiQA, Arc-Easy, and Arc-Challenge using LLaMA models with different 4-bit data types. The NormalFloat data type significantly improves the bit-for-bit accuracy gains compared to regular 4-bit Floats. While Double Quantization (DQ) only leads to minor gains, it allows for a more fine-grained control over the memory footprint to fit models of certain size (33B/65B) into certain GPUs (24/48GB).
16-bit BrainFloat.

5.3 QLoRA vs. Standard Finetuning

We have discussed how QLoRA works and how it can significantly reduce the required memory for finetuning models. The main question now is whether QLoRA can perform as well as full-model finetuning. Furthermore, we want to analyze the components of QLoRA including the impact of NormalFloat4 over standard Float4. The following subsections will discuss the experiments that aimed at answering these questions.

Experimental setup. We consider three architectures (encoder, encoder-decoder, and decoder only) and compare QLoRA with 16-bit adapter-finetuning and with full-finetuning for models up to 3B. Our evaluations include GLUE [Wang et al., 2018] with RoBERTa-large [Liu et al., 2019], Super-NaturalInstructionss [Wang et al., 2022c] with T5 [Raffel et al., 2020b], and 5-shot MMLU [Hendrycks et al., 2020] after finetuning LLaMA on Flan v2 [Longpre et al., 2023] and Alpaca [Taori et al., 2023].

To additionally study the advantages of NF4 over other 4-bit data types, we use the setup of Dettmers and Zettlemoyer [2022] and measure post-quantization zero-shot accuracy and perplexity across different models (OPT [Zhang et al., 2022], LLaMA [Touvron et al., 2023], BLOOM [Scao et al., 2022], Pythia [Biderman et al., 2023]) for model sizes 125m - 13B. We provide more details in the results subsection for each particular setup to make the results more readable.

Table 5.2: Pile Common Crawl mean perplexity for different data types for 125M to 13B OPT, BLOOM, LLaMA, and Pythia models.

Data type	Mean PPL
Int4	34.34
Float4 (E2M1)	31.07
Float4 (E3M0)	29.48
NFloat4 + DQ	**27.41**

While paged optimizers are critical to do 33B/65B QLoRA tuning on a single 24/48GB GPU, we do not provide hard measurements for Paged Optimizers since the paging only occurs when processing mini-batches with long sequence lengths, which is rare. We do, however, perform an analysis of the runtime of paged optimizers for 65B models on 48GB GPUs and find that with a batch size of 16, paged optimizers provide the same training speed as regular optimizers. Future work should measure and characterize under what circumstances slow-downs occur from the paging process.

4-bit NormalFloat yields better performance than 4-bit Floating Point While the 4-bit NormalFloat (NF4) data type is information-theoretically optimal, it still needs to be determined if this property translates to empirical advantages. We follow the setup from Dettmers and Zettlemoyer [2022] where quantized LLMs (OPT [Zhang et al., 2022], BLOOM Scao et al. [2022], Pythia Biderman et al. [2023], LLaMA) of different sizes (125M to 65B) with different data types are evaluated on language modeling and a set of zero-shot tasks. In Figure 5.2 and Table 5.2 we see that NF4 improves performance significantly over FP4 and Int4 and that double quantization reduces the memory footprint without degrading performance.

k-bit QLoRA matches 16-bit full finetuning and 16-bit LoRA performance Recent findings have established that 4-bit quantization for *inference* is possible, but leads to performance degradation relative to 16-bit [Dettmers and Zettlemoyer, 2022; Frantar et al., 2022a]. This raises the crucial question of whether the lost performance can be recovered by conducting 4-bit adapter finetuning. We test this for two setups.

The first setup test if QLoRA can replicate 16-bit full finetuning performance for T5 and RoBERTa model finetuning.

For our second setup, since full finetuning models at and beyond 11B parameters requires more than one server of high memory GPUs, we continue to test whether 4-bit QLoRA can match 16-bit LoRA at the 7B to 65B parameter scales. To this end, we finetune LLaMA 7B through 65B on two instruction

Table 5.3: Mean 5-shot MMLU test accuracy for LLaMA 7-65B models finetuned with adapters on Alpaca and FLAN v2 for different data types. Overall, NF4 with double quantization (DQ) matches BFloat16 performance, while FP4 is consistently one percentage point behind both.

	Mean 5-shot MMLU Accuracy								
LLaMA Size	7B		13B		33B		65B		Mean
Dataset	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	Alpaca	FLAN v2	
BFloat16	38.4	45.6	47.2	50.6	57.7	60.5	61.8	62.5	53.0
Float4	37.2	44.0	47.3	50.0	55.9	58.5	61.3	63.3	52.2
NFloat4 + DQ	39.0	44.5	47.5	50.7	57.3	59.2	61.8	63.9	53.1

following datasets, Alpaca and FLAN v2, and evaluate on the MMLU benchmark via 5-shot accuracy. Results are shown in Table 5.3 where we see that NF4 with double quantization fully recovers the 16-bit LoRA MMLU performance. In addition, we also note that QLoRA with FP4 lags behind the 16-bit brain float LoRA baseline by about 1 percentage point. This corroborates both our findings that (1) QLoRA with NF4 replicates both 16-bit full finetuning and 16-bit LoRA finetuning performance, and (2) NF4 is superior to FP4 in terms of quantization precision.

Summary Our results consistently show that 4-bit QLoRA with NF4 data type matches 16-bit full finetuning and 16-bit LoRA finetuning performance on academic benchmarks with well-established evaluation setups. We have also shown that NF4 is more effective than FP4 and that double quantization does not degrade performance. Combined, this forms compelling evidence that 4-bit QLoRA tuning reliably yields results matching 16-bit methods.

In line with previous work on quantization [Dettmers and Zettlemoyer, 2022], our MMLU and Elo results indicate that with a given finetuning and inference resource budget it is beneficial to increase the number of parameters in the base model while decreasing their precision. This highlights the importance of efficiency benefits from QLoRA. Since we did not observe performance degradation compared to full-finetuning in our experiments with 4-bit finetuning, this raises the question of where the performance-precision trade-off exactly lies for QLoRA tuning, which we leave to future work to explore.

We proceed to investigate instruction tuning at scales that would be impossible to explore with full 16-bit finetuning on academic research hardware.

5.4 Pushing the Chatbot State-of-the-art with QLoRA

Having established that 4-bit QLoRA matches 16-bit performance across scales, tasks, and datasets we conduct an in-depth study of instruction finetuning up to the largest open-source language models available for research. To assess the performance of instruction finetuning these models, we evaluate on a challenging Natural Language Understanding benchmark (MMLU) and develop new methods for real-world chatbot performance evaluation.

5.4.1 Experimental setup

We now describe an overview of the experimental setup.

Data As, to our knowledge, there is no comprehensive study of instruction-following datasets, we select eight recent datasets. We include datasets obtained through crowd-sourcing (OASST1 [Köpf et al., 2023], HH-RLHF [Bai et al., 2022]), distillation from instruction-tuned models (Alpaca [Taori et al., 2023], self-instruct [Wang et al., 2022b], unnatural-instructions [Honovich et al., 2022]), corpora aggregations (FLAN v2 [Chung et al., 2022]), as well as hybrids (Chip2 [LAION, 2023], Longform [Köksal et al., 2023]). These datasets cover different languages, data sizes, and licenses.

Training Setup To avoid confounding effects from different training objectives, we perform QLoRA finetuning with cross-entropy loss (supervised learning) without reinforcement learning, even for datasets that include human judgments of different responses. For datasets that have a clear distinction between instruction and response, we finetune only on the response. For OASST1 and HH-RLHF, multiple responses are available. We then select the top response at every level of the conversation tree and finetune on the full selected conversation, including the instructions. In all of our experiments, we use NF4 QLoRA with double quantization and paged optimizers to prevent memory spikes during gradient checkpointing. We do a small hyperparameter searches for the 13B and 33B LLaMA models and we find that all hyperparameter settings found at 7B generalize (including number of epochs) except learning rate and batch size. We halve the learning rate for 33B and 65B while doubling the batch size.

Baselines We compare our models to both research (Vicuna [Chiang et al., 2023] and Open Assistant [Köpf et al., 2023]) and commercial (GPT-4 [OpenAI, 2023], GPT-3.5-turbo and Bard) chatbot systems. The Open Assistant model is a LLaMA 33B model finetuned with Reinforcement Learning from Human Feedback (RLHF) on the OASST1 dataset. Vicuna does full fine-tuning of LLaMA 13B on proprietary user-shared conversations from ShareGPT and is thus the result of distillation from OpenAI GPT models.

Evaluation setup Chatbot evaluation is not straightforward since each prompt has many high-quality responses which are challenging to rank. We therefore follow a comprehensive approach, which involves (a) standard benchmarks that measures general language understanding performance (MMLU [Hendrycks et al., 2020]), (b) both automatic and human evaluation that measures how many responses from chatbot A are better compared to chatbot B, (c) a round-robin tournament-style evaluation where chatbots compete against each other in games where chatbot performance is measured as ELO.

5.4.2 Guanaco: QLoRA trained on OASST1 is a State-of-the-art Chatbot

Based on our automated and human evaluations, we find that the top QLoRA tuned model, Guanaco 65B, which we finetune on a variant of OASST1, is the best-performing open-source chatbot model and offers performance competitive to ChatGPT. When compared to GPT-4, Guanaco 65B and 33B have an expected win probability of 30%, based on Elo rating from human annotators system-level pairwise comparisons on the Vicuna benchmark - the highest reported to date.

The Vicuna benchmark Chiang et al. [2023] results relative to ChatGPT are shown in Table 5.4. We find that Guanaco 65B is the best-performing model after GPT-4, achieving 99.3% performance relative to ChatGPT. Guanaco 33B has more parameters than the Vicuna 13B model, but uses only 4-bit precision for its weights and is thus much more memory efficient at 21 GB vs 26 GB, providing a three percentage points of improvement over Vicuna 13B. Furthermore, Guanaco 7B easily fits on modern phones at a 5 GB footprint while still scoring nearly 20 percentage points higher than Alpaca 13B.

However, Table 5.4 also has very wide confidence intervals, with many models overlapping in performance. We hypothesize that this uncertainty comes from the lack of clear specification of scale, e.g., it is unclear what 8 on a 10 point scale means across different scenarios. As such, we instead recommend using the Elo ranking method [Elo, 1967], based on *pairwise* judgments from human annotators and GPT-4 to

Table 5.4: Zero-shot Vicuna benchmark scores as a percentage of the score obtained by ChatGPT evaluated by GPT-4. We see that OASST1 models perform close to ChatGPT despite being trained on a very small dataset and having a fraction of the memory requirement of baseline models.

Model / Dataset	Params	Model bits	Memory	ChatGPT vs Sys	Sys vs ChatGPT	Mean	95% CI
GPT-4	-	-	-	119.4%	110.1%	**114.5**%	2.6%
Bard	-	-	-	93.2%	96.4%	94.8%	4.1%
Guanaco	65B	4-bit	41 GB	96.7%	101.9%	**99.3**%	4.4%
Alpaca	65B	4-bit	41 GB	63.0%	77.9%	70.7%	4.3%
FLAN v2	65B	4-bit	41 GB	37.0%	59.6%	48.4%	4.6%
Guanaco	33B	4-bit	21 GB	96.5%	99.2%	**97.8**%	4.4%
Open Assistant	33B	16-bit	66 GB	73.4%	85.7%	78.1%	5.3%
Alpaca	33B	4-bit	21 GB	67.2%	79.7%	73.6%	4.2%
FLAN v2	33B	4-bit	21 GB	26.3%	49.7%	38.0%	3.9%
Vicuna	13B	16-bit	26 GB	91.2%	98.7%	**94.9**%	4.5%
Guanaco	13B	4-bit	10 GB	87.3%	93.4%	90.4%	5.2%
Alpaca	13B	4-bit	10 GB	63.8%	76.7%	69.4%	4.2%
HH-RLHF	13B	4-bit	10 GB	55.5%	69.1%	62.5%	4.7%
Unnatural Instr.	13B	4-bit	10 GB	50.6%	69.8%	60.5%	4.2%
Chip2	13B	4-bit	10 GB	49.2%	69.3%	59.5%	4.7%
Longform	13B	4-bit	10 GB	44.9%	62.0%	53.6%	5.2%
Self-Instruct	13B	4-bit	10 GB	38.0%	60.5%	49.1%	4.6%
FLAN v2	13B	4-bit	10 GB	32.4%	61.2%	47.0%	3.6%
Guanaco	7B	4-bit	5 GB	84.1%	89.8%	**87.0**%	5.4%
Alpaca	7B	4-bit	5 GB	57.3%	71.2%	64.4%	5.0%
FLAN v2	7B	4-bit	5 GB	33.3%	56.1%	44.8%	4.0%

Table 5.5: MMLU 5-shot test results for different sizes of LLaMA finetuned on the corresponding datasets using QLoRA.

Dataset	7B	13B	33B	65B
LLaMA no tuning	35.1	46.9	57.8	63.4
Self-Instruct	36.4	33.3	53.0	56.7
Longform	32.1	43.2	56.6	59.7
Chip2	34.5	41.6	53.6	59.8
HH-RLHF	34.9	44.6	55.8	60.1
Unnatural Instruct	41.9	48.1	57.3	61.3
Guanaco (OASST1)	36.6	46.4	57.0	62.2
Alpaca	38.8	47.8	57.3	62.5
FLAN v2	44.5	51.4	59.2	63.9

avoid the problem of grounding an absolute scale.

Elo ratings of the most competitive models can be seen in Table 5.1. We note that human and GPT-4 ranking of models on the Vicuna benchmark disagree partially, particularly for Guanaco 7B, but are consistent for most models with a Kendall Tau of $\tau = 0.43$ and Spearman rank correlation of $r = 0.55$ at the system level. At the example level, the agreement between GPT-4 and human annotators' majority vote is weaker with Fleiss $\kappa = 0.25$. Overall, this shows a moderate agreement between system-level judgments by GPT-4 and human annotators, and thus that model-based evaluation represents a somewhat reliable alternative to human evaluation. See subsection **??** for further considerations.

Elo rankings in Table 5.6 indicate that Guanaco 33B and 65B models outperform all models besides GPT-4 on the Vicuna and OA benchmarks and that they perform comparably to ChatGPT in line with Table 5.4. We note that the Vicuna benchmark favors open-source models while the larger OA benchmark favors ChatGPT. Furthermore, we can see from Tables 5.5 and 5.4 that the suitability of a finetuning dataset is a determining factor in performance. Finetuning LLaMA models on FLAN v2 does particularly well on MMLU, but performs worst on the Vicuna benchmark (similar trends are observed with other models). This also points to partial orthogonality in current evaluation benchmarks: strong MMLU performance does not imply strong chatbot performance (as measured by Vicuna or OA benchmarks) and vice versa.

Guanaco is the only top model in our evaluation that is not trained on proprietary data as the OASST1 dataset collection guidelines explicitly forbid the use of GPT models. The next best model trained on only open-source data is the Anthropic HH-RLHF model, which scores 30 percentage points lower than Guanaco on the Vicuna benchmark (see Table 5.4). Overall, these results show that 4-bit QLoRA is effective and can produce state-of-the-art chatbots that rival ChatGPT. Furthermore, our 33B Guanaco can be trained on 24 GB consumer GPUs in less than 12 hours. This opens up the potential for future work via QLoRA tuning

Table 5.6: Elo rating for a tournament between models where models compete to generate the best response for a prompt, judged by human raters or GPT-4. Overall, Guanaco 65B and 33B tend to be preferred to ChatGPT-3.5 on the benchmarks studied. According to human raters they have a Each 10-point difference in Elo is approximately a difference of 1.5% in win-rate.

Benchmark	Vicuna		Vicuna		Open Assistant		
# Prompts	80		80		953		
Judge	Human raters		GPT-4		GPT-4		Median Rank
Model	Elo	Rank	Elo	Rank	Elo	Rank	
GPT-4	1176	1	1348	1	1294	1	1
Guanaco-65B	1023	2	1022	2	1008	3	2
Guanaco-33B	1009	4	992	3	1002	4	4
ChatGPT-3.5 Turbo	916	7	966	5	1015	2	5
Vicuna-13B	984	5	974	4	936	5	5
Guanaco-13B	975	6	913	6	885	6	6
Guanaco-7B	1010	3	879	8	860	7	7
Bard	909	8	902	7	-	-	8

on specialized open-source data, which produces models that can compete with the very best commercial models that exist today.

5.5 Conclusion

We have shown evidence that our method, QLoRA, can replicate 16-bit full finetuning performance with a 4-bit base model and Low-rank Adapters. This enables the finetuning of 33B parameter LLMs on a single consumer GPU without any performance degradation thus enabling most research to perform research on the largest LLMs with very few resources.

Chapter 6

SWARM Parallelism: Training Large Models Can Be Surprisingly Communication-Efficient

6.1 Introduction

For the past several years, the deep learning community has been growing more reliant on large pretrained neural networks. The most evident example of this trend is natural language processing, where the parameter count of models has grown from hundreds of millions [Vaswani et al., 2017a; Radford et al., 2018; Devlin et al., 2019] to billions [Narayanan et al., 2021; Raffel et al., 2020b; **?**; Sun et al., 2021] to hundreds of billions [Brown et al., 2020a; Fedus et al., 2021; Chowdhery et al., 2022a; Rae et al., 2021] with consistent gains in quality [Kaplan et al., 2020]. Likewise, many models in computer vision are reaching the billion-parameter scale [Ramesh et al., 2021; Zhai et al., 2021; Dai et al., 2021; Dhariwal and Nichol, 2021].

At this scale, the models no longer fit into a single accelerator and require specialized training algorithms that partition the parameters across devices [Krizhevsky et al., 2012; Dean et al., 2012]. While these model-parallel algorithms use different partitioning strategies, they all share the need to perform intensive device-to-device communication [Narayanan et al., 2019, 2021]. Also, if a single device fails, it will cause the entire training process to break down. As a result, model-parallel algorithms are typically deployed in dedicated

high-performance computing (HPC) clusters or supercomputers [Shoeybi et al., 2019; Rajbhandari et al., 2020; Narayanan et al., 2021].

This kind of infrastructure is notoriously expensive to build and operate, which makes it available only to a few well-resourced organizations [Larrea et al., 2019; Strohmaier et al., 2021; Langston, 2020]. Most researchers cannot afford the experiments necessary for a proper evaluation of their ideas. This ultimately limits the scientific progress for many important research areas, such as solving NLP problems in "non-mainstream" languages.

Several recent works propose more cost-efficient distributed training strategies that leverage fleets of temporary "preemptible" instances that can be dynamically allocated in regions with low demand for hardware and electricity, making them 2–10 times cheaper than their dedicated counterparts [Harlap et al., 2017]. Another solution is to train in "collaborations" by pooling together preexisting resources or using the help of volunteers [Diskin et al., 2021; Atre et al., 2021; Ryabinin and Gusev, 2020; Yuan et al., 2022].

However, training in either of those setups requires specialized algorithms that can adapt to the changing number of workers, utilize heterogeneous devices and recover from hardware and network failures. While there are several practical algorithms for unreliable hardware [Kijsipongse et al., 2018; Lin et al., 2020a; Ryabinin et al., 2021], they can only train relatively small models that *fit into the memory of the smallest device*. This limits the practical impact of cost-efficient strategies, because today's large-scale experiments often involve models with billions of parameters.

In this work, we aim to find a practical way of training large neural networks using **unreliable heterogeneous devices with slow interconnect**. We begin by studying the impact of model size on the balance between communication and computation costs of pipeline-parallel training. Specifically, increasing the size leads computation costs to grow faster than the network footprint, thus making **household-grade connection speeds** more practical than one might think. This idea inspires the creation of **SWARM parallelism**, a pipeline-parallel approach designed to handle peer failures by prioritizing stable peers with lower latency. In addition, this approach periodically rebalances the pipeline stages, which allows handling devices with different hardware and network speeds.

In summary, we make the following contributions:

- We analyze the existing model-parallel training techniques and formulate the "Square-Cube Law" of

distributed training: a counterintuitive observation that, for some methods, *training larger models can actually decrease the network overhead*.

- We develop SWARM parallelism, a decentralized model-parallel algorithm[1] that leverages randomized fault-tolerant pipelines and dynamically rebalances nodes between pipeline stages. To the best of our knowledge, this is the first decentralized algorithm capable of billion-scale training on heterogeneous unreliable devices with slow interconnect.

- Combining insights from the square-cube law, SWARM parallelism, and 8-bit compression, we show that it is possible to train a billion-scale Transformer language model on preemptible servers with low-power GPUs and the network bandwidth of less than 200Mb/s while achieving high training throughput.

6.2 Background & Related Work

6.2.1 Model-parallel training

Over the past decade, the deep learning community has developed several algorithms for training large neural networks. Most of them work by dividing the model between multiple workers, which is known as model parallelism. The exact way in which these algorithms divide the model determines their training performance and the maximum model size they can support.

Traditional model parallelism. Historically, the first general strategy for training large models was to assign each device to compute a subset of each layer (e.g., a subset of neurons), then communicate the results between each other [Krizhevsky et al., 2012; Ben-Nun and Hoefler, 2019; Tang et al., 2020]. Since each device stores a fraction of layer parameters, this technique can train models with extremely wide layers that would not fit into a single GPU. However, applying traditional model parallelism to deep neural networks comes at a significant performance penalty, as it requires all-to-all communication after each layer. As a result, while intra-layer parallelism is still widely used [Shazeer et al., 2018a; Rajbhandari et al., 2020], it is

[1] The code for our experiments can be found at `github.com/iclr2023-submit/swarm`

usually applied within one physical server in combination with other strategies [Krizhevsky, 2014; Chilimbi et al., 2014; Jia et al., 2019b; Narayanan et al., 2021].

Pipeline parallelism circumvents the need for expensive all-to-all communication by assigning each device with one or several layers [Huang et al., 2019]. During the forward pass, each stage applies its subset of layers to the inputs supplied by the previous stage, then sends the outputs of the last layer to the next stage. For the backward pass, this process is reversed, with each pipeline stage passing the gradients to the device that supplied it with input activations.

To better utilize the available devices, the pipeline must process multiple microbatches per step, allowing each stage to run in parallel on a different batch of inputs. In practice, the number of microbatches is limited by the device memory: this results in reduced device utilization when processing the first and the last microbatches, known as the "bubble" overhead [Huang et al., 2019]. To combat this issue, subsequent studies propose using activation checkpointing, interleaved scheduling, and even asynchronous training [Narayanan et al., 2019, 2021; Huang et al., 2019; Shoeybi et al., 2019; Yang et al., 2019].

Aside from model parallelism, there two more strategies for training large models: data parallelism with dynamic parameter loading [Rajbhandari et al., 2020] and model-specific algorithms such as Mixture-of-Experts [Shazeer et al., 2017].

6.2.2 Distributed training outside HPC

The techniques described in Section 6.2.1 are designed for clusters of identical devices with rapid and reliable communication, making them a natural fit for the HPC setup. As we discussed earlier, such infrastructure is not always available, and a more cost-efficient alternative is to use "preemptible" instances [Li et al., 2019; Zhang et al., 2020; Harlap et al., 2017] or volunteer computing [Kijsipongse et al., 2018; Ryabinin and Gusev, 2020; Atre et al., 2021; Diskin et al., 2021]. However, these environments are more difficult for distributed training: each machine can disconnect abruptly due to a failure or preemption. Besides, since there is a limited number of available instances per region, training at scale often requires operating across multiple locations or using different instance types.

To handle unstable peers and heterogeneous devices, the research community has proposed elastic and asynchronous training methods, correspondingly. Moreover, training large models over heterogeneous de-

vices can be optimized with global scheduling Yuan et al. [2022].

By contrast, the largest models have billions of parameters, which exceeds the memory limits of most low-end computers. However, model-parallel algorithms are not redundant, which makes them more vulnerable to hardware and network failures. There exist two methods that allow training large models with unreliable devices [Ryabinin and Gusev, 2020; Thorpe et al., 2022]: however, the first one supports only specific architectures and requires at least 1Gb/s bandwidth, whereas the second one has no publicly available implementations, relies on redundant computations for fault tolerance and considers only the homogeneous setup.

6.2.3 Communication efficiency and compression

In this section, we discuss techniques that address training with limited network bandwidth or high latency, such as gradient compression or overlapping computation with communication phases. These techniques are often necessary for distributed training without high-speed connectivity, because otherwise the performance of the system becomes severely bottlenecked by communication.

Efficient gradient communication. Data-parallel training requires synchronization of gradients after each backward pass, which can be costly if the model has many parameters or the network bandwidth is limited. There exist several methods that approach this problem: for example, Deep Gradient Compression [Lin et al., 2018] sparsifies the gradients and corrects the momentum after synchronization, while PowerSGD [Vogels et al., 2019] factorizes the gradients and uses error feedback to reduce the approximation error. Recently, Wang et al. [2022a] proposed to compress the changes of model activations, achieving high-speed communication for finetuning models of up to 1.5B parameters. Alternatively, Dettmers [2015] uses 8-bit quantization to compress gradients before communication. We evaluate it along with compression-aware architectures, leaving the exploration of more advanced approaches to future work.

Besides gradient compression, another effective technique is to use layer sharing [Lan et al., 2020], which reduces the number of aggregated gradients by a factor of how many times each layer is reused.

Overlapping communication and computation. Model, pipeline, and data parallelism all have synchronization points and require transfer of gradients or activations. One way to reduce the transfer cost is to

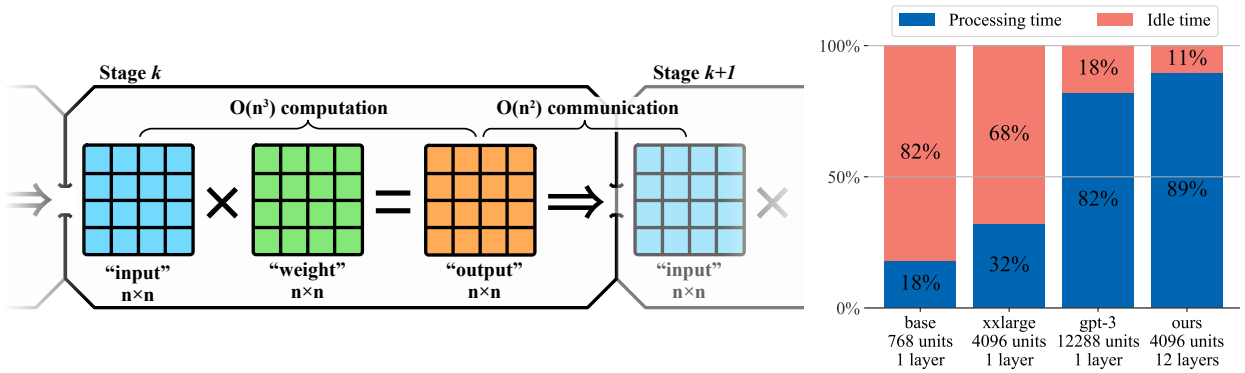


Figure 6.1: (Left) An intuitive explanation of the square-cube law, **(Right)** Relative device utilization for Transformer layers using Tesla V100 and 500Mb/s network bandwidth. See Section 6.4.1.

overlap communication with computation, *hiding* the synchronization latency. This overlap can be achieved by combining parallelism techniques [Krizhevsky, 2014; Rajbhandari et al., 2020], by synchronizing gradients layer-by-layer in lockstep with backpropagation [Paszke et al., 2019b], or by using pure pipeline parallelism [Huang et al., 2019; Narayanan et al., 2019]. However, pure pipeline parallelism requires many stages to effectively hide the latency. To overcome this problem, we study inter-layer compression techniques that work well even with relatively few pipeline stages.

6.3 Communication-efficient model parallelism

In this section, we outline our approach for training large models with heterogeneous unreliable poorly-connected devices. To that end, the section is organized as follows:

- Section 6.3.1 analyzes how existing model-parallel algorithms scale with model size and shows conditions where training increasingly larger models leads to less intense network usage;
- Section 6.3.2 describes SWARM parallelism — a decentralized algorithm for training large models under the conditions outlined in Section 6.2.2.

6.3.1 The square-cube law of distributed training

To better understand the general scaling properties of model parallelism, we need to abstract away from the application-specific parameters, such as model architecture, batch size, and system design. To that end,

we first consider a simplified model of pipeline parallelism. Our "pipeline" consists of k stages, each represented by $n \times n$ matrices. Intuitively, the first matrix represents the input data and all subsequent matrices are linear "layers" applied to that data. This model abstracts away from application-specific details, allowing us to capture general relationships that hold for many models.

During "training", stages iteratively perform matrix multiplication and then send the output to the subsequent pipeline stage over a throughput-limited network. These two operations have different scaling properties. The compute time for naïve matrix multiplication scales as $O(n^3)$. While this can be reduced further in theory [Coppersmith and Winograd, 1990; Alman and Williams, 2021], it is only used for very large matrices [Zhang and Gao, 2015; Fatahalian et al., 2004; Huang et al., 2020]. Therefore, deep learning on GPUs typically relies on $O(n^3)$ algorithms.

In turn, the communication phase requires at most $O(n^2)$ time to transfer a batch of $n \times n$ activations or gradients. Therefore, as we increase the model size, the computation time grows faster than communication time, regardless of which matrix multiplication algorithm we use. We refer to this idea as the *square-cube law* after the eponymous principle in physics [Galileo, 1638; Allen, 2013].

This principle applies to many real-world neural network architectures, albeit with some confounding variables. In convolutional neural networks Fukushima [1980], the computation time scales as $O(BHWC^2)$ and the communication is $O(BHWC)$, where B, H, W and C stand for batch size, height, width and the number of channels. Recurrent neural networks [Rumelhart et al., 1986; Hochreiter and Schmidhuber, 1995] need $O(BLH^2)$ compute in terms of batch size, sequence length, and hidden size, respectively, and $O(BLH)$ or $O(BH)$ communication, depending on the architecture. With the same notation, Transformers [Vaswani et al., 2017a] require $O(BL^2H)$ compute for attention layers, $O(BLH^2)$ compute for feedforward layers, but only $O(BLH)$ communication.

Based on these observations, we conclude that pipeline parallelism naturally grows more communication-efficient with model size. More precisely, increasing the hidden dimension will reduce the communication load per device per unit of time, making it possible to train the model efficiently *with lower network bandwidth* and *higher latency*[2]. While the exact practical ramifications depend on the use case, Section 6.4.1 demonstrates that some of the larger models trained with pipeline parallelism can already train at peak

[2]Latency slows the communication down by a constant factor that also grows less important with model size.

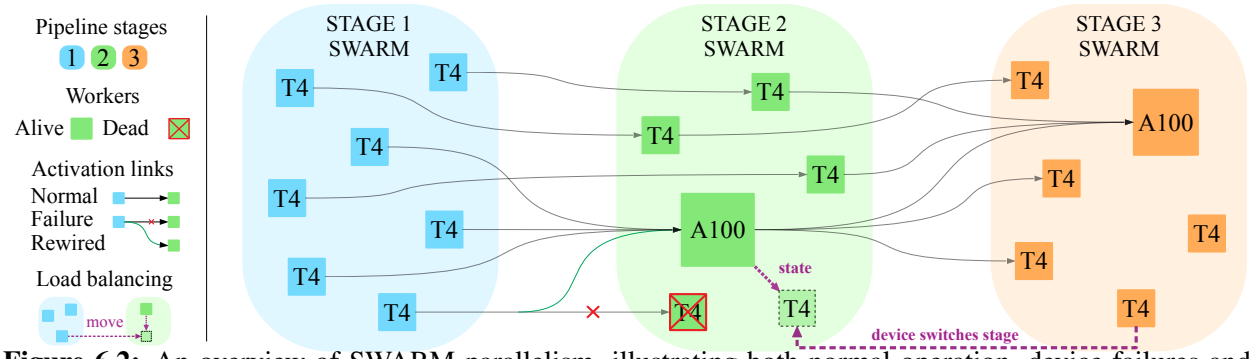


Figure 6.2: An overview of SWARM parallelism, illustrating both normal operation, device failures and adaptive rebalancing. One of the workers at stage 2 leaves; another peer from stage 3 takes its place by downloading the latest stage 2 parameters and statistics from peers.

efficiency with only hundreds of Mb/s bandwidth.

In theory, the square-cube principle also applies to intra-layer parallelism, but using this technique at 500 Mb/s would become practical only for layer sizes of more than 2^{16} units. Data-parallel training with sharding or offloading [Ren et al., 2021] does not scale as well, as its communication time scales with the size of *model parameters* instead of activations. However, it may be possible to achieve similar scaling with gradient compression algorithms.

6.3.2 SWARM parallelism

Traditional pipeline parallelism can be communication-efficient, but this alone is not enough for our setups. Since training devices can have different compute and network capabilities, a pipeline formed out of such devices would be bottlenecked by the single "weakest link", i.e., the participant with the smallest training throughput. As a result, the more powerful nodes along the pipeline would be underutilized due to either lack of inputs or slow subsequent stages. On top of that, if any node fails or leaves training prematurely, it will stall the entire training procedure.

To overcome these two challenges, we replace the rigid pipeline structure with temporary "pipelines" that are built stochastically on the fly during each iteration. Each participant can send their outputs to any peer that serves the next pipeline stage. Thus, if one peer is faster than others, it can process inputs from multiple predecessors and distribute its outputs across several weaker peers to maximize utilization. Also, if any participant disconnects, its predecessors can reroute their requests to its neighbors. New peers can download up-to-date parameters and optimizer statistics from remaining workers at the chosen stage. This

allows the training to proceed as long as there is at least one active participant per stage.

The resulting system consists of several consecutive swarms, as depicted in Figure 6.2. Peers within one swarm serve the same pipeline stage (i.e., the **same subset of layers** with **the same parameters**). We assume that the model consists of similar "blocks" and thus partition it into evenly sized stages, leaving the study of better strategies [Huang et al., 2019; Narayanan et al., 2019] as future work. During the *forward* pass, peers receive inputs from predecessors (determined on each iteration) and send activations to peers in the next stage. For the *backward* pass, peers receive gradients for outputs, compute gradients for layer inputs and accumulate gradients for parameters. Once enough gradients are accumulated, peers form groups, run All-Reduce to average gradients within their pipeline stages and perform the optimizer step.

SWARM parallelism can also use Delayed Parameter Updates (DPU) [Ren et al., 2021] to further improve hardware utilization by performing the optimizer step in parallel with processing the next batch. While it is technically asynchronous, DPU was shown to achieve similar per-iteration convergence as fully synchronous training, both theoretically [Stich and Karimireddy, 2020; Arjevani et al., 2020] and empirically [Ren et al., 2021; Diskin et al., 2021].

Each peer has queues for incoming and outgoing requests to maintain high GPU utilization under latency and to compensate for varying network speeds. Similarly to other pipeline implementations [Huang et al., 2019; Narayanan et al., 2021], SWARM parallelism uses activation checkpointing [Griewank and Walther, 2000; Chen et al., 2016] to reduce the memory footprint.

Stochastic wiring. To better utilize heterogeneous devices and recover from faults, we dynamically "wire" each input through each stage and pick devices in proportion to their training throughput. To achieve this, SWARM peers run "trainer" processes that route training data through the "stages" of SWARM, balancing the load between peers.

For each pipeline stage, trainers discover which peers currently serve this stage via a Distributed Hash Table (DHT, Maymounkov and Mazieres, 2002). Trainers then assign a microbatch to one of those peers based on their performance. If that peer fails, it is temporarily banned and the microbatch is sent to another peer within the same stage. Note that trainers themselves do not use GPUs and have no trainable parameters, which makes it possible to run multiple trainers per peer.

Each trainer assigns data independently using the Interleaved Weighted Round-Robin [Katevenis et al.,

1991; Tabatabaee et al., 2020] scheduler. Our specific implementation of IWRR uses a priority queue: each peer is associated with *the total processing time over all previous requests*. A training minibatch is then routed to the node that has the smallest total processing time. Thus, for instance, if device A takes half as long to process a sample as device B, the routing algorithm will choose A twice as often as B. Finally, if a peer does not respond or fails to process the batch, trainer will "ban" this peer until it reannounces itself in the DHT, which is done every few minutes.

Curiously, different trainers can have different throughput estimates for the same device because of the network topology. For instance, if training nodes are split between two cloud regions, a given peer's trainer will have a higher throughput estimate for peers in the same data center. In other words, trainers automatically adjust to the network topology by routing more traffic to peers that are "nearby".

Adaptive swarm rebalancing. While stochastic wiring allows for automatic rebalancing within a stage, additional cross-stage rebalancing may be required to maximize throughput, especially when devices are very unreliable. As we described in Section 6.2.2, our workers can join and leave training at any time. If any single pipeline stage loses too many peers, the remaining ones will face an increased processing load, which will inevitably form a bottleneck.

SWARM parallelism addresses this problem by allowing peers to dynamically switch between "pipeline stages" to maximize the training throughput. Every T seconds, peers measure the utilization rate of each pipeline stage as the queue size. Peers from the most underutilized pipeline stage will then switch to the most overutilized one (see Figure 6.2 for an overview), download the latest training state from their new neighbors and continue training. Similarly, if a new peer joins midway through training, it is assigned to the optimal pipeline stage by following the same protocol. As a side effect, if one pipeline stage requires more compute than others, SWARM will allocate more peers to that stage.

6.4 Experiments

6.4.1 Communication efficiency at scale

Before we can meaningfully evaluate SWARM parallelism, we must verify our theoretical observations on communication efficiency. Here we run several controlled experiments that measure the GPU utilization

Table 6.1: Relative device utilization at 500 Mb/s bandwidth and varying network latency.

Latency (RTT)	Relative GPU utilization (100% - idle time)			
	base	xxlarge	GPT-3	Ours
none	18.0%	32.1%	82.1%	89.5%
10ms	11.8%	28.9%	79.3%	87.2%
50ms	4.88%	20.1%	70.3%	79.5%
100ms	2.78%	14.9%	60.2%	71.5%
200ms	1.53%	10.1%	48.5%	59.2%

and network usage for different model sizes, using the Transformer architecture [Vaswani et al., 2017a] that has been widely adopted in various fields [Lin et al., 2021]. To decouple the performance impact from other factors, we run these experiments on homogeneous V100 GPU nodes that serve one pipeline stage over the network with varying latency and bandwidth. We use a batch size of 1 and sequences of 512 tokens.

First, we measure how the model size affects the computation to communication ratio at 500 Mb/s network bandwidth in both directions. We consider 4 model configurations: the base configuration from the BERT paper [Devlin et al., 2019], "xxlarge" ("large" with d_{model}=4096), which is used in several recent works [Lan et al., 2020; Sun et al., 2021; He et al., 2020], and a GPT-3-scale model with d_{model}=12288 [Brown et al., 2020a]. We also evaluate a modified Transformer architecture ("Ours") as defined in Section 6.4.3 with d_{model}=4096, 3 layers per pipeline stage and 8-bit quantized activations. This compression strategy can significantly reduce network usage with little effect on convergence. In the first three configurations, the model consists of 12 Transformer layers placed on 12 servers with a single GPU; in the last one, there are 4 servers, each hosting 3 layers.

As depicted in Figure 6.1 (right) and Figure 6.3, larger models achieve better GPU utilization rate in the same network conditions, since their communication load grows slower than computation. More importantly, even at 500 Mb/s, the resulting GPU idle time can be pushed into the 10–20% range, either naturally for GPT-3-sized models or through activation compression for smaller models. In addition, large models maintain most of their training efficiency at the 100ms latency (Table 6.1), which is roughly equivalent to training on different continents [Verizon, 2021].

6.4.2 Detailed performance comparison

Here we investigate how SWARM parallelism compares to existing systems for training large models: **GPipe** [Huang et al., 2019] and **ZeRO-Offload** [Ren et al., 2021]. The purpose of this section is to compare

the training throughput in "ideal" conditions (with homogeneous reliable devices and balanced layers), as deviating from these conditions makes it *infeasible* to train with baseline systems.

We evaluate training performance for sequences of 4 Transformer layers of identical size distributed over 16 workers. The pipeline does not contain embeddings or language modeling heads, as it would result in imbalance between the stages. Similarly to Section 6.4.1, we use two layer configurations: "xxlarge" (d_{model}=4096, d_{FFN}=16384, 32 heads) and "GPT-3" (d_{model}=12288, d_{FFN}=49152, 96 heads). The microbatch size is 4 for "xxlarge" and 1 for "GPT-3", and the sequence length is 512.

To provide a more detailed view of the training performance, we measure two separate performance statistics: the training throughput and the All-Reduce time. The training throughput measures the rate at which the system can process training sequences, i.e., run forward and backward passes. In turn, the All-Reduce time is the time each system spends to aggregate those accumulated gradients across devices. The total time per step can be computed as `batch_size / throughput + all_reduce_time`. Intuitively, training with small batch sizes is more sensitive to the All-Reduce time (since the algorithm needs to run All-Reduce more frequently) and vice versa.

Hardware setup: Each worker uses a V100-PCIe GPU with 16 CPU threads (E5 v5-2660v4) and 128 GB RAM. The only exception is for ZeRO-Offload with "GPT-3" layers, where we had to double the RAM size because the system required 190GB at peak. Similarly to Section 6.4.1, each worker can communicate at a 500 Mb/s bandwidth for both upload and download for a total of 1 Gb/s. In terms of network latency, we consider two setups: with **no latency**, where workers communicate normally within the same rack, and with **latency**, where we introduce additional 100 ± 50ms latency directly in the kernel[3].

GPipe configuration: We use a popular PyTorch-based implementation of GPipe[4]. The model is partitioned into 4 stages repeated over 4 model-parallel groups. To fit into the GPU memory for the "GPT-3" configuration, we offload the optimizer into RAM using ZeRO-Offload. Before averaging, we use PyTorch's built-in All-Reduce to aggregate gradients. We evaluate both the standard GPipe schedule and the 1F1B schedule [Narayanan et al., 2019].

ZeRO-Offload configuration: Each worker runs the entire model individually, then exchanges gradients with peers. For "xxlarge", we use the official implementation from Ren et al. [2021]. However, for

[3]More specifically, `tc qdisc add dev <...> root netem delay 100ms 50ms`

[4]The source code is available at `https://github.com/kakaobrain/torchgpipe`

Table 6.2: Training performance for different model sizes.

System	Throughput, samples/s		All-Reduce time, s/round	
	No latency	Latency	No latency	Latency
		"GPT-3"		
SWARM	0.619	**0.558**	441.7	**455.4**
GPipe	0.633	0.477	**403**	469.6
1F1B	**0.638**	0.482		
Offload	0.382	0.382	1527.9	1635.4
		"xxlarge"		
SWARM	2.358	2.161	45.36	**51.269**
GPipe	2.541	0.957	**44.17**	64.828
1F1B	2.550	0.987		
Offload	**3.08**	**3.08**	168.71	252.26

"GPT-3", we found that optimizer offloading still does not allow us to fit 4 layers into the GPU. For this reason, we also offload the model parameters using the `offload_param` option.

In turn, when training smaller models, ZeRO-Offload outperforms both SWARM and GPipe. This result aligns with our earlier observations in Figure 6.1, where the same model spent most of the time waiting for the communication between pipeline stages.

We also observe that ZeRO-Offload takes longer to aggregate gradients, likely because each peer must aggregate the entire model, whereas in SWARM and GPipe, peers aggregate a single pipeline stage. The variation between All-Reduce time in GPipe and SWARM is due to implementation differences. Overall, SWARM is competitive to HPC baselines even in an idealized homogeneous environment.

6.4.3 Large-scale distributed training

To verify the efficiency of SWARM parallelism in a practical scenario, we conduct a series of large-scale distributed experiments using preemptible (unreliable) cloud T4 and A100 GPUs over a public cloud network.

We train a Transformer language model with the architecture similar to prior work [Brown et al., 2020a; **?**; **?**] and 1.01 billion parameters in total. Our model consists of 3 stages, each containing a single Transformer decoder block with $d_{model} = 4096$ and 16 layers per pipeline stage. All workers within a stage serve the same group of layers, and all layers within each group use the same set of parameters, similarly to ALBERT [Lan et al., 2020]. On top of this, the first stage also contains the embedding layer, and the

Table 6.3: Pipeline throughput, layer sharing.

Hardware setup	Throughput, samples/s		Optimal bandwidth, Mb/s	
	Actual	Best-case	Upload	Download
T4	17.6	19.2	317.8	397.9
A100	16.9	25.5	436.1	545.1
T4 & A100	27.3	—	—	—

Table 6.4: Pipeline throughput, default Transformer.

Hardware setup	Throughput, samples/s	
	Actual	Best-case
T4	8.8	288.1
A100	8.0	382.5
T4 & A100	13.4	—

last stage includes the language modeling head. Because of layer sharing, this model is equivalent to a 13B model from [Brown et al., 2020a] in terms of compute costs.

We use 8-bit compression [Dettmers et al., 2021] for activations and gradients to reduce the communication intensity. SWARM nodes run rebalancing every $T = 300$ seconds, and trainers measure peer performance using a moving average with $\alpha = 0.1$. SWARM is not very sensitive to the choice of these hyperparameters.

First, to verify that model parallelism with asynchronous updates does not have significant convergence issues, we train the model on the Pile [Gao et al., 2020] dataset with 400 preemptible T4 instances, each hosting one accelerator. As a baseline, we use regular data-parallel training with offloading on 128 A100 GPUs. We run both experiments for approximately 4 weeks and compare the learning curves.

Figure 6.4 shows the results of this experiment: it can be seen that the training dynamics of two approaches are indeed similar, which demonstrates the viability of SWARM parallelism for heterogeneous and poorly-connected devices.

In the next experiment, we aim to measure the pipeline throughput in different hardware conditions and to compare it with an estimate of best-case pipeline performance. We consider several setups: first, we use the same 400 preemptible T4 nodes; in another setup, we use 7 instances with 8 A100 GPU each; finally, we combine these fleets to create a heterogeneous setup. We examine the performance of the pipeline both with weight sharing and with standard, more common, Transformer blocks.

We measure the number of randomly generated samples processed by the pipeline both in our infrastructure and the ideal case that ignores all network-related operations (i.e., has infinite bandwidth and zero latency). The ideal case is emulated by executing a single pipeline stage 3 times locally on a single server and multiplying the single-node estimates by the number of nodes.

As demonstrated in the left two columns of Table 6.3 and Table 6.4, asynchronous training of compute-intensive models with 8-bit compressed activations regardless of the architecture specifics allows us to achieve high performance without a dedicated networking solution. Furthermore, the load balancing algorithm of SWARM allows us to dynamically and efficiently utilize different hardware without being bottlenecked by slower devices.

Next, we use the same load testing scenario to estimate the bandwidth required to fully utilize each device type in the above infrastructure. For this, we measure the average incoming and outgoing bandwidth on the nodes that serve the intermediate stage of the pipeline. We summarize our findings in the right two columns of Table 6.3: it turns out that with layer sharing and 8-bit compression, medium-performance GPUs (such as T4) can be saturated even with moderate network speeds. Based on our main experiment, the optimal total bandwidth is roughly 100Mb/s higher than the values reported in Table 3 due to gradient averaging, loading state from peers, maintaining the DHT and streaming the training data. Although training over the Internet with more efficient hardware might indeed underutilize the accelerator, this issue can be offset by advanced compression strategies such as compression-aware architectures or layer sharing, as shown in Table 6.3.

6.4.4 Adaptive rebalancing evaluation

Lastly, we validate the efficiency of the peer rebalancing algorithm proposed in Section 6.3.2. We use statistics of the number of active T4 nodes from the 32-hour segment of the experiment described in the beginning of this section. We compare our strategy with a baseline that has no rebalancing and with an always optimal strategy. Main details and analysis of the results shown are in Figure 6.5 and Table 6.5. Notably, our strategy provides a significant improvement over the baseline that grows over time, and this improvement persists even with infrequent rebalancing.

Table 6.5: Relative throughput comparison of pipeline rebalancing methods.

Rebalancing	% of optimal		
	Overall	First 1h	Last 1h
None	82.7	99.0	45.4
$T = 300$	95.8	99.4	88.9
$T = 60$	97.6	99.8	91.7

6.5 Conclusion

In this work, we evaluate the feasibility of high-throughput training of billion-scale neural networks on unreliable peers with low network bandwidth. We find that this is feasible by training very large models with pipeline parallelism. To this end, we propose SWARM parallelism to overcome the challenges of pipeline parallelism for preemptible devices with heterogeneous network bandwidths and computational throughputs. We show that SWARM parallelism is highly effective at rebalancing peers and maximizing the aggregate training throughput. We also show that training **large models** with **SWARM parallelism** and **compression**-aware architectures enables high utilization of cheap preemptible instances with slow interconnect. As such, our work makes training of large models accessible to researchers that do not have access to dedicated compute infrastructure.

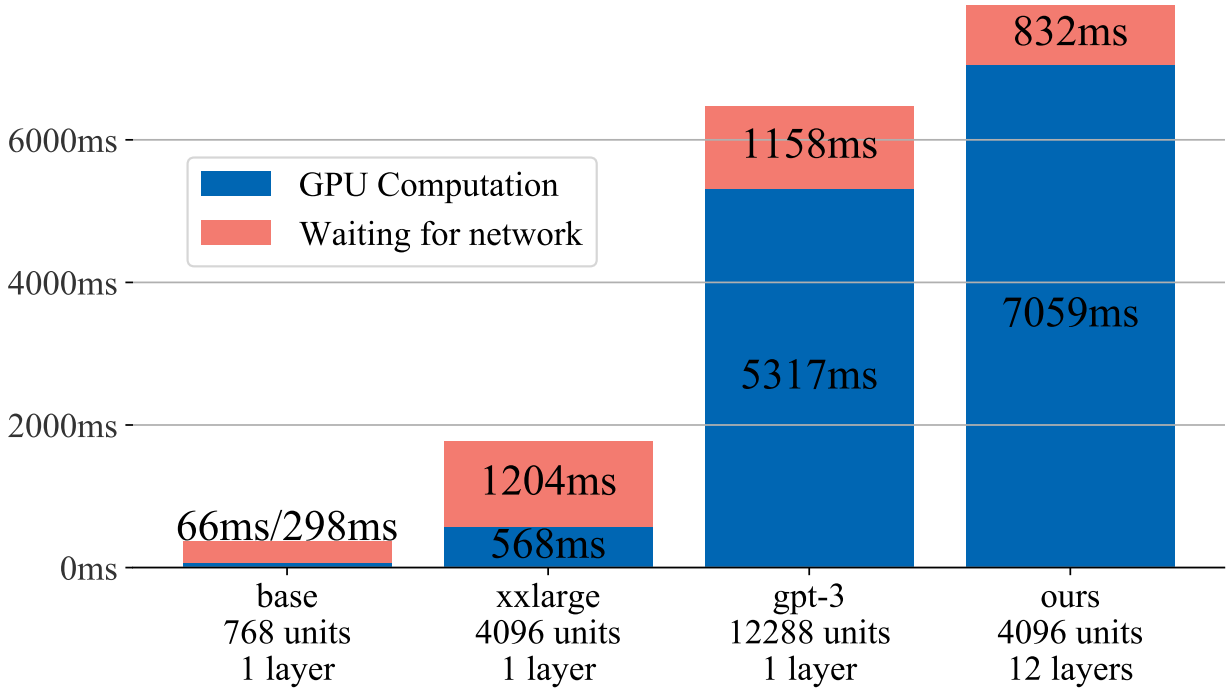


Figure 6.3: Pipeline computation and idle time per batch at 500 Mb/s bandwidth.

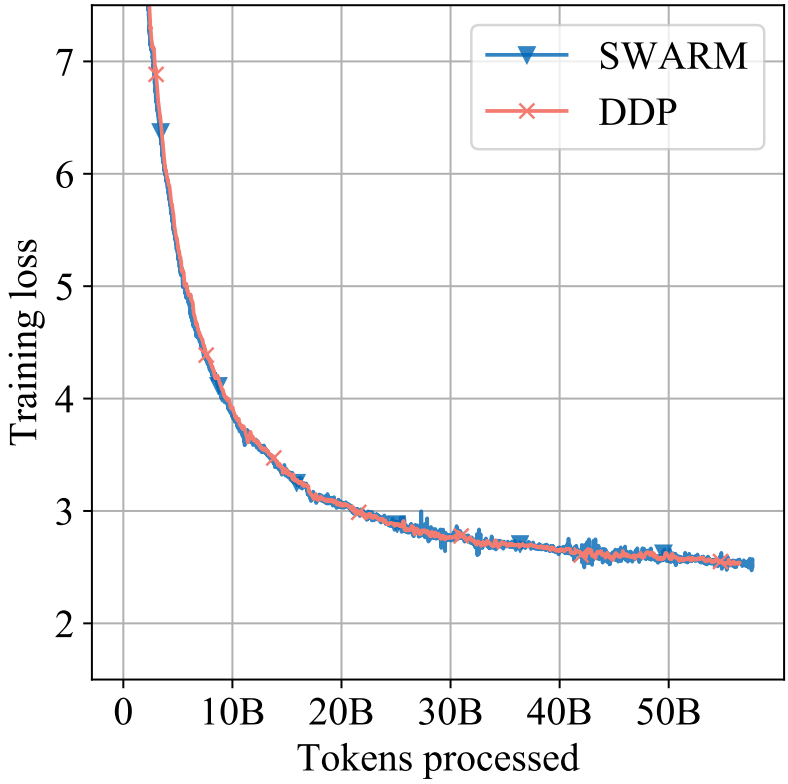


Figure 6.4: Training convergence comparison.

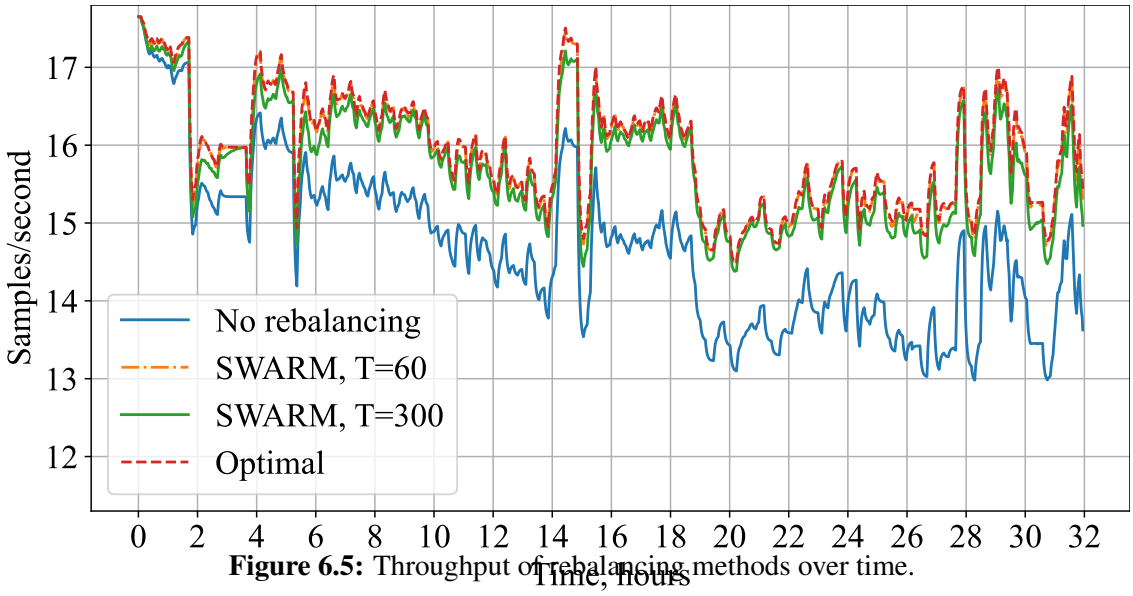


Figure 6.5: Throughput of rebalancing methods over time.

Chapter 7

Conclusion

In this thesis, we have shown that despite their enormous size, foundation models can be made more accessible through a variety of means. We have shown that quantization is an effective means of making inference and finetuning more accessible, while a combination of algorithms can be used to partition the training of foundation models so that globally collaborative training is efficient and effective. All these solutions also always preserve the quality of regular 16-bit approaches and, as such, yield efficiency at no cost to quality.

Reflecting on methodology There are two main approaches that were used in the work in this thesis compared to other work in this area. The first is a careful analysis of failure points and information processing patterns, which in turn led to understanding helpful for the design of methods that overcome difficulties such as instability during quantization. The approach of analysis, until the exact source of the problem is identified, leads to solutions that are generalized well. The second main approach was both broad and deep experimentation with a focus on highly optimized baselines. These experiments often were broader and deeper compared to other work, but the additional experiments led to highly robust results that helped with the adoption of both methods and ideas.

In more detail, the first important step in developing the contributions in this thesis has been to derive a general understanding of information processing patterns in foundation models and how amiable they are for optimization. For example, the analysis and understanding of emerging outliers in large language models (LLMs), how they disrupt quantization, and how these outlier patterns generalize to a wide variety of models have been key to developing the first solution that allows for full LLMs to be run with 8-bit matrix

multiplication without performance degradation.

To develop 4-bit finetuning (QLoRA), it was first necessary to understand how to best quantization LLMs to maximize performance density per bit. As such, the thorough analysis of k-bit inference scaling laws was a strict requirement for the development of QLoRA. Furthermore, analysis of instability patterns was key to the development of QLoRA. In particular, while 4-bit gradients updating 4-bit weights resulted in very unstable finetuning, the addition of LoRA allowed for 16-bit weights to be updated by 4-bit gradients from the frozen base model. This insight, which was borne from the instability analysis, was critical to develop a method that allows for stable 4-bit finetuning.

Finally, to achieve efficient globally distributed training of foundation models, the analysis of often co-dependent components was required: the Square Cube Law showing that larger models are more communication efficient, trade-offs for compressing communication points through quantization vs bottlenecks, and the partition of different parallelization techniques coupled with 1-step stale stochastic gradient descent. As such, in-depth analysis has been key to developing the scientific contributions of this thesis.

Another important step in developing this research was to take a highly empirical approach. The contributions of LLM.in8(), k-bit inference scaling laws, and QLoRA where built by experimenting with a broad variety of open source models. This approach of using a breadth of models combined with extensive hyperparameter search for both baselines and new algorithms led to results that generalized well. Because of this reliability, these contributions have been adopted widely and built upon. As such, experimenting with both a large variety of models and also large variety of hyperparameter settings led to a high robustness.

Summary of results. The key results of this thesis can be summarized as follows.

LLM.int8() showed that large foundation models can be run in 8-bit without performance degradation compared to the standard 16-bit precision. This was achieved by characterizing and isolating outlier hidden states dimensions and their associated weights while all remaining 99.9% of values are quantized to 8-bit.

The k-bit inference scaling laws work used 16-bit inputs, but determined empirically which quantization setup gives the most performance per bit in the model parameters. The main findings are: 4-bit is optimal for most models, a quantization blocksize of 128 or lower is best, and the best data type are float or information theoretically optimal data types.

SpQR is a followup work that showed that k-bit inference scaling laws can be improved from achieving

the highest performance density at around 3.4 bits per parameter. This is achieved through a sparse mask that isolated the outlier values that have the large effect of degrading the performance.

QLoRA applies the insights from k-bit inference scaling laws to finetuning and combines them with LoRA for finetuning stability and achieving higher memory efficiency. It is the first finetuning method that works with a 4-bit base model while also achieving the same performance as 16-bit finetuning. Also, based on the insights from k-bit inference scaling laws, we introduce a data type, NF4, which is an information-theoretically optimal data type for normal distribution that largely contributes to the strong finetuning performance.

Finally, in SWARM we show that carefully composed algorithms, quantization, and system optimizations can lead to efficient training across continents that is 75% as fast as supercomputer training in a single location while also maintaining the same performance as the regular stochastic gradient descent training algorithm.

7.1 Broader Impacts

The broader impact of work in this thesis can be mainly characterized through the impact of the bitsandbytes[1] library. We created the bitsandbytes library to make it easy to use the research algorithms presented in this thesis. With roughly 1.6M installations each month and a total of 18M installations, bitsandbytes, these algorithms are widely used by researchers and practitioners.

Beyond practical impact, the ideas presented in this thesis had a large impact on their own. For example, LLM.int8(), with its characterization of outliers, led to follow-up work that is very distinctly built around these outlier patterns. Such work includes AWQ [Lin et al., 2023] and SmoothQuant [Xiao et al., 2022], QuaRot [Ashkboos et al., 2024], AttentionSinks [Xiao et al., 2023] all of which are influential on their own.

The insights from k-bit inference scaling laws also have been widely been adopted. After our work, it became standard to use a block size of 128 or lower, and to achieve the best performance when limited memory was available, 4-bit quantization is standard, while the precision for best performance has come down to 6-bit – exactly as outlined by our best practices outlined in our k-bit inference scaling law work.

[1] `https://github.com/TimDettmers/bitsandbytes`

7.2 Future work

While the work presented in this thesis makes large foundation models much more accessible for inference, finetuning, and training, the main challenge that remains is that models still get larger and larger. With the upcoming Llama 3 406B[2] the model size of strong open source models increases 5x and 50x compared to the most commonly used 8B parameter Llama 3 model which has been downloaded 3.2M times in the last seven days[3]. This compared to 4-bit inference and QLoRA improving the memory efficiency by a factor of 4x and 18x. Therefore, model capacity grows as fast, or faster when compared to the most commonly used models, then progress in accessibility.

Due to these trends, it is critical to further improve the accessibility of these large models. However, quantization, while being very successful at improving accessibility, has diminishing returns, and new methods are needed to improve accessibility. A very promising future direction is to focus on efficiency through specialization. For example, QLoRA enables specialization and personalization with just 1% additional parameters compared to the base model. As such, one could have a single base model with 100s of adapters that are swapped from and to disk as required for the particular task or domain. With this approach, the main goal would be to distill the performance of a large model, for example, Llama 3 406B model, into a smaller model, for example, Llama 3 8B model, that is specialized for a domain or task. This could be done by generating synthetic data from the large model to finetune the smaller model [Wang et al., 2022b].

[2] https://ai.meta.com/blog/meta-llama-3/
[3] https://huggingface.co/models

Bibliography

David H. Allen. 2013. *How Mechanics Shaped the Modern World*.

Josh Alman and Virginia Vassilevska Williams. 2021. A refined laser method and faster matrix multiplication. In *SODA*.

Yossi Arjevani, Ohad Shamir, and Nathan Srebro. 2020. A tight convergence analysis for stochastic gradient descent with delayed updates. In *Proceedings of the 31st International Conference on Algorithmic Learning Theory*, volume 117 of *Proceedings of Machine Learning Research*, pages 111–132. PMLR.

Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, et al. 2021. Efficient large scale language modeling with mixtures of experts. *arXiv preprint arXiv:2112.10684*.

Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L Croci, Bo Li, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. 2024. Quarot: Outlier-free 4-bit inference in rotated llms. *arXiv preprint arXiv:2404.00456*.

Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, et al. 2021. A general language assistant as a laboratory for alignment. *arXiv preprint arXiv:2112.00861*.

Medha Atre, Birendra Jha, and Ashwini Rao. 2021. Distributed deep learning using volunteer computing-like paradigm.

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain,

Stanislav Fort, Deep Ganguli, Tom Henighan, et al. 2022. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*.

Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4).

Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. *arXiv preprint arXiv:2304.01373*.

Yonatan Bisk, Rowan Zellers, Ronan LeBras, Jianfeng Gao, and Yejin Choi. 2020. PIQA: reasoning about physical commonsense in natural language. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 7432–7439. AAAI Press.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. 2022. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*.

Yelysei Bondarenko, Markus Nagel, and Tijmen Blankevoort. 2021. Understanding and overcoming the challenges of efficient transformer quantization. *arXiv preprint arXiv:2109.12948*.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020a. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind

Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020b. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Jianfei Chen, Yu Gai, Zhewei Yao, Michael W Mahoney, and Joseph E Gonzalez. 2020. A statistical framework for low-bitwidth training of deep neural networks. *Advances in Neural Information Processing Systems*, 33:883–894.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. 2023. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality.

Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, Broomfield, CO. USENIX Association.

Yoojin Choi, Mostafa El-Khamy, and Jungwon Lee. 2016. Towards the limit of network quantization. *arXiv preprint arXiv:1612.01543*.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern,

Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022a. Palm: Scaling language modeling with pathways.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022b. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. 2022. Scaling instruction-finetuned language models. *arXiv preprint arXiv:2210.11416*.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Don Coppersmith and Shmuel Winograd. 1990. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280. Computational algebraic complexity editorial.

Zihang Dai, Hanxiao Liu, Quoc V. Le, and Mingxing Tan. 2021. Coatnet: Marrying convolution and attention for all data sizes. *ArXiv*, abs/2106.04803.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. 2012. Large scale distributed deep networks. *Advances in neural information processing systems*, 25:1223–1231.

Tim Dettmers. 2015. 8-bit approximations for parallelism in deep learning. *ICLR*.

Tim Dettmers. 2016. 8-bit approximations for parallelism in deep learning. *International Conference on Learning Representations (ICLR)*.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022a. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2021. 8-bit optimizers via block-wise quantization. *CoRR*, abs/2110.02861.

Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2022b. 8-bit optimizers via block-wise quantization. *9th International Conference on Learning Representations, ICLR*.

Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. 2023. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*.

Tim Dettmers and Luke Zettlemoyer. 2022. The case for 4-bit precision: k-bit inference scaling laws. *arXiv preprint arXiv:2212.09720*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *NAACL-HLT*.

Prafulla Dhariwal and Alex Nichol. 2021. Diffusion models beat gans on image synthesis. *CoRR*, abs/2105.05233.

Michael Diskin, Alexey Bukhtiyarov, Max Ryabinin, Lucile Saulnier, Quentin Lhoest, Anton Sinitsin, Dmitriy Popov, Dmitry Pyrkin, Maxim Kashirin, Alexander Borzunov, Albert Villanova del Moral, Denis Mazur, Ilia Kobelev, Yacine Jernite, Thomas Wolf, and Gennady Pekhimenko. 2021. Distributed deep learning in open collaborations. *CoRR*, abs/2106.10207.

Jack Dongarra. 2022. A not so simple matter of software. `https://www.youtube.com/watch?v=cSO0Tc2w5Dg`.

Arpad E Elo. 1967. The proposed uscf rating system. its development, theory, and applications. *Chess Life*, 22(8):242–247.

Arpad E Elo. 1978. *The rating of chessplayers, past and present*. Arco Pub.

Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. 2004. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. pages 133–137.

William Fedus, Barret Zoph, and Noam Shazeer. 2021. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*.

Elias Frantar and Dan Alistarh. 2023. Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022a. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*.

Elias Frantar, Sidak Pal Singh, and Dan Alistarh. 2022b. Optimal Brain Compression: A framework for accurate post-training quantization and pruning. *arXiv preprint arXiv:2208.11580*. Accepted to NeurIPS 2022, to appear.

Kunihiko Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36:193–202.

Galilei Galileo. 1638. *Discorsi e dimostrazioni matematiche intorno a due nuove scienze*.

Jun Gao, Di He, Xu Tan, Tao Qin, Liwei Wang, and Tie-Yan Liu. 2019. Representation degeneration problem in training natural language generation models. *arXiv preprint arXiv:1907.12009*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Laria Reynolds, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 2021. A framework for few-shot language model evaluation.

Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2021. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*.

Aaron Gokaslan and Vanya Cohen. 2019. Openwebtext corpus. *urlhttp://Skylion007. github. io/OpenWebTextCorpus*.

Yunchao Gong, Liu Liu, Ming Yang, and Lubomir Bourdev. 2014. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*.

Andreas Griewank and Andrea Walther. 2000. Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 26(1):19–45.

Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. 2017. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 589–604, New York, NY, USA. Association for Computing Machinery.

Mark Harris, Shubhabrata Sengupta, and John D. Owens. 2007. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley.

Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. Deberta: Decoding-enhanced bert with disentangled attention.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B Brown, Prafulla Dhariwal, Scott Gray, et al. 2020. Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad,

Md Patwary, Mostofa Ali, Yang Yang, and Yanqi Zhou. 2017. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*.

S. Hochreiter and J. Schmidhuber. 1995. Long Short-Term Memory. Technical Report FKI-207-95, Fakultät für Informatik, Technische Universität München. Revised 1996 (see www.idsia.ch/˜juergen, www7.informatik.tu-muenchen.de/˜hochreit).

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022a. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022b. Training compute-optimal large language models.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. 2022. Unnatural instructions: Tuning language models with (almost) no human labor. *arXiv preprint arXiv:2212.09689*.

Lu Hou, Quanming Yao, and James T Kwok. 2016. Loss-aware binarization of deep networks. *arXiv preprint arXiv:1611.01600*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. 2020. Strassen's algorithm reloaded on gpus. *ACM Trans. Math. Softw.*, 46(1).

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112.

Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. 2021. Accurate post training quantization with small calibration sets. In *International Conference on Machine Learning (ICML)*.

Gabriel Ilharco, Cesar Ilharco, Iulia Turc, Tim Dettmers, Felipe Ferreira, and Kenton Lee. 2020. High performance natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Tutorial Abstracts*, pages 24–27, Online. Association for Computational Linguistics.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and training of neural networks for efficient integer-arithmetic-only inference. arxiv e-prints, art. *arXiv preprint arXiv:1712.05877*.

Sambhav Jain, Albert Gural, Michael Wu, and Chris Dick. 2020. Trained quantization thresholds for accurate and efficient fixed-point inference of deep neural networks. *Proceedings of Machine Learning and Systems*, 2:112–128.

Sebastian Jaszczur, Aakanksha Chowdhery, Afroz Mohiuddin, Lukasz Kaiser, Wojciech Gajewski, Henryk Michalewski, and Jonni Kanerva. 2021. Sparse is enough in scaling transformers. *Advances in Neural Information Processing Systems*, 34:9895–9907.

Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019a. Dissecting the nvidia turing t4 gpu via microbenchmarking. *arXiv preprint arXiv:1903.07486*.

Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019b. Beyond data and model parallelism for deep neural networks. In *Proceedings of Machine Learning and Systems*, volume 1, pages 1–13.

John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, et al. 2021. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. 1991. Weighted round-robin cell multiplexing in a general-purpose atm switch chip. *IEEE Journal on Selected Areas in Communications*, 9(8):1265–1279.

Daya Khudia, Jianyu Huang, Protonu Basu, Summer Deng, Haixin Liu, Jongsoo Park, and Mikhail Smelyanskiy. 2021. Fbgemm: Enabling high-performance low-precision deep learning inference. *arXiv preprint arXiv:2101.05615*.

Ekasit Kijsipongse, Apivadee Piyatumrong, and Suriya U-ruekolan. 2018. A hybrid gpu cluster and volunteer computing platform for scalable deep learning. *The Journal of Supercomputing*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*.

Abdullatif Köksal, Timo Schick, Anna Korhonen, and Hinrich Schütze. 2023. Longform: Optimizing instruction tuning for long text generation with corpus extraction. *arXiv preprint arXiv:2304.08460*.

Andreas Köpf, Yannic Kilcher, Dimitri von Rütte, Sotiris Anagnostidis, Zhi-Rui Tam, Keith Stevens, Abdullah Barhoum, Nguyen Minh Duc, Oliver Stanley, Richárd Nagyfi, et al. 2023. Openassistant conversations–democratizing large language model alignment. *arXiv preprint arXiv:2304.07327*.

Olga Kovaleva, Saurabh Kulshreshtha, Anna Rogers, and Anna Rumshisky. 2021. Bert busters: Outlier dimensions that disrupt transformers. *arXiv preprint arXiv:2105.06990*.

Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*.

Alex Krizhevsky. 2014. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

LAION. 2023. Open-instruction-generalist dataset. `https://github.com/LAION-AI/Open-Instruction-Generalist`.

Zhen-Zhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. Albert: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*.

Jennifer Langston. 2020. Microsoft announces new supercomputer, lays out vision for future ai work. https://blogs.microsoft.com/ai/openai-azure-supercomputer/. Accessed: 2021-10-1.

Verónica Larrea, Wayne Joubert, Michael Brim, Reuben Budiardja, Don Maxwell, Matt Ezell, Christopher Zimmer, Swen Boehm, Wael Elwasif, Sarp Oral, Chris Fuson, Daniel Pelfrey, Oscar Hernandez, Dustin Leverman, Jesse Hanley, Mark Berrill, and Arnold Tharrington. 2019. *Scaling the Summit: Deploying the World's Fastest Supercomputer*, pages 330–351.

Cong Leng, Zesheng Dou, Hao Li, Shenghuo Zhu, and Rong Jin. 2018. Extremely low bit neural network: Squeeze the last bit out with admm. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Shijian Li, Robert J Walls, Lijie Xu, and Tian Guo. 2019. Speeding up deep learning with transient servers. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*, pages 125–135. IEEE.

Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. 2021. BRECQ: Pushing the limit of post-training quantization by block reconstruction. In *International Conference on Learning Representations (ICLR)*.

Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. 2023. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*.

Jiahuang Lin, Xin Li, and Gennady Pekhimenko. 2020a. Multi-node bert-pretraining: Cost-efficient approach.

Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. 2021. A survey of transformers.

Ye Lin, Yanyang Li, Tengbo Liu, Tong Xiao, Tongran Liu, and Jingbo Zhu. 2020b. Towards fully 8-bit integer inference for the transformer model. *arXiv preprint arXiv:2009.08034*.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2018. Deep Gradient Compression:

Reducing the communication bandwidth for distributed training. In *The International Conference on Learning Representations*.

Zeming Lin, Halil Akin, Roshan Rao, Brian Hie, Zhongkai Zhu, Wenting Lu, Allan dos Santos Costa, Maryam Fazel-Zarandi, Tom Sercu, Sal Candido, et al. 2022. Language models of protein sequences at the scale of evolution enable accurate structure prediction. *BioRxiv*, 2022:500902.

Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. 2022. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35:1950–1965.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V Le, Barret Zoph, Jason Wei, et al. 2023. The flan collection: Designing data and methods for effective instruction tuning. *arXiv preprint arXiv:2301.13688*.

Ziyang Luo, Artur Kulmizev, and Xiaoxi Mao. 2021. Positional artefacts propagate through masked language model embeddings. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 5312–5327, Online. Association for Computational Linguistics.

Mitch Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The penn treebank: Annotating predicate argument structure. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*.

Petar Maymounkov and David Mazieres. 2002. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*.

Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2021. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943*.

Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. 2020. Up or down? Adaptive rounding for post-training quantization. In *International Conference on Machine Learning (ICML)*.

Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. 2019. Data-free quantization through weight equalization and bias correction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1325–1334.

Sebastian Nagel. 2016. Cc-news.

Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. Pipedream: Generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 1–15, New York, NY, USA. Association for Computing Machinery.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters. *arXiv preprint arXiv:2104.04473*.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. 2022. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*.

OpenAI. 2023. Gpt-4 technical report. *arXiv*.

Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang,

Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Ngoc Quan Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. 2016. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1525–1534, Berlin, Germany. Association for Computational Linguistics.

Eunhyeok Park, Junwhan Ahn, and Sungjoo Yoo. 2017. Weighted-entropy-based quantization for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5456–5464.

Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. 2022. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019a. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019b. Pytorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*.

Felix Petersen and Tobias Sutter. 2023. Distributional Quantization.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*.

Giovanni Puccetti, Anna Rogers, Aleksandr Drozd, and Felice Dell'Orletta. 2022. Outliers dimensions that disrupt transformers are driven by frequency. *arXiv preprint arXiv:2205.11380*.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Jack W. Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, Eliza Rutherford, Tom Hennigan, Jacob Menick, Albin Cassirer, Richard Powell, George van den Driessche, Lisa Anne Hendricks, Maribeth Rauh, Po-Sen Huang, Amelia Glaese, Johannes Welbl, Sumanth Dathathri, Saffron Huang, Jonathan Uesato, John Mellor, Irina Higgins, Antonia Creswell, Nat McAleese, Amy Wu, Erich Elsen, Siddhant Jayakumar, Elena Buchatskaya, David Budden, Esme Sutherland, Karen Simonyan, Michela Paganini, Laurent Sifre, Lena Martens, Xiang Lorraine Li, Adhiguna Kuncoro, Aida Nematzadeh, Elena Gribovskaya, Domenic Donato, Angeliki Lazaridou, Arthur Mensch, Jean-Baptiste Lespiau, Maria Tsimpoukelli, Nikolai Grigorev, Doug Fritz, Thibault Sottiaux, Mantas Pajarskas, Toby Pohlen, Zhitao Gong, Daniel Toyama, Cyprien de Masson d'Autume, Yujia Li, Tayfun Terzi, Vladimir Mikulik, Igor Babuschkin, Aidan Clark, Diego de Las Casas, Aurelia Guy, Chris Jones, James Bradbury, Matthew Johnson, Blake Hechtman, Laura Weidinger, Iason Gabriel, William Isaac, Ed Lockhart, Simon Osindero, Laura Rimell, Chris Dyer, Oriol Vinyals, Kareem Ayoub, Jeff Stanway, Lorrayne Bennett, Demis Hassabis, Koray Kavukcuoglu, and Geoffrey Irving. 2021. Scaling language models: Methods, analysis & insights from training gopher.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter Liu. 2020a. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2019. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020b. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1).

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimization towards training a trillion parameter models. In *SC*.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*.

Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 525–542. Springer.

Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459. IEEE.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training.

Jonathan S Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. 2019. A constructive prediction of the generalization error across scales. *arXiv preprint arXiv:1909.12673*.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature*, 323:533–536.

Manuele Rusci, Alessandro Capotondi, and Luca Benini. 2020. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *Proceedings of Machine Learning and Systems*, 2:326–335.

Max Ryabinin, Eduard Gorbunov, Vsevolod Plokhotnyuk, and Gennady Pekhimenko. 2021. Moshpit sgd: Communication-efficient decentralized training on heterogeneous unreliable devices.

Max Ryabinin and Anton Gusev. 2020. Towards crowdsourced training of large neural networks using decentralized mixture-of-experts. In *Advances in Neural Information Processing Systems*, volume 33, pages 3659–3672. Curran Associates, Inc.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: an adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106.

Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. 2022. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake A. Hechtman. 2018a. Mesh-tensorflow: Deep learning for supercomputers. *CoRR*, abs/1811.02084.

Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, et al. 2018b. Mesh-tensorflow: Deep learning for supercomputers. *Advances in neural information processing systems*, 31.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. 2017. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8815–8821.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Sebastian U. Stich and Sai Praneeth Karimireddy. 2020. The error-feedback framework: Sgd with delayed gradients. *Journal of Machine Learning Research*, 21(237):1–36.

Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. 2021. Fugaku. https://www.top500.org/system/179807/. Estimated energy consumption 29,899.23 kW. Accessed: 2021-10-4.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2021. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*.

Yu Sun, Shuohuan Wang, Shikun Feng, Siyu Ding, Chao Pang, Junyuan Shang, Jiaxiang Liu, Xuyi Chen, Yanbin Zhao, Yuxiang Lu, Weixin Liu, Zhihua Wu, Weibao Gong, Jianzhong Liang, Zhizhou Shang, Peng Sun, Wei Liu, Xuan Ouyang, Dianhai Yu, Hao Tian, Hua Wu, and Haifeng Wang. 2021. ERNIE 3.0: Large-scale knowledge enhanced pre-training for language understanding and generation. *CoRR*, abs/2107.02137.

Seyed Mohammadhossein Tabatabaee, Jean-Yves Le Boudec, and Marc Boyer. 2020. Interleaved weighted round-robin: A network calculus analysis. In *2020 32nd International Teletraffic Congress (ITC 32)*, pages 64–72.

Zhenheng Tang, Shaohuai Shi, Xiaowen Chu, Wei Wang, and Bo Li. 2020. Communication-efficient distributed deep learning: A comprehensive survey.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. `https://github.com/tatsu-lab/stanford_alpaca`.

Sandeep Tata and Jignesh M Patel. 2003. PiQA: An algebra for querying protein data sets. In *International Conference on Scientific and Statistical Database Management*.

John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. 2022. Bamboo: Making preemptible instances resilient for affordable training of large dnns.

TII UAE. 2023a. The Falcon family of large language models. https://huggingface.co/tiiuae/falcon-40b.

TII UAE. 2023b. The Refined Web dataset. https://huggingface.co/datasets/tiiuae/falcon-refinedweb.

William Timkey and Marten van Schijndel. 2021. All bark and no bite: Rogue dimensions in transformer language models obscure representational quality. *arXiv preprint arXiv:2109.04404*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Trieu H Trinh and Quoc V Le. 2018. A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017a. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017b. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Verizon. 2021. Monthly ip latency data. Accessed: 2021-10-05.

Jesse Vig. 2019. A multiscale visualization of attention in the transformer model. *arXiv preprint arXiv:1906.05714*.

Thijs Vogels, Sai Praneeth Karimireddy, and Martin Jaggi. 2019. Powersgd: Practical low-rank gradient compression for distributed optimization. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14236–14245.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808, Florence, Italy. Association for Computational Linguistics.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Jue Wang, Binhang Yuan, Luka Rimanic, Yongjun He, Tri Dao, Beidi Chen, Christopher Re, and Ce Zhang. 2022a. Fine-tuning language models over slow networks using activation quantization with guarantees. In *Advances in Neural Information Processing Systems*.

Peisong Wang, Qiang Chen, Xiangyu He, and Jian Cheng. 2020. Towards accurate post-training network quantization via bit-split and stitching. In *International Conference on Machine Learning (ICML)*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022b. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, et al. 2022c. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 5085–5109.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Xiuying Wei, Yunchen Zhang, Xiangguo Zhang, Ruihao Gong, Shanghang Zhang, Qi Zhang, Fengwei Yu, and Xianglong Liu. 2022. Outlier suppression: Pushing the limit of low-bit transformer language models. *arXiv preprint arXiv:2209.13325*.

Guillaume Wenzek, Marie-Anne Lachaux, Alexis Conneau, Vishrav Chaudhary, Francisco Guzmán, Armand Joulin, and Edouard Grave. 2020. CCNet: Extracting high quality monolingual datasets from web crawl data. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 4003–4012, Marseille, France. European Language Resources Association.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*.

Mitchell Wortsman, Tim Dettmers, Luke Zettlemoyer, Ari Morcos, Ali Farhadi, and Ludwig Schmidt. 2023. Stable and low-precision training for large-scale vision-language models. *arXiv preprint arXiv:2304.13013*.

Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*.

Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. 2022. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher R. Aberger, and Christopher De Sa. 2019. Pipemare: Asynchronous pipeline parallel dnn training. *ArXiv*, abs/1910.05124.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *arXiv preprint arXiv:2206.01861*.

Zhewei Yao, Cheng Li, Xiaoxia Wu, Stephen Youn, and Yuxiong He. 2023. A comprehensive study on post-training quantization for large language models.

Binhang Yuan, Yongjun He, Jared Quincy Davis, Tianyi Zhang, Tri Dao, Beidi Chen, Percy Liang, Christo-

pher Re, and Ce Zhang. 2022. Decentralized training of foundation models in heterogeneous environments. In *Advances in Neural Information Processing Systems*.

Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. 2019. Q8bert: Quantized 8bit bert. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 36–39. IEEE.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 4791–4800. Association for Computational Linguistics.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2022. Glm-130b: An open bilingual pre-trained model. *arXiv preprint arXiv:2210.02414*.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. 2021. Scaling vision transformers. *CoRR*, abs/2106.04560.

Dongqing Zhang, Jiaolong Yang, Dongqiangzi Ye, and Gang Hua. 2018. Lq-nets: Learned quantization for highly accurate and compact deep neural networks. In *Proceedings of the European conference on computer vision (ECCV)*, pages 365–382.

Peng Zhang and Yuxiang Gao. 2015. Matrix multiplication on high-density multi-gpu architectures: Theoretical and experimental investigations. volume 9137, pages 17–30.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*.

Xiaoxi Zhang, Jianyu Wang, Gauri Joshi, and Carlee Joe-Wong. 2020. Machine learning on volatile instances. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 139–148. IEEE.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27.