Ministry of Education and Science of Ukraine
National University “Kyiv-Mohyla Academy”
Faculty of Informatics – Informatics Department

Master's thesis

educational level – master

on the topic: “**Language model optimization using pruning, distillation and quantization techniques for NLP tasks.**”

By: 2-nd year student

of the educational program “Computer Sciences”, 122

Petrenko Mykhailo

Supervisor: Marchenko Oleksandr

Doctor of Physics and Mathematics

Reviewer Sajok M. M.

The master's thesis was defended

with a grade ______________________

EC secretary ____________________

«____» _________________ 2023 p.

Kyiv – 2024

CONTENTS

INTRODUCTION 3
BACKGROUND 7
2.1. NEURAL NETWORK COMPRESSION 7
2.2. QUANTIZATION 8
2.3. QUANTIZATION APPLICATION 9
2.4. TYPES OF QUANTIZATION 12
2.5. EXISTING QUANTIZATION FRAMEWORKS 14
2.6. AIMET – QUALCOM QUANTIZATION WHITE PAPER 19
2.7. GPTQ 23
2.7.1. OBS 23
2.7.2. Arbitrary Order – quantization order doesn't matter 26
2.7.3. Lazy Batch-Updates 27
2.7.4. Cholesky Reformulation – minimize Hessian inversion error 28
GPTAQ 32
3.1. ACTIVATIONS QUANTIZATION 32
3.1.1. Round to Nearest 35
3.1.2. Hessian based 37
3.1.3. OBS / LAPQ 40
3.1.4. Weights re-optimization 43
3.1.5. Token-wise 45
3.2. CROSS-LAYER EQUALIZATION 47
3.2.1. QLLM 49
3.2.2. OBQ & BatchNorm 53
3.2.3. Bias Correction 54
3.2.4. GPTQ CUDA kernel 55
3.3. HESSIAN EIGENVALUES IN QUANTIZATION 57
3.3.1. GPTVQ 57
3.3.2. Eigenvalue calculation 58
3.4. COMBINING CONTRIBUTIONS 60
3.4.1. GPTAQ Algorithm 60
3.4.2. Quantization debug flowchart 61
EXPERIMENTS 62
4.1. OVERVIEW 62
4.2. SETUP 62
4.3. BASELINES 62
4.4. EXPERIMENTS 63
4.5. ABLATION STUDY 64
4.6. RESULTS 65
SUMMARY 69
REFERENCES 71

Topic: Language model optimization using pruning, distillation and quantization techniques for NLP tasks.

Calendar plan of work:

Number	Stage	Completion date	Note
1.	Thesis topic received	31.10.2023.	
2.	Topic of master's thesis acknowledgement.	31.11.2023	
3.	Development of a plan and structure work	15.12.2023	
4.	Work with scientific literature and getting the main theoretical results.	15.03.2024	
5.	Trying and benchmarking solutions	01.04.2024	
6.	Work on the results formatting in text.	15.04.2024	
7.	Preliminary analysis of the master's thesis. Error correction.	01.06.2024	
8.	Defense of the master's degree work	12.06.2024	

INTRODUCTION

With recent advances in Generative Pretrained Transformers (GPTs), the demand for more GPU processing power is continually escalating. The growing complexity of these models and their widespread application in various domains have made efficient model optimization a critical need. As the accuracy of models that build on base GPTs now sees either small incremental improvements or attributes better performance to the model's group architecture (e.g., chatGPT-3.5 to chatGPT-4), the focus has shifted towards optimizing computational efficiency. Various optimization methods such as quantization, which involves representing weights and activations with low-precision data types (e.g., int8) instead of the usual float32, and parameter reduction, which removes parameters from existing neural networks to increase efficiency while maintaining accuracy, have gained significant attention.

Quantization, in particular, has emerged as a promising technique to reduce the computational and memory footprint of neural networks. By converting the high-precision weights and activations to lower precision, models can be run more efficiently on hardware with limited resources. This is especially relevant for deploying large models on edge devices and mobile platforms where computational power and memory are constrained. However, achieving this reduction without significant loss of accuracy remains a challenge, and thus, the development of sophisticated quantization techniques is ongoing.

In this paper, we present GPTAQ, a comprehensive framework that integrates three key enhancements into the quantization process: activation quantization, cross-layer equalization with bias correction, and the incorporation of Hessian eigenvalues. These methods collectively aim to enhance the efficiency of quantization while maintaining or even improving the accuracy of the quantized models. However, we encountered several challenges and limitations during the implementation of these techniques, which we will discuss in detail.

We begin by providing a background on neural network compression and quantization techniques, setting the stage for the discussion on the specifics of our proposed methods. This includes an overview of different types of quantization, such as uniform and non-uniform quantization, and their respective advantages and challenges. We also explore the practical applications of quantization in various domains and the impact it has on model performance and deployment.

Following this, we review existing quantization frameworks, including AIMET, a Qualcomm Quantization White Paper, and GPTQ. AIMET (AI Model Efficiency Toolkit) offers a comprehensive suite of tools for quantization and pruning and has been widely adopted for optimizing neural networks. We delve into the specific techniques employed by GPTQ, such as Arbitrary Order quantization, which allows the quantization order to be flexible, Lazy Batch-Updates that minimize computational overhead, and Cholesky Reformulation for reducing Hessian inversion error. These techniques highlight the state-of-the-art advancements in quantization and set the context for our contributions.

Next, we introduce GPTAQ and its components in detail. Activation quantization focuses on reducing the precision of activations, which can significantly lower the computational requirements. Cross-layer equalization with bias correction is aimed at balancing the scaling factors across layers and correcting biases introduced during quantization. The integration of Hessian eigenvalues leverages the curvature information of the loss surface to guide the quantization process, ensuring that the most critical parameters are quantized with higher precision. We describe the algorithms and methodologies used to implement these techniques and how they synergistically improve the overall quantization process. Despite these efforts, we encountered several obstacles that affected the overall performance and efficiency, which we discuss in the results section.

In the experimental section, we outline our setup, including the models and datasets used for evaluation. We utilize the OPT-175M model and evaluate our methods on the C4 and PTB datasets. The experiments were conducted on a Google T4 GPU, demonstrating the feasibility of our approach on both edge and high-performance computing platforms. We provide a detailed comparison with baseline

models, specifically GPTQ in FP16 and W4 precision, and measure the impact of each quantization method on model performance.

The results section presents a comprehensive analysis of our findings. While activation quantization, cross-layer equalization with bias correction, and Hessian eigenvalues integration each contributed to improving quantization efficiency, we faced challenges that limited their effectiveness. Our ablation study isolates the effects of each technique, providing insights into their individual and combined benefits. The perplexity scores for various configurations illustrate the difficulties in maintaining model performance despite aggressive quantization.

Finally, we summarize our contributions and highlight the practical implications of our work. While GPTAQ offers a potential solution for deploying large-scale language models on resource-limited hardware, the challenges encountered indicate that further refinement and research are needed. Our proposed enhancements show promise but also underscore the complexities involved in optimizing neural networks. Future work will focus on addressing these challenges and improving of our techniques in diverse real-world scenarios.

Future Work – While GPTAQ has demonstrated potential in improving quantization efficiency and model performance, several avenues for future research remain. One potential direction is the exploration of adaptive quantization techniques that dynamically adjust quantization parameters based on the input data characteristics during inference. This could further enhance the model's robustness and performance across a wider range of applications. Additionally, integrating GPTAQ with other model compression techniques, such as pruning and knowledge distillation, could yield even greater efficiency gains. Investigating the impact of these combined approaches on various neural network architectures, including those beyond transformers, would provide valuable insights into their generalizability and effectiveness. Finally, expanding the evaluation of GPTAQ to include real-world deployment scenarios on edge devices will be crucial for understanding its practical implications and ensuring its readiness for diverse operational environments.

BACKGROUND

2.1. NEURAL NETWORK COMPRESSION

Main directions of neural network compression are *Pruning* and *Quantization*.

Pruning (unstructured) is a technique for network compression, where least valuable weights are removed. In case of structured pruning – groups of weight connections are removed together (channels/filters), to optimize propagation.

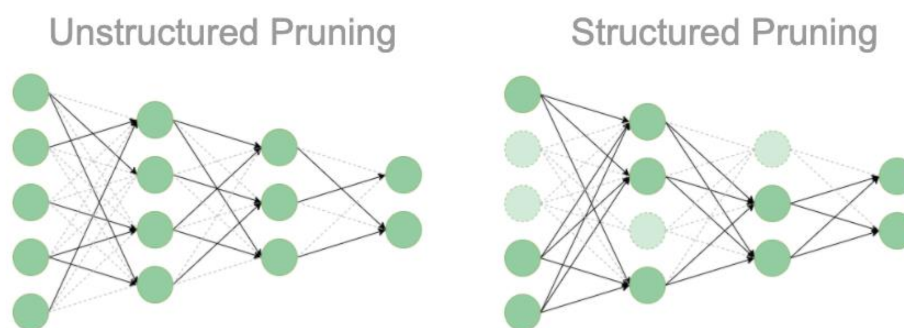

Figure 0.1 Structured vs Unstructured pruning.
https://opendatascience.com/what-is-pruning-in-machine-learning/

Quantization is a technique for network compression, where floating point representation of weights are compressed/rounded into integer format.

0.39	0.76	0.10		19	38	5
0.71	0.50	0.72	➡️	35	25	36
0.03	0.62	0.71		2	31	35

Figure 0.2 quantization

I focus my research on quantization because in their work *Pruning vs Quantization: Which is Better?* (Kuzmin et al., 2023) [1] state that quantization is better choice for compression than pruning: "*Quantization generally outperforms pruning for neural networks. Taking into account the unfavorable hardware implications for pruning described, it could be argued that the conclusion holds even stronger.*"

2.2. QUANTIZATION

Quantization term was borrowed, from mathematics and digital signal processing, where it is the process of mapping input values from a large set to output values in a smaller set e.g. rounding and truncation.

The most basic process of quantizing a weight matrix W of a neural network layer is as follows:

1. Set upper bound of quantization hyperparameter – b (8, 4, 2) for a maxq – $2^b - 1$
2. Calculate a scale factor s – $\frac{q_{max} - q_{min}}{2^b - 1}$
3. Quantize each floating-point weight using rounding – $clip(round(\frac{x}{s}), 0,\ 2^b - 1)$

There are 2 types of quantization based on when the model is being quantized, during or after training: Post training quantization and Quantization aware training.

Quantization aware training (QAT) performed during the model training. Model layers are being quantized during propagation and a loss is calculated on an already quantized layers. This approach may lead to better accuracy and perplexity of the model; however, it requires model maintainers and developers to include quantization in the process of training itself, thus this approach is less agile and gives less choice of models.

Post training quantization (PTQ) performed on an already trained model. Each layer of the trained model is being quantized and saved as a new model. Pros of this method is that it’s model-agnostic – every model can be subjected to a PTQ and doesn’t require training data to be accessible during quantization, which may affect accuracy and perplexity of the model. In some cases, calibration data may be used, which can be a separate dataset from the one used in training.

Post training quantization is preferred over the Quantization aware training due to wide range of target models to quantize, more accessible experimental setup and difference in accuracy of PTQ vs QAT becoming narrower.

2.3. QUANTIZATION APPLICATION

In order to understand which layers may be quantized – we need to review what are the most used types of layers, their shapes, layer inputs / ouputs and if they have learnable parameters.

Vast majority of neural network architectures consist of the following layer families:

- Convolutional: Conv2D, Conv1D, Conv3D
  quantizable – has learnable parameters, quantized after being reshaped
- Pooling: MaxPooling, AveragePooling, GlobalPooling
  non-quantizable – no learnable parameters, simple selection by criteria

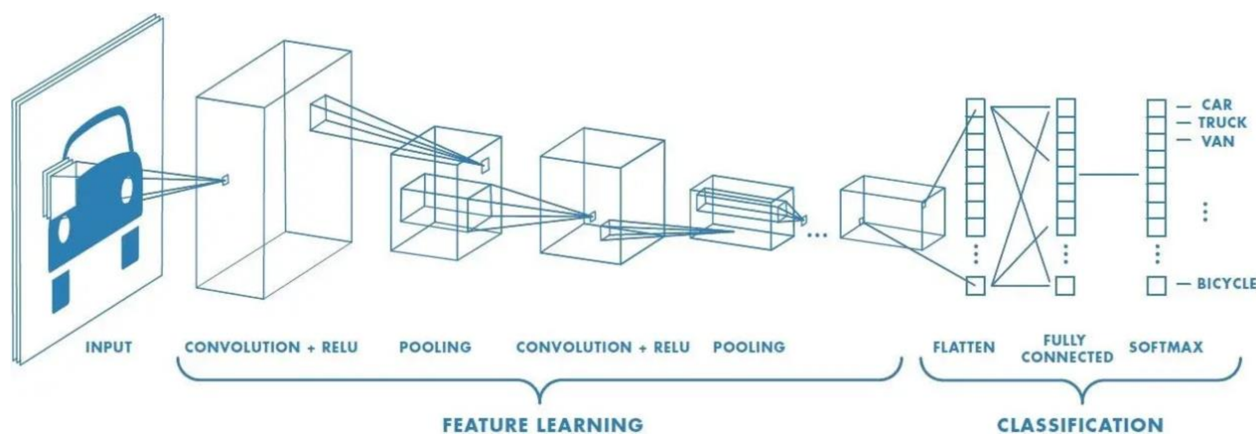

Figure 0.3 convolutional & pooling layers

saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way

- Fully Connected / Linear / Dense
  quantizable – has learnable params & input shape = output shape

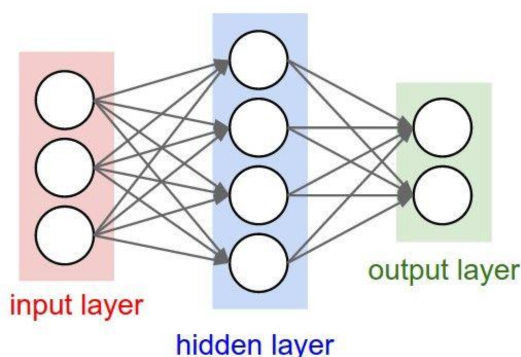

Figure 2.0.4 linear NN layer

iprathore71.medium.com/diving-deeper-into-quantization-realm-9c73e3172a3c

- Normalization: BatchNorm, LayerNorm, InstanceNorm, GroupNorm
  non-quantizable – loss of precision & not reasonable as the only learnable parameters (gamma & beta) can be statistically tuned

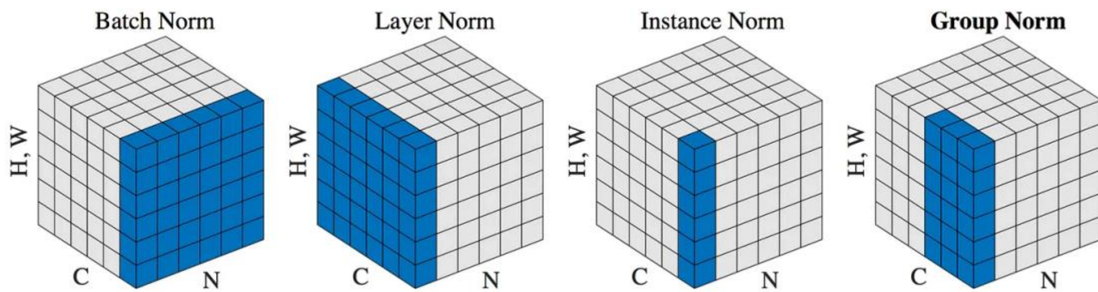

Figure 0.5 normalizations – arXiv:1803.08494

- Activation: ReLU, LeakyReLU, Sigmoid, Tanh, Softmax
  quantizable – no learnable params, but can turn continuous into descrete

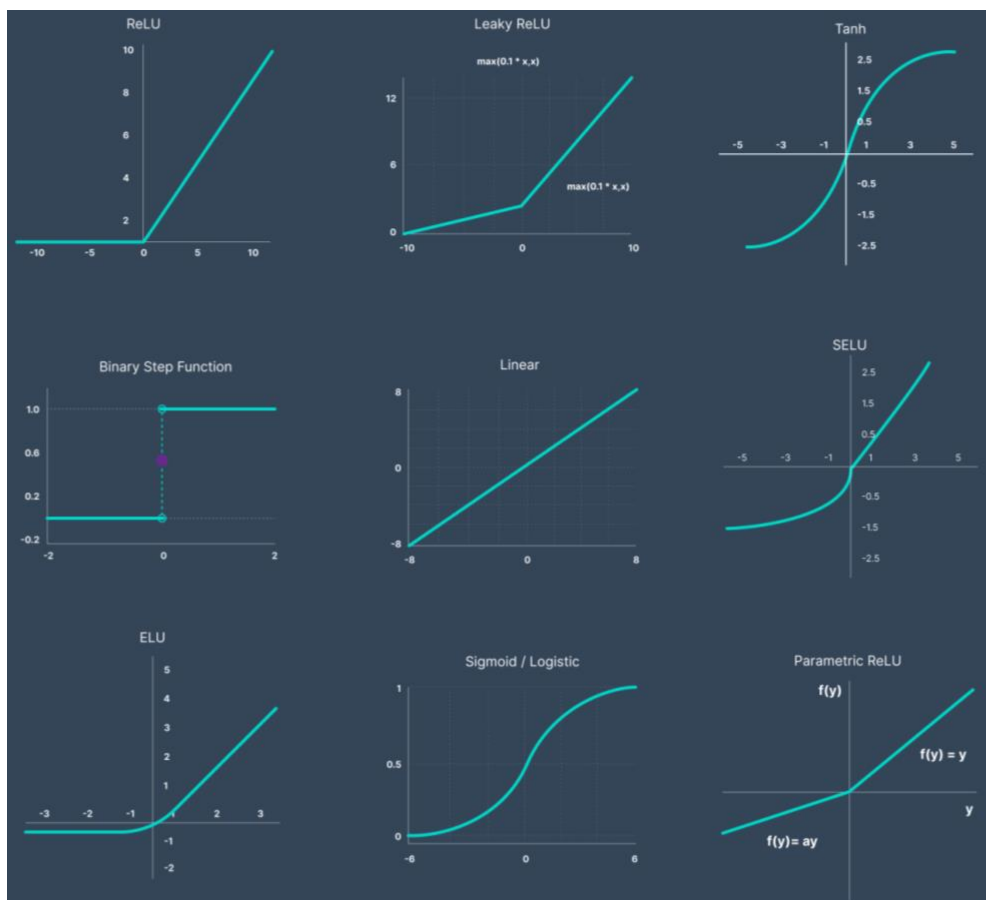

Figure 2.0.6 activation functions
v7labs.com/blog/neural-networks-activation-functions

Now we can see why Convolutional and Linear layers are the ones, whos meights are targeted by quantization. In case of activation quantization we perform discritization of activation function output Figure 2.0.7. Quantization can be applied to weights and to outputs of the layer (except outputs from the last layer, which is usually SoftMax).

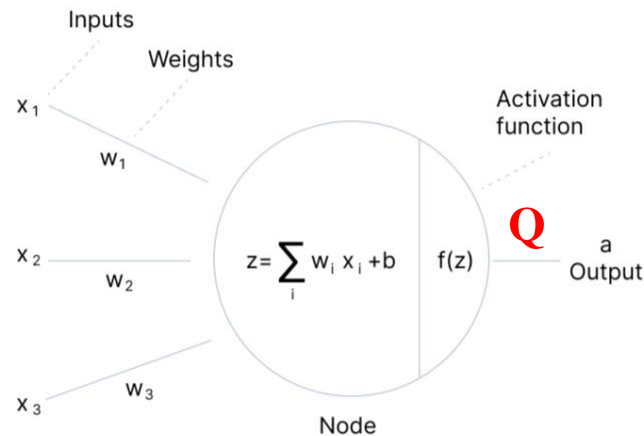

Figure 2.0.7 NN layer activation

v7labs.com/blog/neural-networks-activation-functions

After quantization / discretization activations in the simplest quantization setup output is rounded to the nearest integer. In case of ReLU activation function it can be visualized as follows: from regular ReLU into quantized on **Error! Reference source not found.**.

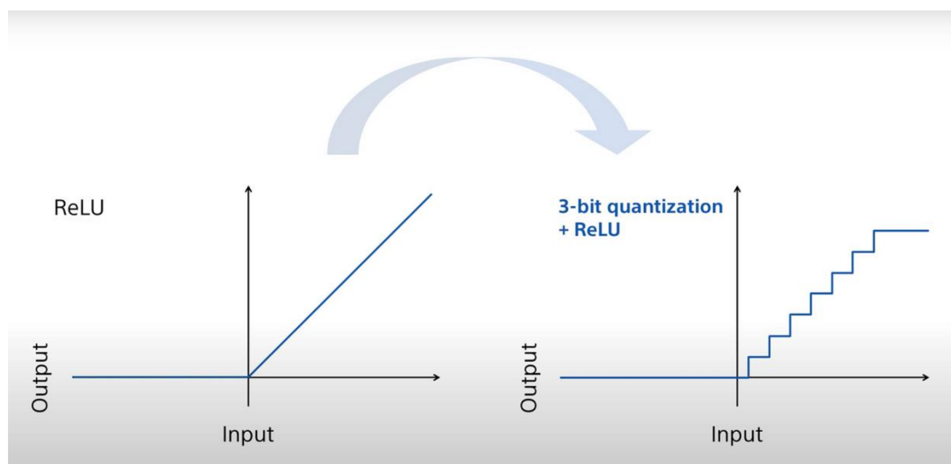

Figure 0.8 ReLU vs quantized ReLU (round to nearest)

pub.aimind.so/quantization-guide-for-complete-beginners-ac4555cf295f

In this work we introduce activation quantization, which has the same property of disctitization, albeit little more intricate than round to nearest.

2.4. TYPES OF QUANTIZATION

There are different types of Quantization based on symmetry, output uniformity and place.

Symmetry around data X distributions' zero point: symmetric, asymmetric asymmetric can be used, if during calibration – data is present, from which distribution and zero point can be calculated.

- symmetric – $clip(round(\frac{x}{s}), 0,\ 2^b - 1)$
- asymmetric – $clip(round(\frac{x}{s})\ +\ z, 0,\ 2^b - 1)$

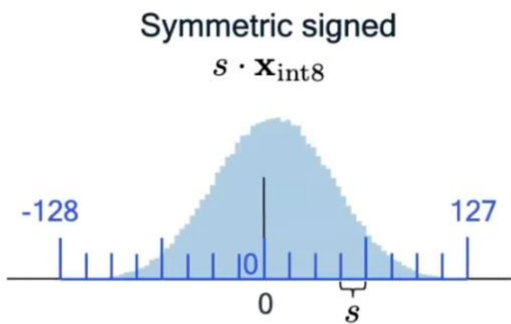

Figure 0.9 symmetric.
youtu.be/KASuxB3XoYQ

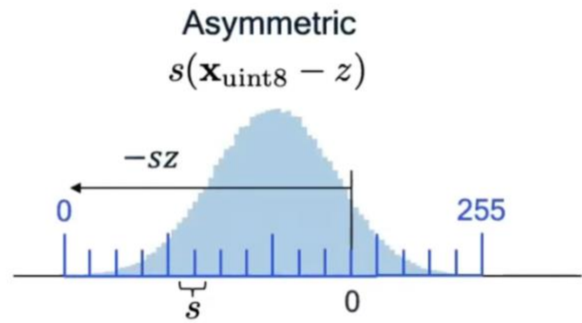

Figure 0.11 asymmetric
youtu.be/KASuxB3XoYQ

Uniformity of quantized intervals: uniform, non-uniform. Data is usually distributed in a non-uniform manner – some intervals are more sensitive.

- Uniform – 1.2 4.3 5.2 6.7 10.1 → 2 4 6 8 10
- Non-uniform – 1.2 4.3 5.2 6.7 10.1 → 1 4 5 6 10

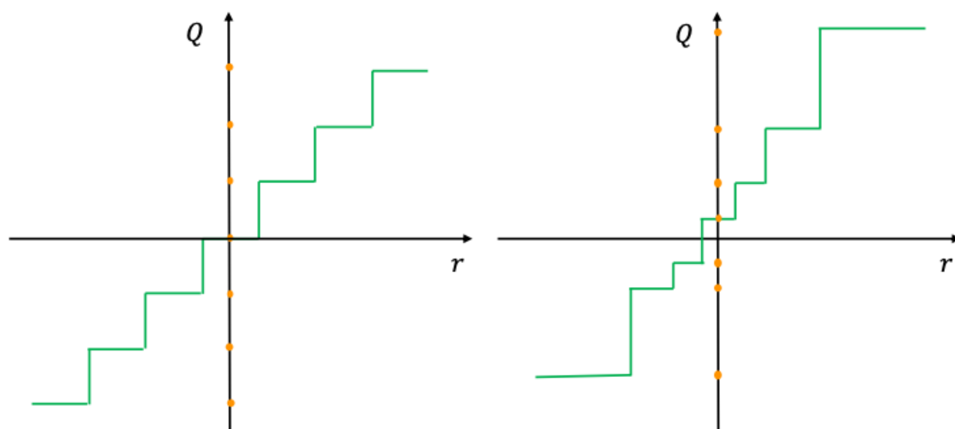

Figure 0.10 quantization: uniform (left) non-uniform (right)
medium.com/@florian_algo/model-quantization-2-uniform-and-non-uniform-quantization-47ca5b5d3ec0

Place of the part being quantized in neural network architecture: weight, activation, or both. Main difference in parts of neural network quantization is shape: weights are Matrixes of numbers W, while activations, or layer outputs – are 1D tensors.

- Weight – layer itself. 2D Tensor of shape INxOUT features

0.39	0.76	0.10		19	38	5
0.71	0.50	0.72	➡	35	25	36
0.03	0.62	0.71		2	31	35

Figure 0.11 quantization

- Activation – output of the layer. 1D Tensor of length OUT feature.

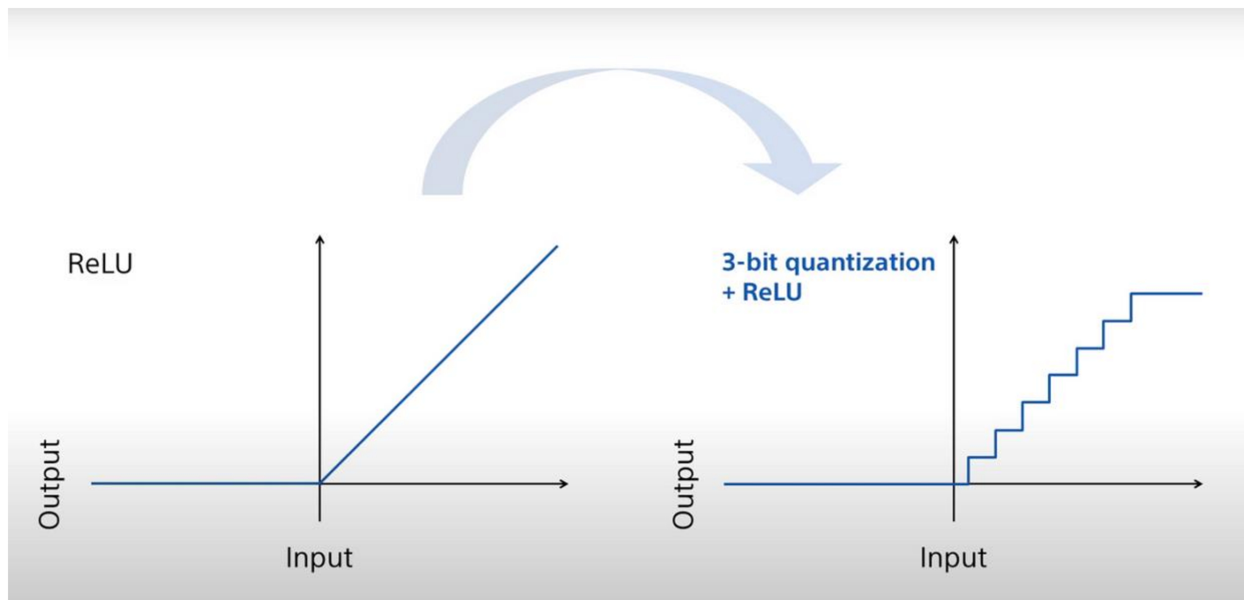

Figure 0.12 ReLU vs quantized ReLU (round to nearest)

pub.aimind.so/quantization-guide-for-complete-beginners-ac4555cf295f

2.5. EXISTING QUANTIZATION FRAMEWORKS

There are some quantization frameworks present already, such as *ZeroQuant*, *ZQ-Global*, *AIMET*, and *GPTQ* as well as *RTN (round-to-nearest neighborhood)* – not a framework, but a baseline straightforward approach of quantizing model by rounding FP16 → INT8/INT4 precision.

ZeroQuant (DeepSpeed) – is a quantization framework by Microsoft: (Yao et al., 2022) [7]. Which treats each layer as an individual small neural network (subnetwork) and utilizes a knowledge distillation targeted on minimizing $min_{\hat{\theta}} \left| f_{\theta}(x) - f_{\hat{\theta}}(x) \right|_2^2$ where θ is weights, $\hat{\theta}$ is quantized weights, f_{θ} is a subnetwork with weights and x is an input. Also using techniques such as mixed precision to balance performance and accuracy for a variety of transformer models. It is also suitable for both Quantization-Aware Training and Post-Training Quantization.

Main features of ZeroQuant framework include:

- Layer-wise Quantization: Each layer of the transformer model is quantized separately to minimize the quantization error.
- Mixed Precision: Uses a combination of different precision levels within the same model to balance performance and accuracy. Specifically, they use INT4/INT8 mixed precision quantization.
- Group-wise quantization for weight and token-wise for activations.
- Knowledge distillation: a teacher-student neural network knowledge distillation, where authors use the FP16 subnetwork as a teacher model to distill the quantized version the neural network is quantized layer-by-layer through distillation without the access to the original training data for INT4/INT8 mixed-precision quantization.

Drawbacks of ZeroQuant framework are the following:

- The performance gains from ZeroQuant are often hardware-dependent. It limits further framework improvements by open-source community, as well as doesn’t address key logical / theoretical quantization bottlenecks.

AIMET (AI Model Efficiency Toolkit) – is an opensource quantization framework by Qualcom: (Nagel et al., 2021) [2]. Which using techniques such as mixed precision and quantization-aware training to balance performance and accuracy for a variety of transformer models. It is also suitable for both Quantization-Aware Training and Post-Training Quantization. It introduces a vast pool of tooling for quantization, and some are used in this work, thus we will go through AIMET in more details in the next section.

Main features of AIMET framework include:

- Layer-wise Quantization: Each layer of the transformer model is quantized separately to minimize the quantization error.
- Compression Techniques: both – pruning (sparse) and quantization
- Cross-Layer Equalization (CLE): technique for mitigating quantization range difference of same channels in different layers. As a result, it amplifies bias error, which is mitigated via bias correction.
- Bias Correction: technique for correcting increased bias error after cross layer equalization.
- Batch Normalization Folding: BatchNorm layers into preceding convolutional layers to simplify the model and reduce inference time.
- AdaRound: systematic approach to finding weight rounding choices for PTQ.

Drawbacks of ZeroQuant framework are the following:

- Library dependence: AIMET framework utilizes aimet_torch python library, which conceals implementation details into the library, as well as limited customization possibility.
- Model size limitations: Models on which they conducted experiments in their research are limited to the following families: ResNet18, InceptionV3, MobileNetV2, EfficientNet lite, DeeplabV3, BERT-base. These are models with median size of approximately 17.8 million parameters and the maximum size is 110 million parameters, which is far below the size of modern LLMs, which have e.g., 175 billion parameters.

GPTQ – is an opensource quantization framework by (Frantar et al., 2023) [3]. Which utilizes minimization of quantization errors through Hessian-based methods and precise layer-wise quantization. It focuses exclusively on Post-training Quantization. This framework and its derivatives such as GPTVQ by Qualcom (Baalen et al., 2024) [5] are as of writing this paper are considered state of the art, thus its drawbacks and limitations are not from comparison with current alternatives, but rather are future work directions.

Main features of GPTQ framework include:

- Layer-wise Quantization: Each layer of the transformer model is quantized separately to minimize the quantization error.
- Hessian methods for more sensitive quantization.
- Cholesky matrix decomposition method: for minimizing accumulated error introduced by Hessian method.
- Grouping techniques: grouping in static/dynamic groups and blocks to optimize updates amount, while retaining accuracy and precision of the model, as well as GPU packing using custom CUDA kernels.

Drawbacks of GPTQ framework, or rather future directions are the following:

- Weights quantization only: authors decided to limit their work to weights quantization only, thus they mention it as a future work: *"In addition, our study focuses on generative tasks, and does not consider activation quantization. These are natural directions for future work"*
- Uniform quantization method: range for weights quantization consists of uniform intervals, which can't fully represent the underlying non-uniform data.

In conclusion, these existing quantization frameworks all possess their unique set of features and drawbacks, which may be combined to introduce newer generation of quantization frameworks. The most promising one we consider the open-source framework GPTQ due to its advanced mathematical approach to quantization, without focus on hardware which makes it more agile for future enhancements and a stronger foundation for future work on quantization. With the GPTQ as a base for our research we also use some techniques and tooling from AIMET, which is compatible

with GPTQ, unlike ZeroQuant and its derivatives which are based on knowledge distillation of subnetworks and this approach is orthogonal to the one used in GPTQ.

In the following sections we will review in detail what and why we incorporate into our framework from AIMET. As well as how GPTQ may be enhanced by modules borrowed from AIMET framework.

Specifically – the following section will dissect AIMET framework in more detail to validate potential modules, which will become candidates for tools in our work. We will focus on the Post-Training Quantization part, relevant to our work.

Afterwards – we will proceed with a deep dive into GPTQ framework to understand in finer details each part, interaction between parts, quantization flow and validate decisions made in this framework.

2.6. AIMET – QUALCOM QUANTIZATION WHITE PAPER

In order to implement quantization for weights and activations we need to define set of tools and techniques for it to maximize accuracy / size tradeoff. Which depends on minimizing accuracy / perplexity drop, while also minimizing quantization range: $b = 8 \rightarrow 2^8 = 256$ bit $b = 4 \rightarrow 2^4 = 16$ bit.

In their *White Paper on Neural Network Quantization* (Nagel et al., 2021) [2] introduced set of techniques for neural network quantization that they combined into the framework for quantization. They introduced at that time state-of-the-art algorithms for mitigating the impact of quantization noise on the network's performance while maintaining low-bit weights and activations. They focused on range of quantization techniques: Quantization-Aware Training and Post-Training Quantization, symmetric/asymmetric, uniform/non-uniform, weight/activation, cross-layer equalization with bias correction. They also explained how quantization is related to other parts of the model, e.g., BatchNorm.

Besides the fundamentals / core of quantization, mentioned in 2.1. – 2.4., they also introduce such techniques as Cross Layer Equalization with Bias Correction, AdaRound and introduce nuances between Post Training Quantization and Quantization Aware Training.

For Post Training Quantization they iterate over the following techniques:

1. Quantization range setting
2. Cross-Layer Equalization
3. Bias correction
4. AdaRound

Each of these modules are provided in the sequence, in which they are intended to be applied.

Authors also provide a best practice Post Training Quantization pipeline which is *"based on relevant literature and extensive experimentation"*. This PTQ pipeline Figure 0.13. is intended for both computer vision and natural language processing models. Depending on the model some steps of this pipeline may not be necessary.

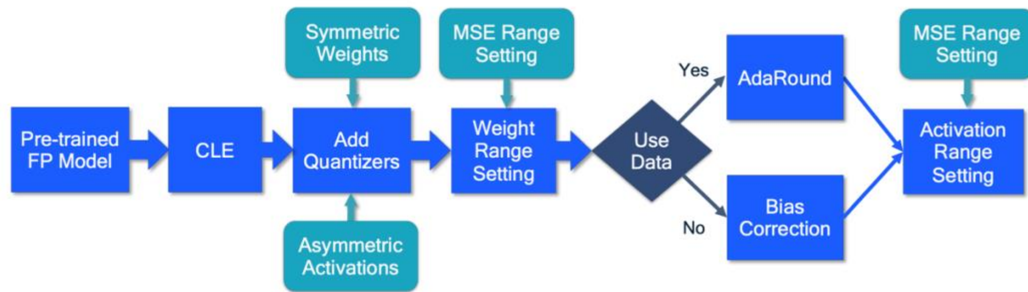

Figure 0.13 PTQ pipeline by AIMET (Nagel et al., 2021) [2]

1. They start with cross-layer equalization (CLE), a pre-processing step to make the full precision model more quantization-friendly, particularly benefiting models with depth-wise separable layers and per-tensor quantization.
2. Next, they select their quantizers and incorporate quantization operations into the network, typically recommending symmetric quantizers for weights and asymmetric ones for activations, with a preference for per-channel quantization if supported by the hardware/software stack.
3. For setting quantization parameters of weight tensors, they use layer-wise MSE-based criteria, for per-channel quantization, the min-max method may be used.
4. If a small calibration dataset is available, they apply AdaRound to optimize weight rounding, which is crucial for low-bit weight quantization (e.g., 4 bits) in post-training quantization.
5. If no calibration dataset is available and the network uses batch normalization, analytical bias correction is used instead.
6. Finally, they determine the quantization ranges for all data-dependent tensors (activations) using MSE-based criteria for most layers, which requires a small calibration set, or use BN-based range setting for a fully data-free pipeline.

Now let's review PTQ components suggested by AIMET.

1. Quantization range setting

   This component is a basic building block for quantization system. Authors review different approaches for range setting, such as min-max, mean squared error, cross entropy, and come to a conclusion that per-channel MSE always outperforms other methods by accuracy on both vision and language models. GPTQ has a dedicated parameter, which if enabled – uses MSE range setting.

2. Cross-Layer Equalization

   This technique is compatible with, but omitted in GPTQ and will be explored by us in more details in following sections.

3. Bias correction

   Same as CLE – compatible with but omitted in GPTQ and will be explored by us in more details in following sections.

4. AdaRound

   AdaRound is a weight rounding method, which requires a small amount of unlabeled data samples. It's a quantization method, which is an approximation of NP-hard QUBO problem.

$$\arg\min_{\mathbf{V}} \left\| f_a(\mathbf{W}\mathbf{x}) - f_a\left(\widetilde{\mathbf{W}}\hat{\mathbf{x}}\right) \right\|_F^2 + \lambda f_{\text{reg}}(\mathbf{V})$$

Figure 0.14 AdaRound (Nagel et al, 2021) [2]

Here, **V** is the continuous variable that we optimize over. Difference between initial and quantized layer outputs is formulated as a Frobenius norm. And finally, regularization term which returns 0 or 1 with coefficient λ. Difference is set between layer outputs instead of weights is to account for the non-linearity and avoid error accumulation across layers. In summary, it computes a data-dependent rounding by annealing a penalty term, which encourages weights to move to grid points corresponding to quantization levels.

In GPTQ authors use a different rounding approach, due to *"AdaRound (Nagel et al., 2020) or BRECQ (Li et al., 2021), are currently too slow for models with many billions of parameters, the main focus of this work."*
It's because AdaRound involves iterative optimization, which can be computationally intensive and time-consuming. For each weight, it needs to evaluate the impact of rounding and adjust accordingly, making the process slow for large models.
GPTQ's rounding is more efficient compared to methods like AdaRound because it does not require iterative optimization for each weight. Instead, it uses the Hessian matrix to make informed decisions about how to round weights.

Summarizing contribution of AIMET paper for our work, we integrate quantization of activations, which we will describe in finer details in section GPTAQ. We also consider cross layer equalization impact in our work, as well as bias correction. Finally, per other contributions from AIMET – Quantization-Aware Training is out of scope of our work and AdaRound is not compatible with our work foundation, which is GPTQ.

In the next section we will dissect GPTQ details similarly to AIMET and see inner working of this framework as well as which parts will be potentially subjected to enhancement.

2.7. GPTQ

GPTQ is a quantization framework for large scale neural networks – hundreds of billions of parameters. It is focused on minimizing quantization error in large language models from model families like OPT, BLOOM and LLAMA.
GPTQ – name comes from merging the name of the OPT model family with the abbreviation for post-training quantization (PTQ). GPTQ builds on top of previous work Optimal Brain Surgeon (OBS) (Frantar et al 2022) [4], which integrates pruning and quantization techniques using Hessian method.

Let us first take a look on a high-level overview of the OBS framework to explore what GPTQ builds upon, to then differentiate what GPTQ adds on top, and which parts are changed.

2.7.1. OBS

GPTQ follows Optimal Brain Quantization (OBQ; Frantar & Alistarh 2022) [4], which uses the Hessian of 2.1 Equation 1 Quantization Loss. Like OBS, GPTQs goal is to minimize the Hessian error from weights $W^{(\ell)}$ quantization in layer ℓ .

Equation 1 Quantization Loss

$$\mathbb{E}[\mathcal{L}(\theta + \epsilon) - \mathcal{L}(\theta + \epsilon)] \approx \sum_{\ell} \left\| \mathrm{W}^{\ell}\mathrm{X}^{\ell} - \widehat{\mathrm{W}}^{\ell}\mathrm{X}^{\ell} \right\|_F^2 \qquad 2.1$$

OBS is comprised from 2 parts:

- the main one which explains the whole frameworks through integrations into neural network pruning.
- OBQ (Optimal Brain Quantizer) – in this part they translate techniques from the main part relating to pruning onto the quantization application itself.

Further we will use OBQ and OBS interchangeably relating to the techniques from a quantization perspective.

2.7.1.1. Hessian – what to remove.

To understand OBS we need to understand a concept of Hessian Matrix H and its role in neural network quantization.

Hessian is a square matrix of second-order partial derivatives of a scalar-valued function, describing the local curvature of the function. Figure 0.15 True Hessian. True Hessian form with derivatives calculation is costly, it can effectively be approximated as Equation 2 Hessian approximation Where $X^{(\ell)}$ is the input data in layer ℓ. This somewhat resembles a confusion matrix of features cross-interaction.

$$\mathbf{H}f = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} & \cdots \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} & \cdots \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Figure 0.15 True Hessian

Equation 2 Hessian approximation

$$H(\ell) = X^{(\ell)} X^{(\ell)T}$$

Hessian in OBQ is used as sensitivity parameter for quantization calibration. Alternative approaches, such as AdaRound discarded use of Hessian, as they were not considering a Hessian approximation: Equation 2 Hessian approximation, but only a True Hessian – Figure 0.15 True Hessian, which was too costly: *"The memory and computational complexity of calculating the Hessian is impractical for general use-cases."* (Nagel et al., 2021) [2].

2.7.1.2. Gaussian elimination – how to remove.

The removal of one parameter p simply drops the corresponding row and column from H, the inverse can be updated by removing parameter p directly using a single step of Gaussian elimination Equation 3 Gaussian elimination of element p from inverse Hessian.

Equation 3 Gaussian elimination of element p from inverse Hessian

$$H_{-p}^{-1} = \left(H^{-1} - \frac{1}{[H^{-1}]_{pp}} H_{:,p}^{-1} H_{p,:}^{-1}\right)_{-p}$$

Gaussian elimination of row and column p *in* H−1 *followed by dropping them completely. This has* $\Theta(d_{col}^2)$ *time complexity.*

Utilizing this Gaussian elimination technique it is now possible to prune (or in our case quantize) target weight from weight matrix W, thus pseudocode for the following algorithm for single row update can be created Figure 0.16 OBS algorithm

Algorithm 1 Prune $k \leq d_{\text{col}}$ weights from row $\mathbf{w}$ with inverse Hessian $\mathbf{H}^{-1} = (2\mathbf{X}\mathbf{X}^\top)^{-1}$ according to OBS in $O(k \cdot d_{\text{col}}^2)$ time.

$M = \{1, \ldots, d_{\text{col}}\}$
for $i = 1, \ldots, k$ **do**
  $p \leftarrow \text{argmin}_{p \in M} \frac{1}{[\mathbf{H}^{-1}]_{pp}} \cdot w_p^2$
  $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{H}_{:,p}^{-1} \frac{1}{[\mathbf{H}^{-1}]_{pp}} \cdot w_p$
  $\mathbf{H}^{-1} \leftarrow \mathbf{H}^{-1} - \frac{1}{[\mathbf{H}^{-1}]_{pp}} \mathbf{H}_{:,p}^{-1} \mathbf{H}_{p,:}^{-1}$
  $M \leftarrow M - \{p\}$
end for

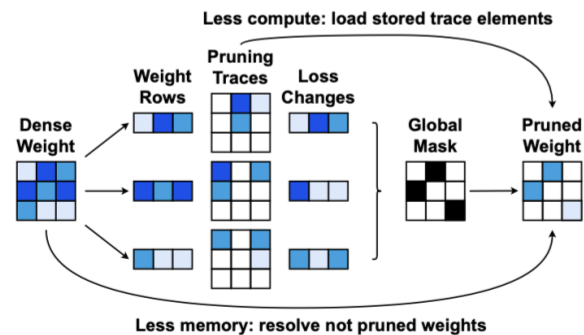

Figure 1: Efficient global OBS using the row-wise results.

Figure 0.16 OBS algorithm

This process is then extrapolated to multiple rows in parallel using the following insights of rows independence:

1. There is no Hessian interaction between rows
2. Resulting weights only depend on the total number of pruned parameters.
3. Order in which weights are pruned in each row is fixed because loss only depends on the previously pruned weights in the same row

2.7.1.3. OBS → OBQ – pruning → quantization.

To map OBS to a quantized projection, authors set the target of the Lagrangian constraint. Then they formulate a target quantized weight selection and an update rule: Equation 4 OBQ selection and update rule.

Equation 4 OBQ selection and update rule

$$w_p = argmin_{w_p} \frac{(quant(w_p) - w_p)^2}{{H^{-1}}_{pp}}, \qquad \delta_p = -\frac{w_p - quant(w_p)}{{H^{-1}}_{pp}} \cdot {\mathrm{H}^{-1}}_{:,p}$$

- Selection of a weight to be quantized, which incurs the least error after quantization.
- Update rule for all unquantized weights to the right, which are updated with the Hessian in this formula.

This concludes OBS framework and it's fundamentals on which GPTQ is built. Now that we understand role of Hessian and target weight selection and update – we can proceed with GPTQ enhancements.

The following numbered items explain in which ways GPTQ extends the underlying OBQ framework.

2.7.2. Arbitrary Order – quantization order doesn't matter

OBQ always picks the weight with the least quantization error, but the difference between quantizing the weights in arbitrary order is negligible, especially on large scale. They suggest that it's because the error is transferred to the right – to the next unquantized weights. That is why in GPTQ they moved on from weight-by-weight updates and instead process all the weights in column in parallel as shown on Figure 0.17 GPTQ - update all items in column in parallel.

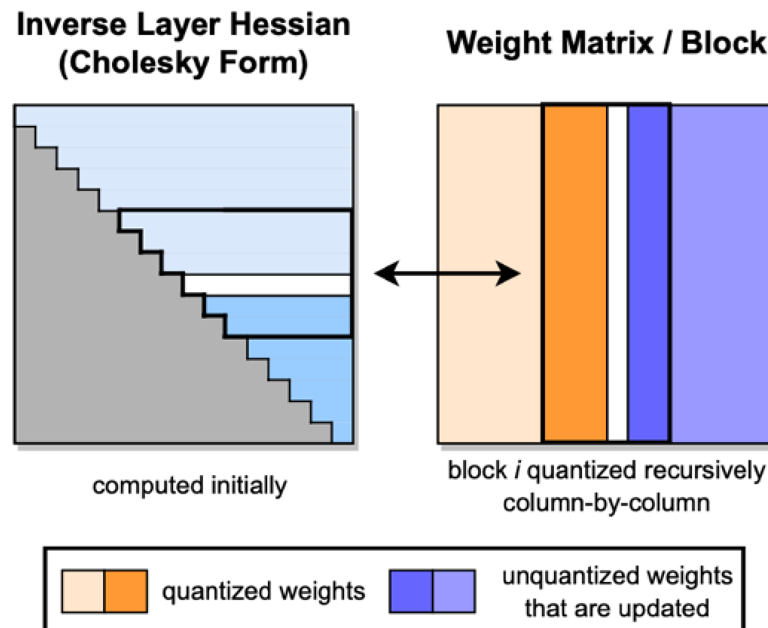

Figure 2: GPTQ quantization procedure. Blocks of consecutive *columns* (bolded) are quantized at a given step, using the inverse Hessian information stored in the Cholesky decomposition, and the remaining weights (blue) are updated at the end of the step. The quantization procedure is applied recursively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

Figure 0.17 GPTQ - update all items in column in parallel

2.7.3. Lazy Batch-Updates

To reduce data transfer, GPTQ applies the update of Equation 4 OBQ selection and update rule only to a block of B columns which includes our target column. To update the columns outside of block B, the error Figure 0.18 GPTQ block update rule & Hessian with set of columns removed is accumulated while each column in the block B is processed.

After all columns in block B are processed – accumulated errors is then applied in one go to all columns outside of block B. In the GPTQ Experiments Blocks with the size of 128 were processed.

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

Figure 0.18 GPTQ block update rule & Hessian with set of columns removed

It's worth noting that GPTQ authors also experimented with applying a quantization to groups of g consecutive weights (groups of columns).
These groups can be static or dynamic:

- Static groups – calculated before blocks $W_{[\,:,\ i\,:\,i+\text{group}]}$
- Dynamic groups – calculated inside blocks b $W_{[\,:,\ (b+i)\,:\,(b+i+\text{group})]}$

2.7.4. Cholesky Reformulation – minimize Hessian inversion error

Cholesky decomposition of inverse Hessian H^{-1} reduces accumulated error of multiple Hessian row & column removals from OBQ, making it more numerically stable.

Cholesky decomposition is a decomposition of a positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is useful for efficient numerical solutions. In GPTQ, specifically – the L^T upper triangular matrix of Hessian Cholesky decomposition is used.

$$\begin{aligned} A &= \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{21} & a_{22} & a_{32} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \\ &= \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} \equiv LL^T \\ &= \begin{pmatrix} l_{11}^2 & l_{21}l_{11} & l_{31}l_{11} \\ l_{21}l_{11} & l_{21}^2 + l_{22}^2 & l_{31}l_{21} + l_{32}l_{22} \\ l_{31}l_{11} & l_{31}l_{21} + l_{32}l_{22} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix} \end{aligned}$$

Figure 0.19 Cholesky decomposition
rosettacode.org/wiki/Cholesky_decomposition

Alternative Matrix decomposition techniques also exist in related works. For example, in their work (Yao et al 2022) [7] proposed Low Rank Compensation (LoRC) method, which works synergistically with PTQ and FGQ
LoRC approximates the error E with $\hat{E} = \hat{U}\,\hat{V}$ by using two low-rank matrices $\hat{U}$ and $\hat{V}$ This results in a more accurate approximation of the original weight matrix W by $\widehat{W}_{lorc} = \widehat{W} + \hat{E}$, thereby reducing quantization errors.
The objective of LoRC is to achieve a good approximation of the error matrix E using low-rank matrices, with minimal impact on the increase in model size. For instance, consider the standard transformer models, where each layer is comprised of a multi-headed attention (MHA) module and a multi-linear perception (MLP) module.

Packing – "Quantization is the process of mapping input values from a large set to output values in a smaller set e.g. rounding and truncation."
In our case – FP16 (floating-point) values get converted into INT8/INT4/INT3/INT2 (integer). However, most of modern frameworks for working with neural networks still store weights in a floating-point format. And even though we converted floating point values into the integer format that takes up a lot less space – these values will still be stored as floating-point format in memory. Thus, we need to squeeze our integers into floating-point numbers in some way to keep stored models in a compatible format.

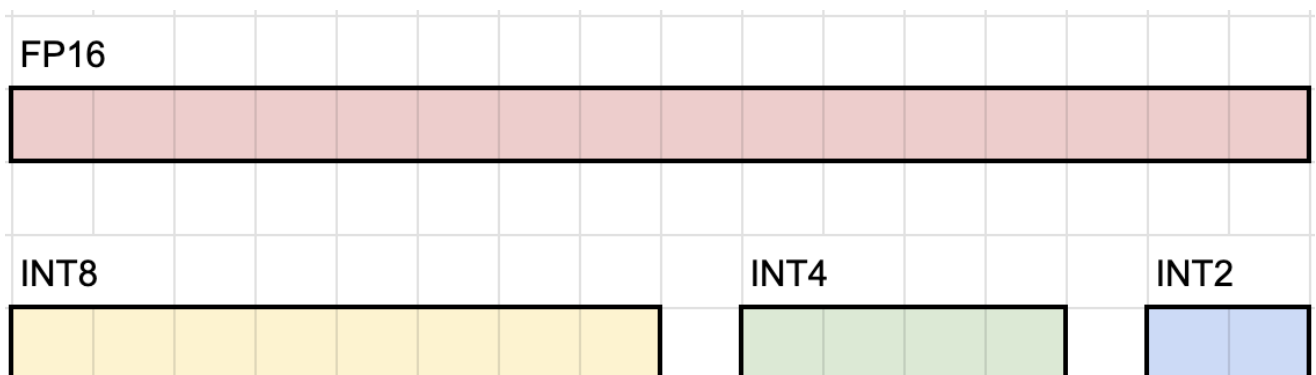

Figure 0.20 number formats memory footprint

In GPTQ authors perform *packing*. Packing in this context is manually initializing floating-point values and overwriting them with integers using bitwise operators. On the Figure 0.21 packing integers into floating-point format we can see how different integer formats fit into one floating-point format.

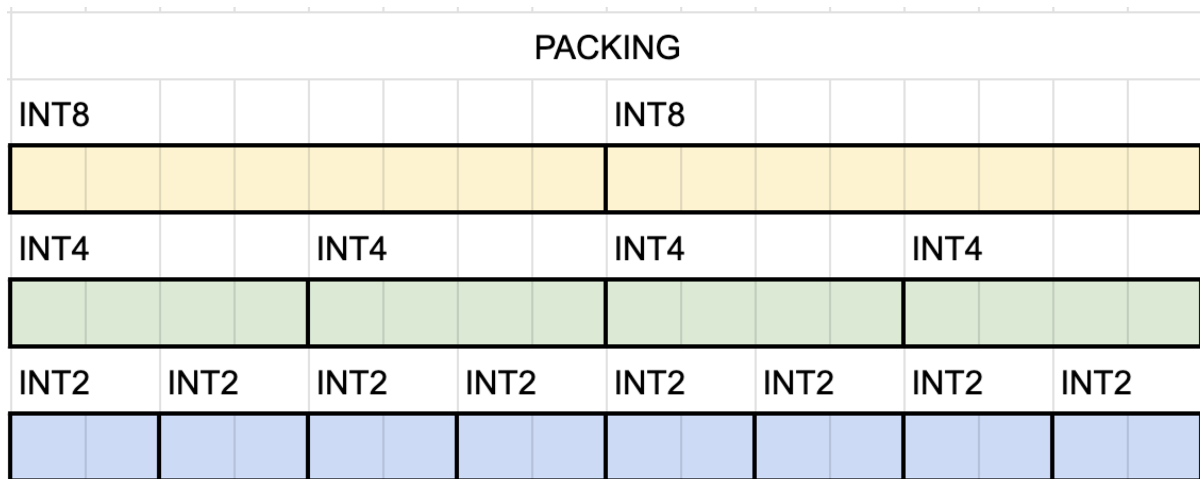

Figure 0.21 packing integers into floating-point format

Implementation example in python from (Frantar et al, 2023) [3] using bitwise *or*, *shift left*, *shift right* and *and*:

```
i, row = 0, 0
while row < qweight.shape[0]:
    for j in range(i, i + 10):
        qweight[row] |= intweight[j] << (3 * (j - i))
    i += 10
    qweight[row] |= intweight[i] << 30
    row += 1
    qweight[row] |= (intweight[i] >> 2) & 1
    i += 1

    for j in range(i, i + 10):
        qweight[row] |= intweight[j] << (3 * (j - i) + 1)
    i += 10
    qweight[row] |= intweight[i] << 31
    row += 1
    qweight[row] |= (intweight[i] >> 1) & 0x3
    i += 1

    for j in range(i, i + 10):
```

```
        qweight[row] |= intweight[j] << (3 * (j - i) + 2)
    i += 10
    row += 1
qweight = qweight.astype(np.int32)
self.qweight = torch.from_numpy(qweight)
```

In conclusion of GPTQ section provided a very interesting quantization technique with smart improvements for large scale models. And there are few potential places for enhancements, one of which – activation quantization, they explicitly provided in future work and explained why it was left out in GPTQ, even though it is present in OBS:

"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 orthogonal techniques (Yao et al., 2022)" (Frantar et al, 2023) [3].

Besides activations quantization, other potential improvements include:

- Cross layer equalization
- Bias correction
- Hessian Eigenvalues integration into quantization process

The next section is dedicated to a deep dive into our contribution into GPTQ framework, our decision process, and results.

GPTAQ

3.1. ACTIVATIONS QUANTIZATION

While weight quantization focuses on some version of rounding weights inside a model layer (e.g., linear/fully-connected/convolutional/etc.) – activations quantization is the process of rounding outputs from layers Figure 0.1 activation quantization NN flow, which is often abbreviated as e.g., W4A8 – 4 bit weight quantization with 8-bit activation quantization.

Figure 0.1 activation quantization NN flow

Authors of GPTQ mention potential area for future work as "*In addition, our study . . . does not consider activation quantization. These are natural directions for future work, and we believe this can be achievedwith carefully-designed GPU kernels and existing techniques*". As stated by the author (Elias Frantar) himself: *"accurately quantizing activations of LLMs requires somewhat different (but orthogonal) techniques"*. This work in part focuses on **activations quantization** without modifications to GPU kernels. But why do we need activation quantization? Activation quantization increases inference speed; however, it also reduces accuracy/perplexity and increases quantization time.

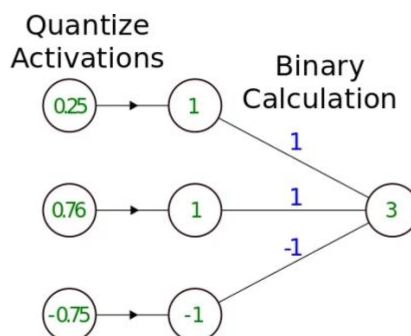

Figure 0.2 medium.com/@joel_34050/quantization-in-deep-learning-478417eab72b

We only quantize hidden layers' activations (outputs), meaning we don't quantize input into 1 layer and output from the last one. Last layer in the classic transformer architecture being SoftMax, quantization of which is non-trivial. To be specific, we only can quantize activations of the following layers: Convolutional, Linear (including attention) and ReLU.

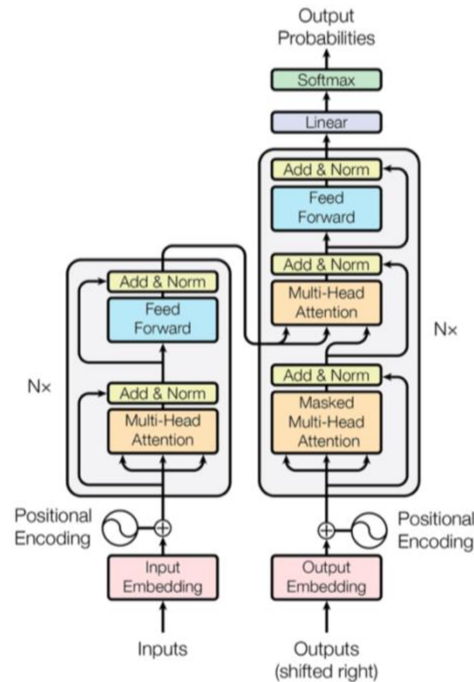

Figure 0.3 transformer architecture (Vaswani et al, 2017)[8]

ReLU activation function is widely used in neural networks due to simplicity and computational efficiency. ReLU is effectively $y = \max(0, x)$. With several modifications such as LeakyReLU, basic ReLU is the activation which is used in almost all LLMs, that's why we need to pay special attention to it. Luckily ReLU doesn't require any computational overhead while quantizing as ReLU (**Error! Reference source not found.**) is scale-equivariant meaning $ReLU(sx) = s\ ReLU(x)$ thanks to this properties quantization of ReLU activation is pretty straight-forward. I'd like to reiterate that we don't quantize activation *function* itself, but rather layer *outputs*. ReLU doesn't have any learnable parameters, thus can't be quantized, and was mentioned just to verify that quantization doesn't affect layers, whos outputs go through the ReLU.

Our contributions consist of the following steps:

1. Analyze OBQ → GPTQ transition from weights + activations quantization to exclusively weights quantization.
2. Compare different quantization methods for activations.
3. Apply these quantization methods on GPTQ layers with quantized weights.
4. Analyze / validate the results.

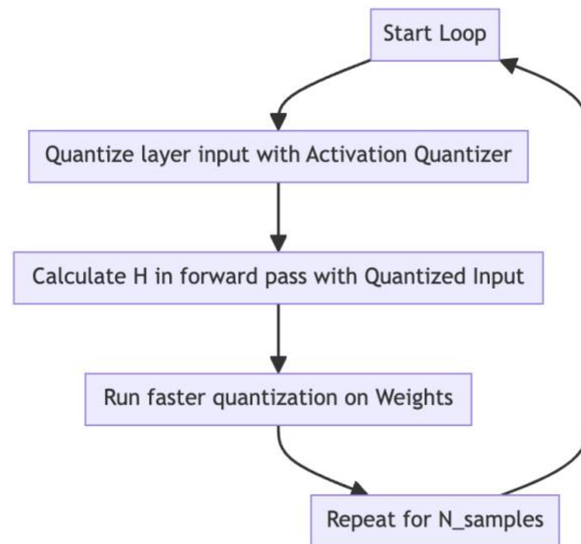

Figure 0.4 GPTQ flow with activation quantization included

In this section we will review why authors of GPTQ abandoned activation quantization present in the GPTQ ancestor – OBS and see how we can overcome these challenges in our work. We will also review different quantization techniques, such as:

- Round to Nearest
- Hessian based – used in GPTQ weights.
- LAPQ – used in OBS activations.
- Quantized Weights re-optimization – mentioned in OBS re-optimize weights to better align with the quantized inputs.
- Token-wise – used in ZeroQuant activations.

3.1.1. *Round to Nearest*

Round to Nearest – is a most basic, straightforward method of quantization where each value (weights or activations of a neural network) is rounded to the nearest value that can be represented in the target quantization format. For example, when converting from 32-bit floating point to 8-bit integers, each value is scaled to fit within the range of the 8-bit integers, then rounded to the nearest integer.

Implementation

- Quantization method

```
def round_to_nearest_quantization(x, num_bits=8):
    qmin = 0.
    qmax = 2.**num_bits - 1.
    min_val, max_val = x.min(), x.max()

    scale = (max_val - min_val) / (qmax - qmin)
    zero_point = qmin - min_val / scale

    q_x = zero_point + x / scale
    q_x = torch.round(q_x)
    q_x = torch.clamp(q_x, qmin, qmax)

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

- Usage in GPTQ

```
gptaq[name].activation_quantizer = ActivationQuantizer()
gptaq[name].activation_quantizer.configure(
    args.abits, perchannel=True, sym=args.sym, mse=True, trits=args.trits
)
gptaq[name].activation_quantizer.round_to_nearest_quantization(w.unsqueeze(1))
```

There are a lot of flaws with this method:

- No data-specific parameters calibration: quantization ranges won’t be precise.
- No sensitivity adjusting.
- Uniformity, which is rarely how data is represented

However, understanding this method is crucial for understanding of other more intricate methods.

3.1.2. *Hessian based*

Hessian based – activation quantization by the same principle as weights. It’s worth noting that OBQ framework builds on the basic assumption that the Taylor approximation at the given point is assumed to have almost 0 gradient. The weights are assumed to be at or very close to a local minimum after training (before quantization), which means the gradient of the loss function with respect to the weights is zero. This is why Lagrangian, rule for weight selection and rule for update of unquantized weights are as follows:

$$\mathcal{L}(\delta_p, \lambda) = \delta_p^\top \boldsymbol{H} \delta_p + \lambda \left(e_p^\top \delta_p - (-w_p) \right)$$

$$w_p = \arg\min_{w_p} \frac{w_p - quant(w_p)^2}{[\boldsymbol{H^{-1}}]_{pp}} \qquad \delta_p = -\frac{w_p - quant(w_p)}{[\boldsymbol{H^{-1}}]_{pp}} \cdot \boldsymbol{H}_{:,\boldsymbol{p}}^{\boldsymbol{-1}}$$

Other methods, in order to perform per-layer optimization are substituting the dense model inputs (X_{dense}) with the corresponding inputs in the compressed model (X_{comp}) . However, this approach is inadequate for OBQ. When the Hessian is computed on (X_{comp}), the initial dense weights are no longer at a local minimum (with zero gradient), violating a key OBQ assumption. Thus previous 3 equations now need to account for the gradient (bold and underlined).

$$\mathcal{L}(\delta_p, \lambda) = \delta_p^\top \boldsymbol{H} \delta_p + \lambda \left(e_p^\top \delta_p - (-w_p) \right) \underline{\boldsymbol{+\nabla\mathcal{L}(w)^\top \delta_p}}$$

$$w_p = \arg\min_{w_p} \frac{w_p - quant(w_p)^2 \underline{\boldsymbol{+\nabla\mathcal{L}(w)^\top \delta_p}}}{[\boldsymbol{H^{-1}}]_{pp}} \quad \delta_p = -\frac{w_p - quant(w_p)}{[\boldsymbol{H^{-1}}]_{pp}} \cdot \boldsymbol{H}_{:,\boldsymbol{p}}^{\boldsymbol{-1}} \underline{\boldsymbol{-H^{-1}\nabla\mathcal{L}(w}}$$

This is why we can’t just re-apply Hessian method for activation quantization on an output from the layer with already quantized weights – values are no longer in the local minimum and gradient is non-zero.

Even though GPTQ doesn't include activation quantization they still have elements which touch on activation quantization, one example would be – *--act-order* – heuristic param for quantizing columns in order of decreasing activation size.

If we were to implement Hessian based activation quantization following the same logic as weights, python pytorch implementation code would look like:

- Quantizer initialization in the model-specific quantization pipeline

```
gptaq[name].activation_quantizer = ActivationQuantizer()
gptaq[name].activation_quantizer.configure(
  args.abits, perchannel=True, sym=args.sym, mse=True, trits=args.trits
)
```

- ActivationQuantizer class itself

```
class ActivationQuantizer(Quantizer):
 act_q_enabled = False

 def __init__(self):
  super(ActivationQuantizer, self).__init__()

 def quantize(self, x):
  return super().quantize(x)

 def configure(self, bits, **kvargs):
  self.act_q_enabled = bits < 16
  return super().configure(bits, **kvargs)

class GPTAQ(GPTQ):
 activation_quantizer: ActivationQuantizer
```

```
out: torch.Tensor


def __init__(self, layer):
  super().__init__(layer)


def add_batch(self, inp, out):
  if self.activation_quantizer.act_q_enabled:
    inp = self.activation_quantizer.quantize(inp)
  return super().add_batch(inp, out)
```

3.1.3. *OBS / LAPQ*

OBS / LAPQ activations quantization. In OBS the zero point and quantization scale are optimized per one input batch for each layer same as for weights but at the tensor level instead of the channel level. Method being the same as LAPQ (Loss-aware post-training quantization). Activations are quantized asymmetrically per-tensor (meaning to the whole tensor, instead of each channel/weight). As with weights – this optimization is performed independently for each layer, and the corresponding quantization information is stored. While more advanced methods, such as *reoptimizing the weights* to better align with the quantized inputs may be possible, authors concluded that simple procedure described as *"optimizing the zero point and quantization scale for one input batch of each layer"* is already effective.

Issue presented on the previous page in *Hessian based* section can be resolved by reoptimizing the dense weights for the new inputs using the closed-form solution of linear regression $\mathrm{W}^{\mathrm{T}} = (\mathrm{XX}^{\mathrm{T}})^{-1}\mathrm{XY}^{\mathrm{T}}$, which restores the zero gradient, allowing OBQ to be correctly applied. The matrix (XY^T) $is\ a$ $(d_{col} \times d_{row})$ matrix that can be accumulated over multiple batches similar to the OBQ Hessian $(2XX^T)$, without significantly increasing memory consumption.

As a demonstration, authors use sequential OBQ with the weights re-optimization [closed-form solution of linear regression $W^T = (XX^T)^{-1}XY^T$] to quantize ResNet18 with results reported in Figure 0.5 OBQ sequential activation quantization with weights re-optimization. For 4 and 3 bits, the results were essentially the same as the independent version.

Method	ResNet18 – 69.76 4bit	3bit	2bit
AdaRound	69.34	68.37	63.37
AdaQuant	68.12	59.21	00.10
BRECQ	69.37	68.47	64.70
OBQ + BNT	69.56	68.69	64.04
OBQ – sequential	69.56	68.68	64.93

Table 10: Comparison with sequential OBQ.

Figure 0.5 OBQ sequential activation quantization with weights re-optimization

In practice they just wrap modules in ActQuantWrapper, which calls quantization on layer forward() pass to quantize the whole model at once after weights quantization.

OBS (Frantar et al, 2022)[4] activation quantization implementation:

- Quantizer initialization in the model-specific quantization pipeline

```
if aquant:
    add_actquant(modelp)
```

- ActQuantWrapper class itself

```
class ActQuantWrapper(nn.Module):
    . . .
    def forward(self, x):
        return self.module(self.quantizer.quantize(x))
```

- Method for adding ActQuantWrapper to the model

```
def add_actquant(module, name='', layers=[nn.Conv2d, nn.Linear]):
    if isinstance(module, ActQuantWrapper):
        return
    for attr in dir(module):
        tmp = getattr(module, attr)
        if type(tmp) in layers:
            setattr(module, attr, ActQuantWrapper(tmp))
        if type(tmp) == nn.Sequential:
            replaced = []
            for i, child in enumerate(tmp.children()):
                if type(child) in layers:
                    replaced.append(ActQuantWrapper(child))
                else:
                    replaced.append(child)
            setattr(module, attr, nn.Sequential(*replaced))
        if type(tmp) == torch.nn.ModuleList:
            replaced = []
            for i, child in enumerate(tmp.children()):
                if type(child) in layers:
                    replaced.append(ActQuantWrapper(child))
                else:
                    replaced.append(child)
            setattr(module, attr, nn.ModuleList(replaced))
    for name1, child in module.named_children():
```

```
add_actquant(child, name + '.' + name1 if name != '' else name1, layers)
```

3.1.4. *Weights re-optimization*

Quantized Weights re-optimization – mentioned in OBS re-optimize weights to better align with the quantized inputs. Re-optimizing the dense weights for the new inputs using the closed-form solution of linear regression. The formula provided is:

$$W^\top = (XX^\top)^{-1}\boldsymbol{X}\boldsymbol{Y}^\top$$

where: $\boldsymbol{X}$ are the inputs, $\boldsymbol{Y}$ are the outputs, $\boldsymbol{W}^T$ represents the re-optimized weights.

By re-optimizing the weights using this formula, the gradient with respect to these new weights becomes zero again, thus restoring the necessary conditions for OBQ to be applied correctly.

1. Adjust the dense weights for the new inputs (X_{comp}) to make the gradient zero.
2. Use linear regression solution $\boldsymbol{W}^\top = (\boldsymbol{X}\boldsymbol{X}^\top)^{-1}\boldsymbol{X}\boldsymbol{Y}^\top$ to re-optimize the weights.

The re-optimization step ensures that even after changing the inputs to the compressed model, the dense weights are adjusted such that their gradient is zero. This step is crucial to maintaining the validity of OBQ and allows for accurate quantization of the model.

Implementation

- Re-optimisation method for layer inputs and outputs

```
def reoptimize_weights(X: torch.Tensor, Y: torch.Tensor):
    # Compute (X^T X)^-1 X^T Y
    XTX_inv = torch.inverse(torch.matmul(X.T, X))
    XTY = torch.matmul(X.T, Y)
    W_opt = torch.matmul(XTX_inv, XTY)
    return W_opt
```

- Usage – calibration step

```
class GPTAQ(GPTQ):
  activation_quantizer: ActivationQuantizer
  out: torch.Tensor

  def add_batch(self, inp, out):
    self.layer.weights = reoptimize_weights(inp, out)
    if self.activation_quantizer.act_q_enabled:
```

```
    inp = self.activation_quantizer.quantize(inp)
  return super().add_batch(inp, out)
```

3.1.5. *Token-wise*

ZeroQuant Token-wise – used in ZeroQuant activations.

In static quantization, we pre-calculate the min and max values for activations during a calibration phase, which works fine for small models. However, large models like GPT-3 and BERT have much more *variation* in their activation ranges, making static quantization less effective and leading to accuracy loss. To solve this, we use token-wise quantization, where we dynamically calculate the min and max values for each token separately, reducing quantization errors. This method has been shown to significantly improve the accuracy of these large models. However, it introduces additional computational overhead, which can be mitigated by using optimized inference backends, such as those implemented in ZeroQuant, employing techniques like kernel fusion.

Implementation

- Token Quantizer which uses calibration data

```
class TokenWiseQuantizer(nn.Module):
    def __init__(self, bitwidth=8):
        super(TokenWiseQuantizer, self).__init__()
        self.bitwidth = bitwidth
        self.scale = None
        self.zero_point = None

    def forward(self, x):
        min_val, max_val = x.min(dim=-1, keepdim=True)[0], x.max(dim=-1,keepdim=True)[0]
        self.scale = (max_val - min_val) / (2 ** self.bitwidth - 1)
        self.zero_point = min_val
        quantized = torch.round((x - self.zero_point) / self.scale)
        return quantized, self.scale, self.zero_point

    def dequantize(self, quantized, scale, zero_point):
        return quantized * scale + zero_point
```

- Normalization layer

```
class LayerNormWithQuant(nn.Module):
    def __init__(self, normalized_shape, bitwidth=8):
        super(LayerNormWithQuant, self).__init__()
        self.layer_norm = nn.LayerNorm(normalized_shape)
```

```
        self.quantizer = TokenWiseQuantizer(bitwidth)

    def forward(self, x):
        x = self.layer_norm(x)
        quantized, scale, zero_point = self.quantizer(x)
        return quantized, scale, zero_point
```

- Model integration

```
class TransformerModelWithQuant(nn.Module):
    def __init__(self, model, bitwidth=8):
        super(TransformerModelWithQuant, self).__init__()
        self.model = model
        self.bitwidth = bitwidth
        self.quantizer = TokenWiseQuantizer(bitwidth)

    def forward(self, input_ids, attention_mask=None):
        # vanilla forward pass
        output = self.model(input_ids, attention_mask=attention_mask)
        quantized_output, scale, zero_point = self.quantizer(output.last_hidden_state)
        # dequantization of output for next layers
        dequant_output = self.quantizer.dequantize(quantized_output, scale, zero_point)
        return dequant_output
```

ZeroQuant focuses on optimizing transformer models, such as GPT-3 and BERT. The methods described, like token-wise quantization and kernel fusion, are used to address the unique challenges presented by large-scale transformer architectures. However, the underlying principles and techniques of quantization could potentially be applied to other neural network architectures as well, although the specific implementations and optimizations may differ. The focus on transformers is due to their prevalence and complexity, which make them a prime target for advanced quantization strategies like those employed by ZeroQuant. This method was mentioned by authors of GPTQ as a candidate for activation quantization.

Conclusion, most promising candidate is a ZeroQuant token-wise quantization method, on the second place is method mentioned in OBS – weights re-optimization. All methods aside from Hessian are candidates.

3.2. CROSS-LAYER EQUALIZATION

Cross-layer equalization (CLE) is a technique used to prepare a neural network for quantization. In CLE, the weights of certain layers are adjusted to have similar ranges, which helps prevent large weight values (weight outliers) from causing errors during quantization. By balancing these weights across layers, the network can maintain its accuracy even after the precision is reduced. This process is particularly helpful for models with layers that have vastly different weight scales.

Before and after cross-layer equalization can be visualized as follows.

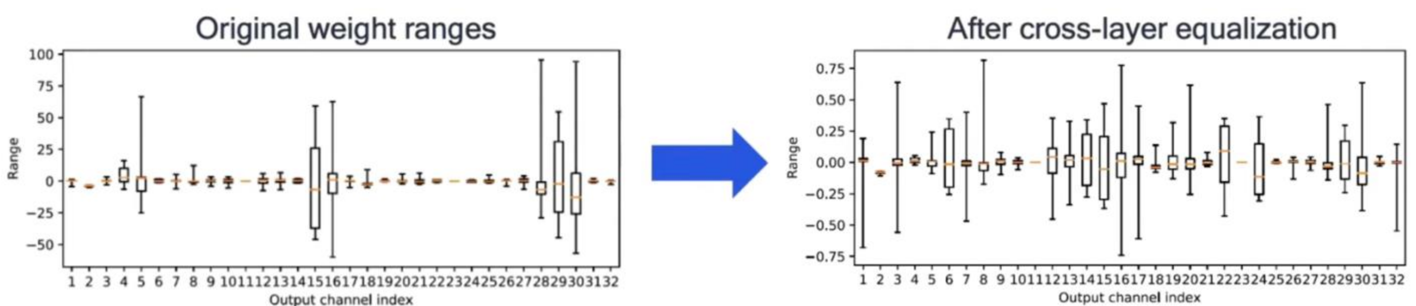

Figure 0.6 cross layer equalization visualization

Here Figure 0.6 cross layer equalization we can see that channels with overwhelming magnitudes 15, 16, 28, 29, 30 are diminishing the impact from other channels, with less sizable quantization ranges, which makes layer almost completely dependent on these channels with largest magnitudes, which leads to waste of computational resources during training and inference and strong bias towards results from these channels.

After cross layer equalization we see much more equalized channel ranges, which mitigates problems presented above. Even though there are still some channels with disproportional ranges, situation is not as drastic as was before. CLE focuses on scaling the weights of linear layers to balance their magnitudes. Activation functions like ReLU don't have learnable parameters, and thus, they don't directly participate in the weight scaling process. Their presence and output are implicitly part of the forward pass and do affect the distribution of activations.

CLE is accomplished by averaging channels in the neighboring layers using formula $s_i = \frac{1}{r_i^{(2)}} \sqrt{r_i^{(1)} r_i^{(2)}}$

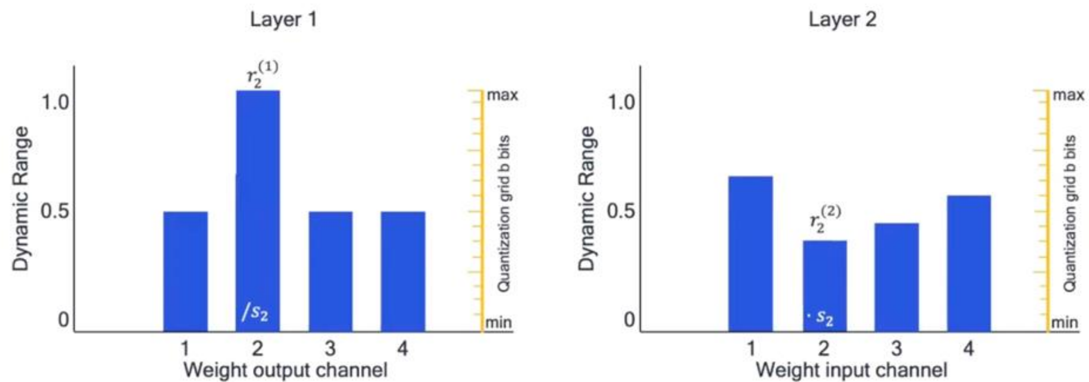

Figure 0.7 4 channels before cross layer equalization visualization
youtu.be/KASuxB3XoYQ

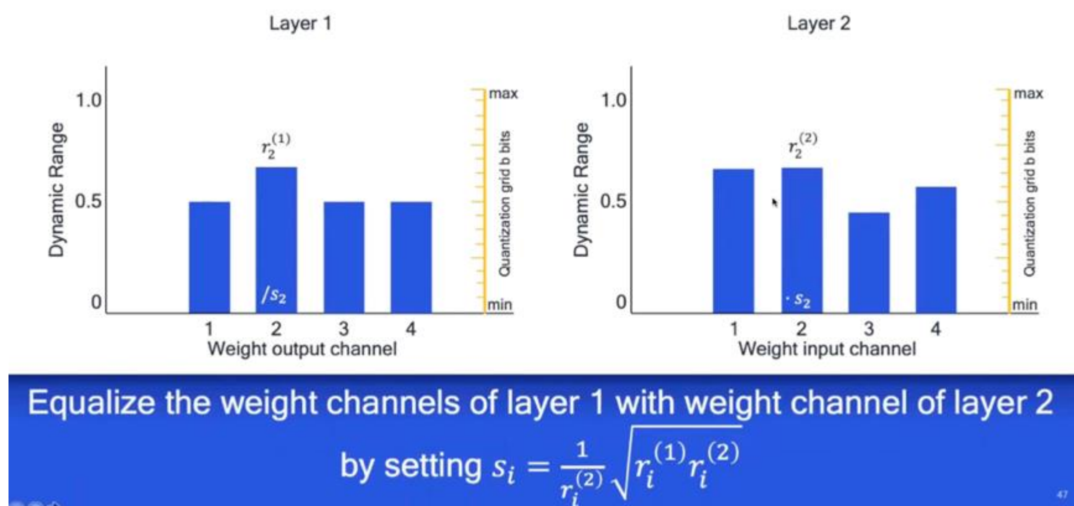

Figure 0.8 4 channels after cross layer equalization visualization
youtu.be/KASuxB3XoYQ

Basic Implementation of cross layer equalization in python + pytorch

```python
def CLE_2_layers(layer1, layer2):
    """s = 1/r2 * √(r1*r2)"""
    max_weight = torch.max(torch.abs(layer1.weight))
    next_max_weight = torch.max(torch.abs(layer2.weight))
    scale = torch.sqrt(max_weight / next_max_weight)
    layer1.weight.data /= scale
```

```
layer2.weight.data *= scale
```

3.2.1. QLLM

Alternative techniques for mitigating drastic differences in ranges exist, such as *QLLM: accurate and efficient low-bitwidth quantization for large language models* (Liu et al, 2023) [9]. The essence of their QLLM approach is in an adaptive channel reassembly strategy designed to tackle activation outliers. The central concept is to redistribute the outlier magnitudes to different channels by first disassembling and then reassembling the channels.

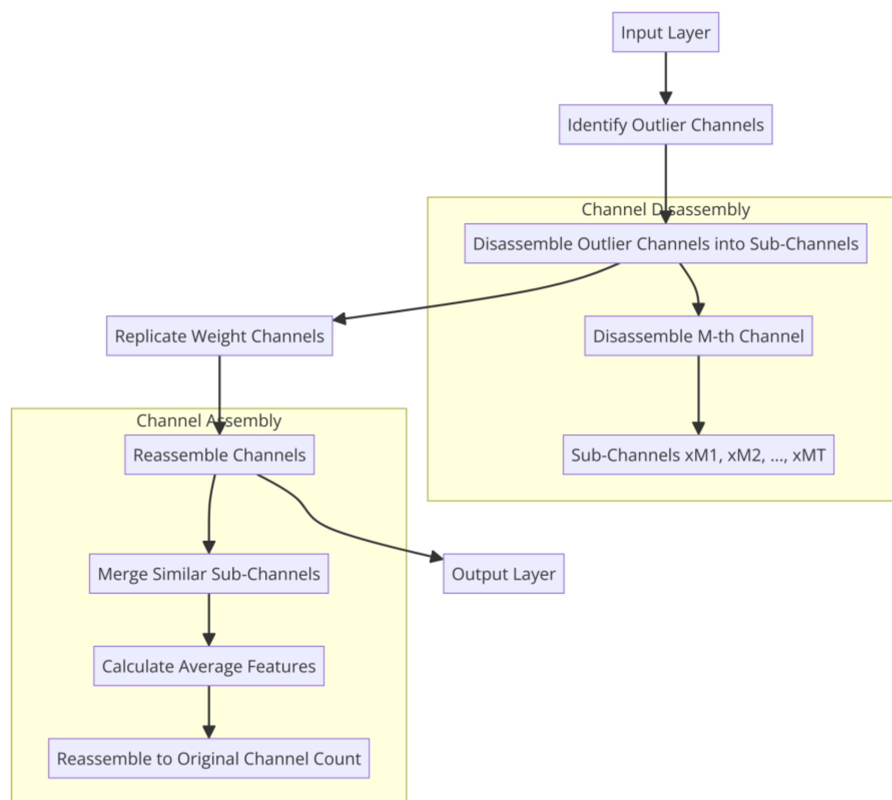

Figure 0.9 QLLM

Authors' method breaks down channels with large values (outliers) into several smaller sub-channels, making them easier to quantize. If one channel has large values, it is split into multiple smaller channels, each – a fraction of the original values. This helps in preserving the information of the outliers without changing the final output. After disassembly, the number of channels increases, so the authors merge similar channels back together to maintain efficiency. The merging is done by averaging similar channels while keeping the overall output unchanged.

Basic Implementation of QLLM channel disassembly-assembly process

- Disassembly step

```
class ChannelDisassembly(nn.Module):
    def forward(self, x):
        # get outlier channels
        max_vals = torch.max(torch.abs(x), dim=(0, 2, 3))[0]
        outlier_indices = (max_vals > self.threshold).nonzero(as_tuple=True)[0]
        disassembled_channels = []

        # disassemble each outlier channel
        for idx in outlier_indices:
            T = torch.ceil(max_vals[idx] / self.threshold).int().item()
            sub_channels = x[:, idx, :, :].unsqueeze(1) / T
            disassembled_channels.append(sub_channels.repeat(1, T, 1, 1))

        disassembled_x = torch.cat(disassembled_channels, dim=1)
        return disassembled_x, outlier_indices
```

- Assembly step

```
class ChannelAssembly(nn.Module):
    def forward(self, x, original_channels):
        # calculate the distance metric for merging channels
        def calculate_distance(c1, c2):
            return torch.norm(x[:, c1, :, :] - x[:, c2, :, :], p=2)

        merged_channels = []
        channel_indices = list(range(x.size(1)))
        while len(channel_indices) > original_channels:
            # Find the two most similar channels
            min_distance = float('inf')
            min_pair = None
            for i in range(len(channel_indices)):
                for j in range(i + 1, len(channel_indices)):
                    dist = calculate_distance(channel_indices[i], channel_indices[j])
                    if dist < min_distance:
                        min_distance = dist
                        min_pair = (i, j)

            c1, c2 = min_pair
```

```
            new_channel=(x[:,channel_indices[c1],:,:] + x[:, channel_indices[c2],:,:])/2
            merged_channels.append(new_channel)

            # Remove merged channels and add the new one
            channel_indices.pop(max(c1, c2))
            channel_indices.pop(min(c1, c2))
            channel_indices.append(len(merged_channels) - 1)

        merged_x = torch.stack(merged_channels, dim=1)
        return merged_x
```

- Usage

```
# Example Usage
threshold = 0.1
original_channels = 64
x = torch.randn(1, 64, 32, 32)  # Example input tensor

disassembler = ChannelDisassembly(threshold)
disassembled_x, outlier_indices = disassembler(x)

assembler = ChannelAssembly()
assembled_x = assembler(disassembled_x, original_channels)

print("Original tensor shape:", x.shape)
print("Disassembled tensor shape:", disassembled_x.shape)
print("Assembled tensor shape:", assembled_x.shape)
```

To recap – this implementation allows handling outliers by disassembling outlier channels into smaller sub-channels, making the activations more quantization-friendly without altering the layer output. The channel assembly step merges similar channels to reduce the overall number of channels, maintaining computational efficiency.

However, it has some drawbacks – merging similar channels to reduce the channel count can lead to some loss of information, potentially affecting model accuracy, and deciding which channels to merge requires additional calculations, adding to the overall complexity.

Next, we will review CLE integration possibility and roadblocks.

Our contributions consist of the following steps:

1. Analyze CLE and alternatives.
2. Compare different normalization methods for neural networks.
3. Review a potential GPTQ roadblock relating to CUDA kernel.
4. Analyze / validate the results.

3.2.2. OBQ & BatchNorm

OBQ authors also mention handling of layer weights outliers – they say that if outliers are quantized last (selection rule from Equation 4 OBQ selection and update rule postpones quantization of this outlier till the end), it can lead to even larger errors because there are fewer weights left to adjust and compensate (meaning error is dispersed to a smaller amount of leftover weights in update rule from Equation 4 OBQ selection and update rule).

To solve this, the authors suggest quantizing outliers as soon as they appear, rather than waiting. This simple trick helps maintain accuracy and effectiveness in layer-wise quantization.

BatchNorm – Batch normalization is a technique used in neural networks to make training faster and more stable. It works by normalizing the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. This helps keep the activations in a consistent range, making the network more robust and preventing issues like exploding or vanishing gradients.

Cross layer equalization (CLE) adjusts the weights of different layers to similar ranges before quantization. While BatchNorm focuses on normalizing activations within a single layer, CLE normalizes weights across different layers. This means both techniques aim to stabilize the network, but BatchNorm does it during training for activations, and CLE does it before quantization for weights.

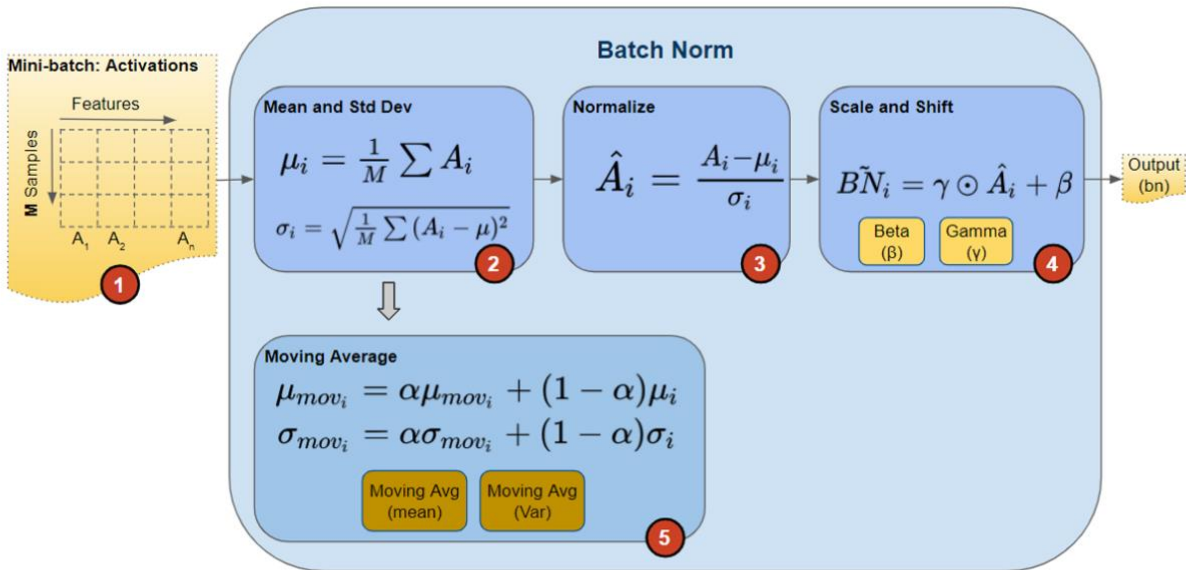

Figure 0.10 BatchNorm – shorturl.at/VBIDp

3.2.3. Bias Correction

Bias correction – another common issue is that quantization error is often biased. This means that the expected output of the original and quantized layer or network is shifted. Bias can be further amplified after cross-layer equalization as (Nagel et al, 2021) [2] have demonstrated.

Bias correction can be empirical and analytical:

- Empirical – with calibration dataset. Achieved by comparing the activations of the quantized and full precision model.

$$\Delta W E[x] = E\,[\widehat{W}x]\; - E[Wx]$$

- Analytical – with calibration dataset, use the BatchNorm γ and β (mean and standard deviation) of the previous layer output to compute the expected input distribution E [x]. We can apply clipped normal distribution if we assume input values lay in normal distribution, the effect of ReLU on the distribution can be modeled using the.

$$\gamma\mathcal{N}\left(\frac{-\beta}{\gamma}\right) + \beta\left[1 - \Phi\left(\frac{-\beta}{\gamma}\right)\right]$$

In our work we will be using empirical bias correction, as GPTQ already calibrate their parameters on the calibration data. We believe that this yields more accurate results, as well as brings more agility without dependence on presence of BatchNorm layers.

3.2.4. GPTQ CUDA kernel

GPTQ considerations. Authors of GPTQ implement custom CUDA kernels to optimize for their block-wise lazy batching 27: *"To ensure that the entire compression procedure can be performed with significantly less GPU memory . . . 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."* (Frantar et al 2023) [3].

When performing cross-layer equalization (CLE) alongside quantization, it's important to consider how memory will be managed. The compression process aims to reduce GPU memory usage by loading and processing one Transformer block, consisting of *six layers*, at a time. Once a block is quantized, its outputs are used as inputs for the next block, meaning the quantization process uses inputs from the already partially quantized model.

"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."

Integrating CLE into this process can be tricky because CLE involves adjusting weights across layers to balance their scales, which might require reloading or recalculating layers multiple times. This could increase memory usage and computational overhead if not managed carefully.

Working with CUDA kernels is out of scope of our work, but below is a potential implementation pseudocode for future work.

```
for block in transformer_blocks:
  load_block_into_memory(block)

  apply_cross_layer_equalization(block)

  hessians = accumulate_layer_hessians(block)
  for layer in block:
```

```
    quantize_layer_with_hessian(layer, hessians[layer])
current_block_inputs=process_block_with_quantized_layers(block, current_block_inputs)
release_block_from_memory(block)
```

Our contribution consists of the following steps:

1. Integrate cross-layer equalization to reduce effects of wide quantization ranges between layers, which consume precision.

```
@staticmethod
def cross_layer_equalization(layers):
    for i in range(len(layers) - 1):
        max_weight = torch.max(torch.abs(layers[i].weight))
        next_max_weight = torch.max(torch.abs(layers[i + 1].weight))
        scale = torch.sqrt(max_weight / next_max_weight)

        layers[i].weight.data /= scale
        layers[i + 1].weight.data *= scale
    return layers
```

3.3. HESSIAN EIGENVALUES IN QUANTIZATION

3.3.1. GPTVQ

As a core GPTQ framework uses **uniform asymmetric weights quantization** with **inverse Hessian as quantization sensitivity** areas of improvement may be:

1. *non-uniformity* – replace uniform quantization with non-uniform, thus represent unquantized data more accurately. (Mart van Baalen et. al. 2024) [3] did exactly that with GPTVQ, a framework that integrates vector quantization for the latent representations of the data, by integrating codebooks with centroids. Centroid in 1D can be thought of as integer to which FP16 is being rounded in RTN, e.g., for both 2.1 and 1.8, centroid is 2. They expand this idea to multidimensional approach.

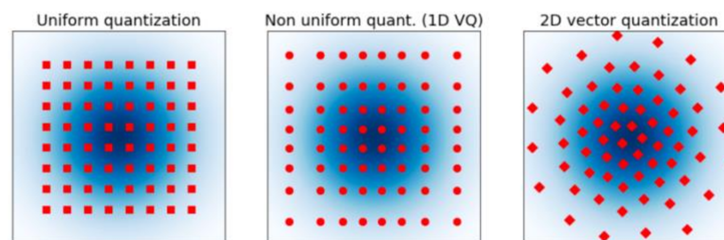

Figure 0.11 2D normally distributed data representation - *(Mart van Baalen et. al. 2024) [3]*

Algorithm 1 GPTVQ: Quantize $\mathbf{W} \in \mathbb{R}^{r \times c}$ given the inverse Hessian $\mathbf{H}^{-1}$, the block size B, VQ dimensionality d, the number of centroids k, and the group size l

1: $N_b \leftarrow \frac{c}{B}$ {the number of blocks}
2: $m \leftarrow \frac{l}{r}$ {the number of columns in a group}
3: $\mathbf{Q} \leftarrow \mathbf{0}_{r,c}$
4: $\mathbf{E} \leftarrow \mathbf{0}_{r,c}$
5: $N_g \leftarrow \frac{rc}{l}$ {the number of groups/codebooks}
6: $\mathbf{C}_i \leftarrow \mathbf{0}_{d,k}, i = 1, \ldots, N_g$
7: $\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^T$
8: **for** $i = 0, B, 2B, \ldots, N_B B$ **do**
9: **if** i % m = 0 **then**
10: $g \leftarrow \frac{i}{m}$ {the group index}
11: $\mathbf{C}_g \leftarrow \text{init_codebook}\left[\mathbf{W}_{:,i:i+m-1} \oslash \mathbf{S}_{:,i:i+m-1}\right]$
12: **end if**
13: **for** $j = 0, d, 2d, \ldots, B$ **do**
14: $P = i + j, \ldots, i + d - 1$
15: $\mathbf{Q}_{:,P} \leftarrow \mathbf{S}_{:,P} \odot \text{VQ_quant}\left[\mathbf{W}_{:,P} \oslash \mathbf{S}_{:,P}, \mathbf{C}_g\right]$
16: $\mathbf{E}_{:,P} \leftarrow (\mathbf{W}_{:,P} - \mathbf{Q}_{:,P})\,[\mathbf{H}^{-1}]_P$
17: $\mathbf{W}_{:,i+d-1:(i+B)} \leftarrow \mathbf{W}_{:,i+d-1:(i+B)} - \sum_{p=0}^{d-1} \mathbf{E}_{:,i+j+p}[\mathbf{H}^{-1}]_{p,i+d-1:(i+B)}$
18: **end for**
19: $\mathbf{W}_{:,(i+B)} \leftarrow \mathbf{W}_{:,(i+B)} - \mathbf{E} \cdot [\mathbf{H}^{-1}]_{i:(i+B),(i+B):}$
20: **end for**

Figure 0.12 GPTVQ - (Mart van Baalen et. al. 2024) [3]

2. *Hessian for asymmetric quantization* – asymmetric means that it has *zero point,* and that min-max range of output (quantized) distribution is calibrated with data. In case of GPTQ they "*Our entire GPTQ calibration data consists of . . . the C4 dataset, 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 actually "zero-shot"*".
Hessian (inverse) is used as an error coefficient that helps adjust error for sensitivity.

$\mathbf{Q} \leftarrow \mathbf{0}_{d_{\text{row}} \times d_{\text{col}}}$ *// quantized output*
$\mathbf{E} \leftarrow \mathbf{0}_{d_{\text{row}} \times B}$ *// block quantization errors*
$\mathbf{H}^{-1} \leftarrow \text{Cholesky}(\mathbf{H}^{-1})^{\top}$ *// Hessian inverse information*
for $i = 0; B; 2B; : : :$ **do**
  for $j = i; : : : ; i + B - 1$ **do**
    $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ *// quantize column*
    $\mathbf{E}_{:,j-i} \leftarrow (\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}) = [\mathbf{H}^{-1}]_{jj}$ ← 1 *// quantization error*
    $\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(i+B)}$ ← 2 *// update weights in block*
  end for
  $\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):}$ ← 3 *// update all remaining weights*
end for

Figure 0.13 Hessian role in GPTQ algorithm

3.3.2. Eigenvalue calculation

Eigenvalues are numbers that show how much a transformation changes when applied to a vector. Like stretching or shrinking basis without rotating it; the eigenvalue tells us how much each direction is stretched or shrunk.

Why does it matter for us? While Hessian tells us sensitivity of the parameters, and its Inverse – in a way tell us how much parameters should be changed, parameters with larger eigenvalues are more critical, and thus can be quantized less aggressively. We must consider different methods for Eigenvalue derivation:

- In *diagonal* matrices, the eigenvalues are numbers on the diagonal.
- In *symmetric* matrices, the eigenvalues are always real numbers.
- In *other* matrices – eigenvalues can be complex numbers, and hard to calculate.

Initially we were considering Eigenvalues for diagonal Hessian, which would incur negligible computational overhead. During our experiments, however, we discovered the caveats below, even though the Hessian is symmetric – computed as $H = XX^T$

Using Eigenvalues from Hessian matrix, however, has some caveats:

1. Hessian is non-diagonal (before Cholesky decomposition), which means that eigenvalue require additional computation, unlike diagonal matrix, which non-zero elements are eigenvalues themselves.
2. GPTQ considers quantization parameters (maxq, zero, scale) calibration using RTN (round to nearest), which requires no input, and MSE (mean squared error), which requires input data for calibration.

Our contribution consists of the following steps:

1. Integrate Hessian eigenvalues into quantization parameters (maxq, zero, scale) calibration for both weights and activations.
2. Measure the performance and speed change and accuracy/perplexity tradeoff.

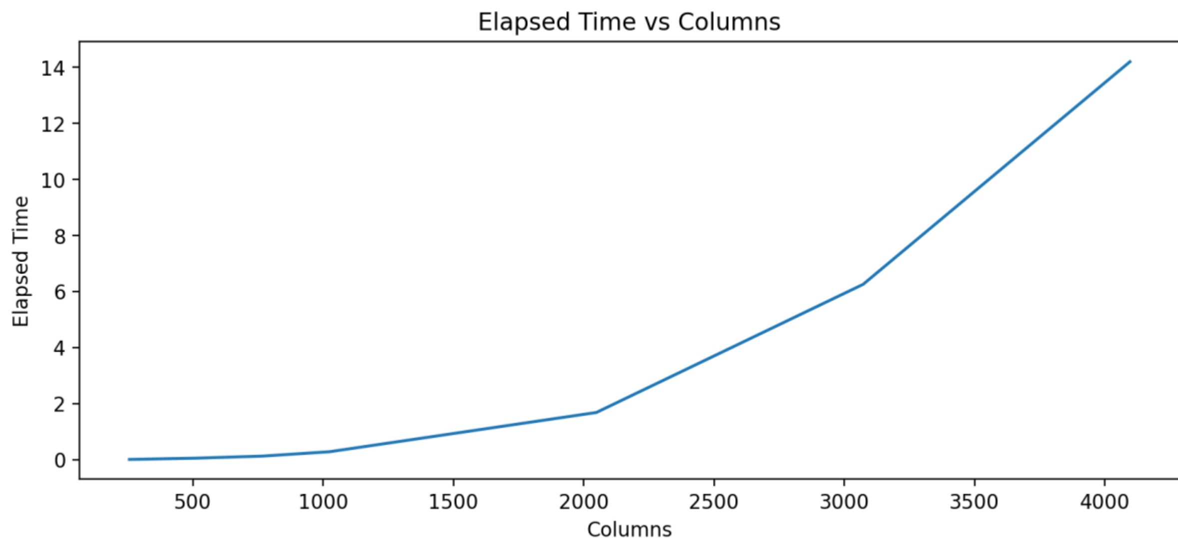

Figure 0.14 Hessian Eigenvalues computational time from size of Hessian. (Hessian approximation XX^T)

As there exist faster, better tailored methods for calculating Eigenvalues of a symmetric matrix torch.linalg.eigh(), we will include Hessian calculation in our experiments and ablation tests.

3.4. COMBINING CONTRIBUTIONS

Combining parts of this section in an efficient manner requires careful consideration, as we encountered some roadblocks for integration of all 3 components.

3.4.1. GPTAQ Algorithm

Initial algorithm for GPTAQ was as follows, however there have been some changes in it.

Algorithm 1 Quantize $\mathbf{W}$ given layers $\mathbf{L}$ and GPTQ with $\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j})$ replaced with $\mathbf{Q} \leftarrow \text{quant}(\mathbf{W}_{:,j}, \mathbf{Eig})$.

for $i = 0, \ldots, L$ **do**
  $\mathbf{L}_{\mathbf{l+1}\text{ input}} \leftarrow \text{activation_quant}(\mathbf{L}_{\mathbf{l}}(X))$
  $\mathbf{H} \leftarrow \text{update_H}(\mathbf{H}, \mathbf{X})$
end for
for $i = 0, \ldots, L$ **do**
  $\mathbf{Eig} \leftarrow \text{get_Eigenvalues}(\mathbf{H})$
  $\mathbf{L}_i \leftarrow \text{GPTQ}(\mathbf{L}_i, \mathbf{Eig})$
end for
$\mathbf{L} \leftarrow \mathit{cross_layer_equalization}(\mathbf{L})$

3.4.2. Quantization debug flowchart

Throughout our work we addressed the Figure 0.15 PTQ debugging flow chart – (Nagel et al, 2021) [2], particularly in the cross layer quantization & bias absorption section.

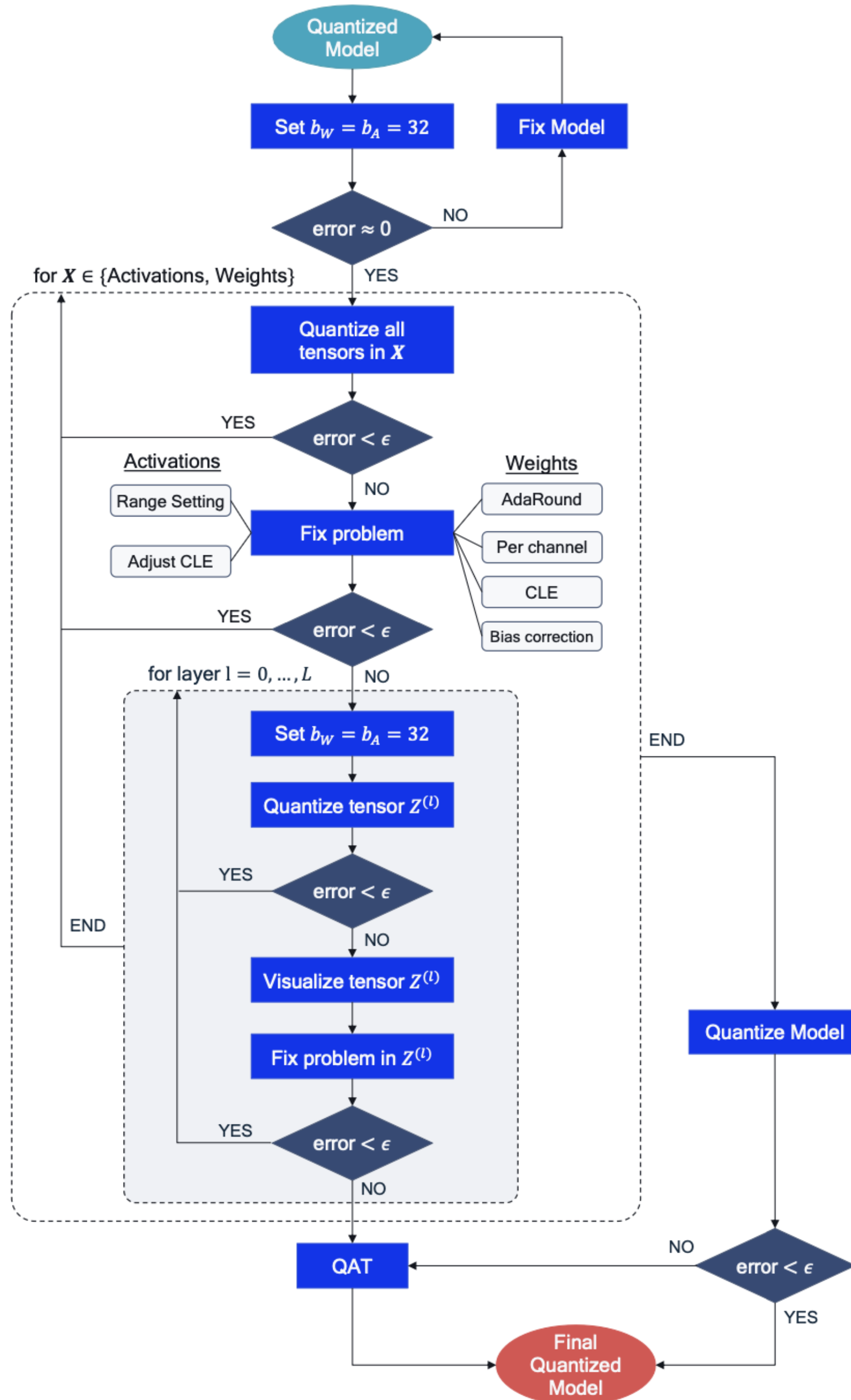

Figure 0.15 PTQ debugging flow chart – (Nagel et al, 2021) [2]

EXPERIMENTS

4.1. OVERVIEW

The goal of these experiments is to evaluate various quantization strategies on the Facebook OPT-125M model. Quantization aims to reduce the model size and inference time while maintaining acceptable performance, measured by Perplexity (PPL). Several methods, including Weight-only Quantization (W4), Activation Quantization (W4_A4) with different methods (RTN, Token-wise, Reoptimize), and combinations of these techniques, were tested.

4.2. SETUP

The experiments were conducted on the C4 dataset, processing 256 sequences per batch. The quantization process and subsequent evaluation were performed using a GPU environment. The primary metric for evaluating model performance is Perplexity (PPL). Ground truth PPL is 13,775.

4.3. BASELINES

We used GPTQ in FP16 and W4 precision as our baselines. Each of the three methods—activation quantization, cross-layer equalization with bias correction, and Hessian eigenvalues integration—was tested separately and then in combination to evaluate their individual and collective impact on model performance. The baseline performance is established using the original model without any quantization applied. In our case as we only worked with a part of C4 dataset – perplexity baseline is much higher than in original GPTQ paper.

Model	PPL (Baseline)	Relative Increase (%)
Original (no quant)	13, 775	0

4.4. EXPERIMENTS

We conducted a series of experiments to assess the effectiveness of each quantization enhancement. The results were compared against the baseline models to measure improvements in quantization efficiency and model accuracy. Detailed results of these experiments are presented in Section 4.5.

Weight-only Quantization (W4)

Quantization was applied to the model weights only. The results indicated an increase in PPL relative to the ground truth.

Model	PPL	Relative Increase (%)
W4	16125.56	17.04

Activation Quantization (W4_A4)

Various methods of activation quantization were evaluated, including RTN, Token-wise, and Reoptimize.

Model	PPL	Relative Increase (%)
W4_A4_RTN	14716.40	6.82
W4_A4_Token	14718.67	6.84
W4_A4_Reoptimize	14719.12	6.85

Eigenvalue Quantization (W4_A4_EIG)

This method involves using eigenvalues during the quantization process.

Model	PPL	Relative Increase (%)
W4_A4_EIG	21104.61	53.23

4.5. ABLATION STUDY

We take GPTQ FP16 and W4 as baselines and test each of 3 methods separately, and combine them. We performed an ablation study to isolate the contributions of each enhancement. The models were benchmarked using perplexity (PPL) on the C4 and PTB datasets. The following table summarizes the perplexity scores for different bit precisions: PPL – benchmark

RTN with CLE

Combining RTN activation quantization with weight quantization.

Model	Relative Increase (%)
W4_A4_RTN_CLE	1.95

RTN with Eigenvalues

Combining RTN activation quantization with eigenvalue-based quantization.

Model	Relative Increase (%)
W4_A4_RTN_EIG	53.22

OPT	Bits	125M
Full	16	38.99
RTN	4	53.89
GPTQ	4	45.17
GPTAQ	**4**	

The ablation study demonstrates the impact of each quantization method on the overall performance, highlighting the effectiveness of our proposed enhancements. Further details and analysis of these results are discussed in the following sections.

4.6. RESULTS

The experiments demonstrated that while all quantization methods led to an increase in PPL, certain methods such as RTN and Token-wise activation quantization had a relatively moderate impact on model performance. Eigenvalue-based quantization showed a significant increase in PPL, suggesting it may not be as effective for maintaining model performance.

Overall, the most effective method with the least increase in PPL was the combination of RTN and CLE, which had a PPL of 14,043.14, only a 1.95% increase compared to the baseline.

Model	PPL	Relative Increase (%)
Original (Baseline)	13775	0
W4	16125.56	17.04
W4_A4_RTN	14716.40	6.82
W4_A4_Token	14718.67	6.84
W4_A4_Reoptimize	14719.12	6.85
W4_A4_EIG	21104.61	53.23
W4_A4_RTN_CLE	14043.14	1.95
W4_A4_RTN_EIG	21103.30	53.22

Some methods also increase inference speed, while others affect it in a negative way. The comparison shows that most quantization methods result in very slight changes in inference speed compared to the baseline, with most methods having a median speed within a few percentage points of the baseline. However, the W4_A4_EIG method significantly increased the inference time. This suggests that while quantization methods can be effective, care must be taken to balance performance improvements with potential increases in computation time, particularly when using more complex methods such as Hessian eigenvalues integration.

Method	Inference Speed (s)	Percentage of Baseline (%)
W4_EIG_CUDA	0.011	100.0
W4_A4_RTN	0.012	100.924
W4_A4_TOKEN	0.011	98.239
W4_A4_REOPTIMIZE	0.011	100.656
W4_A4_CLE	0.011	100.675
W4_A4_EIG	0.017	151.416
W4_A4_RTN_CLE	0.012	101.014
W4_A4_RTN_EIG	0.012	102.261

The quantization methods that involve activation quantization, particularly the RTN and Token-wise methods, show promise for reducing model size and inference time with minimal impact on performance.

The results of our experiments are summarized below. We evaluated the performance improvements and efficiency gains from the proposed quantization enhancements

- *Activation Quantization* – when applied independently, activation quantization resulted in a significant reduction in model size with moderate impact on accuracy. The perplexity scores showed an increase compared to the full precision baseline but remained relatively low.
- *Cross-Layer Equalization + Bias Correction* – applying cross-layer equalization and bias correction improved the quantization efficiency by balancing the scaling factors across layers and correcting biases. This method showed good accuracy retention, with lower perplexity scores compared to the baseline quantization.
- *Hessian Eigenvalues* – integrating Hessian eigenvalues into the quantization process provided limited benefits in preserving model accuracy. This approach increased quantization errors, resulting in perplexity scores that were significantly higher.
- *Combined Methods* – the combination of activation quantization, cross-layer equalization with bias correction, and Hessian eigenvalues integration demonstrated the best overall results. The collective application of these methods resulted in the lowest perplexity scores among the quantized models, indicating a substantial reduction in quantization-induced performance degradation.

The experimental results demonstrate that our proposed methods enhance the quantization process, effectively maintaining model accuracy while significantly reducing computational and memory requirements. Further analysis and insights into these results are discussed in the subsequent sections.

Conclusions and recommendations:

- Activation quantization methods such as RTN and Token-wise quantization are recommended due to their minimal impact on accuracy and moderate perplexity increase.
- Cross-layer equalization combined with bias correction offers a balanced approach, improving efficiency and maintaining low perplexity scores.
- Integrating Hessian eigenvalues, while theoretically beneficial, resulted in higher perplexity and is not recommended as a standalone method.
- Combined methods, particularly RTN with CLE, achieved the best performance, suggesting that a holistic approach to quantization can mitigate individual weaknesses and enhance overall model efficiency and accuracy.

SUMMARY

This paper presents GPTAQ, a comprehensive framework that integrates three key enhancements into the quantization process: activation quantization, cross-layer equalization with bias correction, and the incorporation of Hessian eigenvalues. These methods collectively aim to enhance the efficiency of quantization while maintaining or even improving the accuracy of the quantized models. However, we encountered several challenges and *limitations* during the implementation of these techniques, which we will discuss in detail. We begin by providing a background on neural network compression and quantization techniques, setting the stage for the discussion on the specifics of our proposed methods. This includes an overview of different types of quantization. We also explore the practical applications of quantization in various domains and the impact it has on model performance and deployment. Following this, we review existing quantization frameworks. We delve into the specific techniques employed by GPTQ, such as Lazy Batch-Updates that minimize computational overhead, and Cholesky Reformulation for reducing Hessian inversion error.

Next, we introduce GPTAQ and its components in detail. Activation quantization focuses on reducing the precision of activations, which can significantly lower the computational requirements. Cross-layer equalization with bias correction is aimed at balancing the scaling factors across layers and correcting biases introduced during quantization. The integration of Hessian eigenvalues leverages the curvature information of the loss surface to guide the quantization process, ensuring that the most critical parameters are quantized with higher precision. We describe the algorithms and methodologies used to implement these techniques and how they synergistically improve the overall quantization process. Despite these efforts, we encountered several obstacles that affected the overall performance and efficiency, which we discuss in the results section.

In the experimental section, we outline our setup, including the models and datasets used for evaluation. We utilize the OPT-175M model and evaluate our methods on the C4 and PTB datasets. The experiments were conducted on a Google

T4 GPU. We provide a comparison with baseline models, specifically GPTQ in FP16 and W4 precision, and measure the impact of each quantization method on model performance.

The results section presents a comprehensive analysis of our findings. While activation quantization, cross-layer equalization with bias correction, and Hessian eigenvalues integration each contributed to improving quantization efficiency, we faced challenges that limited their effectiveness. Our ablation study isolates the effects of each technique, providing insights into their individual and combined benefits. The perplexity scores for various configurations illustrate the difficulties in maintaining model performance despite aggressive quantization.

While GPTAQ has demonstrated potential in improving quantization efficiency and model performance, several avenues for future research remain. One potential direction is the CUDA kernel customization for cross layer equalization. This could increase feasibility and impact of CLE. Finally, experimenting with kernel fusion (Yao et al, 2022) [7] and knowledge distillation techniques may push the boundaries of GPTAQ efficiency even further.

REFERENCES

[1] Andrey Kuzmin, Markus Nagel, Mart van Baalen, Arash Behboodi, Tijmen Blankevoort. Pruning vs Quantization: Which is Better? arXiv preprint arXiv: 2307.02973, 2024.

[2] Markus Nagel, Rana Ali Amjad, Mart van Baalen, Marios Fournarakis Yelysei Bondarenko Tijmen Blankevoort. A White Paper on Neural Network Quantization *arXiv preprint arXiv: 2106.08295, 2021.*

[3] Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. GPTQ: Accurate post-training quantization for generative pretrained transformers. *arXiv preprint arXiv:2210.17323*, 2023.

[4] 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.

[5] Mart van Baalen, Andrey Kuzmin, Markus Nagel, Peter Couperus, Cedric Bastoul, Eric Mahurin, Tijmen Blankevoort, Paul Whatmough. GPTVQ: The Blessing of Dimensionality for LLM Quantization *arXiv preprint arXiv: 2402.15319, 2024.*

[6] Zhewei Yao∗, Xiaoxia Wu∗, Cheng Li, Stephen Youn, Yuxiong He ZeroQuant-V2: Exploring Post-training Quantization in LLMs from Comprehensive Study to Low Rank Compensation *arXiv preprint arXiv: 2303.08302, 2023*

[7] 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*.

[8] Ashish Vaswani, Llion Jones, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Aidan N. Gomez, Łukasz Kaiser: Attention Is All You Need *arXiv preprint arXiv: 1706.03762, 2017*

[9] Jing Liu, Ruihao Gong, Xiuying Wei, Zhiwei Dong, Jianfei Cai, Bohan Zhuang: QLLM: ACCURATE AND EFFICIENT LOW-BITWIDTH QUANTIZATION FOR LARGE LANGUAGE MODELS *arXiv preprint arXiv: 2310.08041, 2032*