# GPTQ: ACCURATE POST-TRAINING QUANTIZATION FOR GENERATIVE PRE-TRAINED TRANSFORMERS

A PREPRINT

**Elias Frantar***
IST Austria
Klosterneuburg, Austria
elias.frantar@ist.ac.at

**Saleh Ashkboos**
ETH Zurich
Switzerland
saleh.ashkboos@inf.ethz.ch

**Torsten Hoefler**
ETH Zurich
Switzerland
htor@inf.ethz.ch

**Dan Alistarh**
IST Austria & Neural Magic, Inc.
Klosterneuburg, Austria
dan.alistarh@ist.ac.at

**ABSTRACT**

Generative Pre-trained Transformer (GPT) models set themselves apart through breakthrough performance across complex language modelling tasks, but also by their extremely high computational and storage costs. Specifically, due to their massive size, even inference for large, highly-accurate GPT models may require multiple performant GPUs to execute, which limits the usability of such models. While there is emerging work on relieving this pressure via model compression, the applicability and performance of existing compression techniques is limited by the scale and complexity of GPT models. In this paper, we address this challenge, and propose GPTQ, a new one-shot weight quantization method based on approximate second-order information, that is both highly-accurate and highly-efficient. Specifically, GPTQ can quantize GPT models with 175 billion parameters in approximately four GPU hours, reducing the bitwidth down to 3 or 4 bits per weight, with negligible accuracy degradation relative to the uncompressed baseline. Our method more than doubles the compression gains relative to previously-proposed one-shot quantization methods, preserving accuracy, allowing us for the first time to execute an 175 billion-parameter model inside a single GPU. We show experimentally that these improvements can be leveraged for end-to-end inference speedups over FP16, of around 2x when using high-end GPUs (NVIDIA A100) and 4x when using more cost-effective ones (NVIDIA A6000). The implementation is available at https://github.com/IST-DASLab/gptq.

## 1 Introduction

Pre-trained generative models from the Transformer [31] family, commonly known as GPT or OPT [26, 4, 35], have shown breakthrough performance for complex language modelling tasks, leading to massive academic and practical interest. One major obstacle to the usability of such models is their computational and storage cost, which ranks among the highest for known models. For instance, the best-performing model variants, e.g. GPT3-175B, have in the order of 175 billion parameters and require tens-to-hundreds of GPU-years to train [35]. Even the simpler task of inferencing over a pre-trained model, which is our focus in this paper, is highly challenging: for example, the parameters of GPT3-175B occupy 326GB of memory (counting in multiples of 1024) when stored in an already-compact float16 format. This exceeds the memory capacity of even the highest-end single GPUs, and thus inference must be performed using more complex and expensive setups, such as multi-GPU deployments.

A standard approach to eliminating these overheads is *model compression*, e.g. [12, 9]. Yet, surprisingly little is known about compressing such models for inference. One reason is that more complex, but accurate methods for low-bitwidth

*Corresponding author.

quantization or model pruning usually require *model retraining*, which is extremely expensive for billion-parameter models. Alternatively, *post-training* methods [20, 32, 13, 22], which compress the model in one shot, without retraining, would be very appealing in this setting. Unfortunately, the more accurate variants of such methods [17, 14, 8] are complex, and do not scale to billions of parameters [34]. To date, only basic variants of round-to-nearest quantization [34, 5] have been applied at the scale of GPT-175B; while this works well for low compression targets, e.g., 8-bit weights, they fail to preserve accuracy at higher rates. It therefore remains open whether one-shot *post-training quantization* to higher compression rates is generally-feasible.

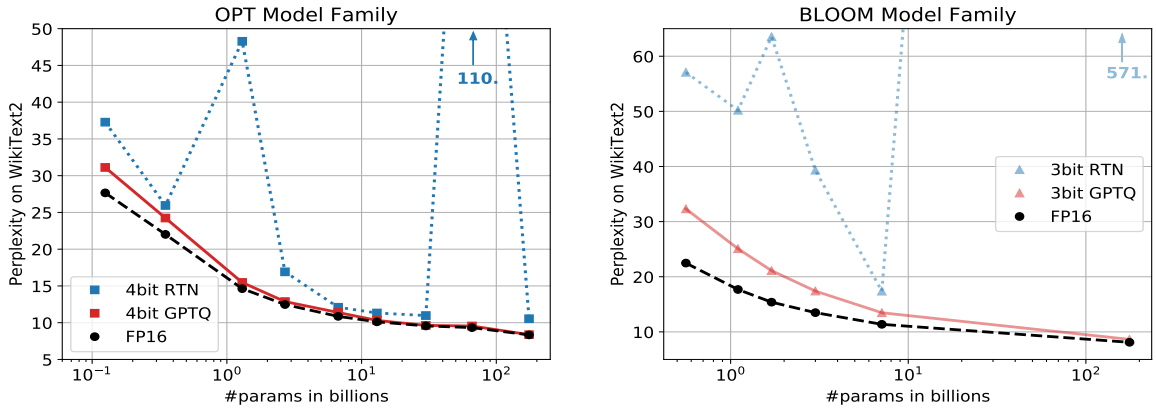


Figure 1: Quantizing OPT models to 4 and BLOOM models to 3 bit precision, comparing GPTQ with the FP16 baseline and round-to-nearest (RTN) [34, 5].

**Contribution.** In this paper, we present a new post-training quantization method, called GPTQ[2], which is efficient enough to execute on models with hundreds of billions of parameters in at most a few hours, and precise enough to compress such models to 3 or 4 bits per parameter without significant loss of accuracy. GPTQ is the first method to leverage approximate second-order (Hessian) information at this scale, and can be implemented extremely efficiently on modern GPUs. For illustration, GPTQ can quantize the largest publicly-available models, OPT-175B and BLOOM-176B, in approximately four GPU hours, with minimal increase in perplexity, known to be a very stringent accuracy metric. In addition, we develop an execution harness which allows us to execute the resulting compressed models efficiently for generative tasks. Specifically, we are able to run language generation on the compressed OPT-175B model for the first time on a single NVIDIA A100 GPU, or using only two more cost-effective NVIDIA A6000 GPUs. We also implement bespoke GPU kernels which are able to leverage compression for faster memory transfer, resulting in speedups of $\approx 2\times$ when using A100 GPUs, and $4\times$ when using A6000 GPUs.

At a high level, our work tackles a standard problem in the low-precision quantization literature: for each layer, we wish to identify an assignment of standard-precision weights to quantization levels, which minimizes the discrepancy between the output of the original layer and that of the compressed one [20, 17, 14, 34, 8], which we approach at unprecedented scale. Usually, this discrepancy is measured in $\ell_2$-distance on a small amount of calibration data. Once such accurate layer-wise weight assignments are found, the layers can be "stitched together" to reconstruct the quantized model. This layer-wise assignment problem is the basis for most state-of-the-art quantization schemes, and is approached via increasingly-sophisticated solvers, using integer programming [13] or second-order approximations [17, 8]. However, existing approaches simply do not scale to large models from the GPT family [34].

The GPTQ algorithm starts from a similar strategy to the recently-proposed Optimal Brain Quantization (OBQ) layer-wise solver [8]. The OBQ approach quantizes weights *one-at-a-time*, always updating the remaining unquantized weights to minimize the layer's output error. Both the choice of the next weight to quantize and the weight update leverage second-order information, which is reasonable to work with given that the output error is quadratic, and the layer Hessian is thus constant. Yet, this approach is impractical at GPT scale: we estimate that quantizing GPT-175B using this approach would take at least 6 months of computation, even if one completely ignores memory constraints.

In this context, the first key idea behind our approach is that there exists a way to quantize weights *in large blocks* that is orders of magnitude more efficient relative to OBQ, but that can still leverage second-order information so that it stays *highly accurate*. Concretely, central to GPTQ is a new second-order quantization algorithm with per-layer runtime $O(d_{\text{col}}^2 \cdot \max\{d_{\text{row}}, d_{\text{col}}\})$, where $d_{\text{row}}$ and $d_{\text{col}}$ are the layer matrix's row and column dimensions, respectively. Asymptotically, this is faster than OBQ by a factor of $\Theta(\min\{d_{\text{row}}, d_{\text{col}}\})$, which is often around $1000\times$ in practice.

[2]This combines the name of the GPT model family with the abbreviation for post-training quantization (PTQ).

Yet, this algorithmic insight is not sufficient for a fast implementation: we additionally resolve a number of practical barriers, including the low compute-to-memory ratio of the above algorithm, and numerical stability issues when working with matrix inverses at this scale.

We implemented the resulting GPTQ algorithm in Pytorch [25], and used it to quantize publicly-available models from the OPT [35] and BLOOM [16] model families, counting from 125M to 176B parameters. Our results show that, both for standard perplexity tasks, e.g., WikiText2 [19] or C4 [27] and for standard zero-shot tasks, e.g. LAMBADA [23], our method is able to compress models down to 4 or even 3 bits per weight with negligible loss of accuracy. We are even able to compress models accurately down to around 2 bits per weight when quantizing at lower granularity. Moreover, GPTQ compresses the largest models in approximately 4 GPU hours, and can execute on a single GPU. As illustrated in Figure 1, relative to prior work, GPTQ is the first method to reliably compress LLMs to 4 bits or less, more than doubling compression at minimal accuracy loss, and allowing for the first time to fit an OPT-175B model for inference inside a single GPU.

To realize speedups in practice, we observe that our method particularly benefits generative inference, which is one of the most interesting use-cases for GPT models. Specifically, generative inference produces outputs in token-by-token fashion, in an autoregressive manner. Thus, the method cannot benefit from batching, and so most of the execution reduces to matrix-vector products, which have low arithmetic intensity and put particular stress on the memory subsystem, as it must transfer the parameter matrices between main memory and the memory of a computational accelerator (e.g., GPU or IPU). By quantizing the model, we allow weight matrices to fit into the faster accelerator-local memory, leading to speedup and usability improvements. We illustrate this technique by implementing a fast GPU kernel, which allows us to maintain weights in quantized format, "decoding" them layer-by-layer at execution time. We show that this approach can result in end-to-end speedups of $1.9 - 4\times$ when executed on modern GPUs.

To our knowledge, we are the first to show that extremely accurate language models with hundreds of billions of parameters can be quantized to $2.5 - 4$ bits per component on average: prior *post-training methods* only remain accurate at 8 bits [34, 5], while prior *training-based* techniques have only tackled models that are smaller by one to two orders of magnitude [33]. This high degree of compression may appear unsurprising, as these networks are overparametrized; yet, as we discuss in our detailed analysis of results, compression induces non-trivial tradeoffs between the accuracy of the language modeling (perplexity), bit-width, and the size of the original model.

We hope that our work will stimulate further research in this area, and can be a further step towards making these models available to a wider audience. In terms of limitations, our method currently does not provide speedups for the actual multiplications, due to the lack of direct hardware support for mixed-precision operands (e.g. FP16 $\times$ INT4) on mainstream architectures. Moreover, our current results do not include activation quantization, as they are not a significant bottleneck in our target scenarios; however, this can be supported using complementary techniques [5, 34].

## 2 Related Work

Quantization methods fall broadly into two categories: quantization during training, and post-training methods. The former methods quantize models during typically extensive retraining and/or finetuning, using some approximate differentiation mechanism for the rounding operation [9, 21]. By contrast, post-training ("one-shot") methods quantize a pretrained model using modest resources, typically a few thousand data samples and a few hours of computation. Post-training approaches are particularly interesting for huge models, for which full model training or even finetuning can be extremely resource intensive. We thus focus on this scenario here.

**Post-training Quantization.** Historically, post-training methods have focused on vision models: for instance, quantizing the ResNet50 model [11] is as a standard benchmark. Usually, accurate methods operate by quantizing either individual layers, or small blocks of consecutive layers. (See Section 3.1 for more details.) The AdaRound method [20] computes a data-dependent rounding by annealing a penalty term, which encourages weights to move towards grid points corresponding to quantization levels. BitSplit [32] constructs quantized values bit-by-bit using a squared error objective on the residual error, while AdaQuant [14] performs optimization of the quantization levels based on straight-through estimation. BRECQ [17] introduces Fisher information into the objective, and optimizes layers within a single residual block jointly. Finally, Optimal Brain Quantization (OBQ) [8] generalizes the classic Optimal Brain Surgeon (OBS) second-order weight pruning framework [10, 29, 7] to apply to quantization. OBQ quantizes weights one-by-one, in order of quantization error, always adjusting the remaining weights. While these approaches can produce good results for models up to $\approx$ 100 million parameters in a few GPU hours, they will not scale to networks orders of magnitude larger. We compare against all these methods on the classic ResNet18/ResNet50 benchmark (see Table 1). Although it is not our goal to compete with these highly-accurate methods, GPTQ is in fact fairly competitive in terms of accuracy, while being at least an order of magnitude faster in practice than these methods, even at this model scale.

**Large-model Quantization.** With the recent open-source releases of models like BLOOM [16] or OPT-175B [35], researchers have started to develop affordable methods for compressing such giant networks for inference. To our knowledge, all existing works—ZeroQuant [34], LLM.int8() [5], and nuQmm [24]—employ relatively simple quantization schemes based on rounding to the nearest (RTN) quantization level. This simple approach has the advantage of maintaining acceptable runtimes for very large models. ZeroQuant further proposes layer-wise knowledge distillation, similar to AdaQuant, but the largest model it can apply this approach to has only 1.3 billion parameters. At this scale, ZeroQuant already takes $\approx 3$ hours of compute: by our conservative estimate, ZeroQuant would require at least 500 hours to quantize OPT-175B. LLM.int8() observes that *activation outliers* in a few feature dimensions break the quantization of larger models, and proposes to fix this problem by keeping those dimensions in higher precision. Lastly, nuQmm develops efficient GPU kernels for a binary-coding based quantization scheme, which can, at low bit-width, bring significant speedups over full precision, when executing single layers in isolation.

Relative to this line of work, we show that a significantly more complex and accurate weight quantizer can be implemented efficiently at large model scale. Specifically, GPTQ more than doubles the amount of compression relative to these prior techniques, at negligible accuracy loss, as it is the first method to achieve accurate quantization to between $2.5$ to $4$ bits per weight, which brings significant practical gains.

# 3 Methods

## 3.1 Layer-Wise Quantization

At a high level, our method follows the structure of state-of-the-art post-training quantization methods [20, 32, 14, 8], by performing quantization layer-by-layer, solving a corresponding reconstruction problem for each layer. Concretely, let $\mathbf{W}$ be the weights corresponding to a linear layer $\ell$ and let $\mathbf{X}$ denote the layer input corresponding to a small set of $m$ data points running through the network. Then, the objective is to find a matrix of quantized weights $\widehat{\mathbf{W}}$ which minimizes the squared error, relative to the full precision layer output. Formally, this can be restated as

$$\text{argmin}_{\widehat{\mathbf{W}}} ||\mathbf{W}\mathbf{X} - \widehat{\mathbf{W}}\mathbf{X}||_2^2. \tag{1}$$

Further, similar to [20, 17, 8], we assume that the quantization grid for $\widehat{\mathbf{W}}$ is fixed before the process, and that individual weights can move freely as in [14, 8].

## 3.2 Optimal Brain Quantization (OBQ)

Our approach builds on the recently-proposed Optimal Brain Quanization (OBQ) method [8] for solving the layer-wise quantization problem defined above, to which we perform a series of major modifications, which allow it to scale to large language models, providing more than *three orders of magnitude* computational speedup in practice. To aid understanding, we first briefly summarize the original OBQ method.

The OBQ method starts from the observation that Equation (1) can be written as the sum of the squared errors over each row of $\mathbf{W}$. Then, OBQ handles each row $\mathbf{w}$ independently, quantizing one weight at a time while always updating all not-yet-quantized weights, in order to compensate for the error incurred by quantizing a single weight. The corresponding objective is a quadratic with Hessian $\mathbf{H}_F = 2\mathbf{X}_F\mathbf{X}_F^\top$, where $F$ denotes the set of remaining full-precision weights. Then, the greedy-optimal weight to quantize next, which we denote by $w_q$, and the corresponding optimal update of all weights in $F$, denoted by $\boldsymbol{\delta}_F$, are given by the following formulas, where $\text{quant}(w)$ rounds $w$ to the nearest value on the quantization grid:

$$w_q = \text{argmin}_{w_q} \frac{(\text{quant}(w_q) - w_q)^2}{[\mathbf{H}_F^{-1}]_{qq}}, \quad \boldsymbol{\delta}_F = -\frac{w_q - \text{quant}(w_q)}{[\mathbf{H}_F^{-1}]_{qq}} \cdot (\mathbf{H}_F^{-1})_{:,q}. \tag{2}$$

OBQ quantizes weights iteratively using these two equations, until all the weights of $\mathbf{w}$ are quantized. This is done efficiently, avoiding expensive full recomputations of $\mathbf{H}^{-1}$, by removing the $q$th row and column of $\mathbf{H}$, which is necessary after quantizing $w_q$, directly in the inverse via one step of Gaussian elimination. Namely, the updated inverse is given by the formula

$$\mathbf{H}_{-q}^{-1} = \left(\mathbf{H}^{-1} - \frac{1}{[\mathbf{H}^{-1}]_{qq}} \mathbf{H}_{:,q}^{-1}\mathbf{H}_{q,:}^{-1}\right)_{-q}. \tag{3}$$

This method comes with a vectorized implementation, handling multiple rows of $\mathbf{W}$ in parallel. Eventually, the algorithm can achieve reasonable runtimes on medium-sized models: for instance, it can fully quantize the ResNet50 model (25M parameters) in $\approx 1$ hour on a single GPU, which is roughly in line with other post-training methods achieving state-of-the-art accuracy [8]. However, the fact that OBQ's runtime for a $d_{\text{row}} \times d_{\text{col}}$ matrix $\mathbf{W}$ has *cubic* input dependency $O(d_{\text{row}} \cdot d_{\text{col}}^3)$ means that applying it to models with billions of parameters is impractical.

### 3.3 The GPTQ Algorithm

**Step 1: Initial Insight.** As explained in the previous section, OBQ quantizes weights in greedy order, i.e. it always picks the weight which currently incurs the least additional quantization error. Interestingly, we find that, while this natural strategy does indeed seem to perform very well, its improvement over quantizing the weights in arbitrary order is generally small, in particular on large, heavily-parametrized layers. We speculate that this is because the slightly lower number of quantized weights with large individual error is balanced out by those weights being quantized towards the end of the process, when only few other unquantized weights that can be adjusted for error compensation remain. As we will now discuss, the simple insight that *any fixed order may perform well*, especially on large models, has interesting practical ramifications.

The original OBQ method quantizes rows of $\mathbf{W}$ independently, in a specific order defined by the corresponding errors. By contrast, we will aim to quantize the weights of *all rows in the same order*, and will show that this typically yields results with a final squared error that is similar to the original solutions. As a consequence, the set of unquantized weight indices $F$ and therefore the corresponding Hessian inverse $\mathbf{H}_F^{-1}$ will be always the same for all rows. (See Figure 2 for an illustration. In more detail, the latter is due to the fact that $\mathbf{H}_F$ depends only on the layer inputs $\mathbf{X}_F$, which are the same for all rows, and not on any weights.) We can leverage this to perform the update of $\mathbf{H}_F^{-1}$ given by Equation (3) only $d_{\text{col}}$ times, once per column, rather than $d_{\text{row}} \cdot d_{\text{col}}$ times, once per weight. This reduces the overall runtime from $O(d_{\text{row}} \cdot d_{\text{col}}^3)$ to $O(\max\{d_{\text{row}} \cdot d_{\text{col}}^2, d_{\text{col}}^3\})$, i.e., by a factor of $\min\{d_{\text{row}}, d_{\text{col}}\}$. For larger models, this difference is of several orders of magnitude. However, before this algorithm can actually be applied to very large models in practice, two additional major obstacles remain.

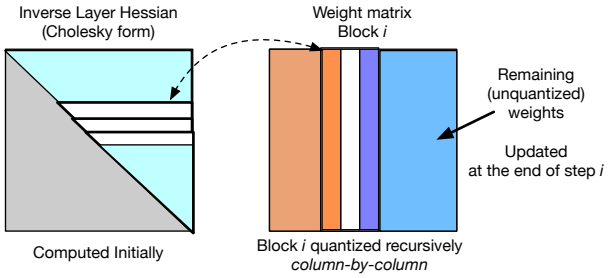


Figure 2: Illustration of the GPTQ quantization procedure. Blocks of consecutive *columns* (bolded) are quantized at a given step, using the inverse Hessian information stored in the Cholesky decomposition, and the remaining weights (blue) are updated at the end of the step. The quantization procedure is applied recursively inside each block: the white middle column is currently being quantized.

**Step 2: Achieving Fast Runtime.** First, a direct implementation of the scheme described previously will not be fast enough in practice because the algorithm has a relatively low compute-to-memory-access ratio. For example, Equation (3) needs to update all elements of a potentially huge matrix using just a few FLOPs for each entry. Here, the implementation will be bottlenecked by the significantly lower memory bandwidth of modern GPUs, relative to their computational power.

Fortunately, this problem can be resolved by the following observation: The final rounding decisions for column $i$ are only affected by updates performed on this very column, and so updates to later columns are irrelevant at this point in the process. This makes it possible to "batch" updates together, thus achieving much better computational utilization. Concretely, we apply the algorithm to $B = 128$ columns at a time, keeping updates contained to those columns and the corresponding $B \times B$ block of $\mathbf{H}^{-1}$. (Please see Figure 2 for an illustration.) Only once a block has been fully processed, we perform global updates of the entire $\mathbf{H}^{-1}$ and $\mathbf{W}$ matrices using the multi-weight versions of Equations (2) and (3) given below, with $Q$ denoting a set of indices, and $\mathbf{H}_{-Q}^{-1}$ denoting the inverse matrix with the corresponding rows and columns removed:

$$\boldsymbol{\delta}_F = -\Big(\mathbf{w}_Q - \text{quant}(\mathbf{w}_Q)\Big)([\mathbf{H}_F^{-1}]_{QQ})^{-1}(\mathbf{H}_F^{-1})_{:,Q}, \tag{4}$$

$$\mathbf{H}_{-Q}^{-1} = \left(\mathbf{H}^{-1} - \mathbf{H}_{:,Q}^{-1}([\mathbf{H}^{-1}]_{QQ})^{-1}\mathbf{H}_{Q,:}^{-1}\right)_{-Q}. \tag{5}$$

Although this strategy does not reduce the theoretical amount of compute, it effectively addresses the memory-throughput bottleneck. This provides an order of magnitude speedup for very large models in practice, making it a critical component of our algorithm.

**Step 3: Numerical Stability.** The final technical issue we have to address is given by numerical inaccuracies, which can become a major problem at the scale of existing models, especially when combined with the block updates discussed in the previous step. Specifically, it can occur that the matrix $\mathbf{H}_F^{-1}$ becomes indefinite, which we notice can cause the algorithm to aggressively update the remaining weights in incorrect directions, resulting in an arbitrarily-bad quantization of the corresponding layer. In practice, we observed that the probability of this happening increases with

model size: concretely, it almost certainly occurs for at least a few layers on models that are larger than a few billion parameters, and so it would prevent us from tackling the very largest models. The main issue is given by repeated applications of Equation (5), which accumulate various numerical errors, especially through the additional matrix inversion.

For smaller models, applying dampening, that is adding a small constant $\lambda$ (we always choose $1\%$ of the average diagonal value) to the diagonal elements of $\mathbf{H}$ appears to be sufficient to avoid numerical issues. However, larger models require a more robust and general approach.

To address this, we begin by noting that the only information required from $\mathbf{H}_{F_q}^{-1}$, where $F_q$ denotes the set of unquantized weights when quantizing weight $q$, is in $q$'s row, or, more precisely, the elements in this row starting with the diagonal. The consequence is that we could precompute all of these rows using a more numerically-stable method without any significant increase in memory consumption. Indeed, the row removal via (3) for our symmetric $\mathbf{H}^{-1}$ essentially corresponds to taking a Cholesky decomposition, except for the minor difference that the latter divides row $q$ by $\sqrt{[\mathbf{H}_{F_q}^{-1}]_{qq}}$. Hence, we can leverage efficient Cholesky kernels to compute all information we will need from $\mathbf{H}^{-1}$ upfront. In combination with mild diagonal dampening, the resulting method is robust enough to execute on huge models without issues. As a bonus, using a well-optimized Cholesky kernel also yields further speedup.

**The Full Algorithm.** Finally, we present the full pseudocode for our algorithm, including the optimizations discussed above. We provide an efficient Pytorch implementation at `https://github.com/IST-DASLab/gptq`.

**Algorithm 1** Quantize $\mathbf{W}$ given inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top + \lambda\mathbf{I})^{-1}$ and blocksize $B$.

$\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$ *// quantized output*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ *// block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})$ *// Hessian inverse information*
**for** $i = 0, B, 2B, \ldots$ **do**
  **for** $j = i, \ldots, i + B - 1$ **do**
    $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ *// quantize column*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) \,/\, [\mathbf{H}^{-1}]_{jj}$ *// quantization error*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ *// update weights in block*
  **end for**
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ *// update all remaining weights*
**end for**

## 4 Experimental Validation

**Overview.** We begin our experiments by validating the accuracy of GPTQ relative to other accurate-but-expensive quantizers, on smaller models, for which these methods provide reasonable runtimes. Next, we examine GPTQ's runtime scaling for very large models. Then, we present 3- and 4-bit quantization results for the entire BLOOM and OPT model families, evaluated via perplexity on challenging language generation tasks. In addition, we show that our method is also stable for 2-bit quantization when the granularity is reduced to small blocks of consecutive weights. To complement this perplexity analysis, we also evaluate the resulting quantized models on a series of standard zero-shot tasks. Finally, we focus on the two largest (and interesting) openly-available models, BLOOM-176B and OPT-175B, where we perform a detailed evaluation on several tasks. For these models, we also present practical improvements, namely reducing the number of GPUs required for inference as well as end-to-end speedups for generative tasks.

**Setup.** We implemented GPTQ in PyTorch [25] and worked with the HuggingFace integrations of the BLOOM [16] and OPT [35] model families. We quantized all models (including the 175 billion parameter variants) *using a single NVIDIA A100 GPU* with 80GB of memory. Our entire GPTQ calibration data consists of 128 random 2048 token segments from the C4 dataset [27], i.e. excerpts from randomly crawled websites, which represents generic text data. We emphasize that this means that GPTQ does not see any task-specific data, and our results thus remain truly "zero-shot". We perform standard uniform per-row asymmetric quantization on the min-max grid, similar to [5].

To ensure that the entire compression procedure can be performed with significantly less GPU memory than what would be required to run the full precision model, some care must be taken. Specifically, we always load one Transformer block, consisting of 6 layers, at a time into GPU memory and then accumulate the layer-Hessians and perform quantization. Finally, the current block inputs are sent through the fully quantized block again to produce the new inputs for the quantization of the next block. Hence, the quantization process operates not on the layer inputs in the

full precision model but on the actual layer inputs in the already partially quantized one. We find that this brings noticeable improvements at negligible extra cost.

**Baselines.** Our primary baseline, denoted by RTN, consists of rounding all weights to the nearest quantized value on the same grid that is also used for GPTQ. This is currently the method of choice in all works on quantization of very large language models [5, 34, 24]: its runtime scales well to networks with many billions of parameters since it simply performs direct weight rounding in a single pass. As we will also discuss in detail, more accurate methods, such as AdaRound [20] or BRECQ [17], are currently far too slow for models with many billions of parameters, the main focus of this work. Nevertheless, GPTQ is competitive with such methods for small models, while scaling to massive ones.

**Quantizing Small Models.** As a first ablation study, we compare GPTQ's performance relative to state-of-the-art post-training quantization (PTQ) methods, on ResNet18 and ResNet50, which are standard PTQ benchmarks, in the same setup as [8]. As can be seen in Table 1, GPTQ performs on par at 4-bit, and slightly worse than the most accurate methods at 3-bit. At the same time, it significantly outperforms AdaQuant, the fastest amongst prior PTQ methods. Further, we compare against the full greedy OBQ method on two smaller language models: BERT-base [6] and OPT-125M [35]. The results are shown in Appendix Table 7. At 4 bits, both methods perform similarly, and for 3 bits, GPTQ surprisingly performs slightly better. We suspect that this is because some of the additional heuristics used by OBQ, such as early outlier rounding, might require careful adjustments for optimal performance on non-vision models. Overall, GPTQ appears to be competitive with state-of-the-art post-training methods for smaller models, while taking only $< 1$ minute rather than $\approx 1$ hour. This enables scaling to much larger models.

| Method | RN18 – 69.76% | | RN50 – 76.13% | |
|---|---|---|---|---|
| | 4bit | 3bit | 4bit | 3bit |
| AdaRound | 69.34 | 68.37 | 75.84 | 75.14 |
| AdaQuant | 68.12 | 59.21 | 74.68 | 64.98 |
| BRECQ | 69.37 | 68.47 | 75.88 | 75.32 |
| OBQ | 69.56 | 68.69 | 75.72 | 75.24 |
| GPTQ | 69.37 | 67.88 | 75.71 | 74.87 |

Table 1: Comparison with state-of-the-art post-training methods for vision models.

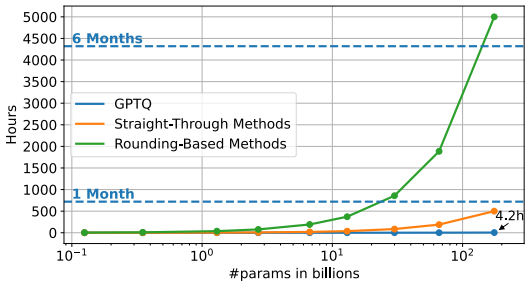


Figure 3: GPTQ runtime and estimated runtimes for other methods on OPT models.

**Runtime.** Figure 3 compares GPTQ's runtime for fully quantizing OPT models of varying size on a single NVIDIA A100 GPU, with runtime estimates for other post-training quantization approaches (precise runtime numbers for all models are provided in Appendix A.2). Concretely, we compare with methods based on the straight-through estimator (STE) [34, 14]. We estimate their runtime by linearly extrapolating the results of the most efficient implementation, ZeroQuant-LKD [34], taking $\approx 1$ hour per 350 million parameters. Further, we estimate the runtime of rounding-based methods [20, 17], which are significantly more accurate, but typically perform $10\times$ more SGD steps. Hence, we optimistically approximate their runtime as $10\times$ the estimate for STE methods. While GPTQ can quantize 175 billion parameter models in $\approx 4$ hours, this would take current STE methods about 3 weeks, and current adaptive rounding methods would need half a year. Our estimates are generous, as they do not consider factors such as memory constraints, and assume that the number of SGD steps is constant w.r.t. model size, although some methods, e.g., [34], increase them linearly with model size. Thus, GPTQ is the first highly-accurate post-training method that can scale to extremely large models.

**Language Generation.** We begin our large-scale study by compressing the entire OPT and BLOOM model families to 3 and 4 bits per weight. We then evaluate those models on several language modelling tasks including Wiki-Text2 [19] (see Figure 1 and Appendix A.3), Penn Treebank (PTB) [18] (see Tables 2 and 3) and C4 [27] (see Appendix A.3). We focus on these perplexity-based tasks, as they are known to be particularly sensitive to model quantization [34]. On OPT models, GPTQ clearly outperforms RTN, by significant margins. For example, GPTQ loses 0.14 perplexity at 4-bit on the 175B model, while RTN drops 4.53 points, performing worse than the nearly $50\times$ smaller full-precision 2.7B model. At 3-bit, RTN collapses completely, while GPTQ can still maintain reasonable perplexity, in particular for larger models. BLOOM shows a similar pattern: the gaps between methods are usually smaller, indicating that this model family might be easier to quantize. One interesting trend (see also Figure 1) is that

larger models generally (with the exception of OPT-66B[3]) appear easier to quantize. This is good news for practical applications, as these are the cases where compression is also the most necessary.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| Baseline | 16 | 32.55 | 26.08 | 16.96 | 15.11 | 13.09 | 12.34 | 11.84 | 11.36 | 10.33 |
| RTN | 4 | 45.01 | 31.12 | 34.14 | 22.10 | 16.09 | 15.40 | 14.17 | 274 | 15.00 |
| GPTQ | 4 | **36.96** | **28.85** | **18.16** | **15.96** | **13.80** | **12.58** | **11.98** | **11.58** | **10.47** |
| RTN | 3 | 1.2e3 | 81.07 | 1.1e4 | 9.4e3 | 3.4e3 | 2.5e3 | 1.4e3 | 3.6e3 | 4.8e3 |
| GPTQ | 3 | **65.18** | **39.48** | **26.08** | **20.30** | **18.45** | **13.89** | **12.73** | **15.79** | **10.92** |

Table 2: OPT model family perplexity results on Penn Treebank.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| Baseline | 16 | 41.24 | 46.98 | 27.93 | 23.12 | 19.40 | 13.63 |
| RTN | 4 | 48.57 | 54.54 | 31.21 | 25.40 | 20.93 | 14.07 |
| GPTQ | 4 | **44.42** | **50.98** | **29.70** | **24.27** | **20.16** | **13.76** |
| RTN | 3 | 117 | 152 | 115 | 59.96 | 32.05 | 190 |
| GPTQ | 3 | **65.48** | **70.68** | **42.23** | **30.70** | **24.16** | **14.60** |

Table 3: BLOOM model family perplexity results for Penn Treebank.

**175-Billion Parameter Models.** We now focus on BLOOM-176B and OPT-175B, the largest dense openly-available models. Table 4 summarizes results across Wikitext-2, PTB and C4. We observe that, at 4 bits, GPTQ models reach only $< 0.2$ lower perplexity than the full-precision versions, with a large gap to RTN results on OPT-175B. At 3-bit, RTN collapses, while GPTQ is still able to maintain good performance on most tasks, losing only $0.3 - 0.5$ points for more than $5\times$ compression. We note that GPTQ's accuracy can be further improved via finer-granularity grouping/bucketing [1, 24]: with group-size 1024, GPTQ's OPT-175B 3-bit WikiText2 PPL further improves from 8.68 to 8.45. We examine the impact of group size more carefully below.

| Method | Bits | OPT-175B | | | | BLOOM-176B | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Wiki2 | PTB | C4 | LAMB. ↑ | Wiki2 | PTB | C4 | LAMB. ↑ |
| Baseline | 16 | 8.34 | 10.33 | 9.56 | 75.59 | 8.11 | 13.63 | 10.98 | 67.40 |
| RTN | 4 | 10.54 | 15.00 | 10.92 | 71.34 | 8.37 | 14.07 | 11.26 | 66.70 |
| GPTQ | 4 | **8.37** | **10.47** | **9.67** | **76.80** | **8.21** | **13.76** | **11.06** | **67.71** |
| RTN | 3 | 7.3e4 | 4.8e4 | 4.4e4 | 0 | 571. | 190. | 309 | 0.17 |
| GPTQ | 3 | **8.68** | **10.92** | **9.98** | **76.19** | **8.64** | **14.60** | **11.45** | **65.10** |
| GPTQ | 3G | 8.45 | 10.69 | 8.84 | 77.39 | 8.35 | 14.01 | 11.22 | 67.47 |

Table 4: Results summary for OPT-175B and BLOOM-176B; 3G indicates 3-bit results with group-size 1024.

**Practical Speedups.** Finally, we focus on practical applications. As an interesting use-case, we focus on the OPT-175B model: quantized to 3 bits, this model takes approximately 63GB of memory, including the embeddings and the output layer, which are kept in full FP16 precision. Additionally, storing the complete history of keys and values for all layers, a common optimization for generation tasks, consumes another $\approx$ 9GB for the maximum of 2048 tokens. Hence, we can actually fit the entire quantized model into a single 80GB A100 GPU, which can be executed by dynamically dequantizing layers as they are required during inference. (The model would not fully fit using 4 bits.) For reference, standard FP16 execution requires $5\times$80GB GPUs, and the state-of-the-art 8-bit LLM.int8() quantizer [5] requires 3 such GPUs.

We focus on language generation, one of the most appealing applications of these models, with the goal of latency reduction. Unlike LLM.int8(), which reduces memory costs but has the same runtime as the FP16 baseline, we show that our quantized models can achieve significant speedups for this application. For language generation, the model processes and outputs one token at-a-time, which for OPT-175B can easily take a few 100s of milliseconds per token.

[3]Upon closer inspection of the OPT-66B model, it appears that this is correlated with the fact that this trained model has a significant fraction of dead units in the early layers, which may make it harder to compress.

Increasing the speed at which the user receives generated results is challenging, as compute is dominated by matrix-vector products. Unlike the more intensive matrix-matrix products, matrix-vector products are primarily limited by memory bandwidth. We address this problem by developing a quantized-matrix full-precision-vector product kernel which performs a matrix vector product by dynamically dequantizing weights when needed. Most notably, this does *not* require any activation quantization. While dequantization consumes extra compute, the kernel has to access a lot less memory, leading to significant speedups, as shown in Table 5. We note that almost all of the speedup is due to our kernels, as communication costs are negligible in the setup of Dettmers et al. [5], which we also consider here.

| GPU | FP16 | 3bit | Speedup | GPU reduction |
|---|---|---|---|---|
| A6000 – 48GB | 589ms | 132ms | 4.46× | $8 \rightarrow 2$ |
| A100 – 80GB | 230ms | 121ms | 1.90× | $5 \rightarrow 1$ |

Table 5: Average per-token latency (batch size 1) when generating sequences of length 128.

For example, using our kernels, the 3-bit OPT-175B model obtained via GPTQ running on a single A100 is over $\mathbf{1.9}\times$ faster than the FP16 version (running on 5 GPUs) in terms of average time per token. More accessible GPUs, such as the NVIDIA A6000, have much lower memory bandwidth, so this strategy is even more effective: executing the 3-bit OPT-175B model on $2\times$ A6000 GPUs reduces latency from 589 milliseconds for FP16 inference (on 8 GPUs) to 132 milliseconds, a $\mathbf{4.46}\times$ reduction. This is only $\approx 10\%$ slower than 3-bit inference on the more expensive A100 GPU.

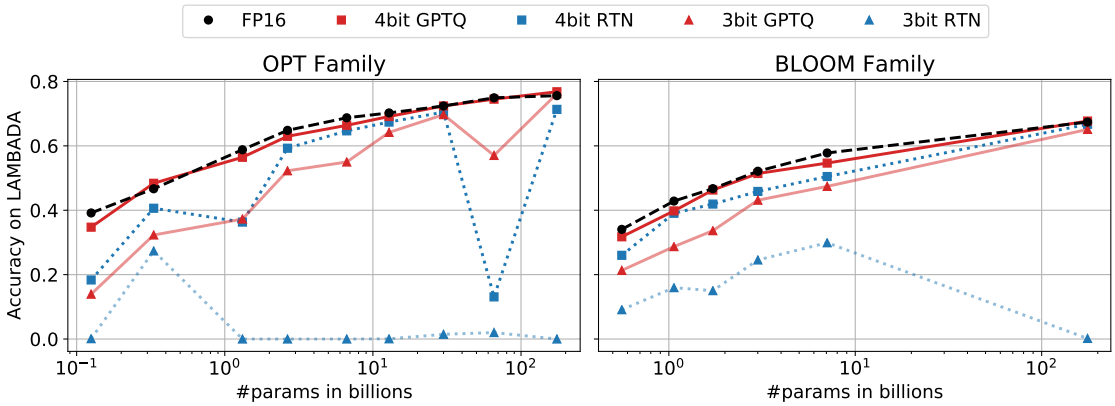


Figure 4: The accuracy of OPT and BLOOM models post-GPTQ, measured on LAMBADA.

**Zero-Shot Tasks.** While our focus is on language generation, we also evaluate the performance of quantized models on some popular zero-shot tasks, namely LAMBADA [23], ARC (Easy and Challenge) [3] and PIQA [30]. Figure 4 visualizes model performance on LAMBADA (and see also the LAMB. results in Table 4). We observe similar behavior as before: the outliers are that 1) quantization appears "easier" across the whole spectrum of models at 4-bit, where even RTN performs relatively well, and 2) at 3-bit, RTN breaks down, while GPTQ still provides good accuracy. We provide additional results in Appendix A.4.

**Additional Tricks & 2-bit Quantization.** While our experiments so far have focused exclusively on vanilla row-wise quantization, we now show that GPTQ is also compatible with more advanced quantization tricks, leading to further improvements. Specifically, we investigate standard *grouping* [1, 24], i.e. applying independent quantization to groups of $G$ consecutive weights. This interacts very well with GPTQ, as the group parameters can actually be determined during the quantization process of each layer, always using the most current updated weights. The last row of Table 4 (marked 3G) includes 3-bit results with group-size 1024 for BLOOM and OPT-175B. At the cost of only $< 0.05$ bits extra storage per weight, this reduces the accuracy drops by another $0.2 - 0.3$ points, bringing 3-bit even closer to FP16 performance.

| Model | FP16 | 1024 | 512 | 256 | 128 | 64 | 32 | 3-bit |
|---|---|---|---|---|---|---|---|---|
| OPT-175B | 8.34 | 11.84 | 10.85 | 10.00 | 9.58 | 9.18 | 8.94 | 8.68 |
| BLOOM | 8.11 | 11.80 | 10.84 | 10.13 | 9.55 | 9.17 | 8.83 | 8.64 |

Table 6: 2-bit GPTQ quantization results with varying group-sizes; perplexity on WikiText2.

While we find grouping at 4-bit and further reducing the group-size at 3-bit to bring only rather minor gains, grouping actually makes it possible to achieve reasonable performance for extreme 2-bit quantization. Table 6 shows results on WikiText2 when quantizing the biggest models to 2-bit with varying group-sizes. Even at group-size 1024, the perplexity drops only by about $3.5$ points, with the accuracy loss quickly decreasing for smaller group-sizes to only $\approx 1$ at group-size 64, corresponding to 2.5 bit. Interestingly, we find that the performance of $G = 32$ is only $0.2 - 0.3$ points worse than vanilla 3-bit quantization, at the same memory consumption, which might also be interesting for practical kernel implementations. In summary, we think that these results are a very encouraging first step towards pushing highly-accurate *one-shot* compression of very large language models even lower than 3-bit.

## 5 Discussion

**Summary and Limitations.** We have presented GPTQ, an approximate second-order method for quantizing truly large language models. GPTQ can accurately compress some of the largest publicly-available models down to $2.5 - 4$ bits per component on average, which leads to significant usability improvements, and to end-to-end speedups, at low accuracy loss. We hope that our method will make these models accessible to more researchers and practitioners. At the same time, we emphasize some significant limitations: On the technical side, our implementation obtains speedups from reduced memory movement, and does not lead to computational reductions. In addition, our study focuses on generative tasks, and does not consider activation quantization, nor speedups in batched execution. These are natural directions for future work, and we believe this can be achieved with carefully-designed GPU kernels and extensions of existing complementary techniques [34, 33].

**Ethical Concerns.** Our work introduces a general method for compressing large language models (LLMs) via quantization, with little-to-no accuracy loss in terms of standard accuracy metrics such as perplexity. Our method is task-agnostic, as it only uses a tiny amount of randomly-chosen input data for calibration. We do not foresee any significant ethical implications arising directly from the technical details of our method. However, one notable consideration is that our study focused on "leading accuracy" metrics, such as perplexity, which is essentially standard in the literature [5, 34]. We believe a thorough study of the impact of compression upon secondary measures, and in particular transferrability [15] or bias effects [2] is warranted, and may be rendered easier through our work. At the same time, our work makes inference on extremely large language models more accessible, for better or for worse. In time, these massive models will become much easier to use and deploy, making the need to understand their power and limitations even more stringent.

## Acknowledgments

Elias Frantar and Dan Alistarh gratefully acknowledge funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement No. 805223 ScaleML), as well as experimental support from Eldar Kurtic, and from the IST Austria IT department, in particular Stefano Elefante, Andrei Hornoiu, and Alois Schloegl. The work of Saleh Ashkboos and Torsten Hoefler was supported by the PASC DaCeMI project, received EuroHPC-JU funding under grant MAELSTROM, No. 955513. We thank the Swiss National Supercomputing Center (CSCS) for supporting us with compute infrastructure.

## References

[1] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD: Randomized quantization for communication-efficient stochastic gradient descent. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[2] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *2021 ACM Conference on Fairness, Accountability, and Transparency*, 2021.

[3] Michael Boratko, Harshit Padigela, Divyendra Mikkilineni, Pritish Yuvraj, Rajarshi Das, Andrew McCallum, Maria Chang, Achille Fokoue-Nkoutche, Pavan Kapanipathi, Nicholas Mattei, et al. A systematic classification of knowledge, reasoning, and context within the ARC dataset. *arXiv preprint arXiv:1806.00358*, 2018.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[5] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. *arXiv preprint arXiv:2208.07339*, 2022.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics (NAACL)*, 2019.

[7] Elias Frantar, Eldar Kurtic, and Dan Alistarh. M-FAC: Efficient matrix-free approximations of second-order information. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

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

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

[10] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE International Conference on Neural Networks*, 1993.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[12] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

[13] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Improving post training neural quantization: Layer-wise calibration and integer programming. *arXiv preprint arXiv:2006.10518*, 2020.

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

[15] Eugenia Iofinova, Alexandra Peste, Mark Kurtz, and Dan Alistarh. How well do sparse ImageNet models transfer? In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[16] Hugo Laurençon, Lucile Saulnier, Thomas Wang, Christopher Akiki, Albert Villanova del Moral, Teven Le Scao, Leandro Von Werra, Chenghao Mou, Eduardo González Ponferrada, Huu Nguyen, et al. The BigScience corpus: A 1.6 TB composite multilingual dataset. 2022.

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

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

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

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

[21] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.

[22] Yury Nahshan, Brian Chmiel, Chaim Baskin, Evgenii Zheltonozhskii, Ron Banner, Alex M Bronstein, and Avi Mendelson. Loss aware post-training quantization. *Machine Learning*, 110(11):3245–3262, 2021.

[23] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

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

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

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

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

[28] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2016.

[29] Sidak Pal Singh and Dan Alistarh. WoodFisher: Efficient second-order approximation for neural network compression. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

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

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

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

[33] Xiaoxia Wu, Zhewei Yao, Minjia Zhang, Conglong Li, and Yuxiong He. Extreme compression for pre-trained transformers made simple and efficient. *arXiv preprint arXiv:2206.01859*, 2022.

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

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

# A Appendix

## A.1 Additional Comparison with OBQ

We now provide an additional comparison between GPTQ and OBQ on BERT-base/SQuAD [28] and OPT-125M/WikiText2, which is one of the largest models to which OBQ can be reasonably applied.

| Method | BERT-base 88.53 F1 ↑ | | OPT-125M 27.66 PPL ↓ | |
|---|---|---|---|---|
| | 4bit | 3bit | 4bit | 3bit |
| OBQ | 88.23 | 85.29 | 32.52 | 69.32 |
| GPTQ | 88.18 | 86.02 | 31.12 | 53.85 |

Table 7: Comparison of GPTQ relative to OBQ on BERT-base/SQuAD and OPT-125M/WikiText2.

## A.2 GPTQ Runtimes

Tables 8 and 9 provide runtime numbers for GPTQ on all models.

| OPT | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|
| Time | 0.6m | 1.5m | 3.4m | 6.0m | 11.8m | 20.9m | 44.9m | 1.6h | 4.2h |

Table 8: GPTQ runtime for full quantization of OPT models on a single A100 GPU.

| BLOOM | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|
| Time | 1.4m | 2.1m | 2.9m | 5.2m | 10.0m | 3.8h |

Table 9: GPTQ runtime for full quantization of BLOOM models on a single A100 GPU.

### A.3 Additional Language Generation Results

This section (Tables 10–13) contains additional results for language generation tasks.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| full | 16 | 27.66 | 22.01 | 14.63 | 12.46 | 10.86 | 10.12 | 9.56 | 9.33 | 8.34 |
| RTN | 4 | 37.28 | 25.95 | 48.24 | 16.92 | 12.10 | 11.32 | 10.97 | 110 | 10.54 |
| GPTQ | 4 | **31.12** | **24.24** | **15.47** | **12.87** | **11.39** | **10.31** | **9.63** | **9.55** | **8.37** |
| RTN | 3 | 1.3e3 | 64.57 | 1.3e4 | 1.6e4 | 5.8e3 | 3.4e3 | 1.6e3 | 6.1e3 | 7.3e3 |
| GPTQ | 3 | **53.85** | **33.79** | **20.97** | **16.88** | **14.86** | **11.61** | **10.26** | **14.16** | **8.68** |

Table 10: OPT perplexity results on WikiText2.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| full | 16 | 22.43 | 17.69 | 15.39 | 13.49 | 11.37 | 8.11 |
| RTN | 4 | 25.90 | 22.00 | 16.97 | 14.76 | 12.10 | 8.37 |
| GPTQ | 4 | **24.03** | **19.05** | **16.48** | **14.20** | **11.73** | **8.21** |
| RTN | 3 | 57.08 | 50.19 | 63.53 | 39.32 | 17.38 | 571 |
| GPTQ | 3 | **32.31** | **25.08** | **21.11** | **17.40** | **13.47** | **8.64** |

Table 11: BLOOM perplexity results for WikiText2.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| full | 16 | 24.60 | 20.71 | 14.72 | 13.16 | 11.74 | 11.20 | 10.69 | 10.28 | 9.56 |
| RTN | 4 | 31.62 | 23.93 | 24.67 | 17.52 | 13.38 | 12.35 | 11.90 | 249 | 10.92 |
| GPTQ | 4 | **26.97** | **22.61** | **15.57** | **13.75** | **12.15** | **11.36** | **10.80** | **10.50** | **9.67** |
| RTN | 3 | 731 | 49.82 | 6.3e3 | 1.2e4 | 4.7e3 | 2.2e3 | 1.3e3 | 3.4e3 | 4.4e3 |
| GPTQ | 3 | **39.05** | **28.34** | **19.79** | **16.64** | **15.49** | **12.28** | **11.34** | **13.68** | **9.98** |

Table 12: OPT perplexity results on C4. We note that the calibration data used by GPTQ is sampled from the C4 training set, this task is thus not fully zero-shot.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| full | 16 | 24.38 | 20.28 | 17.97 | 16.14 | 14.13 | 10.98 |
| RTN | 4 | 27.43 | 22.42 | 19.56 | 17.33 | 14.88 | 11.26 |
| GPTQ | 4 | **25.60** | **21.35** | **18.85** | **16.66** | **14.46** | **11.06** |
| RTN | 3 | 59.02 | 55.57 | 96.14 | 76.32 | 20.67 | 409 |
| GPTQ | 3 | **32.23** | **26.31** | **22.92** | **19.42** | **16.22** | **11.45** |

Table 13: BLOOM perplexity results for C4. We note that the calibration data used by GPTQ is sampled from the C4 training set, this task is thus not fully zero-shot.

### A.4 Additional ZeroShot Results

This section contains additional results for zero-shot tasks.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| full | 16 | 39.16 | 46.67 | 58.80 | 64.82 | 68.72 | 70.23 | 72.39 | 74.93 | 75.59 |
| RTN | 4 | 18.34 | 40.62 | 36.31 | 59.27 | 64.66 | 67.38 | 70.48 | 13.08 | 71.34 |
| GPTQ | 4 | **34.74** | **48.38** | **56.45** | **62.97** | **66.37** | **69.12** | **72.40** | **74.50** | **76.80** |
| RTN | 3 | 0.10 | 27.36 | 0.00 | 0.00 | 0.00 | 0.06 | 1.46 | 2.00 | 0.00 |
| GPTQ | 3 | **13.93** | **32.31** | **37.26** | **52.26** | **54.98** | **64.18** | **69.69** | **57.02** | **76.19** |

Table 14: OPT accuracy on LAMBADA.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| full | 16 | 34.06 | 42.85 | 46.71 | 52.12 | 57.79 | 67.40 |
| RTN | 4 | 26.00 | 39.06 | 41.92 | 45.84 | 50.48 | 66.70 |
| GPTQ | 4 | **31.75** | **39.80** | **46.28** | **51.41** | **54.65** | **67.71** |
| RTN | 3 | 9.10 | 15.95 | 15.02 | 24.55 | 29.90 | 0.17 |
| GPTQ | 3 | **21.31** | **28.70** | **33.65** | **43.12** | **47.41** | **65.10** |

Table 15: BLOOM accuracy on LAMBADA.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| full | 16 | 62.02 | 64.74 | 72.36 | 74.81 | 76.39 | 76.88 | 78.18 | 79.76 | 81.07 |
| RTN | 4 | **61.43** | 63.44 | 67.63 | 73.72 | **76.44** | 76.01 | 77.26 | 60.07 | 78.23 |
| GPTQ | 4 | 61.26 | **63.71** | **70.73** | **73.99** | 76.28 | **76.61** | **79.00** | **79.33** | **81.00** |
| RTN | 3 | 56.09 | 60.61 | 52.77 | 51.90 | 50.49 | 52.99 | 56.37 | 50.87 | 51.25 |
| GPTQ | 3 | **59.25** | **61.32** | **68.34** | **71.38** | **73.29** | **75.24** | **77.58** | **71.27** | **80.03** |

Table 16: OPT accuracy on PIQA.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| full | 16 | 65.07 | 67.14 | 69.97 | 70.51 | 73.72 | 79.16 |
| RTN | 4 | 63.11 | 65.29 | 67.74 | **69.86** | 72.69 | **79.00** |
| GPTQ | 4 | **64.31** | **66.05** | **68.77** | 69.42 | **72.96** | **79.00** |
| RTN | 3 | 58.60 | 60.80 | 60.88 | 66.28 | 69.70 | 53.32 |
| GPTQ | 3 | **61.62** | **62.62** | **65.18** | **68.34** | **70.95** | **77.70** |

Table 17: BLOOM accuracy on PIQA.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| full | 16 | 39.69 | 40.36 | 50.93 | 54.34 | 60.14 | 61.83 | 65.40 | 67.26 | 71.04 |
| RTN | 4 | 36.32 | **38.55** | 49.20 | 52.90 | 57.68 | 61.31 | 61.11 | 40.66 | 63.93 |
| GPTQ | 4 | **39.02** | 37.92 | **59.97** | **53.11** | **59.72** | **61.32** | **65.11** | **65.35** | **68.69** |
| RTN | 3 | 30.43 | 36.07 | 27.97 | 26.05 | 25.04 | 30.60 | 34.22 | 25.84 | 26.77 |
| GPTQ | 3 | **36.15** | **36.91** | **46.17** | **48.19** | **53.41** | **56.82** | **59.72** | **52.44** | **65.36** |

Table 18: OPT accuracy on ARC-easy.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| full | 16 | 41.71 | 45.41 | 48.11 | 53.24 | 57.37 | 67.47 |
| RTN | 4 | 39.40 | 42.51 | **44.70** | 51.35 | **56.14** | 66.33 |
| GPTQ | 4 | **40.24** | **44.49** | 44.49 | **52.82** | **56.14** | **67.42** |
| RTN | 3 | **45.44** | **46.87** | 37.58 | 45.08 | 48.61 | 28.87 |
| GPTQ | 3 | 39.14 | 41.79 | **42.85** | **46.63** | **51.56** | **62.84** |

Table 19: BLOOM accuracy on ARC-easy.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| full | 16 | 22.87 | 24.06 | 29.44 | 31.31 | 34.56 | 35.75 | 38.14 | 40.02 | 43.94 |
| RTN | 4 | 22.44 | 23.81 | 24.91 | 29.18 | 32.59 | **35.24** | 35.41 | 22.87 | 37.71 |
| GPTQ | 4 | **22.95** | **24.83** | **28.24** | **30.12** | **33.70** | 34.90 | **37.80** | **39.16** | **42.75** |
| RTN | 3 | 21.76 | 22.18 | 23.55 | 25.43 | 25.85 | 23.81 | 19.97 | 25.77 | 23.81 |
| GPTQ | 3 | **22.53** | **25.09** | **27.65** | **27.82** | **31.91** | **33.02** | **35.84** | **31.66** | **41.04** |

Table 20: OPT accuracy on ARC-challenge.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| full | 16 | 24.15 | 25.68 | 26.79 | 30.55 | 33.45 | 44.97 |
| RTN | 4 | **23.89** | 23.34 | **26.45** | **29.52** | 32.17 | 43.17 |
| GPTQ | 4 | 23.46 | **25.51** | 25.94 | 28.92 | **32.25** | **44.20** |
| RTN | 3 | 21.67 | 22.86 | 23.29 | 27.13 | **31.31** | 24.74 |
| GPTQ | 3 | **23.21** | **24.06** | **24.91** | **28.58** | 30.97 | **40.70** |

Table 21: BLOOM accuracy on ARC-challenge.

| OPT | Bits | 125M | 350M | 1.3B | 2.7B | 6.7B | 13B | 30B | 66B | 175B |
|---|---|---|---|---|---|---|---|---|---|---|
| full | 16 | 59.96 | 63.21 | 70.78 | 71.74 | 74.60 | 76.64 | 77.28 | 77.34 | 79.82 |
| RTN | 4 | **60.02** | 63.08 | 59.13 | **70.78** | 73.65 | 74.47 | 75.37 | 51.24 | 78.04 |
| GPTQ | 4 | 59.58 | **63.46** | **69.64** | 70.46 | **73.90** | **76.19** | **77.08** | **77.15** | **80.08** |
| RTN | 3 | 49.65 | 56.78 | 47.61 | 46.98 | 48.12 | 49.20 | 49.84 | 48.19 | 46.47 |
| GPTQ | 3 | **57.03** | **60.15** | **65.25** | **68.43** | **70.97** | **73.07** | **75.68** | **71.23** | **78.04** |

Table 22: OPT accuracy on StoryCloze.

| BLOOM | Bits | 560M | 1.1B | 1.7B | 3B | 7.1B | 176B |
|---|---|---|---|---|---|---|---|
| full | 16 | 61.94 | 63.27 | 65.44 | 67.79 | 71.99 | 76.89 |
| RTN | 4 | 60.15 | 60.66 | 62.95 | 67.09 | 70.72 | 76.00 |
| GPTQ | 4 | **61.17** | **62.32** | **64.48** | **67.22** | **71.36** | **76.32** |
| RTN | 3 | 54.87 | 56.08 | 55.79 | 59.83 | 66.20 | 48.50 |
| GPTQ | 3 | **57.80** | **59.77** | **61.81** | **63.97** | **69.26** | **75.37** |

Table 23: BLOOM accuracy on StoryCloze.