NeuZip: Memory-Efficient Training and Inference with Dynamic Compression of Neural Networks

Yongchang Hao[♠*] **Yanshuai Cao**[◇] **Lili Mou**[♠♡]
[♠]Dept. Computing Science & Alberta Machine Intelligence Institute (Amii), University of Alberta
[◇]Borealis AI [♡]Canada CIFAR AI Chair
yongcha1@ualberta.ca
yanshuai.cao@borealisai.com doublepower.mou@gmail.com

Abstract

The performance of neural networks improves when more parameters are used. However, the model sizes are constrained by the available on-device memory during training and inference. Although applying techniques like quantization can alleviate the constraint, they suffer from performance degradation. In this work, we introduce NeuZip, a new weight compression scheme based on the entropy of floating-point numbers in neural networks. With NeuZip, we are able to achieve memory-efficient training and inference without sacrificing performance. Notably, we significantly reduce the memory footprint of training a Llama-3 8B model from 31GB to less than 16GB, while keeping the training dynamics fully unchanged. In inference, our method can reduce memory usage by more than half while maintaining near-lossless performance. Our code is publicly available.[1]

1 Introduction

Deep learning with neural networks has become the backbone of numerous artificial intelligence applications. The search for better-performing networks is a longstanding topic in deep learning. Without modifying the design, scaling up the number of parameters (e.g., number of hidden dimensions or layers) has been demonstrated as an effective practice to boost the performance of neural networks of the same kind [1]. This idea has been successfully applied to text, image, audio, and multi-modal tasks with a wide range of model architectures [2, 3, 4]. Recently, the number of parameters in the state-of-the-art models has become more than 100 billion or even a trillion parameters. For example, one of the state-of-the-art language models in 2020, GPT-3, has 175B parameters [4], growing by nearly 100 times compared with the largest Transformer architecture in the 2017 paper [5].

Despite the growth in the model size, the hardware capacity is not keeping up with the pace: the largest on-device memory of GPUs was 32GB in 2017, and is 80GB to this date in 2024, growing by only 2.5 times. The available hardware supply poses a limitation on the trainable model size, bottlenecking the scaling capacity. Although this problem can be alleviated by using more GPUs and sharding the model in multiple devices [6], such a practice introduces more communication overheads among GPUs, making large-scale distributed training less efficient. Therefore, saving the total memory usage is critical in scaling up neural networks.

The peak memory usage is dominated by three relatively independent parts: the optimizer, the saved activations for back-propagation, and the model itself. For the optimizer, there are already memory-efficient optimizers achieving a sublinear space complexity [7, 8]; for the activations, the memory can be saved by enabling activation checkpointing [9], which saves the storage by recomputing the

[*]Project done during Mitacs internship at Borealis AI.
[1]`https://github.com/BorealisAI/neuzip`

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

forward activations during the back-propagation. For the model parameters, there has not been an effective method to save the memory while preserving the ability to train the model. Recently, [10] proposed the quantized low-rank adaptation (QLoRA), which freezes the parameters using a 4-bit data type for the backbone pre-trained model. While significantly saving the memory for the model, it imposes a constraint on the overall change of the model to be low-rank, limiting the model capacity.

In this paper, we propose NeuZip, an algorithm to compress the neural networks while maintaining their full abilities. Specifically, each floating-point number is represented by three parts: the sign bit, the exponent bits, and the mantissa bits. Following the observation that weights are concentrated around zero [11], we demonstrate that this corresponds to the low-entropy nature of the exponent bits. We hence compress the exponent bits using the asymmetric numeral system (ANS [12]), a lossless compression algorithm that achieves a high throughput on parallel computing devices like GPUs. Since the compression is lossless, the memory reduction comes without compromising any precision loss and enables full-parameter training.

In addition to lossless compression for training, we also propose a lossy variant of NeuZip for inference that further reduces the memory footprint. Specifically, we control the relative change of each parameter by storing only the top-k significant bits of the mantissa. We empirically show that lossy NeuZip lies at the Pareto frontier of the memory–performance trade-off when compared with several state-of-the-art quantization baselines.

2 Our Approach

The Shannon entropy [13] is used to measure the "stochasticity" of a random variable with the following definition:

$$H(X) := \mathop{\mathbb{E}}_{X\sim p(X)}[-\log_2 p(X)] \tag{1}$$

for a random variable X with probability p. A lower entropy indicates a less stochasticity of a random variable. In fact, the entropy equals the minimum number of bits required, in expectation, to represent a random variable, therefore corresponding to data compressibility. For the non-concentrating random variable with all possible values sharing an equal probability, the entropy of which reaches the maximum value $\log_2 n$, where n is all possible values X can take. On the other hand, for highly-concentrating (e.g., fully deterministic) random variables, the entropy can be as low as 0.

2.1 Low-Entropy Nature of Neural Network Parameters

We argue that the parameters in neural network tend to have low entropy. First, parameters are typically initialized with Gaussian distribution for matrices [14, 15]. This encourages all weights to be centered around zero, effectively reducing the entropy (or randomness). In addition, regularization is also applied for better generalization ability. For example, the weight decay technique reduces the magnitudes of weights at every update iteration. Similarly in Bayesian inference, prior distributions (e.g., Gaussian and Laplace distributions) are often applied, imposing a zero-concentrated preference over the parameters.Even without explicit regularization, stochastic gradient descent (SGD) or its variants are shown to have the implicit regularization effect on neural networks, meaning the model parameters are implicitly encouraged to have smaller magnitudes during training [16, 17]. All the above effects and techniques lead to the following observation:

Observation 2.1 *Assuming neural network parameters are i.i.d. random variables, the entropy of the distribution is likely to be low.*

Specifically, each parameter is represented is represented by three components: the sign bit, the exponent bits, and the mantissa bits in the IEEE 754 standard [18].[2] Therefore, we conduct a fine-grained analysis and investigate the distribution of each component of a floating-point number in neural networks.

As shown in Figure 1, the sign bit has a high entropy as it is evenly distributed; hence, it is not compressible. For the exponent bits, there is a clear pattern that they demonstrate a low-entropy nature, carrying only less than 3 bits of information with 8 bits of capacity. For the mantissa bits,

[2] We use BF16 [11] in this paper.

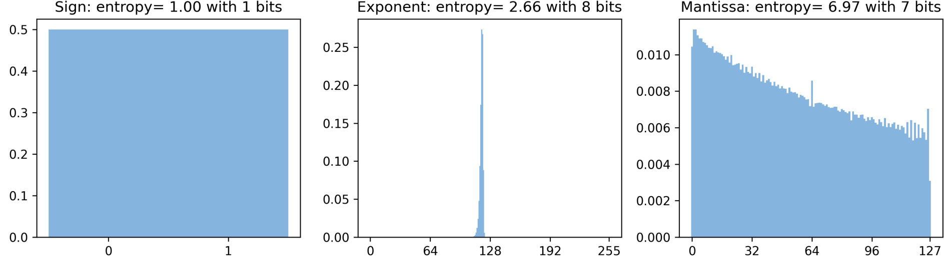


Figure 1: The histograms of different components of the parameters of LLama-3 8B model [19]. The x-axis is all possible binary values and the y-axis represent the frequency of each value.

they store nearly 7-bit information with 7-bit capacity. In fact, we shown in Appendix A that this is common in deep learning.

This phenomenon suggests that by simply compressing the exponents, we are able to recover the overall optimal compression ratio. In this example, an ideal compression algorithm is able to achieve a ratio as high as 1.501 (the sum of the three entropy values), only marginally below the overall compression ratio 1.505.

2.2 Lossless NeuZip: Compressing Exponents for Training

Compressed representation. Based on observation, we see that the number of bits per exponent is largely inflated compared with the information entropy. However, previous research demonstrates that the dynamic range provided by the 8-bit exponents are critical for neural networks [11]. We therefore propose to compress the exponent bits in a lossless manner based on the entropy. This practice mainly has three benefits: (1) it increases the throughput of compression as only part of the bits are processed by the compression; (2) it reduces the burden of maintaining the statistics of a large set of symbols (e.g., 256 symbols for 8-bit exponents versus 65,536 symbols for 16-bit representations), enabling a great efficiency of compression algorithms; (3) most importantly, it recovers most of the compressibility as shown in Figure 1.

Multi-layer neural networks. The compression alone does not save any memory for maintaining a single array. This is because, either compression or decompression, requires at least one buffer of the same size as the uncompressed array. In the scope of neural networks, the whole model is prohibitively large and it is infeasible to duplicate the memory. In NeuZip, however, we exploit the multi-layer structure of modern neural networks to avoid creating a large buffer. Without loss of generality, we focus on the linear function as a common building block in neural networks at layer l:

$$\boldsymbol{x}_l \leftarrow \boldsymbol{W}_l \boldsymbol{x}_{l-1} + \boldsymbol{b}_l, \tag{2}$$

where $\boldsymbol{W}_l \in \mathbb{R}^{m \times n}$ is the weight matrix, $\boldsymbol{b}_l \in \mathbb{R}^m$ is the bias vector of layer l, and $\boldsymbol{x}_l$ is the input of layer l. We propose to modify the compressed forward pass in the following form

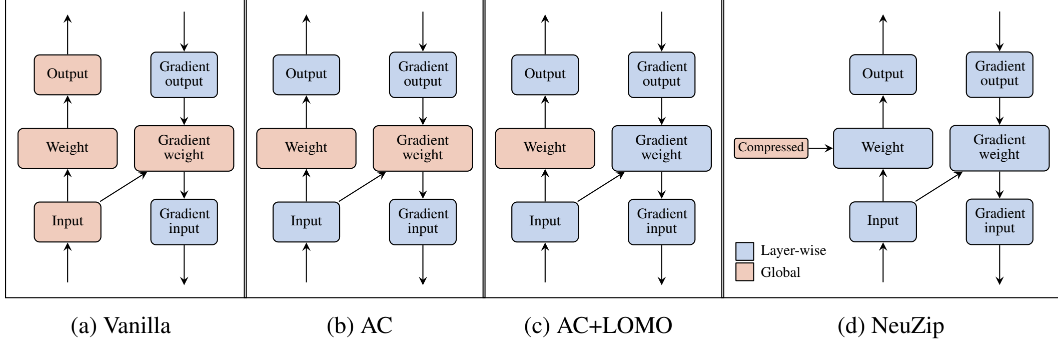
$$\hat{\boldsymbol{W}} \leftarrow \text{decompress}(\boldsymbol{c}_l) \tag{3}$$

$$\boldsymbol{x}_l \leftarrow \hat{\boldsymbol{W}} \boldsymbol{x}_{l-1} + \boldsymbol{b}_l, \tag{4}$$

where $\boldsymbol{c}_l$ is the compressed storage of the matrix $\boldsymbol{W}_l$. In this way, we only need to store $\boldsymbol{c}_i$ for each layer, enjoying low-memory usage. During each forward pass, weight matrices stay in the compressed form until the original data is needed, in which case it is decompressed into a temporary space $\hat{\boldsymbol{W}}$ for computation. As a result, the entire network is never fully decompressed at any point in time, making the overall forward pass memory efficient. The per-layer procedure is shown in Figure 2.

Note that although we alter the forward pass, the back-propagation for each linear layer is fully unaffected. This is because

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_l} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_l} \frac{\partial \boldsymbol{x}_l}{\partial \boldsymbol{W}_l} = (\nabla_{\boldsymbol{x}_l} \mathcal{L})\, \boldsymbol{x}_{l-1}^\top. \tag{5}$$



(a) Vanilla (b) AC (c) AC+LOMO (d) NeuZip

Figure 2: Reverse-mode automatic differentiation (e.g., back-propagation) with different memory-saving techniques for a linear layer. Blocks colored blue are loaded in memory temporarily for the calculation of this layer, whereas the blocks colored red are always in memory throughout training.

Therefore, we are able to obtain the gradient as long as the activations are saved. Similarly, we can also propagate the gradient of inputs with

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_{l-1}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_l} \frac{\partial \boldsymbol{x}_l}{\partial \boldsymbol{x}_{l-1}} = (\nabla_{\boldsymbol{x}_l} \mathcal{L})^\top \boldsymbol{W}_l, \tag{6}$$

where $\boldsymbol{W}_l$ can be constructed by decompression. It is worth noting that our NeuZip is compatible with activation checkpointing [9] by recomputing the activations, opening more opportunity for memory saving.

For weight updates, we decompress the matrix into the original floating-point format and compress the updated matrix again. This procedure is done in a layer-by-layer fashion, similar to LOMO [20]. The overall training procedure is described in Appendix B.

Compression algorithm. In our implementation, we choose to use the asymmetric numeral systems (ANS) [12] as our backbone compression algorithm because it can be easily parallelized and achieves a high throughput with parallel execution, making it an ideal candidate on deep learning accelerators like GPUs. Specifically, ANS encodes a sequence of symbols by treating them as base-n numbers. However, unlike the common numerical system that uses a uniform base for each digit, ANS treats every single digit with a different base $\lceil 1/\hat{p}_i \rceil$, where $\hat{\boldsymbol{p}}$ is the frequency of symbols. As a result, it achieves a near-optimal compression rate by suing around $1/\hat{p}_i$ bits for the ith symbol.

2.3 Lossy NeuZip: Additionally Truncating Mantissa for Inference

In the algorithm above, we show that the training of neural networks can be completely unaffected by lossless compression. On the other hand, inference is known to be less sensitive to precision loss compared with training [21, 22]. This enables further memory reduction of NeuZip by reducing the precision. In our study, we conduct a pilot experiment that perturbs each weight with a noise proportional to the weight magnitude. We observe that with a small noise ratio there is little or no effect on the overall performance (Appendix C). Motivated by this, we propose a variant of NeuZip that compresses mantissa in a lossy way during inference.

In its core, we simply round and truncate the mantissa to fewer bits. Specifically, we assume the original floating-point number f has an exponent e and mantissa m. After rounding, the mantissa is denoted by $\hat{m}$ and the resulting floating-point number is denoted by $\hat{f}$.

The rounding introduces an error expressed as:

$$|f - \hat{f}| = \left| 2^{e-127} \cdot \frac{m}{2^7} - 2^{e-127} \cdot \frac{\hat{m}}{2^7} \right| = 2^{e-134} \cdot |m - \hat{m}| \tag{7}$$

where $e - 127$ interprets the exponent bits e as an integer, which can be either positive, negative, or 0. In the fraction $m/2^7$, m is the significand (an unsigned integer) and 7 is the precision. It is straightforward to see that the relative error is given by

$$\frac{|f - \hat{f}|}{|f|} = \frac{2^{e-134} \cdot |m - \hat{m}|}{2^{e-134} \cdot |m|} = \frac{|m - \hat{m}|}{m}. \tag{8}$$

Suppose our rounding keeps k most significant bits in $\hat{m}$, the earliest point where $\hat{m}$ could differ from the original number m is at the $(k+1)$th bit. This means that the maximum possible relative change introduced by this rounding is $1/2^k$. Given that the mantissa bits are highly uniform as shown in Figure 1, such a practice resembles the weight perturbation based on relative magnitudes, justifying the rounding trick applied to mantissas.

In our implementation, we store the sign and mantissa bits together as a signed integer to minimize the requests of memory write. Further, given that modern architectures are mostly byte (8-bit) addressable, we pack multiple such signed integers into a single byte for memory efficiency. To align with an 8-bit byte, we let the precision after rounding to be $\{0, 1, 3\}$, ensuring that all the bits in a byte are utilized efficiently. We illustrate the process in Figure 7b.

Lastly, we enable a block-wise normalization technique [10], where a block is a chunk of weights that are stored contiguously in memory. Such block-wise normalization makes sure that the weight with the largest magnitude in a block will always be normalized to 1, invariant to mantissa rounding and truncation. The normalization coefficient—which handles mantissa while ignoring the exponent—is stored with 8 bits, and is used for de-normalization during the decompression of the weight. This strategy is based on the observation that larger weights play a more important role in neural networks [23].

3 Experiments

We empirically verify the effectiveness of NeuZip across different model architectures and datasets. Given the success of large language models, we mainly consider Transformer-based models for our experiments. We choose two designs of Transformer, decoder-only and encoder–decoder models, to show the generality of our method. All experiments are conducted on RTX A6000 GPUs where the uncompressed data type is BFloat16.

3.1 Lossless NeuZip for Pre-Training

Settings. We choose decoder-only models to evaluate our method on the pre-training task. We select 3 models with different sizes to study the scaling effect, including GPT-Neo 2.7B [24], Llama-3 8B [19], and LLama-2 13B [25]. For fair comparison, all competing methods are initialized with the same random weights.

For the task, we consider language modeling, which requires the model to predict the next token given the context. We use the Wikitext-2 dataset [26], where each data sample is a fixed-length sequence from an article on Wikipedia. We set the length to 1024 following the common practice [3].

For each experiment, we report the loss (negative log-likelihood) on unseen samples. To study memory saving, we report the peak memory usage for each run during the training process. The numbers are shown in gibibyte (GiB, 1024^3 bytes). We also report the speed by the number of iterations per second to demonstrate the time-efficiency of each method.

We apply the vanilla SGD update to all runs for efficiency. The activation checkpointing technique [9] is enabled by default. It is worth noting that pre-training these large models to the optimal performance is extremely expensive [6]. Given that our NeuZip training method is lossless, we only train the models for 1 epoch to showcase its effectiveness. We use the same hyper-parameters for all runs.

Results. We present the results in Table 1. We first test the vanilla training method, where only the activation checkpointing is applied (shown in Figure 2b). As shown, the vanilla training requires the highest amount of memory because it stores the uncompressed weights and gradients for all layers.

We also test the LOMO technique [20], which promptly updates the weights in a layer-by-layer fashion (shown in Figure 2c). This allows LOMO to reuse a buffer to store the gradients for each layer. As a result, LOMO approximately reduces the peak memory usage by the size of a model.

Finally, we apply our NeuZip on top of LOMO (shown in Figure 2d). For all models, NeuZip additionally reduces more than 20% percentage of memory compared with LOMO, accounting for a total memory reduction of more than 50%. Notably, NeuZip reduces the peak memory of training a Llama-2 13B model to less than 20GB, enabling training a 13B model on consumer-grade GPUs without any precision loss.

Table 1: Pre-training decoder-only models on the language modeling task. The loss numbers are calculated on the validation set with the cross-entropy loss. Memory is reported in GiB (1024^3 B). Speed represents the number of iterations per second. The **bold** numbers represent the top results.

	GPT-Neo-XL 2.7B			Llama-3 8B			LLama-2 13B		
Name	Loss	Mem	Speed	Loss	Mem	Speed	Loss	Mem	Speed
Vanilla	**8.81**	11.22	**0.96**	**8.61**	30.97	0.77	-	OOM	-
LOMO	**8.81**	6.97	0.94	**8.61**	19.47	**0.78**	**9.10**	26.26	**0.49**
+NeuZip Lossless	**8.81**	**5.54**	0.70	**8.61**	**15.25**	0.45	**9.10**	18.58	0.28

Table 2: Fine-tuning encoder–decoder models on the SQL generation task. The BLEU scores are calculated with SacreBLEU. Memory is reported in GiB (1024^3 B). Speed represents the number of iterations per second. The **bold** numbers represent the top results.

	T5 1B			T5 3B			T5 11B		
Name	BLEU	Mem	Speed	BLEU	Mem	Speed	BLEU	Mem	Speed
Vanilla	**79.9**	3.82	**3.69**	**85.1**	11.32	2.43	-	OOM	-
LOMO	**79.9**	2.75	3.68	**85.1**	7.07	**2.47**	**82.3**	25.95	**0.69**
+ NeuZip Lossless	**79.9**	**2.39**	2.02	**85.1**	**5.21**	1.33	**82.3**	**20.68**	0.46
QLoRA INT8	70.4	5.84	1.11	72.1	11.54	1.12	63.5	33.36	0.37
QLoRA FP4	70.1	3.63	1.70	72.1	7.35	1.74	63.3	22.73	0.58
QLoRA $FP4^2$	70.6	3.61	1.63	72.0	7.27	1.61	60.6	22.38	0.57
QLoRA NF4	70.4	3.63	1.83	71.2	7.35	1.65	59.4	22.73	0.57
QLoRA $NF4^2$	70.5	3.61	1.64	71.2	7.07	1.57	57.9	22.38	0.57

3.2 Lossless NeuZip for Fine-Tuning

Settings. A benefit of using lossless compression comes from retaining the pre-trained weight without any information loss. We conduct a fine-tuning experiment with encoder–decoder models to test the performance of our NeuZip on broader architectures. In particular, we choose three T5 models: T5 1B, T5 3B, and T5 11B [27], where the pre-trained parameters are used for initialization.

The T5 models are pre-trained on the C4 dataset [28], which is filtered to contain natural language only. To avoid data leaks from pre-training, we choose a non-natural language generation dataset for fine-tuning. Specifically, we use a public SQL generation dataset [29, 30] as the test bed. For each sample, the model is required to generate the SQL command from a human question. For example, the question could be "`CREATE TABLE head (age INTEGER)`. How many heads of the departments are older than 56 ?". The model is expected to generate "`SELECT COUNT(*) FROM head WHERE age > 56`". We feed the question and response into the encoder and decoder, respectively. The objective is to minimize the cross-entropy loss on the response.

Similar to the pre-training experiments, we also sweep the learning rate from 10^{-3} to 3×10^{-1} for each run. After fine-tuning, we generate with the model on the validation set with greedy decoding. The generated SQL commands are then compared with the ground truths by SacreBLEU [31], a metric that evaluates the similarity between corpora based on precision scores.

Results. The results are reported in Table 2. All baselines in the pre-training experiment (i.e., the vanilla training, LOMO, and NeuZip) are included in this table. Similar to the results in Section 3.1, they achieve the same BLEU scores for each model. Specifically, our NeuZip is able to train a 11B model within 24GB.

For fine-tuning, it is possible to apply other memory-efficient training techniques. For example, QLoRA [10] compresses the pre-trained model by using low-precision data types and train the LoRA modules only [32]. In our comparison experiment, we choose the widely used quantization data types for QLoRA, including INT8 [21], FP4, and NF4 [10]. We apply the LoRA modules [32] on all linear layers, where every LoRA rank is set to 8 to control the memory usage.[3] As shown in the second half of Table 2, all quantization methods underperform NeuZip in terms of both generation quality and memory usage. In terms of time efficiency, some quantization methods are slower than others, but in general, they are in the same magnitude as our method. Overall, NeuZip achieves the least memory usage while maintaining the highest performance. The results strongly suggests the practicality of our NeuZip.

[3]It should be noted that the down-projection matrices in each T5 feed-forward network are not quantized for stability, as otherwise the model performance is seriously jeopardized. See `https://github.com/huggingface/transformers/issues/20287` for more details.

Table 3: Evaluating lossy NeuZip on different models and tasks. 'PPL" represents the perplexity values. Memory is reported in GiB. Speed represents the number of iterations per second. The **bold** numbers represent the top results, whereas the underlined numbers are the second-best ones.

(a) Evaluating decoder-only models on the language modeling task. Here, the perplexities are adjusted to word level to compare across different tokenizations.

	Llama-3 8B			Llama-2 13B			Yi-1.5 34B		
Name	PPL	Mem	Speed	PPL	Mem	Speed	PPL	Mem	Speed
Vanilla	**9.89**	15.08	**5.07**	**10.87**	24.36	**3.59**	-	OOM	-
Quant INT8	10.07	8.63	3.54	10.97	12.74	2.27	10.87	33.41	1.13
Quant FP4	11.51	5.77	3.45	11.38	7.37	1.87	11.57	19.54	**1.75**
Quant NF4	10.75	5.77	3.38	11.15	7.37	1.83	11.06	19.54	1.67
Quant FP4[2]	11.50	5.44	3.41	11.38	6.87	1.86	11.57	18.11	1.61
Quant NF4[2]	10.75	5.44	3.34	11.15	6.87	1.81	11.06	18.11	1.54
NeuZip 0-bit	13.64	**5.24**	3.44	12.46	**6.30**	1.87	12.06	**16.20**	0.94
NeuZip 1-bit	10.77	6.05	3.38	11.17	7.77	1.86	11.04	20.14	0.93
NeuZip 3-bit	9.93	7.70	3.38	10.90	10.73	1.84	10.76	27.92	0.93
NeuZip 7-bit (lossless)	**9.89**	10.95	3.39	**10.87**	16.66	1.84	10.72	43.40	0.94

(b) Evaluating encoder–decoder models on the language modeling task. Since all models use the same tokenizer, we reported perplexities at the token level for simplicity.

	T5 1B			T5 3B			T5 11B		
Name	PPL	Mem	Speed	PPL	Mem	Speed	PPL	Mem	Speed
Vanilla	**2.614**	1.37	**23.73**	**2.571**	5.31	**19.86**	**2.568**	21.06	**6.20**
Quant INT8	2.615	1.28	4.24	2.573	4.94	4.28	2.569	19.59	2.58
Quant NF4	2.632	1.08	11.64	2.588	4.12	11.82	2.579	16.28	4.48
Quant FP4	2.646	1.08	11.92	2.594	4.12	11.99	2.585	16.28	4.59
Quant FP4[2]	2.646	1.05	10.39	2.594	4.03	9.72	2.585	15.93	4.52
Quant NF4[2]	2.632	1.05	10.39	2.587	4.03	9.96	2.579	15.93	4.39
NeuZip 0-bit	2.731	**0.40**	11.82	2.668	**1.41**	8.70	2.651	**5.35**	3.24
NeuZip 1-bit	2.641	0.48	11.68	2.591	1.78	8.61	2.581	6.65	3.21
NeuZip 3-bit	**2.614**	0.66	11.99	2.574	2.42	8.60	2.569	9.27	3.19
NeuZip 7-bit (lossless)	**2.614**	0.99	11.55	**2.571**	3.73	8.77	**2.568**	14.46	3.23

3.3 Lossy Compression for Inference

As mentioned in Section 2.3, the inference process is less sensitive in precision loss, which provides an opportunity for compressing mantissa in a lossy fashion during inference. We evaluate the performance of our lossy NeuZip in such scenarios.

Settings. Following the settings in previous sections, we test our approach with both decoder-only and encoder–decoder architectures. For the decoder-only models, we select the LLama-3 8B [19], LLama-2 13B [25], and Yi-1.5 34B [33]. For the encoder–decoder architecture, we use the T5 1B, 3B, and 11B models as in Section 3.2.

Since all decoder-only models are trained for language modeling, we evaluate the performance with language modeling tasks. Specifically, we test all methods on the Wikitext-2 validation set [26] following Section 3.1, where each sequence consists of 1024 tokens. On the other hand, the encoder–decoder models (T5 series) contain multiple tasks in pre-training. Since they excel at zero-shot translation, we evaluate them on the WMT14 En-De translation task [34], where each source sentence is prepended with "translate from English to German:" based on the pre-training format [27].

Following the standard evaluation pipeline for lossy compression [35, 22], we evaluate all models with the perplexity metric, which is sensitive to how distorted the compressed model is.

Results. The results for decoder-only and encoder–decoder models are shown in Tables 3a and 3b, respectively. We see that the vanilla (uncompressed BFloat16) models achieve the best perplexity scores in all experiments at a cost of the excessive memory usage. For quantization methods, we choose the same INT8 [21], FP4, and NF4 [10] data types mentioned in Section 3.2. In general, quantization methods suffer from notable perplexity degradation. Although the INT8 variant [21] manages to better preserve the perplexity, it uses around 50% more memory compared with other quantization methods.

For our lossy NeuZip, we set three different levels of precision: 0-bit, 1-bit, and 3-bit mantissa preserved. We choose these values because they are aligned in 8-bit byte arrays (discussed in

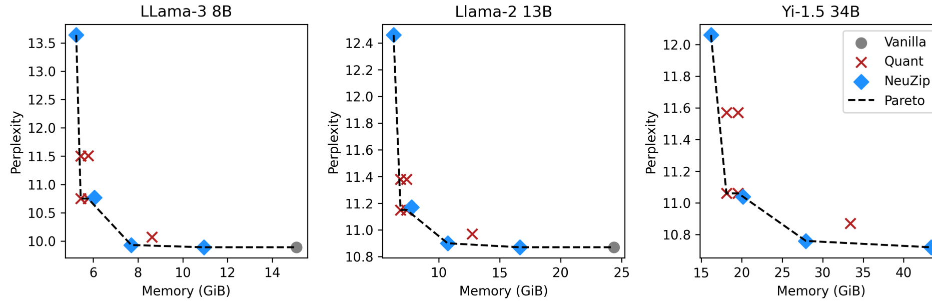


Figure 3: The trade-off between memory and performance for different methods.

Section 2.3). All these variants use a block size of 512 for normalization. We additionally include the lossless NeuZip (7-bit mantissa) for a full comparison. As shown in the table, our lossy NeuZip demonstrates a spectrum of memory saving and performance preservation. The 0-bit NeuZip attains the best memory efficiency in all experiments, whereas the lossless 7-bit NeuZip obtains the best perplexity scores. Notably, the 3-bit NeuZip achieves nearly lossless performance in all experiments while using less than 50% memory compared with the uncompressed model. The results confirm the effectiveness of our method.

4 Related Work

Model compression. Previous work has explored different techniques to reduce the memory usage of neural networks, including knowledge distillation [36] and pruning [37]. Most related to our work is the quantization technique, which represents each parameter with fewer bits; common approaches include k-means-based quantization [38], linear quantization [38], and mixed precision quantization [21, 10]. When training data are available, one may incorporate the quantization into the training process to improve performance [39, 35]. In this paper, our NeuZip compression is a zero-shot method, and therefore, our experiments consider the widely used zero-shot quantization methods [21, 10] for fair comparison. We leave the utilization of additional data of NeuZip to future work.

Memory-efficient optimizers. The optimizer also occupies a considerable amount of memory during training [6]. To address this, memory-efficient optimizers [7, 40, 8] are developed to reduce the memory footprint of training. Our NeuZip is orthogonal to these optimization techniques, as it can be seamlessly combined with any of these methods for further memory saving. In particular, the lossless NeuZip is expected to have exactly the same results with less memory.

Parameter-efficient training. Another line of research saves memory by training a subset of parameters [41, 42] so the optimizer only stores information about a small set of trainable parameters. One notable example is the low-rank adaptation (LoRA [32]). However, such a practice restricts the optimization space of parameters, and thus usually leads to significant performance degradation. Moreover, low-rank methods are unsuitable for pre-training.

It is important to mention that memory-efficient optimizers and parameter-efficient training cannot reduce the memory cost during inference. By contrast, our NeuZip is suitable for both training and inference.

5 Conclusion

Summary. In this work, we present NeuZip, a novel compression scheme for neural networks that achieves memory-efficient training and inference. By analyzing the floating-point structures, we propose to compress the exponent in a lossless way and to compress the mantissa in a lossy way. The lossless variant of our NeuZip may be applied to both training and inference, while yielding exactly the same result as the uncompressed model. The lossy NeuZip provides additional memory saving for inference, achieving superior memory–performance trade-off.

Acknowledgments

The research is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), a Mitacs Accelerate project, the Amii Fellow Program, the Canada CIFAR AI Chair Program, an Alberta Innovates Program, and the Digital Research Alliance of Canada (alliancecan.ca).

References

[1] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv: 2001.08361*, 2020. URL `https://arxiv.org/abs/2001.08361`.

[2] Jiahui Yu, Yuanzhong Xu, Jing Yu Koh, Thang Luong, Gunjan Baid, Zirui Wang, Vijay Vasudevan, Alexander Ku, Yinfei Yang, Burcu Karagol Ayan, B. Hutchinson, Wei Han, Zarana Parekh, Xin Li, Han Zhang, Jason Baldridge, and Yonghui Wu. Scaling autoregressive models for content-rich text-to-image generation. *TMLR*, 2022. URL `https://openreview.net/forum?id=AFDcYJKhND`.

[3] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019. URL `https://openai.com/research/better-language-models`.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *NeurIPS*, pages 1877–1901, 2020. URL `https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017. URL `https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html`.

[6] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. *International Conference For High Performance Computing, Networking, Storage And Analysis*, 2019. URL `https://arxiv.org/abs/1910.02054`.

[7] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *ICML*, pages 4596–4604, 2018. URL `https://proceedings.mlr.press/v80/shazeer18a.html`.

[8] Yongchang Hao, Yanshuai Cao, and Lili Mou. Flora: Low-rank adapters are secretly gradient compressors. In *ICML*, 2024. URL `https://openreview.net/forum?id=uubBZKM99Y`.

[9] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. URL `https://arxiv.org/abs/1604.06174`.

[10] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *NeurIPS*, 2023. URL `https://openreview.net/forum?id=OUIFPHEgJU`.

[11] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. A study of BFloat16 for

deep learning training. *arXiv preprint arXiv: 1905.12322*, 2019. URL https://arxiv.org/abs/1905.12322.

[12] Jarek Duda. Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv: 1311.2540*, 2013. URL https://arxiv.org/abs/1311.2540.

[13] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948. URL https://doi.org/10.1002/j.1538-7305.1948.tb01338.x.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, pages 249–256, 2010. URL https://proceedings.mlr.press/v9/glorot10a.html.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *ICCV*, pages 1026–1034, 2015. URL https://doi.org/10.1109/ICCV.2015.123.

[16] Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *JMLR*, 19(70):1–57, 2018. URL http://jmlr.org/papers/v19/18-188.html.

[17] Gal Vardi and Ohad Shamir. Implicit regularization in relu networks with the square loss. In *COLT*, pages 4224–4258, 2021. URL http://proceedings.mlr.press/v134/vardi21b.html.

[18] IEEE. IEEE standard for floating-point arithmetic, 2019. URL https://doi.org/10.1109/IEEESTD.2019.8766229.

[19] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, and et al. The Llama 3 herd of models. *arXiv preprint arXiv: 2407.21783*, 2024. URL https://arxiv.org/abs/2407.21783.

[20] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. Full parameter fine-tuning for large language models with limited resources. *arXiv preprint arXiv: 2306.09782*, 2023. URL https://arxiv.org/abs/2306.09782.

[21] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. GPT3.int8(): 8-bit matrix multiplication for Transformers at scale. In *NeurIPS*, 2022. URL https://openreview.net/forum?id=dXiGWqBoxaD.

[22] Tim Dettmers and Luke Zettlemoyer. The case for 4-bit precision: k-bit inference scaling laws. *ICML*, 2023. URL https://proceedings.mlr.press/v202/dettmers23a.html.

[23] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/ae0eb3eed39d2bcef4622b2499a05fe6-Paper.pdf.

[24] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. GPT-Neo: Large scale autoregressive language modeling with Mesh-Tensorflow, 2021. URL https://doi.org/10.5281/zenodo.5297715.

[25] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan,

Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv: 2307.09288*, 2023. URL https://arxiv.org/abs/2307.09288.

[26] Stephen Merity, Caiming Xiong, James Bradbury, and R. Socher. Pointer sentinel mixture models. *ICLR*, 2016. URL https://arxiv.org/abs/1609.07843.

[27] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text Transformer. *JMLR*, 21(1):5485–5551, 2020. URL http://jmlr.org/papers/v21/20-074.html.

[28] Zhaojiang Lin, Andrea Madotto, and Pascale Fung. Exploring versatile generative language model via parameter-efficient transfer learning. In *EMNLP Findings*, pages 441–459, 2020. URL https://aclanthology.org/2020.findings-emnlp.41.

[29] Victor Zhong, Caiming Xiong, and Richard Socher. Seq2SQL: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017. URL https://arxiv.org/abs/1709.00103.

[30] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *EMNLP*, pages 3911–3921, 2018. URL https://aclanthology.org/D18-1425.

[31] Matt Post. A call for clarity in reporting BLEU scores. In *WMT*, pages 186–191, 2018. URL https://aclanthology.org/W18-6319.

[32] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *ICLR*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

[33] Alex Young, Bei Chen, Chao Li, Chengen Huang, Ge Zhang, Guanwei Zhang, Heng Li, Jiangcheng Zhu, Jianqun Chen, Jing Chang, Kaidong Yu, Peng Liu, Qiang Liu, Shawn Yue, Senbin Yang, Shiming Yang, Tao Yu, Wen Xie, Wenhao Huang, Xiaohui Hu, Xiaoyi Ren, Xinyao Niu, Pengcheng Nie, Yuchi Xu, Yudong Liu, Yue Wang, Yuxuan Cai, Zhenyu Gu, Zhiyuan Liu, and Zonghong Dai. Yi: Open foundation models by 01.ai. *arXiv preprint arXiv: 2403.04652*, 2024. URL https://arxiv.org/abs/2403.04652.

[34] Ondrej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Ale s Tamchyna. Findings of the 2014 workshop on statistical machine translation. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 12–58, 2014. URL http://www.aclweb.org/anthology/W/W14/W14-3302.

[35] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained Transformers. In *ICLR*, 2023. URL https://openreview.net/forum?id=tcbBPnfwxS.

[36] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv: 1503.02531*, 2015. URL https://arxiv.org/1503.02531.

[37] Woosuk Kwon, Sehoon Kim, Michael W Mahoney, Joseph Hassoun, Kurt Keutzer, and Amir Gholami. A fast post-training pruning framework for Transformers. In *NeurIPS*, pages 24101–24116, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/987bed997ab668f91c822a09bce3ea12-Abstract-Conference.html.

[38] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In *ICLR*, 2016. URL http://arxiv.org/abs/1510.00149.

[39] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. SmoothQuant: Accurate and efficient post-training quantization for large language models. In *ICML*, 2023. URL `https://arxiv.org/abs/2211.10438`.

[40] Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. GaLore: Memory-efficient LLM training by gradient low-rank projection. In *ICML*, 2024. URL `https://arxiv.org/abs/2403.03507`.

[41] N. Houlsby, A. Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and S. Gelly. Parameter-efficient transfer learning for NLP. In *ICML*, pages 2790–2799, 2019. URL `https://proceedings.mlr.press/v97/houlsby19a.html`.

[42] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. BitFit: Simple parameter-efficient fine-tuning for Transformer-based masked language-models. In *ACL*, volume 2, pages 1–9, 2022. URL `https://aclanthology.org/2022.acl-short.1`.

[43] Vilfredo Pareto. *Manual of Political Economy: A Variorum Translation and Critical Edition*. Oxford University Press UK, 2014. URL `https://global.oup.com/academic/product/manual-of-political-economy-9780198867661`.

[44] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv: 2009.03300*, 2020. URL `https://arxiv.org/abs/2009.03300`.

A Inspecting the Entropy on More Models

Random initialization. When training from scratch, the parameters are randomly initialized. To verify the compressibility in this case, we check the parameter entropy of a randomly initialized model with the same architecture as Lllama-3. The initialization methods follow the standard procedure provided in the Hugging Face library. The results show a similar pattern to what the released Llama model has, suggesting the compressibility with NeuZip occurs even with random weights.

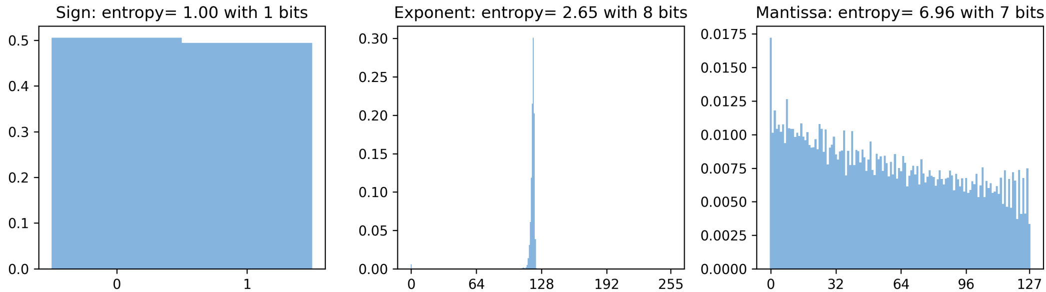


Figure 4: The histograms of different floating-point components of the parameters of a randomly initialized Llama-3 8B model.

Diffusion. We also inspect the parameter entropies beyond Transformer models. In Figure 5, we check all four models in a diffusion pipeline. We see that the low-entropy exponents not only occur in Transformer models but other architectures like convolution-based VAE and U-Net models.

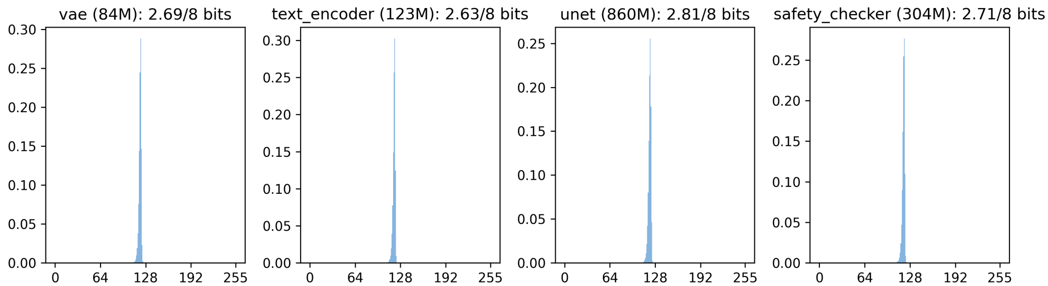


Figure 5: The histograms of the exponent bits in different components of Stable Diffusion 1.5 model. We omit the sign and mantissa bits for simplicity as we do not compress them based on their entropies.

Both experiments show that the occurrence of low-entropy components is a common phenomenon in deep learning.

B The Algorithm for Training with Lossless NeuZip

In this section, we describe the forward-backward procedure of NeuZip. First, we compress all the linear layers in the original model and store the compressed information on-device. During the training iterations, we decompress the compressed weights in a layer-by-layer manner for the forward pass. For the backward pass, the input is recalculated again following the forward pass like activation checkpointing [9]. A linear operation calculates the gradients for both the weight matrix and input. To do so, we need to decompress the weight again, which is used to calculate the gradient of input. After the gradient is calculated, we directly update the weight without storing it similar to LOMO [20].

C The Tolerance of Random Perturbation

In this experiment, we would like to explore the sensitivity of neural network weights to random perturbations. For each parameter, we have two types of magnitudes: absolute and relative magnitudes.

Algorithm 1 Memory-efficient training with NeuZip

Require: number of linear layers L, linear layer weights $\{\boldsymbol{W}_i\}_{i=1}^{L}$.
Require: data stream $\mathcal{D}$ that yields training data $\boldsymbol{x}$ for each iteration
  ▷ Initialization
1: **for** $l \in 1 \dots L$ **do**
2:   $\boldsymbol{s}_l, \boldsymbol{e}_l, \boldsymbol{m}_l \leftarrow \text{split}(\boldsymbol{W}_l)$ ▷ Split each element in the matrix into three components
3:   $\boldsymbol{c}_l \leftarrow \text{compression}(\boldsymbol{e}_l)$ ▷ Compress the exponents losslessly
4:   $\text{store}(\boldsymbol{s}_l, \boldsymbol{c}_l, \boldsymbol{m}_l)$ ▷ Store the compressed exponents $\boldsymbol{c}_L$ on device
5: **end for**
  ▷ Training loop
6: **for** $\boldsymbol{x}$ in $\mathcal{D}$ **do**
7:   ▷ Model forward
8:   $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$
9:   **for** $l \in 1 \dots L$ **do**
10:     $\hat{\boldsymbol{e}} \leftarrow \text{decompression}(\boldsymbol{c}_l)$ ▷ Decompress the exponents using temporary space
11:     $\hat{\boldsymbol{W}} \leftarrow \text{merge}(\boldsymbol{s}_l, \hat{\boldsymbol{e}}_l, \boldsymbol{m}_l)$ ▷ Concatenate into a floating-point number matrix using temporary space
12:     $\boldsymbol{x}_l \leftarrow \hat{\boldsymbol{W}}^\top \boldsymbol{x}_{l-1} + \boldsymbol{b}_l$ ▷ Linear calculation
13:     save_for_backward($\boldsymbol{x}_l$) ▷ Label the variable required for back-propagation
14:   **end for**
15:   ▷ Model backward and update
16:   $\Delta_{\boldsymbol{x}} \leftarrow \partial \mathcal{L} / \partial \boldsymbol{x}_L$ ▷ Calculate the gradient w.r.t. the model output
17:   **for** $l \in L \dots 1$ **do**
18:     $\hat{\boldsymbol{e}} \leftarrow \text{decompression}(\boldsymbol{c}_l)$ ▷ Decompress the exponents using temporary space
19:     $\hat{\boldsymbol{W}} \leftarrow \text{merge}(\boldsymbol{s}_l, \hat{\boldsymbol{e}}_l, \boldsymbol{m}_l)$ ▷ Concatenate into a floating-point number matrix using temporary space
20:     $\Delta_{\boldsymbol{W}} \leftarrow (\Delta_{\boldsymbol{x}}) \boldsymbol{x}_{l-1}^\top$ ▷ Calculate gradient by Equation (5)
21:     $\hat{\boldsymbol{W}} \leftarrow \hat{\boldsymbol{W}} - \text{optimizer}(\Delta_{\boldsymbol{W}})$ ▷ Update the weight on-the-fly
22:     $\boldsymbol{s}_l, \boldsymbol{e}_l, \boldsymbol{m}_l \leftarrow \text{split}(\hat{\boldsymbol{W}})$ ▷ Split each element in the matrix into three components again
23:     $\boldsymbol{c}_l \leftarrow \text{compression}(\boldsymbol{e}_i)$ ▷ Compress the exponents losslessly again
24:     $\text{store}(\boldsymbol{s}_l, \boldsymbol{c}_l, \boldsymbol{m}_l)$ ▷ Replace the stored components for layer l on device
25:     $\Delta_{\boldsymbol{x}} \leftarrow \hat{\boldsymbol{W}} \Delta_{\boldsymbol{x}}$ ▷ Calculate the gradient of input for next layers
26:   **end for**
27: **end for**

The former one represents the actual numerical error, whereas the second one is calculated based on the original value. For example, when the original value is -1.5, an absolute magnitude of 0.125 means the perturbed range is $[-1.5 - 0.125, -1.5 + 0.125]$. On the other hand, a relative magnitude of 0.125 means the perturbed range is $[-1.5 * (1 + 0.125), -1.5 * (1 - 0.125]$. We conduct such a experiment with the perturbation grid in Figure 6. For each cell, we choose the maximum error between relative error and absolute value for perturbing. A random value is sampled from the perturbed range uniformly as the perturbation. The weight value is then set to the random sample.

As shown in Figure 6, we see a clear pattern that the model tends to tolerate the relative change rather than the absolute change.

D The Storage for Lossless and Lossy Compression

In this section, we describe the underlying storage layout for NeuZip in Figure 7.

Essentially, each BFloat16 number is first split into an exponent and a signed mantissa. We group all the exponents in the matrix and perform the lossless compression. The signed mantissa is optionally truncated, depending on the required precision. The signed mantissa is then stored separately in memory.

E In-depth analyses

The memory–performance trade-off. In Section 3.3, we observe that the performance is generally decreased with less memory usage. We analyze this trade-off of our NeuZip as well as quantization methods in Figure 3. Note that the optimal methods are the ones on the Pareto frontier [43], i.e., the

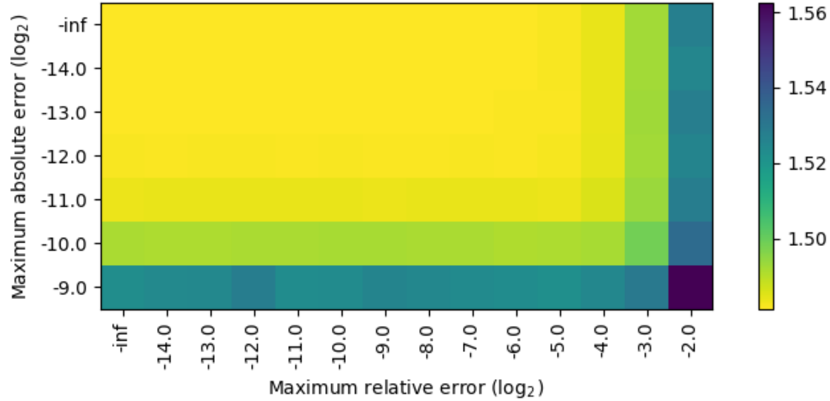
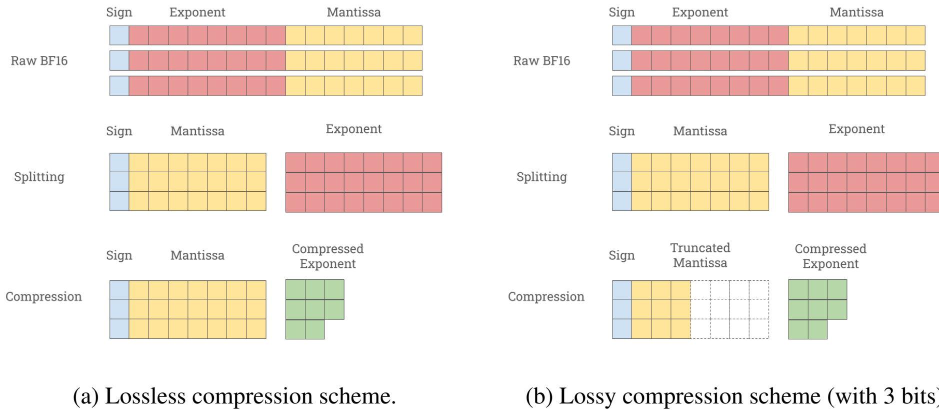


Figure 6: Evaluating the byte-level perplexity with perturbed LLama-3 8B model [19] on Wikitext-2 [26]. Each parameter is perturbed with controlled noises. Both the x- and y-axes are log-scale with base 2.



(a) Lossless compression scheme.

(b) Lossy compression scheme (with 3 bits).

Figure 7: The storage structures for NeuZip.

Table 4: The effect of block size.

	Block 32		Block 64		Block 128		Block 256		Block 512	
Name	PPL	Mem	PPL	Mem	PPL	Mem	PPL	Mem	PPL	Mem
NeuZip 0-bit	6.341	35.7	6.694	34.6	6.853	34.2	7.639	33.8	7.104	33.5
NeuZip 1-bit	-	OOM	4.611	42.7	4.662	42.2	4.640	41.8	4.649	41.4

more bottom-left, the better. In addition to measuring the perplexity, we also include a preliminary study by evaluating the end-to-end performance on the MMLU dataset [44] in Appendix F.

As shown, three out of four NeuZip variants are on the Pareto frontier, with the remaining one staying fairly close to the frontier. On the other hand, there is only one quantization technique that lies on the Pareto frontier. This result demonstrates that our NeuZip generally achieves a better memory–performance trade-off than quantization.

The effect of block size in lossy compression. As introduced in Section 2, we apply normalization to lossy NeuZip to ensure the weight with the largest absolute value will not be affected by truncation. We show the effect of block size in this experiment with a giant model, Llama-3 70B evaluated on the Wikitext-2 dataset.

As seen in Table 4, a smaller block size clearly leads to better performance at the cost of compromising memory efficiency due to the overhead of storing normalization coefficients. Therefore, the block-

wise normalization provides a more fine-grained trade-off between memory and performance by varying the block size.

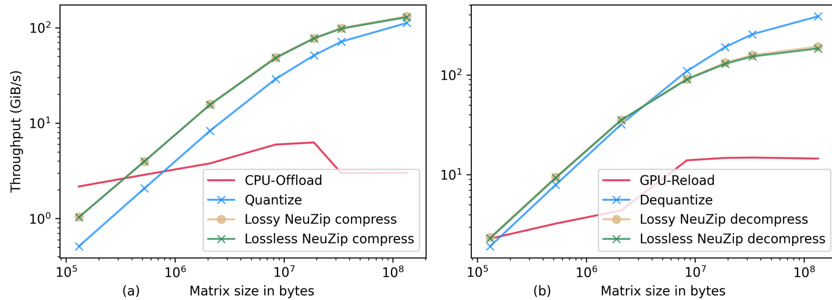


Figure 8: The throughput experiment. (a) Comparison of CPU-offloading, quantization, lossy NeuZip compression, and lossless NeuZip compression. (b) Comparison of GPU-reloading, de-quantization, lossy NeuZip decompression, and lossless NeuZip decompression.

Throughputs of NeuZip. In addition to overall time efficiency presented in Tables 1–3, we analyze the throughput of matrix compression and decompression with our NeuZip, in comparison with the throughput of matrix quantization and de-quantization based on the NF4 data type [10] using the popular library `bitsandbytes`.[4] We additionally include the CPU-offloading technique as a baseline, which lowers the GPU memory pressure by transferring data to CPU and reloading them to GPU when needed. Figure 8 measures the throughput of matrix processing in GiB/s when we vary the matrix size from 10^5 to 10^8 bytes.

We see that CPU-offloading is generally slow across different sizes of matrices. This is due to the bottleneck of CPU–GPU communication through PCIe. For quantization, the `bitsandbytes` package has been highly optimized for GPU, and its throughput is one magnitude higher than the CPU-offloading technique when the matrix size is large. Profoundly, our NeuZip achieves the highest throughput for compression among all methods (Figure 4a), and a high throughput for decompression similar to de-quantization (Figure 4b). The results suggest that our NeuZip, albeit causing overhead compared with uncompressed vanilla models, is still highly efficient in practice.

F Evaluating on MMLU

We provide the results on MMLU [44] in Figure 9. Here, the theoretical optimal point should be at the top left corner.

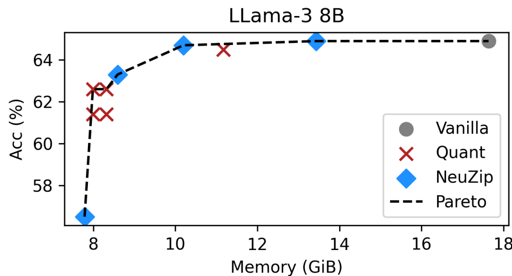


Figure 9: The memory–performance trade-off on the MMLU dataset.

Similar to the results in Appendix E, all of our NeuZip variants are on the Pareto frontier, suggesting the optimal trade-off between memory and performance.

[4] Available at `https://github.com/bitsandbytes-foundation/bitsandbytes`