CompAct: Compressed Activations for Memory-Efficient LLM Training

Yara Shamshoum[*], **Nitzan Hodos**[*], **Yuval Sieradzki, Assaf Schuster,**
Department of Computer Science, Technion - Israel Institute of Technology
{yara-sh, hodosnitzan, syuvsier}@campus.technion.ac.il

Abstract

We introduce CompAct, a technique that reduces peak memory utilization on GPU by 25-30% for pretraining and 50% for fine-tuning of LLMs. Peak device memory is a major limiting factor in training LLMs, with various recent works aiming to reduce model memory. However most works don't target the largest component of allocated memory during training: the model's compute graph, which is stored for the backward pass. By storing low-rank, compressed activations to be used in the backward pass we greatly reduce the required memory, unlike previous methods which only reduce optimizer overheads or the number of trained parameters. Our compression uses random projection matrices, thus avoiding additional memory overheads. Comparisons with previous techniques for either pretraining or fine-tuning show that CompAct substantially improves existing compute-performance trade-offs. We expect CompAct's savings to scale even higher for larger models.

1 Introduction

Training Large Language Models (LLMs) and fine-tuning them on downstream tasks has led to impressive results across various natural language applications (Raffel et al., 2023a; Brown et al., 2020). However, as LLMs scale from millions to hundreds of billions of parameters, the computational resources required for both pre-training and fine-tuning become prohibitive.

While compute power is the primary bottleneck for those who train very large LLMs, memory requirements become the main limitation for researchers without access to vast hardware resources. This disparity severely limits the ability to advance the field of LLM training to only a select few.

Recent works tackle memory reductions by applying a low rank approximation to model parameters (Hu et al., 2021; Lialin et al., 2023), or to

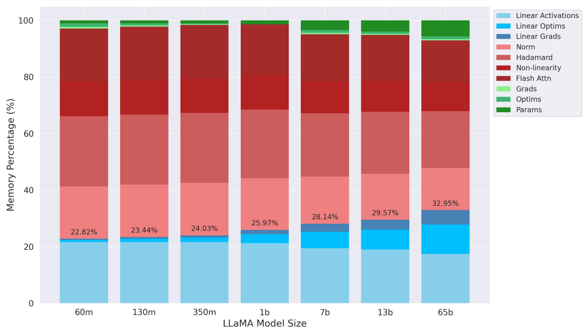


Figure 1: **Breakdown of memory components for various LLaMA model sizes,** with batch size 256. **Blue**: linear operations compressed by CompAct; **Red**: non-linear operations which CompAct doesn't compress; **Green**: model parameters and non-linear operation's optimizer states. Most of the memory is used by the computational graph. CompAct's compression gets more significant as model size increases, reaching almost 33% for LLaMA 65B. With $r = n/8$, this translates to almost 30% total memory saved.

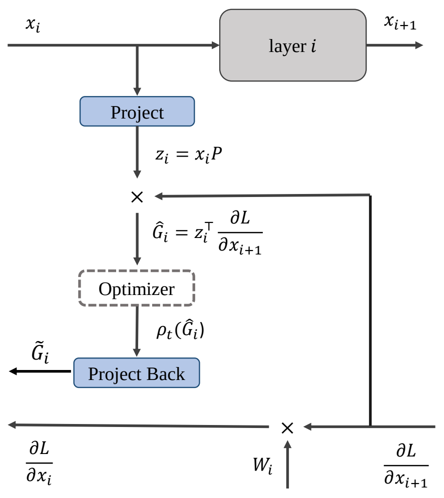
gradients after the backward pass (Muhamed et al., 2024; Hao et al., 2024; Zhao et al., 2024). However, as seen in Figure 1, the main memory component is the computation graph itself. Its size also scales with batch size, in contrast with other memory components.

In this work, we introduce CompAct, a novel optimization technique that saves low-rank-compressed activations during the forward pass, instead of the full activation tensors. Consequently, the resulting gradients are low-rank as well, also reducing the size of optimizer states. As CompAct decompresses the gradients back to full size only for the update step, it compresses a large part of the compute graph, which in turn translates to major memory savings. Figure 2 presents an overview of our method, while Table 1 lists the scaling of different memory components, compared to other compression methods. CompAct is a logical next step from previous work, moving from low-rank pa-

[*]Equal contribution.

	Original	LoRA	GaLore	Flora	CompAct
Weights	mn	$mr + nr$	mn	mn	mn
Gradients	mn	$mr + nr$	mn	mn	mr
Optim States	$2mn$	$2(mr + nr)$	$nr + 2mr$	$2mr$	$2mr$
Activations	bln	bln	bln	bln	blr

Table 1: **Theoretical Memory Consumption by the different stages of the training pipeline**, assuming linear layers $W_t \in \mathbb{R}^{n \times m}$ and $m > n$. b is batch size and l is sequence length. r is the dimensionality of the compressed activations and states.



x_i
layer i
x_{i+1}
Project
$z_i = x_i P$
$\hat{G}_i = z_i^T \frac{\partial L}{\partial x_{i+1}}$
Optimizer
$\rho_t(\hat{G}_i)$
$\tilde{G}_i$
Project Back
$\frac{\partial L}{\partial x_i}$
W_i
$\frac{\partial L}{\partial x_{i+1}}$

Figure 2: **Overview of CompAct.** For a given linear layer $x_{i+1} = x_i W_{i+1}$, we project its input x_i using a random projection matrix P, and save the result z_i for the backward pass. During the backward pass, we first compute the compressed gradients $\hat{G}_i$ and update the optimizer's parameter update function $\rho_t(\hat{G}_i)$. For Adam, ρ_t represents gradient normalization using the first and second gradient moments. Finally, we decompress the gradient back to the full parameter size $\tilde{G}_i$ and perform an update step.

rameters in (Hu et al., 2021), through compressed low-rank gradients in (Zhao et al., 2024), to compressed activations.

Overall, CompAct achieves savings of about 17.3% of *peak device memory with practical batch sizes*, for pretraining LLaMA-350M (Touvron et al., 2023), and 50% for fine-tuning RoBERTa-Base (Liu et al., 2019). As seen in Figure 1, the estimated memory reduction achieved by CompAct increases with model size. For LLaMA-65B, the estimated reduction is approximately 30%. For LLaMA-350M, the estimated reduction is around 21%, which is 4% higher than the observed empirical value. This suggests a realistic memory saving range of 25%-30% for LLaMA-65B.

Choosing the low-rank projection used for compression is critical, as it can impact performance and reduce training throughput. By using a random projection matrix sampled on the fly as in (Hao et al., 2024), we eliminate the cost of computing and storing optimal projection matrices, which can be slow to compute and large enough to reduce our memory savings.

We show sound theoretical motivation justifying the use of random projections from recent works in Random Sketching theory (Meier and Nakatsukasa, 2024). Finally, we present experimental measurements demonstrating that CompAct reduces memory significantly with minimal impact on model performance for both pretraining and finetuning.

2 Related Work

Memory-Efficient LLM Training Memory-efficient methods have become crucial for training LLMs, especially during pretraining where the memory requirements for activations, gradients, and optimizer states are significant as these models scale.

There are various approaches to making the training process more efficient, including mixed precision training (Micikevicius et al., 2017), quantization methods (Pan et al., 2022; Liu et al., 2022; Anonymous, 2024b; Dettmers et al., 2023; Anonymous, 2024a), parallelism techniques (Dean et al., 2012; Li et al., 2014; Shoeybi et al., 2020; Huang et al., 2019), and low rank approximation methods. The latter being mostly unexplored for pretraining, it is the focus of this work.

Low Rank Approximation Prior works on efficient pretraining such as LoRA (Hu et al., 2021) have largely focused on low-rank parametrization techniques, where the model's weights W are decomposed into a product of two smaller matrices $W = BA$. While this method can reduce memory usage and computational cost, it often leads to performance degradation, as the low-rank constraint limits the network's ability to represent complex patterns.

To address these limitations, approaches such as ReLoRA (Lialin et al., 2023) and SLTrain (Han et al., 2024) introduce more flexibility into the decomposition process, achieving a better balance between efficiency and performance. These methods go beyond strict low-rank parametrization by allowing for dynamic factorization, improving the network's expressiveness while retaining computational benefits.

Low Rank Gradient A recent approach, GaLore (Zhao et al., 2024) utilizes the low-rank property of gradients to learn a full-rank model efficiently, even during pretraining. Instead of approximating the model's weights, GaLore projects the gradients after the backward pass into a lower-dimensional subspace for the weight update, reducing the memory overhead without severely limiting the model's expressiveness.

As GaLore relies on periodic SVD computations to maintain adequate low-rank approximations of the gradients, which is very expensive in both time and memory, a variety of works focus on relieving this computational cost, either by strategically choosing the projection matrices (Anonymous, 2024c), quantizing them (Anonymous, 2024b), or replacing them by random projections (Muhamed et al., 2024; Hao et al., 2024). However, these methods remain inapplicable when pretraining large models, as peak device memory is primarily determined by the activations stored on GPU (when the batch size is large) (Anonymous, 2024b; Muhamed et al., 2024).

Essentially, compared to GaLore, our approach may be viewed as a change in the order of operations, applying the compression one step before GaLore does: when storing activations in memory for the backward pass, rather than to the gradients when updating the optimizer state. As a result, CompAct satisfies their convergence theorem, which explains how it achieves comparable performance despite the drastic memory savings.

Activation Compression Various works aim at reducing the memory cost of activations in deep learning in general. (Yang et al., 2024) is a complementary work focusing on saving activation memory generated by nonlinear functions and normalization layers, whereas our work focuses on the activations generated by linear layers. The two methods can be combined to achieve even greater savings, although some adaptation is required.

VeLoRA (Miles et al., 2024) also aims to compress linear layer activations, however, they apply their paradigm to two specific layers of the model only, thus making their benefit marginal. We compare with their projection in our experimental section, see Section 5. In any case, both (Miles et al., 2024) and (Yang et al., 2024) remain unexplored for the setting of pretraining LLMs.

Activation Checkpointing CKPT (Chen et al., 2016), also known as gradient checkpointing, reduces the memory footprint of the entire computation graph by saving the activations only at specific layers, or checkpoints. During backpropagation they recompute the forward pass between the current checkpoint and the previous one. The memory footprint of the entire compute graph can be reduced significantly, while incurring a 20%-30% compute cost overhead in most cases, as we empirically point out in Section 4.3.

3 Method

3.1 Background

Consider an input $x \in \mathbb{R}^{b \cdot l \times n}$ where b is the batch size, l is the sequence length, and n is the number of input features. A linear layer with parameter $W_t \in \mathbb{R}^{n \times m}$ at learning step t is applied as follows:

$$o = xW_t \in \mathbb{R}^{b \cdot l \times m}, \tag{1}$$

where we eliminated the bias term for simplicity, as it is unaffected by the method. During the forward pass, for each linear layer in the network, the input x is stored in memory at every intermediate layer. This is necessary for backpropagation, as it is used to compute the gradient of the weights using the chain rule:

$$G_t = \frac{\partial \mathcal{L}}{\partial W_t} = x^\top \frac{\partial \mathcal{L}}{\partial o} \in \mathbb{R}^{n \times m}. \tag{2}$$

Once the gradient is computed, it is used to update the weights in the subsequent time step

$$W_{t+1} = W_t - \eta \rho_t \left(\frac{\partial \mathcal{L}}{\partial W_t} \right). \tag{3}$$

Here, η represents the learning rate and ρ_t is an element-wise operation defined by the choice of optimizer, such as Adam.

Following the formulation in Zhao et al., 2024, GaLore projects the gradients into a lower-dimensional subspace before applying the optimizer, and projects them back to the original subspace for the weight update:

$$W_T = W_0 + \eta \sum_{t=0}^{T-1} \tilde{G}_t, \tag{4}$$

$$\tilde{G}_t = P_t \rho_t (P_t^\top G_t Q_t) Q_t^\top. \tag{5}$$

Here $P_t \in \mathbb{R}^{n \times r}$ and $Q_t \in \mathbb{R}^{m \times r}$ are projection matrices, and ρ_t is the optimizer such as Adam. In practice, to save memory, the projection is typically performed using only one of the two matrices, based on the smaller dimension between m and n. This approach allows for efficient gradient compression and memory savings. GaLore's theoretical foundation, including its convergence properties, is captured in the following theorem:

Theorem 1. *(Convergence of GaLore with fixed projections). Suppose the gradient follows the parametric form:*

$$G_t = \frac{1}{N} \sum_{i=1}^{N} (A_i - B_i W_t C_i) \tag{6}$$

with constant A_i, PSD matrices B_i and C_i after $t > t_0$, and A_i, B_i and C_i have L_A, L_B and L_C continuity with respect to W and $\|W_t\| \leq D$. Let $R_t := P_t^\top G_t Q_t, \hat{B}_{it} := P_t^\top B_i(W_t) P_t, \hat{C}_{it} := Q_t^\top C_i(W_t) Q_t$ and $\kappa_t := \frac{1}{N}\sum_i \lambda_{min}(\hat{B}_{it} \lambda_{min} \hat{C}_{it})$. If we choose constant $P_t = P$ and $Q_t = Q$, then GaLore with $\rho_t = 1$ satisfies:

$$\|R_t\|_F \leq \left[1 - \eta(\kappa_{r-1} - L_A - L_B L_C D^2\right] \|R_{t-1}\|_F \tag{7}$$

As a result, if $min_t \kappa_t > L_A + L_B L_C D^2, R_t \to 0$, and thus GaLore converges.

As stated in Theorem 1, the fastest convergence is achieved when projecting into a subspace corresponding to the largest eigenvalues of the matrices B_t, C_t. To approximate this, GaLore employs a Singular Value Decomposition (SVD) on the gradient G_t every T timesteps to update the projection matrix. T is called the *projection update period*.

Although this method reduces the memory cost of storing optimizer states, it introduces computational overhead due to the SVD calculation, and still requires saving the projection matrices in memory. The update period T also creates a tradeoff between optimal model performance and training time, since for small T the added SVD overhead becomes prohibitive, for large T the projection might become stale and hurt model performance.

3.2 CompAct

An overview of the method is described in Algorithms 1,2,3.

To reduce memory usage during training, we propose saving a projected version of the input $z = xP \in \mathbb{R}^{b \cdot l \times r}$ during the forward pass, where $P \in \mathbb{R}^{n \times r}$ is a projection matrix that maps the input to a lower-dimensional subspace. We choose r to be a fraction of each layer's total dimensionality n, to achieve a consistent compression rate. Other works such as (Zhao et al., 2024) chose the same r for all compressed layers, which we think reduces potential compression gains. In Section 4 we experiment with ratios $1/2, 1/4, 1/8$.

Using this low-rank projection P, the gradients with respect to the weights and the input are calculated as follows:

$$\hat{G}_t = z^\top \frac{\partial \mathcal{L}}{\partial o} \in \mathbb{R}^{r \times m}, \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial o} W_t^\top \in \mathbb{R}^{b \cdot l \times n}. \tag{9}$$

Our approach maintains the full forward pass, as well as the gradients with respect to the input. However, the gradients with respect to the weights are computed within the reduced subspace. This means that the optimizer states are also maintained in this smaller subspace. Similar to (Zhao et al., 2024),

$$M_t = \beta_1 M_{t-1} + (1 - \beta_1)\hat{G}_t,$$
$$V_t = \beta_2 V_{t-1} + (1 - \beta_2)\hat{G}_t^2,$$
$$\rho_t(\hat{G}_t) = M_t / \sqrt{V_t + \epsilon}$$

describes the Adam optimizer which we use in our analysis and experiments. Once the reduced gradient is obtained, we project it back to the original subspace for the full weight update using the same projection matrix P:

$$W_{t+1} = W_t - \eta \tilde{G}_t, \tag{10}$$

$$\tilde{G}_t = \alpha P \rho_t(\hat{G}_t). \tag{11}$$

Where α is an optional gradient scaling constant. By choosing P such that $PP^\top \approx I$, $\hat{G}_t$ is a good approximation for the full gradient G_t. This weight update is equivalent to GaLore's (Equation 4) when $Q = I$, so our method follows the convergence properties outlined in Theorem 1.

Algorithm 1 Forward Pass with CompAct

Input: An input $x \in \mathbb{R}^{b \cdot l \times n}$, a weight $W_t \in \mathbb{R}^{n \times m}$, a layer seed $s \in \mathbb{N}$, a rank r.

```
1: set_random_seed($s$)
2: $P \leftarrow \mathcal{N}(0, \frac{1}{r}) \in \mathbb{R}^{n \times r}$
3: $o \leftarrow xW_t \in \mathbb{R}^{b \cdot l \times m}$
4: $z \leftarrow xP \in \mathbb{R}^{b \cdot l \times r}$
5: save_for_backward($z$, $W_t$)
6: return $o$
```

Algorithm 2 Backward Pass with CompAct

Input: An output gradient $\frac{\partial \mathcal{L}}{\partial o} \in \mathbb{R}^{b \cdot l \times m}$, A compressed activation $z \in \mathbb{R}^{b \cdot l \times r}$ a weight $W_t \in \mathbb{R}^{n \times m}$.

```
1: $\hat{G}_t \leftarrow z^\top \frac{\partial \mathcal{L}}{\partial o} \in \mathbb{R}^{r \times m}$
2: $\frac{\partial \mathcal{L}}{\partial x} \leftarrow \frac{\partial \mathcal{L}}{\partial o} W_t^\top \in \mathbb{R}^{b \cdot l \times n}$
3: return $\frac{\partial \mathcal{L}}{\partial x}, \hat{G}_t$
```

3.3 Random Projection Matrix

Choosing a data-dependent projection such as the SVD used by (Zhao et al., 2024), invariably forces extra compute to generate the projection, as well as memory allocation to store it for the backward pass. Instead, following (Hao et al., 2024), we opt for a random, data-independent projection which can be efficiently sampled on-the-fly and resampled for the backward pass by using the same seed as the forward pass. We use a Gaussian Random Matrix, sampled independently for each layer with $\mu = 0$ and $\sigma = 1/r$: $P \in \mathbb{R}^{m \times r}, P_{ij} \sim \mathcal{N}(0, \frac{1}{r})$. Scaling the variance by $1/r$ ensures that $PP^\top \approx I$.

Using a random matrix from a Gaussian distribution ensures that the projected subspace maintains the norm of the original vectors with high probability (Dasgupta and Gupta, 2003; Indyk and Motwani, 1998), which is critical for preserving information (Hao et al., 2024). Additionally, changing the projection every T steps only required replacing the seed, which will not impact training throughput.

The following theorem by (Meier and Nakatsukasa, 2024) demonstrates that training converges quickly with high probability, while using a Gaussian Random Matrix P:

Theorem 2. ***(Sketching roughly preserves top singular values).*** *Let $P \in \mathbb{R}^{m \times r}$ have i.i.d entries sampled from $\mathcal{N}(0, \frac{1}{r})$, and a low rank matrix $A \in \mathbb{R}^{m \times n}$. We have:*

$$\frac{\sigma_i(P^\top A)}{\sigma_i(A)} = \mathcal{O}(1). \tag{12}$$

Where $\sigma_i(M)$ is the ith largest singular value of matrix M.

Theorem 2 shows that the ratio of singular values $\sigma_i(P^\top A)/\sigma_i(A)$ is fairly close to 1. As the main point in proving the convergence of GaLore is preserving the top singular values of the approximated matrix, this provides further motivation for why random sketching should work well.

Algorithm 3 Adam Update Step with CompAct

Input: A weight $W_t \in \mathbb{R}^{n \times m}$, a compressed gradient $\hat{G}_t \in \mathbb{R}^{r \times m}$, a random seed $s \in \mathbb{N}$, Adam decay rates β_1, β_2, scale α, learning rate η, rank r, projection update gap T.
Initialize Adam Moments $M_0, V_0 \in \mathbb{R}^{r \times m} \leftarrow 0, 0$
Initialize step $t \leftarrow 0$

```
1: $M_t \leftarrow \beta_1 \cdot M_{t-1} + (1 - \beta_1) \cdot \hat{G}_t$
2: $V_t \leftarrow \beta_2 \cdot V_{t-1} + (1 - \beta_2) \cdot \hat{G}_t^2$
3: $M_t \leftarrow M_t/(1 - \beta_1^t)$
4: $V_t \leftarrow V_t/(1 - \beta_2^t)$
5: $N_t \leftarrow M_t/(\sqrt{V_t} + \epsilon)$
6: set_random_seed($s$)
7: $P \leftarrow \mathcal{N}(0, \frac{1}{r})$
8: $\tilde{G}_t \leftarrow \alpha \cdot PN_t$          ▷ Project back to full dimension
9: $W_{t+1} \leftarrow W_t - \eta \cdot \tilde{G}_t$
10: $t \leftarrow t + 1$
11: if $t \mod T = 0$ then
12:     $s \leftarrow s + 1$          ▷ Update Random Seed
13: end if
14: return $W_t$
```

4 Experiments

In this section, we evaluate CompAct on both pretraining (Section 4.1) and finetuning tasks (Section 4.2). In all experiments, we apply CompAct to all attention and MLP blocks in all layers of the model, except for the output projection in the attention mechanism. For further details, see Appendix B. Moreover, we provide a comparison of CompAct's throughput and memory usage with other methods in Section 4.3, and explore various types of projection matrices in Section 5.

Model size	60M		130M		350M		1B	
	Perplexity	**GPU Peak**	**Perplexity**	**GPU Peak**	**Perplexity**	**GPU Peak**	**Perplexity**	**GPU Peak**
Full-Rank	34.06	11.59	25.08	18.66	18.80	39.97	15.56	-
GaLore	34.88	11.56	25.36	18.48	18.95	39.24	15.64	75.40
CompAct $r = n/2$	32.78	11.32	25.37	17.97	19.26	37.94	17.40	72.82
CompAct $r = n/4$	34.41	10.80	26.98	16.78	20.45	34.71	18.02	65.57
CompAct $r = n/8$	36.42	10.54	28.70	16.19	21.91	33.03	19.23	61.88
Training Tokens	1.1B		2.2B		6.4B		13.1B	

Table 2: **Pretraining perplexity and peak GPU memory for different model sizes and different training techniques.** Total training tokens are shown in the last row. As can be seen, CompAct reduces peak memory by up to 17% for LLaMA 350M, with comparable perplexity to baseline. For larger model sizes we estimate the total memory saving to be roughly 30%. The baseline for LLaMA 1B did not fit within the ~ 81 GB memory available at the same batch size.

4.1 Pretraining

For pretraining, we apply CompAct to LLaMA-based models (Touvron et al., 2023) of various sizes and train on the C4 (Colossal Clean Crawled Corpus) dataset, a commonly used dataset for training large-scale language models (Raffel et al., 2023b). The models were trained without any data repetition.

Our experimental setup follows the methodology outlined in (Zhao et al., 2024), using a LLaMA-based architecture that includes RMSNorm and SwiGLU activations (Shazeer, 2020). For each model size, we maintain the same set of hyperparameters, with the exception of the learning rate and the projection update gap which were tuned. Further details regarding the training setup and hyperparameters can be found in Appendix B.

As shown in Table 2, CompAct achieves performance comparable to full-rank training, while displaying a superior performance-to-memory trade-off at smaller ranks, successfully decreasing the peak allocated GPU memory by 17% in the largest model.

Additionally, We provide memory estimates of the various components for LLaMA 350M. As shown in Table 3, CompAct's memory savings are substantial across all stages of the training process, with notable reductions in the memory required for activations, gradients, and optimizer states in the linear layers. These savings are critical, as they significantly lower the overall memory footprint during training, possibly enabling larger models or batch sizes to be processed within the same hardware constraints.

	Original	**GaLore**	**CompAct**
Weights	0.65GB	0.65GB	0.65GB
Gradients	0.65GB	0.65GB	0.26GB
Optim States	1.3GB	0.54GB	0.52GB
Activations	7.0GB	7.0GB	2.87GB
Peak Memory	39.97GB	39.21GB	34.71GB

Table 3: **Estimated GPU memory consumption by different components of the training pipeline of LLaMA 350M, along the measured peak allocated GPU Memory.** All methods share an additional constant of activations that are not linear, explaining the gap between the sum of parts and the peak memory. Galore utilizes $r = 128$, while CompAct was measured with $r = n/4$.

4.2 Finetuning

We finetune the pretrained RoBERTa-base model (Liu et al., 2019) on the GLUE benchmark, a widely used suite for evaluating NLP models across various tasks, including sentiment analysis and question answering (Wang et al., 2019). We apply CompAct and compared its performance to GaLore. Following the training setup and hyperparameters from GaLore, we only tuned the learning rate. More details can be found in Appendix C

As shown in Table 4, CompAct achieves an extreme 50% reduction in the peak allocated GPU memory while delivering comparable performance.

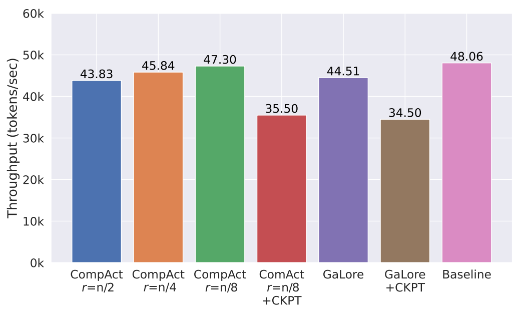
4.3 Peak Memory and Throughput

Methods that primarily compress the optimizer states, such as GaLore, often need to be combined with other memory-saving techniques like activation checkpointing to achieve meaningful reductions in memory usage during training. However,

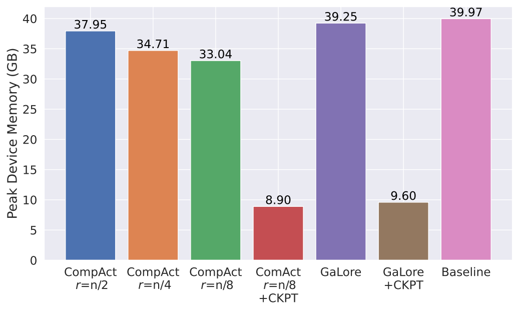
	Peak (MB)	CoLA	STS-B	MRPC	RTE	SST2	MNLI	QNLI	QQP	Avg
Full Fine-Tuning	6298	62.24	90.92	91.30	79.42	94.57	87.18	92.33	92.28	86.28
GaLore (r=4)	5816	60.35	90.73	92.25	79.42	94.04	87.00	92.24	91.06	85.89
CompAct (r=4)	**3092**	60.40	90.61	91.70	76.17	93.84	85.06	91.70	90.79	85.03
GaLore (r=8)	5819	60.06	90.82	92.01	79.78	94.38	87.17	92.20	91.11	85.94
CompAct (r=8)	**3102**	60.66	90.57	91.70	76.90	94.27	86.40	92.70	91.31	85.56

Table 4: **Finetuning performance on several benchmarks for various compression rates with GaLore and CompAct.** We report the empirical mean of three runs of our approach per task. Peak Memory was measured on RTE task. It is clear that both CompAct's and GaLore's performance is comparable with full finetuning, and very close to each other. However peak memory is vastly reduced with CompAct, with as much as 50% total memory saved. See Appendix C for the more details.

activation checkpointing introduces additional computational overhead by requiring activations to be recomputed during the backward pass (Chen et al., 2016), which can degrade training throughput. This trade-off means that while such methods may showcase memory benefits, they can negatively impact overall training efficiency.



(a)



(b)

Figure 3: **(a) Throughput and (b) peak device memory during pretraining of LLaMa-350M.** As can be seen, using smaller ranks with CompAct achieves better compression than GaLore while increasing the throughput. When applying activation checkpointing (CKPT), CompAct remains competitive, achieving better throughput and a smaller memory footprint.

We evaluate the throughput and memory peak of CompAct across various ranks and compare it against GaLore with and without activation checkpointing. All experiments were conducted using LLaMA-350M with the same hyperparameters. For Galore, we utilized their official repository and adopted their optimal rank $r = 256$ and projection update period $T = 200$ for training this model.

Our results in Figure 3 show that CompAct's reduction in peak GPU memory scales with r as expected, reaching 17.3% for $r = n/8$, while throughput also improves. This contrasts with the 1% reduction achieved by standard GaLore, highlighting our assertion that optimizer state isn't a major contributor to total memory.

In both methods, applying activation checkpointing (CKPT) improves memory savings significantly while hurting total throughput. CompAct is still better than GaLore when using CKPT, though only slightly.

5 Ablation Study

This section presents a series of ablation experiments examining how different design choices and hyperparameters affect training performance, memory usage, and convergence.

First, we explore the effects of different projection matrices on training performance when applied within the CompAct framework. We evaluate the following projection matrices:

Gaussian Projection This is our primary method, where each layer samples a different Gaussian random matrix.

Gaussian Projection with Shared Seed by setting the same random seed for all layers, we sample identical projection matrices for all layers (where dimensions permit). This investigates whether sharing the same subspace among different layers influences learning performance.

Sparse Johnson-Lindenstrauss (JL) Projection Matrices JL matrices have guarantees for norm preservation, while being sparse. As shown in (Muhamed et al., 2024), sparse operations can be highly efficient, and could point at future improvements for CompAct. We use the sparse JL matrix proposed in (Dasgupta et al., 2010).

VeLoRa Projection Matrices VeLoRa (Miles et al., 2024) is, to our knowledge, the only other work that addresses the compression of activations of linear layers. However, their approach projects the activations back to the original space during the backward pass and computes the gradients in full rank. They also projected only the Down and Value layers of LLaMA, where CompAct applies to all linear layers. We employ their projection matrix within CompAct to evaluate its impact on our method.

We opted not to experiment with Singular Value Decomposition (SVD)-based projections in our method due to practical considerations. More on SVD in Compact is discussed in Appendix A

For each type of projection matrix, we train LLaMA-60M with rank $r = n/4$. We conducted experiments using learning rates from $[1e-2, 5e-3, 1e-3]$. All other hyperparameters were identical to those used in Section 4.1.

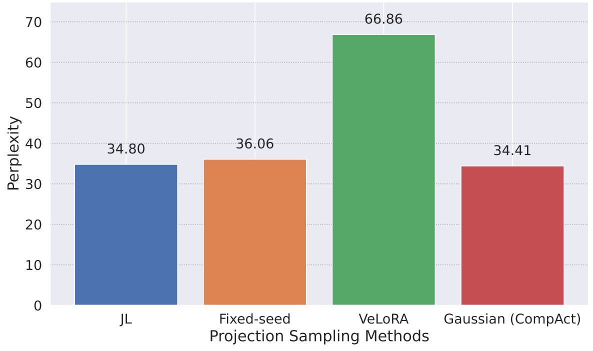


Figure 4: **Final model perplexity of CompAct with $r = n/4$ for different choices of projection matrices.** Both Gaussian seed choices and the JL projection achieve comparable results.

As shown in Figure 4, using Gaussian projections with different seeds per layer slightly improved performance compared to a shared seed, suggesting that utilizing different subspaces for different layers enhances the learning capacity of the model. Additionally, the sparse JL projections performed comparably to the dense Gaussian projections. This is a promising result, suggesting the viability of efficient sparse operations to further improve the benefits of CompAct. Finally, incorporating the projection matrix from VeLoRa into CompAct performed poorly. This can be attributed to differences in how VeLoRa handles the backward pass, by projecting activations back and computing full-rank gradients. This gap is somewhat expected, as they only used their projection on two types of layers, whereas we applied the compression more broadly. For the loss curves, see Appendix D.

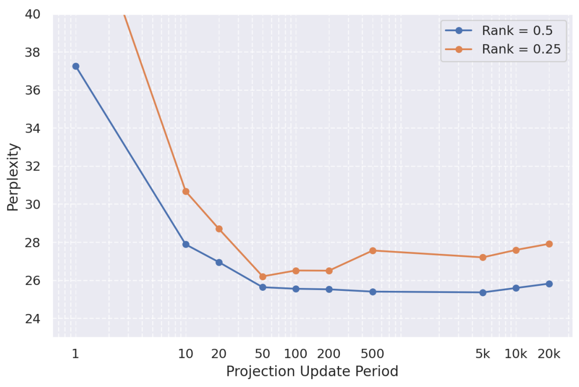


Figure 5: **Ablation on LlaMA 130M** - Effect of varying projection update periods T on performance across different ranks in CompAct.

Next, we examine how different training hyperparameters impact model convergence and memory efficiency.

Projection Update Period T: We first analyze the influence of the projection update period T on the convergence of of LLaMA-130M when trained with CompAct.To do so, we conduct experiments with varying update intervals. A learning rate of $5e-3$ is used by default, but if training becomes unstable at a given T, we reduce the learning rate until stability is achieved.
As shown in Figure 5, an optimal T range emerges: updating the projection matrix too frequently or too infrequently both slow down convergence. This trend remains consistent across the couple of ranks we show.

Rank and Training Steps: Next, we investigate the effect of rank and training duration on model convergence.Here, LLaMA-130M is trained with a projection update period of $T = 200$ and an initial learning rate of $5e-3$, which is reduced to $4e-3$ if instability occurs.
Figure 6 shows the impact of varying the rank of the projection matrix over different training durations.

As expected, lower ranks lead to more performance degradation, while increasing the rank improves model performance. Additionally, for any given rank, training for more steps yields better results. Crucially, larger ranks require fewer training steps to match or surpass the baseline, whereas lower ranks need extended training to compensate for their reduced expressivity. This highlights a trade-off between memory, training time, and final performance. For instance, when constrained by a small GPU, using a more aggressive compression (e.g., rank 0.25) allows training to fit within memory limits. Depending on the number of training steps available, this trade-off can still yield competitive performance, and with sufficient steps, even exceed the baseline.

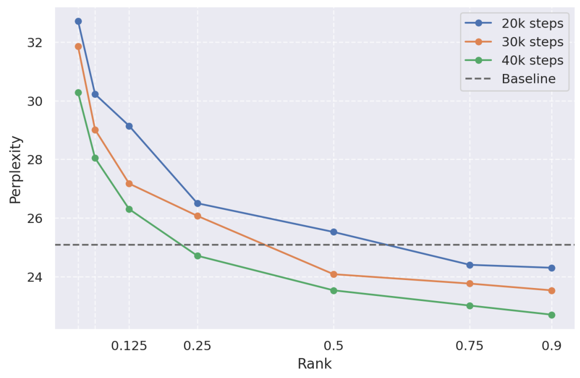


Figure 6: **Perplexity vs. Rank** - Effect of different ranks on the performance of CompAct-trained LLaMa-130M across varying training steps. The baseline (no compression) is trained for 20K steps.

Training Batch Size: Finally, we examine how batch size affects peak GPU memory usage, demonstrating the benefits of CompAct when handling varying activation sizes. We measure the peak GPU memory consumption while training LLaMA-350M without gradient accumulation across different batch sizes. The comparison includes the baseline model, GaLore (with a rank of 256), and CompAct (with a rank of 0.25).

As shown in Figure 7, CompAct significantly reduces peak memory usage, and its benefit scales with batch size, whereas GaLore provides a fixed reduction in peak memory relative to the baseline. This difference arises because GaLore primarily compresses optimizer states, which are independent of batch size, while CompAct reduces the memory footprint of activations, which grow with batch size.

This result is particularly important because larger batch sizes are known to improve training throughput(Andoorveedu et al., 2023). By lowering peak memory requirements, CompAct enables the use of larger batches, potentially increasing training efficiency without exceeding hardware constraints.

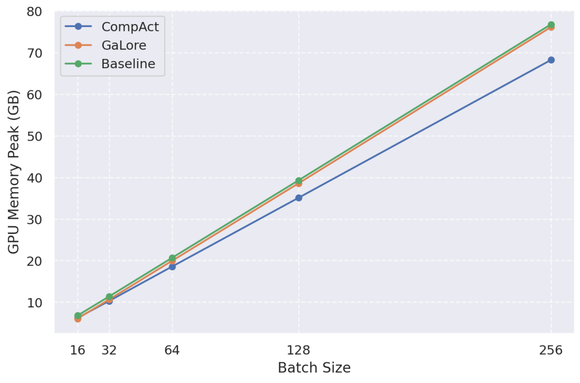


Figure 7: **Peak GPU memory vs. Batch Size.** Peak GPU memory usage of LLaMa-350M for different batch sizes is shown for CompAct, GaLore, and the baseline.

6 Conclusion

In this work, we presented CompAct, a memory-efficient method for training LLMs by compressing activations, gradients, and optimizer states of linear layers. We demonstrate that CompAct achieves significant memory savings for training LLMs, reaching 25%-30% memory reduction for pretraining LLaMA-65B and 50% for RoBERTa-Base, with minimal impact on training throughput and performance. Our method is easily scalable, applicable to various model sizes, and is easily composable with other techniques.

By directly compressing the compute graph during training, CompAct targets a major component of peak device memory which was neglected in recent works. We believe this approach should guide future work for further memory gains. A good example could be incorporating sparse random projections into CompAct, which would reduce the computational cost associated with sampling and matrix operations. Another area for improvement is the approximation of intermediate activations, such as those generated by FlashAttention, Hadamard products, and non-linearities, which require significant memory. By addressing these memory-intensive operations, CompAct's memory reductions can be extended even further.

7 Limitations

While CompAct offers significant memory savings and maintains high throughput, a few limitations should be noted.

First, although using Gaussian random matrices allows for on-the-fly sampling and eliminates the memory overhead of storing projection matrices, it can introduce some computational overhead due to frequent sampling and multiplications. A possible solution is to replace them with sparse random projections. These could not only reduce the computational cost but also improve throughput by fusing the sampling and multiplication steps, potentially outperforming the current baseline.

Another limitation of CompAct is that it currently focuses on compressing linear layers, leaving other memory-intensive operations, such as FlashAttention and Hadamard products, uncompressed. These operations consume substantial memory, and future work could explore compressing their activations directly within the computation graph without compromising model performance.

Finally, while we demonstrated that CompAct provides additional memory savings when combined with activation checkpointing—by avoiding the need to store full gradients in memory—the integration could be further optimized. Recomputing the activation compression during the backward pass could reduce the overhead introduced by checkpointing and improve throughput. Integrating CompAct with methods such as those proposed in (Yang et al., 2024) could further smooth this process and enhance training efficiency.

References

Muralidhar Andoorveedu, Zhanda Zhu, Bojian Zheng, and Gennady Pekhimenko. 2023. Tempo: Accelerating transformer-based model training through memory footprint reduction. *Preprint*, arXiv:2210.10246.

Anonymous. 2024a. COAT: Compressing optimizer states and activations for memory-efficient FP8 training. In *Submitted to The Thirteenth International Conference on Learning Representations*. Under review.

Anonymous. 2024b. Q-galore: Quantized galore with INT4 projection and layer-adaptive low-rank gradients. In *Submitted to The Thirteenth International Conference on Learning Representations*. Under review.

Anonymous. 2024c. Subtrack your grad: Gradient subspace tracking for memory-efficient LLM training and fine-tuning. In *Submitted to The Thirteenth International Conference on Learning Representations*. Under review.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. *Preprint*, arXiv:2005.14165.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *Preprint*, arXiv:1604.06174.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Preprint*, arXiv:2205.14135.

Anirban Dasgupta, Ravi Kumar, and Tamás Sarlós. 2010. A sparse johnson–lindenstrauss transform. *CoRR*, abs/1004.4240.

Sanjoy Dasgupta and Anupam Gupta. 2003. An elementary proof of a theorem of johnson and lindenstrauss. *Random Structures & Algorithms*, 22(1):60–65.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. 2012. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Preprint*, arXiv:2305.14314.

Andi Han, Jiaxiang Li, Wei Huang, Mingyi Hong, Akiko Takeda, Pratik Jawanpuria, and Bamdev Mishra. 2024. Sltrain: a sparse plus low-rank approach for parameter and memory efficient pretraining. *Preprint*, arXiv:2406.02214.

Yongchang Hao, Yanshuai Cao, and Lili Mou. 2024. Flora: Low-rank adapters are secretly gradient compressors. *Preprint*, arXiv:2402.03293.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, page 604–613, New York, NY, USA. Association for Computing Machinery.

Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. 2014. Scaling distributed machine learning with the parameter server. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 583–598, USA. USENIX Association.

Vladislav Lialin, Namrata Shivagunde, Sherin Muckatira, and Anna Rumshisky. 2023. Relora: High-rank training through low-rank updates. *Preprint*, arXiv:2307.05695.

Xiaoxuan Liu, Lianmin Zheng, Dequan Wang, Yukuo Cen, Weize Chen, Xu Han, Jianfei Chen, Zhiyuan Liu, Jie Tang, Joey Gonzalez, Michael Mahoney, and Alvin Cheung. 2022. Gact: Activation compressed training for generic network architectures. *Preprint*, arXiv:2206.11357.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *Preprint*, arXiv:1907.11692.

Maike Meier and Yuji Nakatsukasa. 2024. Fast randomized numerical rank estimation for numerically low-rank matrices. *Preprint*, arXiv:2105.07388.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2017. Mixed precision training. *Preprint*, arXiv:1710.03740.

Roy Miles, Pradyumna Reddy, Ismail Elezi, and Jiankang Deng. 2024. Velora: Memory efficient training using rank-1 sub-token projections. *Preprint*, arXiv:2405.17991.

Aashiq Muhamed, Oscar Li, David Woodruff, Mona Diab, and Virginia Smith. 2024. Grass: Compute efficient low-memory llm training with structured sparse gradients. *Preprint*, arXiv:2406.17660.

Zizheng Pan, Peng Chen, Haoyu He, Jing Liu, Jianfei Cai, and Bohan Zhuang. 2022. Mesa: A memory-saving training framework for transformers. *Preprint*, arXiv:2111.11124.

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

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

Noam Shazeer. 2020. Glu variants improve transformer. *Preprint*, arXiv:2002.05202.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-lm: Training multi-billion parameter language models using model parallelism. *Preprint*, arXiv:1909.08053.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. Glue: A multi-task benchmark and analysis platform for natural language understanding. *Preprint*, arXiv:1804.07461.

Yuchen Yang, Yingdong Shi, Cheems Wang, Xiantong Zhen, Yuxuan Shi, and Jun Xu. 2024. Reducing fine-tuning memory overhead by approximate and memory-sharing backpropagation. *Preprint*, arXiv:2406.16282.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. 2024. Galore: Memory-efficient llm training by gradient low-rank projection. *Preprint*, arXiv:2403.03507.

A CompAct with SVD

In this work, random projections are used to compress activations, thereby avoiding the costly SVD step. Performing an SVD on the activation tensor $x \in \mathbb{R}^{b \times l \times n}$, where b is the batch size and l is the sequence length, can introduce considerable overhead since $b \times l$ is typically large. By contrast, GaLore applies SVD to the gradient $G \in \mathbb{R}^{n \times m}$, which does not depend on b or l, making it generally less expensive than compressing activations via SVD.

Nonetheless, one could adopt GaLore's SVD-based projection within the CompAct framework by running an iteration with uncompressed activations to compute the gradient, from which the SVD projection is derived and then stored for subsequent updates. However, each time the projection matrix is updated, storing the entire activation tensor would significantly increase the GPU memory peak, since the activations typically dominate memory usage. A potential mitigation strategy might involve updating the projection matrix on smaller batches to reduce peak memory requirements. Further exploration of this approach is left for future work.

B Pretraining

This appendix provides further details about our pre-training experiments.

B.1 Hyperparameters

We adopt the training setup outlined in (Zhao et al., 2024), and apply compact to LLaMA-based models of various sizes. Table 5 outlines the amount of data and steps used to train the models.

Model Size	Steps	Training Tokens
60M	10K	1.3 B
130M	20K	2.6 B
350M	60K	7.8 B
1B	100K	13.1 B

Table 5: **Training Step for Llama models.**

For all the trained models, we use a maximum sequence length of 256, with a batch size of 512. We tune the optimal learning rate for each model size from the range $1e-3 \leq lr \leq 1e-2$, and choose the best one based on the validation perplexity. We also adopt a learning rate warmup for the first 10% of the training steps and use a cosine learning rate scheduler that decreases to 0.1 of the initial learning rate.
We use a constant scaling factor of $\alpha = 0.25$. Additionally, we tuned the projection update, using $T = 50$ period T for after searching over [1,10,50,100,200,500,∞] for the LLaMA-130M model, which was then used to train 60M - 350M. For the 1B model, we use $T = 200$.

B.2 CompAct and FlashAttention

We applied CompAct to all linear modules within each Transformer block of our LLaMA-based models, except for the output-projection layer in the attention mechanism. FlashAttention (Dao et al., 2022), a fast and efficient implementation of attention, is widely adopted in many architectures, including the models used in this work for both pretraining and finetuning.

However, FlashAttention stores its output in memory as part of the activation memory. This same tensor is then fed into the output-projection layer of the attention block, meaning that these two layers share the same activation memory. Consequently, compressing the activation tensor of the attention's output-projection does not result in overall memory savings, because the FlashAttention output tensor remains in memory. Further, compressing the shared activation between these two layers would introduce errors in the gradient with respect to the FlashAttention input, potentially causing error accumulation across layers. Addressing this issue lies outside the current scope of CompAct.
As a result, we left the output-projection layer uncompressed in our experiments. In future work, a custom CUDA kernel for FlashAttention may allow compression of this layer without introducing errors.

However, not compressing this layer introduced instability in the pretraining experiments when training with larger learning rates, due to the inconsistent learning rate across different linear layers induced by the scale α. To mitigate this, one can either adjust the learning rate of the output-projection layer by a scale α_{out}, or simply use a smaller learning rate.

In our experiments, when training the smaller models 60M-350M we used a large learning rate of 0.01 and scaled the learning of the output projection with $\alpha_{out} = 0.5\alpha$, while for the larger model we simply used a smaller learning rate of 0.003.

B.3 Type Conversion in Normalization Layers

We note that our implementation of LLaMA's RMSNorm layers did not apply type conversion during pretraining, as we observed that it did not affect model perplexity, but required extra activations. The baseline was measured without type conversion as well making the comparison fair. Hence, all layers were computed in the type of bfloat16 floating point format.

For our finetuning experiments, we presented Roberta-Base, which applies Layer Norm normalization layers rather than RMSNorm, whose default implementations do include type conversions, from bfloat16 to float32 floating point format, although this is very negligible as the effect of these on peak memory is tiny in finetuning.

C Fine-tuning

To be comparable to the results reported in GaLore (Zhao et al., 2024) as shown in Table 4 we report the same metrics as they did, namely, F1 score on QQP and MRPC, Matthew's Correlation for CoLA, Pearson's Correlation for STS-B, and classification accuracy for all others. The numbers reported for GaLore and Baseline are taken from (Zhao et al., 2024). We report the *average* performance over three seeds due to the noisy behavior of the training process. All models were trained for 30 epochs with batch size 16, except for CoLA where we used batch size 32 as in GaLore, and a maximum sequence length of 512, a scale $\alpha = 2$ was used with $r = 8$ and $\alpha = 4$ for $r = 4$, all with $T = 500$. Again, as in GaLore, we searched for a best learning rate per task, searching in $\{1e-5, 2e-5, 3e-5\}$.

D Choice of Projection Matrix - Loss Curve

As can be seen below, all different projection types did converge, strengthening the comparison. We can see small spikes in loss when applying VeLoRA's projection every $T = 50$ timesteps.

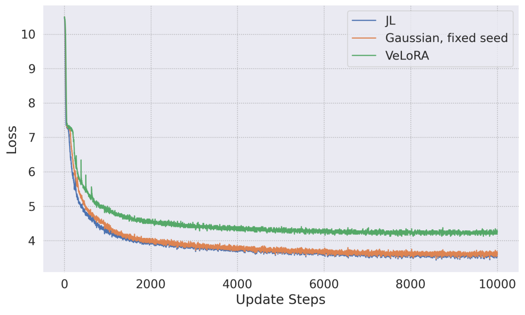


Figure 8: **Training Loss with $r = n/4$ for different choices of projection matrices.** Both Gaussian seed choices and the JL projection achieve comparable results. VeLoRA achieves poor results and is more sensitive to the spike at the beginning of training.

E Memory Estimation for Llama Model

In this section, we describe how we calculated the memory estimates used throughout the paper to illustrate CompAct's memory savings.

First, note that given the number of parameters in a model, the memory needed to store gradients and optimizer states can be estimated directly, as it scales with the number of parameters. To estimate activation memory, we examined the activations required by a single Transformer block in a LLaMA-based model as mapped in Table 6. We then used this to calculate the total activation memory necessary for training LLaMA models of various sizes.

Lastly, we used PyTorch's memory profiler to confirm that our estimated values are consistent with the actual memory consumption observed during training.

Table 6: Activations saved from each operation in a single transformer block from a LlaMA model. In this block, the input is of shape (b, l, n), the attention weights W_q, W_k, W_v, W_o of shape (n, n) and the MLP weights $W_{down}^{\top}, W_{up}, W_{gate}$ are of shape $(n, m), n < m$. Note that the activations of RMSNorm don't include the precision conversion as we didn't use it in our training. Smaller activations in CompAct are marked in **Red**

Operation	**Tensors Saved for Backward**	**Tensors Saved With CompAct** $(0 <= r <= 1)$
$x_2 = RMSNorm(x_1)$	$x_1 \in \mathbb{R}^{b \times l \times n}$ $(\sigma(x_1)^2 + \varepsilon)^{-\frac{1}{2}} \in \mathbb{R}^{b \times l}$	$x_1 \in \mathbb{R}^{b \times l \times n}$ $(\sigma(x_1)^2 + \varepsilon)^{-\frac{1}{2}} \in \mathbb{R}^{b \times l}$
$q = x_2 W_q^{\top}$ $k = x_2 W_k^{\top}$ $v = x_2 W_v^{\top}$	$x_2 \in \mathbb{R}^{b \times l \times n}$	$(x_2 P_q), (x_2 P_v), (x_2 P_k) \in \mathbb{R}^{b \times l \times nr}$
reshape q, k, v to $(b\,h, l, n/h)$	None	None
$x_3 = flash\_attn(q, k, v)$	$q, k, v \in \mathbb{R}^{b \times h \times l \times n/h}$ two buffers $\in \mathbb{R}^{b \times l \times h}$ $x_3 \in \mathbb{R}^{b \times h \times l \times n/h}$	$q, k, v \in \mathbb{R}^{b \times h \times l \times n/h}$ two buffers $\in \mathbb{R}^{b \times l \times h}$ $x_3 \in \mathbb{R}^{b \times h \times l \times n/h}$
reshape x_3 to (b, l, n)	None	None
$x_4 = x_3 W_o^{\top}$	$x_3 \in \mathbb{R}^{b \times h \times l \times n/h}$ Shared with flash-attn	$x_3 \in \mathbb{R}^{b \times h \times l \times n/h}$ Shared with flash-attn
residual: $x_5 = x_4 + x_1$	None	None
$x_6 = RMSNorm(x_5)$	$x_5 \in \mathbb{R}^{b \times l \times n}$ $(\sigma(x_5)^2 + \varepsilon)^{-\frac{1}{2}} \in \mathbb{R}^{b \times l}$	$x_5 \in \mathbb{R}^{b \times l \times n}$ $(\sigma(x_5)^2 + \varepsilon)^{-\frac{1}{2}} \in \mathbb{R}^{b \times l}$
$x_{gate} = x_6 W_{gate}^{\top}$ $x_{up} = x_6 W_{up}^{\top}$	$x_6 \in \mathbb{R}^{b \times l \times n}$	$(x_6 P_{gate}), (x_6 P_{up}) \in \mathbb{R}^{b \times l \times nr}$
$x_{act} = SiLU(x_{gate}$	$x_{gate} \in \mathbb{R}^{b \times l \times m}$	$x_{gate} \in \mathbb{R}^{b \times l \times m}$
$x_7 = x_{act} * x_{up}$	$x_{up} \in \mathbb{R}^{b \times l \times m}$ $x_{act} \in \mathbb{R}^{b \times l \times m}$	$x_{up} \in \mathbb{R}^{b \times l \times m}$ $x_{act} \in \mathbb{R}^{b \times l \times m}$
$x_{down} = x_7 W_{down}^{\top}$	$x_7 \in \mathbb{R}^{b \times l \times m}$	$(x_7 P_{down}) \in \mathbb{R}^{b \times l \times mr}$
residual: $x_8 = x_down + x_5$	None	None