8-BIT OPTIMIZERS VIA BLOCK-WISE QUANTIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Stateful optimizers maintain gradient statistics over time, e.g., the exponentially smoothed sum (SGD with momentum) or squared sum (Adam) of past gradient values. This state can be used to accelerate optimization significantly, compared to plain stochastic gradient descent, but uses memory that might otherwise be allocated to model parameters, thereby limiting the maximum size of models trained in practice. In this paper, we develop the first optimizers that use 8-bit statistics while maintaining the performance levels of using 32-bit optimizer states. To overcome the resulting computational, quantization, and stability challenges, we develop block-wise dynamic quantization. Block-wise quantization divides input tensors into smaller blocks that are independently quantized. Each block is processed in parallel across cores, yielding faster optimization and high precision quantization. To maintain stability and performance, we combine block-wise quantization with two additional changes: (1) dynamic quantization, a form of non-linear optimization that is precise for both large and small magnitude values, and (2) a stable embedding layer to reduce gradient variance that comes from the highly non-uniform distribution of input tokens in language models. As a result, our 8-bit optimizers maintain 32-bit performance with a small fraction of the memory footprint on a range of tasks, including 1.5B parameter language modeling, GLUE finetuning, ImageNet classification, WMT'14 machine translation, MoCo v2 contrastive ImageNet pretraining+finetuning, and RoBERTa pretraining, without changes to the original optimizer hyperparameters. We open-source our 8-bit optimizers as a drop-in replacement that only requires a two-line code change.

Increasing model size is an effective way to achieve better performance for given resources (Kaplan et al., 2020; Henighan et al., 2020; Raffel et al., 2019; Lewis et al., 2021). However, training such large models requires storing the model, gradient, and state of the optimizer (e.g., exponentially smoothed sum and squared sum of previous gradients for Adam), all in a fixed amount of available memory. Although significant research has focused on enabling larger model training by reducing or efficiently distributing the memory required for the model parameters (Shoeybi et al., 2019; Lepikhin et al., 2020; Fedus et al., 2021; Brown et al., 2020; Rajbhandari et al., 2020), reducing the memory footprint of optimizer gradient statistics is much less studied. This is a significant missed opportunity since these optimizer states use 33-75% of the total memory footprint during training. For example, the Adam optimizer states for the largest GPT-2 (Radford et al., 2019) and T5 (Raffel et al., 2019) models are 11 GB and 41 GB in size. In this paper, we develop a fast, high-precision non-linear quantization method – block-wise dynamic quantization – that enables stable 8-bit optimizers (e.g., Adam, AdamW, and Momentum) which maintain 32-bit performance at a fraction of the memory footprint and without any changes to the original hyperparameters.[1]

While most current work uses 32-bit optimizer states, recent high-profile efforts to use 16-bit optimizers report difficultly for large models with more than 1B parameters (Ramesh et al., 2021). Going from 16-bit optimizers to 8-bit optimizers reduces the range of possible values from $2^{16} = 65536$ values to just $2^8 = 256$. To our knowledge, this has not been attempted before.

Effectively using this very limited range is challenging for three reasons: quantization accuracy, computational efficiency, and large-scale stability. To maintain accuracy, it is critical to introduce some form of non-linear quantization to reduce errors for both common small magnitude values

[1]We study 8-bit optimization with current best practice model and gradient representations (typically 16-bit mixed precision), to isolate optimization challenges. Future work could explore further compressing all three.

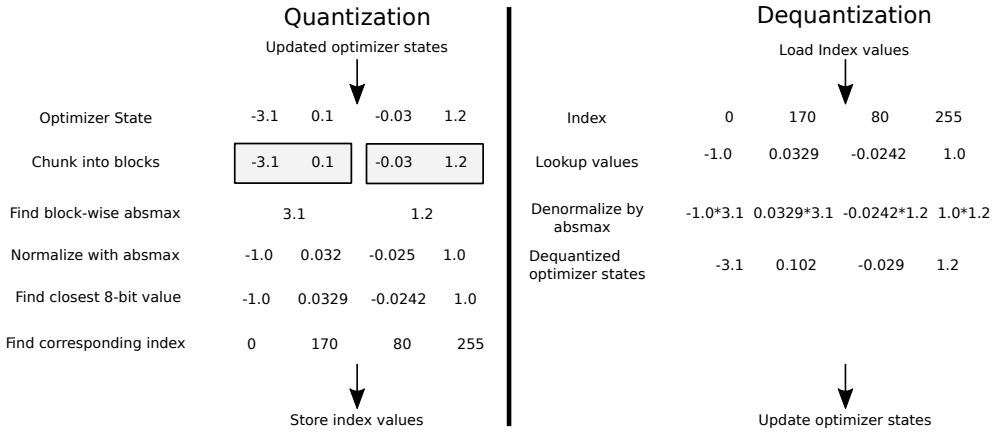

Figure 1: Schematic of 8-bit optimizers via block-wise dynamic quantization, see Section 2 for more details. After the optimizer update is performed in 32-bit, the state tensor is chunked into blocks, normalized by the absolute maximum value of each block. Then dynamic quantization is performed, and the index is stored. For dequantization, the index is loaded, a lookup is performed, with subsequent denormalization by multiplication with the block-wise absolute maximum value. Outliers are confined to a single block through block-wise quantization, and their effect on normalization is limited.

and rare large ones. However, to be practical, 8-bit optimizers need to be fast enough to not slow down training, which is especially difficult for non-linear methods that require more complex data structures to maintain the quantization buckets. Finally, to maintain stability with huge models beyond 1B parameters, a quantization method needs to not only have a good mean error but excellent worse case performance since a single large quantization error can cause the entire training run to diverge.

We introduce a new block-wise quantization approach that addresses all three of these challenges. Block-wise quantization splits input tensors into blocks and performs quantization on each block independently. This block-wise division reduces the effect of outliers on the quantization process since they are isolated to particular blocks, thereby improving stability and performance, especially for large-scale models. Block-wise processing also allows for high optimizer throughput since each normalization can be computed independently in each core. This contrasts with tensor-wide normalization, which requires slow cross-core synchronization that is highly dependent on task-core scheduling. We combine block-wise quantization with two novel methods for stable, high-performance 8-bit optimizers: dynamic quantization and a stable embedding layer. Dynamic quantization is an extension of dynamic tree quantization for unsigned input data. The stable embedding layer is a variation of a standard word embedding layer that supports more aggressive quantization by normalizing the highly non-uniform distribution of inputs to avoid extreme gradient variation.

Our 8-bit optimizers maintain 32-bit performance at a fraction of the original memory footprint. We show this for a broad range of tasks: 1.5B and 355M parameter language modeling, GLUE finetuning, ImageNet classification, WMT'14+WMT'16 machine translation, MoCo v2 contrastive image pretraining+finetuning, and RoBERTa pretraining. We also report additional ablations and sensitivity analysis showing that all components – block-wise quantization, dynamic quantization, and stable embedding layer – are crucial for these results and that 8-bit Adam can be used as a simple drop-in replacement for 32-bit Adam, with no hyperparameter changes. We open-source our custom CUDA kernels and provide a PyTorch implementation that enables 8-bit optimization by changing two lines of code.

1 Background

1.1 Stateful Optimizers

An optimizer updates the parameters $\mathbf{w}$ of a neural network by using the gradient of the loss with respect to the weight $\mathbf{g}_t = \frac{\partial \mathbf{L}}{\partial \mathbf{w}}$ at update iteration t. Stateful optimizers compute statistics of the

gradient with respect to each parameter over time for accelerated optimization. Two of the most commonly used stateful optimizers are Adam (Kingma and Ba, 2014), and SGD with momentum (Qian, 1999) – or Momentum for short. Without damping and scaling constants, the update rules of these optimizers are given by:

$$\text{Momentum}(\mathbf{g}_t, \mathbf{w}_{t-1}, \mathbf{m}_{t-1}) = \begin{cases} \mathbf{m}_0 = \mathbf{g}_0 & \text{Initialization} \\ \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + \mathbf{g}_t & \text{State 1 update} \\ \mathbf{w}_t = \mathbf{w}_{t-1} - \alpha \cdot \mathbf{m}_t & \text{Weight update} \end{cases} \tag{1}$$

$$\text{Adam}(\mathbf{g}_t, \mathbf{w}_{t-1}, \mathbf{m}_{t-1}, \mathbf{r}_{t-1}) = \begin{cases} \mathbf{r}_0 = \mathbf{m}_0 = \mathbf{0} & \text{Initialization} \\ \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\mathbf{g}_t & \text{State 1 update} \\ \mathbf{r}_t = \beta_2 \mathbf{r}_{t-1} + (1-\beta_2)\mathbf{g}_t^2 & \text{State 2 update} \\ \mathbf{w}_t = \mathbf{w}_{t-1} - \alpha \cdot \frac{\mathbf{m}_t}{\sqrt{\mathbf{r}_t}+\epsilon} & \text{Weight update,} \end{cases} \tag{2}$$

where β_1 and β_2 are smoothing constants, ϵ is a small constant, and α is the learning rate.

For 32-bit states, Momentum and Adam consume 4 and 8 bytes per parameter. That is 4 GB and 8 GB for a 1B parameter model. Our 8-bit non-linear quantization reduces these costs to 1 GB and 2 GB.

1.2 Non-linear Quantization

Quantization compresses numeric representations to save space at the cost of precision. Quantization is the mapping of a k-bit integer to a real element in D, that is, $\mathbf{Q}^{\text{map}} : [0, 2^k - 1] \mapsto D$. For example, the IEEE 32-bit floating point data type maps the indices $0...2^{32} - 1$ to the domain [-3.4e38, +3.4e38]. We use the following notation: $\mathbf{Q}^{\text{map}}(i) = \mathbf{Q}^{\text{map}}_i = q_i$, for example $\mathbf{Q}^{\text{map}}(2^{31} + 131072) = 2.03125$, for the IEEE 32-bit floating point data type.

To perform general quantization from one data type into another we require three steps. (1) Compute a normalization constant N that transforms the input tensor $\mathbf{T}$ into the range of the domain D of the target quantization data type $\mathbf{Q}^{\text{map}}$, (2) for each element of $\mathbf{T}/N$ find the closest corresponding value q_i in the domain D, (3) store the index i corresponding to q_i in the quantized output tensor $\mathbf{T}^Q$. To receive the dequantized tensor $\mathbf{T}^D$ we look up the index and denormalize: $\mathbf{T}^D_i = \mathbf{Q}^{\text{map}}(\mathbf{T}^Q_i) \cdot N$.

To perform this procedure for dynamic quantization we first normalize into the range [-1, 1] through division by the absolute maximum value: $N = \max(|\mathbf{T}|)$.

Then we find the closest values via a binary search:

$$\mathbf{T}^Q_i = \arg\min_{j=0}^{2^n} |\mathbf{Q}^{\text{map}}_j - \frac{\mathbf{T}_i}{N}| \tag{3}$$

1.3 Dynamic Tree Quantization

Dynamic Tree quantization (Dettmers, 2016) is a method that yields low quantization error for both small and large magnitude values. Unlike data types with fixed exponent and fraction, dynamic tree quantization uses a datatype with a dynamic exponent and fraction that can change with each number. It is made up of four parts, as seen in Figure 2: (1) The first bit of the data type is reserved for a sign. (2) The number of subsequent zero bits indicates the magnitude of the exponent. (3) The first bit that is set to one indicates that all following values are reserved for (4) linear quantization. By moving the indicator bit, numbers can have a large exponent 10^{-7} or precision as high as $1/63$. Compared to linear quantization, dynamic tree quantization has better absolute and relative quantization errors for non-uniform distributions. Dynamic tree quantization is strictly defined to quantize numbers in the range [-1.0, 1.0], which is ensured by performing tensor-level absolute max normalization.

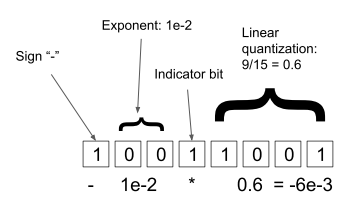

Figure 2: Dynamic tree quantization.

2 8-BIT OPTIMIZERS

Our 8-bit optimizers have three components: (1) block-wise quantization that isolates outliers and distributes the error more equally over all bits; (2) dynamic quantization, which quantizes both small and large values with high precision; and (3) a stable embedding layer to improve stability during optimization for models with word embeddings.

With these components, performing an optimizer update with 8-bit states is straightforward. We dequantize the 8-bit optimizer states to 32-bit, perform the update, and then quantize the states back to 8-bit for storage. We do this 8-bit to 32-bit conversion element-by-element in registers, which means no slow copies to GPU memory or additional temporary memory are needed to perform quantization and dequantization. For GPUs, this makes 8-bit optimizers faster than regular 32-bit optimizers, as we show in Section 3.

2.1 BLOCK-WISE QUANTIZATION

Our block-wise quantization reduces the cost of computing normalization and improves quantization precision by isolating outliers. In order to dynamically quantize a tensor, as defined in Section 1.2, we need to normalize the tensor into the range [-1, 1]. Such normalization requires a reduction over the entire tensor, which entails multiple synchronizations across GPU cores. Block-wise dynamic quantization reduces this cost by chunking an input tensor into small blocks of size $B = 2048$ and performing normalization independently in each core across this block.

More formally, using the notation introduced in Section 1.2, in block-wise quantization, we treat $\mathbf{T}$ as a one-dimensional sequence of elements that we chunk in blocks of size B. This means for an input tensor $\mathbf{T}$ with n elements we have n/B blocks. We proceed to compute a normalization constant for each block: $N_b = \max(|\mathbf{T}_b|)$, where b is the index of the block $0..n/B$. With this block-wise normalization constant, each block can be quantized independently:

$$\mathbf{T}^Q_{bi} = \arg\min_{j=0}^{2^n} |\mathbf{Q}^{\text{map}}_j - \frac{\mathbf{T}_{bi}}{N_b}| \Big|_{0<i<B} \tag{4}$$

This approach has several advantages, both for stability and efficiency. First, each block normalization can be computed independently. Thus no synchronization between cores is required, and throughput is enhanced.

Secondly, it is also much more robust to outliers in the input tensor. For example, to contrast block-wise and regular quantization, if we create an input tensor with one million elements sampled from the standard normal distribution, we expect less than 1% of elements of the tensor will be in the range [3, $+\infty$). However, since we normalize the input tensor into the range [-1,1] this means the maximum values of the distribution determine the range of quantization buckets. This means if the input tensor contains an outlier with magnitude 5, the quantization buckets reserved for numbers between 3 and 5 will mostly go unused since less than 1% of numbers are in this range. With block-wise quantization, the effect of outliers is limited to a single block. As such, most bits are used effectively in other blocks.

Furthermore, because outliers represent the absolute maximum value in the input tensor, block-wise quantization approximates outlier values without any error. This guarantees that the largest optimizer states, arguably the most important, will always be quantized with full precision. This property makes block-wise dynamic quantization both robust and precise and is essential for good training performance in practice.

2.2 DYNAMIC QUANTIZATION

In this work, we extend dynamic tree quantization (Section 1.3) for non-signed input tensors by re-purposing the sign bit. Since the second Adam state is strictly positive, the sign bit is not needed. Instead of just removing the sign bit, we opt to extend dynamic tree quantization with a *fixed* bit for the fraction. This extension is motivated by the observation that the second Adam state varies around 3-5 orders of magnitude during the training of a language model. In comparison, dynamic tree quantization already has a range of 7 orders of magnitude. We refer to this quantization as

dynamic quantization to distinguish it from dynamic tree quantization in our experiments. A study of additional quantization data types and their performance is detailed in Appendix E.

2.3 Stable Embedding Layer

Our stable embedding layer is a standard word embedding layer variation designed to ensure stable training for NLP tasks. This embedding layer supports more aggressive quantization by normalizing the highly non-uniform distribution of inputs to avoid extreme gradient variation. See Appendix B for a discussion of why commonly adopted embedding layers (Ott et al., 2019) are so unstable.

We initialize the Stable Embedding layer with Xavier uniform initialization (Glorot and Bengio, 2010) and apply layer normalization (Ba et al., 2016) before adding position embeddings. This method maintains a variance of roughly one both at initialization and during training. Additionally, the uniform distribution initialization has less extreme values than a normal distribution, reducing maximum gradient size. Like Ramesh et al. (2021), we find that the stability of training improves significantly if we use 32-bit optimizer states for the embedding layers. This is the only layer that uses 32-bit optimizer states. We still use the standard precision for weights and gradients for the embedding layers – usually 16-bit. We show in our Ablation Analysis in Section 4 that this change is a necessary detail.

3 8-bit vs 32-bit Optimizer Performance for Common Benchmarks

Experimental Setup We compare the performance of 8-bit optimizers to their 32-bit counterparts on a range of challenging public benchmarks. These benchmarks either use Adam (Kingma and Ba, 2014), AdamW (Loshchilov and Hutter, 2018), or Momentum (Qian, 1999).

We do not change any hyperparameters or precision of weights, gradients, and activations/input gradients for each experimental setting compared to the public baseline– the only change is to replace 32-bit optimizers with 8-bit optimizers. This means that for most experiments, we train in 16-bit mixed-precision (Micikevicius et al., 2017). We also compare with Adafactor (Shazeer and Stern, 2018), with the time-independent formulation for β_2 (Shazeer and Stern, 2018) – which is the same formulation used in Adam. We also do not change any hyperparameters for Adafactor.

We report on benchmarks in neural machine translation (Ott et al., 2018)[2] trained on WMT'16 (Sennrich et al., 2016) and evaluated on en-de WMT'14 (Macháček and Bojar, 2014), large-scale language modeling (Lewis et al., 2021; Brown et al., 2020) and RoBERTa pretraining (Liu et al., 2019) on English CC-100 + RoBERTa corpus (Nagel, 2016; Gokaslan and Cohen, 2019; Zhu et al., 2015; Wenzek et al., 2020), finetuning the pretrained masked language model RoBERTa (Liu et al., 2019)[3] on GLUE (Wang et al., 2018a), ResNet-50 v1.5 image classification (He et al., 2016)[4] on ImageNet-1k (Deng et al., 2009), and Moco v2 contrastive image pretraining and linear finetuning (Chen et al., 2020)[5] on ImageNet-1k (Deng et al., 2009).

We use the stable embedding layer for all NLP tasks except for finetuning on GLUE. Beyond this, we follow the exact experimental setup outlined in the referenced papers and codebases. We consistently report replication results for each benchmark with public codebases and report median accuracy, perplexity, or BLEU over ten random seeds for GLUE, three random seeds for others tasks, and a single random seed for large scale language modeling. While it is standard to report means and standard errors on some tasks, others use median performance. We opted to report medians for all tasks for consistency.

Results In Table 1, we see that 8-bit optimizers match replicated 32-bit performance for all tasks. While Adafactor is competitive with 8-bit Adam, 8-bit Adam uses less memory and provides faster optimization. Our 8-bit optimizers save up to 8.5 GB of GPU memory for our largest 1.5B parameter language model and 2.0 GB for RoBERTa. Thus, 8-bit optimizers maintain performance and improve accessibility to the finetuning of large models for those that cannot afford GPUs with

[2] https://github.com/pytorch/fairseq/tiny/master/examples/scaling_nmt/README.md

[3] https://github.com/pytorch/fairseq/blob/master/examples/roberta/README.glue.md

[4] https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/Classification/ConvNets/

[5] https://github.com/facebookresearch/moco

Table 1: Median performance on diverse NLP and computer vision tasks: GLUE, object classification with (Moco v2) and without pretraining (CLS), machine translation (MT), and large-scale language modeling (LM). While 32-bit Adafactor is competitive with 8-bit Adam, it uses almost twice as much memory and trains slower. 8-bit Optimizers match or exceed replicated 32-bit performance on all tasks. We observe no instabilities for 8-bit optimizers. Time is total GPU time on V100 GPUs, except for RoBERTa and GPT3 pretraining, which were done on A100 GPUs.

Optimizer	Task	Data	Model	Metric†	Time	Mem saved
32-bit AdamW	GLUE	Multiple	RoBERTa-Large	88.9	–	Reference
32-bit AdamW	GLUE	Multiple	RoBERTa-Large	88.6	17h	0.0 GB
32-bit Adafactor	GLUE	Multiple	RoBERTa-Large	**88.7**	24h	1.3 GB
8-bit AdamW	GLUE	Multiple	RoBERTa-Large	**88.7**	**15h**	**2.0 GB**
32-bit Momentum	CLS	ImageNet-1k	ResNet-50	77.1	–	Reference
32-bit Momentum	CLS	ImageNet-1k	ResNet-50	77.1	118h	0.0 GB
8-bit Momentum	CLS	ImageNet-1k	ResNet-50	**77.2**	**116 h**	**0.1 GB**
32-bit Adam	MT	WMT'14+16	Transformer	29.3	–	Reference
32-bit Adam	MT	WMT'14+16	Transformer	29.0	126h	0.0 GB
32-bit Adafactor	MT	WMT'14+16	Transformer	29.0	127h	0.3 GB
8-bit Adam	MT	WMT'14+16	Transformer	**29.1**	**115h**	**1.1 GB**
32-bit Momentum	MoCo v2	ImageNet-1k	ResNet-50	67.5	–	Reference
32-bit Momentum	MoCo v2	ImageNet-1k	ResNet-50	67.3	30 days	0.0 GB
8-bit Momentum	MoCo v2	ImageNet-1k	ResNet-50	**67.4**	**28 days**	**0.1 GB**
32-bit Adam	LM	Multiple	Transformer-1.5B	9.0	308 days	0.0 GB
32-bit Adafactor	LM	Multiple	Transformer-1.5B	**8.9**	316 days	5.6 GB
8-bit Adam	LM	Multiple	Transformer-1.5B	9.0	**297 days**	**8.5 GB**
32-bit Adam	LM	Multiple	GPT3-Medium	10.62	795 days	0.0 GB
32-bit Adafactor	LM	Multiple	GPT3-Medium	10.68	816 days	1.5 GB
8-bit Adam	LM	Multiple	GPT3-Medium	**10.62**	**761 days**	**1.7 GB**
32-bit Adam	Masked-LM	Multiple	RoBERTa-Base	3.49	101 days	0.0 GB
32-bit Adafactor	Masked-LM	Multiple	RoBERTa-Base	3.59	112 days	0.7 GB
8-bit Adam	Masked-LM	Multiple	RoBERTa-Base	**3.48**	**94 days**	**1.1 GB**

†**Metric**: GLUE=Mean Accuracy/Correlation. CLS/MoCo = Accuracy. MT=BLEU. LM=Perplexity.

large memory buffers. We show models that are now accessible with smaller GPUs in Table 2. A breakdown of individual dataset results on GLUE can be found in Appendix A).

The broad range of tasks and competitive results demonstrate that 8-bit optimizers are a robust and effective replacement for 32-bit optimizers, do not require any additional changes in hyperparameters, and save a significant amount of memory while speeding up training slightly.

Table 2: With 8-bit optimizers, larger models can be finetuned with the same GPU memory compared to standard 32-bit optimizer training. We use a batch size of one for this comparison.

	Largest finetunable Model (parameters)	
GPU size in GB	32-bit Adam	8-bit Adam
6	RoBERTa-base (110M)	RoBERTa-large (355M)
11	MT5-small (300M)	MT5-base (580M)
24	MT5-base (580M)	MT5-large (1.2B)
24	GPT-2-medium (762M)	GPT-2-large (1.5B)

4 Analysis

We analyze our method in two ways. First, we ablate all 8-bit optimizer components and show that they are necessary for good performance. Second, we look at the sensitivity to hyperparameters compared to 32-bit Adam and show that 8-bit Adam with block-wise dynamic quantization is a reliable replacement that does not require further hyperparameter tuning.

Experimental Setup We perform our analysis on a strong 32-bit Adam baseline for language modeling with transformers (Vaswani et al., 2017). We subsample from the RoBERTa corpus (Liu et al., 2019) which consists of the English sub-datasets: Books (Zhu et al., 2015), Stories (Trinh and Le, 2018), OpenWebText-1 (**?**), Wikipedia, and CC-News (Nagel, 2016). We use a 50k token BPE encoded vocabulary (Sennrich et al., 2015). We find the best 2-GPU-day transformer baseline for 32-bit Adam with multiple hyperparameter searches that take in a total of 440 GPU days. Key hyperparameters include 10 layers with a model dimension of 1024, a fully connected hidden dimension of 8192, 16 heads, and input sub-sequences with a length of 512 tokens each. The final model has 209m parameters.

Table 3: Ablation analysis of 8-bit Adam for small (2 GPU days) and large-scale (≈1 GPU year) transformer language models on the RoBERTa corpus. The runs without dynamic quantization use linear quantization. The percentage of unstable runs indicates either divergence or crashed training due to exploding gradients. We report median perplexity for successful runs. We can see that dynamic quantization is critical for general stability and block-wise quantization is critical for large-scale stability. The stable embedding layer is useful for both 8-bit and 32-bit Adam and enhances stability to some degree.

Parameters	Optimizer	Dynamic	Block-wise	Stable Emb	Unstable (%)	Perplexity
209M	32-bit Adam				0	16.7
	32-bit Adam			✓	0	**16.3**
	8-bit Adam				90	253.0
	8-bit Adam			✓	50	**194.4**
	8-bit Adam	✓			10	18.6
	8-bit Adam	✓		✓	0	**17.7**
	8-bit Adam	✓	✓		0	16.8
	8-bit Adam	✓	✓	✓	0	**16.4**
1.3B	32-bit Adam				0	10.4
1.3B	8-bit Adam	✓			100	N/A
1.3B	8-bit Adam	✓		✓	80	10.9
1.5B	32-bit Adam				0	9.0
1.5B	8-bit Adam	✓	✓	✓	0	9.0

Ablation Analysis For the ablation analysis, we compare small and large-scale language modeling perplexity and training stability against a 32-bit Adam baseline. We ablate components individually and include combinations of methods that highlight their interactions. The baseline method uses linear quantization, and we add dynamic quantization, block-wise quantization, and the stable embedding layer to demonstrate their effect. To test optimization stability for small-scale language modeling, we run each setting with different hyperparameters and report median performance across all successful runs. A successful run is a run that does not crash due to exploding gradients or diverges in the loss. We use the hyperparameters ϵ {1e-8, 1e-7, 1e-6}, β_1 {0.90, 0.87, 0.93}, β_2 {0.999, 0.99, 0.98} and small changes in learning rates. We also include some partial ablations for large-scale models beyond 1B parameters. In the large-scale setting, we run several seeds with the same hyperparameters. We use a single seed for 32-bit Adam, five seeds for 8-bit Adam at 1.3B parameters, and a single seed for 8-bit Adam at 1.5B parameters.[6] Results are shown in Table 3.

The Ablations show that dynamic quantization, block-wise quantization, and the stable embedding layer are critical for either performance or stability. In addition, block-wise quantization is critical for large-scale language model stability.

Sensitivity Analysis We compare the perplexity of 32-bit Adam vs 8-bit Adam + Stable Embedding as we change the optimizer hyperparameters: learning rate, betas, and ϵ. We change each hyperparameter individually from the baseline hyperparameters β_1=0.9, β_2=0.995, ϵ=1e-7, and lr=0.0163 and run two random seeds for both 8-bit and 32-bit Adam for each setting. If 8-bit Adam is perfectly insensitive to hyperparameters compared to 32-bit Adam, we would expect the same constant offset

[6]We chose not to do the full ablations with such large models because each training run takes one GPU year.

in performance for any hyperparameter combination. The results can be seen in Figure 3. The results show a relatively steady gap between 8-bit and 32-bit Adam, suggesting that 8-bit Adam does not require any further hyperparameter tuning compared to 32-bit Adam.

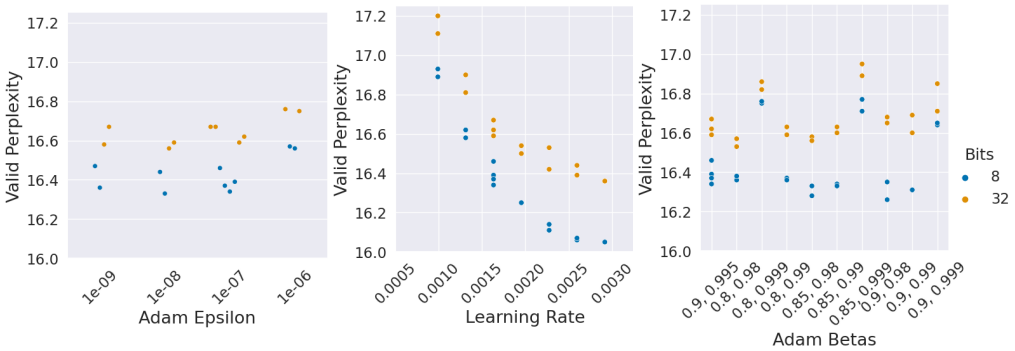

Figure 3: Sensitivity analysis of 8-bit vs 32-bit Adam hyperparameters. We can see that there is little variance between 8 and 32-bit performance, which suggests that 8-bit Adam can be used as a drop-in replacement for 32-bit Adam without any further hyperparameter tuning.

5 Related Work

Compressing & Distributing Optimizer States While 16-bit Adam has been used in several publications, the stability of 16-bit Adam was first explicitly studied for a text-to-image generation model DALL-E (Ramesh et al., 2021). They show that a stable embedding layer, tensor-wise scaling constants for both Adam states, and multiple loss scaling blocks are critical to achieving stability during training. Our work reduces the memory footprint of Adam further, from 16 to 8-bit. In addition, we achieve stability by developing new training procedures and non-linear quantization, both of which complement previous developments.

Adafactor (Shazeer and Stern, 2018) uses a different strategy to save memory. All optimizer states are still 32-bit, but the second Adam state is factorized by a row-column outer product resulting in a comparable memory footprint to 16-bit Adam. We compare 8-bit Adam with Adafactor in our experiments. Our methods could also be used to create 8-bit Adafactor in future work.

Optimizer sharding (Rajbhandari et al., 2020) splits optimizer states across multiple accelerators such as GPUs/TPUs. While very effective, it can only be used if multiple accelerators are available and data parallelism is used. Optimizer sharding can also have significant communication overhead (Rajbhandari et al., 2021). Our 8-bit optimizers work with all kinds of parallelism. They can also complement optimizer sharding, as they reduce communication overhead by 75%.

General Memory Reduction Techniques Other complementary methods for efficient training can be either distributed or local. Distributed approaches spread out the memory of a model across several accelerators such as GPUs/TPUs. Such approaches are model parallelism (Krizhevsky et al., 2009), pipeline parallelism (Krizhevsky et al., 2009; Huang et al., 2018; Harlap et al., 2018), and operator parallelism (Lepikhin et al., 2020). These approaches are useful if one has multiple accelerators available. Our 8-bit optimizers are useful for both single and multiple devices.

Local approaches work for a single accelerator. They include gradient checkpointing (Chen et al., 2016), reversible residual connections (Gomez et al., 2017), and offloading (Pudipeddi et al., 2020; Rajbhandari et al., 2021). All these methods save memory at the cost of increased computational or communication costs. Our 8-bit optimizers reduce the memory footprint of the model while maintaining 32-bit training speed.

Quantization Methods and Data Types While our work is the first to apply 8-bit quantization to optimizer statistics, quantization for neural network model compression, training, and inference are well-studied problems. One of the most common formats of 8-bit quantization is to use data types composed of static sign, exponent, and fraction bits. The most common combination is 5 bits for

the exponent and 2 bits for the fraction (Wang et al., 2018b; Sun et al., 2019; Cambier et al., 2020; Mellempudi et al., 2019) with either no normalization or min-max normalization. These data types offer high precision for small magnitude values but have large errors for large magnitude values since only 2 bits are assigned to the fraction.

Data types lower than 8-bit are usually used to prepare a model for deployment, and the main focus is on improving network inference speed and memory footprint rather than maintaining accuracy. There are methods that use 1-bit (Courbariaux and Bengio, 2016; Rastegari et al., 2016; Courbariaux et al., 2015), 2-bit/3 values (Zhu et al., 2017; Choi et al., 2019), 4-bits (Li et al., 2019), more bits (Courbariaux et al., 2014), or a variable amount of bits (Gong et al., 2019). See also Qin et al. (2020) for a survey on binary neural networks. While these low-bit quantization techniques allow for efficient storage, they likely lead to instability when used for optimizer states.

The work most similar to our block-wise quantization is work on Hybrid Block Floating Point (HBFP) (Drumond et al., 2018) which uses a 24-bit fraction data type with a separate exponent for each tile in matrix multiplication to perform 24-bit matrix multiplication. However, unlike HBFP, block-wise dynamic quantization has the advantage of having both block-wise normalization *and* a dynamic exponent for each number. This allows for a much broader range of important values since optimizer state values vary by about 5 orders of magnitude. Furthermore, unlike HBFP, block-wise quantization approximates the maximum magnitude values within each block without any quantization error, which is critical for optimization stability, particularly for large networks.

6 Discussion & Limitations

Here we have shown that high precision quantization can yield 8-bit optimizers that maintain 32-bit optimizer performance without requiring any change in hyperparameters. One of the main limitations of our work is that 8-bit optimizers for natural language tasks require a stable embedding layer to be trained to 32-bit performance. On the other hand, we show that 32-bit optimizers also benefit from a stable embedding layer. As such, the stable embedding layer could be seen as a general replacement for other embedding layers.

We show that 8-bit optimizers reduce the memory footprint and accelerate optimization on a wide range of tasks. However, since 8-bit optimizers reduce only the memory footprint proportional to the number of parameters, models that use large amounts of activation memory and little memory for parameters, such as convolutional networks, have few benefits from using 8-bit optimizers. Thus, 8-bit optimizers are most beneficial for training or finetuning models with many parameters on highly memory-constrained GPUs.

Furthermore, there remain sources of instability that, to our knowledge, are not well understood. For example, we observed that models with over 1B parameters often have hard systemic divergence, where many parameters simultaneously cause exploding gradients. In other cases, a single parameter among those 1B parameters assumed a value too large, caused an exploding gradient, and led to a cascade of instability. It might be that this rare, soft cascading instability is related to the phenomena where instability disappears after reloading a model checkpoint and rolling a new random seed – a method standard for training huge models. This cascading instability might also be related to the observation that the larger a model is, the more unstable it becomes. For our 8-bit optimizers, we primarily needed the stable embedding layer to avoid cascading instability. Thus the stable embedding layer could potentially be viewed as decreasing the probability of extreme outlier gradients. If such phenomena were better understood, it could lead to better 8-bit optimizers and more stable training in general.

7 Broader Impact

Our 8-bit optimizers enable training models that previously could not be trained on various GPUs, as shown in Table 2. Furthermore, while many options exist to reduce the memory footprint via parallelism (Rajbhandari et al., 2020; Lepikhin et al., 2020) our 8-bit optimizers are one of the few options that can reduce the optimizer memory footprint significantly for single devices without degrading performance. Therefore, it is likely that our 8-bit optimizers will improve access to larger models – especially for the users that have the least resources.

REFERENCES

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Cambier, L., Bhiwandiwalla, A., Gong, T., Elibol, O. H., Nekuii, M., and Tang, H. (2020). Shifted and squeezed 8-bit floating point format for low-precision training of deep neural networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Chen, X., Fan, H., Girshick, R., and He, K. (2020). Improved baselines with momentum contrastive learning. *arXiv preprint arXiv:2003.04297*.

Choi, J., Venkataramani, S., Srinivasan, V., Gopalakrishnan, K., Wang, Z., and Chuang, P. (2019). Accurate and efficient 2-bit quantized neural networks. In Talwalkar, A., Smith, V., and Zaharia, M., editors, *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org.

Courbariaux, M. and Bengio, Y. (2016). Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830.

Courbariaux, M., Bengio, Y., and David, J. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3123–3131.

Courbariaux, M., Bengio, Y., and David, J.-P. (2014). Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

Dettmers, T. (2016). 8-bit approximations for parallelism in deep learning. *International Conference on Learning Representations (ICLR)*.

Drumond, M., Lin, T., Jaggi, M., and Falsafi, B. (2018). Training dnns with hybrid block floating point. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 451–461.

Fedus, W., Zoph, B., and Shazeer, N. (2021). Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings.

Gokaslan, A. and Cohen, V. (2019). Openwebtext corpus.

Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. (2017). The reversible residual network: Backpropagation without storing activations. *arXiv preprint arXiv:1707.04585*.

Gong, R., Liu, X., Jiang, S., Li, T., Hu, P., Lin, J., Yu, F., and Yan, J. (2019). Differentiable soft quantization: Bridging full-precision and low-bit neural networks. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 4851–4860. IEEE.

Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., and Gibbons, P. (2018). Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

Henighan, T., Kaplan, J., Katz, M., Chen, M., Hesse, C., Jackson, J., Jun, H., Brown, T. B., Dhariwal, P., Gray, S., et al. (2020). Scaling laws for autoregressive generative modeling. *arXiv preprint arXiv:2010.14701*.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2018). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. (2020). Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*.

Lewis, M., Bhosale, S., Dettmers, T., Goyal, N., and Zettlemoyer, L. (2021). Base layers: Simplifying training of large, sparse models. *arXiv preprint arXiv:2103.16716*.

Li, R., Wang, Y., Liang, F., Qin, H., Yan, J., and Fan, R. (2019). Fully quantized network for object detection. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 2810–2819. Computer Vision Foundation / IEEE.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Loshchilov, I. and Hutter, F. (2018). Fixing weight decay regularization in adam.

Macháček, M. and Bojar, O. (2014). Results of the wmt14 metrics shared task. In *Proceedings of the Ninth Workshop on Statistical Machine Translation*, pages 293–301.

Mellempudi, N., Srinivasan, S., Das, D., and Kaul, B. (2019). Mixed precision training with 8-bit floating point. *CoRR*, abs/1905.12334.

Micikevicius, P., Narang, S., Alben, J., Diamos, G., Elsen, E., Garcia, D., Ginsburg, B., Houston, M., Kuchaiev, O., Venkatesh, G., et al. (2017). Mixed precision training. *arXiv preprint arXiv:1710.03740*.

Nagel, S. (2016). Cc-news.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. (2019). fairseq: A fast, extensible toolkit for sequence modeling. *arXiv preprint arXiv:1904.01038*.

Ott, M., Edunov, S., Grangier, D., and Auli, M. (2018). Scaling neural machine translation. *arXiv preprint arXiv:1806.00187*.

Pudipeddi, B., Mesmakhosroshahi, M., Xi, J., and Bharadwaj, S. (2020). Training large neural networks with constant memory using a new execution algorithm. *arXiv preprint arXiv:2002.05645*.

Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks : the official journal of the International Neural Network Society*, 12 1:145–151.

Qin, H., Gong, R., Liu, X., Bai, X., Song, J., and Sebe, N. (2020). Binary neural networks: A survey. *CoRR*, abs/2004.03333.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. (2019). Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683*.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. (2020). Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16. IEEE.

Rajbhandari, S., Ruwase, O., Rasley, J., Smith, S., and He, Y. (2021). Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *arXiv preprint arXiv:2104.07857*.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. (2021). Zero-shot text-to-image generation. *arXiv preprint arXiv:2102.12092*.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In Leibe, B., Matas, J., Sebe, N., and Welling, M., editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 525–542. Springer.

Sennrich, R., Haddow, B., and Birch, A. (2015). Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Sennrich, R., Haddow, B., and Birch, A. (2016). Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*.

Shazeer, N. and Stern, M. (2018). Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*.

Sun, X., Choi, J., Chen, C., Wang, N., Venkataramani, S., Srinivasan, V., Cui, X., Zhang, W., and Gopalakrishnan, K. (2019). Hybrid 8-bit floating point (HFP8) training and inference for deep neural networks. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 4901–4910.

Trinh, T. H. and Le, Q. V. (2018). A simple method for commonsense reasoning. *arXiv preprint arXiv:1806.02847*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., and Bowman, S. R. (2018a). Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Wang, N., Choi, J., Brand, D., Chen, C., and Gopalakrishnan, K. (2018b). Training deep neural networks with 8-bit floating point numbers. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7686–7695.

Wenzek, G., Lachaux, M.-A., Conneau, A., Chaudhary, V., Guzmán, F., Joulin, A., and Grave, E. (2020). CCNet: Extracting high quality monolingual datasets from web crawl data. In *Proceedings of the 12th Language Resources and Evaluation Conference*, pages 4003–4012, Marseille, France. European Language Resources Association.

Zhu, C., Han, S., Mao, H., and Dally, W. J. (2017). Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Zhu, Y., Kiros, R., Zemel, R., Salakhutdinov, R., Urtasun, R., Torralba, A., and Fidler, S. (2015). Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27.