TÉCNICO LISBOA

A Study on Non-Linear Functions for Quantization of Machine Learning Models

Diogo Emanuel da Costa Venâncio

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Nuno P. Lopes

Examination Committee

Chairperson: Prof. Pedro T. Monteiro
Supervisor: Prof. Nuno P. Lopes
Member of the Committee: Prof. David Martins de Matos

October 2024

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First, I would like to express my deepest gratitude to my family. To my mother, Paula, and my brother, Martim – thank you for always believing in me, for your endless support, and for being my rock through every step of this journey. To my girlfriend, Sofia, thank you for your patience, your constant care, and for standing by my side during the highs and the lows. Your encouragement and love have been my foundation, and I would not have made it this far without you.

I would like to sincerely thank my supervisor, Prof. Nuno Lopes, for his invaluable guidance and continuous support throughout this thesis. Your mentorship has not only shaped this work but also contributed to my growth as a student and, most importantly, as a person. I am also grateful to Prof. João Pimentel Nunes for the valuable discussion that helped bring a fresh perspectives to parts of this work.

I extend my thanks to Fundação para a Ciência e a Tecnologia (FCT) and to Minho Advanced Computing Center (MACC) for providing access to a HPC cluster, which was essential for the success of this research.

Finally, a heartfelt thank you to my friends and colleagues for your companionship and encouragement along the way. Sharing this journey with you has made all the difference.

To each and every one of you – Thank you.

Abstract

Quantization is one of the leading techniques to reduce the memory usage of machine learning models. It works by approximating the weights of a model by some function with a smaller domain (e.g., replace 32-bit floats with 8-bit integers that are coefficients in some function that maps back to 32-bit floats). Although most quantization methods approximate weights with a linear or affine function, the weights of current machine learning models often exhibit non-linear behavior at the extremities. Moreover, some studies suggest that the extremities are important for the end-to-end accuracy. In this thesis, we introduce Post-Training Non-Linear Quantization (PTNQ), a post-training quantization framework designed to efficiently compress machine learning models by following a pipeline whose focus is to search for the best non-linear quantization function from a pool while featuring many different techniques to better tune the non-linear transformations. We show that PTNQ provides significant advantages over affine functions, achieving similar accuracy while requiring 2 to 4 fewer bits per coefficient.

Keywords

Machine Learning; Quantization

Resumo

Quantização é uma das principais técnicas para reduzir a utilização de memória dos modelos de aprendizagem automática. Esta técnica aproxima os pesos de um modelo utilizando uma função com um domínio mais pequeno (por exemplo, substituir floats de 32 bits por inteiros de 8 bits, que servem como coeficientes numa função capaz de reconverter em floats de 32 bits). Embora a maioria dos métodos de quantização utilizar funções lineares ou afins para essa aproximação, os pesos dos modelos atuais de aprendizagem automática tendem a comportar-se de forma não-linear nas extremidades. Além disso, vários estudos sugerem que essas extremidades são cruciais para manter a precisão global dos modelos. Nesta dissertação, apresentamos o Post-Training Non-Linear Quantization (PTNQ), uma framework de quantização pós-treino que permite comprimir modelos de aprendizagem automática de forma eficiente, utilizando um processo de procura que identifica a melhor função de quantização não-linear de um dado conjunto e incorporando diversas técnicas para afinar as transformações não-lineares. Demonstramos que o PTNQ oferece vantagens significativas em relação às funções afins, atingindo uma precisão semelhante com menos 2 a 4 bits por coeficiente.

Palavras Chave

Aprendizagem Automática; Quantização

Contents

List of Figures

List of Tables

Listings

Acronyms

DQ	Dynamic Quantization
KV	Key-Value
LR	Learning Rate
LUT	Lookup Table
MACC	Minho Advanced Computing Center
MSE	Mean Squared Error
NLR	Non-Linear Regression
PTNQ	Post-Training Non-Linear Quantization
PTSQ	Post-Training Static Quantization
QAT	Quantization-Aware Training
QM	Quantization Method
QP	Quantization Parameter
WOQ	Weight-Only Quantization
WER	Word Error Rate

1

Introduction

Contents

Over the past decade, machine learning models, especially deep neural networks, have demonstrated exceptional capabilities across various domains. In natural language processing, models like Llama3 [Dubey and et al., 2024] and BERT [Devlin et al., 2019] have become foundational tools in various contexts. Llama3 excels at tasks such as conversational AI, making it ideal for applications ranging from customer service automation to content generation [Chaudhary et al., 2024, Cabezas et al., 2024]. BERT, and its family of models, is widely utilized for sentiment analysis, where understanding the nuanced contextual meaning of words is important [Batra et al., 2021].

In the field of computer vision, models like YOLO [Redmon et al., 2016] and Stable Diffusion [Rombach et al., 2021] have made significant impacts. YOLO enables real-time object detection, which is essential in various applications, including surveillance [Nguyen et al., 2021] and autonomous robotics [Azevedo and Santos, 2022]. Stable Diffusion supports photorealistic image generation from text prompts, proving beneficial in creative industries for design and content creation [Sultan et al., 2024].

With respect to autonomous systems, AlphaFold [Abramson et al., 2024] has revolutionized bioinfor-

matics by accurately predicting protein structures, thus accelerating drug discovery [Ren et al., 2023]. Additionally, reinforcement learning models are also employed in robotic-assisted surgeries, where they enhance precision and reduce human error, leading to better patient outcomes [Qian and Ren, 2023].

Moreover, multimodal models like CLIP [Radford et al., 2021] are pushing the boundaries of what machine learning can achieve. CLIP analyzes both text and images, enabling it to generate descriptive captions or answer questions about image content. This capability is particularly impactful in medical diagnostics, where integrating visual and textual information allows for more comprehensive assessments [Acosta et al., 2022].

These deep neural networks are a specialized class of machine learning algorithms composed of multiple layers that process and analyze data. Fundamentally, they function as data-flow graphs, where inputs traverse through interconnected layers of artificial neurons, each applying mathematical functions to transform the data. This layered architecture enables the models to recognize complex patterns and make predictions or decisions without requiring explicit programming to do so.

In large part, after the conception of transformers [Vaswani et al., 2017], models have become increasingly more accurate and have gained a greater presence in our daily lives. This evolution has also led to a substantial increase in their size. This trend is clearly reflected in Figure 1.1, which highlights the rapid growth of model sizes over the past six years. For instance, the BERT model, released in 2018, required 512MB of memory, whereas the more recent PaLM 540B model [Chowdhery et al., 2024] requires a minimum of 1.26TB — an increase of over 2500x.

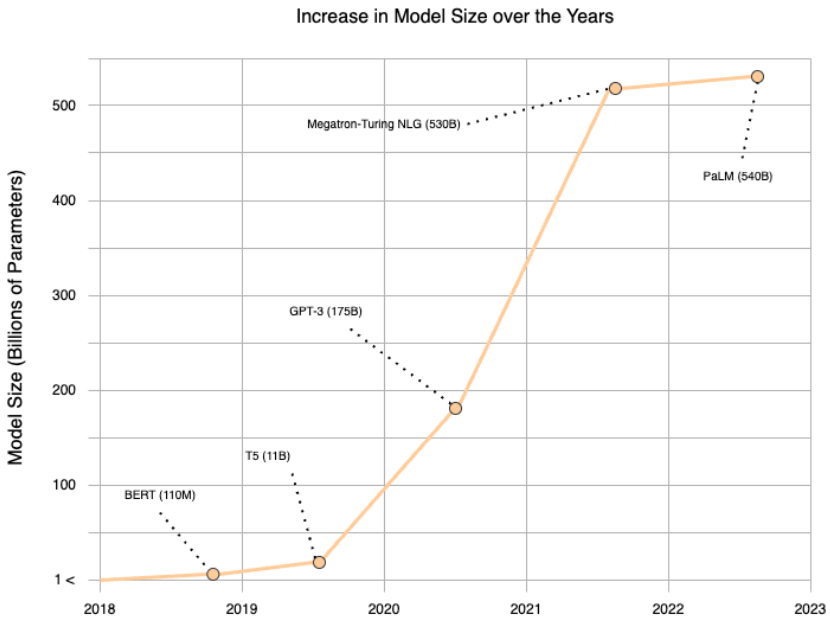


Figure 1.1: Increase in model size over the years. We can observe an exponential increase in the number of model parameters.

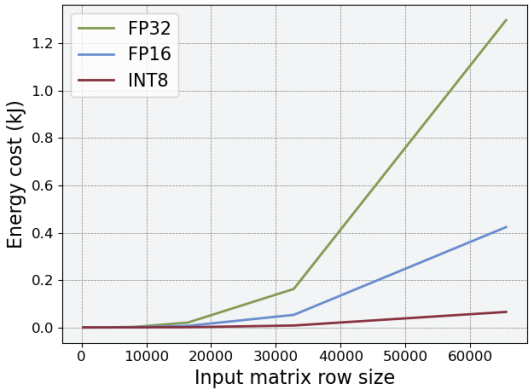


Figure 1.2: Cost of a square matrix multiplication (including memory transfers) in different data formats and matrix sizes.

The growth in model size is not just beneficial but essential for unlocking higher performance. Research has shown that scaling up models often leads to substantial improvements in accuracy [Wei et al., 2022]. More intriguingly, these gains are not linear; models exhibit near-random behavior until they surpass a critical size threshold, after which performance increases dramatically [Wei et al., 2022]. This suggests that expanding the number of parameters is necessary to fully harness the potential of a model and to achieve breakthroughs that smaller versions simply cannot reach [Kaplan et al., 2020].

However, this necessary growth in model scale presents several challenges. The increase in memory requirements creates a bottleneck due to model size far outpacing improvements in memory bandwidth, latency and capacity. As more parameters are added, most operations end up waiting for data transfers due to the memory wall bottleneck. To put the scale of this bottleneck in context, while computing throughput has grown 60,000x over the past 20 years, DRAM improved 100x and interconnect bandwidth about 30x.

To provide real-time answers, models need to be loaded entirely into and executed on accelerators, such as GPUs, instead of CPUs. GPUs have thousands of simple cores optimized for the massively parallel computation required by neural networks, while most CPUs only have a few cores.

At the time of writing, the largest GPU memory capacity widely available at data centers is 80GB (Nvidia H100) [Choquette, 2023]. This means that models on the scale of Llama3 70B need to be split across multiple GPUs thus incurring a cost in communication overhead for synchronization. As more devices are added, the penalty can get worse (depends on parallelization method). Weak scaling limits arise as communication begins to dominate computation time when parallelizing even larger models such as Megatron-Turing (530B parameters) [Smith et al., 2022].

Scaling up model sizes increases the number of operations. Llama3 (70B) has over 600 times more parameters than BERT (110M). This translates into a substantial increase in the number of floating point operations (FLOPs) performed during inference, making the model execution slower.

Larger models also require more energy. Models can scale in size along two main dimensions: number of parameters and data type bit-width. Specifically, larger models have more parameters leading to higher FLOP counts during matrix multiplications while using larger bit-width data types like 32 bits floating point (FP32) instead of 16 bits (FP16) increases the memory and energy footprint.

Due to the scaling laws of matrices, the energy costs become more salient when increasing their size and when using data formats with higher bit-widths (Figure 1.2) therefore, more compact layers with smaller data formats should be preferred to reduce costs.

For instance, executing an FP16 multiply on 45nm/0.9V hardware expends 1.1pJ [Horowitz, 2014]. A single forward pass on the smaller 8B variant of the Llama3 model using an input sequence of 128 tokens, uses 1700 GFLOPs [xiaoju ye, 2023]. This translates into 1.87 Joules.

Since models perform multiple forward passes when generating content, the per-inference energy cost compounds rapidly especially during model training where intermediate values and gradients need to be computed and cached for backpropagation.

With such constraints, models have become harder to develop and deploy, requiring specialized hardware and expertise. Researchers have responded to these scaling walls with solutions to reduce costs and improve accessibility. Techniques such as model parallelization, pruning and quantization have been proposed.

Model parallelism involves partitioning the layers of a model across multiple accelerators to enable training and inference of models too large to fit within a single device. Pruning entails selectively removing model parameters based on feature importance scores. This compresses model size, reducing memory footprint and computations, with minimal impact on model accuracy.

Quantization has emerged as one of the most widely researched techniques to tackle these resource demands, complementing model parallelism and pruning methods. It works by converting parameters and/or activations from higher to lower numerical precision data formats like 8-bit integers (INT8) that are more computationally efficient, occupy 2x less memory and require over 3x less energy than FP16 (see Figure 1.2). This reduction can also possibly allow the model to fit into a single device.

At its core, as illustrated in Figure 1.3, quantization uses a mapping function $Q(x)$ to convert higher precision data to lower bit-width formats. To revert it, the outputs can be mapped back to the original data format via a dequantization function $Q^{-1}(x)$.

Quantization techniques can be categorized into three main forms – Dynamic Quantization (DQ) quantizes the weights of a model prior to execution while, during inference, dynamically converting activations to lower precision. Post-Training Static Quantization (PTSQ) involves quantizing weights and activations after the model has been trained, but requires an additional small fine-tuning step to calibrate it. Quantization-Aware Training (QAT), on the other hand, simulates quantization during the training process, allowing the model to adapt and maintain higher accuracy once deployed in lower precision.

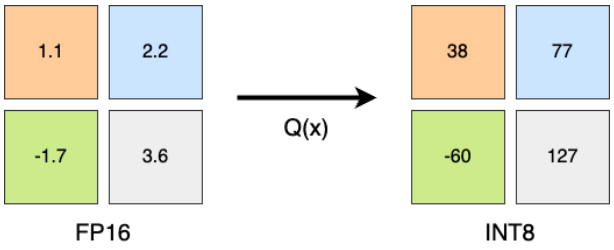


Figure 1.3: Illustration of quantization. A mapping function $Q(x)$ is applied to an FP16 tensor, converting it to INT8.

Each of these methods balances efficiency and accuracy differently, with trade-offs depending on the technique used. While quantization techniques usually increase the number of computations, with compute improving faster than memory bandwidth over the years, reducing memory transfer time helps offset the extra computations incurred, providing overall speedup for memory-bound workloads like large language models.

In recent years, quantization solutions like LLM.int8 [Dettmers et al., 2022] and SqueezeLLM [Kim et al., 2023] achieved state-of-the-art model compression ratios with little detriment to model accuracy even on extreme quantization scenarios such as 3-bits bit-widths. Most of these methods approximate a weight (or parts of it) by interpolating it through a $Q(x)$ function that corresponds to an affine/linear function.

However, this approach operates under the assumption that all weights contribute equally to the accuracy of a model. Previous work has established that outliers are crucial for maintaining end-to-end accuracy [Dettmers et al., 2022] but affine functions do not adequately capture these critical values, compromising model accuracy in low-bit representations.

1.1 Objectives

This thesis revolves around the quantization of machine learning models. Particularly, it addresses the limitations of affine transformations commonly used in quantization techniques,

Affine transformations are linear by nature and often fail to properly represent the distribution of weights (usually normal or t-student [van Baalen et al., 2023]) as they are not flexible enough to handle the inherent non-linearity found in the data. This misrepresentation becomes more pronounced in the presence of outliers, where linear functions struggle to interpolate them appropriately, degrading overall model performance significantly.

To address this issue, we propose a novel approach called Post-Training Non-Linear Quantization (PTNQ). PTNQ is designed as a post-training quantization framework that leverages non-linear functions to more accurately capture the distribution of weights in deep learning models, resulting in a more precise representation with fewer bits, setting it apart from traditional affine quantization techniques.

PTNQ incorporates a search mechanism that identifies the most suitable non-linear quantization function from a pool of candidates and features many different techniques to better tune the non-linear transformations.

By focusing on invertible and differentiable functions, PTNQ integrates seamlessly into existing training workflows, allowing standard optimization techniques to be used for fine-tuning. This also ensures that dequantization can be performed efficiently, restoring the original precision when necessary.

Figure 1.4 illustrates how PTNQ works by comparing an affine function and a non-linear function, specifically the $\mathrm{arcsinh}$ function, applied to the sorted values of a weight channel from the OPT [Zhang et al., 2022] model. The figure demonstrates that non-linear functions, such as $\mathrm{arcsinh}$, interpolate the weights more effectively than affine functions, capturing the extremities (outliers) of the distribution much more accurately. This improved interpolation leads to less quantization error and, consequently, a smaller performance drop when compared to affine quantization methods.

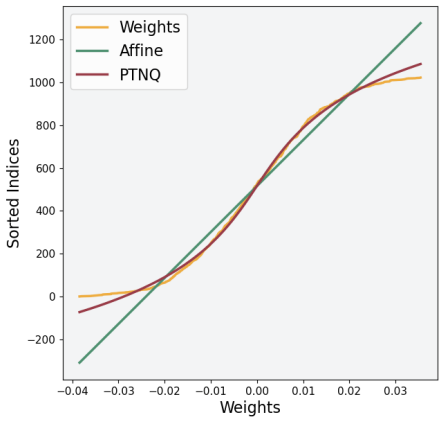


Figure 1.4: Sorted values of a channel of a weight of the OPT model. In blue, the best-fit affine function. In red, the data is interpolated with a non-linear function ($\mathrm{arcsinh}$).

The results presented in this thesis show that PTNQ outperforms traditional affine quantization by achieving similar levels of accuracy with significantly fewer bits per weight. For instance, PTNQ achieves the same accuracy as an 8-bit affine quantization scheme but with only 4 to 6 bits, representing a 25-50% reduction in weight size. This reduction in bit usage translates to more efficient model storage, making PTNQ an attractive solution for deploying deep learning models in resource-constrained environments.

1.2 Outline

This document is organized as follows. In Chapter 2 we introduce the fundamental of quantization of machine learning models. Then, in Chapter 3, we go over some of the related work in this field while covering our own proposed technique, PTNQ, in Chapter 4.

Chapter 5 describes the experimental setup utilized both during the development process of PTNQ (discussed in Chapter 6) and its final evaluation (Chapter 7). Finally, Chapter 8 concludes with a summary of findings and future work recommendations.

2

Quantization of Machine Learning Models

Contents

A powerful approach to machine learning that has driven many of its recent advances is neural networks. Neural networks are data-flow graphs composed of many processing layers capable of learning representations of data with multiple levels of abstraction [LeCun et al., 2015].

These networks rely on two key components: weights, which are the parameters learned during training that help the model make predictions by determining the strength of connections between neurons [Rumelhart et al., 1986], and activations, the outputs of neurons after applying a non-linear function. Both weights and activations are stored as numerical values, often in high-precision formats (e.g., 32-bit floats).

The increase in model size over the years coupled with the use of these data types, has significantly

increased memory demands. These requirements often surpass current hardware capacities and contribute to the “memory wall” phenomenon, where processing is delayed as computations are forced to wait for data transfer from memory, leading to performance bottlenecks.

Quantization addresses this challenge by reducing the precision of numerical data types for weights and, optionally, activations, while minimizing the impact on model performance. This reduction is essential for enabling real-time responses and significantly lowering the deployment costs of large models.

In this chapter, we examine the fundamentals of quantization, exploring how it operates, its various forms, and the inherent challenges it presents. We also discuss essential related topics, such as the data types frequently utilized in quantization to provide the reader with a well-rounded understanding of this optimization strategy.

2.1 Overview on Quantization

Quantization dates back to 1948 and is used as a lossy compression technique in many fields such as signal processing to reduce storage requirements [Gray and Neuhoff, 1998], or in analog-to-digital conversion to enable digital representation of real-world analog signals [Pelgrom, 2010]. In the context of machine learning models, this often involves representing floating-point weights and activations using lower precision integers.

This has some benefits and tradeoffs that should be kept in mind. For instance, reduced memory requirements allow the use of cheaper hardware [Krishnamoorthi, 2018]. This also reduces the transfer time of data, addressing the pressing constraints posed by the “memory wall” issue. Moreover, models with lower bit-widths data formats reduce overall energy consumption.

However, lowering precision can result in some degradation of model accuracy due to loss of information. This drop is typically minor - commonly just a few percentage points even at extreme 4-bit quantization scenarios [Liu et al., 2023].

Additionally, quantization and dequantization operations performed on the inputs and outputs of layers constitute extra computations not present in full precision models. Despite this additional computations, quantized models still yield faster overall execution due to the reduction in memory transfer time and the fact that chips process smaller data formats faster [Choquette, 2023].

Overall, a small reduction in model precision unlocks many improvements in important hardware metrics like speed, memory, and power efficiency. For most applications, this tradeoff proves highly favorable, evidenced by the rapid adoption of quantization by the industry [Krishnamoorthi, 2018].

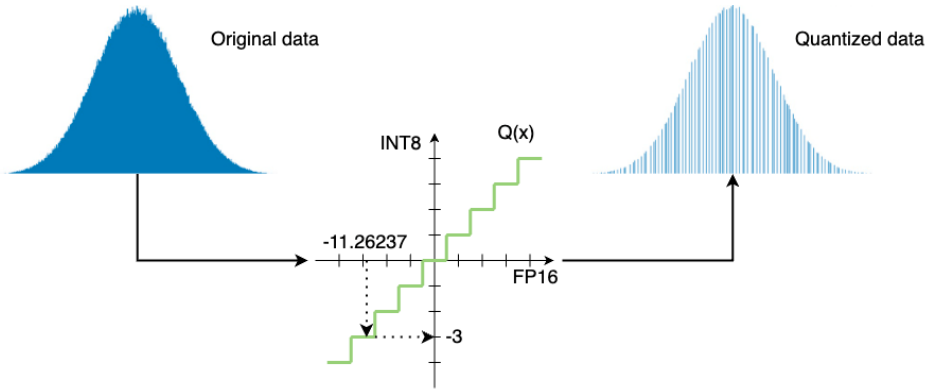


Figure 2.1: Discretization of continuous data by applying some convertion function $Q(x)$.

2.1.1 Quantization: discretization of continuous data

From first principles, quantization is a process where continuous data is approximated by a finite set of discrete values through some function (see Figure 2.1) at the cost of a slight information loss.

The most straightforward quantization method is a naive uniform quantization that simply casts the floating point values to the nearest integer, truncating the fractional values. However, this approach significantly alters the underlying distribution of the floating point values and introduces large quantization errors.

To preserve the distribution of the data and maintain model performance after quantization, a more involved approach is required. Standard quantization methods typically follow a three-stage pipeline that ensures the transformation of data into a lower-precision format while reducing quantization noise.

2.1.1.A Calibration

Calibration consists of transforming the initial distribution of the data into another distribution that is more amenable to quantization. This step ensures that the resulting values span across the lower-precision format range, minimizing the information loss.

In most quantization approaches, this transformation is performed using an affine/linear function $Q(x)$, which rescales and shifts the original data x to fit within the quantized range, as follows:

$$Q(x) = \left\lfloor \frac{x}{S} \right\rceil + ZP \tag{2.1}$$

Where S and ZP are quantization parameters that tune the transformation. The scale S normalizes the range of values, while ZP, the zero-point, aligns the zero value in the floating-point domain with the zero in the integer domain, ensuring that important values like zero are preserved exactly during the quantization process.

To determine the optimal S and ZP, different calibration techniques can be employed, including

entropy minimization via KL divergence, mean squared error minimization or percentile thresholds [Wu et al., 2020a]. The predominant technique simply inspects the original data x, and records the minimum (α) and maximum (β) values found before subsequently deriving the rest of the parameters. This method guarantees elimination of clipping errors at the expense of potentially higher rounding errors due to outliers. We will be going over these topics later in this section.

S is then computed as the ratio between the range of the input data $[\alpha, \beta]$ and the supported range of the output data format $[\alpha_q, \beta_q]$ (e.g., for integer of 8 bits, the range is $[-128, 127]$):

$$S = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{2.2}$$

while ZP is determined as:

$$ZP = -(\frac{\alpha}{S} - \alpha_q) \tag{2.3}$$

To better tune and adjust the calibration process, two main strategies are used to compute S and ZP. These approaches differ in their balance between efficiency and the accuracy of the quantization:

- **Per-tensor** uses a single S and ZP values across the entire tensor. Enables greater compression yet accumulates more error if distributions vary significantly between channels. It is commonly used for activations since all channels usually exhibit similar distributions.

- **Per-channel** computes distinct S and ZP for each channel within a tensor. Allows fine-grained calibration to the distribution of each channel but requires more compute and memory to store the additional parameters. It is predominantly used for quantizing weights to compensate for divergence between channels within the tensor.

2.1.1.B Rounding

Once the initial floating-point values are transformed, they need to be converted to integer values that can be represented with the appropriate data types. Rounding plays a key factor in determining how well the quantized values approximate the original data by mapping the scaled values to their integer counterparts in the quantized domain.

The most common form of rounding [Wu et al., 2020a], relies on a deterministic approach that always rounds to the nearest integer. For example, 3.7 would be rounded to 4, and 2.4 would be rounded to 2. While simple, deterministic rounding introduces a systematic error known as rounding bias that will compound across many layers of a deep neural network, potentially reducing overall model accuracy.

To address this problem, researchers developed a technique called stochastic rounding [Croci et al., 2022] that probabilistically rounded up or down based on the fractional part of the number. For instance,

if the value is 3.7, it would have a 70% chance of being rounded to 4 and a 30% chance of being rounded to 3.

This strategy helps reduce rounding bias over large datasets and has been shown to maintain the accuracy of models under aggressive quantization schemes.

2.1.1.C Clipping

Clipping is the final stage of quantization and is responsible for handling values that fall outside the range representable by the quantized domain. When a value exceeds the upper or lower bound of the quantized range, it is constrained to the maximum or minimum representable value, respectively. For instance, in an 8-bit quantization with values limited to the range $[-128, 127]$, any value exceeding 127 or below -128 is converted to the boundary values.

Clipping can result in significant loss of information, especially when large parts of the data fall outside the representable bit-width range or if the clipping range encompasses values that are never present in the input data [Wu et al., 2020a]. This can introduce distortion, particularly in models with outliers or high-variance data. To address this concern, we may choose between two distinct quantization schemes:

- The **asymmetric** (also called affine) scheme assigns the input range to the min and max observed values allowing asymmetric mapping around a distribution mean. Tipically used when quantizing non-negative activations.

- The **symmetric** scheme, centers the input range around 0, eliminating the need to calculate a zero point offset ZP. Commonly employed when quantizing weights due to, predominantly, following symmetric normal distributions.

Figure 2.2 illustrates the impact of these different schemes on activations and weights. As shown, applying symmetric quantization to a non-negative activation can introduce new values that distort the initial representation, emphasizing the importance of selecting the correct quantization scheme to maintain model performance.

2.1.2 Dequantization

The inverse operation of quantization is dequantization, which is used to recover an approximation of the original floating-point values from the quantized integers. Dequantization applies the reverse of the affine transformation used during quantization, and can be defined as:

$$Q^{-1}(q) = (q - ZP) \times S \tag{2.4}$$

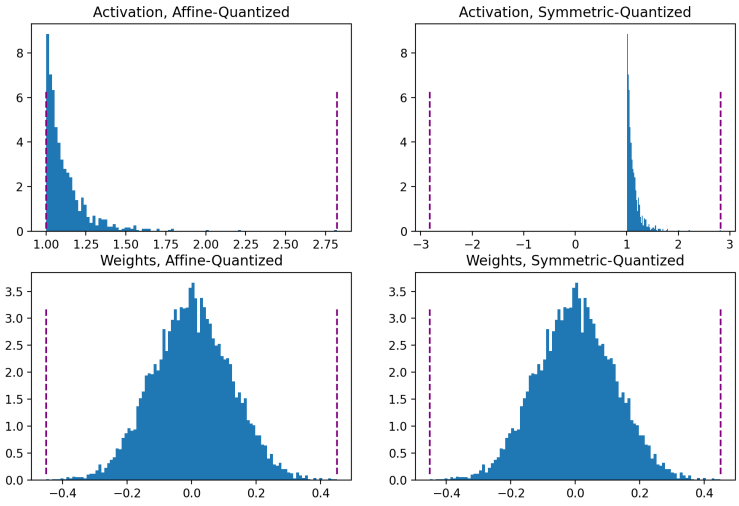


Figure 2.2: Comparison between asymmetric/affine quantization schemes (left) and symmetric quantization schemes (right). Image from https://pytorch.org/blog/quantization-in-practice/.

Here, q is the quantized integer values obtained from applying $Q(x)$ using the same S and ZP parameters. This function maps the quantized values back to the original range, but since some information is lost during quantization, the dequantized values x' are only an approximation of the original floating-point values x.

The difference between the original input x and the dequantized output x' is referred to as the quantization noise (also called quantization error), representing irrecoverable information loss that ultimately causes a slight degradation in model accuracy.

This error arises primarily from the rounding stage, where continuous values are mapped to discrete integer levels, and from clipping, where out-of-range values are truncated to the nearest representable integer.

Figure 2.3 summarizes the calibration process, showing the flow of converting an FP16 tensor into an INT8 tensor, dequantizing it, and computing the quantization noise.

2.2 Types of Quantization

Quantization techniques vary widely in their applications and the components they target offering distinct advantages in terms of model compression, computational efficiency, and memory savings.

While many approaches can be applied across different parts and life cycles of a model, such as Key-Value (KV) cache during inference [Lin* et al., 2024], and even optimizer state during training [Dettmers et al., 2021, Li et al., 2023], this section focuses on weight quantization, which remains the primary focus

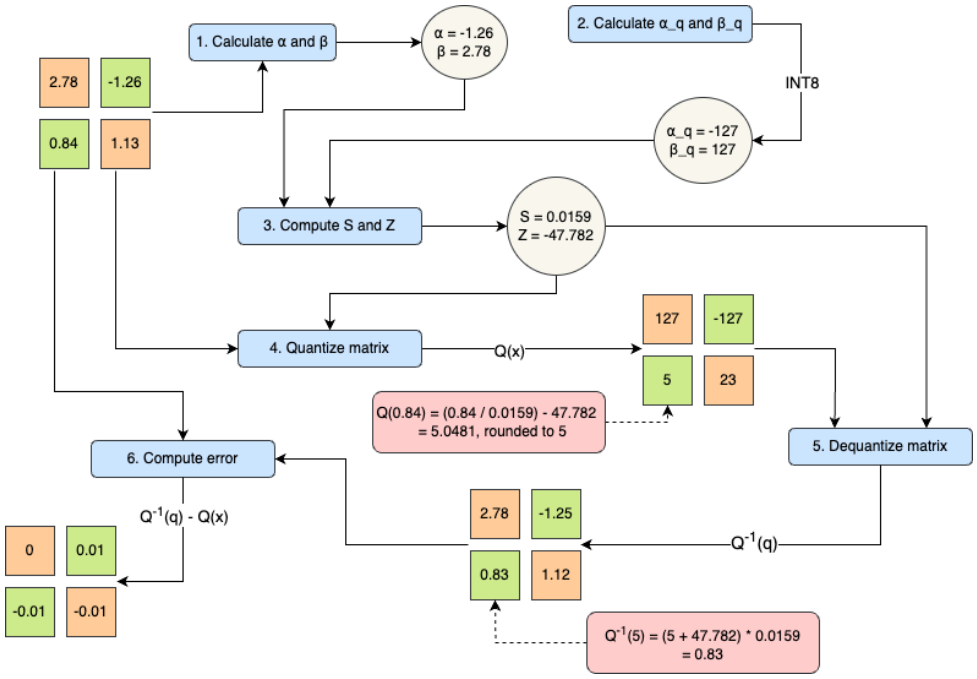


Figure 2.3: Calibration steps required to convert an FP16 tensor to INT8. Also demonstrates the use of $Q^{-1}(q)$ to dequantize it back to the original data format and computing the quantization noise.

of most quantization strategies.

2.2.1 Dynamic Quantization (DQ)

In Dynamic Quantization (DQ), the weights of the model are quantized just before being used for inference, while activations remain in full precision. Then, during inference, activations are converted to a lower precision format on the fly (details in Figure 2.4), just before doing the computation.

This approach is particularly useful in compute-bound environments where memory transfer times are not the primary bottleneck but reducing the computation cost of operations is more relevant.

Another key advantage is flexibility since the model quantization adapts across inputs. However, dynamically calibrating activations incurs in some runtime overhead, making the model slower.

On the other hand, DQ can follow a Weight-Only Quantization (WOQ) approach where the weights are dequantized just before computation instead of quantizing the activations. This is particularly useful in memory-bound workloads, where minimizing memory usage and access speeds is of higher priority than reducing computational demands.

2.2.2 Post-Training Static Quantization (PTSQ)

Post-Training Static Quantization (PTSQ) is the most commonly used quantization techniques and, unlike DQ, where quantization happens during inference, PTSQ quantizes the weights and activations after

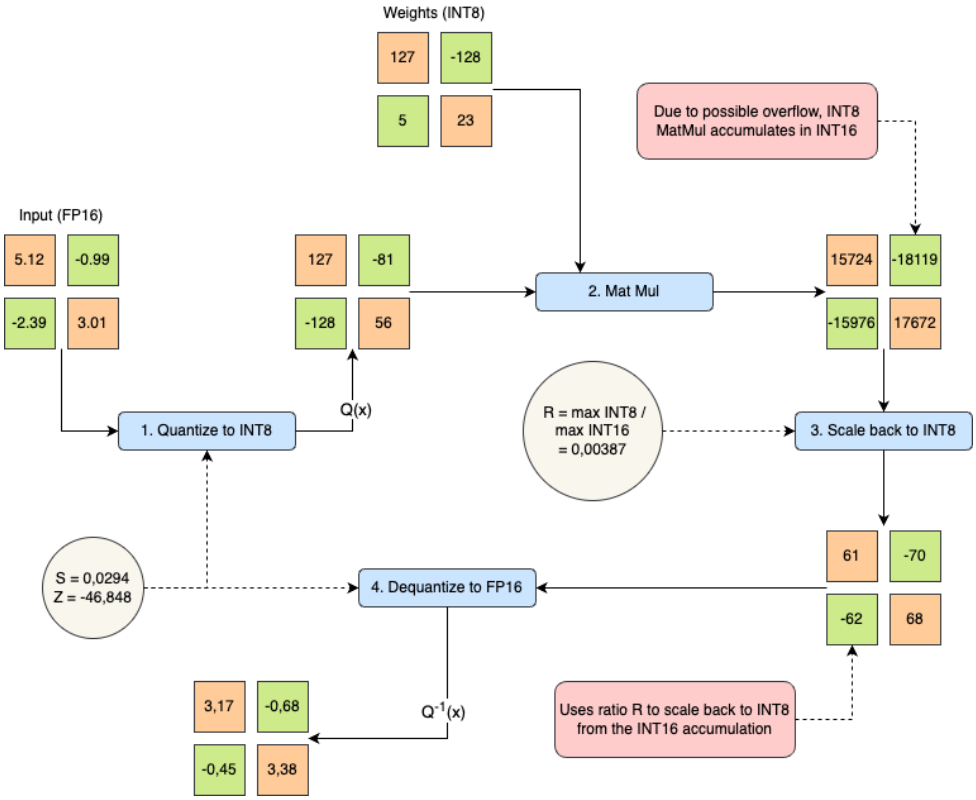


Figure 2.4: Example execution of Dynamic Quantization.

training, during a separate calibration step (see Figure 2.5).

This calibration step requires a small "fine-tune" dataset, usually a sample from the initial training data, to record metrics and distributions used to determine the optimal quantization parameters (e.g., scaling factors and zero-points) and quantize activations at inference time without needing to retrain the model.

PTSQ is generally faster than DQ, as the quantization parameters of the activations are computed offline, meaning that the quantization overhead is mostly removed from the inference process. However, this approach loses the flexibility benefits of DQ when faced with broader set of inputs.

2.2.3 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) typically results in the highest accuracy. With QAT, all weights and activations are "fake-quantized" (they are quantized and immediately dequantized to add quantization noise to the values) during training but all computations are still done in the initial format (refer to Figure 2.6).

This techniques allows the model to learn robustness to the effects of quantization, resulting in minimal accuracy loss since the quantization error is fed into the training loss calculation. This encourages

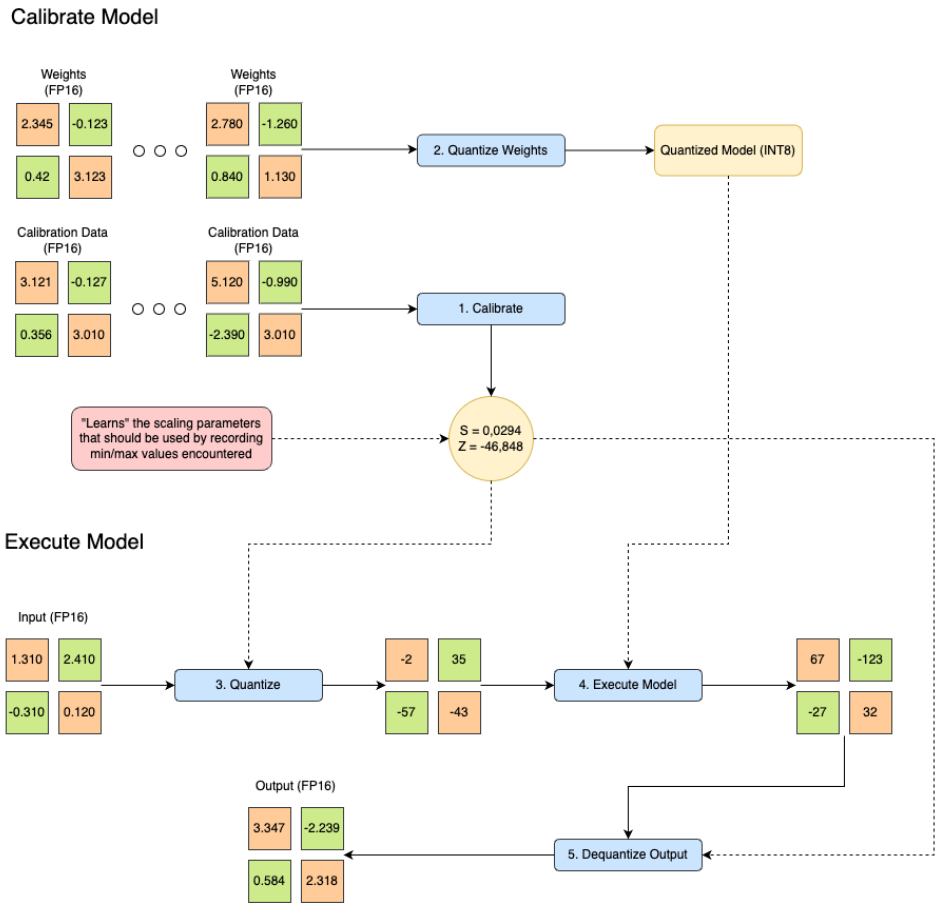


Figure 2.5: Example execution of Post-Training Static Quantization.

the model to train parameters amenable to quantization rather than relying purely on post-hoc conversion.

2.3 Limitations and Challenges

Quantizing machine learning models presents several limitations and challenges that must be addressed to achieve optimal performance. Throughout this chapter, we have discussed some of them, including rounding and clipping.

One significant challenge is that traditional quantization methods relying on affine transformations often struggle to capture the true distribution of the weights within neural networks. Given that machine learning models typically exhibit weight distributions akin to t-student or normal distributions [van Baalen et al., 2023], they usually contain outliers. When such outliers are mapped through an affine function, they become distorted due to the limited representational capacity of the function (see Figure 1.4),

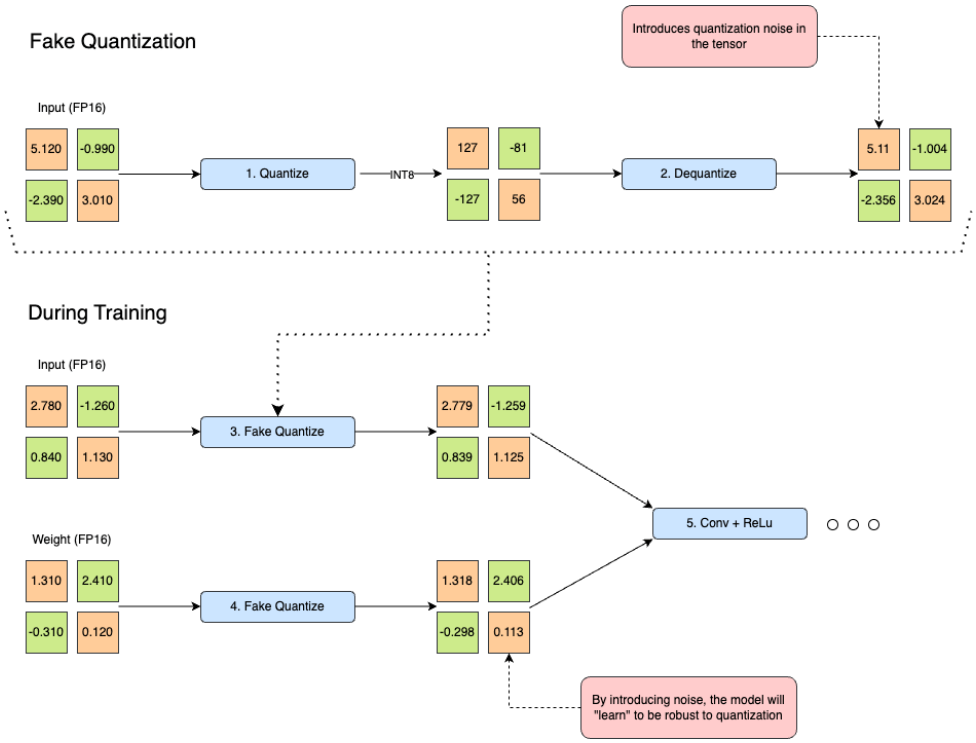


Figure 2.6: Explanation of fake quantization and its impact in the forward pass in Quantization Aware Training.

leading to substantial performance drops.

Moreover, the tradeoff between reducing bit-width and maintaining model accuracy is another core challenge. As bit-widths are reduced, the amount of representable information diminishes, which often leads to a drop in accuracy. In extreme cases, such as with 4-bit quantization, the accuracy loss can be substantial if not mitigated by applying more complex techniques such as LLM.int8 [Dettmers et al., 2022].

2.4 Data Types used in Machine Learning

Machine learning models can be represented using various data types, including floating-point and integer formats. Typically, floating-point formats are employed during the training phase due to their precision and flexibility, while integer formats are used during inference, after applying some quantization technique, due to their speed and memory improvements.

2.4.1 Floating-Point Formats

The most common floating-point formats used in machine learning are FP32, FP16 [Micikevicius et al., 2017], and BFloat16 [Kalamkar et al., 2019]. Floating-point representations allow for a distinct allocation

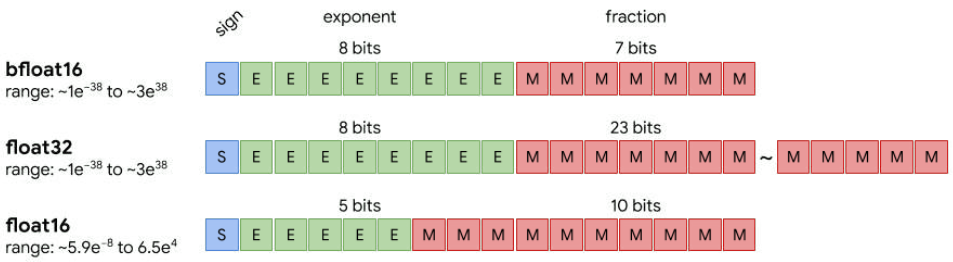


Figure 2.7: Comparison between FP32, BFloat16 and FP16 data formats. Image from [Kalamkar et al., 2019].

of bits between the exponent and mantissa, facilitating a wide range of numerical values. These formats are generally expressed as follows:

$$z = (1 + \frac{\hat{m}}{2^M}) \times 2^{\hat{e}-B} \tag{2.5}$$

Where $\hat{m} = m_1...m_M$ is the mantissa, $\hat{e} = e_1...e_E$ the exponent, B the exponent bias, M the number of mantissa bits and E the number of exponent bits.

When comparing the above data formats, FP16 and BFloat16 with FP32:

- FP16 requires 2$\times$ less memory and bandwidth, enabling faster processing and larger model training. However, its reduced range leads to some accuracy loss.
- BFloat16 also requires 2$\times$ less memory and bandwidth but retains the same number of exponent bits as FP32, retaining its full dynamic range (see Figure 2.7). This makes BFloat16 a data format that balances the speed of FP16 with the precision of FP32 making it a popular choice for training.

Once the model has been trained and is ready for deployment, it can be quantized, converting the weights/activations of the model from floating-point formats to integer formats for efficient execution.

2.4.2 Integer Formats

As for the integer data formats used in quantization, INT8 [Jacob et al., 2018, Dettmers et al., 2022] is the most commonly employed format, followed by INT4 [Wu et al., 2023]. Additionally, some techniques enable quantization down to binary formats [Hubara et al., 2016], allowing for even greater reductions in model size and memory usage.

Integer representations typically have simpler structures compared to floating-point formats, which allows for more efficient storage and computation. A standard integer representation can be defined as follows:

$$z = s \times \hat{z} + b \tag{2.6}$$

Where s is the scale parameter, and b is an optional bias, the $\hat{z} = z_1...z_N$ values are individual bits and N the number of bits.

It should be noted that some data formats are not supported in all GPUs (e.g., Nvidia V100 does not natively support BFloat16 while H100 [Choquette, 2023] already does). Using unsupported formats requires simulating them which hinders performance considerably.

3

Related Work

Contents

In this chapter, we examine various techniques for quantization of machine learning models. There are multiple methods to compress models, each with its unique features and tradeoffs. These methods differ in their application stages, performance characteristics, and balance of memory and computational overhead.

We cover both traditional affine quantization approaches and more unconventional methods, such as those that utilize lookup tables or non-linear functions, highlighting their advantages and limitations.

The field of machine learning quantization is extensive, and for readers seeking a broad overview of the subject, we recommend recent surveys [Gholami et al., 2022, Rokh et al., 2023, Li et al., 2024] that explore the entire landscape in broader detail.

3.1 Affine Quantization

Affine quantization is a widely studied method that, as mentioned in Chapter 2, represents floating-point numbers with a uniform distribution of values across a fixed range, using a linear function to map between both formats. This allows for a more compact model representation and the use of more efficient instructions on many hardware platforms.

One prominent approach within affine quantization is Post-Training Static Quantization (PTSQ). This method is more popular when compared with DQ and QAT due to its cost-effectiveness. Furthermore, since the technique presented in this thesis, PTNQ, aligns more closely with PTSQ and DQ, this section will focus primarily on these approaches rather than on QAT.

Earlier works usually applied per-tensor quantization methods [Wu et al., 2020b]. However, as neural network models increased in size and complexity, using a single scaling factor for quantizing an entire tensor resulted in considerable information loss.

This gave rise to more advanced quantization techniques like block-wise and vector-wise quantization that divide the weight tensor into smaller sections, and compute separate scaling factors for each of them. This reduces distortion of the scale and range used for quantizing, reducing the overall error. One of the first techniques to follow this approach was ZeroQuant [Yao et al., 2022].

Later work realized that a key challenge in achieving accurate quantized models is handling outliers in both weights and activations since they can significantly reduce model accuracy if not handled properly [Kovaleva et al., 2021]. This finding fueled the development of many techniques in the following years that aimed to address this issue.

Techniques like LLM.int8 [Dettmers et al., 2022] introduced methods for isolating and correctly quantizing tensors with outliers to enhance performance. This is achieved through a mixed-precision decomposition scheme, where the outlier feature dimensions are stored in an FP16 matrix, while the remaining 99.9% of the values are kept in an INT8 matrix.

Moreover, since weights are generally easier to quantize than activations, which exhibit significant variability, SmoothQuant [Xiao et al., 2023] shifts the quantization challenge from activations to weights. This is accomplished through a per-channel transformation that scales down the activations while scaling up the weights. However, this technique requires selecting a migration strength hyperparameter α that controls how much quantization difficulty should be transferred from activations to weights.

Finding the most impactful outlier values and handling them appropriately was also a complex task. This motivated development of techniques that incorporate weight sensitivity analysis, such as GPTQ [Frantar et al., 2022] and SqueezeLLM [Kim et al., 2023].

GPTQ quantizes model weights in a layer-wise manner by solving an optimization problem that minimizes the squared error between layer outputs in full precision and low precision. This leverages approximate second-order information from the Hessian matrix of the layer to determine the importance of

each weight for quantization.

In contrast, SqueezeLLM proposed a sensitivity-based non-uniform quantization using second-order Hessian to allocate more bins to sensitive weights and an additional dense-and-sparse decomposition to isolate outlier values into a sparse structure to maximize runtime efficiency using sparse kernels.

Nowadays, research has shifted towards methods that aim to reduce the impact of outliers without the need for separate processing. Techniques like QuaRot [Ashkboos et al., 2024] and SpinQuant [Liu et al., 2024] propose rotating matrices to effectively eliminate outliers. While these methods show promise in maintaining model accuracy, they come at the cost of increased computational overhead due to the additional transformations required during inference.

3.2 Non-Linear Quantization

Traditionally, the term "non-linear quantization" has been associated with the quantization of non-linear layers, such as Softmax, GELU, or LayerNorm. These layers pose unique challenges due to their non-linear activation functions, and various methods have been developed to address this [Lin et al., 2022, Kim et al., 2024, Kim et al., 2021, Li and Gu, 2023].

In this thesis, however, we adopt a different interpretation. Here, non-linear quantization pertains to the application of non-linear approaches to compress linear layers.

While affine quantization methods dominate the field, they are not always sufficient for models with complex data distributions, where uniform scaling introduces significant information loss. Non-linear quantization methods aim to address these limitations by employing more flexible techniques, such as using non-linear functions or lookup tables to encode values.

3.2.1 Lookup Tables (LUTs)

Lookup Table (LUT) quantization has emerged as an alternative to the uniform quantization techniques, offering the ability to map intervals to arbitrary values, as can be seen in Figure 3.1. This approach, explored by LLUT [Wang et al., 2022] and further developed by FLUTE [Guo et al., 2024], allows for better preservation of outlier information compared to uniform quantization.

LUTs can potentially capture the distribution of weights more accurately, leading to improved model performance. However, the learning process for these lookup tables is challenging since LUTs are not differentiable, thus requiring sophisticated optimization techniques. Additionally, models quantized with LUTs consume more memory when compared to other approaches due to needing to store the whole function mapping.

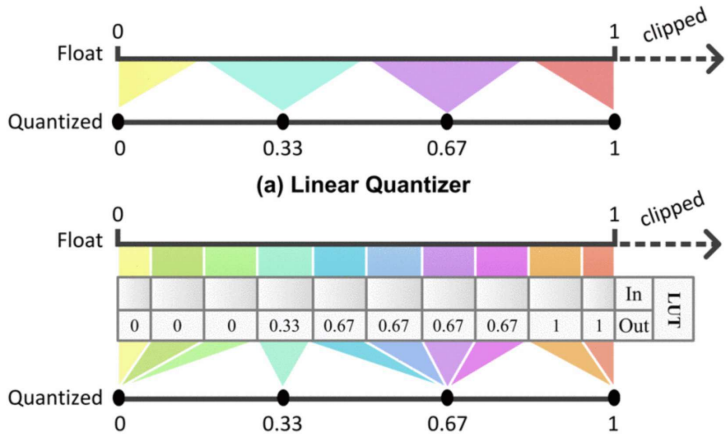


Figure 3.1: Value mapping comparison between affine quantizer (top) and learned lookup table for 2-bit (bottom). Image from [Wang et al., 2022].

3.2.2 Non-Linear Functions

Another approach to non-linear quantization is the use of functions that more accurately reflect the distribution of weights and activations. One of the earliest techniques in this area was logarithmic quantization, which uses base-two logarithmic functions to represent values at lower precision [Miyashita et al., 2016, Cai et al., 2018]. This method is particularly useful for networks where weights follow a normal distribution.

More recently, techniques like Power-of-Two (PoT) quantization [Yan et al., 2024] have been developed, where both weights and activations are quantized into power-of-two representations. This allows for efficient integer-only computations while still approximating the original data distributions.

While these methods provide improved accuracy compared to affine quantization, they come at the cost of increased computational complexity, thus potentially making the model execution slower.

Even though not widely supported, logarithm-based quantization functions can be implemented efficiently in hardware using architectures like FPGAs [Jiang et al., 2024] to accelerate computation and reduce energy consumption.

Despite the potential benefits, non-linear quantization remains relatively underdeveloped compared to traditional affine methods. There is significant potential for further research, especially in exploring new non-linear functions and optimizing their performance on various hardware platforms.

4

Post Training Non-Linear Quantization

Contents

In this chapter, we present Post-Training Non-Linear Quantization (PTNQ), a technique designed to address the limitations of traditional affine quantization methods by leveraging the expressivity of non-linear functions to achieve more effective model compression and preserving performance.

The chapter starts with a high-level overview, allowing readers to grasp the broader context of the technique before delving into the specific details. Subsequent sections then break down the key components of PTNQ, examining each in detail to illustrate how they contribute to the overall functionality of the system.

Following the overview, we will examine the four main aspects of the technique:

1. **Transformation of Mathematical Functions to Executable Code** (Section 4.2): This section

explores how we transform user-provided primitive mathematical functions into Torch code that can be executed for quantizing and dequantizing weights.

2. **Initialization of Quantization Parameters** (Section 4.3): Here, we discuss the methods used to initialize Quantization Parameters (QPs). Proper initialization is essential to ensure that the non-linear functions operate within an optimal range, providing a good starting point that enhances the overall quantization performance.

3. **Training Methodologies to Minimize Quantization Noise** (Section 4.4): In this section, we introduce various training methodologies that can be employed to fine-tune the QPs and thus, minimize quantization noise as well as preserve the accuracy of the model after quantization.

4. **Evaluation and Application of Quantization Methods** (Section 4.5): Finally, we demonstrate how the generated Quantization Methods (QMs) are used to quantize the model. This section will also cover the benchmarking process, where different QMs are evaluated against key performance metrics such as accuracy and perplexity.

4.1 Overview of the Technique

The PTNQ technique is categorized under the Weight-Only Quantization (WOQ) techniques, which focus on compressing the model by quantizing its weights while maintaining as much of its original performance as possible. Then, during inference, each weight is dequantized into its original data type (e.g., 16-bit floats) before executing the matrix multiplication on each layer.

The first stage of the PTNQ process involves generating Quantization Methods (QMs). These methods are defined as a combination of three crucial components: a non-linear function, a QP initialization method, and a QP training method. By combining them to create a diverse set of methods, it provides the technique with the capability to handle different aspects of the weight distribution and characteristics.

In the second stage, these QMs are systematically benchmarked on the target model. This benchmarking process evaluates the performance of each QM against metrics such as accuracy. By comparing these metrics across different QMs, the technique identifies the most effective quantization strategy for the specific model and task at hand.

Finally, after evaluating the proposed QMs, the system selects the best-performing method and applies it to quantize the model. The entire process is illustrated in Figure 4.1, which provides a broad representation of the stages involved in the PTNQ technique.

The key motivation for this pipeline is that there is no one-size-fits-all quantization method and, by following these stages, the PTNQ technique provides a structured approach to model quantization by searching through a list of QMs and selecting the best one.

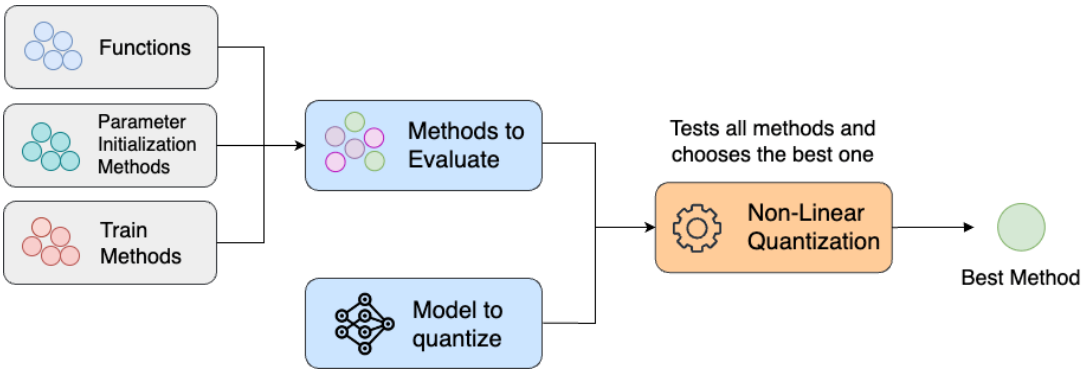


Figure 4.1: Broad representation of the stages in the PTNQ technique.

While this approach may result in a slower quantization process, the tradeoff is justified, as the benefit of fitting a model onto smaller devices often outweighs the additional quantization cost. Moreover, the cost of training and inference is significantly higher than the cost of quantization, making it is reasonable to invest additional time in this stage, as the long-term gains in efficiency and deployment flexibility make the extra effort worthwhile.

4.2 Generating Quantization Functions

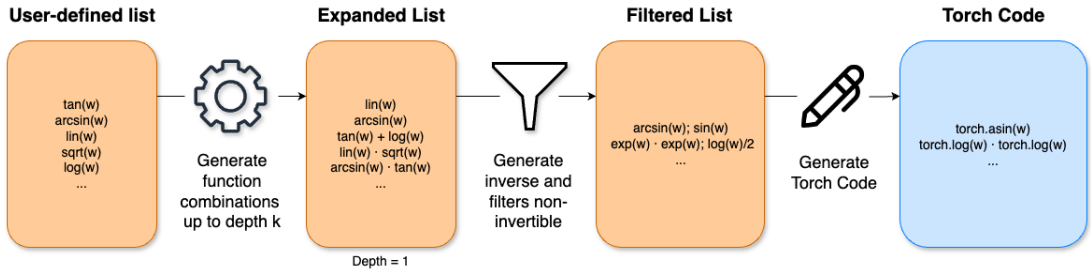


Figure 4.2: Different stages of generating quantization/dequantization PyTorch code from a list of functions.

The broader process of generating quantization and dequantization PyTorch code for our PTNQ technique can be viewed on Figure 4.2.

It begins with a user-defined list of primitive mathematical functions in SymPy[1]. These functions are systematically combined up to a specified depth k, using basic arithmetic operations and by composing the functions with each other, creating an expanded set of potential quantization methods.

The set of generated functions is then filtered so that non-invertible functions are dropped. This restriction has two motivations: (a) we need to dequantize the weights during inference and having the inverse function guarantees that we can do it efficiently and accurately; and (b) it simplifies the

[1] https://www.sympy.org/en/index.html

optimization of the function parameters - refer to Section 4.3.

The final set of functions and their inverses is then translated into PyTorch code, implementing both the quantization and dequantization functions.

This set of stages, as depicted in Figure 4.2, results in diverse set of quantization and dequantization functions that will later be composed with a QPs initialization method and a training method to generate a Quantization Method (QM).

4.2.1 Initializing and Composing Functions

The process of initializing and composing functions in our PTNQ technique begins with the definition of an initial list of primitive functions. This list can be customized by the user or left to utilize the default values provided by the system (see Section 5.1 for a complete list).

When defining a `QuantizationExpression`, the user provides a SymPy expression that must adhere to two specific requirements:

1. The expression must include the SymPy `Symbol`[2] X, which represents a linear layer weight within the function and that will be replaced by the actual weight during execution.

2. The expression must contain at least one QP, defined as a SymPy `Symbol` prefixed with an underscore (e.g., $\_a$). These QPs are crucial for tuning the transformation, allowing the quantization process to be adjusted for optimal performance.

To further clarify the process of defining and working with SymPy expressions within the `QuantizationExpression` class, we provide an example in Listing 4.1. This example demonstrates how to create a simple SymPy expression using the required symbols.

Listing 4.1: Initialize SymPy expressions and QuantizationExpressions.

```
import sympy as sp
from fn_gen.src.expression import QuantizationExpression

x, _a = sp.Symbol('x'), sp.Symbol("_a")  # Initialize Symbols
e = sp.tanh(x * _a)                      # Primitive sympy expression

# Build QuantizationExpression that is going to be converted to PyTorch code
qe = QuantizationExpression(qtz_expr=f1, ...)
```

Once the initial set of functions is defined, they can be composed by adding or multiplying two functions together, creating a more complex function that can possibly adjust better to the weight distributions and reduce quantization error. The depth of this composition is controlled by a user-defined parameter k, which determines the maximum depth of function composition. By default, k is set to zero, meaning that only the primitive functions are used without any additional composition.

[2] https://docs.sympy.org/latest/explanation/glossary.html#term-Symbol

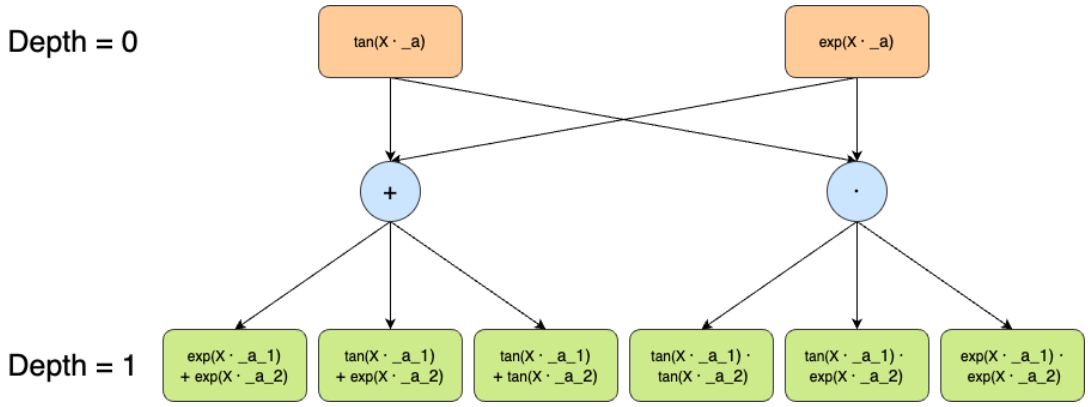


Figure 4.3: Tree structure representing the composition of functions up to depth k.

During the composition process, if two functions contain a symbol with the same name, a deduplication step is performed to ensure that there is only one reference to each QP. This deduplication is crucial to avoid potential errors and to maintain higher degrees of freedom during training.

The composition process can be visualized as a tree structure, as depicted in Figure 4.3, where each node represents a function and each branch represents an operation such as addition or multiplication.

Finally, an additional scale parameter, denoted as $\_s$, is appended to each expression. This parameter is specifically used to fine-tune the output of the function, ensuring that the transformed weight values remain within a manageable range. The importance of this parameter and its role in the quantization process will be further discussed in Section 4.3.1.

4.2.2 Obtaining Dequantization Functions

Since our method follows a Weight-Only Quantization (WOQ) approach, we need to dequantize the weights during inference and, as such, we require the inverse expression of each of the previous ones. To compute their inverses, we leverage SymPy `solve`[3] function and its symbolic computation capabilities that allow inverting arbitrary expressions.

During this process, we modify the original expression by introducing a temporary symbol to represent the output of the expression, and solve for the original variable. If the expression has multiple valid inverses, the function returns all of them as a list. To ensure consistency, the temporary symbol is substituted back with the original variable once the inverse is computed.

If a timeout - set to a default value of thirty seconds per expression - occurs or the `solve` function raises an exception, possibly because the equation can not be solved, we mark the function as non-invertible and discard it.

With all the expressions computed, we apply a post-processing pipeline to ensure efficiency and

[3] https://docs.sympy.org/latest/modules/solvers/solvers.html

compatibility with PyTorch of all expressions. The pipeline is as follows:

1. **Filtering incompatible expressions**: First, we discard all expressions that contain operations or functions unsupported by PyTorch such as certain symbolic solutions that may involve non-differentiable components (e.g., non-elementwise functions or complex logarithms).

2. **Simplifying expressions**: Simplifying symbolic expressions reduces computational overhead during inference, ensuring faster execution by eliminating redundant terms. We use the `simplify`[4] function from SymPy to achieve this.

3. **Heuristic for selecting the inverse function**: Since the inversion process can output more than one expression, it is necessary to pick one. We adopt a heuristic based on the size of the symbolic expression. For each original expression, we convert the simplified inverse expressions to their string form and select the one with the smallest length, based on the assumption that smaller expressions generally require fewer operations and, therefore, result in faster computations. Although this method may not always yield the optimal solution, it has proven to be effective.

This approach ensures that we obtain compatible and fast pairs of quantization and dequantization expressions that are directly deployable into PyTorch code.

4.2.3 Generating Torch Code from SymPy Expressions

Having obtained the quantization and dequantization expressions in Section 4.2.2, the next step is to convert these mathematical expressions, represented in SymPy, into executable PyTorch code that will be used in later stages of the pipeline.

To achieve this, we process each SymPy expression by recursively converting its components into their PyTorch equivalents. For instance, basic symbols like variables are directly mapped to a parameter that will be added as input to the wrapping function, while numerical values (e.g., integers and floats) are converted into tensor objects.

Our translation also handles more complex operations such as addition, multiplication and powers by mapping them to their corresponding PyTorch operations. For special mathematical functions (e.g., square roots, trigonometric), we use a predefined map that ensures these are correctly translated into the correct native PyTorch function.

It should be noted that many of the non-linear functions we use, such as logarithms and trigonometric functions, have restricted domains. For instance, logarithmic functions are only defined for positive numbers and certain trigonometric functions can encounter undefined behavior outside specific intervals. If not properly handled, these undefined values can lead to instabilities during training or inference, manifesting as NaN values or crashes in computation graphs.

[4] https://docs.sympy.org/latest/modules/simplify/simplify.html

Table 4.1: Domain guards for common non-linear functions. The input to these functions is clipped to stay within the allowed domain.

Function	Domain Guard
$\log(x)$	$x \geq 1 \times 10^{-5}$
$\sqrt{x}$	$x \geq 0.1$
$\arccos(x)$	$-0.99999 \leq x \leq 0.99999$
$\arcsin(x)$	$-0.99999 \leq x \leq 0.99999$
$\tan(x)$	Sets the input to ± 1 when $x = \pm\infty$ to prevent infinite outputs.
$\operatorname{arctanh}(x)$	$-0.9999 \leq x \leq 0.9999$
$\operatorname{arccosh}(x)$	$x \geq 1$
x^y	$x \geq 0 \vee y \geq 1$

To mitigate this risk, when generating the PyTorch code, we automatically introduce domain guards around the input of these functions to clip their values and ensure they are within valid ranges. The list of all domain guards used in our implementation is provided in Table 4.1, along with their respective ranges and applied functions.

4.3 Defining Parameter Initialization Methods

In the previous sections, we discussed the process of generating quantization and dequantization PyTorch functions from symbolic expressions. However, it is important to recognize that these functions do not operate in isolation.

They depend on a set of Quantization Parameters (QPs) that tune their transformations, and the effectiveness of the quantization process relies heavily on how well the values of these QPs reduce the quantization error.

For instance, consider the candidate function $\cos(\_a \cdot x) \cdot \_s$, where x represents the value of a weight to be quantized, while $\_a$ and $\_s$ are QPs. These QPs play a critical role in determining how well the function approximates the original weight distribution after dequantization.

Poor initialization of $\_a$ or $\_s$ can lead to suboptimal quantization results, affecting the performance of the model. We will later show that a good initialization strategy is critical to achieve good performance.

In this section, we describe the QPs initialization techniques we employ and explore how they contribute to the overall success of the quantization process. All of the techniques we used adopt a per-channel QPs initialization strategy, meaning that for each channel in the layer, we use a distinct set of parameters. This choice is common among quantization workflows [Li et al., 2024] as it allows for better adjustment to each channel distribution, reducing the overall accuracy degradation of the model. However, this comes at the cost of higher memory usage due to an increase in parameter storage.

4.3.1 Initialization of the Scale parameter

As discussed in Section 4.2.1, the scale QP, denoted as $\_s$, plays a crucial role in adjusting the output of the quantization function, ensuring that the transformed weight values range over the full domain of the chosen quantization bitwidth.

For instance, consider the $\sin$ function, which outputs values in the range of $[-1, 1]$. For an 8-bit integer quantization scheme, the valid range of quantized values is $[-128, 127]$. If the scale parameter is not correctly initialized, we fail to fully utilize this range, which can result in sub-optimal representation of the weights. Later, we illustrate the impact of an incorrect scale parameter initialization in Section 6.1.1.

To address this, the scale parameter $\_s$ is initialized based on the principles of affine quantization. Recalling Chapter 2, in affine quantization, the scale factor s is defined as the ratio between the range of the original data [α, β] and the range of quantized values [α_q, β_q]:

$$S = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{4.1}$$

We adapted this concept to our PTNQ technique by computing the scale parameter $\_s$ based on the transformed weight tensor, which results from applying the non-linear function. Here, α_q and β_q correspond to the minimum and maximum values representable in the selected quantization format. This initialization ensures that the quantized weights are appropriately scaled to fully utilize the supported range as seen in Listing 4.2.

Listing 4.2: Initialization of the $\_s$ parameter in the quantization process.

```
def init_scale(W):
    F_q = torch.sin                     # A non-linear function
    _a = torch.randn(W.size(-1))        # Example of _a parameter
    alpha, beta = -128, 127             # For 8 bits quantization

    # Quantize the weights
    W_trf = F_q(W * _a)
    W_q = torch.clamp(torch.round(W_trf), alpha, beta)

    # Obtain the quantized range before calculating S
    alpha_q, beta_q = W_q.min(dim=-1), W_q.max(dim=-1)
    _s = (beta - alpha) / (beta_q - alpha_q)

    return _s
```

It is important to note that the scale parameter is treated separately from the other QPs. The QPs initialization methods discussed in the following sections will focus on the other parameters of the quantization functions, as the scale parameter follows this dedicated initialization process.

4.3.2 Simple Initialization

Our technique supports two fairly straightforward initialization strategies of QPs. The first method involves setting all QP to a constant value of one. This approach is simple and ensures a uniform starting

point for all parameters.

The second method initializes the QPs through random sampling from a standard normal distribution within the range of $[-1, 1]$. This introduces some variability into the initialization process, which may be beneficial in scenarios where a more diverse set of starting QPs helps reducing the quantization noise.

Although these initialization methods are basic, they are nonetheless effective for specific models, such as TinyLlama [Zhang et al., 2024]. In fact, empirical results demonstrate that simpler initialization schemes can yield satisfactory performance, as shown in Section 7.4.

4.3.3 Space Search

Space Search is an initialization technique whose motivation stems from the observation that certain functions, like $\log$, require initial values outside the typical range of $[-1, 1]$ (refer to Section 6.1.3) supported by the other two methods.

The technique follows an iterative process designed to efficiently explore a high-dimensional parameter space that begins by generating random parameters with a large range of values as initial candidates. It evaluates these by calculating the Mean Squared Error (MSE) between the original weights and their quantized-dequantized versions, after which the top 10% are kept.

From this group, the algorithm computes the average and maximum values for each channel to generate new parameters within the narrower range of $[-(\mathsf{max} + \mathsf{avg}), \mathsf{max} + \mathsf{avg}]$. The whole process is repeated 50 times and, in the final step, the best parameters from the last iteration are chosen as the optimal solution.

After the initial QPs values are determined by Space Search, they are compared against QPs initialized as ones and those initialized randomly. If either of these fallback initializations yield a lower loss value, the algorithm defaults to using them instead of the Space Search result.

This process ensures that the final QPs are validated against simpler alternatives to avoid poor local minima often due to insufficient iterations or inadequate exploration of the parameter space. The Space Search technique extended with fallback options can be viewed in Listing 4.3.

Listing 4.3: Initializing QPs using Space Search with fallback options.

```
def space_search_fb(W, F_q, F_dq):

    # Computes QPs and loss when using Space Search
    ss_qps = space_search(W, F_q, F_dq)
    W_ss = fake_quantize(W, F_q, F_dq, *ss_qps)
    loss_ss = mse_loss(W, W_ss)

    # Computes QPs and loss when initializing as ones
    ones_qps = init_ones(W)
    W_ones = fake_quantize(W, F_q, F_dq, *ones_qps)
    loss_ones = mse_loss(W, W_ones)

    # Computes QPs and loss when initializing at random
    rnd_qps = init_random(W)
    W_rnd = fake_quantize(W, F_q, F_dq, *rnd_qps)
    loss_rnd = mse_loss(W, W_rnd)

    # Picks best results out of the three
    if loss_ones < loss_rnd and loss_ones < loss_ss:
        return ones_qps
    elif loss_rnd < loss_ones and loss_rnd < loss_ss:
        return loss_rnd
    else:
        return ss_qps
```

4.3.4 Approximating through Non-Linear Regression (NLR)

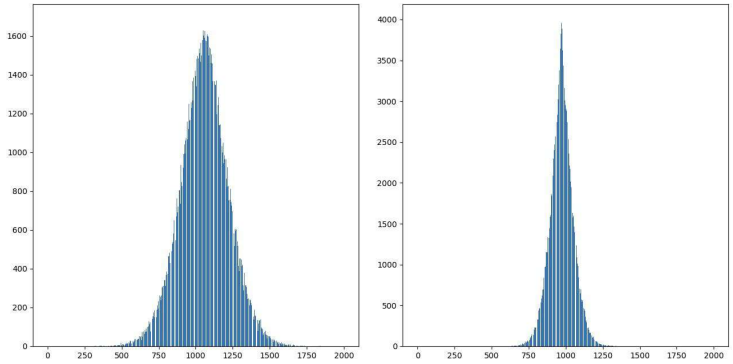
After initializing the QPs, we can further optimize them by applying Non-Linear Regression (NLR), specifically using non-linear least squares fitting. NLR allows us to refine the QPs by fitting the output of a candidate function f (in our case, f is a function that quantizes and immediately dequantizes the weights) to the distribution of weight values within each channel. This process minimizes the difference between the actual weight values and their corresponding quantized-dequantized values, thereby reducing the reconstruction error between the two.

The motivation for this approach lies in the fact that weight distributions in neural networks often resemble well-known distributions, such as normal or t-student, as seen in Figure 4.4a. In particular, when these weights are sorted within a channel and plotted, they frequently exhibit shapes that resemble smooth curves, such as tangent. As such, we can fit a function such as $\operatorname{arcsinh}$ to the original values as can be observed in Figure 4.4b.

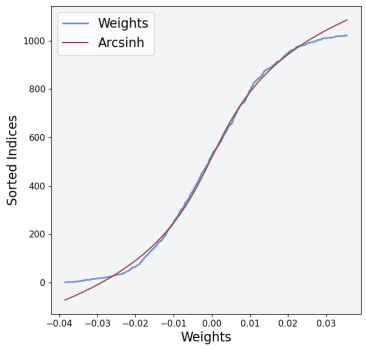
To implement this approach, we sort the weights within each channel and apply the `curve_fit`[5] function from SciPy, which uses non-linear least squares optimization. This method adjusts the QPs iteratively (as seen in Figure 4.5) to minimize the error between the original weight values and those approximated by the function f. The optimization works by fine-tuning the parameters to find the best fit for the weight distribution, ensuring that the values are as close as possible to the original weights.

By using NLR, we tune the QPs beyond their initial estimates, ensuring that the function f captures

[5] https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.curve_fit.html



(a) Common weight distributions in neural networks. Taken from OPT [Zhang et al., 2022].



(b) Example of curve fitting

Figure 4.4: Common weight distributions in neural networks (left) and the application of curve fitting (right) to model these distributions, facilitating non-linear regression for quantization.

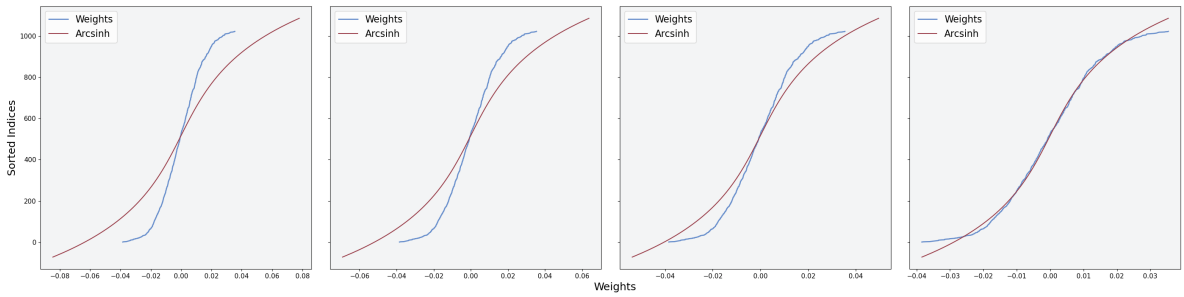


Figure 4.5: Fitting an $\mathrm{arcsinh}$ function to the values of a channel from a layer in Wav2Vec [Baevski et al., 2020].

the specific characteristics of the weight distributions more effectively. In practice, this leads to lower reconstruction errors and more accurate quantization, particularly in cases where the weights exhibit non-linear relationships.

4.4 Refining QPs Through Training

Following the initialization of QPs in Section 4.3, training aims to refine them by minimizing the layer-wise Mean Squared Error (MSE) between the original weights and their quantized-dequantized counterparts. This step may allow the model to retain more of its performance post-quantization. We will see in Chapter 6 that there are exceptions.

This approach is similar to the QAT technique, explained in Section 2.2.3, particularly the "fake-quantization" process, which incorporates quantization error into the training loss, encouraging the model to train parameters amenable to quantization. However, unlike QAT, which aims to optimize the end-to-end performance of the entire model, our method focuses on minimizing the layer-wise MSE, thereby reducing quantization error at the level of individual layers.

The weights of each linear layer are trained for 500 epochs and, throughout the training process, the best values are stored across all epochs, ensuring that the optimal results are preserved. By checkpointing the best-performing weights, the model is robust against temporary increases in loss.

During training, all functions used during the forward pass have to differentiable otherwise, the non-differentiable function results in zero gradients during backpropagation, effectively halting the learning process. The round function, commonly used in quantization, falls among the latter example.

To address its non-differentiability, we employ a technique known as Backward Pass Differentiable Approximation (BPDA) [Athalye et al., 2018]. BPDA resolves this by approximating the backward pass with a differentiable function (in this case, the identity function), allowing gradients to flow and enabling optimization to continue. This approach is critical when training the QPs, as it ensures that the model can still tune them despite the non-differentiability of the rounding operation.

The Learning Rate (LR) is a key factor in stabilizing this training process. Since the QPs are initialized to closely approximate their final values, an improperly tuned learning rate, especially one that is too high, could destabilize the training. This instability can lead to parameter divergence, where the performance of the model degrades rather than improves. Thus, careful learning rate adjustment is vital to maintain stability and ensure that the training converges as intended.

To handle this, we employ a dynamic initial learning rate scaled based on the initial loss value, defined as $10^{E/2-1}$, where E represents the exponent of the initial loss. An illustration of this can be seen on Listing 4.4. This approach allows for a smaller learning rate when the initial approximations are accurate, preserving the benefits of good initializations. In cases where the initial QPs are poorly approximated, a higher learning rate accelerates convergence. This balance ensures that training is stable for well-initialized QPs while promoting faster optimization for less accurate ones.

Listing 4.4: Computing the initial learning rate value based on the initial loss.

```
def init_lr(W, F_q, F_dq, *qps):
    base_lr = 0.1          # Base initial learning rate value

    # Computes the initial loss
    W_q = quantize(W, F_q, *qps)
    W_noisy = dequantize(W_q, F_dq, *qps)
    loss = MSE(W, W_noisy)

    # Computes the intial learning rate base on the initial loss
    exponent = int(torch.floor(torch.log10(loss)))
    lr = base_lr * (10 ** (exponent // 2))
    return lr
```

Moreover, the point above also speaks to the necessity of using LR schedulers. A learning rate scheduler dynamically adjusts the learning rate during training, allowing for better control over the optimization process. By reducing the learning rate at appropriate times, a scheduler can help the model avoid overshooting minima and stabilize the training process [Loshchilov and Hutter, 2017]. The following sections describe the three different LR schedulers supported by our system and their advantages.

4.4.1 Learning Rate with Linear Decay

The LR decay scheduler gradually decreases the learning rate over time by a constant amount at each epoch. This approach reduces the learning rate smoothly from its initial value until a predefined minimum value or a milestone epoch is reached. The formula for linear decay is given by:

$$\mathrm{LR}(t) = \mathrm{LR}_0 - \frac{t}{T}(\mathrm{LR}_0 - \mathrm{LR}_{\text{final}}) \tag{4.2}$$

where:

- $\mathrm{LR}(t)$ is the LR at epoch t,
- LR_0 is the initial LR,
- $\mathrm{LR}_{\text{final}}$ is the final LR at the end of training,
- T is the total number of epochs, and
- t is the current epoch.

The motivation for using linear decay is to allow the training process to begin with a high enough LR to quickly approach a good solution while reducing the risk of overshooting or instability in later epochs. In the early stages of training, a larger LR helps the model explore the loss surface and converge toward a promising region. As training progresses, the reduced LR enables more fine-grained adjustments, preventing the model from overshooting optimal values and leading to more stable convergence.

4.4.2 LR Scheduling through Cosine Annealing and Warm Restarts

Cosine annealing with warm restarts [Loshchilov and Hutter, 2017] is a dynamic LR scheduling technique that adjusts the LR according to a cosine function, gradually decreasing to a minimum before restarting at a higher value.

The warm restarts allow the model to escape local minima and explore the solution space more thoroughly by introducing periodic boosts to the LR. An illustration of the process as well as a comparison with a standard LR scheduler can be found in Figure 4.6.

The LR at any epoch η_t is computed using the equation:

$$\eta_t = \eta_{\text{min}} + \frac{1}{2}(\eta_{\text{max}} - \eta_{\text{min}})\left(1 + \cos\left(\frac{T_{\text{cur}}}{T_i}\pi\right)\right) \tag{4.3}$$

where:

- η_t: Represents the LR at the current time step or epoch t. The LR determines the step size during each iteration in gradient-based optimization algorithms.

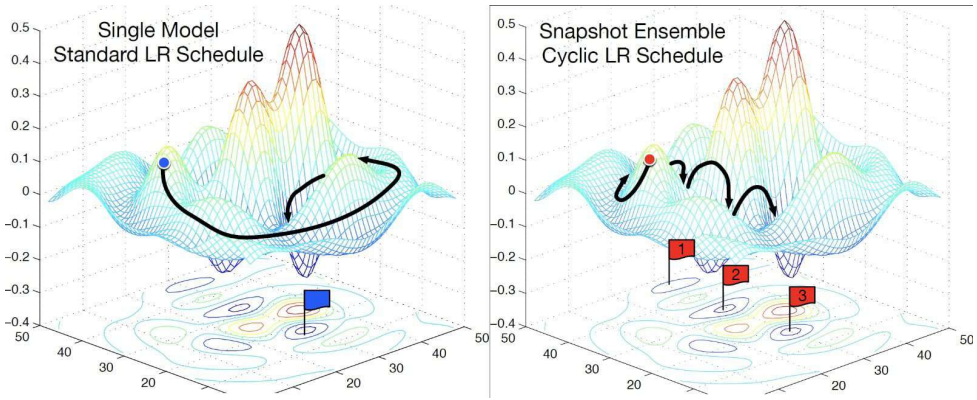


Figure 4.6: Typical learning rate scheduling vs. a warm restarts approach. Image from [Huang et al., 2017].

- η_{min}: The minimum LR that the scheduler will reach. It acts as a lower bound for the LR.
- η_{max}: The maximum LR at the start of a new cycle. Each time the LR is restarted, it starts again from this value.
- T_{cur}: The current epoch number within the current cycle. It resets to zero after every cycle. This parameter controls the current position within the cosine curve.
- T_i: The number of epochs in the current cycle. This defines the period of the cosine function and determines how quickly the LR decays within a cycle.
- π: The constant π is used in the cosine function to define the shape of the cosine curve over the interval $[0, T_i]$.

In essence, this method effectively balances exploration and exploitation during training. The cosine decay enables the model to fine-tune its parameters with a decreasing LR, which is crucial for convergence. The warm restarts provide timely interventions to increase the LR and enhance the overall training efficiency.

4.4.3 No Learning Rate Scheduler

In the case of no learning rate scheduler, the learning rate is maintained at a constant value throughout the entire training process. This static approach uses the initial learning rate as described in Section 4.4.

Using no LR scheduler, where the LR remains constant throughout the training process, without the ability to gradually reduce its value, may cause the model to struggle to fine-tune its QPs during the later stages of training. However, in our experiments, we have found scenarios where maintaining a fixed learning rate can still be effective.

4.5 Evaluating Quantization Methods

After defining the functions, QPs initialization methods, and training techniques, we proceed to evaluate each QM with the objective of selecting the most suitable approach for compressing the model.

To this effect, we provide an evaluation framework that factors various domain-specific metrics, including accuracy, perplexity, and Word Error Rate (WER). Each of these metrics is relevant for a specific set of models (Section 5.4).

In this section, we describe how PTNQ benchmarks candidate QMs and selects the best.

4.5.1 Benchmark

For each of the QMs to be tested, the benchmarking process begins by quantizing the model using a higher bit-width (e.g., 8 bits) and progressively reducing it, with each successive evaluation testing the ability of the QM to maintain performance at lower precisions.

However, not all QMs are evaluated exhaustively at every bit-width. PTNQ employs a pruning strategy to avoid unnecessary computation: if a QM fails to achieve acceptable performance at a higher bit-width (e.g., zero accuracy), it is discarded from further evaluation at lower bit-widths. This strategy focuses resources on promising methods and accelerates the overall benchmarking process.

During the quantization of the model, we apply the QM that is being tested independently to each of the model layers. Treating each layer separately allows for fine-grained control over the quantization process and ensures that the characteristics of each layer are considered when quantizing the weights.

The algorithm for each of the layers can be broken down into three steps, which can also be viewed in Figure 4.7:

1. **Parameter Initialization**: The first step is to initialize the non-linear quantization function QPs using the selected initialization method.

2. **Parameter Training**: Once the parameters are initialized, they are refined through training. Specifically, by using the training method in the QM.

3. **Quantization**: After the training phase, the quantization function, with its trained parameters, is applied to the weights, quantizing them.

At the end, the quantized model is benchmarked against a suitable dataset and metric that will vary depending on the nature of the model task.

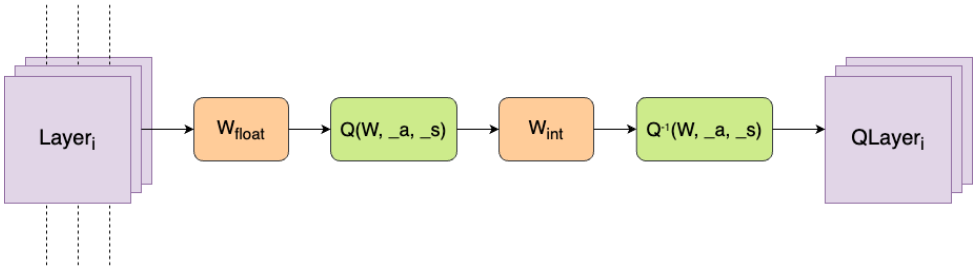


Figure 4.7: In each layer, we first compute the best QPs that best reduce the quantization noise and afterwards, quantize the weights.

4.5.2 Pick Method and Quantize Model

Following the benchmarking phase, the best QM is selected based on the performance metrics relevant to the task of the model. Typically, the QM that achieves the best result in its metric at the lowest possible bit-width is chosen. This choice can be tuned by any user of the technique.

At the end of each benchmarking iteration, all QPs and relevant metadata are saved to disk for future reference. Once the optimal QM is selected, its corresponding state — including QPs and other associated settings — is restored from disk and used to reapply the quantization process to the entire model. This ensures that the final model quantization is consistent with the best-performing configuration from the benchmarking phase.

Ultimately, PTNQ produces a fully quantized model that is both compressed and tuned for its specific performance requirements.

5

Experimental Setup

Contents

In this chapter, we introduce the experimental setup used to develop and evaluate the PTNQ technique.

To continuously test and refine the proposed approach, we developed a custom benchmark system tailored to test various ideas and update the method based on the empirical results that yielded the most optimal outcomes in terms of model accuracy and performance. This chapter details its structure and covering the specific datasets, metrics, models and conditions under which the technique was evaluated as well as ensuring both reproducibility and future extension of the technique.

Furthermore, key components of the setup, such as the datasets, metrics, and configurations, are reused in the benchmarking phase of PTNQ, thus making sure that the process utilized to iterate on the technique is also used to select the best fitting QM for a particular model and bit-width.

5.1 Functions

In developing the PTNQ technique, a critical aspect of the design involved selecting a set of primitive quantization functions that would effectively serve as a good starting point by striking a balance between mathematical expressiveness and computational efficiency. Below is the list of the selected functions:

Table 5.1: Quantization functions. x is the weights; $\_a$ and $\_s$ are QPs.

$x \cdot \_s$	$\log(x \cdot \_a) \cdot \_s$	$\cos(x \cdot \_a) \cdot \_s$	$\tanh(x \cdot \_a) \cdot \_s$	$\text{arcsinh}(x \cdot \_a) \cdot \_s$
$x^2 \cdot \_s$	$\sqrt{x \cdot \_a} \cdot \_s$	$\tan(x \cdot \_a) \cdot \_s$	$\arcsin(x \cdot \_a) \cdot \_s$	$\text{arccosh}(x \cdot \_a) \cdot \_s$
$x^3 \cdot \_s$	$\sqrt[3]{x \cdot \_a} \cdot \_s$	$\sinh(x \cdot \_a) \cdot \_s$	$\arccos(x \cdot \_a) \cdot \_s$	$\text{arctanh}(x \cdot \_a) \cdot \_s$
$e^{x \cdot \_a} \cdot \_s$	$\sin(x \cdot \_a) \cdot \_s$	$\cosh(x \cdot \_a) \cdot \_s$	$\arctan(x \cdot \_a) \cdot \_s$	

The primary reason for selecting these functions was their native support in both PyTorch and SymPy. The Autograd module in PyTorch,[1] seamlessly integrates with them, allowing for gradient calculations during the training of QPs (failure to meet this requirement would halt the learning process). This set of functions constitutes the default primitives defined in our system.

During the tests, we set the `depth` parameter to zero, meaning that none of the aforementioned functions were combined to create more complex expressions. This decision was driven by two main factors: the need to minimize computational costs by limiting the number of functions and the observation that most combined expressions, when tested on a smaller scale, resulted in zero accuracy. As a result, exploring functions with a depth greater than zero has been reserved for future work.

This functions incorporate two important QPs, $\_a$ and $\_s$. The parameter $\_a$ is responsible for adjusting the weights according to the specific domain and the transformations applied by the function, allowing the quantization process to adapt and fit to varying data characteristics. On the other hand, $\_s$ is used to tune the output of the function, ensuring that the transformed values remain within a manageable range. This last parameter is automatically appended to the function as per Section 4.3.1. Both parameters are essential for maintaining the accuracy and performance of the model.

5.2 Models

For a comprehensive evaluation, we selected a set of machine learning models that are representative of different architectures and applications. The chosen models are as follows:

- **Llama3**, **OPT**, **Phi-2** and **TinyLLama**: Language models designed for natural language processing (NLP) tasks, such as text generation, translation, and comprehension.
- **Wav2Vec**: An audio model focused on speech recognition and processing.

[1] https://pytorch.org/docs/stable/autograd.html

Table 5.2: Models used for evaluation, their number of parameters, and memory required to hold the weights with 32-bit floats (the baseline). For each model, we also indicate the GPU used for the experiments, as well as their memory capacity.

Model	Parameters	Memory	GPU	vRAM
ViT [Dosovitskiy et al., 2021]	307.0M	1.23 GB	Nvidia RTX 3070	8 GB
Wav2Vec [Baevski et al., 2020]	317.0M	1.27 GB	Nvidia RTX 3070	8 GB
OPT [Zhang et al., 2022]	350.0M	1.4 GB	Nvidia RTX 3070	8 GB
TinyLLama [Zhang et al., 2024]	1.1B	4.40 GB	Nvidia RTX 3090	24 GB
Phi-2 [et al., 2023]	2.7B	10.80 GB	Nvidia RTX A6000	48 GB
Llama3 [Dubey and et al., 2024]	8.0B	32.00 GB	Nvidia A100	80 GB

- **ViT (Vision Transformer)**: A model designed for image related tasks such as classification and segmentation.

The selection of these models was guided by the need to cover a broad range of domains such as text, image, and audio processing. This diversity was intentional to show that the proposed method is not limited to a specific application but instead, offers a universal improvement across different machine learning tasks.

On a similar note, the memory requirements of a model are also a critical factor in quantization, particularly as models exceed 7 billion parameters, where the presence of outliers increases, making the quantization process more challenging [Dettmers et al., 2022]. To demonstrate that our proposed technique is effective across various model sizes, we selected models ranging from smaller sizes, such as OPT with 350 million parameters, to larger ones, like Llama3 with 8 billion parameters (Table 5.2).

All models used in this study can be accessed via Hugging Face[2] and the specific links to each model is provided in appendix A.

5.3 Datasets

In our tests, we employed three datasets, tailored to the specific domains of the models under consideration. For language models, we chose the WikiText [Merity et al., 2016] dataset, a well-established benchmark in natural language processing with human-curated text, which primarily consists of Wikipedia articles. For vision models, we utilized the ImageNet [Deng et al., 2009] dataset, a widely used dataset for image classification which contains over 14 million labeled images spanning more than 20,000 categories. For audio models, we selected the LibriSpeech [Panayotov et al., 2015] dataset that consists of a large corpus of English speech derived from audiobooks, specifically designed for training and evaluating speech recognition systems.

[2]https://huggingface.co

To maintain consistency across experiments, we utilized the validation split provided by each dataset. Specifically, for each model and dataset, we selected 1,000 samples from the validation set to serve as our test cases. This consistent sample ensured that our results were comparable across different tests for any particular model.

5.4 Metrics

To measure the performance of the PTNQ technique, we used a range of metrics to provide meaningful insights into its effectiveness, given the characteristics of the tasks at hand.

5.4.1 Text Generation Metrics: Accuracy and Perplexity

For language models, we focused on two primary metrics: accuracy and perplexity.

Accuracy quantifies how often the predictions of the model match the true outcomes and it is defined as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{5.1}$$

Higher accuracy indicates better performance, as it reflects that the model was more capable of correctly predicting the next token (small sequence of characters) in a sentence.

Perplexity evaluates how well a model predicts a sample and it is defined as:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log p(x_i)\right) \tag{5.2}$$

where $p(x_i)$ is the probability assigned to the i-th word, and N is the total number of words in the text. A lower perplexity signifies better performance, indicating that the model is more confident in its token predictions.

In summary, accuracy assesses the ability of a model to select the correct token, while perplexity measures the confidence in its choice, reflecting how certain the model is that the predicted token is the correct one.

5.4.2 Vision Model Metrics: Accuracy

For vision models, we used accuracy as the primary metric. Similar to its use in text generation, accuracy for image classification is calculated as:

$$\text{Accuracy} = \frac{\text{Number of Correctly Classified Images}}{\text{Total Number of Images}} \tag{5.3}$$

In image classification tasks, higher accuracy demonstrates improved model capability by correctly classifying more images out of the total tested.

5.4.3 Speech Recognition Metric: Word Error Rate (WER)

For evaluating audio models in the context of speech recognition, we used WER. WER ranges from 0 to 1, where 0 indicates that the compared pieces of text are exactly identical, and 1 indicates that they are completely different with no similarity. It is computed as:

$$\text{WER} = \frac{S + D + I}{N} \tag{5.4}$$

where S represents the number of substitutions, D denotes deletions, I is the number of insertions, and N is the total number of words in the reference. A lower WER indicates better performance and shows fewer errors in the transcriptions generated by the speech recognition model.

5.4.4 Considerations on Speed and Memory

While speed and memory are critical factors in practical deployments, and especially in the context of quantization, they were not the primary focus of this evaluation. Current accelerators, such as GPUs, are not optimized for non-linear quantization transformations, which can impact processing speed. Therefore, our evaluation centered on the effectiveness of the quantization method in preserving model performance on a given task rather than speed.

Memory considerations were also secondary in this context since, in our tests, we simulated quantization to bit widths that are not currently supported by PyTorch (7, 6, 5 and 4 bits) to evaluate the evolution of model accuracy under varying levels of quantization.

To get around this limitation, we converted all values to 8-bit integers and restricted the value range accordingly (i.e. 7 bits is restricted to the interval $[-64, 63]$). This approach allowed us to evaluate the quantization technique's impact on model effectiveness without the need for intricate memory management.

Nonetheless, in Section 7.5 we will analyse and comment on the current performance and memory usage of PTNQ so it can serve as a baseline for future improvements.

5.5 Hardware

Each machine learning model used in the benchmark has a different memory requirement and, to keep the costs low, the hardware utilized during the experiments was chosen to meet the specific memory demands of each model. The hardware choices are summarized in Table 5.2.

It should be noted that the selected accelerator has more available vRAM than the minimum required to execute the model. This is necessary to accommodate the storage of intermediate results, such as gradients and avoid potentially causing an out-of-memory error.

The Nvidia A100s were provided by Minho Advanced Computing Center (MACC)[3], while the remaining accelerators were reserved from RunPod,[4] a cloud computing service.

5.6 Software

The experimental setup also relied on several key libraries to use, process, and evaluate the machine learning models under the PTNQ technique.

The primary framework used was **PyTorch**, a deep learning library that enabled us to instrument and perform custom modifications on the chosen models, particularly, quantizing specific layers.

TorchAO[5] is the PyTorch quantization library and it was integrated to serve as a baseline for our technique since it supports the most widely used quantization methods providing a standard against which the PTNQ technique could be measured.

Both our implementation and TorchAO perform quantization per channel, ensuring precise control over individual weight distributions. For 8-bit integer quantization, TorchAO employs the standard affine technique, while for 4-bit integer quantization, it uses piecewise affine functions [Shen et al., 2020], which offer potentially greater precision at the cost of increased memory usage due to the additional parameters required per channel. We use the default group size (128), which regulates the granularity of the piecewise function.

Additionally, the **Transformers**[6] library from Hugging Face provided straightforward APIs for accessing pre-implemented and pre-trained models, including those mentioned in previous subsections. Complementing this, the **Datasets**[7] library facilitated access to benchmark datasets, streamlining the data loading and ingestion.

Lastly, to ensure reproducibility all intermediate QPs and results of each experiment were stored on Hugging Face. These resources are accessible through links provided in Appendix A, where a complete list of the software as well as their versions is also documented. This setup guarantees that the experiments can be replicated and verified by other researchers.

[3] https://www.macc.fccn.pt/resources
[4] https://www.runpod.io
[5] https://github.com/pytorch/ao
[6] https://huggingface.co/docs/transformers/index
[7] https://huggingface.co/docs/datasets/index

6

Development of PTNQ

Contents

This chapter presents a comprehensive analysis of the experiments conducted during the iterative refinement of the PTNQ technique. These experiments helped shape the final version of the method, detailed in Chapter 4, allowing us to incrementally enhance its performance.

In the previous chapter, we introduced the setup in which the experiments were conducted. This foundation ensures consistency and reproducibility across all tests presented here. Each section in this chapter delves into a specific experiment, beginning with a brief overview of its purpose and followed by a detailed analysis of the results, highlighting the key findings and how they influenced the technique.

By providing a detailed account of the experimental results, we aim to demonstrate the rationale behind each modification and how these iterative adjustments contributed to the overall effectiveness of the technique.

Finally, the chapter concludes with a summary with the most significant findings across all experiments. This summary synthesizes the key insights gained throughout the iterative process, providing a consolidated view of the improvements achieved and the current state of the method.

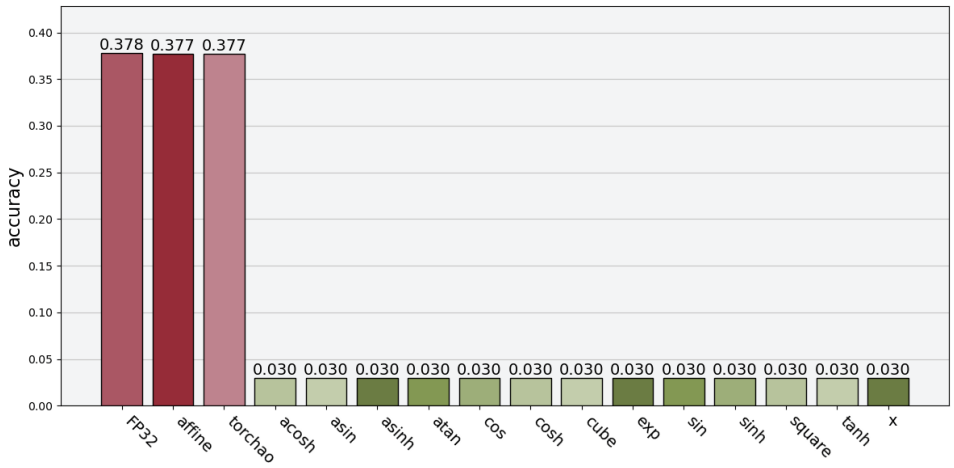


Figure 6.1: Results of applying naive initialization to QPs in TinyLlama for 8-bit quantization. The image shows that most quantization functions produced identical and poor accuracy outcomes.

6.1 Experiments

6.1.1 Simple Initialization

In the initial phase of our experimentation, we employed a naive approach to initializing the QPs. Specifically, we initialized all parameters either as ones or with random values.

This approach was chosen as a preliminary step to establish a baseline for our technique. The hypothesis was that this basic initialization would provide insight into the fundamental behavior of the quantization process without the influence of more sophisticated initialization methods. The main goal was to observe whether the naive initialization could yield functional results and, if not, to identify the underlying causes of any failures. These insights would then guide further optimizations to the method.

The quantization functions were tested with QPs initialized in the two ways described: either all ones or random values drawn from a normal distribution, with a mean of 0 and a variance of 1.

The results were largely negative. Nearly all experiments resulted in model accuracy dropping to 0, indicating that the quantized models were non-functional. In the few cases where the models did not yield a complete loss of accuracy, such as in the 8-bit quantization of TinyLlama [Zhang et al., 2024], the accuracy was low and presented an unusual pattern where all outcomes converged to the same value of 0.030, as shown in Figure 6.1.

Upon analyzing the results, we concluded that the weights were distorted during quantization. This was confirmed by observing the weight distributions after quantization and dequantization. For instance, as seen in Figure 6.2, the weights were either driven to 0 or failed to accurately represent their initial values after dequantization. Since nearly all weights were zero, all forward passes, independent of the non-linear function being used, yielded the same result, thus explaining the unusual accuracy value of

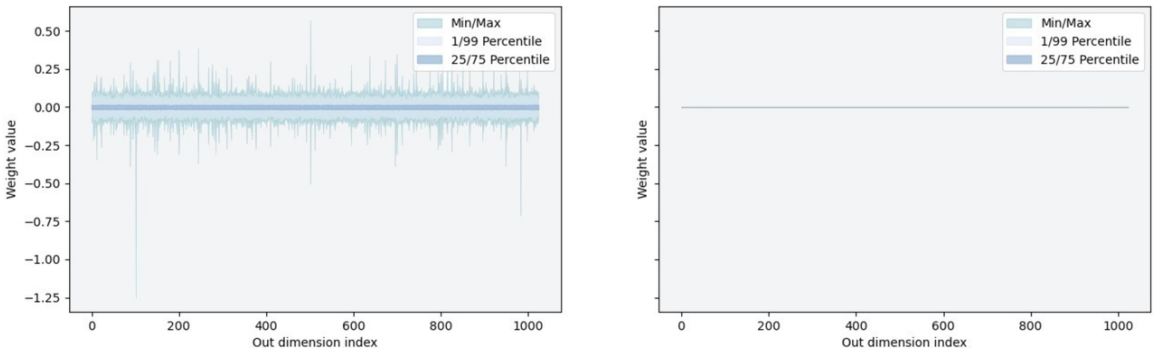


Figure 6.2: Distribution of weight values from a layer in ViT. Demonstrates the distortion post-quantization that converted them to zeros.

0.030 across the tested functions.

The distortion was traced back to the quantized values not fully spanning the entire range available for their bit-width, primarily due to improper initialization of the scaling parameter, $\_s$. Specifically, $\_s$ was initialized with values between $[-1, 1]$. When a non-linear function—typically yielding outputs in the same range—was applied and the result was multiplied by $\_s$, the resulting values were close to zero. These values were then rounded and clipped, causing most of them to be quantized to zero, leading to the observed distortion.

The naive initialization approach provided valuable insights. The key takeaway was that the initialization of the $\_s$ parameter is pivotal to the proper functioning of PTNQ.

6.1.2 Simple Initialization using Linear Scale

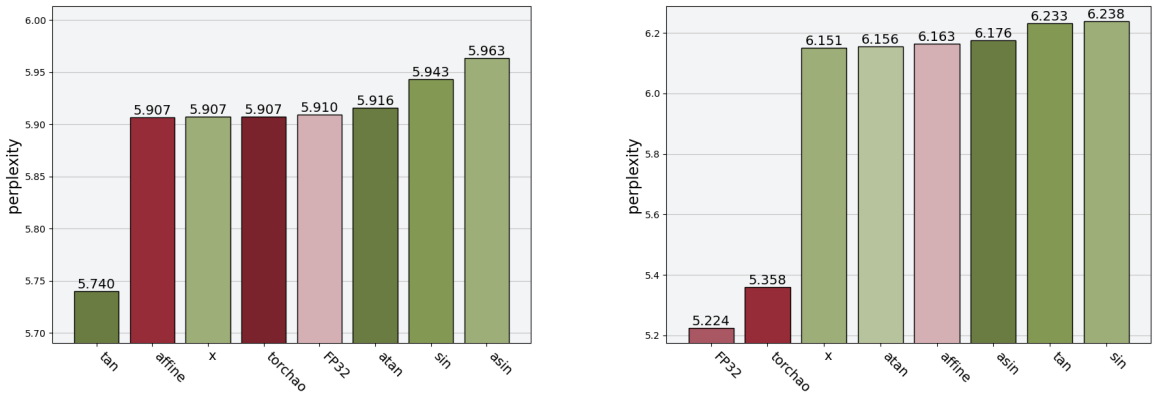
Following the results from our previous experiment, we explored an approach inspired by the calibration step in affine quantization. The primary motivation behind this experiment was to mitigate our previous issues where improper scaling led to significant weight distortions and a complete loss of model accuracy.

Recalling Section 4.3.1, in affine quantization, the scale parameter, S, represents the ratio between the range of the input data and the quantized output data, allowing us to map between the two ranges of values.

By adopting a calculated scale $\_s$ based on the above approach, we hypothesized that the quantized values would more effectively utilize the available bit width, thus preserving the integrity of the weight distributions and improving model performance.

To test this hypothesis, we conducted experiments where the parameter $\_a$ was initialized either as ones or as random values drawn from a normal distribution while the initialization of the $\_s$ parameter, would follow the usual formula used in affine quantization.

The results from these experiments demonstrated a significant improvement over the naive initial-



(a) Phi, 8 bits

(b) Llama3, 4 bits

Figure 6.3: Perplexity values for Phi (left) and Llama3 (right) across many different non-linear functions.

ization approach. For instance, when using 8-bit quantization, most models exhibited accuracies comparable to those achieved with affine quantization. As shown in Figure 6.3a, the Phi model achieved performance on par with other affine quantization methods.

As illustrated by Figure 6.3b, even in lower bit-widths, such as 4-bit quantization, the results remained close to the linear quantization baseline (labeled as "affine" in the figure). However, at this stage, there is still some discrepancy between the results of PTNQ and TorchAO.

Looking at Figure 6.4, we can also observe that the quantization process no longer completely distorts the original weights distribution.

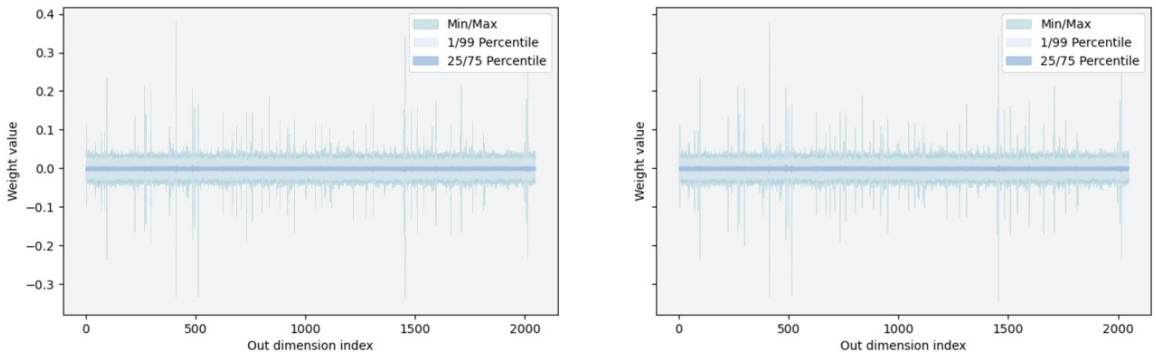


Figure 6.4: Distribution of weight values from a layer in TinyLlama. Weights are no longer distorted post-quantization.

These findings highlighted the critical role of the $\_s$ parameter in scaling the outputs of the non-linear functions to match the range of values supported by the bit width (e.g., $[-128, 127]$ for 8-bit quantization).

As such, we adopted this initialization strategy as a standard component of our PTNQ technique and, moving forward, the $\_s$ parameter will be consistently initialized using the method described above to ensure more reliable and effective quantization results.

6.1.3 Space Search

We conducted an experiment to evaluate the effectiveness of a technique that we named Space Search. The goal was to determine if this technique could optimize QPs better than previous initialization methods, thereby reducing quantization noise and improving model performance.

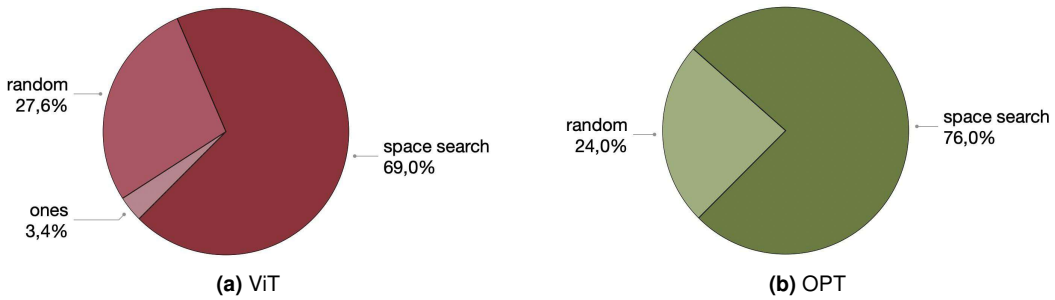
In previous trials, we observed that functions like $\log$, $\operatorname{arctanh}$, and $sqrt$ often produced zero accuracy when QPs were initialized either at random or as ones. This led us to believe that these initial values, typically confined within the range $[-1, 1]$, might not be optimal. For instance, if the best parameter value for a given function was closer to 100, it would be impossible to find it within this narrow range, leading to poor performance.

Our hypothesis was that expanding the QP search space to cover a wider range would improve performance, particularly for functions that previously performed poorly under standard initialization.

To test this, we developed Space Search, an algorithm designed to search for optimal QPs across a wide parameter space. Recalling from Section 4.3.3, the Space Search algorithm optimizes QPs by iteratively generating and evaluating random weight tensors. It progressively refines the search space based on the best-performing parameters, ultimately finding a set that minimizes quantization loss.

To test this hypothesis, the experiment applied Space Search to initialize the QPs, while the scale factors ($\_s$) remained initialized using the method from affine quantization.

The results of this experiment showed a marked improvement in the quantization performance for several functions that previously yielded suboptimal results. For instance, the $\operatorname{arctanh}$ function at 4-bit quantization in the Phi model, which previously achieved zero accuracy, now delivered improved performance. Furthermore, in a broader set of experiments, Space Search either matched or outperformed other initialization methods. This is evident in performance metrics for the ViT model (Figure 6.5b) and the OPT model (Figure 6.5a).



(a) ViT **(b)** OPT

Figure 6.5: Best QPs initialization methods across all bit-widths and all functions. Each section denotes the percentage of runs a QM with that particular initialization method was selected as the most suitable one.

Given these promising results, Space Search has been integrated into the PTNQ framework as a new initialization method. This addition not only enhances the reliability of the quantization but also expands its applicability to a wider range of functions, making PTNQ a more versatile solution.

6.1.4 Non-Linear Regression (NLR)

In this section, we explore the application of NLR to approximate QPs after initialization within our Post-Training Non-Linear Quantization (PTNQ) technique. This experiment aims to determine whether NLR-based curve fitting can effectively find better QPs that minimize quantization noise.

Since curve fitting adjusts free parameters (our QPs) to fit curves, we designed an experiment to investigate the hypothesis that we can leverage this approach to identify optimal QPs.

This hypothesis is grounded in the observation that, recalling from Section 4.3.4, the weights when sorted within a channel, resemble the shape of a tangent curve when plotted. By applying NLR to fit the curve of the quantized-dequantized weights to their original conterpart (see Figure 4.5), we can minimize the reconstruction error between the two and, at the same time, find a better approximation to the QPs.

To test this hypothesis, we performed an experiment where the initial values for the curve fitting algorithm (p_0 values) were obtained through the Space Search algorithm. This approach allows the NLR method to potentially converge faster by starting closer to the optimal solution. However, unlike the other parameters, the scale parameter, $\_s$, was initialized using the method from standard affine quantization.

The results of the experiment with NLR indicated that, in most cases, NLR provided better results when compared with using only Space Search. For example, when quantizing OPT, approximating through NLR outperformed Space Search with no approximation, as demonstrated in Figure 6.6.

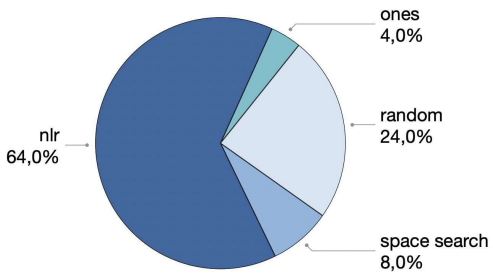


Figure 6.6: Best QPs initialization methods across all bit-widths and functions for OPT. Each section denotes the percentage of runs a QM with that particular initialization method was selected as the most suitable one.

However, the improvement was not uniform across all functions and models. The effectiveness of NLR was found to be context-dependent, varying between different models and quantization functions. For instance, as illustrated in Figure 6.7, when applying the $\mathrm{arcsinh}$ function to Phi quantized to 4 bits, the results were nearly identical regardless of whether NLR was used or not.

Based on these findings, we have integrated the NLR-based approximation into our suite of initialization methods within PTNQ.

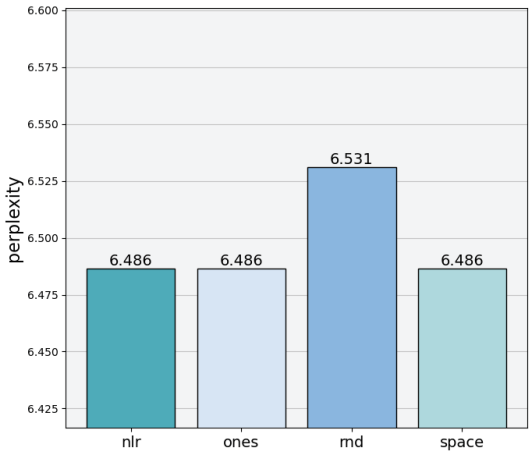


Figure 6.7: Perplexity results of different QPs initialization methods for $\mathrm{arcsinh}$ in Phi2 for 4 bits.

6.1.5 Train Quantization Parameters

In previous sections we explored various ways of initializing the QPs as a way of improving the model performance after quantization. In this section, we take a different approach. Inspired by some of the techniques used in QAT we conducted an experiment that assess whether training the QPs—initialized using the techniques discussed previously—could further minimize the quantization error.

Our hypothesis was that by training the QPs after their initialization, we could further optimize the quantization process, resulting in better overall model performance compared to using fixed QPs derived solely from the initialization methods. Specifically, the goal was to determine if this fine-tuning process could yield quantization parameters that lead to lower accuracy loss.

To test this hypothesis, we employed all available initialization strategies for the QPs. However, the scaling parameter $\_s$ was computed similarly to as in the affine quantization, as described in subsection 6.1.2. To avoid disrupting the initial loss value, we also initialize the LR as per Listing 4.4.

Training was performed using the Adam [Kingma and Ba, 2014] optimizer, which is well-suited for handling the non-linear optimization problem posed by the QPs fine-tuning. A LinearLR scheduler was employed to gradually decrease the learning rate over time, ensuring that the training process remained stable and that the QPs converged to values that minimized the quantization error.

The results indicated that layer-wise training of QPs generally led to improvements in model performance. For example, when quantizing OPT to 4 bits using the $\arcsin$ function, layer-wise training of QPs resulted in superior accuracy when compared to using only the initialized QPs (Figure 6.8).

However, the results were not universally positive across all scenarios. For some models like TinyLlama, training the QPs after initialization occasionally hindered performance and thus, other methods were preferred as can be seen in Figure 6.9. This suggests that in certain cases, the initial QPs provided a sufficiently good approximation, and further training introduced unnecessary noise, leading to a

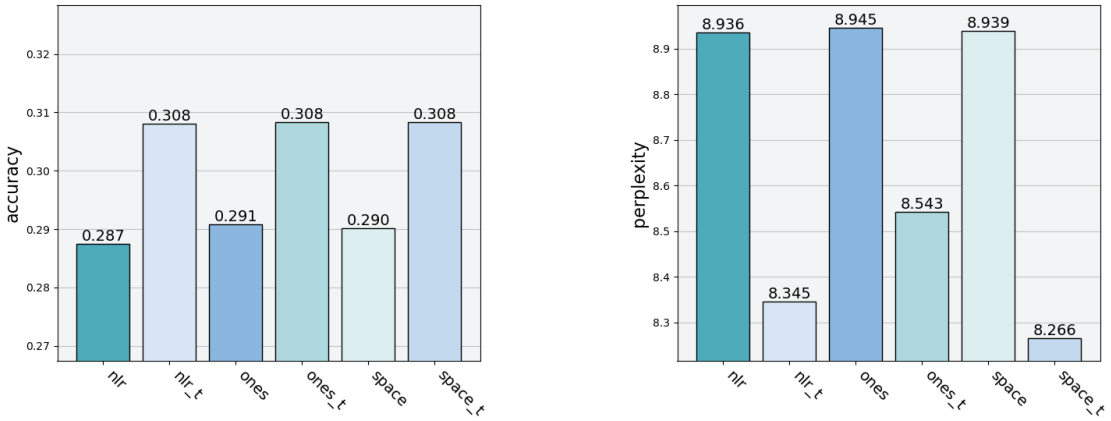
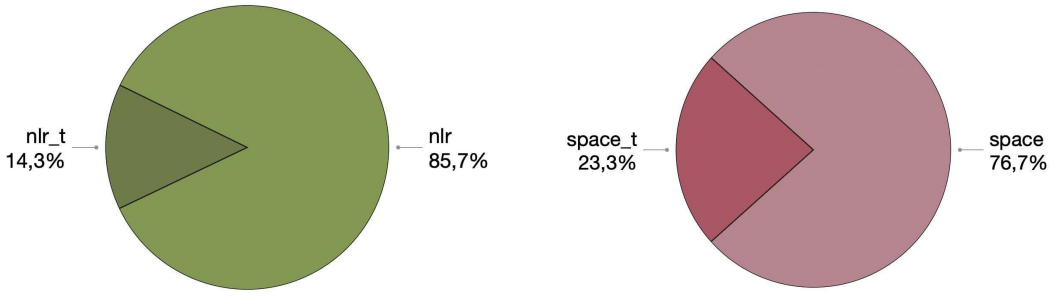


Figure 6.8: Accuracy (left) and perplexity (right) of $\arcsin$ when training the QPs in OPT and for 4 bits quantization. Bars suffixed with an "_t" represent trained variations.

decline in model accuracy.

Moreover, in some cases, PTNQ still could not yield better performance than methods that use piecewise affine functions such as TorchAO (for 4-bits). An example of this scenario is 4 bit quantization on larger models like Phi (Figure 6.10b). On the other hand, for smaller models like OPT, PTNQ resulted in better performance than TorchAO (Figure 6.10a).

We want to note that, at this stage, with the combination of initialization followed by QP training, we outperform traditional affine quantization, across all tested models and bit-widths, which demonstrates the robustness of PTNQ in various practical applications.

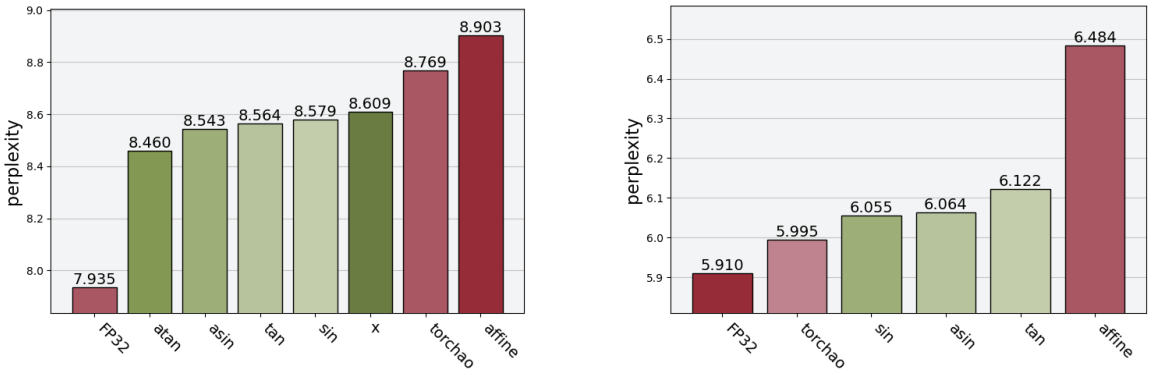


(a) NLR **(b)** Space Search

Figure 6.9: Comparison between training and not training in NLR (left) and Space Search (right) across all bit-widths for TinyLlama. Sections suffixed with an "_t" represent trained variations. Each section denotes the percentage of runs a QM with that particular initialization method (and optionally, training) was selected as the most suitable one.

Based on the results, we can conclude that training the QPs after initialization can significantly improve quantization outcomes in most cases. Consequently, we decided to incorporate this layer-wise QP training into our overall quantization technique, with the understanding that careful consideration must be given to the initialization method and model architecture. In particular, we recommend avoiding

unnecessary QP training in scenarios where initial QPs already yields good performance.



(a) OPT **(b)** Phi

Figure 6.10: Perplexity results of PTNQ when training QPs in OPT (left) and Phi (right), for 4 bits.

6.1.6 Evaluating Learning Rate Schedulers

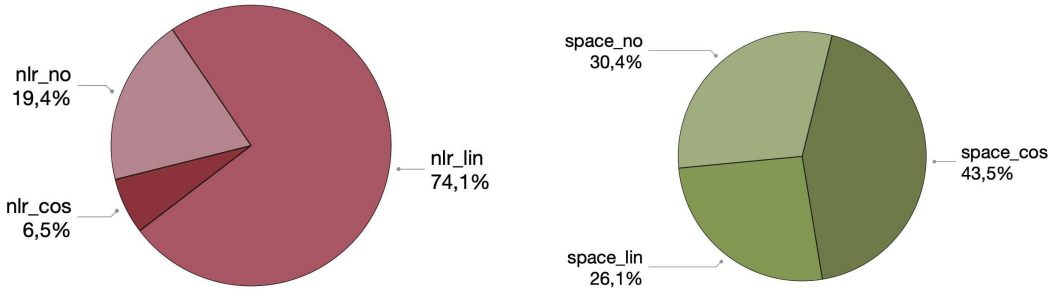
In order to optimize the process of training the QPs for our PTNQ technique, we conducted a series of experiments to identify the most effective LR scheduler. The purpose of these tests was to ensure that the training process for the QPs remains stable and converges towards an optimal solution.

The rationale behind focusing on LR schedulers stems from our initial setup, where the QPs are initialized using methods that provide already good approximations. However, if we use a learning rate during training that is too high, it can destabilize the learning process, causing the parameters to diverge. Therefore, the primary goal of this experiment was to identify the best LR scheduler configuration, hypothesizing that a well-tuned scheduler would enable the model to converge more effectively to an optimal solution.

To evaluate the impact of different LR scheduling strategies, we conducted experiments using three distinct schedulers: (1) no LR, (2) linear LR, and (3) cosine annealing with warm restarts.

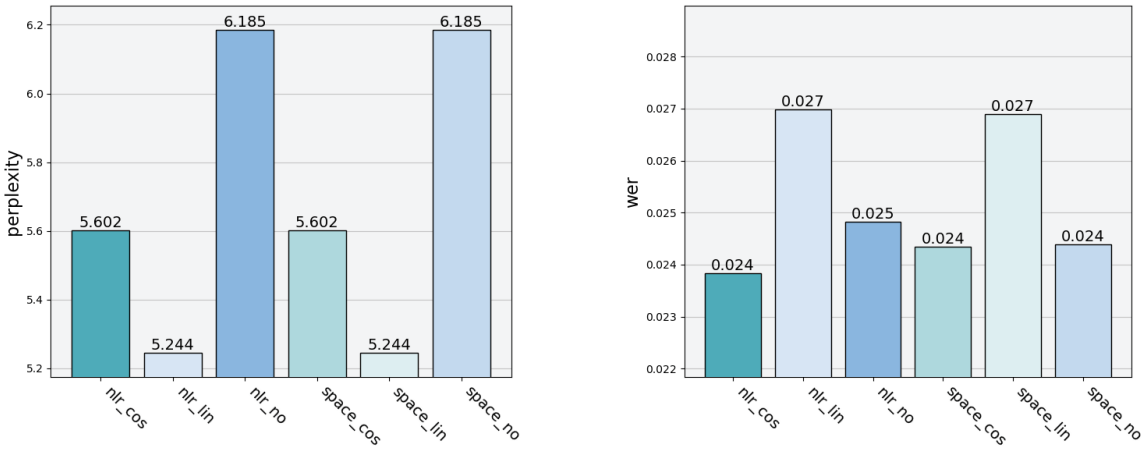
The results of these experiments demonstrated that the optimal LR scheduling approach varies depending on the model and the bit-width configuration. For instance, in the case of TinyLlama using NLR approximation, the LinearLR scheduler achieved the best performance, as shown in Figure 6.11a. On the other hand, for Wav2Vec using Space Search, Cosine Annealing with Warm Restarts outperformed the other methods, as illustrated in Figure 6.11b.

Additionally, the results indicated that even within the same non-linear function, the optimal LR scheduler could vary based on the model. For example, when quantizing Llama3 to 5 bits using the $\mathrm{arctanh}$ function, the best results were obtained with LinearLR, as depicted in Figure 6.12a. However, for Wav2Vec, also quantized to 5 bits using the same function, Cosine Annealing with Warm Restarts yielded better results, as shown in Figure 6.12b.



(a) TinyLlama **(b)** Wav2Vec

Figure 6.11: Best QPs initialization methods across all bit-widths for TinyLlama, considering only NLR, and Wav2Vec, considering only Space Search. Each section denotes the percentage of runs a QM with that particular lr scheduler was selected as the most suitable one.



(a) Llama3 **(b)** Wav2Vec

Figure 6.12: Results of quantizing Llama3 (left) and Wav2Vec (right) to 5 bits using the same function, $\mathrm{arctanh}$, with different LR schedulers.

The results of these experiments indicate that each LR scheduling approach has its own strengths, depending on the specific model and quantization configuration. Consequently, we decided to incorporate all of these LR scheduling options into our PTNQ technique.

6.1.7 Global Training

In this section, we explore the effects of globally training the quantization parameters QPs of a model, as opposed to the previously employed layer-wise training approach.

From our previous experiments, we demonstrated that layer-wise training of QPs has proven effective in minimizing the Mean Squared Error (MSE) between the original weights and the quantized-dequantized weights at each layer, thereby reducing quantization noise. However, this method does not account for the interactions between layers when the model is evaluated on its overall task performance.

The primary objective of this experiment was to determine whether global training could recover some of the accuracy typically lost during the quantization process by fine-tuning the QPs from a more holistic approach rather than focusing on individual layers of the model.

To test this hypothesis, we first performed the previously described layer-wise training and afterwards, the QPs were tuned by employing a technique similar to Quantization-Aware Training (QAT). During the forward pass of the model, we applied "fake quantization" to the weights prior to any linear operations, keeping them in full precision throughout the training process. The model was only fully quantized at the conclusion of the training.

A critical modification in this experiment was freezing the weights of the model and exclusively training the QPs. This ensured that the weight values remained static while allowing the QPs to be fine-tuned from a global perspective.

Because of budget constraints, the experiment did not include the Llama3 nor the Phi2 models, but the selected models provided sufficient insight into the impact of global training on QP optimization.

Table 6.1: Perplexity*/Accuracy†/WER‡ for each model per bit width. Comparison between global training and not doing global training.

Model	Baseline	Method	Bits				
			4	5	6	7	8
OPT*	7.935	No Train	8.197	8.017	7.770	7.886	7.872
		Train	**6.388**	**5.665**	**5.127**	**4.726**	**4.593**
TinyLlama*	6.596	No Train	7.418	6.898	6.469	6.653	6.564
		Train	**5.328**	**4.092**	**3.693**	**3.523**	**3.710**
ViT†	0.801	No Train	0.801	0.803	0.806	0.805	**0.804**
		Train	**0.804**	**0.807**	**0.810**	**0.813**	0.802
Wav2Vec‡	0.02395	No Train	0.02418	0.02327	**0.02328**	**0.02317**	**0.02290**
		Train	**0.02418**	**0.02327**	0.02340	0.02353	0.02355

The results of global training were notable. Across all tested models, performance metrics showed a clear improvement when compared to layer-wise QP training. As shown in Table 6.1, global training not only recovered some of the accuracy lost during quantization but in several instances even surpassed the baseline levels. These findings initially suggested that employing a global train after our initial layer-wise train might be a superior approach for fine-tuning the QPs.

However, a deeper analysis revealed that this performance improvement stemmed from an unintended consequence: the QPs, when trained globally, began to act as proxies for further training the weights of the model, effectively optimizing the model for the given task rather than merely enhancing the quantization technique. This phenomenon, while beneficial in terms of performance, diverges from the original intent of Post-Training Non-Linear Quantization (PTNQ), which aims to optimize quantization quality rather than task-specific accuracy through QP manipulation.

Based on these findings, we decided against incorporating global QP training into our final PTNQ technique, despite its apparent performance benefits.

For a more robust and generalizable quantization technique, it is preferable to continue training the model at full precision before applying quantization, rather than relying on global QP training as a substitute for proper model optimization since it will use less memory, because it won't have the overhead of storing the QPs, and will also have a higher degree of freedom since the weights provide a larger number of tunable parameters than the QPs.

We present these findings to inform the reader of the potential tradeoffs involved in global QP training and to underscore the importance of aligning the training objectives with the goals of the quantization technique.

6.2 Summary

6.2.1 Key Findings and Implications

In this study, we conducted a comprehensive set of experiments to evaluation and iteratively improve our Post-Training Non-Linear Quantization (PTNQ). This process revealed several critical insights into the factors that influence the effectiveness of our technique, highlight important considerations for optimizing quantization in different scenarios and also promote further possible improvements. The following key findings outline the implications of our research and suggest directions for future work.

6.2.1.A Importance of Scaling Parameter

The experiment conducted in subsection 6.1.2 underscore the critical role of the $\_s$ parameter in the PTNQ pipeline. Proper initialization of the $\_s$ parameter is pivotal for maintaining the integrity of weight distributions, ensuring effective quantization and improving model performance, making it a standard component of the PTNQ technique.

6.2.1.B Need for Extended Search Range for Non-Linear Functions

Certain non-linear functions, such as $\log$, $\operatorname{arctanh}$, and $sqrt$, require QPs that hold values outside the more conventional range of $[-1, 1]$. The Space Search technique, which extends the search for optimal QP values over a wider range, was found to be effective in improving the performance of these functions. This approach ensures that the QPs are better suited for the specific non-linear functions being used.

6.2.1.C Layer-Wise Training of QPs

Combining parameter initialization with layer-wise training of QPs consistently outperformed traditional affine quantization methods. This approach was found to be particularly effective in minimizing quantization error and enhancing overall model performance, making it one of the corner stones of our PTNQ technique.

6.2.1.D Matrix Size and Function Expressiveness

Our experiment in subsection 6.1.5 revealed an interesting relationship between model size and function expressiveness. Larger models, such as Llama3 and Phi2, exhibit different results compared to smaller models like OPT and ViT when subjected to our non-linear quantization technique. Specifically, PTNQ outperforms piecewise affine quantization in smaller models (OPT and ViT) but not in larger models (Llama3 and Phi2).

This suggests the existence of a critical matrix size N, at which the non-linear function reaches an inflection point in its ability to maintain the weight distribution without distortion. Beyond this point, piecewise affine quantization becomes more effective. Using piecewise functions essentially simulate non-linearity by applying individual quantization parameters to smaller sections of the tensor, which appears to better preserve the underlying distribution of larger matrices.

This finding highlights the need for adaptive quantization strategies that consider model size, specifically weight matrices size, and are capable of identifying the optimal size N for each individual function enabling the tensor to be optimally split into groups of size N. This approach would minimize the overhead of grouping while maximizing the expressiveness of the function, leading to more precise computation of quantization parameters for each group.

6.2.2 Limitations of the Study

While the findings of this study provide valuable insights into the reasoning behind many of the choices made in PTNQ as well as its effectiveness, there are several limitations that must be acknowledged. Understanding these constraints is crucial for interpreting the outcome of the study and identifying areas where further research is needed to enhance the robustness and applicability of the PTNQ method.

While our experimental setup from Chapter 5 aimed to incorporate a diverse array of model architectures and domains, the conclusions of the study are inherently constrained by the specific subset of models and functions evaluated.

To establish broader generalizability of PTNQ, additional models would have to be incorporated into our benchmarking system. This addition would serve to further validate the robustness and versatility of PTNQ across a more comprehensive spectrum of model typologies and compression scenarios.

6.2.3 Final Remarks

The PTNQ technique, as developed and refined through these experiments, offers an approach for improving the quantization of machine learning models, particularly in scenarios where traditional affine quantization methods may fall short.

The iterative refinement process demonstrated in this chapter, involving various initialization methods and layer-wise training of QPs, has significantly enhanced the effectiveness of the PTNQ technique.

By continuing to refine and adapt the PTNQ approach to different models and functions, there is potential to enhance the efficiency and accuracy of quantization in a variety of machine learning applications. Ongoing exploration in this area will be important for ensuring that PTNQ remains a valuable tool in the field of model optimizations.

7

Evaluation

Contents

In this chapter, we evaluate PTNQ and compare it against other quantization techniques. The goal is to provide a clear, objective analysis of the benefits and limitations of PTNQ, supported by quantitative data, while offering a thorough comparison with conventional techniques and highlighting the conditions under which PTNQ is most effective.

We explore, among other topics, whether PTNQ delivers measurable improvements in model accuracy and compression rates and we examine which non-linear functions are most frequently selected during the quantization process and how this selection impacts overall performance.

Table 7.1: Perplexity*/Accuracy†/WER‡ for each model per method and bitwidth.

Model	Baseline	Method	Bits				
			4	5	6	7	8
Llama3*	5.223	PTNQ	5.833	**5.181**	**5.163**	**5.205**	**5.202**
		affine	6.163	5.449	5.193	5.249	5.208
		torchao	**5.358**	–	–	–	5.208
OPT*	7.935	PTNQ	**8.197**	**8.017**	**7.770**	**7.886**	**7.872**
		affine	8.903	8.223	7.806	7.905	7.916
		torchao	8.768	–	–	–	7.917
Phi2*	5.910	PTNQ	**5.778**	**5.710**	**5.761**	**5.723**	**5.694**
		affine	6.484	5.969	5.950	5.903	5.907
		torchao	5.995	–	–	–	5.907
TinyLlama*	6.596	PTNQ	7.418	**6.898**	**6.469**	**6.653**	**6.564**
		affine	7.655	6.966	6.502	6.663	6.605
		torchao	**7.043**	–	–	–	6.606
ViT†	0.801	PTNQ	**0.801**	**0.803**	**0.806**	**0.805**	**0.804**
		affine	0.789	0.799	0.805	0.800	0.802
		torchao	0.795	–	–	–	0.802
Wav2Vec‡	0.02395	PTNQ	**0.02418**	**0.02327**	**0.02328**	**0.02317**	**0.02290**
		affine	0.02689	0.02373	0.02340	0.02340	0.02397
		torchao	crash	–	–	–	0.02397

7.1 End-to-end Accuracy

Table 7.1 shows the end-to-end accuracy/perplexity/WER (as appropriate) for each of the models and quantization techniques. We observe that PTNQ outperforms affine in every case. This is expected since we include an affine function in the pool, which will be picked in the cases where it is the best option. When compared with TorchAO, PTNQ outperforms in all but two cases (due to its piecewise implementation - refer to Section 5.6), while having less parameters and thus requiring less memory.

The key result is that we obtain an accuracy similar to baseline (FP32 weights) with only 4 to 6 bits. In contrast, affine functions need 6 to 8 bits to achieve the same results, except for the ViT model, where affine performs reasonably well with 4 bits.

In summary, PTNQ enables a 25% reduction in weight memory usage, on average, when compared to affine quantization methods.

7.2 Non-Linear vs Affine

Although we briefly touched on the comparison of results between affine quantization and PTNQ in Section 7.1, we would like to further highlight them.

In Figure 7.1, the empirical results demonstrate that PTNQ consistently outperforms affine quantiza-

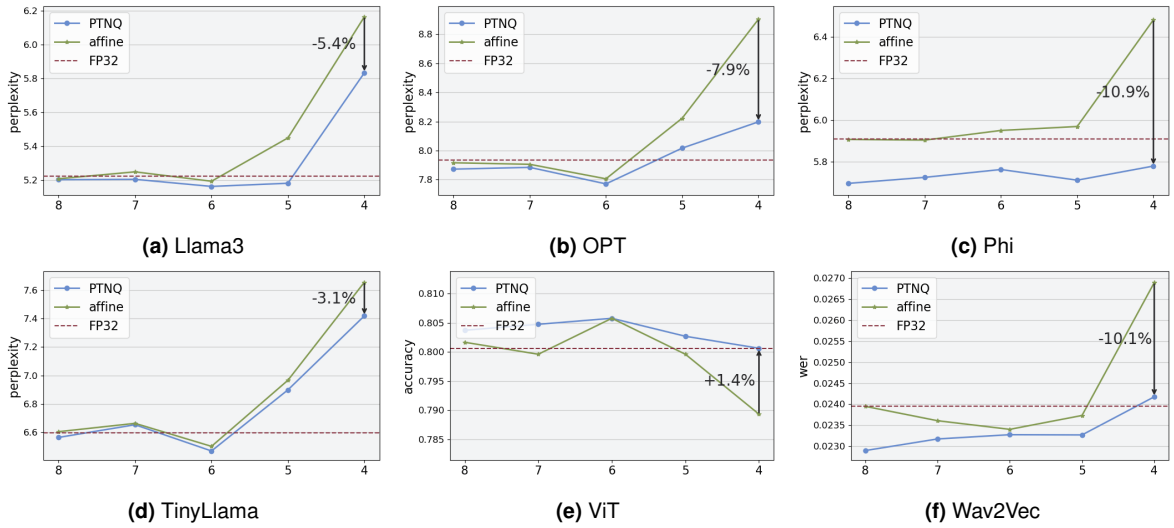


Figure 7.1: Comparison of the accuracy/perplexity/WER between using traditional affine quantization and PTNQ for different bit-widths. For PTNQ, the best QM for each bit-width is chosen.

tion across various models and bit-widths. For instance, at 4-bit precision, PTNQ achieves a significant reductions in perplexity of 10.9% for Phi2.

We can also observe in Figure 7.2 that, the advantage of PTNQ is most evident in extreme quantization scenarios, such as 4-bit quantization, where affine methods typically suffer significant increases in perplexity. PTNQ mitigates this by keeping the quantized-dequantize weights closely aligned with the original distribution, unlike affine quantization, which degrades quickly due to its simplified linear mapping.

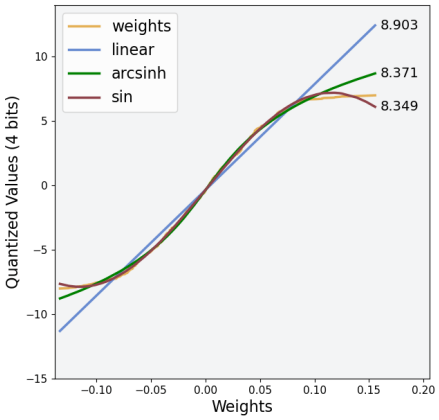


Figure 7.2: Sorted and scaled values of a channel of a weight of OPT. Comparison of perplexity between affine and non-linear approaches.

Alternative methods such as TorchAO attempt to compensate for the limitations of affine functions by using piecewise functions and adjusting group size parameters to simulate a non-linear fit. However, this simulation requires more parameters, increasing memory consumption and complexity. In contrast, PTNQ inherently supports non-linear transformations, providing a more efficient and accurate solution with lower memory usage.

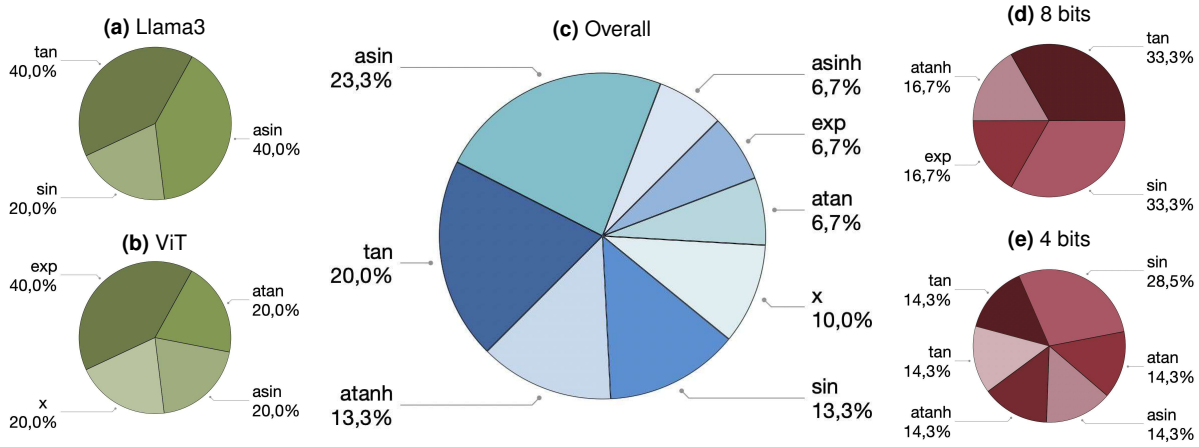


Figure 7.3: Distribution of the functions selected for quantization for particular models (4 to 8 bits), bit-widths (across all models), and overall. Each section denotes the percentage of runs a QM with that particular function was selected as the most suitable one.

7.3 Selected Functions for Quantization

We used a pool of 19 functions (refer to Chapter 5) for the experiments, from which PTNQ selects the best function for the whole model. The parameters of the function ($\_a$, $\_s$) are selected per channel.

We now investigate how many of these functions are used in practice and whether the usage varies across models. Figure 7.3 shows the distribution of the functions selected for quantization for two models (across all bit-widths), two bit-widths (across all models), and overall. We show the results for the best initialization mode for each combination of model and bit-width only. Each model was quantized from 4 to 8 bits.

The first thing we note is that only 8 functions are used out of the 19 in the pool. This includes the affine function, but which is used only for 10% of the cases, showing that the traditionally-used affine functions are usually not the best choice.

Another interesting observation is that the set of selected functions is very different across models and bit-widths. For example, for the ViT model we use a different function per bit-width, repeating only once. This means that it makes sense to instantiate PTNQ with a large pool of functions so it can select the best for each case since there is no one-size-fits-all function.

We now investigate what is the contribution of each of the 8 functions to the quantization performance. For example, is the accuracy between non-linear functions similar and thus we can shrink the pool of functions? What is the additional accuracy that each function brings?

Figure 7.4 shows the comparison between the 8 functions. The results refer to PTNQ with 4 bits, and to the best initialization method for each function. TinyLlama is very interesting: the best function ($\arcsin$) is over 8% better than the second best. On the other hand, $\arcsin$ is almost 14% worse than the best function for Phi2. We conclude that each function in the pool can have a substantial impact in the

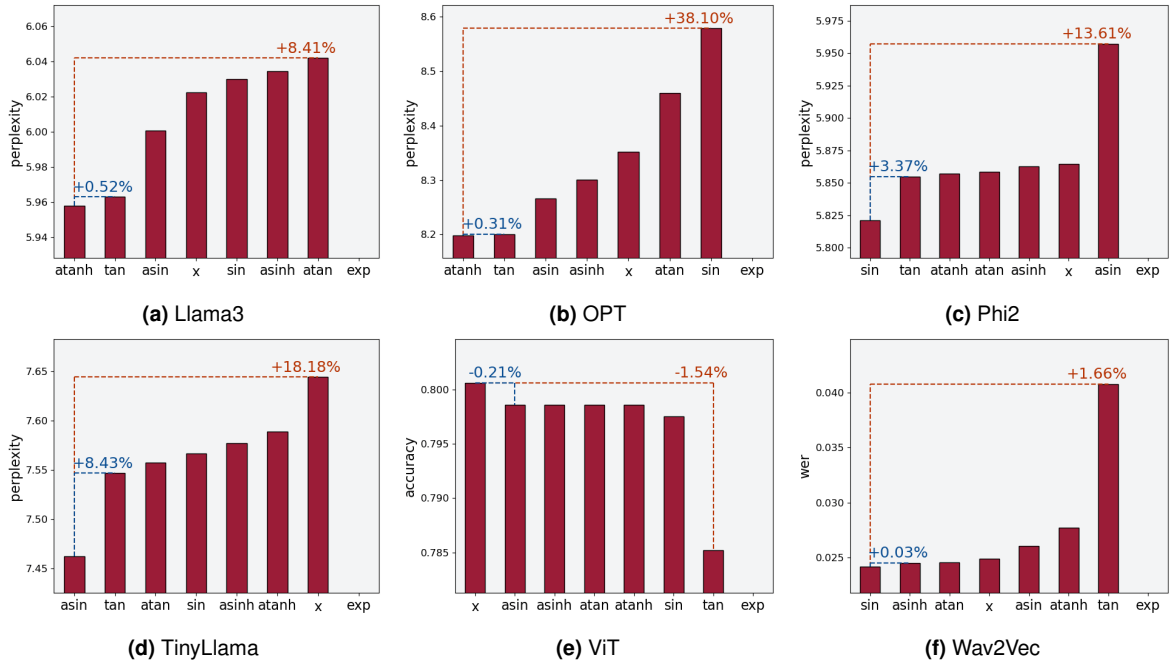


Figure 7.4: Comparison of the accuracy/perplexity/WER for 4-bit PTNQ between the best 8 functions. In blue, we mark the % of improvement between the best and second best functions. In orange, we mark the improvement against the worst function.

accuracy, and therefore the pool should be kept as large as possible.

We note that the exponential function did not yield meaningful results for 4 bits, as witnessed by the missing bars in Figure 7.4. However, it was the best function for almost 7% of the cases of higher bit-widths (Figure 7.3).

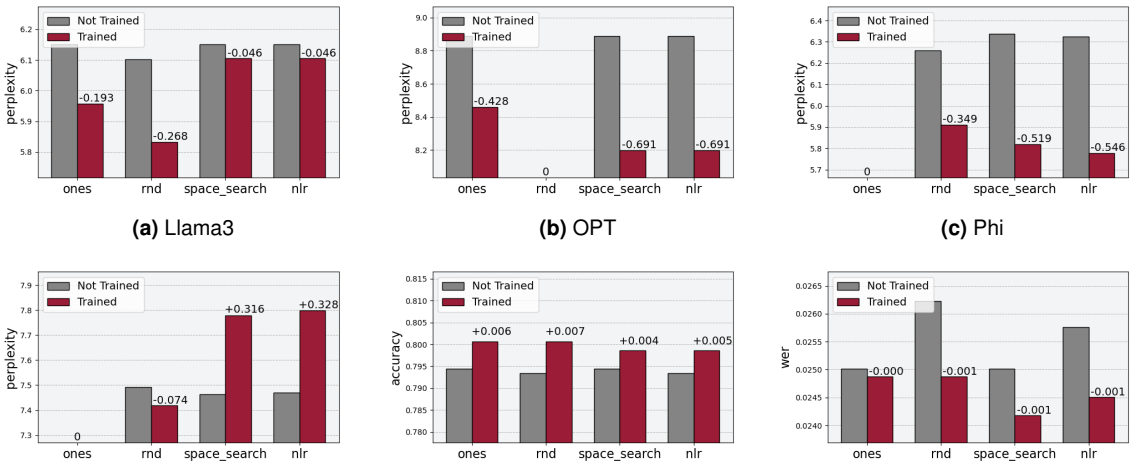
7.4 Impact of Initialization and Training Alternatives

We now investigate the impact of the initialization method on the accuracy, as well as whether the training step is beneficial. Figure 7.5 summarizes the results for 4-bit PTNQ. The 'nlr' column corresponds to the best of the three initialization methods followed by an approximation using non-linear regression.

In general, we observe that the training step is beneficial, with just one exception (TinyLlama). For the initialization method, all four are the best for some model. Differences in accuracy between different initialization methods are substantial, which suggests that trying multiple methods is a good strategy.

Additionally, the influence of each LR scheduler is shown in Figure 7.6, which reveals distinct preferences among the models. This figure presents the results for 4-bit quantization, with each column representing the optimal performance achieved for the specific training method across all QPs initialization methods.

Notably, half of the models benefit from using LinearLR, while the remaining half demonstrate im-



(a) Llama3 **(b)** OPT **(c)** Phi

(d) TinyLlama **(e)** ViT **(f)** Wav2Vec

Figure 7.5: Comparison of the accuracy/perplexity/WER for 4-bit PTNQ for different initialization methods with and without training. Empty values mean that the method did not result in acceptable performance.

Table 7.2: Inference time (ms) and memory (GB) for each model per quantization method with a batch size of 1.

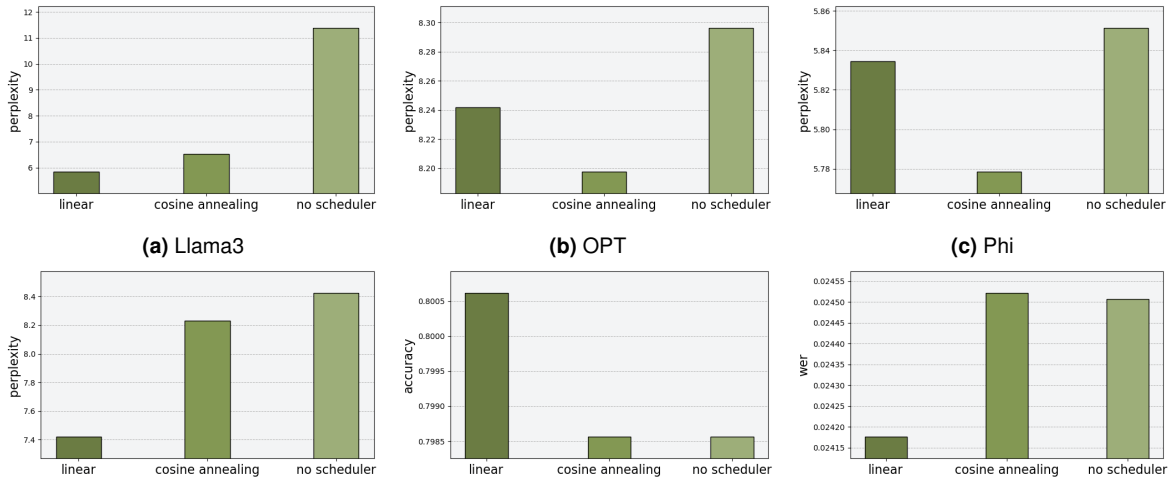
	PTNQ		Affine		torchao 4-bit		torchao 8-bit		Baseline (FP32)	
Model	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem
Llama3	144	13.5	96	13.9	39	5.4	52	11.8	27	32.2
OPT	45	0.6	21	0.7	17	0.3	15	0.6	10	1.3
Phi2	105	4.2	62	4.3	30	1.9	29	3.8	19	11.1
TinyLlama	41	1.8	20	1.9	22	0.7	19	1.6	15	4.4
ViT	40	0.4	19	0.4	13	0.2	16	0.3	10	1.2
Wav2Vec	36	0.6	21	0.6	crash	–	20	0.6	18	1.5

proved performance with CosineAnnealing. This indicates that the choice of LR scheduler is not only model-dependent but also relevant for maximizing the results of the training process.

7.5 Inference Performance and Memory Usage

Table 7.2 shows the time and memory usage for inference per query (batch size of 1). For PTNQ, we only show one set of numbers since they are the same for all bit-widths. This is because our prototype uses 8-bit integers always (setting the leftover bits to zero), rather than packing them more tightly for bit-widths lower than 8. Obviously, a production-ready implementation would pack the bits more tightly and would likely use custom kernels.

This data allow us to compare the performance between affine and non-linear functions, as well as to extrapolate the possible memory savings by comparing with TorchAO (which packs two 4-bit coefficients per byte).



(a) Llama3 **(b)** OPT **(c)** Phi

(d) TinyLlama **(e)** ViT **(f)** Wav2Vec

Figure 7.6: Comparison of the accuracy/perplexity/WER for 4-bit PTNQ for different lr schedulers used during training.

First, we observe that PTNQ is slower than using affine functions. Since we use functions that are less commonly used in ML models, this is to be expected. Frameworks, compilers, and even possibly the hardware, may need changes to bring the performance of PTNQ on par with affine functions.

Secondly, we note that the memory consumption of PTNQ and affine functions is roughly the same, despite PTNQ having better accuracy. On the other hand, TorchAO is obviously faster and consumes less memory since it has a production-quality implementation. We include the results for TorchAO as our implementation of affine functions is directly comparable with TorchAO 8-bits, thus enabling the extrapolation of results for PTNQ.

7.6 Quantization Time

Figure 7.7 shows the average quantization time (in minutes) for each model for a single quantization method (function and parameter initialization method). We further break down the time into the three main steps of PTNQ: initialization, approximation, and training.

Note that the figure shows the time for a single run of each model. In our experiments, we tested 19 functions, 4 initialization methods, 3 training methods, and 5 bit-widths yielding a total of 1140 runs per model. Nevertheless, these trials can all be run in parallel.

Table 7.3: Percentage of pruned runs over all total tests by model.

Model	**Llama3**	**Phi2**	**OPT**	**TinyLlama**	**ViT**	**Wav2Vec**
Percentage	8.9%	9.0%	9.4%	10.1%	11.0%	12.2%

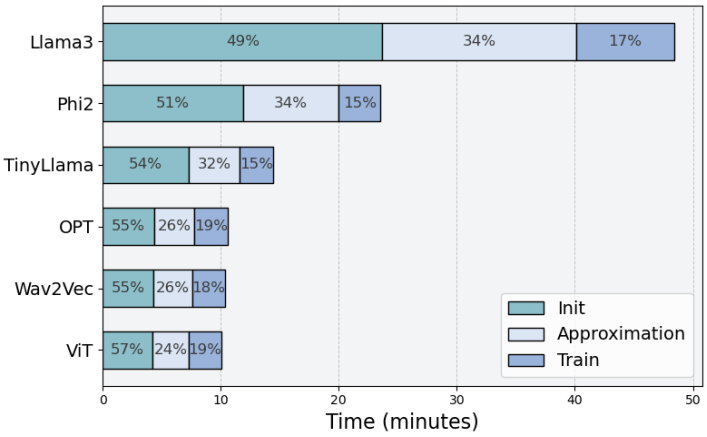


Figure 7.7: Quantization time per model and time per quantization step. Initialization of of p0 values; Approximation methods such as NLR; Training time. This is an average value over multiple runs

Table 7.4: Percentage of pruned runs over all total stops by function across all models.

Function	Percentage	Function	Percentage	Function	Percentage
square	11.5%	tanh	6.9%	tan	1.5%
cubert	11.0%	acos	5.0%	atanh	1.5%
cube	10.5%	exp	3.6%	lin	1.4%
log	10.3%	cosh	3.8%	sin	1.4%
cos	9.1%	sinh	3.8%	asinh	1.4%
sqrt	7.0%	asin	1.5%		
acosh	7.5%	atan	1.5%		

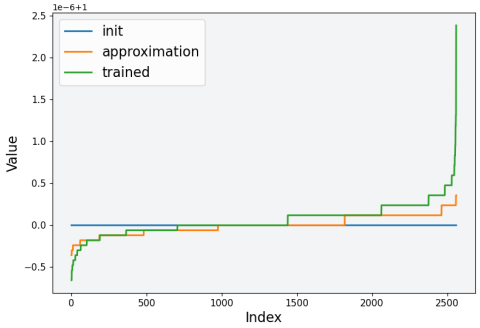
To reduce the used resources, we implemented an heuristic to prune early functions that showed no promise of working. Overall, 691 (10.1%) of the trial runs were cut short.

Among the models analyzed, Wav2Vec exhibited the highest rate of pruned runs at 12.2%, as detailed in Table 7.3. This indicates that the weight distributions of Wav2Vec are more challenging to encode, thus requiring more complex functions to effectively quantize them.

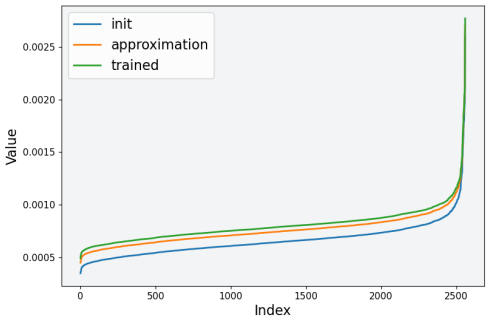
Trigonometric functions had the lowest pruning rate (as low as 1.4%). The functions that exceed 10% were: log, x^2, x^3, and $\sqrt[3]{x}$ (refer to Table 7.4). Perhaps unsurprisingly, as can be seen in Figure 7.3, these functions were never selected.

7.7 Impact of each PTNQ Phase on Quantization Parameters

Each phase of the PTNQ pipeline plays a role in fine-tuning the QPs. As shown in Figure 7.8, the evolution of parameters like $\_a$ (left) and $\_s$ (right) for the $\mathrm{arcsinh}$ function illustrates how every stage contributes to the final quantized model.



(a) Quantization parameter $\_a$.



(b) Quantization parameter $\_s$.

Figure 7.8: Evolution of QPs values across different stages of PTNQ for a layer in the Phi2 model, using the $\mathrm{arcsinh}$ function. The values are sorted to better illustrate the transformation.

PTNQ does not treat quantization as a static one-step process but instead adapts these parameters over multiple phases, allowing for a more fine-grained tuning that results in less quantization noise. This incremental tuning of QPs ensures that the final quantized-dequantized weights are better aligned with the original distribution after applying its corresponding non-linear function, reducing the performance degradation that often accompanies lower bit-width quantization.

However, while this progressive tuning provides clear improvements, a potential limitation lies in the complexity of this process. The multi-phase tuning introduces overhead in terms of computational time, which could be optimized further in future work. Additionally, it raises the question of whether simpler, faster methods might achieve similar results with fewer steps. Nonetheless, the current pipeline showcases a robust ability to handle the challenges of extreme quantization scenarios effectively.

8

Conclusion

Contents

In this thesis, we introduced PTNQ, a post-training quantization framework designed to efficiently compress deep learning models.

The key innovation of PTNQ lies in its use of non-linear quantization functions that can better capture complex weight distributions, resulting in a more precise representation with fewer bits, setting it apart from traditional affine quantization techniques.

PTNQ follows a pipeline whose focus is to search for the best non-linear quantization function from a pool and features many different techniques to better tune the non-linear transformations.

Our technique supports 3 different parameter initialization methods and 1 approximation method whose purpose is to provide a good initial starting point for their values. Additionally, it employs 3 different training methods that can be used to further adjust them and reduce the overall quantization noise.

In Chapter 7, we presented the results obtained from applying PTNQ to a range of models. Among these findings, two key observations stand out. First, PTNQ consistently achieves accuracy levels comparable to affine quantization methods, while reducing the bit-width by up to 50%. Second, model

performance is strongly influenced by the fidelity of the dequantized weights to the original values - the closer the match, the higher the resulting performance.

In conclusion, PTNQ proves effective in quantizing deep learning models through the use of non-linear functions. It represents a significant step towards achieving near-lossless compression by using more efficient, adaptive methods to encode model weights. With further improvements, we hope this technique will contribute to a future where AI models are universally accessible, empowering a wider array of users.

8.1 Future Work

While PTNQ has demonstrated promising results, as discussed in Section 6.2, our development process has uncovered several areas for potential improvement. In this section, we highlight these directions for future exploration.

The first improvement that should be tackled is applying distinct non-linear functions to individual layers or channels instead of a single function for the entire model. This idea arises from our findings that different layers can exhibit varying weight distribution characteristics, within the same model. Using a more fine-grained approach to function selection would provide a more carefully adjusted fit and thus, reduce the accuracy degradation caused by quantization. Transitioning to a more tailored approach will require modifications to our existing framework however, the implementation of layer-wise functions should be feasible and can significantly enhance the adaptability of PTNQ.

Additionally, our findings suggest the need for an adaptive approach to function expressivity based on matrix size. Identifying the critical matrix size N at which a given function begins to lose its effectiveness, when applied to a particular layer, is essential and can become a complex task. This endeavor will involve investigating the optimal way to partition larger weight matrices to maximize function effectiveness while minimizing overhead. Understanding how to balance these factors will be crucial and will likely demand significant experimentation and analysis.

Improving the inference speed is another key consideration. Developing custom kernels tailored to PTNQ can significantly accelerate inference. Further research into hardware extensions - whether at the kernel level or through dedicated acceleration units - will be necessary to fully exploit the potential of PTNQ in real-world applications. We propose creating kernels that are either agnostic to the specific function being used or optimized for the most used functions identified in Figure 7.3.

Furthermore, expanding our investigation to include a wider variety of non-linear functions, particularly more complex polynomials or hybrid transformations, could yield additional benefits. This research will necessitate rigorous testing and validation to ensure that new functions maintain the desired levels of accuracy and efficiency.

Bibliography

[Abramson et al., 2024] Abramson, J., Adler, J., Dunger, J., Evans, R., Green, T., Pritzel, A., Ronneberger, O., Willmore, L., Ballard, A. J., Bambrick, J., Bodenstein, S. W., Evans, D. A., Hung, C.-C., O'Neill, M., Reiman, D., Tunyasuvunakool, K., Wu, Z., Žemgulytė, A., Arvaniti, E., Beattie, C., Bertolli, O., Bridgland, A., Cherepanov, A., Congreve, M., Cowen-Rivers, A. I., Cowie, A., Figurnov, M., Fuchs, F. B., Gladman, H., Jain, R., Khan, Y. A., Low, C. M. R., Perlin, K., Potapenko, A., Savy, P., Singh, S., Stecula, A., Thillaisundaram, A., Tong, C., Yakneen, S., Zhong, E. D., Zielinski, M., Žídek, A., Bapst, V., Kohli, P., Jaderberg, M., Hassabis, D., and Jumper, J. M. (2024). Accurate structure prediction of biomolecular interactions with AlphaFold 3. *Nature*.

[Acosta et al., 2022] Acosta, J. N., Falcone, G., Rajpurkar, P., and Topol, E. (2022). Multimodal biomedical AI. *Nature Medicine*, 28.

[Ashkboos et al., 2024] Ashkboos, S., Mohtashami, A., Croci, M. L., Li, B., Jaggi, M., Alistarh, D., Hoefler, T., and Hensman, J. (2024). QuaRot: Outlier-free 4-bit inference in rotated LLMs. *arXiv preprint arXiv:2404.00456*.

[Athalye et al., 2018] Athalye, A., Carlini, N., and Wagner, D. (2018). Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. In *ICML*.

[Azevedo and Santos, 2022] Azevedo, P. and Santos, V. (2022). YOLO-based object detection and tracking for autonomous vehicles using edge devices. In *ROBOT*.

[Baevski et al., 2020] Baevski, A., Zhou, H., Mohamed, A., and Auli, M. (2020). wav2vec 2.0: A framework for self-supervised learning of speech representations.

[Batra et al., 2021] Batra, H., Punn, N. S., Sonbhadra, S. K., and Agarwal, S. (2021). BERT-based sentiment analysis: A software engineering perspective. In *DEXA*.

[Cabezas et al., 2024] Cabezas, D., Fonseca-Delgado, R., Reyes-Chacón, I., Vizcaino, P., and Morocho-Cayamcela, M. E. (2024). Integrating a LLaMa-based chatbot with augmented retrieval generation as a complementary educational tool for high school and college students. In *ICSOFT*.

[Cai et al., 2018] Cai, J., Takemoto, M., and Nakajo, H. (2018). A deep look into logarithmic quantization of model parameters in neural networks. In *IAIT*.

[Chaudhary et al., 2024] Chaudhary, D., Vadlamani, S. L., Thomas, D., Nejati, S., and Sabetzadeh, M. (2024). Developing a Llama-based chatbot for CI/CD question answering: A case study at Ericsson.

[Choquette, 2023] Choquette, J. (2023). NVIDIA Hopper H100 GPU: Scaling performance. *IEEE Micro*, 43(3):9–17.

[Chowdhery et al., 2024] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A. M., Pillai, T. S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., and Fiedel, N. (2024). PaLM: scaling language modeling with pathways. *J. Mach. Learn. Res.*, 24(1).

[Croci et al., 2022] Croci, M., Fasi, M., Higham, N. J., Mary, T., and Mikaitis, M. (2022). Stochastic rounding: implementation, error analysis and applications. *Royal Society Open Science*, 9(3):211631.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In *CVPR*.

[Dettmers et al., 2022] Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. (2022). LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *NeurIPS*.

[Dettmers et al., 2021] Dettmers, T., Lewis, M., Shleifer, S., and Zettlemoyer, L. (2021). 8-bit optimizers via block-wise quantization. *ArXiv*, abs/2110.02861.

[Devlin et al., 2019] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *NAACL*.

[Dosovitskiy et al., 2021] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale.

[Dubey and et al., 2024] Dubey, A. and et al. (2024). The Llama 3 herd of models.

[et al., 2023] et al., M. J. (2023). Phi-2: The surprising power of small language models. In *NeurIPS*.

[Frantar et al., 2022] Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. (2022). GPTQ: Accurate post-training compression for generative pretrained transformers. *arXiv preprint arXiv:2210.17323*.

[Gholami et al., 2022] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2022). *Low-Power Computer Vision*, chapter A Survey of Quantization Methods for Efficient Neural Network Inference.

[Gray and Neuhoff, 1998] Gray, R. M. and Neuhoff, D. L. (1998). Quantization. *IEEE Transactions on Information Theory*, 44(6):2325–2383.

[Guo et al., 2024] Guo, H., Brandon, W., Cholakov, R., Ragan-Kelley, J., Xing, E. P., and Kim, Y. (2024). Fast matrix multiplications for lookup table-quantized LLMs.

[Horowitz, 2014] Horowitz, M. (2014). 1.1 computing's energy problem (and what we can do about it). In *ISSCC*.

[Huang et al., 2017] Huang, L., Li, Y., Huang, J., and Liu, J. (2017). Snapshot ensembles: Train 1, get M for free. In *NIPS*.

[Hubara et al., 2016] Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks. In *NIPS*.

[Jacob et al., 2018] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., and Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *CVPR*.

[Jiang et al., 2024] Jiang, T., Xing, L., Yu, J., and Qian, J. (2024). A hardware-friendly logarithmic quantization method for CNNs and FPGA implementation. *J. Real-Time Image Process.*, 21(4).

[Kalamkar et al., 2019] Kalamkar, D. D., Mudigere, D., Mellempudi, N., Das, D., Banerjee, K., Avancha, S., Vooturi, D. T., Jammalamadaka, N., Huang, J., Yuen, H., Yang, J., Park, J., Heinecke, A., Georganas, E., Srinivasan, S. M., Kundu, A., Smelyanskiy, M., Kaul, B., and Dubey, P. K. (2019). A study of bfloat16 for deep learning training. *ArXiv*, abs/1905.12322.

[Kaplan et al., 2020] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *ArXiv*, abs/2001.08361.

[Kim et al., 2024] Kim, G., Lee, J., Park, S., Kwon, Y., and Kim, H. (2024). Mixed non-linear quantization for vision transformers.

[Kim et al., 2021] Kim, S., Gholami, A., Yao, Z., Mahoney, M. W., and Keutzer, K. (2021). I-BERT: Integer-only BERT quantization. In *ICML*.

[Kim et al., 2023] Kim, S., Hooper, C., Gholami, A., Dong, Z., Li, X., Shen, S., Mahoney, M., and Keutzer, K. (2023). SqueezeLLM: Dense-and-sparse quantization. *arXiv*.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

[Kovaleva et al., 2021] Kovaleva, O., Kulshreshtha, S., Rogers, A., and Rumshisky, A. (2021). BERT busters: Outlier dimensions that disrupt transformers.

[Krishnamoorthi, 2018] Krishnamoorthi, R. (2018). Quantizing deep convolutional networks for efficient inference: A whitepaper.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436–444.

[Li et al., 2023] Li, B., Chen, J., and Zhu, J. (2023). Memory efficient optimizers with 4-bit states. In *NeurIPS*.

[Li et al., 2024] Li, M., Huang, Z., Chen, L., Ren, J., Jiang, M., Li, F., Fu, J., and Gao, C. (2024). Contemporary advances in neural network quantization: A survey. In *IJCNN*.

[Li and Gu, 2023] Li, Z. and Gu, Q. (2023). I-ViT: Integer-only quantization for efficient vision transformer inference. In *ICCV*.

[Lin* et al., 2024] Lin*, Y., Tang*, H., Yang*, S., Zhang, Z., Xiao, G., Gan, C., and Han, S. (2024). QServe: W4A8KV4 quantization and system co-design for efficient LLM serving. *arXiv preprint arXiv:2405.04532*.

[Lin et al., 2022] Lin, Y., Zhang, T., Sun, P., Li, Z., and Zhou, S. (2022). FQ-ViT: Post-training quantization for fully quantized vision transformer. In *IJCAI*.

[Liu et al., 2023] Liu, S.-y., Liu, Z., Huang, X., Dong, P., and Cheng, K.-T. (2023). LLM-FP4: 4-bit floating-point quantized transformers. In *EMNLP*.

[Liu et al., 2024] Liu, Z., Zhao, C., Fedorov, I., Soran, B., Choudhary, D., Krishnamoorthi, R., Chandra, V., Tian, Y., and Blankevoort, T. (2024). SpinQuant–LLM quantization with learned rotations. *arXiv preprint arXiv:2405.16406*.

[Loshchilov and Hutter, 2017] Loshchilov, I. and Hutter, F. (2017). SGDR: Stochastic gradient descent with warm restarts. In *ICLR*.

[Merity et al., 2016] Merity, S., Xiong, C., Bradbury, J., and Socher, R. (2016). Pointer sentinel mixture models. *ArXiv*, abs/1609.07843.

[Micikevicius et al., 2017] Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaev, O., Venkatesh, G., and Wu, H. (2017). Mixed precision training. *CoRR*.

[Miyashita et al., 2016] Miyashita, D., Lee, E. H., and Murmann, B. (2016). Convolutional neural networks using logarithmic data representation.

[Nguyen et al., 2021] Nguyen, H. H., Ta, T. N., Nguyen, N. C., Bui, V. T., Pham, H. M., and Nguyen, D. M. (2021). YOLO based real-time human detection for smart video surveillance at the edge. In *ICCE*.

[Panayotov et al., 2015] Panayotov, V., Chen, G., Povey, D., and Khudanpur, S. (2015). Librispeech: An ASR corpus based on public domain audio books. In *ICASSP*.

[Pelgrom, 2010] Pelgrom, M. J. (2010). *Analog-to-digital conversion*. Springer Science & Business Media.

[Qian and Ren, 2023] Qian, C. and Ren, H. (2023). Deep reinforcement learning in surgical robotics: Enhancing the automation level.

[Radford et al., 2021] Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. (2021). Learning transferable visual models from natural language supervision. In *ICML*.

[Redmon et al., 2016] Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *CVPR*.

[Ren et al., 2023] Ren, F., Ding, X., Zheng, M., Korzinkin, M., Cai, X., Zhu, W., Mantsyzov, A., Aliper, A., Aladinskiy, V., Cao, Z., Kong, S., Long, X., Man Liu, B. H., Liu, Y., Naumov, V., Shneyderman, A., Ozerov, I. V., Wang, J., Pun, F. W., Polykovskiy, D. A., Sun, C., Levitt, M., Aspuru-Guzik, A., and Zhavoronkov, A. (2023). AlphaFold accelerates artificial intelligence powered drug discovery: efficient discovery of a novel CDK20 small molecule inhibitor. *Chem. Sci.*, 14:1443–1452.

[Rokh et al., 2023] Rokh, B., Azarpeyvand, A., and Khanteymoori, A. (2023). A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Trans. Intell. Syst. Technol.*, 14(6).

[Rombach et al., 2021] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. (2021). High-resolution image synthesis with latent diffusion models.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088):533–536.

[Shen et al., 2020] Shen, S., Dong, Z., Ye, J., Ma, L., Yao, Z., Gholami, A., Mahoney, M. W., and Keutzer, K. (2020). Q-BERT: Hessian based ultra low precision quantization of BERT. In *AAAI*.

[Smith et al., 2022] Smith, S., Patwary, M., Norick, B., LeGresley, P., Rajbhandari, S., Casper, J., Liu, Z., Prabhumoye, S., Zerveas, G., Korthikanti, V., Zhang, E., Child, R., Aminabadi, R. Y., Bernauer, J., Song, X., Shoeybi, M., He, Y., Houston, M., Tiwary, S., and Catanzaro, B. (2022). Using deepspeed and megatron to train Megatron-Turing NLG 530B, a large-scale generative language model.

[Sultan et al., 2024] Sultan, Y., Ma, J., and Liao, Y.-Y. (2024). Fine-tuning stable diffusion xl for stylistic icon generation: A comparison of caption size.

[van Baalen et al., 2023] van Baalen, M., Kuzmin, A., Nair, S. S., Ren, Y., Mahurin, E., Patel, C., Subramanian, S., Lee, S., Nagel, M., Soriaga, J., and Blankevoort, T. (2023). FP8 versus INT8 for efficient deep learning inference.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *NIPS*.

[Wang et al., 2022] Wang, L., Dong, X., Wang, Y., Liu, L., An, W., and Guo, Y. (2022). Learnable lookup table for neural network quantization. In *CVPR*.

[Wei et al., 2022] Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., and Fedus, W. (2022). Emergent abilities of large language models. *Transactions on Machine Learning Research*.

[Wu et al., 2020a] Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. (2020a). Integer quantization for deep learning inference: Principles and empirical evaluation. *ArXiv*, abs/2004.09602.

[Wu et al., 2020b] Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. (2020b). Integer quantization for deep learning inference: Principles and empirical evaluation. *ArXiv*, abs/2004.09602.

[Wu et al., 2023] Wu, X., Li, C., Aminabadi, R. Y., Yao, Z., and He, Y. (2023). Understanding INT4 quantization for language models: latency speedup, composability, and failure cases. In *ICML*.

[Xiao et al., 2023] Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. (2023). SmoothQuant: Accurate and efficient post-training quantization for large language models. In *ICML*.

[xiaoju ye, 2023] xiaoju ye (2023). calflops: a FLOPs and params calculate tool for neural networks in PyTorch framework.

[Yan et al., 2024] Yan, D., He, F., Cai, F., Jiang, M., and Li, J. (2024). A novel hardware acceleration method for deep neural networks. In *ICCIP*.

[Yao et al., 2022] Yao, Z., Aminabadi, R. Y., Zhang, M., Wu, X., Li, C., and He, Y. (2022). ZeroQuant: Efficient and affordable post-training quantization for large-scale transformers. *ArXiv*, abs/2206.01861.

[Zhang et al., 2024] Zhang, P., Zeng, G., Wang, T., and Lu, W. (2024). TinyLlama: An open-source small language model.

[Zhang et al., 2022] Zhang, S., Roller, S., Goyal, N., Artetxe, M., Chen, M., Chen, S., Dewan, C., Diab, M., Li, X., Lin, X. V., Mihaylov, T., Ott, M., Shleifer, S., Shuster, K., Simig, D., Koura, P. S., Sridhar, A., Wang, T., and Zettlemoyer, L. (2022). OPT: Open pre-trained transformer language models.

A

Evaluation Sources

This appendix provides links to all models, datasets, and software libraries used throughout this research. It is designed to ensure the reproducibility of the experiments presented in this work and provide proper references to external resources.

A.1 Models

All machine learning models used in this study were obtained from **Hugging Face**[1]. Table A.1 lists the models used, their parameters, and the respective access links.

Table A.1: List of models used in the experiments.

Model	Parameters	Link
Llama3 [Dubey and et al., 2024]	8.0B	Llama3 on Hugging Face
OPT [Zhang et al., 2022]	350.0M	OPT on Hugging Face
TinyLlama [Zhang et al., 2024]	1.1B	TinyLlama on Hugging Face
Wav2Vec [Baevski et al., 2020]	317.0M	Wav2Vec on Hugging Face
Phi2 [et al., 2023]	2.7B	Phi2 on Hugging Face
ViT [Dosovitskiy et al., 2021]	307.0M	ViT on Hugging Face

[1] https://huggingface.co/

These models were chosen to cover various domains, including natural language processing, audio recognition, and image processing, demonstrating the generalization of the Post-Training Non-Linear Quantization (PTNQ) technique.

A.2 Datasets

To test the models effectively, we employed three distinct datasets, each specialized for different tasks. Table A.2 contains references to all datasets.

Table A.2: Datasets used in experiments.

Dataset	Split	Domain	Link
WikiText	Validation	Natural Language Processing	WikiText on Hugging Face
ImageNet	Validation	Image Classification	ImageNet on Hugging Face
LibriSpeech	Test (cleaned)	Speech Recognition	LibriSpeech on Hugging Face

Minimal preprocessing was applied to ensure compatibility with the requirements of each model, and standard validation splits were used for consistent evaluation across experiments.

A.3 Results of Experiments

All intermediate results, including Quantization Parameters (QPs) and outcomes of all tests were stored on Hugging Face. They are publicly accessible through the links found in Table A.3.

Table A.3: Results of all experiments done during the development and evaluation of PTNQ.

Experiment	Results Link
Simple Initialization	Simple Initialization Results
Simple Initialization using Linear Scale	Simple Initialization using Linear Scale Results
Space Search	Space Search Results
Non-Linear Regression (NLR)	Non-Linear Regression (NLR) Results
Train Quantization Parameters	Train Quantization Parameters Results
Evaluating Learning Rate Schedulers	Evaluating Learning Rate Schedulers Results
Global Training	Global Training Results

These links provide all the configurations and necessary files to replicate the tests used to develop and evaluate the technique. In the links, it is also possible to find logs, final results and more images and tables to further explore each individual experiment.

A.4 Software Versions

The experiments were conducted using specific versions of the software tools listed in Table A.4. The versions used are necessary to replicate the experiments.

Table A.4: Software versions used during the experiments.

Software	Version	Software	Version
python	3.12	torchviz	0.0.2
scipy	1.10.1	torch	2.2.2
datasets	2.16.0	torchvision	0.17.2
evaluate	0.4.0	torchaudio	2.2.2
scikit-learn	1.2.2	cuda	12.1
matplotlib	3.7.2	torchao	0.3.1
transformers	4.38.2	torchmetrics	1.3.2
sympy	1.12	sympytorch	0.1.4

These libraries provided the framework and tools to implement and compare PTNQ with other quantization approaches, ensuring the reproducibility and applicability of the results.