Acceleration of Backpropagation in Linear Layers of Transformer Models Based on Gradient Structure

Dmitrii Topchii[1], **Alexander Panchenko**[1,2], **and Viktoriia A. Chekalina**[2]
[1]Skoltech, [2]AIRI
Correspondence: Dmitriy.Topchiy@skol.tech, chekalina@airi.net

Abstract

Fine-tuning Transformer models is often dominated by the backward computation in linear layers. In many NLP tasks, input sequences are short and padded to a fixed context length, inducing structured sparsity in the output gradients. We propose Sparsity-Exploiting Backward Pass (SEBP), a heuristic method that reduces backward computation by exploiting this sparsity with negligible memory overhead. We show that, for short input sequences, the output gradients of BERT-based and LLaMA models exhibit pronounced sparsity, allowing for optimisation in the backward computation. We optimized the autograd function in the linear layers, significantly reducing the number of FLOPs during the backward.

Our method achieves a backward pass speedup of approximately 2.15x for BERT-base on GLUE tasks and 1.99x for a 3B LLaMA model on reasoning benchmarks, while maintaining memory usage nearly identical to the regular PyTorch fine-tuning. Crucially, this speedup comes at no cost to performance. We show that our method matches standard convergence rates, offering a memory-efficient way to accelerate LLM fine-tuning.

1 Introduction

Deep neural networks in general and the Transformer architecture (Vaswani et al., 2017) in particular created the foundation of modern Large Language Models (LLMs). However, training of such models is computationally intensive, with backpropagation through linear layers being a primary bottleneck. While these models are designed for long context lengths, many NLP benchmarks like GLUE (Wang et al., 2019) use short sequences that are padded to meet the model's required input size. This padding introduces numerous zero-value tokens, leading to significant redundant computation during the backward pass.

Although SOTA optimization libraries like DeepSpeed (Rasley et al., 2020) effectively address training speed, they often do so at the cost of substantially increased memory consumption. This paper introduces the Sparsity-Exploiting Backward Pass (SEBP), a training method which achieves a significant training acceleration with virtually no memory overhead by exploiting the padding-induced gradient sparsity to create a highly efficient, memory-frugal training process. Crucially, SEBP distinguishes itself by operating dynamically on activation gradients rather than model weights. By identifying and retaining only the rows with significant gradient magnitudes -which inherently align with non-padded, information-rich tokens -our method effectively decouples computational cost from the fixed sequence length. This allows us to convert large, wasteful sparse-dense multiplications into compact dense -dense operations using a custom Triton kernel, ensuring that hardware resources are dedicated solely to learning valid features rather than processing void padding tokens.

The **contributions** of this work are following:

1. We provide a detailed *analysis* of structured row sparsity in the output gradients of models like BERT, RoBERTa and LLaMa, demonstrating that this is a direct result of processing padded sequences common in many NLP benchmarks.

2. We propose a lightweight training heuristic *method* SEBP, that accelerates the backward pass by processing only a fixed number of top gradient rows. The method provides speedup by reducing number of FLOPs without the need for additional memory.

3. We validate *experimentally* that efficiency gains do not compromise model quality.

The code for the method is available online.[1]

[1]https://github.com/Dmitrii-Topchii/SEBP

Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics
Volume 4: Student Research Workshop, pages 426–436
March 26, 2026 ©2026 Association for Computational Linguistics

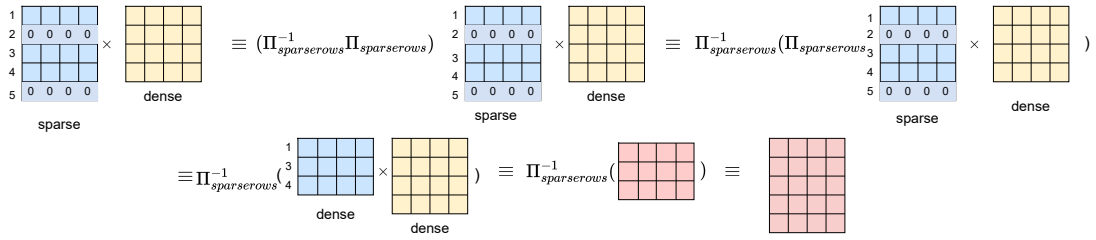


Figure 1: SEBP overview. Row sparsity in $\frac{\partial L}{\partial Y}$ lets us replace a large sparse–dense product with a compact dense–dense one, *reducing backward-pass FLOPs*.

2 Related Work

Training acceleration for Large Language Models (LLMs) (Vaswani et al., 2017; Devlin et al., 2019; Meta AI, 2024) is a critical area of research, as standard fine-tuning incurs substantial memory and computational costs. Recent surveys on distributed LLM training (Amini et al., 2025) and memory-efficient techniques (Tian et al., 2025) highlight that while Parameter-Efficient Fine-Tuning (PEFT) methods reduce parameter-associated memory, they often fail to address the high computational load and activation memory footprint of the backward pass. Several strategies have been proposed to tackle these challenges, which we categorize below and contrast with our proposed method.

System-Level Optimization: DeepSpeed (Rasley et al., 2020) is a comprehensive optimization library that partitions model states (parameters, gradients, and optimizer states) across data-parallel GPUs using the Zero Redundancy Optimizer (ZeRO). Comparison: While DeepSpeed effectively reduces per-device memory, it does so by distributing the load, which can increase total system memory usage and communication overhead. In contrast, SEBP reduces the fundamental computational cost (FLOPs) of the backward pass itself without requiring multi-GPU communication, making it orthogonal and potentially complementary to ZeRO-based systems.

Layer Dropping: DropBP (Dropping Backward Propagation) (Woo et al., 2024) reduces computational cost by randomly dropping entire layers during the backward pass based on a sensitivity metric. This is conceptually equivalent to training smaller submodules defined by the undropped layers. Comparison: DropBP operates at the granularity of whole layers, essentially skipping updates entirely. SEBP operates at a much finer granularity—individual rows within the dense layers—allowing us to retain updates for the most critical features in every layer, rather than sacrificing entire layers stochastically.

Sparse Training: RigL (Rigging the Lottery) (Evci et al., 2020) optimizes sparse training by dynamically updating a sparse weight mask throughout the process. It uses gradient information to periodically remove less salient weights and activate new ones, allowing the exploration of different connectivity patterns. Comparison: RigL focuses on weight sparsity (pruning connections permanently or temporarily) to find performant subnetworks (Cheng et al., 2024). SEBP, however, introduces *gradient sparsity* during backpropagation. We do not prune the model weights themselves; instead, we sparsify the error signal propagating backward. This ensures the forward pass remains fully dense and accurate, while the backward pass becomes computationally cheaper.

Gradient Sparsification: Our work builds directly upon SparseGrad (Chekalina et al., 2024), which introduced the concept of selective backpropagation for MLP layers. Comparison: We extend this foundation by implementing a highly optimized Triton (Tillet et al., 2019) kernel that translates theoretical FLOP reductions into tangible wall-clock speedups on modern hardware (A100), demonstrating its effectiveness on large-scale models like Llama 3 (Meta AI, 2024).

3 Methodology

In PyTorch's autograd backward through a linear layer $Y = XW^T$, we need to calculate the gradient with respect to the weights $\frac{\partial L}{\partial W}$ and the gradient with respect to the input $\frac{\partial L}{\partial X}$.

Assuming output gradient as $\frac{\partial L}{\partial Y}$, the following matrix multiplications that occur during the backward pass imply that:

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W \tag{1}$$

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y}^T X \tag{2}$$

The elimination of rows of the left matrix in Equation 2 can significantly accelerate the computation. What we need is to examine the structure $\frac{\partial L}{\partial Y}^T X$ across different cases.

3.1 Sparsity Analysis

We define a matrix as *row-sparse* if it contains rows composed entirely of zero elements. The *sparsity level*, or simply *sparsity*, refers to the ratio of zero elements to the total number of elements in the matrix.

Hypothesis. *Input sequences with a small number of tokens relative to the model context length lead to row sparsity.*

Proof. We aimed to analyze how the length of the model's input sequence influences the sparsity of the output gradient in each linear layer of the model.

Sparsity on encoder models. We fine-tune pretrained BERT-base and RoBERTa- base models for one epoch on datasets with a small number of tokens per sample (all datasets from the GLUE benchmark fall into this category), as well as on a dataset whose token length equals the model's context length (WikiText-103). We analyze the output gradients in the linear layers of each MLP block (*intermediate* and *output*).

In Table 1, we report statistics of dataset object lengths as well as the number of zero elements in $\frac{\partial L}{\partial Y}$. The results show that when the input sequence is short, sparsity is high; conversely, when the input sequence matches the context length, the number of zero elements is low. Table 1 therefore provides evidence that short input sequences lead to a high number of zero elements.

An example snapshot of the gradients is shown in Figure 2 (top row), which clearly demonstrates the presence of row sparsity.

The corresponding results for RoBERTa are in Appendix.

Sparsity on decoder models. Following our analysis on encoder models, we extended the investigation to include a causal decoder-only architecture. We fine-tuned LLaMa 3.2 3B on the OASST1Köpf et al. (2023) dataset with an object length of 512 for a "short" input, and on the same dataset with a length equal to the model's context for "long" ones.

As is shown in Figure 2 (bottom row), the row-sparsity pattern is also observed. However, some elements in gradients were not strictly zero but

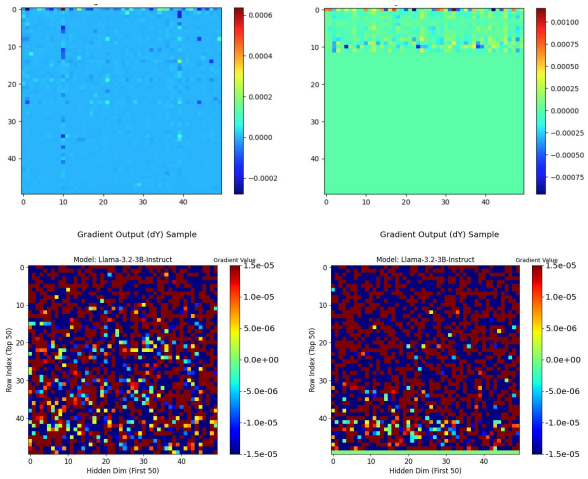


Figure 2: Output-gradient heatmaps: top row is for BERT with 512 context, bottom for LLaMA-3.2-3B with 4096 context length. On the left, the gradient for a "long" sample is shown; on the right, for a "short" one.

rather appeared as noise close to zero. The histogram of these element values for a short LLaMa 3.2 3B input is shown in Figure 3. We decided to determine a threshold below which an element can be considered zero. To find the precise noise truncation threshold (ϵ) for LLaMa 3, we analyze the distribution of gradient values (Figure 3) using their Cumulative Distribution Function (CDF) derived from a histogram. After applying smoothing to the CDF and automatically detecting the "knee point" - the location of the maximum slope change. This identified point $\epsilon = 3.98 * 10^{-5}$ is subsequently used to zero out the noise, revealing the true underlying *row-sparse* structure of the gradients.

Using this thresholding, we computed the sparsity of the output gradients with respect to the average object length across OASST1, language modeling datasets, and commonsense reasoning datasets. Table 2 shows that for samples whose length is significantly smaller than the context length, the sparsity in the output gradients is substantial.

Therefore, the hypothesis is confirmed. □

Considering the rule of matrix multiplication in Equation 2, we utilized the row-sparse structure of the output gradient to accelerate the backward pass. For short inputs, the dense-to-dense matrix multiplication of $\frac{\partial L}{\partial Y}$ and W, we provide sparse-dense multiplication. This is described in detail in Figure 1. First, we apply permutation to a sparse matrix which selects only non-zero rows, then multiply the obtained dense matrix with a lower height to dense. If sparsity coefficient is sufficient, left matrix size reduces significantly and we get savings in

Dataset	WikiText	STSB	CoLA	WNLI	MRPC	RTE	QQP	SST2	MNLI
	#Tokens per sample								
AVG	512	27	47	37	53	66	30	13	39
Max	512	125	111	108	103	128	128	66	128
Min	512	10	4	16	19	13	6	3	5
	#Sparsity per layer intermediate								
AVG	0.08	0.71	0.86	0.91	0.89	0.78	0.79	0.77	0.54
Max	0.14	0.79	0.98	0.97	0.94	0.88	0.82	0.81	0.62
Min	0.01	0.60	0.90	0.75	0.86	0.85	0.74	0.72	0.48
	#Sparsity per layer output								
AVG	0.10	0.54	0.40	0.66	0.26	0.52	0.84	0.85	0.81
Max	0.11	0.94	0.51	0.91	0.34	0.73	0.94	0.96	0.94
Min	0.01	0.41	0.27	0.56	0.19	0.48	0.78	0.77	0.77

Table 1: Average sparsity of the gradient output $\frac{\partial L}{\partial Y}$ in BERT's linear layers across datasets of varying lengths. Full model context is 512. Datasets with fewer tokens per sample exhibit higher sparsity.

Dataset	WikiText	PTB	Winogrande	ARC-E	ARC-C	HellaSwag	OASST1 (512)	OASST1 (4096)
#Tokens per sample								
AVG	104	27	32	57	67	194	512	4096
Max	582	112	48	196	194	364	512	4096
Min	3	3	26	26	26	53	512	4096
#Sparsity per layer intermediate								
AVG	0.82	0.81	0.80	0.80	0.80	0.79	0.87	0.007
Max	0.99	0.99	0.99	0.99	0.99	1.00	0.87	0.009
Min	0.49	0.48	0.50	0.51	0.51	0.49	0.87	0.002
#Sparsity per layer output								
AVG	0.50	0.49	0.48	0.48	0.48	0.47	0.87	0.002
Max	0.94	0.94	0.94	0.94	0.94	0.94	0.87	0.002
Min	0.08	0.07	0.08	0.08	0.08	0.08	0.87	0.002

Table 2: Average sparsity of the gradient output $\frac{\partial L}{\partial Y}$ in Llama-3.2-3B-Instruct linear layers across datasets of varying lengths. Full model context is 4096. Datasets with fewer tokens per sample exhibit higher sparsity.

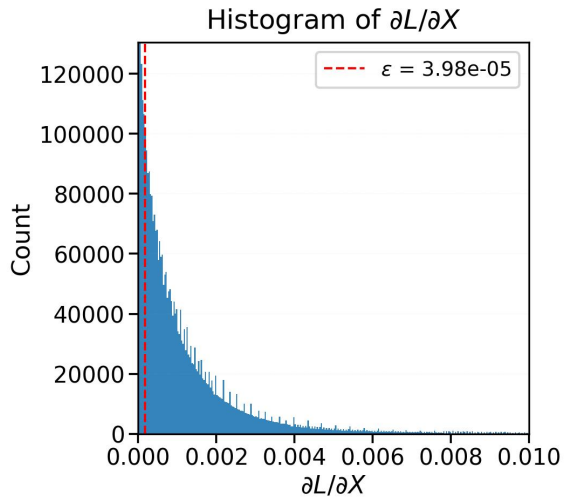


Figure 3: Histogram of gradient magnitudes for a short LLaMA-3.2-3B input (used to select ϵ).

computational iterations in matrix multiplication. Finally, we restore the original size of the result matrix by applying inverted permutation to it.

3.2 FLOPS estimation

Let us estimate the FLOPS required for the operation $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} W$. If b - batchsize, $W \in \mathbb{R}^{m \times n}$, $\frac{\partial L}{\partial Y} \in \mathbb{R}^{b \times m}$:

- For dense-to-dense matrix multiplication, we have FLOPS $\approx 2 \cdot m \cdot b \cdot n$.
- For sparse-dense matrix multiplication where $\frac{\partial L}{\partial Y}$ has k non-zeros rows, we have FLOPS $\approx 2 \cdot k \cdot b \cdot n$.

It is notable that if we consider only the non-zero rows (which, for the backward pass in GLUE tasks, constitute on average approximately 20% of the total, so $\frac{m}{k} = 5$), we achieve a 5-fold acceleration in FLOPS in average. So, SEBP is a training-time heuristic that exploits row sparsity to *reduce backward-pass FLOPs* with negligible overhead.

Implementation. To run SEBP, we provide custom `torch.autograd.Function` that overrides

only the backward path of linear layers. Forward semantics and the definition of gradients are unchanged, so SEBP is a drop-in optimization.

To accelerate the selection of non-zero rows and the subsequent matrix multiplication, we implemented a custom Triton kernel. Triton Kernels provide a speedup by describing the multiplication algorithm at a higher level and optimizing the compiled version for a specific GPU architecture without employing additional memory.

4 Experimental Setup

We benchmarked our SEBP method against the standard PyTorch (Original) and DeepSpeed baselines. Experiments were run on an NVIDIA A100 (40GB) GPU using BFloat16 precision. For the GLUE tasks with encoder models, hyperparameter optimization revealed optimal performance with a learning rate of 1.23×10^{-4} and keeping $N = 3456$ dense rows for SEBP.

For the DeepSpeed baseline, we utilized ZeRO Stage 2 to offload optimizer states to CPU RAM, enabling training of larger batches. It is important to note that DeepSpeed achieves its high throughput partly through aggressive hardware-aligned optimizations. Specifically, on NVIDIA Ampere architectures (A100), DeepSpeed leverages 2:4 Structured Sparsity (where every contiguous block of 4 values must contain at least 2 zeros) to minimize memory bandwidth usage.

We conducted experiments using a BERT-base model with a context length of 512 on the GLUE benchmark tasks, and a LLaMA 3.2 model with a context length of 4096 on language modelling datasets (WikitextMerity et al. (2017), PTBMarcus et al. (1993)), and a set of common sense reasoning tasks: ARC-CClark et al. (2018), ARC-EasyClark et al. (2018), HellaSwagZellers et al. (2019), and WinograndeSakaguchi et al. (2020). To analyze the acceleration achieved during training with a short-context setting (as shown in Tables 1 and 2), we trained the model for one epoch on a given dataset, assessed the backward pass time, memory usage, and the resulting quality improvement.

5 Results and Analysis

5.1 Acceleration and Memory Consumption

Table 3 and Table 4 report backward latency for BERT-base. The backward pass of linear layers is dominated by dense matrix multiplications for activation gradients and weight gradients. SEBP accelerates these computations by reducing the effective number of active rows in the output gradient (e.g., padding rows or rows below a magnitude threshold), while keeping the forward pass unchanged. DeepSpeed accelerates the same dense GEMMs without changing their nominal dimensions, primarily via system-level optimizations (e.g., contiguous buffers, kernel fusion, and fewer kernel launches), which is why it can achieve higher backward speedups in our measurements (up to $\sim3\times$).

Task	Orig Bwd (ms)	Mod Bwd (ms)	Bwd Speedup
sst2	47.550	22.907	2.08x
mrpc	47.609	23.028	2.07x
rte	47.624	22.960	2.07x
cola	47.552	22.977	2.07x

Table 3: Backward acceleration (SEBP vs. Original) for BERT-base model.

	Bwd Pass (ms)			Bwd Speedup	
Task	Orig	DS	SEBP	DS	SEBP
sst2	93.618	30.062	43.692	3.11x	2.14x
mrpc	93.684	30.501	43.690	3.07x	2.14x
cola	93.749	30.373	43.667	3.09x	2.15x
mnli	93.832	30.059	43.720	3.12x	2.15x

Table 4: Backward Pass Performance Analysis for SEBP and DeepSpeed Methods, BERT-base.

Table 5 presents a comparison of total training time for Llama-3.2 with a fixed batch size of 1. Under these conditions, DeepSpeed outperforms the original PyTorch baseline by 2.75x in total training time. However, this comes at a cost. As shown in Table 6, DeepSpeed's requirement to buffer offloaded states increases the total memory footprint.

	Total Training Time (min)		Speedup
Task	Original	DeepSpeed	DS
Winogrande	17.73	6.45	2.75x
ARC-E	0.99	0.36	2.75x
ARC-C	0.49	0.18	2.75x
HellaSwag	17.52	6.37	2.75x
WikiText-2	19.04	6.92	2.75x
PTB	18.47	6.72	2.75x

Table 5: Training time (minutes) comparison: Original PyTorch vs. DeepSpeed Llama-3.2-3B (times converted from seconds by dividing by 60).

SEBP Backward Pass Efficiency: Focusing on the gradient computation phase, Table 7 demonstrates that SEBP reduces the backward pass latency from 9.95 ms to 4.78 ms, achieving a max of 2.08x speedup. Crucially, Table 8 confirms that

Method	Wino	ARC-E	ARC-C	Hella	Wiki	PTB
Original (GB)	25.98	25.98	25.98	25.98	25.98	25.98
DeepSpeed (GB)	**31.55**	**31.55**	**31.55**	**31.55**	**31.55**	**31.55**

Table 6: Memory Usage (GB) comparison: PyTorch vs. DeepSpeed.

this speedup is achieved with negligible memory overhead (measured at ∼0.37 GB), preserving the VRAM availability of the original model.

	Backward Pass Latency (ms)		Speedup
Task	Original	SEBP	SEBP
Winogrande	9.95	4.78	2.08x
ARC-E	9.52	4.77	1.99x
ARC-C	9.53	4.78	1.99x
HellaSwag	9.52	4.79	1.99x
WikiText-2	9.52	4.79	1.99x
PTB	9.52	4.79	1.99x

Table 7: Backward pass latency (ms) comparison: Original vs. SEBP Llama-3.2-3B.

Method	Wino	ARC-E	ARC-C	Hella	Wiki	PTB
Original (GB)	25.98	25.98	25.98	25.98	25.98	25.98
SEBP (GB)	**26.35**	**26.35**	**26.35**	**26.35**	**26.35**	**26.35**

Table 8: Memory Usage (GB) comparison during Backward Pass: PyTorch vs. SEBP Llama-3.2-3B.

Figure 4 illustrates the trade-off between Speedup and Memory Usage for different backward pass methods. The plot positions SEBP and DeepSpeed relative to the standard PyTorch baseline in terms of Speedup (y-axis) and Memory Cost (x-axis).

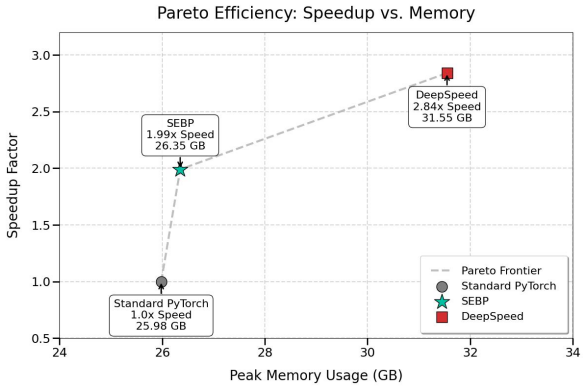


Figure 4: Comparison of backward pass speedup vs. Memory Footprint on Llama-3.2-3B between PyTorch, DeepSpeed and SEBP.

In Figure 4, the PyTorch baseline runs the standard dense backward pass and stores only the usual autograd intermediates. DeepSpeed improves throughput mainly by reorganizing the execution around these same dense GEMMs: it groups parameters/gradients into contiguous (flattened) buffers, uses larger buckets, and can apply fused optimizer/communication kernels. The underlying GEMM shapes are unchanged (the same forward and backward matrix multiplications are performed), but in our setup the surrounding execution introduces additional temporary and/or persistent buffers (e.g., gradient buckets/partitions, optimizer workspaces, and if offload is enabled GPU staging buffers for CPU-GPU transfers). These allocations increase peak memory and place DeepSpeed in the ”higher memory, higher speed” region.

SEBP targets a different axis by reducing the effective number of active rows during backward. By compacting the non-zero rows and computing only what is necessary, SEBP reduces the dominant backward compute while adding only small index and compaction buffers. This explains why SEBP stays close to the baseline in memory while still providing ∼2× backward speedup.

5.2 Impact on Model Quality

The small regression observed under DeepSpeed (Table 10) is explained by numerical and optimizer-path differences relative to the baseline. DeepSpeed commonly uses fused optimizer implementations and mixed precision update paths that are not bitwise identical to PyTorch AdamW. Differences in when gradients are cast/reduced, how moment estimates are accumulated (e.g., FP32 master weights vs mixed precision), and how weight decay is applied inside fused kernels can produce slightly different parameter updates even with the same hyperparameters. Over short fine-tuning runs, these small per-step deviations can accumulate into measurable differences on sensitive evaluation tasks. In contrast, SEBP keeps the optimizer path unchanged and modifies only the backward GEMMs by skipping rows with negligible signal, which is consistent with the near-identical convergence curves.

Dataset	Before FT	After FT	Delta
ARC-C	0.4599	0.4701	+0.010
ARC-E	0.7134	0.7243	+0.011
HellaSwag	0.6716	0.6800	+0.008
Winogrande	0.6827	0.6953	+0.013

Table 9: SEBP Validation: Quality metrics show consistent improvement after Fine-Tuning Llama-3.2-3B .

Dataset	Before FT	After FT	Delta
ARC-C	0.4599	0.4565	-0.003
ARC-E	0.7134	0.7054	-0.008
HellaSwag	0.6716	0.6762	+0.005
Winogrande	0.6827	0.6867	+0.004

Table 10: DeepSpeed Validation: Aggressive optimizations lead to slight quality degradation (negative delta) on ARC tasks Llama-3.2-3B.

5.3 Practical Applicability

Finally, the observed fine-tuning speedups show that SEBP enables efficient fine-tuning on small, isolated devices. The method achieves up to a 2× speedup in the backward pass without degrading model quality and with negligible memory overhead, in contrast to system-level approaches such as DeepSpeed.

This makes SEBP well suited for on-device fine-tuning of decoder-only LLMs, where models must rapidly adapt to user-specific data without access to cloud infrastructure or large GPUs. By exploiting structural gradient sparsity induced by short input sequences, SEBP enables fast and memory-efficient personalization on resource-constrained user devices.

5.4 Convergence Analysis

We evaluate SEBP not only by runtime, but also by its effect on training behavior and final quality. An acceleration method is only useful if it preserves convergence and downstream performance.

For LLaMA-3.2-3B, Figure 5 compares the Hugging Face baseline with SEBP. The curves are nearly identical, suggesting no noticeable change in convergence. Figure 6 shows the corresponding comparison between the baseline and DeepSpeed.

For encoder fine-tuning, Figure 7 (Appendix) reports training loss on six GLUE tasks, comparing the PyTorch baseline with SEBP. Across SST-2, RTE, MNLI, QNLI, MRPC, and QQP, SEBP closely tracks the baseline and reaches similar final loss values.

SEBP modifies only the backward computation by skipping rows that correspond to padding or have very small magnitude. The forward pass, loss, optimizer, and learning-rate schedule are unchanged, so rows with negligible signal have limited effect on parameter updates.

We vary the number of kept rows N and the threshold ϵ and observe stable training across a broad range. Once N covers the non-padding tokens, results largely saturate. The runtime benefit

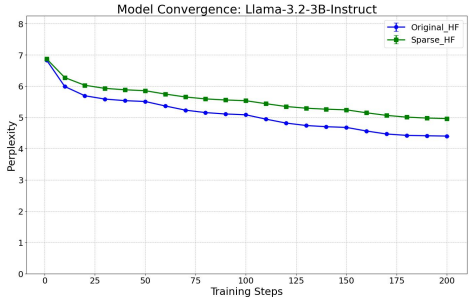


Figure 5: Original HF vs. SEBP (SparseHF).

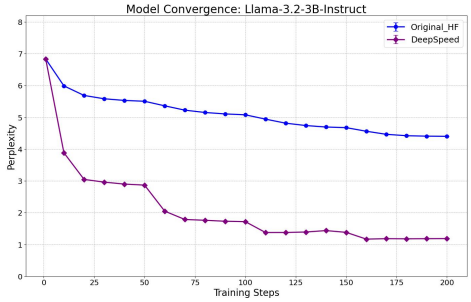


Figure 6: Original HF vs. DeepSpeed.

increases with the padding ratio, while convergence remains similar. Across random seeds, variability is comparable to the baseline and we do not observe SEBP-specific instabilities.

SEBP reduces backward-pass computation without requiring changes to the training setup, and the observed convergence behavior matches standard training within typical run-to-run variation.

6 Conclusion

This work has successfully demonstrated that exploiting padding-induced gradient sparsity is a viable strategy for accelerating transformer training. In particular, we formulated and empirically validated the hypothesis that for input samples whose length is short relative to the model context, the backward pass can be accelerated by leveraging the resulting structural sparsity in the gradients.

Our key contribution, the Sparsity-Exploiting Backward Pass (SEBP), delivers a significant backward pass speedup—2.15× for BERT and 1.99× for LLaMA-3.2-3B—while incurring minimal memory overhead.

The results highlight an important trade-off: while libraries such as DeepSpeed provide superior overall throughput, they achieve this at the cost of increased resource consumption. In contrast, SEBP offers an effective alternative for memory-constrained environments. Our validation confirms that this gradient approximation does not compromise model quality or convergence across both en-

coder and decoder architectures.

Finally, the observed fine-tuning speedups indicate that SEBP can be effectively applied to accelerate fine-tuning on small, isolated devices, enabling faster adaptation of LLM without sacrificing memory efficiency or model quality.

Limitations

SEBP reduces FLOPs - most notably in the backward pass, and is effective in many settings, but it has several limitations. First, its benefit depends on the presence of padded tokens, when sequences are long and padding is scarce -such as the wikitext task- there are fewer FLOPs to remove, so the acceleration shrinks or disappears. Second, our FLOPs-reduction configuration has been validated only for fine-tuning pre-trained models and is not intended for pre-training from scratch, because it relies on the knowledge already encoded in the weights. Third, we currently use a fixed, manually tuned Top-N; what works for one task or batch size may be suboptimal for another, suggesting that an adaptive mechanism is a promising direction for future work. Fourth, the reported speedups achieved by cutting FLOPs in custom Triton kernels were measured on an NVIDIA A100 and may not transfer directly to other hardware or software stacks.

Ethical Considerations

By reducing FLOPs-especially in the backward pass-SEBP lowers computation, memory traffic, and energy use (and thus the carbon footprint) of fine-tuning, making powerful models more accessible to researchers and organizations with limited hardware. At the same time, any method that reduces FLOPs and speeds up fine-tuning can have dual-use implications by lowering barriers for malicious actors to adapt models for harmful purposes, such as generating misinformation. The heuristic is content-agnostic: it operates on padding structure rather than textual content, so it is not expected to introduce new biases, though responsible downstream use remains essential.

Acknowledgements

The work of Alexander Panchenko was supported by the RSF project № 25-71-30008 “Laboratory for reliable, adaptive, and trustworthy Artificial Intelligence”.

References

Hirad Amini, Md JUEAL Mia, Yalda Saadati, Ahmed Imteaj, Seyedali Nabavirazavi, Urmish Thakker, M Zakir Hossain, A A Fime, and S S Iyengar. 2025. Distributed LLMs and multimodal large language models: A survey on advances, challenges, and future directions. *arXiv preprint arXiv:2503.16585*.

Viktoriia A. Chekalina, Anna Rudenko, Gleb Mezentsev, Aleksandr Mikhalev, Alexander Panchenko, and Ivan Oseledets. 2024. SparseGrad: A selective method for efficient fine-tuning of MLP layers. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14929–14939.

Hongrong Cheng, Miao Zhang, and Javen Qinfeng Shi. 2024. A survey on deep neural network pruning: Taxonomy, comparison, analysis, and recommendations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):10558–10578.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try ARC, the AI2 reasoning challenge. Dataset (ARC-Easy/ARC-Challenge): https://huggingface.co/datasets/allenai/ai2_arc.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186. ArXiv: https://arxiv.org/abs/1810.04805.

Utku Evci, Trevor Gale, Jacob Menick, Pablo Samuel Castro, and Erich Elsen. 2020. Rigging the lottery: Making all tickets winners. In *Proceedings of the 37th International Conference on Machine Learning*, pages 2901–2911. PMLR. ArXiv: https://arxiv.org/abs/1911.11134.

Andreas Köpf and 1 others. 2023. Openassistant conversations — democratizing large language model alignment. OASST1 dataset: https://huggingface.co/datasets/OpenAssistant/oasst1.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of english: The Penn treebank. *Computational Linguistics*, 19(2):313–330. Corpus distribution (LDC): https://catalog.ldc.upenn.edu/LDC99T42.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer sentinel mixture models. In *International Conference on Learning Representations*. Introduces WikiText-2/WikiText-103. OpenReview PDF: https://openreview.net/pdf?id=Byj72udxe. Dataset: https://huggingface.co/datasets/Salesforce/wikitext.

Meta AI. 2024. Llama 3 model card. https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3505–3506. Project: https://www.deepspeed.ai/. Code: https://github.com/deepspeedai/DeepSpeed.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2020. WinoGrande: An adversarial Winograd schema challenge at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 8732–8734. ArXiv: https://arxiv.org/abs/1907.10641. Dataset: https://huggingface.co/datasets/allenai/winogrande.

Kevin Tian, L Qiao, B Liu, G Jiang, and D Li. 2025. A survey on memory-efficient large-scale model training in AI for science. *arXiv preprint arXiv:2501.11847*.

Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 28–39.

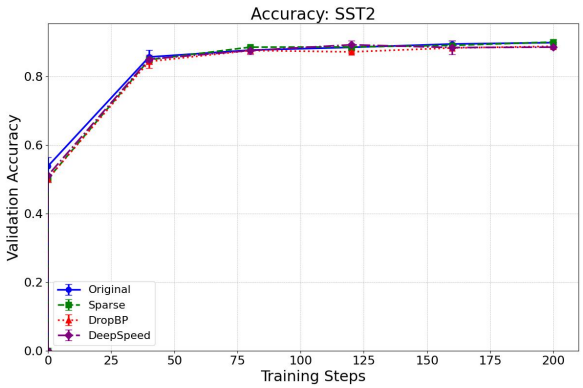
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. ArXiv: https://arxiv.org/abs/1706.03762.

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. In *International Conference on Learning Representations*. OpenReview PDF: https://openreview.net/pdf?id=rJ4km2R5t7. arXiv: https://arxiv.org/abs/1804.07461. Dataset: https://huggingface.co/datasets/nyu-mll/glue.

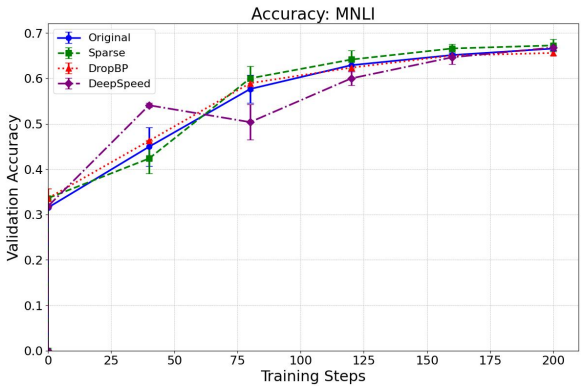
Sunghyeon Woo, Baesung Park, Byeongwook Kim, Minjung Jo, Se Jung Kwon, Dongsuk Jeon, and Dongsoo Lee. 2024. DropBP: Accelerating fine-tuning of large language models by dropping backward propagation. In *Advances in Neural Information Processing Systems*, volume 37. Code: https://github.com/WooSunghyeon/dropbp. arXiv: https://arxiv.org/abs/2402.17812.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. HellaSwag: Can a machine really finish your sentence? Dataset: https://huggingface.co/datasets/Rowan/hellaswag.

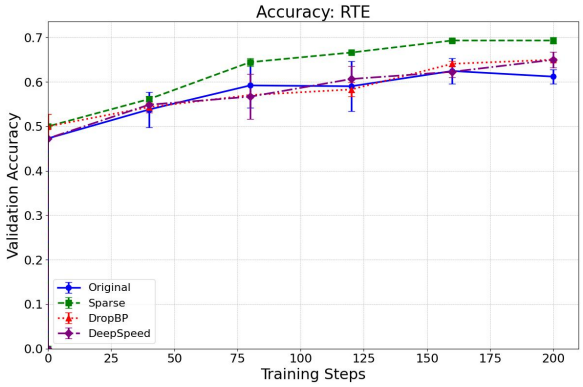
Appendix A



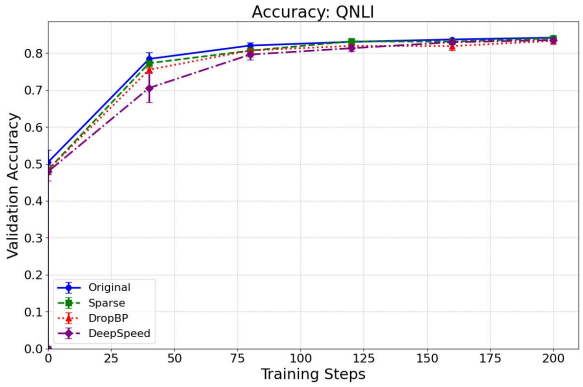
(a) SST-2



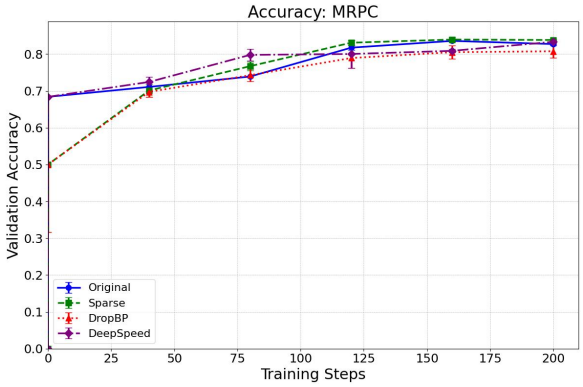
(b) MNLI



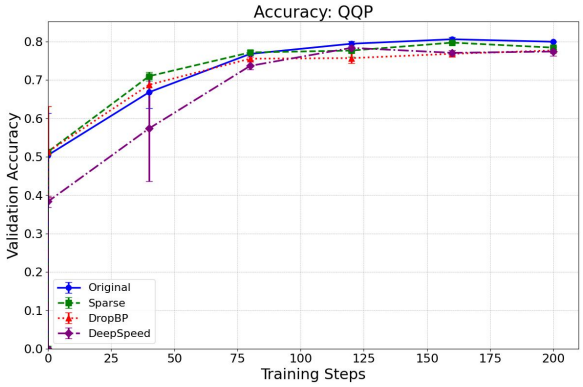
(c) RTE



(d) QNLI



(e) MRPC



(f) QQP

Figure 7: Convergence on six GLUE tasks (BERT-base). Baseline (blue) vs. SEBP (orange); curves nearly overlap.

Appendix B

Dataset	WikiText	STSB	CoLA	WNLI	MRPC	RTE	QQP	SST2	MNLI
	#Tokens per sample								
AVG	512	27	47	37	53	66	30	13	39
Max	512	125	11	108	103	128	128	66	128
Min	512	10	4	16	19	13	6	3	5
	#Sparsity per layer intermediate								
AVG	0.16	0.79	0.92	0.75	0.60	0.48	0.80	0.88	0.70
Max	0.19	0.79	0.99	0.99	0.99	0.99	0.99	0.99	0.99
Min	0.07	0.77	0.92	0.72	0.56	0.43	0.78	0.87	0.67
	#Sparsity per layer output								
AVG	0.11	0.81	0.93	0.77	0.64	0.53	0.82	0.89	0.73
Max	0.15	0.81	0.99	0.99	0.99	0.99	0.99	0.99	0.99
Min	0.08	0.79	0.92	0.75	0.61	0.49	0.80	0.88	0.71

Table 11: Average sparsity of the gradient output $\frac{\partial L}{\partial \tilde{Y}}^T$ in RoBERTa's linear layers across datasets of varying lengths.

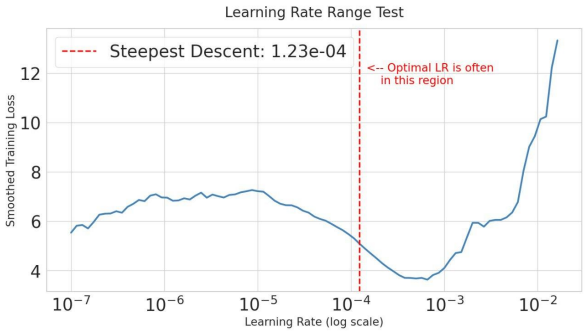


Table 12: Learning rate range test for Llama-3.2-3B. The point of steepest descent is marked.

Task	Orig Bwd (ms)	Mod Bwd (ms)	Bwd Speedup
sst2	47.779	23.148	**2.06x**
mrpc	47.935	23.049	**2.08x**
rte	47.933	23.070	**2.08x**
cola	47.842	22.985	**2.08x**

Table 13: Backward acceleration (SEBP vs. Original) RoBERTa-base model