Matryoshka Quantization

Pranav Nair[*,1], Puranjay Datta[*,1], Jeff Dean[1], Prateek Jain[1] and Aditya Kusupati[1]
[1]Google DeepMind, [*]Equal contribution

Quantizing model weights is critical for reducing the communication and inference costs of large models. However, quantizing models – especially to low precisions like int4 or int2 – requires a trade-off in model quality; int2, in particular, is known to severely degrade model quality. Consequently, practitioners are often forced to maintain multiple models with different quantization levels or serve a single model that best satisfies the quality-latency trade-off. On the other hand, integer data types, such as int8, inherently possess a nested (Matryoshka) structure where smaller bit-width integers, like int4 or int2, are nested within the most significant bits. This paper proposes Matryoshka Quantization (MatQuant**), a novel multi-scale quantization technique that addresses the challenge of needing multiple quantized models. It allows training and maintaining just one model, which can then be served at different precision levels. Furthermore, due to the co-training and co-distillation regularization provided by** MatQuant**, the int2 precision models extracted by** MatQuant **can be up to** 10% **more accurate than standard int2 quantization (using techniques like QAT or OmniQuant). This represents significant progress in model quantization, demonstrated by the fact that, with the same recipe, an int2 FFN-quantized Gemma-2 9B model is more accurate than an int8 FFN-quantized Gemma-2 2B model.**

1. Introduction

Due to their impressive performance, there is a strong push to deploy deep learning models, particularly large language models (LLMs) (Achiam et al., 2023; Dubey et al., 2024; G Team et al., 2024) in a large number of scenarios. Due to auto-regressive nature of LLMs, decode latency tends to dominate inference cost. Decode latency itself is dominated by communication cost of transferring model weights from high-bandwidth memory (HBM) to the SRAM or due to transferring weights/activations in a distributed cluster.

Quantizing weights and/or activations can significantly reduce the overall communication load and is, therefore, one of the most popular techniques for reducing inference costs (Dettmers et al., 2022). While floating-point representations are standard for training, integer data types such as int8, int4, and int2 are appealing alternatives for inference. However, current methods for quantizing to these varying integer precisions typically treat each target precision as an independent optimization problem, leading to a collection of distinct models rather than a single, versatile one. Furthermore, quantizing to extremely low precisions like int2 is known to be highly inaccurate.

In this work, we pose the question of whether both of the above challenges can be addressed; that is, can we train a single model from which we can extract multiple accurate lower-precision models? We answer this question in the affirmative by introducing Matryoshka Quantization (MatQuant), a novel multi-scale training method that leverages the inherent nested (Matryoshka) structure (Kusupati et al., 2022) within integer data types (Figure 1a). Specifically, *slicing* the top bits of an int8-quantized weight can directly yield an int4 or int2 model. Existing quantization techniques often neglect this structure, which limits the potential for multi-scale adaptable models operating at various bit-widths with optimal performance.

Instead, MatQuant simultaneously optimizes model weights across multiple precision levels (e.g., int8, int4, int2). At a high level, we represent each model parameter at different precision levels using shared most significant bits (MSBs), and then jointly optimize the loss for each precision level. This allows us to develop a single quantized model that can effectively operate at any of the chosen bit-widths, offering a spectrum of accuracy-versus-cost options. MatQuant is a general-purpose technique, applicable to most

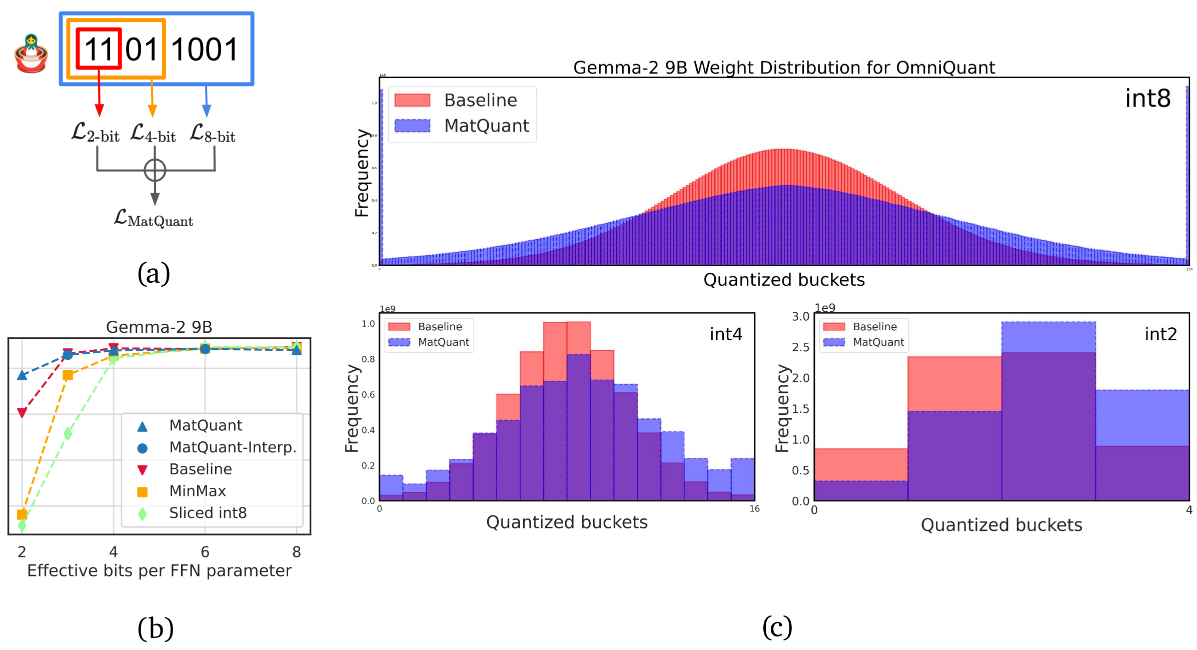

Figure 1 | (a) MatQuant is a multi-scale quantization training technique using the inherent Matryoshka structure of int8 → int4 → int2. (b) Empirical gains of MatQuant on downstream tasks, especially > 8% for int2, on Gemma-2 9B with OmniQuant. (c) The right-shifted quantized weight distribution as a consequence of MatQuant's training mechanism that maximises accuracies across all precisions.

learning-based quantization methods, such as Quantization Aware Training (QAT) (Jacob et al., 2018) and OmniQuant (Shao et al., 2023).

We demonstrate the efficacy of MatQuant when applied to quantizing the Feed-Forward Network (FFN) parameters of standard LLMs (Gemma-2 2B, 9B, and Mistral 7B) (Vaswani et al., 2017) – typically, FFN is the main latency block hence the focus on improving the most significant component's latency. Our results show that MatQuant produces int8 and int4 models with comparable accuracy to independently trained baselines, despite the benefit of shared model parameters. Critically, the int2 models generated by MatQuant significantly outperform their individually trained counterparts, with 8% higher accuracy on downstream tasks (Figure 1b). We also extend MatQuant to quantize all weights of a Transformer layer. Finally, we find that quantizing with MatQuant shifts the quantized weight distribution toward higher values, contributing to improved int2 performance (Figure1c).

Beyond improving chosen precision performance, MatQuant allows for seamless extraction of interpolative bit-widths, such as int6 and int3. MatQuant also admits a dense accuracy-vs-cost pareto-optimal trade-off by enabling layer-wise Mix'n'Match of different precisions. This ensures deployment of say an effective int3 sized model even if the underlying hardware only supports int4 and int2. Overall, MatQuant and its variants present a significant step toward developing multi-scale models with high flexibility and performance, pushing the boundaries of low-bit quantization for efficient LLM inference.

2. Related Work

Model weight quantization is an extremely powerful and prevalent technique for making resource-intensive neural networks suitable for deployment constraints – especially modern-day LLMs. Quantization algorithms can be categorized as either learning-free or learning-based. Learning-free methods use limited data to calibrate model parameters without relying on gradient descent. Learning-based methods, however, utilize gradient descent to update either model parameters or auxiliary parameters to aid in quantization.

Learning-free Quantization Methods. Naive quantization methods, such as MinMax, absmax, and zero-point quantization, aim to directly map the range of model weights to the target bit-width – see (Dettmers et al., 2022) for a detailed background. Dettmers et al. (2022) further improved this by identifying the need to handle outliers with higher precision than the rest of the model weights. The core principle of more recent learning-free quantization methods remains similar while improving various aspects of it and using small amounts of data for calibration. For example, GPTQ (Frantar et al., 2022) improves upon min-max quantization by iterating over all the coordinates, quantizing them one at a time, and updating the remaining full-precision coordinates to minimize the layer-wise activation reconstruction error. AWQ (Lin et al., 2023), SmoothQuant (Xiao et al., 2023), and AffineQuant (Ma et al., 2024) scale the weights and activations to reduce outliers, thus making them easier to quantize. QuIP (Chee et al., 2024), FrameQuant (Adepu et al., 2024), and QuaRoT (Ashkboos et al., 2024) multiply the weights and activations by orthonormal matrices before quantizing to reduce the number of outliers. SqueezeLLM (Kim et al., 2024) uses clustering to obtain the optimal buckets for quantization, and CDQuant (Nair and Suggala, 2024) improves upon GPTQ by greedily choosing the coordinates to descend along. While learning-free methods are inexpensive and work well at higher bit-widths, they are often suboptimal in the low-precision regime, which benefits greatly from learning-based techniques.

Learning-based Quantization Methods. Quantization Aware Training (QAT) (Abdolrashidi et al., 2021; Jacob et al., 2018) is a logical approach to ensure that models are easy to quantize during inference while retaining high accuracy. However, because QAT involves updating all the model parameters, its adoption for LLMs has been limited. Several recent works improve the performance and efficiency of QAT. LLM-QAT (Liu et al., 2024a) and BitDistiller (Du et al., 2024) enhance QAT with knowledge distillation from the full-precision model. EfficientQAT (Chen et al., 2024) minimizes the block-wise reconstruction error before performing end-to-end training. This significantly reduces the time it takes for QAT to converge. On the other hand, some techniques significantly reduce the overhead by learning only the auxiliary parameters, such as scaling factors and zero-points, that aid in quantization instead of updating the actual weight matrices. For example, OmniQuant (Shao et al., 2023) does not update the model parameters; instead, it learns additional scales and shifting parameters (that aid with quantization) through gradient descent over the block-wise reconstruction error and achieves better accuracy than most QAT techniques. Likewise, SpinQuant (Liu et al., 2024b) uses gradient descent to learn its rotation matrices. This class of learning-based quantization techniques (OmniQuant, SpinQuant, etc.) is widely adopted due to their appeal of achieving QAT-level accuracy at a fraction of the cost.

Multi-scale Training. Training across multiple data scales (resolutions) was heavily popularized in computer vision for both recognition and generation (Adelson et al., 1984; Denton et al., 2015; Lin et al., 2017). More recently, the paradigm of multi-scale training has shifted to models (Devvrit et al., 2023; Kusupati et al., 2022; Rippel et al., 2014; Yu et al., 2018), where the data remains the same, and models of varying capacity, all nested within one large model, are trained jointly. This joint, nested (Matryoshka-style) learning with varying model sizes results in a smooth accuracy-vs-compute trade-off and is beneficial in many downstream applications and real-world deployments. However, the most obvious structure with a nested nature is the bit structure of the integer data type. Given the success of multi-scale training for inputs, outputs, and model weights, it is imperative to explore it further for integer data types, especially in the context of quantization, which aids in the deployment of resource-intensive LLMs.

3. Matryoshka Quantization

We introduce MatQuant, a general-purpose, multi-scale training technique that works seamlessly with popular learning-based quantization methods such as Quantization Aware Training (QAT) (Jacob et al., 2018) and OmniQuant (Shao et al., 2023). As long as the model or auxiliary parameters are optimized with gradient descent, MatQuant's multi-scale training technique can be used across chosen bit-widths, leveraging the inherent nested structure of integer data types. In this section, we will elaborate on the preliminaries behind QAT and OmniQuant, alongside our novel proposed approach, MatQuant.

3.1. Preliminaries

3.1.1. Quantized Aware Training

Quantized Aware Training (QAT) learns a c-bit quantized model by optimizing for the end-to-end cross entropy loss using gradient descent. It uses the quantized weights for the forward pass and a straight through estimator (STE) (Bengio et al., 2013) to propagate gradients through the quantization operator during the backward pass.

To mathematically formulate QAT, we define MinMax quantization of a real-valued vector w in c bits as follows:

$$Q_{\text{MM}}(w, c) = \text{clamp}\left(\left\lfloor \frac{w}{\alpha} + z \right\rceil, 0, 2^c - 1\right)$$
$$\alpha = \frac{\max(w) - \min(w)}{2^c - 1}, \quad z = -\frac{\min(w)}{\alpha} \tag{1}$$

where $Q_{\text{MM}}(w, c)$ is the c-bit quantized version of w, α is the scaling factor and z is the zero point.

Let W_F represent weights of a Transformer LLM and let $\mathcal{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$ be a labelled dataset where x_i and y_i represent the input and output respectively. With L_{CE} as the cross entropy loss, the optimization of QAT is:

$$\min_{W_F} \frac{1}{N} \sum_{i \in [N]} \mathcal{L}_{\text{CE}}\left(F(x_i; Q_{\text{MM}}(W_F, c)), y_i\right) \tag{2}$$

where $F(\cdot)$ represents the LLM's forward pass.

3.1.2. OmniQuant

OmniQuant, unlike QAT, does not update the model parameters. Instead, it learns additional scaling and shifting parameters through gradient descent over layer-wise L2 error reconstruction. These auxiliary parameters aid with quantization. Similar to QAT, OmniQuant also uses a straight through estimator during optimization. However, unlike QAT, OmniQuant operates with limited data, making it much more attractive for resource-scarce settings.

OmniQuant adds two learnable scales, γ and β, to MinMax quantization as follows:

$$Q_{\text{Omni}}(w, c) = \text{clamp}\left(\left\lfloor \frac{w}{\alpha} + z \right\rceil, 0, 2^c - 1\right)$$
$$\alpha = \frac{\gamma \cdot \max(w) - \beta \cdot \min(w)}{2^c - 1}, \quad z = -\frac{\beta \cdot \min(w)}{\alpha} \tag{3}$$

OmniQuant also adds another set of learnable shifting and scaling parameters to the FFN's affine projections as follows:

$$XW + b \longrightarrow ((X - \delta) \oslash s) \cdot Q_{\text{Omni}}(W \odot s) + b + \delta \cdot W \tag{4}$$

where $X \in \mathbb{R}^{n \times d}$ is the input to the affine transformation, $W \in \mathbb{R}^{d \times d_o}$ is the linear projection associated with the affine transformation, $b \in \mathbb{R}^{d_o}$ is the bias vector, $\delta \in \mathbb{R}^d$ and $s \in \mathbb{R}^d$ are learnable shift and scale parameters respectively.

With the goal of optimizing the layer-wise L2 error (where a layer consists of an Attention block followed by an FFN block), OmniQuant's overall objective can be portrayed as follows:

$$\min_{\gamma, \beta, \delta, s} ||F_l(W_F^l), X_l) - F_l(Q_{\text{Omni}}(W_F^l), X_l)||_2^2 \tag{5}$$

where $F_l(\cdot)$ represents the forward pass for a single layer l, W_F^l represents the layer parameters and X_l represents the layer's input. Note that the above objective is optimized independently for each of the L Transformer layers.

3.2. MatQuant

MatQuant is a general purpose framework to develop a single model that can do well at any precision. It is a multi-scale training technique that works with most learning-based quantization schemes like QAT and OmniQuant discussed earlier. At its core, taking inspiration from Kusupati et al. (2022), MatQuant optimizes the quantization loss for several target bit-widths jointly.

To have a single model for various integer precisions, we nest smaller bit-widths into large ones

– leveraging the inherent Matryoshka nature of the integer data type. So, if we want to extract a r-bit model from a c-bit model ($0 < r < c$), we can just *slice out* the r most significant bits (MSBs) – using a right shift, followed by a left shift of the same order. Formally, the $S(q^c, r)$ operator slices the most significant r bits from a c-bit quantized vector q^c:

$$S(q^c, r) = \left(\left\lfloor \frac{q^c}{2^{c-r}} \right\rceil \right) * 2^{c-r} \tag{6}$$

Once we have this structure, we can optimize for several precisions by slicing the MSBs from the largest bit-width we are optimizing for. Let $R = \{r_1, r_2, ..., r_K\}$ be the bit-widths we want to optimize for, $Q(\cdot,)$ represent the quantization function of the base algorithm (i.e., any learning-based quantization scheme), $\mathcal{L}(\cdot)$ represent the loss function pertaining to the base algorithm, $F(\cdot)$ represent the forward pass required to compute the loss, θ represent the set of model/auxiliary parameters we are optimizing for and let W_F represent the model parameters. MatQuant's overall objective can be formulated as follows:

$$\min_{P} \frac{1}{N} \sum_{i \in [N]} \sum_{r \in R} \lambda_r \cdot \mathcal{L}\left(F(S(Q(\theta, c), r), x_i'), y_i' \right) \tag{7}$$

where $y_i' = y_i$ for QAT and $y_i' = F_l(W_F^l, X_l^i)$ for OmniQuant, and $x_i' = x_i$ for QAT and $x_i' = X_l^i$ for OmniQuant. λ_r is the loss reweighing factor for bit-width r.

In this work, we default to training MatQuant with three bit-widths, $R = \{8, 4, 2\}$, and subsequently perform a grid search over λ_r. This process aims to optimize performance such that the model performs well across all targeted precision levels. Further, while the focus of this paper is primarily on integer data types, we discuss the possibility of extending MatQuant to floating-point representations in Section 5.5.

A key point to note is that MatQuant primarily alters the quantized weight distributions across precision levels compared to the base quantization algorithm (OmniQuant or QAT). Figure 1c illustrates the differences in the quantized weight histograms obtained with and without MatQuant on Gemma-2 9B using OmniQuant. Upon close observation, we find that all the distributions of MatQuant are shifted to the right; that is, weights quantized with MatQuant tend to use more higher-valued weights. While this might not significantly impact int8 or even int4 models, int2 models benefit from utilizing more of the possible quantized weights compared to the baseline. Because int2 favors higher-valued weights, this effect propagates to higher-valued weights for int4, and then to int8. This observation highlights the potential overparameterization and freedom in the int8 data type to accommodate the more stringent needs of int2 during joint training. We further explore the effects of this phenomenon in Section 5.3 to develop a better standalone quantization technique for a single target precision.

3.2.1. *Interpolative Behavior*

Slicing. Although we explicitly train MatQuant for three precisions (int8, int4, int2), we find that the resulting model, when quantized to interpolated bit-widths like int6 & int3 by slicing (Eq. 6) the int8 model, performs on par with a baseline trained explicitly for that precision. It is also significantly better than slicing an int8 quantized model. We attribute this strong interpolation in bit-width space to MatQuant, and present more results in Sections 4.1 & 4.2.

Mix'n'Match. MatQuant also enables the use of different precisions at different layers through layer-wise Mix'n'Match (Devvrit et al., 2023), even though we never trained for these combinatorial possibilities. These large number of models, obtained at no cost, densely span the accuracy-vs-memory trade-off. We explore several Mix'n'Match strategies and find that having a higher precision (int8) in the middle layers and a lower precision (int2) at the start and end is Pareto-optimal among hundreds of possible models. See Section 4.3 for detailed experiments.

4. Experiments

In this section, we present an empirical evaluation of MatQuant working with two popular learning-

Table 1 | MatQuant with OmniQuant across Gemma-2 2B, 9B and Mistral 7B models. MatQuant performs on par with the baseline for int4 and int8 while significantly outperforming it for int2. Even the int3, int6 models obtained for free through interpolation from MatQuant perform comparably to the explicitly trained baselines. Task Avg. is average accuracy on the evaluation tasks (↑) while log pplx (perplexity) is computed on C4 validation set (↓).

Data type	Method	Gemma-2 2B		Gemma-2 9B		Mistral 7B	
	OmniQuant	Task Avg.	log pplx.	Task Avg.	log pplx.	Task Avg.	log pplx.
bfloat16		68.21	2.551	74.38	2.418	73.99	2.110
int8	Baseline	68.25	2.552	74.59	2.418	73.77	2.110
	MatQuant	67.85	2.580	74.33	2.446	73.46	2.132
int4	Sliced int8	62.98	2.794	72.19	2.546	46.59	4.139
	Baseline	67.03	2.598	74.33	2.451	73.62	2.136
	MatQuant	66.54	2.617	74.26	2.470	73.13	2.155
int2	Sliced int8	37.68	17.993	35.75	14.892	36.25	10.831
	Baseline	51.33	3.835	60.24	3.292	59.74	3.931
	MatQuant	**55.70**	**3.355**	**68.25**	**2.823**	**65.99**	**2.569**
int6	Sliced int8	67.66	2.565	74.61	2.424	73.50	2.122
	Baseline	68.06	2.554	74.23	2.420	74.10	2.112
	MatQuant	68.01	2.582	74.50	2.446	73.59	2.139
int3	Sliced int8	42.00	5.781	55.76	3.830	34.60	8.539
	Baseline	64.37	2.727	73.23	2.549	71.68	2.211
	MatQuant	63.24	2.757	73.25	2.535	71.55	2.228

based quantization methods: OmniQuant (Section 4.1) and QAT (Section 4.2). We demonstrate MatQuant's efficiency on Transformer-based LLMs. Unless otherwise mentioned, our primary focus is on weight quantization within the parameter-intensive FFN blocks of the Transformer layer.

For our experiments, we chose the default target quantization precisions to be int8, int4, and int2. Furthermore, we showcase the interpolative nature of MatQuant through evaluations on int6 and int3, as well as its elastic ability to densely span the accuracy-vs-cost trade-off using layer-wise Mix'n'Match (Section 4.3). Finally, we ablate on improving the performance of MatQuant (Sections 5.1 and 5.2) and extend MatQuant to the quantization of FFN and Attention parameters. (Section 5.3). Further training and fine-grained evaluation details are in the Appendix.

Models and Data. We experiment with Gemma-2 (Gemma-Team, 2024) 2B, 9B, and Mistral 7B (Jiang et al., 2023) models. For OmniQuant experiments, we sample 128 examples with a sequence length of 2048 from the C4 dataset (Raffel et al., 2020) and train using a batch size of 4. We train for a total of 10M tokens for all models except the int2 baseline, where we train the model for 20M tokens (Shao et al., 2023). For QAT experiments, we sample a fixed set of 100M tokens from the C4 dataset and train all our models using a batch size of 16 and a sequence length of 8192 for a single epoch.

Baselines. For OmniQuant and QAT, our primary baselines (referred to as "Baseline" in the tables and figures) are models trained explicitly for a given precision. When interpolating the models trained with MatQuant for int6 and int3, we do not perform any additional training. However, the baselines are trained explicitly for 6 and 3 bits respectively. We also compare against a sliced int8 OmniQuant/QAT baseline model to the corresponding precision (referred to as "Sliced int8" in the tables).

Evaluation Datasets. Following recent work (Frantar et al., 2022; Ma et al., 2024), we evaluate all the methods based on log perplexity and average zero-shot accuracy across a collection of downstream tasks. We use C4's test

Table 2 | MatQuant with QAT across Gemma-2 2B, 9B and Mistral 7B models. MatQuant performs on par with the baseline for int4 and int8 while significantly outperforming it for int2. Even the int3, int6 models obtained for free through interpolation from MatQuant perform comparably to the explicitly trained baselines. Task Avg. is average accuracy on the evaluation tasks (↑) while log pplx (perplexity) is computed on C4 validation set (↓).

Data type	Method	Gemma-2 2B		Gemma-2 9B		Mistral 7B	
	QAT	Task Avg.	log pplx.	Task Avg.	log pplx.	Task Avg.	log pplx.
bfloat16		68.21	2.551	74.38	2.418	73.99	2.110
int8	Baseline	67.82	2.458	74.17	2.29	73.48	2.084
	MatQuant	67.68	2.471	74.77	2.301	72.41	2.085
int4	Sliced int8	67.20	2.458	73.25	2.338	71.83	2.164
	Baseline	67.03	2.512	73.26	2.324	72.13	2.105
	MatQuant	67.05	2.521	73.71	2.332	71.63	2.111
int2	Sliced int8	39.67	9.317	40.35	7.144	38.40	10.594
	Baseline	47.74	3.433	56.02	2.923	54.95	2.699
	MatQuant	**52.43**	**3.153**	**62.32**	**2.756**	**61.29**	**2.474**
int6	Sliced int8	67.55	2.462	74.12	2.294	73.30	2.088
	Baseline	67.75	2.460	74.31	2.293	72.71	2.077
	MatQuant	67.60	2.476	74.55	2.303	72.70	2.089
int3	Sliced int8	60.23	2.913	68.57	2.565	65.29	2.441
	Baseline	61.75	2.678	69.9	2.43	68.82	2.197
	MatQuant	62.51	2.798	70.68	2.486	66.44	2.308

set to calculate perplexity, and for downstream evaluations, we test on ARC-c, ARC-e (Clark et al., 2018), BoolQ (Clark et al., 2019), HellaSwag (Zellers et al., 2019), PIQA (Bisk et al., 2020), and Winogrande (Sakaguchi et al., 2020).

4.1. MatQuant **with OmniQuant**

Table 1 shows the efficacy of MatQuant when used with FFN-only OmniQuant and compared to explicitly trained OmniQuant baselines for the target precisions, i.e., int8, int4, and int2, across all the models. While the average downstream accuracy of MatQuant for int8 and int4 quantization is within 0.5% of the corresponding independently trained baselines, the int2 quantized models of MatQuant are 4.37%, 8.01%, and 6.35% more accurate for Gemma-2 2B, 9B, and Mistral 7B, respectively. Similar trends and improvements follow when measuring performance through validation log perplexity. Further, the quantized int4 and int2 models *sliced* from the int8 OmniQuant baseline suffer a significant drop in accuracy around int4, demonstrating that the nested structure of int8 is not well utilized.

Sliced Interpolation. Beyond the target quantization granularities (int8, int4, and int2), MatQuant allows for bit-width interpolation to bit-widths not optimized during training. We find that the accuracy of the int6 and int3 models obtained by slicing the MatQuant models is comparable to explicitly trained baselines for both precisions.

4.2. MatQuant **with QAT**

To further demonstrate the generality of MatQuant, we experiment on the same models using the popular QAT technique. Following the trend of experimental results with OmniQuant, we show in Table 2 that the models trained using MatQuant with QAT are comparable to the explicitly trained baselines for all the targeted bit-widths of int8 and int4. However, int2 quantized models using MatQuant are 4.69%, 6.30%, and 6.34% more accurate for Gemma-2 2B, 9B, and Mistral 7B, respectively.

Sliced Interpolation. Models trained using MatQuant with QAT exhibit strong interpolative performance similar to that of MatQuant with

OmniQuant. We find that the accuracy of the int6 and int3 models obtained by *slicing* the MatQuant models is comparable to explicitly trained baselines for both interpolated bit-widths.

While OmniQuant only trains the auxiliary parameters needed for quantization, QAT also updates the weight parameters. This potentially results in severe overfitting to the C4 subset used in the experiments. We observe this overfitting in all the experiments presented in Table 2, where the log perplexities improve for QAT compared to OmniQuant, while the downstream accuracies suffer. This also highlights the need for high-quality data for QAT to realize its benefits; otherwise, users are better off using resource-friendly methods like OmniQuant.

Table 3 | Design choice ablation for loss re-weighting of the 3 target bit-widths (int8, int4, int2) that MatQuant explicitly optimizes. Note that MatQuant $(0, 0, 1) \equiv$ Single Precison MatQuant.

Data type	Weightings	Gemma-2 2B	Gemma-2 9B	Mistral 7B
		Task Avg.		
int8	$(1, 1, 1)$	67.42	73.97	73.46
	$(1, 1, \sqrt{2})$	67.31	73.45	73.41
	$(2, 2, 1)$	**67.85**	74.02	**73.82**
	$(\sqrt{2}, \sqrt{2}, 1)$	67.3	**74.33**	**73.82**
int4	$(1, 1, 1)$	66.11	73.88	73.13
	$(1, 1, \sqrt{2})$	**66.70**	73.75	73.29
	$(2, 2, 1)$	66.54	**74.33**	**73.5**
	$(\sqrt{2}, \sqrt{2}, 1)$	66.46	74.26	72.97
int2	$(1, 1, 1)$	**55.71**	**68.52**	65.99
	$(1, 1, \sqrt{2})$	57.08	67.93	**66.28**
	$(2, 2, 1)$	**55.70**	66.72	63.49
	$(\sqrt{2}, \sqrt{2}, 1)$	55.29	68.25	57.85

4.3. Layerwise Mix'n'Match

Alongside the strong slicing-based interpolative properties, quantization with MatQuant also enables another form of elastic and interpolative behavior through Mix'n'Match. Mix'n'Match provides a mechanism to obtain a combinatorial number of strong models by using different quantization granularities, from the target bit-widths – i.e., int8, int4, and int2 across layers. Figure 2 shows the ability of Mix'n'Match to densely span the Pareto-optimal accuracy-vs-bits-per-FFN-parameter (memory/cost) trade-off for the Gemma-2 9B model trained using MatQuant with OmniQuant – sometimes even improving on the bfloat16 model accuracy. While there are many more feasible models, we only showcase the best models obtained through the strategy described in Section 3.2.1 and further expanded in Appendix A. Interestingly, the Mix'n'Match models with effective bit-width of 3 and 6 are as accurate as models obtained through slicing. This opens up possibilities for effective serving depending on hardware support (Section 5.4).

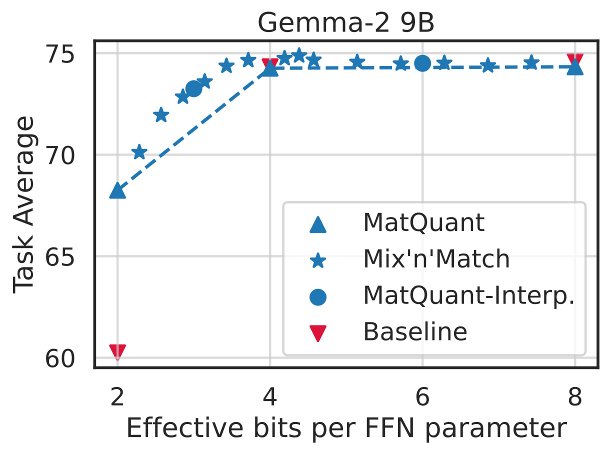

Figure 2 | Mix'n'Match on Gemma-2 9B model trained using MatQuant with OmniQuant allows elastic pareto-optimal accuracy-vs-cost model extraction for free during deployment.

5. Ablations and Discussion

In this section, we present design ablations to improve MatQuant. Section 5.1 discusses the effect of non-uniform weighting across target precisions (int8, int4, int2), and Section 5.2 explores enabling co-distillation of lower precision levels (int4, int2) from the highest precision quantized model (int8). During the process of extending MatQuant to all Transformer parameters, not just the FFN block, we uncovered an interesting hybrid quantization algorithm (between Baseline and MatQuant). Section 5.3 further details this method, called Single Precison MatQuant, which stabilizes the otherwise QAT baseline for all the Transformer weights. Finally, we also discuss extending MatQuant beyond integer data types and the considerations for effective deployment on current hardware.

5.1. Weightings (λ_r) for MatQuant

Depending on the constraints, we may wish to maximize the accuracy of one of the target bit-widths in MatQuant. Equation 7 provides a general formulation of MatQuant that supports a grid search on the weights λ_r for bit-width r. The results in Section 4 are with the weights that have balanced performance across target precisions. Table 3 shows the weight multiplier ablation results for Gemma-2 2B, 9B, and Mistral 7B. While equal weighting for all precisions works well, we see that higher weights for a specific precision results in increased accuracy for that bit-width. This re-weighting to improve int8 and int4 models often results in a minor accuracy drop for the int2 models. We can consider re-weighting as scaling the importance of the bits during training, and finding an optimal grid-search-free recipe is an interesting research question.

5.2. Co-distillation for MatQuant

Given the nested nature of the models trained using MatQuant, we explored co-distillation, where the outputs from a higher-precision model are used as the target for the lower-precision nested model, either in a standalone fashion or alongside the ground truth target (weighted equally). Table 4 shows the effects of co-distillation applied to MatQuant with both OmniQuant and QAT on Gemma-2 9B. While int8 and int4 show no significant improvement, the nested int2 model benefits substantially from the int8 supervision, reaching 1.65% higher accuracy than the non-co-distilled MatQuant with OmniQuant. This helps us push the int2 quantized Gemma-2 9B beyond 70% average downstream accuracy for the first time across all our experiments. Co-distillation in MatQuant opens up avenues for interesting design choices that can further leverage the inherent nested structure of integer data types.

5.3. Single Precison MatQuant

In Tables 1 and 2, MatQuant performs on par with the explicitly trained baselines for int4, int8, and the interpolated int3 and int6 precisions. However, the int2 models show a significant accuracy improvement. To investigate this, we conducted a simple ablation in MatQuant by removing the loss terms for int4 and int8 (i.e., $R = \{2\}$ in Equation 7 or setting $\lambda_4 = \lambda_8 = 0$) and present the results in Table 5. We call this version of MatQuant as Single Precison MatQuant. With Single Precison MatQuant, we observe a further boost of up to 1.67%, in the accuracy of int2 models at a ~2% accuracy drop in the corresponding int4 and int8 models – int2 is still nested within int8. This improvement likely stems from the six additional bits available during MatQuant-style training to optimize the int2 representation.

In the case of Single Precison MatQuant, gradient descent is free to tune these six additional bits to improve the overall quality of the int2 model. In MatQuant, since we have additional losses to preserve the performance of the int4

Table 4 | Design choice ablations for co-distillation within MatQuant. x → y represents distilling the y-bit model from the x-bit model. We note that the accuracy for int2 has significantly improved while minimally impacting the other bit-widths.

		OmniQuant		QAT	
Data type	Config.	Task Avg.	log pplx.	Task Avg.	log pplx.
int8	[8, 4, 2]	**73.97**	**2.451**	74.77	2.301
	[8, 4, 8 → 2]	73.40	2.467	74.72	**2.298**
	[8, 4, 2, 8 → 2]	73.46	2.466	74.62	2.299
	[8, 4, 2, 8 → 4; 2]	73.32	2.466	**74.80**	2.302
int4	[8, 4, 2]	**73.88**	**2.481**	73.71	2.332
	[8, 4, 8 → 2]	73.84	2.488	73.76	**2.328**
	[8, 4, 2, 8 → 2]	73.01	2.495	**73.78**	2.329
	[8, 4, 2, 8 → 4; 2]	73.12	2.518	73.48	2.330
int2	[8, 4, 2]	68.52	2.809	62.32	2.756
	[8, 4, 8 → 2]	69.2	2.796	61.81	**2.740**
	[8, 4, 2, 8 → 2]	**70.17**	**2.778**	**62.51**	2.746
	[8, 4, 2, 8 → 4; 2]	69.72	2.804	62.12	2.746

Table 5 | Single Precison MatQuant significantly improves upon the baseline for int2 and, at times, outperforms MatQuant. Crucially, int8 and int4 performances of Single Precison MatQuant experience a significant accuracy decrease (Tables 21 & 22).

int2	Gemma-2 2B		Gemma-2 9B		Mistral 7B	
Method	Task Avg.	log pplx.	Task Avg.	log pplx.	Task Avg.	log pplx.
OmniQuant	51.33	3.835	60.24	3.292	59.74	3.931
S.P. MatQuant	**57.38**	**3.185**	**68.58**	2.857	**67.36**	**2.464**
MatQuant	55.71	3.292	68.52	**2.809**	65.99	2.569
QAT	47.74	3.433	56.02	2.923	54.95	2.699
S.P. MatQuant	**53.18**	**3.090**	**62.53**	**2.706**	**61.55**	**2.435**
MatQuant	52.43	3.153	62.32	2.756	61.29	2.474

Table 6 | Extending MatQuant with QAT to FFN + Attention parameters. Baseline QAT destabilizes for int2 and int3 but improves significantly through MatQuant & Single Precison MatQuant.

Data type	Method	Gemma-2 9B		Mistral 7B	
	QAT	Task Avg.	log pplx.	Task Avg.	log pplx.
bfloat16		74.38	2.418	73.99	2.110
int8	Baseline	74.61	2.353	73.73	2.091
	MatQuant	75.07	2.374	73.58	2.101
int4	Sliced int8	73.56	2.43	71.42	2.246
	Baseline	72.98	2.40	71.87	2.132
	MatQuant	74.11	2.436	71.5	2.166
int2	Sliced int8	39.05	13.116	38.39	12.066
	Baseline	-	-	-	-
	S.P. MatQuant	**47.78**	**3.705**	34.69	7.564
	MatQuant	47.17	3.837	**43.33**	**3.806**
int6	Sliced int8	74.56	2.358	73.71	2.094
	Baseline	74.65	2.357	73.72	2.093
	MatQuant	75.04	2.379	73.36	2.106
int3	Sliced int8	64.23	2.908	39.36	4.918
	Baseline	-	-	-	-
	S.P. MatQuant	68.69	2.569	68.41	2.245
	MatQuant	66.94	2.91	59.45	2.703

and int8, the int2 performance is slightly worse than Single Precison MatQuant. However, since the int4 and int8 models are typically very close in accuracy to the bfloat16 model, MatQuant can shift some of the weights to improve the int2 model. As int4 and int8 models have substantially more quantized buckets than int2, we hypothesize that shifting some weights into adjacent buckets may not significantly affect their performance; however, it can significantly impact int2's performance. In fact, in the weight distributions presented in Fig 1c, we observe that MatQuant results in a model where larger number of weights are assigned to the higher-valued buckets. Conclusively, MatQuant and Single Precison MatQuant inherently seem to be a better way of doing low-bit quantization.

FFN + Attention Weight Quantization. We present results for FFN + Attention quantization for QAT in Table 6. For int8, int4 and the interpolated int6 model, MatQuant performs on par with the *Baseline*. However, we found int2 and int3 to be very unstable while quantizing both, the FFN and the Attention parameters. Most recent works that do QAT for both the blocks Chen et al. (2024); Du et al. (2024); Liu et al. (2024a) either do some form of warm starting for the quantized parameters, or have additional distillation and auxiliary loss functions. In the naive setup of minimizing the loss with respect to the ground truth, we find QAT to be very unstable at lower precisions. However, both MatQuant and Single Precison MatQuant are very stable further highlighting the benefits brought by MatQuant style training.

5.4. Deployment Considerations

Current hardware accelerators have native support for serving int8 and int4 quantized models. Additionally, custom-implemented CUDA kernels can can support various low-precision bit-widths, like int2 and int3 (Chee et al., 2024; Frantar et al., 2022). MatQuant can generate a large number of models at inference time. Depending on the serving environment, we can choose between Mix'n'Match models and homogeneous sliced models. For example, suppose the serving environment has a memory constraint equivalent to an int3 model but lacks optimized support for int3, while supporting int2. In this case, a Mix'n'Match model performing comparably to the int3 model could be deployed. More generally, as depicted in Figure 2, MatQuant densely spans the memory-versus-accuracy curve and can be leveraged to obtain the most performant model for a specific serving constraint. MatQuant can enable further research on hardware software co-design to effectively support elastic bit-widths on-the-fly during inference time.

5.5. Extension to Floating Point

Extending MatQuant to floating-point representations, such as FP8 and FP4, presents significant challenges. Given that the exponent is encoded within the bit representation and contributes to the value as a power of 2 (i.e., effectively $\log_2$), slicing it results in buckets whose sizes increase exponentially, unlike the integer case, where bucket sizes are constant. For example, slicing the first two bits from int8 yields buckets of 0, 64, 128, 192. Here, the bucket size (64) is constant; however, this would not be the case when slicing two exponent bits from FP8. This is a promising avenue for future research that could