Gaussian Weight Sampling for Scalable, Efficient and Stable Mixed-Precision QAT

Anonymous ACL submission

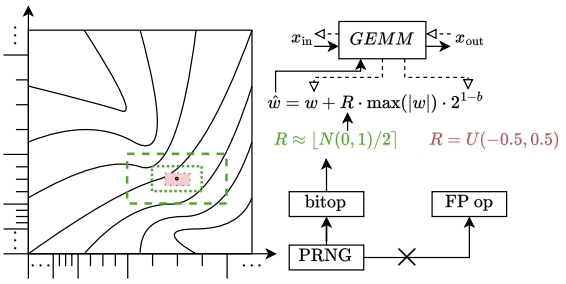
Abstract

The resources required for training transformer models increase as the model size grows, leading to the proposal and implementation of hardware-friendly computation methods such as FP8 and OCP MX. These methods introduce the mixed-precision problem, which has an exponentially large search space and negatively impacts training stability during extensive training over 200B tokens.

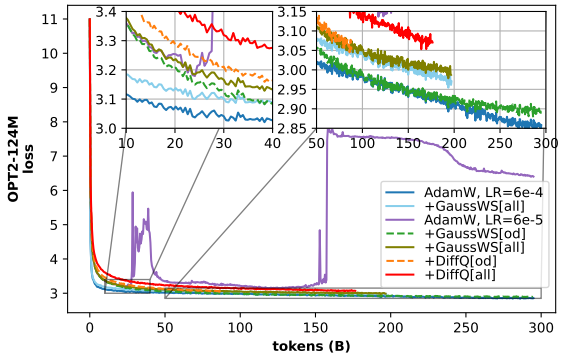
Based on our observation that FP mixed precision training shares the same issues of conventional mixed-precision Quantization-Aware Training (QAT), including the oscillation problem of Straight-Through Estimator (STE)-based QAT, we propose Gaussian weight sampling. The proposed method, or GaussWS, addresses the problem on mixed-precision by extending Pseudo Quantization Training (PQT) with an FP-friendly noise distribution and a GPU-friendly noise generation method.

We demonstrate that Gaussian weight sampling is scalable, *i.e.,* supports low-precision FP down to MXFP4, both analytically and empirically. The proposed method is efficient, incurring a low computational overhead as low as 0.47% on the A100 GPU in terms of Llama2 training tokens per second, and requiring 2 bytes per parameter in GPU memory.

We demonstrate that the proposed method is stable, closely following or even surpassing pre-training performance of BF16 baseline with the OPT2-124M model on the OpenWebText dataset, the Llama2-134M model on the C4 dataset (up to 300B tokens) and the Llama2-1B model on the C4 dataset (up to 100B tokens).



(a) Overview of proposed method. The figure on the left illustrates the loss landscape in real numbers with FP approximation. The dot in the middle of the rectangle represents a parameter instance and the rectangles represent sampling range with the corresponding noise distribution on the right. The figure on the right depicts computation graph of the proposed method, where solid arrows indicate the forward pass, and dotted arrows indicate the backward pass. The proposed method determines the optimal bitwidth b through training. We propose an FP-friendly noise distribution and a GPU-friendly noise generation method.



(b) Result of pre-training the OPT2-124M model on the OpenWebText dataset. [part] indicates that the corresponding method is applied to "part" layers of the transformer module. [od] is used as shorthand for [out,down]. Both PQT methods mitigate training instability of baseline AdamW (purple). GaussWS[od] (dashed green) yields the best result with learning rate 6×10^{-5}. GaussWS (green and olive) outperforms the corresponding DiffQ (orange and red) throughout the training process.

Figure 1: Summary of Gaussian weight sampling.

1 Introduction

The training cost of Large Language Model (LLM) has increased as the model size has grown over time. For Grattafiori et al. (2024), the training requires 16K H100 and at least 11.2MW for total TDP of the GPUs. Studies have been conducted to reduce training cost. PEFT, *e.g.,* Hu et al. (2021), Dettmers et al. (2023) and Loeschcke et al. (2024), reduce the size of training parameters. Ren et al.

(2021) and Rajbhandari et al. (2021) offload GPU memory into CPU memory. Peng et al. (2023) reduces required throughput of collective communication. Zhang et al. (2024) and Zhu et al. (2024) introduce optimizer with less internal state which results in less GPU memory required.

Especially, hardware-friendly low precision data types such as FP8 (Micikevicius et al., 2022) and OCP MX (Rouhani et al., 2023) have been proposed. However, low-precision (*e.g.,* FP8 based) training faces two critical problems. First, it suffers from training instability due to quantization-induced oscillations like traditional STE-based QAT. Second, the problem of mapping parts of the model to specific bit precision is of an exponential complexity. For example, mapping n operations to either FP8 or BF16 results in $O(2^n)$ cases that requires extensive training with over 200B tokens to validate training stability. This makes manual search methods inefficient and suboptimal, thereby often resorting to suboptimal designs, *e.g.,* all BF16 training.

Pseudo Quantization Training (PQT), *e.g.,* Défossez et al. (2022) and Park et al. (2022), can solve the problem. PQT employs Pseudo Quantization Noise (PQN), which generalizes actual quantization noise during the training process. PQT is fully differentiable without approximation like that of STE. However, existing PQT methods are neither FP-friendly nor GPU-friendly, leading to numerical instablility and computational overhead. Specifically, as shown in Figure 1a, the limited dynamic range of $U(-0.5, 0.5)$ degrades the effect of PQN when cast into low-precision FP, while requiring unnecessary precision for representation and FP operations for generation.

In this paper, we extend PQT to be FP- and GPU-friendly, thereby ensuring scalability to low precision operators with minimal overhead. Specifically, we propose an FP-friendly noise distribution and a GPU-friendly noise generation method. The proposed Gaussian weight sampling is scalable down to MXFP4, incurs an overhead as low as 0.47% in terms of training throughput on A100, and requires only 2 bytes per parameter in terms of GPU memory. The latter can be compensated using a parameter-efficient optimizer, such as Adam-mini (Zhang et al., 2024).

We demonstrate that the proposed method enables stable pre-training that closely follows, or even outperforms, the BF16 counterpart for the OPT2-124M and the Llama2-{134M, 1B} models up to 300B tokens.

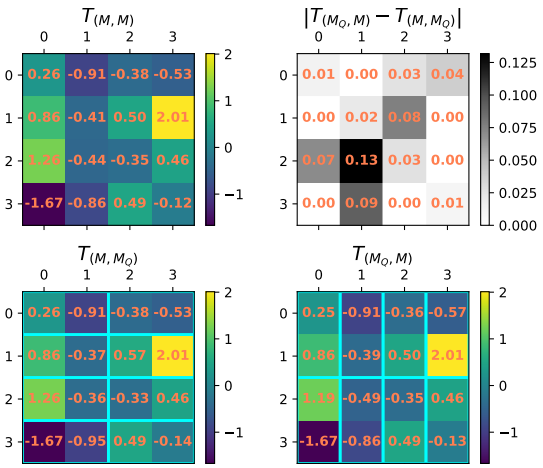


Figure 2: Forward-backward error when following vector-wise quantization. Used internal datatype of INT4 with block size of 2 for simplicity. Note that visualized values are fake-quantized. $T_{(M,M)}$ is randomly sampled from $N(0, 1)$.

To sum up, our contributions are as follows:

- We show that FP-based mixed-precision training shares the same problems of STE-based QAT, namely an exponential search space and challenges with training stability.
- We extend the existing solution, differentiable mixed precision QAT, *i.e.,* PQT, to be FP- and GPU-friendly, making it scalable down to MXFP4 with minimal overhead.
- We demonstrate that the proposed method enables stable PQT that closely follows, or even outperforms, the baseline BF16 pre-training.

2 Preliminaries and Problems

2.1 Oscillation issue in low-precision training

Reducing bit precision for training is one of the most effective methods to reduce the training cost of large language model, *e.g.,* Peng et al. (2023), Fishman et al. (2024), Wang et al. (2025), DeepSeek-AI et al. (2024), Rouhani et al. (2023). Specifically, the bottom row of Figure 2 visualizes how OCP MX works in a simple example. Consider $T_{(M,M_Q)} \cdot T_{(M_Q,M)}$. The matrix multiplication is realized via vector dot products (of size-2 MX blocks) which are realized by MX-based compute units.

A naïve appalication of low-precision training can incur an oscillation problem which hurts training stability. Consider an MX-compliant matrix

multiplication where the quantization axis lies along the inner dimension:

- forward $T_{(M,N)} = A_{(M,K_Q)} W_{(K_Q,N)}$
- gradient $\frac{\partial L}{\partial W}_{(K,N)} = A^T_{(K,M_Q)} \frac{\partial L}{\partial T}_{(M_Q,N)}$
- backprop $\frac{\partial L}{\partial A}_{(M,K)} = \frac{\partial L}{\partial T}_{(M,N_Q)} W^T_{(N_Q,K)}$

where A is the input activation, W the parameter, T the output activation and L the target loss. Subscript corresponds to the shape of the given matrix, where $_Q$ marks the quantization axis.

Note the difference between A and W in the forward and backward passes, *i.e.,* $A_{(M,K_Q)}$ and $W_{(K_Q,N)}$ compared to $A^T_{(K,M_Q)}$ and $W^T_{(N_Q,K)}$, as demonstrated in Figure 2. This discrepancy can lead to oscillation issues, similar to those observed with STE-based QAT, resulting in suboptimal model training and preventing the model from reaching the desired optimal point in the loss landscape. (Défossez et al., 2022) (Park et al., 2022)

2.2 Problem of existing PQT

PQT effectively addresses a problem of mixed precision QAT by directly training quantization parameter, *i.e.,* bit precision, without the forward-backward discrepancy introduced by the STE. This results in a fully differentiable training process with forward-backward consistency. However, current PQT methods are neither FP-friendly nor GPU-friendly.

Current PQT methods are not FP-friendly, as they use uniform noise $U(-0.5, 0.5)$ as the basis for PQN. This requires unnecessary precision and disrupts forward-backward consistency with numerical instability. Consequently, these methods are limited in the range of "safe" bitwidths for PQN and necessitate high-precision operators, such as BF16. Refer to Section 3.3 for detail.

Current PQT methods are not GPU-friendly. They generate random values by performing FP operations on random integer streams produced by Pseudo-Random Number Generator (PRNG). This exacerbates the bottleneck on vector operator (CUDA core) during the training process. This issue is particularly pronounced on NVIDIA's datacenter GPUs like the A100. Datacenter GPUs have relatively lower vector operation throughput compared to their consumer counterparts. See empirical result in Figure 5.

3 Method

3.1 Overview

We aim to address the problem of PQT with low-precision floating point hardware in mind. The proposed Gaussian weight sampling method extends PQT to be MX-compliant (Section 3.2), resilient to low precision floating point operations (Section 3.3) and computationally lightweight (Section 3.4). We discuss the effect of training with the proposed PQN in Section 3.5. The proposed method is implemented in Triton (Mattson et al., 2019) with decisions that favor predictably optimal throughput and straightforward implementation (Section 3.6). Section 3.7 describes the implementation details, including the method to ensure unbiased PQN while maintaining forward-backward consistency, and the bitwidth parameter implementation.

While Park et al. (2022) applied PQT to both weights and activations, our study focuses on applying the method exclusively to weights. This approach has the potential to decrease the volume of collective communication, a major bottleneck in LLM training, thereby accelerating the training process. (Peng et al., 2023) Moreover, it offers faster training with reduced overhead compared to applying the method to both weights and activations.

3.2 Gaussian weight sampling

As discussed in Section 2.1, a vector-wise quantization, aligned with the inner dimension of matrix multiplication, can cause discrepancies between forward and backward passes, leading to the oscillation problem. The issue arises because the absolute maximum value of the block, *e.g.,* size-2 blocks in Figure 2, changes when transposed. Square-blockwise quantization can resolve this problem and ensures transpose-commutativity. Therefore, in Gaussian weight sampling, parameters are grouped into square block units, as in DeepSeek-AI et al. (2024). Note that square-blockwise quantization is a special case of vectorwise quantization where adjacent vectors share the same scale, making it MX-compliant.

The formulation of Gaussian weight sampling is as follows:

$$\hat{w} = w + R \cdot \text{broadcast}_{b_l}\left(\max_{b_l}(|w|) \cdot 2^{1-b_t}\right) \quad (1)$$

where $w, \hat{w}, R \in \mathbb{R}^{m \times n}$, $b_t \in \mathbb{R}^{\lceil m/b_l \rceil \times \lceil n/b_l \rceil}$, and

$b_l = 32$ is the square block size following OCP MX. w is an original parameter, $\hat{w}$ is a sampled parameter, R represents random and b_t is blockwise bitwidth. We refer to the right-hand side of the addition as PQN.

Note that the formula above is fully differentiable. With an approximation of $\frac{\partial \max_{b_l}(|w|)}{\partial w} \approx 0$ assuming gradient to single element out of 32 by 32 block is negligible, we can calculate the gradient as follows:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{w}} \tag{2}$$

$$\frac{\partial L}{\partial b_t} = -\ln 2 \cdot \max_{b_l}(|w|) \cdot 2^{1-b_t} \cdot \sum_{b_l} \left(\frac{\partial L}{\partial \hat{w}} \odot R \right) \tag{3}$$

where $\odot$ denotes the Hadamard product.

3.3 Choice of FP-friendly *R*

A uniform distribution $U(-0.5, 0.5)$ is theoretically the best fit as a basis for PQN. However, a rounded normal distribution $\lfloor N(0,1)/2 \rceil$ enhances numerical stability because of its limited data points with limited precision, *i.e.,* {-3, -2, -1, 0, 1, 2, 3}, allowing a less stringent constraint on the precision of the subsequent operator, *i.e.,* matrix multiplication, and the range of b_t.

In particular, the value of $\hat{w}$ is cast into floating point, *e.g.,* BF16 or FP8, to serve as the input for the subsequent matrix multiplication. Note that FP casting, similar to integer casting, introduces rounding errors. However, the backward pass in Equation 3 has no indication of whether the PQN was rounded to zero in the forward pass of Equation 1 and the subsequent FP casting. To ensure stable training without errors between the forward and the backward pass, the PQN should not underflow, *i.e.,*

$$fp_{e,m}(\hat{w}_{ij}) \neq fp_{e,m}(w_{ij})^{\forall} i, j|_{R_{ij} \neq 0} \tag{4}$$

where $fp_{e,m}(x)$ denotes casting x into a floating point representation with an e-bit exponent and an m-bit mantissa.

Without loss of generality, assume FP8_e5m2 and scaled maximum value in range [1, 2) where possible values are {1.0, 1.25, 1.5, 1.75}. Absolute value of the addition delta should be ≥ 0.25 to ensure the value is updated in an unbiased manner. Note that the stepsize is $0.25 = 2^{-2} = 2^{\lfloor \log_2 |w| \rfloor - m}$. We need to ensure that the magnitude of every non-zero element of the PQN is greater than or equal to it. Specifically, for each block and ${}^{\forall} i, j|_{R_{ij} \neq 0}$,

$$R_{ij} \left(\max_{b_l}(|w|) \cdot 2^{1-b_t} \right) \geq 2^{\lfloor \log_2 |w_{ij}| \rfloor - m} \tag{5}$$

With $2^{-r} = \min_{R_{ij} \neq 0} |R_{ij}|$, this simplifies to:

$$b_t \leq m + 2 - r \tag{6}$$

Equation 6 sets the upper bound of the bitwidth b_t, or the lower bound of the magnitude of the PQN, to prevent underflow in a floating-point representation with an m-bit mantissa. For example, with a BF16 operator, $R = U(-0.5, 0.5)$ represented in 4-bit requires $b_t \leq 5$, while $R \in \{-2, -1, 0, 1, 2\}$ requires a less stringent constraint of $b_t \leq 9$.

3.4 Efficient generation of *R*

A Pseudo-Random Number Generator (PRNG), *e.g.,* Lathrop et al. (2011) and Overton (2020), produces a random bit stream, effectively generating random integers. Random numbers in the real number domain are derived from these random integers. To obtain a value from $U(0,1)$, random integers are divided by their maximum possible value. To generate a value from $N(0,1)$, two values from $U(0,1)$ are used and transformed via the Box-Muller method. (Box and Muller, 1958)

Note that the rounded normal $\lfloor N(0,1)/2 \rceil$ does not require aforementioned floating point operations. Assuming that each bit of the random bit stream from the PRNG is independently random, we create a random distribution that approximates the rounded normal distribution using two base cases:

$$\begin{cases} P(A\&B) = P(A) \cdot P(B) \\ P(A|B) = P(A) + P(B) - P(A\&B) \end{cases} \tag{7}$$

where A and B represent bitwise random variables, $\&$ and $|$ are bitwise operators, and $P(X)$ is shorthand for $Pr(X = 1)$.

Specifically, the distribution that we generate is:

$$\begin{cases} Pr(-2) = Pr(+2) = 3/4 \cdot 2^{-9} \approx 1/682.7 \\ Pr(-1) = Pr(+1) = (3/4)^2 \cdot 2^{-2} \approx 1/7.1 \\ Pr(0) = 1 - Pr(\pm 1) - Pr(\pm 2) \approx 0.716 \end{cases} \tag{8}$$

The generated R values are represented in a sign-mantissa format with 4 bits per element, and 8 elements are packed into a 32-bit register. Compared to 2's complement, the sign-mantissa format is simpler to generate and reconstruct into floating point.

3.5 Effect of training with the proposed PQN

The proposed method effectively performs stochastic annealing on the precision of near-zero values of high-precision parameters.

Assume a block w, which is represented in high-precision FP, contains small non-negative value 2^s, and resilient to PQN with $b_t = 6$. Without loss of generality, assume the absolute maximum value of w is scaled to be in the interval [1, 2). With $R_{ij} = 1$, adding the PQN shifts 2^s to the range $[2^s + 2^{-5}, 2^s + 2^{-4})$. Assuming s is small enough, the subsequent FP rounding limits the precision of the value to that of low-precision FP in the range $[2^{-5}, 2^{-4})$, with a stepsize of 2^{-m-5}. $2^s \geq 2^{-m-5}$ prevents underflow while $2^s < 2^{-m-5}$ risks underflow. In other words, the proposed method incorporates stochastic precision annealing, nullifying near-zero values of high-precision FP parameters with a probability of up to ≈ 0.284.

3.6 Design decisions

Separate kernels. While baseline BF16 training require only one kernel call for the linear layer, we need three kernel calls: (1) generating R, (2) unpacking R and adding scaled maximum, and (3) the linear operation. Fusing consecutive operations typically helps achieve maximum throughput by reducing GPU memory communication. In our case, however, fusing the operations does more harm than good.

Firstly, R generation is not fused. PRNG is an algorithm that loops based on its internal state to generate random values iteratively. The longer a PRNG's internal state is reused, the more it reduces the degree of parallelism, limiting the utilization of parallel hardware. In other words, there exists an optimal ratio of parallelization to achieve optimal throughput. If the number of random values R generated and used per operator (CUDA core) does not match, additional communication is required. In practice, fusing the generation of R with subsequent operations led to significant throughput variations depending on the shape of w.

Secondly, we do not fuse the scaled addition with the subsequent matrix multiplication. This decision allows us to use the highly optimized PyTorch implementation of the linear operation and straightforward implementation.

GPU memory. Calculating the gradient of input activation in matrix multiplication requires $\hat{w}$, while calculating the gradient of b_t requires regenerating R. R is regenerated using the same seed value used in the forward pass, requiring 0.5 byte per parameter temporarily.

Although reconstructing $\hat{w}$ in the backward computation would reduce GPU memory overhead, we store $\hat{w}$ explicitly in BF16. It helps keep the implementation simple at the cost of a marginal increase in GPU memory. Note that overhead of 2 bytes per parameter can be offset by adopting parameter-efficient optimizers, such as Adam-mini.

In conjunction, the design decisions described above enable a straightforward implementation where $f(w) = \hat{w}$ is modularized into a single PyTorch layer.

3.7 Implementation details

Managing seed. A seed value is required to initialize the PRNG. For proper training, the value of R in the forward pass must be identical to the value of R in the backward pass. Additionally, to avoid bias across the entire model, the R values for each layer should be distinct and independent.

To achieve these properties, a multi-layer PRNG is employed to manage seeds and their corresponding random values effectively. First, a PRNG or seed generator is created using the user-specified seed value. Second, this seed generator is used to produce seed values to initialize the PRNG of each layer. Finally, the output of each layer's PRNG serves as the seed value for the GPU's PRNG, which then generates R.

The state of the final PRNG is updated every training iteration but remains frozen during gradient accumulation to mimic the virtual batch effect.

Bitwidth. We implemented an internal bitwidth parameter b_i for each 32 by 32 square unit of parameters in the linear layers. b_i is linearly scaled to represent bitwidth b_t as follows:

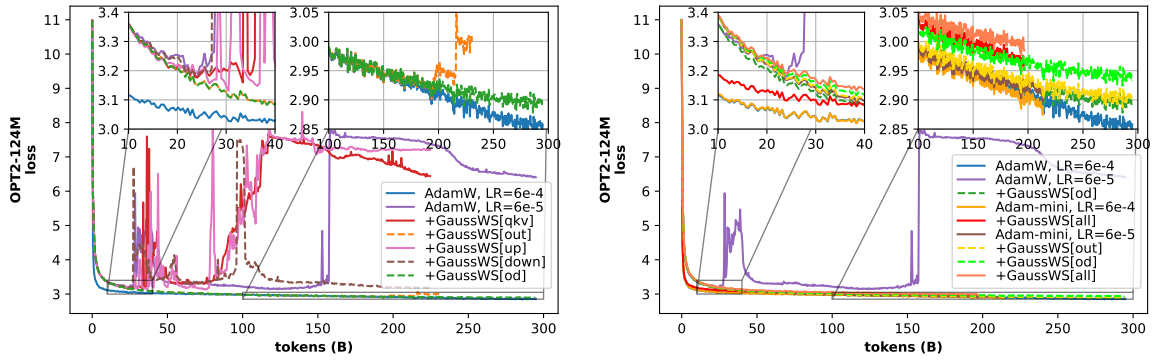
$$b_t = b_{\text{target}} + b_i \cdot (b_{\text{init}} - b_{\text{target}}) \tag{9}$$

where b_{init} and b_{target} are hyperparameters representing the initial and target bitwidths, respectively. b_i should be initialized with 1. b_t is guided towards b_{target} through the weight decay applied to b_i.

A loss term related to b_t can also be incorporated into the training loss:

$$L' = L + \frac{\lambda}{n} \sum_{i=1}^{n} \left| b_t^i - b_{\text{target}} \right| \tag{10}$$

where n is the number of b_t, and b_t^i denotes bitwidth of i-th block. In this scenario, an additional hyper-



(a) Stability case study with AdamW optimizer.

(b) With Adam-mini optimizer.

Figure 3: Training loss curve. OPT2-124M on OpenWebText dataset.

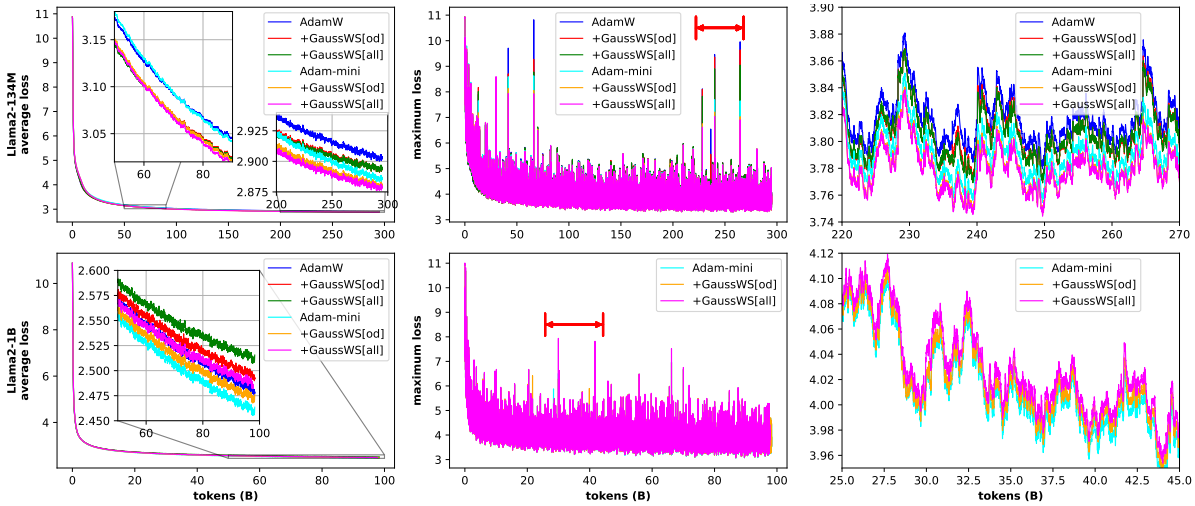


Figure 4: Training loss curve. Llama2-{134M, 1B} on C4 dataset. First column represents average loss and the other two represent maximum loss. Each datapoint corresponds to 19.66M tokens for Llama2-134M and 6.55M tokens for Llama2-1B. Weighted moving average is used with $\alpha = 1/16$ on left column and $\alpha = 1/128$ on right column for better visualization. Third column corresponds to range that is annotated with orange arrow on the second column.

parameter, λ, is required to appropriately scale the loss associated with the bitwidth parameter.

4 Experimental results

Transformer-based language models were trained from scratch: the OPT2-124M model on the OpenWebText dataset (Gokaslan and Cohen, 2019) (Section 4.1), and the Llama2-134M and Llama2-1B models on the C4 dataset (Raffel et al., 2023) (Section 4.2). The overhead of the proposed method is presented in Section 4.3. Resulting bitwidth is presented in Section 4.4.

We use "GaussWS[part]" to represent which part of the transformer block adopts the proposed method. We further shorten the notation as "[part]" in text. [od] is used as shorthand for [out,down]. Note that the OPT2 transformer block comprises four linear layers: qkv, out, up, and down. The qkv and out layers, along with the self-attention operation, constitute the attention module, while the up and down layers form the feed-forward module. "DiffQ[all]" represents that all linear layers of the transformer block adopt an extension of DiffQ, which is equivalent to GaussWS with BF16 $U(-0.5, 0.5)$ in place of $\lfloor N(0,1)/2 \rceil$.

We used BF16 GEMM with FP32 accumulation. While the proposed method supports low-precision operators, the use of high-precision operator allows for differential search over a larger range of b_t, as discussed in Section 3.3.

4.1 Pre-train OPT2-124M on OpenWebText

The OPT2-124M model is trained from scratch on the OpenWebText dataset up to 300B tokens. We used $b_{\text{init}} = 6$, $b_{\text{target}} = 0$ for weight decay, and $b_{\text{target}} = 4$ for bitwidth loss with $\lambda = 10^{-4}$.

GaussWS stabilizes training with minimal loss increase. The baseline BF16 with the AdamW optimizer is highly sensitive to the choice of learning rate. Results in Figure 1b shows that the training with a learning rate of 6×10^{-4} proceeds smoothly whereas a learning rate of 6×10^{-5} diverges and fails to recover. Both PQT methods mitigate such training instability while the proposed method incurs minimal increase in loss, especially with [od]. The difference in performance between GaussWS and DiffQ is attributed to the choice of R. GaussWS consistently outperforms DiffQ, which supports the argument presented in Section 3.3.

Stability case study. To identify the source of training instability, we restrict the application of the proposed method to each of the linear layers within the transformer block.

As shown in Figure 3a, at ≈30B tokens of training, [qkv], [up], and [down] begin to diverge and fail to recover. In contrast, [out] does not diverge and closely approximates the optimal training loss curve of AdamW with a learning rate of 6×10^{-4} until ≈200B tokens. [od], which applies the proposed method to the last layers of the residual addition branches in the transformer block, reduces divergence and yields the best results with a learning rate of 6×10^{-5}.

These training results show that the attention module is the source of instability at ≈30B tokens of training, while the feed-forward module is the source of instability at ≈200B tokens of training. The latter is consistent with Fishman et al. (2024).

Adam-mini. As shown in Figure 3b, baseline Adam-mini also stabilizes training at ≈30B tokens of training. [out] with Adam-mini do not suffer from instability at ≈200B tokens and approximates [od] with AdamW. [od] with Adam-mini slightly degrades in performance.

4.2 Pre-train Llama2 on C4

The Llama2-134M and Llama2-1B models are trained from scratch on the C4 dataset up to 300B and 100B tokens, respectively. We used $b_{\text{init}} = 6$, $b_{\text{target}} = 4$.

Llama2-134M training. As shown on the first row of Figure 4, Gaussian weight sampling improves Llama2-134M pre-training, for both average and worst case. Additionally, with GaussWS, it requires fewer tokens for Adam-mini to surpass AdamW.

Llama2-1B training. As shown on the second row of Figure 4, Gaussian weight sampling slightly degrades Llama2-1B pre-training, both for average and worst case. [od] minimizes loss degradation.

Note that in all cases shown in Figure 4, training with the proposed method closely follows the baseline.

tps (k)	**134M**	**360M**	**1B**	**3B**
AdamW	143.3	57.2	26.0	7.58
+GaussWS[od]	142.9	57.0	25.8	7.50
+GaussWS[all]	141.3	56.3	25.5	4.23⋆
+DiffQ[all]	116.6	52.2	23.1	-
Adam-mini	93.9	40.6	21.1	6.18
+GaussWS[od]	91.8	39.3	20.6	5.96
+GaussWS[all]	85.7	36.3	19.3	5.44
+DiffQ[all]	82.3	34.3	17.8	1.91⋆
GMEM (GiB)	**134M**	**360M**	**1B**	**3B**
AdamW	34.00	27.83	30.69	30.66
+GaussWS[od]	34.04	28.00	31.22	32.47
+GaussWS[all]	34.16	28.37	32.42	36.59
+DiffQ[all]	34.18	28.44	32.64	37.35
Adam-mini	33.87	27.50	29.70	27.53
+GaussWS[od]	33.92	27.66	30.23	29.34
+GaussWS[all]	34.03	28.03	31.43	33.45
+DiffQ[all]	34.05	28.10	31.65	34.21

Table 1: Tokens per second per GPU (top) and GPU memory usage (bottom) during Llama2 pre-training on the A100 GPU. Tokens per second in thousands and GPU memory in GiB. We used local batch size of {24, 12, 8, 3} for each case with fixed sequence length of 2048. ⋆ corresponds to low throughput due to frequent reallocation of GPU memory. "-" represents failed case with out-of-memory.

4.3 Overhead

Table 1 reports the throughput and GPU memory usage during Llama2 training.

The geometric mean of the overhead on training throughput for Llama2-{134M, 360M, 1B} with AdamW is 0.47%, 1.63% and 12.95% for [od], [all] and DiffQ[all], respectively, and 2.60%, 9.29% and 14.52% with Adam-mini. The proposed generation method enables efficient generation of the basis for PQN and reduces computational overhead.

GPU memory overhead is ≈2 bytes per parameter to store $\hat{w}$ in BF16. The proposed method requires less temporary memory to store R, using 0.5 bytes per element for $\lfloor N(0,1)/2 \rceil$ compared to 2 bytes for $U(-0.5, 0.5)$.

Figure 5 presents the results of the unit benchmark for the forward pass of the proposed method. Both the bitwise manipulation method and the Box-

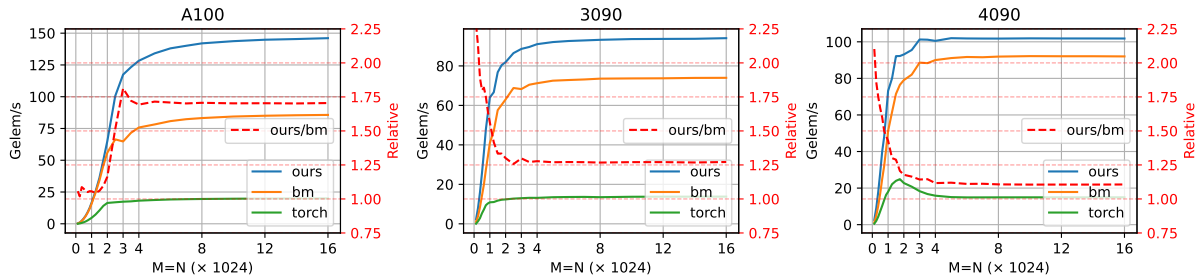


Figure 5: Forward pass benchmark results for the PyTorch layer implementing Equation 1 on a matrix $W_{(M,N)}$. Absolute throughput in 10^9 elements per second. "torch" indicates PyTorch baseline while the other two are implemented in Triton. "bm" implements Box-Muller transform and "ours" implements the proposed generation method described in Section 3.4.

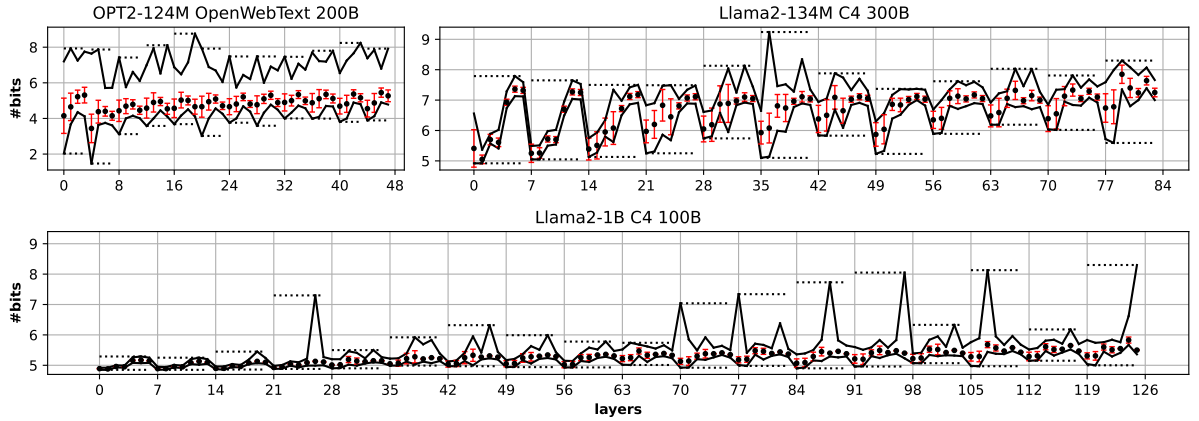


Figure 6: Resulting bitwidth b_t. Upper and lower solid lines represent layerwise maximum and minimum while dotted lines represent transformer-blockwise maximum and minimum. Dots and red lines indicate layerwise mean and standard deviation.

Muller method demonstrate at least a $3\times$ improvement compared to the baseline, as they are implemented in Triton and reduce global memory communication. The proposed noise generation method enhances throughput compared to the Box-Muller method across all test cases. It is particularly effective with larger matrix and the A100 GPU. Note that weight dimension of Llama 3.2 1B ranges from (2k, 0.5k) to (2k, 8k) while Llama 3.1 405B ranges from (16k, 1k) to (16k, 16k).

4.4 Bitwidth

Resulting bitwidth b_t is visualized in Figure 6 and Appendix C. Layers towards the end of the model tend to be more sensitive and require more precision. Layers in the feed-forward module tend to be more sensitive than those in the attention module. For the Llama-style transformer block, the out projection layer, or w_2, requires the highest precision on average among the three layers of the feed-forward module. The Llama2-1B model typically contains one layer with outlier per transformer block, unlike the other two models.

b_t	Exponent (#range)	Mantissa
3	2 (3+1)	1
6	3 (6+1)	4
12	4 (12+1)	10

Table 2: The number of exponent and mantissa bits that is analytically safe for inference when the model is trained using the proposed method. #range refers to the number of exponent ranges, with $+1$ accounting for the subnormal range. The number of mantissa bits is derived from Equation 6. Refer to Section 5 for detail.

5 Discussion

Stable pre-training results with the proposed method imply that $|w_{ij}| < 2^{-m-b_t+1}$ is not necessary, given appropriate scaling as discussed in Section 3.5. With $b_t = 6$, the minimum number of exponent ranges that satisfies the condition is 7, with 6 normal ranges, 1 subnormal range, and an optional Inf/NaN range. Note that more exponent ranges are required during training to support updates of w that are smaller than non-zero PQN values. We obtain Table 2 by repeating this logic.

Limitations

Proposed method is applied only on weight, leaving activation and gradient same as baseline. In particular, it is impossible to conduct differentiable search on gradient. Extending the proposed method to activation is left as future work.

The results are limited to pre-training the OPT2-124M model and the Llama2-134M model up to 300B tokens, and the Llama2-1B model up to 100B tokens. Further validation with larger models and longer training is required.

To improve the noise distribution R, we need to align the results of PQT with actual or fake quantization. This is left for future work.

References

G. E. P. Box and Mervin E. Muller. 1958. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610–611.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, et al. 2024. DeepSeek-V3 Technical Report. *arXiv*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. *arXiv*.

Alexandre Défossez, Yossi Adi, and Gabriel Synnaeve. 2022. Differentiable model compression via pseudo quantization noise. *Preprint*, arXiv:2104.09987.

Maxim Fishman, Brian Chmiel, Ron Banner, and Daniel Soudry. 2024. Scaling FP8 training to trillion-token LLMs. *arXiv*.

Aaron Gokaslan and Vanya Cohen. 2019. Openwebtext corpus. `http://Skylion007.github.io/OpenWebTextCorpus`.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, and other. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, et al. 2021. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv*.

Andrej Karpathy. 2022. NanoGPT. `https://github.com/karpathy/nanoGPT`.

Scott Lathrop, Jim Costa, William Kramer, John K Salmon, Mark A Moraes, et al. 2011. Parallel random numbers: As easy as 1, 2, 3. *2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. 2024. Torchtitan: One-stop pytorch native solution for production ready llm pre-training. *Preprint*, arXiv:2410.06511.

Sebastian Loeschcke, Mads Toftrup, Michael J Kastoryano, Serge Belongie, and Vésteinn Snæbjarnarson. 2024. LoQT: Low Rank Adapters for Quantized Training. *arXiv*.

Tim Mattson, Abdullah Muzahid, Armando Solar-Lezama, Philippe Tillet, H T Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19.

Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, et al. 2022. FP8 Formats for Deep Learning. *arXiv*.

Mark A Overton. 2020. Romu: Fast Nonlinear Pseudo-Random Number Generators Providing High Quality. *arXiv*.

Sein Park, Junhyuk So, Juncheol Shin, and Eunhyeok Park. 2022. NIPQ: Noise Injection Pseudo Quantization for Automated DNN Optimization. *arXiv*.

Houwen Peng, Kan Wu, Yixuan Wei, Guoshuai Zhao, Yuxiang Yang, et al. 2023. FP8-LM: Training FP8 Large Language Models. *arXiv*.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the limits of transfer learning with a unified text-to-text transformer. *Preprint*, arXiv:1910.10683.

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *arXiv*.

Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, et al. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. *arXiv*.

Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, et al. 2023. Microscaling Data Formats for Deep Learning. *arXiv*.

Ruizhe Wang, Yeyun Gong, Xiao Liu, Guoshuai Zhao, Ziyue Yang, et al. 2025. Optimizing Large Language Model Training Using FP4 Quantization. *arXiv*.

Yushun Zhang, Congliang Chen, Ziniu Li, Tian Ding, Chenwei Wu, et al. 2024. Adam-mini: Use Fewer Learning Rates To Gain More. *arXiv*.

Maxim Zhelnin, Viktor Moskvoretskii, Egor Shvetsov, Egor Venediktov, Mariya Krylova, et al. 2024. Gift-sw: Gaussian noise injected fine-tuning of salient weights for llms. *Preprint*, arXiv:2408.15300.

Hanqing Zhu, Zhenyu Zhang, Wenyan Cong, Xi Liu, Sem Park, et al. 2024. APOLLO: SGD-like Memory, AdamW-level Performance. *arXiv*.

A Related works

Zhelnin et al. (2024) proposed Quantization Noise Injection (QNI) to finetune LLM in a parameter-efficient way.

Research leveraging FP8 operators, *e.g.,* Fishman et al. (2024) DeepSeek-AI et al. (2024), and FP4 operators, *e.g.,* Wang et al. (2025), for LLM training has been conducted. They employed scaling in various ways to compensate for the limited dynamic range of internal datatype. Additionally, Wang et al. (2025) introduces the Differentiable Gradient Estimator (DGE) to address the limitations of the STE in low-bit settings of MXFP4.

B Pre-training setup and resource

For pre-training OPT2, Karpathy (2022) with commit `9755682b` was used as the starting point and `nvcr.io/nvidia/pytorch:24.10-py3` was used as the training environment. For pre-training Llama2, Liang et al. (2024) with commit `90567fc9` was used as the starting point and `ghcr.io/pytorch/pytorch-nightly` with a tag `2.7.0.dev20250107-cuda12.4-cudnn9-devel` was used as the training environment. We used A100-SXM4-40G, RTX 3090 and RTX 4090 GPUs for pre-training. Pre-training the OPT2-124M model with the AdamW optimizer and GaussWS[od] for 300B tokens took 1541 GPU-hour using four RTX 3090s. Pre-training the Llama2-134M model with the AdamW optimizer and GaussWS[all] for 300B tokens took 896 GPU-hours using eight RTX 4090s. Pre-training the Llama2-1B model with the AdamW optimizer and GaussWS[all] for 100B tokens took 2926 GPU-hours using eight RTX 4090s. Pre-training the baseline Llama2-1B model with the AdamW optimizer for 100B tokens took 1055 GPU-hours using four A100s.

C Detailed bitwidth

The resulting bitwidth b_t is presented in Figures C.1 through C.36. Each row corresponds to the 12 transformer blocks of the OPT2-124M model which was trained with GaussWS[all] and Adam-mini optimizer up to 300B tokens. Each column corresponds to the number of tokens processed, at 50B, 100B and 200B.

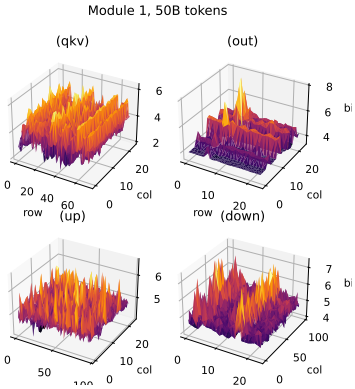


Figure C.1

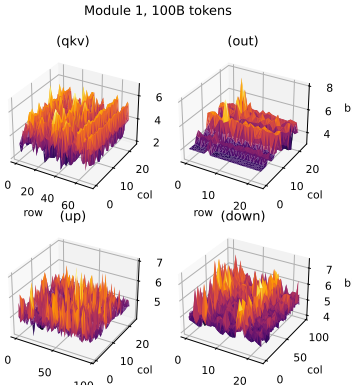


Figure C.2

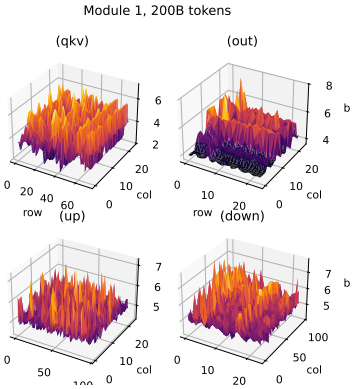


Figure C.3

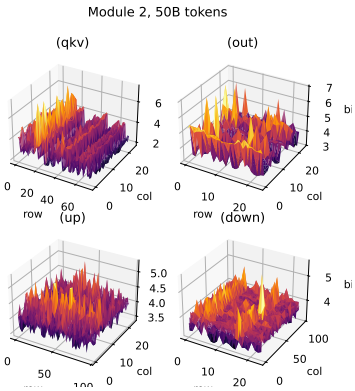


Figure C.4

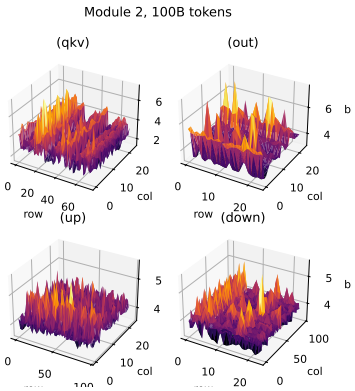


Figure C.5

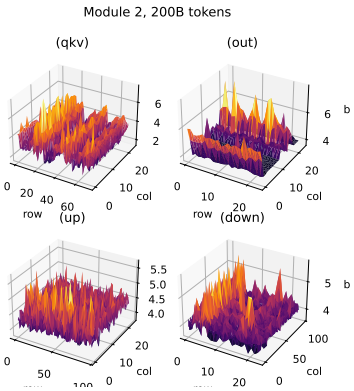


Figure C.6

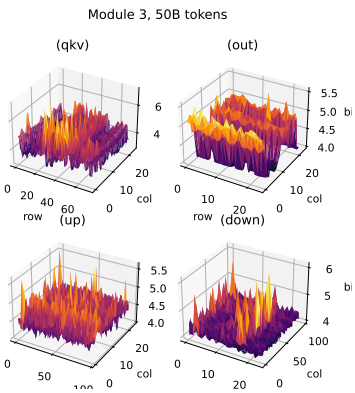


Figure C.7

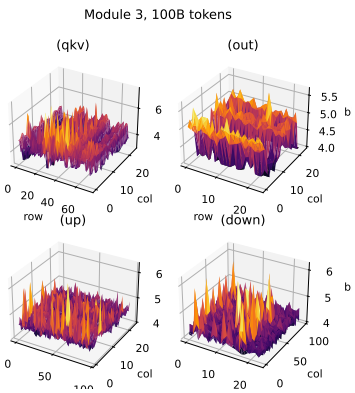


Figure C.8

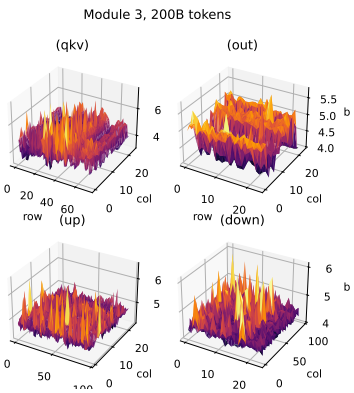


Figure C.9

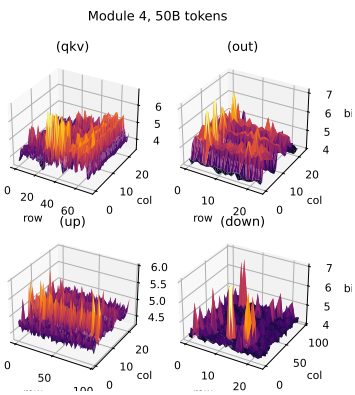


Figure C.10

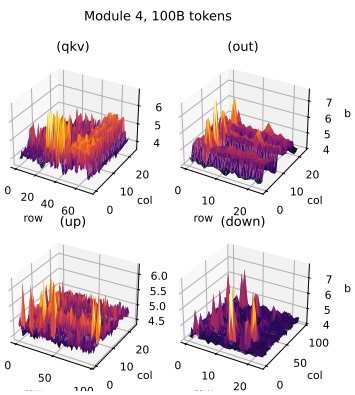


Figure C.11

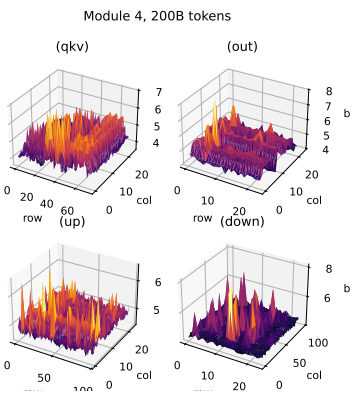


Figure C.12

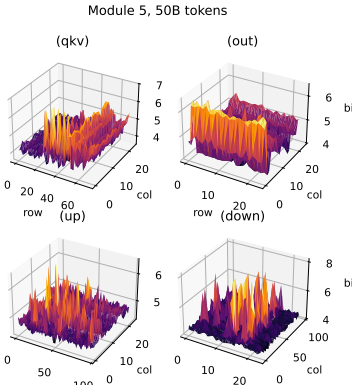


Figure C.13

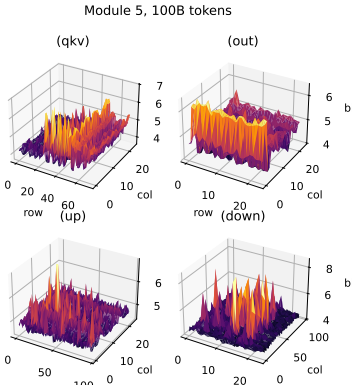


Figure C.14

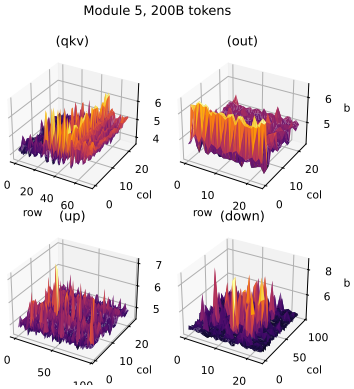


Figure C.15

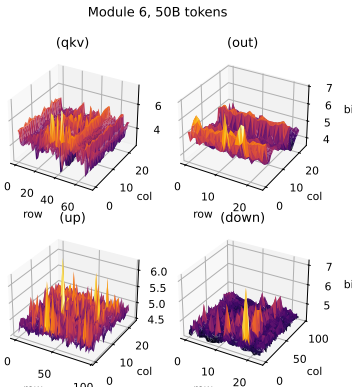


Figure C.16

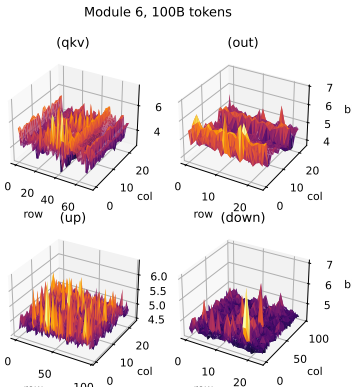


Figure C.17

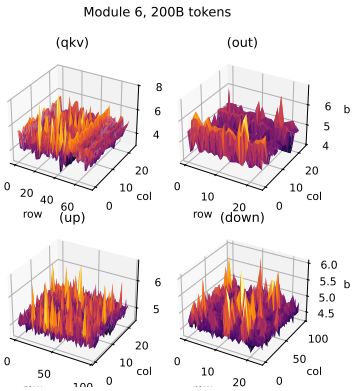


Figure C.18

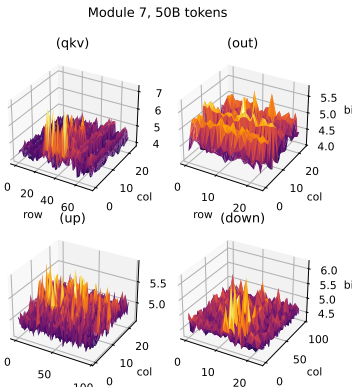


Figure C.19

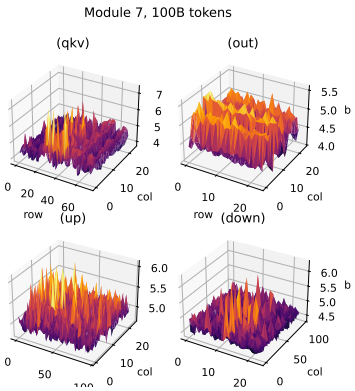


Figure C.20

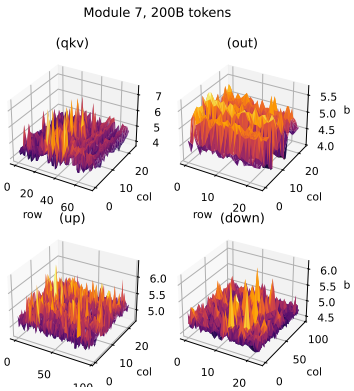


Figure C.21

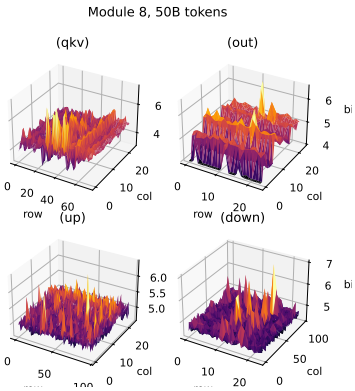


Figure C.22

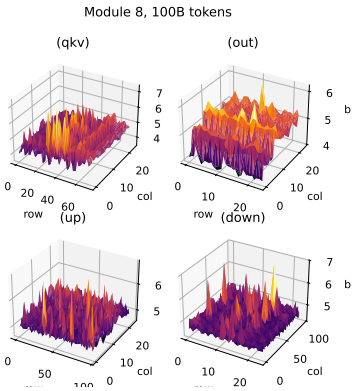


Figure C.23

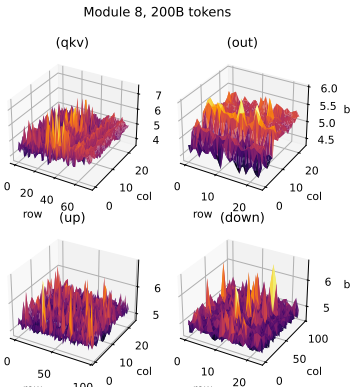


Figure C.24

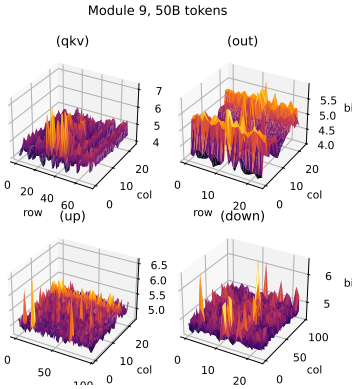


Figure C.25

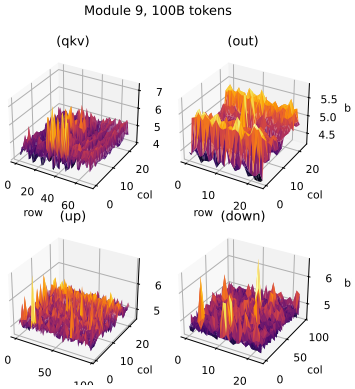


Figure C.26

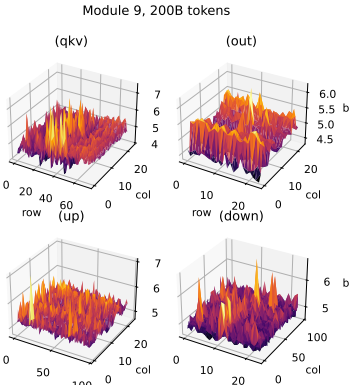


Figure C.27

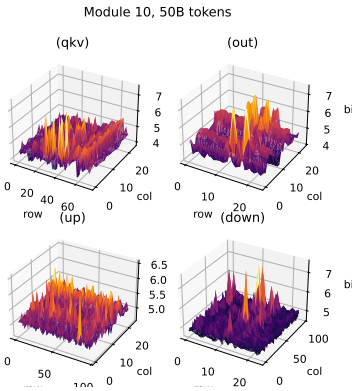


Figure C.28

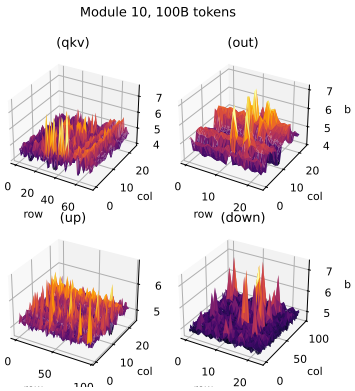


Figure C.29

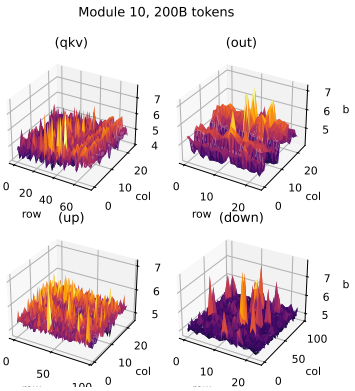


Figure C.30

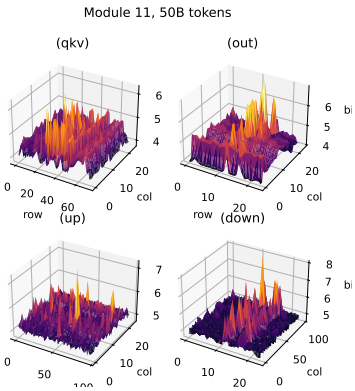


Figure C.31

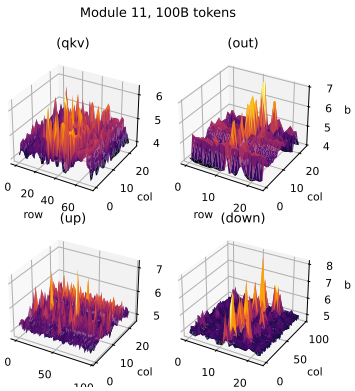


Figure C.32

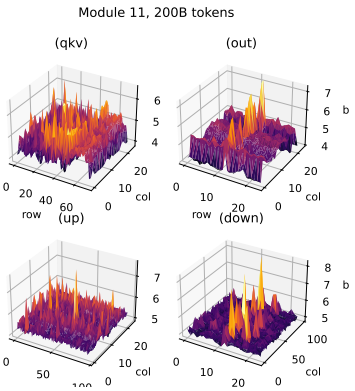


Figure C.33

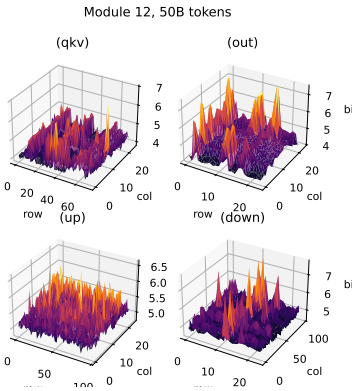


Figure C.34

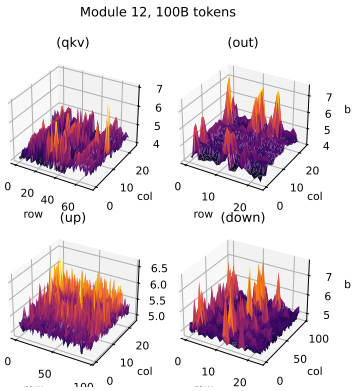


Figure C.35

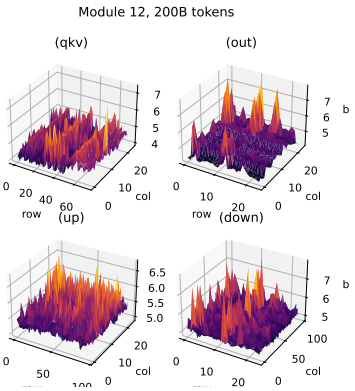


Figure C.36