Tempo: Accelerating Transformer-Based Model Training through Memory Footprint Reduction

Anonymous Author(s)
Affiliation
Address
email

Abstract

Training deep learning models can be computationally expensive. Prior works have shown that increasing the batch size can potentially lead to better overall throughput. However, the batch size is frequently limited by the accelerator memory capacity due to the activations/feature maps stored for the training backward pass, as larger batch sizes require larger feature maps to be stored. Transformer-based models, which have recently seen a surge in popularity due to their good performance and applicability to a variety of tasks, have a similar problem. To remedy this issue, we propose *Tempo*, a new approach to efficiently use accelerator (e.g., GPU) memory resources for training Transformer-based models. Our approach provides drop-in replacements for the GELU, LayerNorm, and Attention layers, reducing the memory usage and ultimately leading to more efficient training. We implement *Tempo* and evaluate the throughput, memory usage, and accuracy/loss on the BERT$_{LARGE}$ pre-training task. We demonstrate that *Tempo* enables up to 2$\times$ higher batch sizes and 16% higher training throughput over the state-of-the-art baseline. We also evaluate *Tempo* on GPT2 and RoBERTa models, showing 19% and 26% speedup over the baseline.

1 Introduction

Transformer-based models such as BERT [12] and GPT-2 [49] have found success in numerous general natural language processing tasks including question answering [51], paraphrasing [13], natural language inference [68], and even areas outside language tasks such as image recognition [14]. However, training such models can be highly expensive in terms of time, monetary resources and carbon footprint [24, 60]. For instance, the pre-training of BERT$_{LARGE}$ takes 4 days to complete on 16 Cloud TPUs (64 TPU chips total) [12], which costs about $10,000 [56]. Training a more recent Transformer-based model, GPT-3, has an even more astonishing price tag - $12 million[66]. Hence, even a small decrease in the end-to-end training time of Transformer-based models matters.

Although there has been significant progress made in accelerating Transformers using specialized hardware (e.g., Google TPUs [30], NVIDIA Tensor Cores [39]) in the past few years, a fundamental issue with Transformer-based models is that they are limited by the memory capacity of hardware accelerators. For example, even a batch size of 1 does not fit into a modern GPU with 12GB of memory when training BERT with sequence length 512 [15]. Reducing memory footprint [48, 8, 52] is a viable option to allow larger batch training, leading to better hardware utilization and ultimately improved training throughput [73].

Many existing approaches to memory footprint reduction (e.g., offloading [52, 65, 48], checkpointing [8, 73, 33, 28], and data compression/encoding [26, 6]) either have high computational overhead or do not apply to Transformer-based models directly. Prior approaches fall into two main categories, neither of which are satisfactory for the Transformer-based model case. First, these techniques may

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

be too general [48, 6, 50, 33, 28] to utilize the specifics of Transformer-based models well, such as the multi-headed attention mechanism used in Transformers [63], or optimization opportunities available in specific layers such as the LayerNorm [4] layer. For example, although checkpointing [8, 28] can significantly enlarge batch size, it also brings high overhead (e.g., 30% performance degradation observed in some prior works [8]). Second, if prior works are specific, they focus on other types of models/layers with ideas not being applicable to Transformers. For example, Gist and In-Place ABN deal with CNNs [26, 53].

Unlike prior works which do not leverage the specifics of Transformer-based models, we propose a new approach specifically tailored for Transformer-based models, called *Tempo*. We achieve this using three new techniques: (i) In-place GELU, (ii) In-place LayerNorm, and (iii) Sub-Layer Dropout Recomputation. In-place GELU and In-place LayerNorm both use alternative derivations for the computation of the backward passes of these layers. These derivations allow some activations that are normally retained during the forward pass (to be later used in the backward pass) to be discarded, leading to a more memory-efficient implementation. Sub-Layer Dropout Recomputation discards activations within the high memory footprint attention mechanism during the forward pass, then recomputes these during the backward pass without recomputing extra unnecessary tensors. *Tempo* is able to increase the training throughput with larger batch sizes by reducing the total memory footprint of the models during training. To our best knowledge, this is the first work to explore memory footprint optimizations specifically for Transformer-based layers that show not just footprint reduction, but the actual increase in throughput using the extra memory savings. Tempo reduces the memory footprint of training Transformer-based models by targeting a major part of the total footprint – the *activation memory* [74] (the saved feature maps during the forward pass of the model that are required for backpropagation [54]). All the proposed techniques provide a large memory footprint reduction with very low throughput degradation (as low as 1%). Our results show up to $2\times$ improvement in batch size for BERT$_{LARGE}$ pre-training at a sequence length of 512 on modern GPUs while increasing the training throughput by up to 16% .

2 Background and Motivation

2.1 Memory Footprint of BERT

BERT [12] is a popular natural language processing model that is based on the Transformer architecture [63]. The model has been successfully applied to a variety of tasks such as question answering (SQuAD [51]), paraphrasing (MRPC [13]), natural language inference (MNLI [68]), and others [57, 72] through a two step training process. The training process entails first training on a general unlabelled data set (*pre-training*) [12]. The faster second part of the training process (*fine-tuning*) takes the parameter weights produced by the pre-training section and further trains on a downstream task such as question answering [51] or sentiment analysis [57] which it accomplishes through the addition of a specialized output layer [12].

The BERT architecture allows for multiple different configurations depending on model hyperparameters selected, some being derived from the original Transformers paper; these include the hidden layer size (H), sequence length (S), number of attention heads (A) and number of layers (L).

In the context of this work, we point out some of the relevant parts of the model and their activation memory footprint with respect to these hyperparameters referring to Figure 1.

① At this point, where attention [63] is calculated we observe that the size of each of the feature maps goes as $\mathcal{O}(S^2)-$ there are a variety of previous techniques and models that have been explored in the literature to deal with this problem [61]. Additionally, at this point note that we store three feature maps of size $[B \times A \times S^2]$. Calculations based on Figure 1 at the BERT$_{BASE}$ parameters show that at a sequence length of 512 these three feature maps **account for** 56% **of the encoder layer activation memory.**

② At these two points, we store the input to the two LayerNorm layers of size $[B \times S \times H]$

③ Here a GELU [21] layer is used as the activation function for the preceding fully-connected layer of size $[B \times S \times 4H]$. The activation memory for this function stores **almost** 17% **of the total layer activation memory of** BERT$_{BASE}$ at a sequence length of 128.

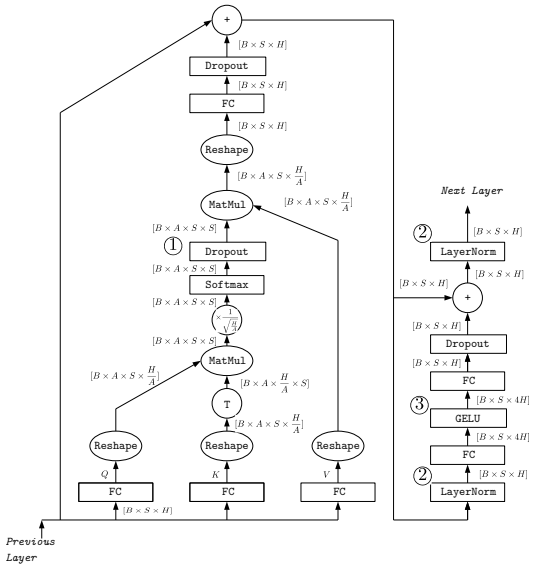


Figure 1: A diagram of a single Transformer encoder [63] layer used in BERT [12]. This is based on the Huggingface implementation of BERT [69]. As in the BERT paper, A represents the number of attention heads, and H represents the hidden size. We represent the batch size by B and the sequence length by S.

2.2 Why Activation Memory Matters

As iterated in previous works [52, 26, 48, 73, 6] there are multiple benefits to reducing the memory footprint of models. First, it allows for larger models which can positively affect the model's performance on downstream tasks [12]. Second, memory footprint reduction can allow for a larger batch size. This, in turn, could lead to better utilization of the GPU hardware [17], increasing the overall throughput [73]. In order to verify this possibility for Transformer-based models, we conduct our own experiments using Huggingface's BERT implementation [69] to train BERT$_{LARGE}$ on the MRPC [13] fine-tuning task. Figure 2 shows the throughput on this task for sequence lengths of 128 and 512. From the figure, we conclude that there is a steady improvement in batch size when the sequence length is 128. This is also the case when the sequence length is 512, however, in this situation the trend ends more abruptly as the memory consumption of the model exceeds the GPU memory capacity, showing a clear opportunity to take advantage of memory footprint reduction.

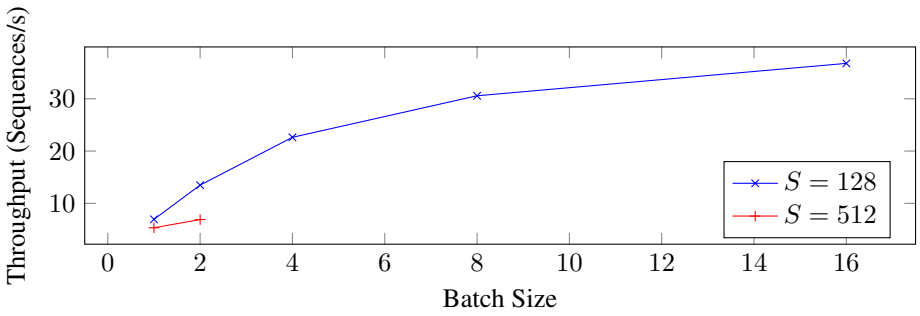


Figure 2: Plots of throughput (sequences/s) vs batch size for BERT$_{LARGE}$ [12] fine-tuning on the MRPC [13] task at sequence lengths 128 and 512 on four 2080Ti [40] GPUs. The maximum batch sizes are respectively 16 and 2.

We note that previous works on Transformer-based models show that although the model parameters contribute to the memory footprint, the main memory capacity consumer during training is actually the activation feature maps [74, 28, 8, 48, 26, 33, 6]. In addition, the majority of this activation memory will be used in each of the BERT Transformer encoder layers. Profiling the Huggingface

BERT$_{BASE}$ implementation [69] on the MRPC [13] fine-tuning task at a batch size of 32 and sequence length of 128 shows that 66% of the total memory is taken up by these encoder activations. More details on this are shown in Appendix A.

2.3 Key Prior Works

There are three major prior techniques used in training memory footprint reduction of deep learning models. The first of these is *Checkpointing* [8, 33, 28, 73]. This technique involves discarding certain feature maps in the forward pass while retaining others. Later, in the backward pass, these discarded feature maps may be recomputed from the retained feature maps, and thus used in the computation of the gradients. The second technique is *Offloading* [48, 52, 65]. In this case, the main idea involves taking feature maps that would be stored in the GPU memory, and instead offloading them to the CPU memory. These techniques can also involve *pre-fetching* tensors from the CPU memory in anticipation of their use. Offloading suffers from a dependence on system variables such as the communication channel bandwidth [52, 48]. It also requires extensive engineering effort to avoid high overhead [6]. Finally, *Compression/encoding*; this can be divided into two different categories, lossless and lossy [26, 6]. However, the fundamental idea is to compress, or reduce the space taken up by feature maps in the forward pass, then decompress it for use in the backward pass.

These techniques are usually largely orthogonal to one another as was shown in prior works where both offloading and checkpointing are used simultaneously [48, 65]. We expand on these techniques in Appendix C.

2.4 Why Tempo?

Although the techniques in the previous section show good performance on a variety of models, they suffer from a variety of issues. Checkpointing's scope is often too broad to consider certain layer-specific optimizations and alternative derivations that can provide lower overhead [53]. Furthermore, overhead can be high (as much as 30%) [8]. Offloading can be system- dependent and requires significant engineering effort, while compression can be lossy or not applicable to the Transformer case. Hence, there is a clear need for a deeper look at activation memory optimizations for Transformer-based neural networks in particular. To our best knowledge, our work is the first to explore such optimizations tuned to improving the throughput of Transformer-based models. Table 1 shows a summary comparison of Tempo and various other techniques, with the major points that differentiates our technique from prior work.

Feature	Capuchin	Checkmate	ActNN	Gist	**Tempo**
Layer-Specific	✗	✗	✗	✓	✓
Transformer-Specific	✗	✗	✗	✗	✓
Lossless	✓	✓	✗	∼[1]	∼[2]
Drop-In Layer Replacement	✗	✗	✓	✓	✓
Online	✓	✗	✓	✓	✓

Table 1: Comparison between Tempo and Capuchin [48], Checkmate [28], ActNN [6], and Gist [26].

3 Tempo: Key Ideas

We now present the major ideas that lays behind the design of Tempo: (1) **In-place GELU**, (2) **In-place LayerNorm**, and (3) **Sub-Layer Dropout Recomputation**. The major theme behind all of these ideas is to compute the backward pass as normal, while *using less storage* to do so. To this end, In-place GELU and In-place LayerNorm compute the output of each layer *in-place*; instead using the output activation to compute the gradient. Sub-Layer Dropout Recomputation also discards the output, and through a closer look at the structure of the Dropout layer is able to recompute the output without excessive recomputation. We strongly suggest reading Appendix E for the implementation

[1] Some of the Gist [26] optimizations are lossy.

[2] Accuracy of our lossy optimization is tunable, offering a flexible tradeoff between the accuracy and the hardware cost.

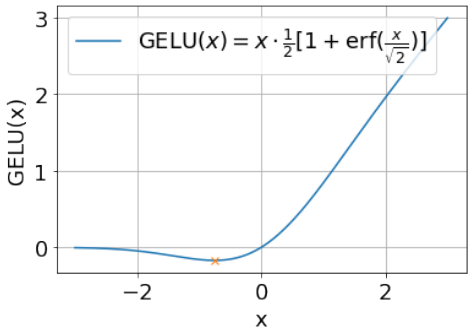
details. We also add in this appendix a new optimization of softmax that we use that further reduces memory [18].

3.1 In-place GELU

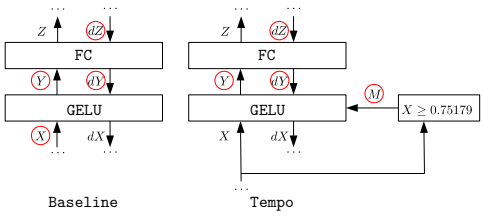
The GELU layer is used as an activation function for the feed-forward section of the BERT layer (③ in Figure 1) [12]. A plot of this function is shown in Figure 3a. Referring to the baseline in Figure 3b, note that both X and Y are stored for the backward pass. Y is needed for the downstream fully connected layer, while X is stored for the GELU layer itself [46]. Prior work has demonstrated that certain activation functions such as ReLU may be computed in-place [26]. This can be done without affecting the calculation of the backward pass. If we were able to compute the GELU function in-place, potentially by recovering the input from the output on the backward pass, we could save the storage required for X. However, this is impossible to do directly. A key observation to make with respect to the GELU function is that it is not bijective – hence there is no function that will be able to compute the input from the output without additional information.

However, we observe that the GELU function is both continuous and has only one extremum, a minimum value at $x \approx -0.75179$ as can be seen in Figure 3a. Notably, this implies that just one additional piece of information: which **side of the minimum** the input originates from, allows us to compute the inverse of the GELU. This is because on each side of the minimum the function is one-to-one, and hence the input is recoverable from the output in *each section*. Based on this key observation, we can discard the input, and simply retain the output of the GELU, as well as the additional information on whether the input is greater than or equal to the value at which the minimum occurs. Figure 3b illustrates the difference between our method and the baseline.

In order to execute this efficiently on a real system, we note that the original derivative in terms of the input can be composed with the function inverse in order to create a composite kernel. This kernel consists of a polynomial approximation of this composite function, the approximation being necessary since GELU is transcendental, and therefore the inverse cannot be solved in terms of elementary functions [58]. Further details are discussed in Appendix E.



(a) A plot of the GELU[21] function near the origin, along with the marked minimum point.



(b) Saved feature maps between the baseline and Tempo. Note that our method only saves a 8-bit mask[3] that denotes whether the input is greater or less than the minimum value, instead of the full 32-bit input feature map.

3.2 In-place LayerNorm

The LayerNorm layer is used at multiple points in the Transformer encoder layer [63], which we denote by ② in Figure 1. Usually, the gradient computation of LayerNorm relies on the gradient input from the next layer, as well as the input feature map which is stashed for this computation [46].

Similar to GELU, we are able to derive an expression for the gradient of the LayerNorm layer *as a function of its output*. In this context, the output of LayerNorm must be stashed to compute the gradient of the successive fully connected layer anyways. Using this approach, the memory footprint overhead of LayerNorm is just the intermediate mean and variance computed in the forward pass. The full derivation is presented in Appendix E.

[3] Pytorch boolean masks use 8-bits per value [46]. Masks can also be implemented as 1-bit manually but this brings extra overhead due to unpacking and packing bit tensors.

Comparison with Checkpointing: Note that although In-place GELU requires more memory compared to recomputing Y from X, it will have increased overhead due to the recomputation. Additionally, our technique is orthogonal to conventional Checkpointing, as it could take advantage of the fact that no recomputation is required for the input X for both In-place GELU and In-place LayerNorm.

3.3 Sub-Layer Dropout Recomputation

In this section we explore the idea of sub-layer granularity checkpointing, or partial recomputation applied to the Dropout layer [59] found in ① in Figure 1. The function of a dropout layer is to set the output of $p\%$ of entries in the incoming feature map to zero ("drop" the outputs) and then scale the remaining outputs by the factor $\frac{1}{1-p}$, which makes the network less sensitive to any output of the preceding feature map, thereby making it more robust [59].

We define sub-layer recomputation as a technique where recomputation of only *some* of the feature maps is necessary for the backward pass that may be produced by a given layer's output. We observe that better recomputation strategies are possible if we carefully deconstruct layers in the case where they store multiple outputs. This observation can be directly applied to the Dropout layer. In the computation of Dropout, both a mask (which records the entries which are set to zero in implementations of Dropout [46, 7]) and output are produced. If a layer-based checkpointing implementation [16] was used, it would cause both the mask and the output to be recomputed in the backward pass if the layer is checkpointed, thus requiring higher overhead. However, we notice that nothing precludes us from simply doing only one of these recomputations. Storing the mask would only reduce the recomputation time, while the fact that the mask itself only has Boolean values allows us to keep most of the memory benefit of recomputation. In this way, we can save the storage required for the output at the critical $\mathcal{O}(S^2)$ Attention section (③ in Figure 1) for the cost of a simple mask multiply. This technique is illustrated in Figure 4.

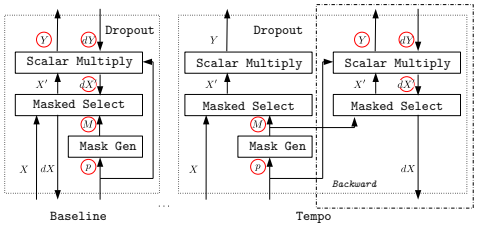


Figure 4: Comparison of dropout implementation between the baseline, and our method. Note that we only save the mask, and recompute the other output. The representation on the left is not an exact copy of the PyTorch implementation, rather it is an illustrative representation.

4 Evaluation

4.1 Methodology

Infrastructure Our main test setup consists of 4 NVIDIA RTX 2080 Ti GPUs [40], each with 11 GB of memory connected over PCIe v3 [47]. We also use an Amazon Web Services p3.8xlarge [3] instance consisting of 4 NVIDIA Tesla V100 GPUs [39] each with 16 GB of memory connected using NVLink [42]. For our ablation studies, we employ a system with an NVIDIA A100 GPU [43] with 40 GB of memory. We summarize the detailed setup in Appendix G.

Applications We evaluate our work using both the BERT pre-training and fine-tuning tasks [12]. For pre-training, we employ the NVIDIA DeepLearningExamples library [41] with the English Wikipedia dataset [67]. We perform the training in two phases, the first (i.e., longer) phase at a sequence length of 128, and the second (i.e., shorter) phase at a sequence length of 512 [12, 41]. For throughput and memory experiments, we use the BERT$_{LARGE}$ configuration. For our fine-tuning task, we use the MRPC [13] paraphrasing task on BERT$_{BASE}$ using the Huggingface library [69].

For our ablation studies, we also train both RoBERTa [34] and GPT2[49]. For the evaluation of RoBERTa, we use the Fairseq library [44], while GPT2 uses the Huggingface GPT2 model [69]. Both of these models use the WikiText Dataset for evaluation [35].

Technique	Sequence Length	Batch Size	Technique	Sequence Length	Batch Size
Baseline	128	15	Baseline	128	28
Baseline	512	1	Baseline	512	4
Checkpoint	128	50	Checkpoint	128	96
Checkpoint	512	4	Checkpoint	512	18
Tempo	128	24	Tempo	128	41
Tempo	512	2	Tempo	512	7

Table 2: The maximum batch size on both 2080 Ti (left) and V100 (right).

Metrics The first metric we focus on is the total *memory footprint* of our method compared to the baselines. There are two ways to look at this metric. First, we compare the maximum batch size possible for each method. We compare this across sequence lengths of 128 and 512 [5] on BERT$_{LARGE}$ for both 2080Ti and V100 GPUs. Second, we compare the total memory used by PyTorch at a given commonly used batch size for the same parameters. The second metric we use is the *throughput* for which we count the total number of sequences per second processed. Finally, we provide a comparison between our method and the baseline method on BERT$_{BASE}$ pre-training in order to compare the loss curves and show the change due to our lossy optimizations. We also provide fine-tuning curves on the MRPC [13] task, training for 10 epochs to ensure no significant accuracy deviations.

Our ablation studies only use the throughput metric.

4.2 Results

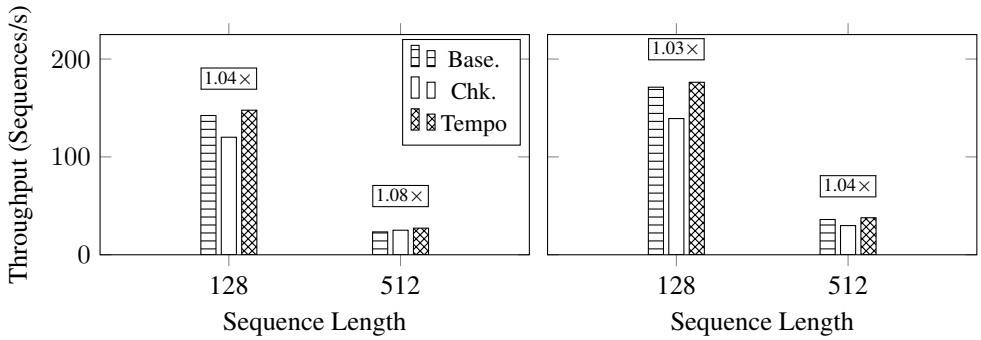
We use two major baselines. The first baseline is the NVIDIA BERT$_{LARGE}$ model [41], with no memory footprint techniques applied which we refer to as the *Baseline*. The second one is the same model, with the default checkpointing applied, based on the PyTorch implementation, applied at the input of each Transformer encoder layer [46, 41] and is similar to the Huggingface implementation [69]. We refer to this baseline as *Checkpoint*. We refer to our method that uses In-Place GELU, In-Place LayerNorm, Sub-Layer Dropout Recomputation, and the softmax engineering optimization as *Tempo*.

Impact on Memory Footprint Table 2 shows the maximum batch size and memory consumed at a fixed batch size for all three methods. Additionally, the total memory used at a batch size of 15 at a sequence length of 128 is 11.3 GB, 8.3 GB and 9.2 GB respectively for *Baseline*, *Checkpoint*, and *Tempo*. From this, we conclude that *Checkpoint* reduces the memory footprint to a much higher degree than both *Baseline* and *Tempo*. This is expected, as *Checkpoint* discards most of the feature maps to be recomputed [69, 41] no matter the performance cost. *Tempo* still provides a significant increase in batch size over *Baseline* at the sequence length of 512 – **we see 2× and 1.5× larger batches over *Baseline*** for the 2080 Ti and V100 respectively but, as the next section shows, with much better throughput.

Impact on Throughput Figure 5 illustrates our main results with respect to throughput. From the figure, we can see that *Tempo* outperforms both *Checkpoint* and *Baseline* across **both sequence lengths** and **across different hardware setups**. We observe an improvement of **16%** over *Baseline* on the 2080 Ti at a sequence length of 512. At these settings, we also have an improvement of 8% over *Checkpoint*. We also observe up to **27%** over *Checkpoint* on the V100 at a sequence length of 512, which also corresponds to a 5% improvement over *Baseline*. This is despite the fact that *Checkpoint* uses the largest batch size as per Table 2. This is because *Checkpoint* stores feature maps at the beginning of each Transformer encoder layer, and recomputes these layers [69, 41]. Hence, an increased batch size also means more recomputation. In contrast, *Tempo* is able to decrease the total memory footprint, and then convert this decrease into a substantial performance improvement over the *Baseline* due to the use of only low overhead mechanisms.

Impact on Loss and Accuracy We pre-train BERT$_{BASE}$ to ensure that our model's loss curve is not affected by approximate optimizations (e.g., In-Place GELU). Figure 6a shows the loss curve of phase 1 of BERT$_{BASE}$ pre-training [12]. We observe almost complete overlap in the loss curves

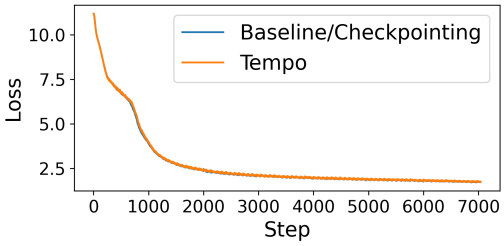
[5] These are the sequence lengths of Phase 1 and Phase 2 of pre-training [12, 41].



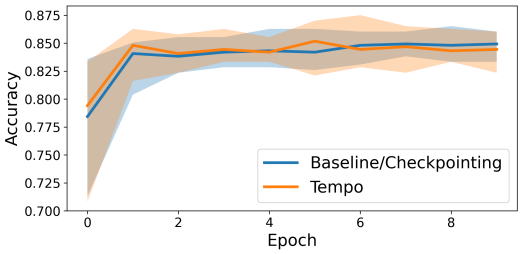
(a) 2080 Ti (b) V100

Figure 5: Throughput experiments at the maximum batch size annotated with the speedup *over the best baseline*.

with no more than a *0.5%* difference between *Tempo* and the baseline at the endpoint. We conclude that within that margin of error our method is satisfactory.



(a) Phase 1 BERT$_{BASE}$ pre-training curve on the English Wikipedia dataset [67].



(b) Accuracy of BERT$_{BASE}$ fine-tuning [12] on the MRPC [13] task. We run 10 trials of 10 epochs. The solid line represents the median accuracy of these trials, and the maximum and minimum along the training process by the transparent curves' boundaries.

For the fine-tuning accuracy, we use the pre-trained Huggingface [69] implementation. Figure 6b shows the results of BERT$_{LARGE}$ fine-tuning [12] on the MRPC [13] task. The figure shows a consistent overlap between the maximum and minimum accuracy of *Tempo* and the baseline, so we can conclude that *Tempo* has little impact on the accuracy of the trained model.

4.3 Ablation Studies

Ablation Study With Respect to Larger Model Parameters on Modern Hardware Platforms We also evaluate on other hardware platforms as well as model parameters. First, we use an increased hidden layer size for various configurations. These experiments are conducted on a platform with an NVIDIA A100 GPU [43] across sequence lengths of 128 and 512. We maintain the hidden layer size H to the number of attention heads A ratio of 64 which is in line with prior works [63, 12]. The results are shown in Figure 7. The figure demonstrates two important generalizations of *Tempo*. First, note that even on newer and more advanced GPUs, *Tempo* continues to provide a tangible benefit. Second, across larger hidden layer sizes *Tempo* consistently demonstrates a clear improvement over the baseline (as shown in the figure, this can be as high as a **39% speedup** over *Baseline* which corresponds to a 16% speedup over *Checkpoint*). The speedup over *Checkpoint* is as high as **20%** . We conclude that *Tempo* will continue to be applicable to new hardware and larger models.

We also conduct experiments on BERT$_{LARGE}$ (modified to use 12 Layers instead of 24 for more data points) for sequence lengths larger than 512. Figure 8 shows the results for this experiment, where we demonstrate that *Tempo* outperforms *Baseline* on longer sequence lengths as well which can be as high as a **27% speedup** over *Baseline*. At the same settings, we observe 16% speedup over *Checkpoint*. Tempo also outperforms *Checkpoint* by as much as **20%**. We conclude that yet

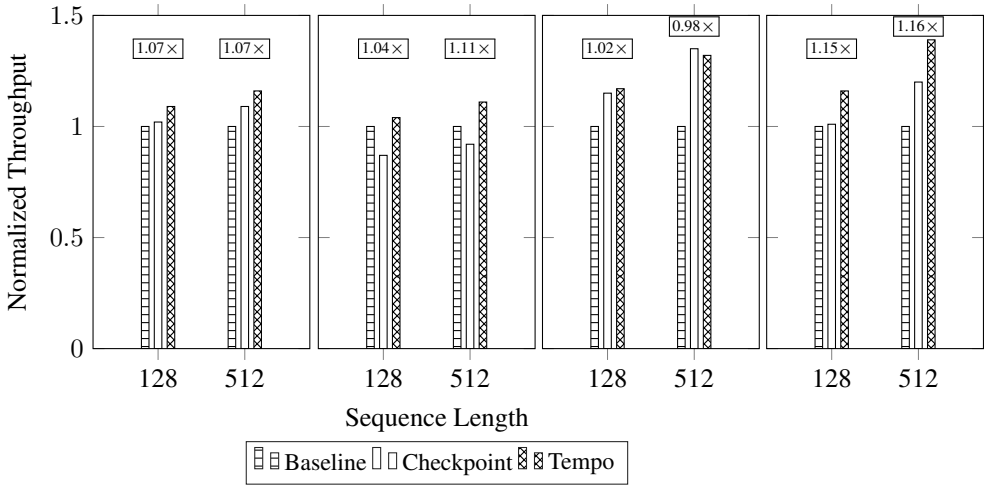


Figure 7: Normalized throughput at the maximum batch size, with annotated speedup *over the best baseline*. From left to right the configurations are (a) BERT$_{LARGE}$ ($H = 1024$), (b) BERT$_{BASE}$ $H = 2048$, (c) BERT$_{LARGE}$ $H = 2048$, (d) BERT$_{BASE}$ $H = 3072$

again *Tempo* will be able to take advantage of modern hardware, as well as remain advantageous as sequence lengths increase. Note that the largest sequence length of 3072 on *Baseline* does not have enough memory to run.

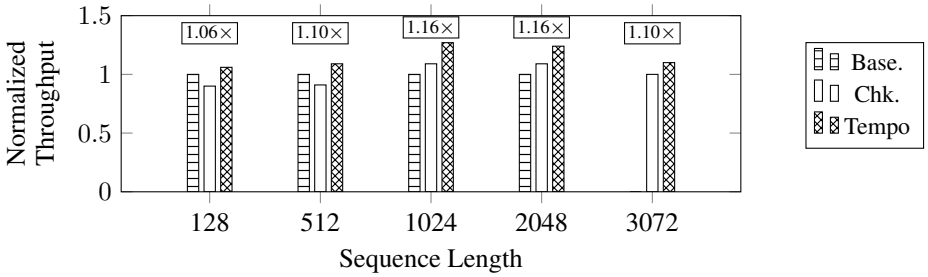


Figure 8: Normalized throughput relative to the Baseline across different sequence lengths on the NVIDIA A100 GPU for BERT$_{LARGE}$ modified to use 12 layers. We annotate each bar group with *Tempo's* speedup *over the best baseline*.

Results on Other Models We conduct experiments on other Transformer-based models as well: RoBERTa [34] and GPT2 [49]. For the evaluation of RoBERTa, we use the Fairseq library [44] as well as a sequence length of 512, while GPT2 uses the Huggingface GPT2 model [69]. These experiments are conducted on both 2080 Ti and V100 setups. We note that the improvement over the baseline is substantial (up to **19% and 26%** for GPT2 and RoBERTa respectively on the 2080 Ti setup. This improvement corresponds to a increase in batch size of **3**× and **2**×. Furthermore, we also see speedups of 5% and 4% on the V100 setup as well. From these results, we conclude that *Tempo* generalizes well to other Transformer-based models besides BERT.

5 Conclusion

We propose *Tempo*, a new mechanism that reduces the memory footprint of Transformer-based models at low cost. It shows an improvement in throughput of up to 16% over the state-of-the-art baseline for BERT$_{LARGE}$ pre-training task and also shows an improvement in maximum batch size of up to 2× on both V100 and 2080 Ti GPUs. Our technique also generalizes well to new models, more modern hardware, as well as diverse model parameters in terms of memory footprint and throughput, demonstrating the robustness of our technique. Our hope is that *Tempo* will be used with other footprint reduction methods to improve training efficiency of Transformer-based models.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2016)*. USENIX Association, 2016. `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi`.

[2] Jorge Albericio, Patrick Judd, Tayler H. Hetherington, Tor M. Aamodt, Natalie D. Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA 2016)*, 2016. `https://doi.org/10.1109/ISCA.2016.11`.

[3] AWS. Amazon EC2 P3 Instance Product Details, 2019. `https://aws.amazon.com/ec2/instance-types/p3`.

[4] Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer Normalization. *CoRR*, 2016. `http://arxiv.org/abs/1607.06450`.

[5] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer. *CoRR*, abs/2004.05150, 2020. `https://arxiv.org/abs/2004.05150`.

[6] Jianfei Chen, Lianmin Zheng, Zhewei Yao, Dequan Wang, Ion Stoica, Michael W Mahoney, and Joseph E Gonzalez. ActNN: Reducing Training Memory Footprint via 2-Bit Activation Compressed Training. In *International Conference on Machine Learning (ICML)*, 2021.

[7] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, 2015. `http://arxiv.org/abs/1512.01274`.

[8] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training Deep Nets with Sublinear Memory Cost. *CoRR*, 2016. `http://arxiv.org/abs/1604.06174`.

[9] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems, (ASPLOS 2014)*. ACM, 2014. `https://doi.org/10.1145/2541940.2541967`.

[10] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA 2016)*, 2016. `https://doi.org/10.1109/ISCA.2016.40`.

[11] Krzysztof Marcin Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Quincy Davis, Afroz Mohiuddin, Lukasz Kaiser, David Benjamin Belanger, Lucy J. Colwell, and Adrian Weller. Rethinking Attention with Performers. In *9th International Conference on Learning Representations, (ICLR 2021)*, 2021. `https://openreview.net/forum?id=Ua6zuk0WRH`.

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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*. Association for Computational Linguistics, 2019. `"https://aclanthology.org/N19-1423"`.

[13] William B. Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*, 2005. `"https://aclanthology.org/I05-5002"`.

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, (ICLR 2021)*, 2021. https://openreview.net/forum?id=YicbFdNTTy.

[15] Google. Out-of-memory issues. https://github.com/google-research/bert#out-of-memory-issues.

[16] Priya Goyal. [Re-checkpointing] Autograd container for trading compute for memory, 2018. https://github.com/pytorch/pytorch/blob/e1348973ac9a557aa6018e3fd2d5489619dd81a7/torch/utils/checkpoint.py.

[17] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *CoRR*, 2017. http://arxiv.org/abs/1706.02677.

[18] Sylvain Gugger. PyTorch DeBERTa model, 2020. https://github.com/huggingface/transformers/blob/5b6bd4e7880cd51375c2d6c33bbd8173acfd920b/src/transformers/models/deberta/modeling_deberta.py.

[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA 2016)*, 2016. https://doi.org/10.1109/ISCA.2016.30.

[20] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, (ICLR 2016)*, 2016. http://arxiv.org/abs/1510.00149.

[21] Dan Hendrycks and Kevin Gimpel. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units. *CoRR*, abs/1606.08415, 2016. http://arxiv.org/abs/1606.08415.

[22] Geoffrey E. Hinton, Oriol Vinyals, and Jeffrey Dean. Distilling the Knowledge in a Neural Network. *CoRR*, abs/1503.02531, 2015. http://arxiv.org/abs/1503.02531.

[23] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, 2012. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372.

[24] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In *Proceedings of Machine Learning and Systems (MLSys)*, 2021. https://proceedings.mlsys.org/paper/2021/file/c9e1074f5b3f9fc8ea15d152add07294-Paper.pdf.

[25] Animesh Jain, Parker Hill, Shih-Chieh Lin, Muneeb Khan, Md. Enamul Haque, Michael A. Laurenzano, Scott A. Mahlke, Lingjia Tang, and Jason Mars. Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO 2016)*, pages 41:1–41:13. IEEE Computer Society, 2016. https://doi.org/10.1109/MICRO.2016.7783744.

[26] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *International Symposium on Computer Architecture (ISCA 2018)*, 2018. /https://www.microsoft.com/en-us/research/publication/gist-efficient-data-encoding-deep-neural-network-training/.

[27] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate, 2019. https://github.com/parasj/checkmate.

[28] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Joseph Gonzalez, Kurt Keutzer, and Ion Stoica. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems (MLSys)*, 2020. /https://proceedings.mlsys.org/paper/2020/file/084b6fbb10729ed4da8c3d3f5a3ae7c9-Paper.pdf.

[29] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. TinyBERT: Distilling BERT for natural language understanding. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, 2020. https://aclanthology.org/2020.findings-emnlp.372.

[30] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Singh Bajwa, Sarah Bates, Suresh Bhatia, Nanette J. Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert B. Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Daniel Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle A. Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, J. W. Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[31] Patrick Judd, Jorge Albericio, Tayler H. Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO 2016)*, 2016. https://doi.org/10.1109/MICRO.2016.7783722.

[32] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic Tensor Rematerialization (DTR) Prototype, 2020. https://github.com/uwsampl/dtr-prototype.

[33] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. Dynamic Tensor Rematerialization. In *9th International Conference on Learning Representations, (ICLR 2021)*, 2021.

[34] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR*, abs/1907.11692, 2019. http://arxiv.org/abs/1907.11692.

[35] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer Sentinel Mixture Models. In *5th International Conference on Learning Representations, (ICLR 2017)*, 2017. https://openreview.net/forum?id=Byj72udxe.

[36] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed Precision Training. In *6th International Conference on Learning Representations, (ICLR 2018)*, 2018. https://openreview.net/forum?id=r1gs9JgRZ.

[37] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA: Is CUDA the Parallel Programming Model That Application Developers Have Been Waiting For? *Queue*, 2008. https://doi.org/10.1145/1365490.1365500.

[38] NVIDIA. NVIDIA Driver Downloads. https://www.nvidia.com/Download/index.aspx.

[39] NVIDIA. Tesla V100 Data Center GPU, 2017. https://www.nvidia.com/en-us/data-center/v100/.

[40] NVIDIA. GEFORCE RTX 2080 Ti, 2018. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/.

[41] NVIDIA. BERT For PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT/, 2019.

[42] NVIDIA. NVLink, 2019. https://www.nvidia.com/en-us/data-center/nvlink/.

[43] NVIDIA. NVIDIA A100 Tensor Core GPU, 2020. https://www.nvidia.com/en-us/data-center/a100/.

[44] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.

[45] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel S. Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, (ISCA 2017)*. ACM, 2017. https://doi.org/10.1145/3079856.3080254.

[46] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, (NeurIPS 2019)*, 2019. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[47] PCI-SIG. Specifications, 2005. /https://pcisig.com/specifications.

[48] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. Capuchin: Tensor-Based GPU Memory Management for Deep Learning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '20. Association for Computing Machinery, 2020. /https://doi.org/10.1145/3373376.3378505.

[49] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 2019.

[50] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021. https://doi.org/10.1145/3458817.3476205.

[51] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2016. "https://aclanthology.org/D16-1264".

[52] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-Efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[53] Samuel Rota Bulò, Lorenzo Porzi, and Peter Kontschieder. In-Place Activated BatchNorm for Memory-Optimized Training of DNNs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[54] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning Representations by Back-propagating Errors. *Nature*, 323(6088), 1986. http://www.nature.com/articles/323533a0.

[55] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *CoRR*, abs/1910.01108, 2019. http://arxiv.org/abs/1910.01108.

[56] Or Sharir, Barak Peleg, and Yoav Shoham. The Cost of Training NLP Models: A Concise Overview. *CoRR*, abs/2004.08900, 2020. https://arxiv.org/abs/2004.08900.

[57] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D. Manning, A. Ng, and Christopher Potts. Recursive Deep Models for Semantic Compositionality Over a Sentiment Treebank. In *EMNLP*, 2013.

[58] Phillip Spencer. Solution to the Transcendental Equation $2^x + 3^x = 5$, 1999. https://www.math.toronto.edu/mathnet/questionCorner/transsol.html.

[59] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56), 2014. http://jmlr.org/papers/v15/srivastava14a.html.

[60] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, (ACL 2019)*, 2019. https://doi.org/10.18653/v1/p19-1355.

[61] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient Transformers: A Survey. *CoRR*, 2020. https://arxiv.org/abs/2009.06732.

[62] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, 2009.

[63] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. /https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[64] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, and Anand Raghunathan. ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, (ISCA 2017)*. ACM, 2017. https://doi.org/10.1145/3079856.3080244.

[65] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic GPU Memory Management for Training Deep Neural Networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Association for Computing Machinery, 2018. /https://doi.org/10.1145/3178487.3178491.

[66] Kyle Wiggers. OpenAI's massive GPT-3 model is impressive, but size isn't everything, 2020. https://venturebeat.com/2020/06/01/ai-machine-learning-openai-gpt-3-size-isnt-everything/.

[67] Wikipedia. English Wikipedia, 2021. /https://en.wikipedia.org/.

[68] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. Association for Computational Linguistics, 2018. "https://aclanthology.org/N18-1101".

[69] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, (EMNLP)*. Association for Computational Linguistics, 2020. "https://www.aclweb.org/anthology/2020.emnlp-demos.6".

[70] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *Proceedings of the 33rd ACM Symposium on User Interface Software and Technology (UIST'20)*, 2020.

[71] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big Bird: Transformers for Longer Sequences. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, (NeurIPS 2020)*, 2020. `https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html`.

[72] Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. SWAG: A Large-Scale Adversarial Dataset for Grounded Commonsense Inference. In *EMNLP*, 2018.

[73] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA 2020)*, 2020. `https://doi.org/10.1109/ISCA45697.2020.00092`.

[74] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Benchmarking and Analyzing Deep Neural Network Training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A] This is a system level work which does not have any direct negative impact other than that of the underlying model. We discuss the positive impact in Section 1.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [Yes] See Appendix/
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] Code will be cleaned up and submitted with final submission.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] For accuracy experiments, otherwise all others are system level experiments which should not change based on random seed and are expensive to run.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [No]
   (c) Did you include any new assets either in the supplemental material or as a URL? [No]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]

5. If you used crowdsourcing or conducted research with human subjects...

(a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Summary of Appendices

These appendices contain additional details to cover different aspects of our work: motivation, related work, implementation, methodology, and extra ablation studies.

Appendix A contains a more detailed breakdown of the memory usage of BERT$_{BASE}$.

Appendix B goes over relevant related works and the difference between them and our work.

Appendix C goes more in depth into the major memory reduction footprint techniques, and compare them to our method.

Appendix D covers details of the LayerNorm backward pass derivation.

Appendix E includes implementation details of our optimizations.

Appendix F includes additional information on the implementation of our optimizations.

Appendix G includes a more detailed description of our experimental setups.

Appendix G includes a memory footprint ablation study with respect to Tempo optimizations.

A A Closer Look at the Memory Breakdown of BERT$_{BASE}$

Figure 9 shows a detailed memory breakdown of the Huggingface BERT$_{BASE}$ implementation [69] on the MRPC [13] fine-tuning task at a batch size of 32, profiled using the skyline tool [70]. From the figure, encoder layer activations are clearly the major contributor to the memory footprint compared to parameter weights, gradients, and optimizer states.

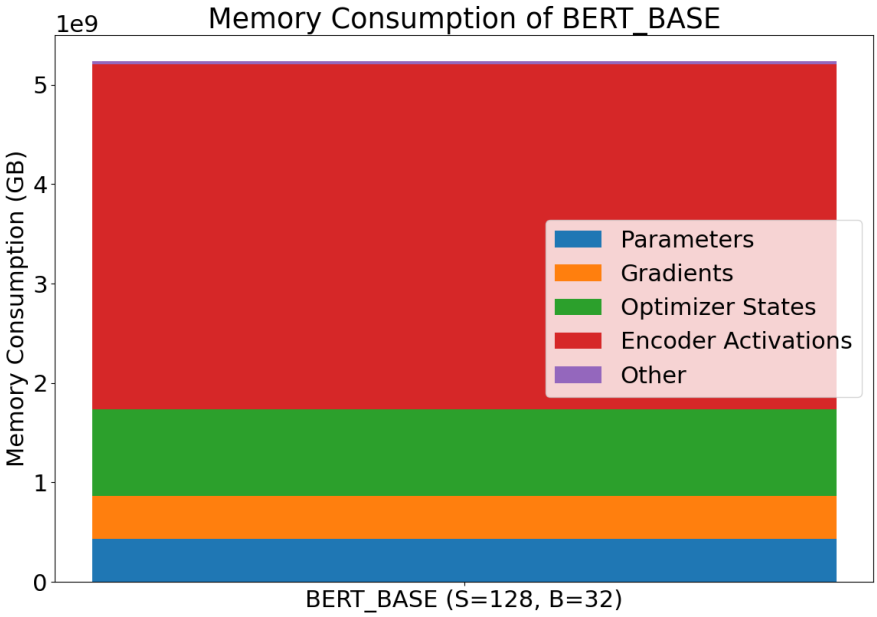


Figure 9: GPU memory breakdown for BERT$_{BASE}$ [12] fine-tuning on the MRPC [13] task using the Huggingface [69] at a sequence length of 128 and batch size of 32.

B Related Works

There has been a number of previous works [55, 29, 5, 11, 71] which focus on developing models that remain competitive with BERT and other Transformer-based models while requiring less memory and

compute. One approach that has been taken is to tackle the $\mathcal{O}(S^2)$ nature of the attention mechanism [61]. Out of these, a few interesting and relevant ideas in terms of memory footprint reduction involve sparsifying the attention mechanism [5, 71], or using a decomposable softmax layer in order to avoid doing the $\mathcal{O}(S^2)$ computation in the first place [11]. In contrast, our technique targets a broad spectrum of sequence lengths. Other models such as TinyBERT [29] and DistilBERT [55] take a different approach in reducing the total model size. In these works, the technique of *Distillation* is used to train a smaller student network from a larger teacher network [22]. These approaches are model level algorithm changes, and therefore orthogonal to our work.

Two other important techniques are mixed precision training [36] and in-place activated BatchNorm [53]. Mixed precision training involves training using both 32-bit and 16-bit IEEE floating point numbers depending on the numerical sensitivity of different layers [36]. This both decreases the training time and the memory pressure of the model dependent on the hardware [36]. Reversible BatchNorm uses techniques designed to optimize the memory footprint of CNN models which make extensive use of RELU and BatchNorm operators [53]. They do this by deriving in-place expressions for these operators (the input feature map for these operators no longer needs to be stored). Although this work shows better performance than checkpointing, it is also specific to CNN models, so is not directly comparable to our work. Other recent works such as Substation [24] aims to improve training performance by reducing the total movement of data with operator fusion; this is also orthogonal to our memory footprint based approach.

Other techniques are more focused on inference and model weights, both of which are not applicable in this context of Transformer-based model *training*, where the *activations* are the memory bottleneck [20, 19, 10, 25, 2, 9, 31, 45, 64].

C General Memory Footprint Reduction Techniques

We expand on the techniques we referred to in section 2 of checkpointing, offloading, and compression in this section. As previously iterated in section 2, (1) these techniques do not look closely at the specific structure of the BERT model, (2) At a per layer level, our techniques can provide better performance at a cheaper computational cost, and (3) these techniques are orthogonal to our work.

C.1 Checkpointing

Early work in this direction was able to reduce the memory cost of linear models to $\mathcal{O}(\sqrt{n})$ for general linear models in the number of layers. Further work has expanded on this. Echo has innovative optimizations for specific models [73], Checkmate computes an optimal checkpointing schedule minimizing recomputation time with respect to a specific memory budget [28], and Dynamic Tensor Rematerialization uses online heuristics to minimize recomputation time while training [33]. Furthermore, there are works in this area that combine checkpointing with other techniques such as offloading detailed below.

C.2 Offload

Initial work in this direction such as vDNN focused on specific schemes to offload layer outputs depending on their computation cost [52]. There was also some consideration into pre-fetching feature maps in anticipation of their use in the backward pass. More work in that direction focuses on several different directions. For one, Superneurons [65] combines both checkpointing and offloading. In this work, they checkpoint computationally cheap layers, avoiding transfer overhead, while simultaneously offloading computationally expensive layers, avoiding recomputation overhead for those cases. Capuchin takes this further in considering tensor level accesses, as well as considering the runtime fetching and computation time in making decisions on which strategy to apply [48]. Additionally, ZeRO-Infinity combines offloading of feature maps with offloading of model states and other optimizations [50].

C.3 Compression

Works like Gist [26] and ActNN [6] both include forms of lossy compression. Gist specifically targets CNN based models with a variety of different lossy and lossless optimizations, taking an

approach similar to our own in examining the model structure closely [26]. However, as this is a work focused on CNNs, the techniques described do not directly apply to the BERT model we are aiming to optimize for. ActNN on the other hand is a more general technique, using a quantization strategy designed to reduce the number of bits needed for feature map storage, while preserving certain theoretical guarantees regarding model conversion [6]. It additionally shows good performance relative to techniques such as Dynamic Tensor Rematerialization [33] and Capuchin [48]. This work does not consider the BERT model however [6].

D Backward pass of In-place LayerNorm

In the following, we show how to compute the gradients of LayerNorm layer using the output.

D.1 Notations

We use x, $\hat{x}$, y, μ and σ^2 to represent the input, intermediate normalized input, output, and mean and variance of the input, respectively. The parameters of LayerNorm function, scaling factor and bias, are denoted by γ and β, respectively. L represents the loss. For simplicity but without the loss of generality, we assume the size of input is (N, M), where the second dimension represents all the dimensions that are needed to be normalized. The meanings, definitions and sizes of variables are listed in Table 3.

Meaning	Definition	Size
input	$x = \{x_{ij}, i = 1, ..., N,\ j = 1, ..., M\}$	(N, M)
norm-input	$\hat{x} = \{\hat{x}_{ij}, i = 1, ..., N,\ j = 1, ..., M\}$	(N, M)
output	$y = \{y_{ij}, i = 1, ..., N,\ j = 1, ..., M\}$	(N, M)
mean	$\mu = \{\mu_i, i = 1, ..., N\}$	(N)
variance	$\sigma^2 = \{\sigma_i^2, i = 1, ..., N\}$	(N)
weight	$\gamma = \{\gamma_j, j = 1, ..., M\}$	(M)
bias	$\beta = \{\beta_j, j = 1, ..., M\}$	(M)

Table 3: The meanings, definitions and sizes of variables used in LayerNorm layer.

D.2 Forward Pass

In the forward pass, input is firstly normalized along the second dimension, and then scaled and shifted accordingly.

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$
$$y_{ij} = \gamma_j \hat{x}_{ij} + \beta_j$$

where $\mu_i = \frac{1}{M}\sum_{j=1}^{M} x_{ij}$ and $\sigma_i^2 = \frac{1}{M}\sum_{j=1}^{M}(x_{ij} - \mu_i)^2$. ϵ is added for numerical stability.

D.3 Backward Pass

Our goal is to use output to compute the gradients with minimum overhead. Intuitively however, the input is needed to compute the gradients of LayerNorm, which means we need to compute backwards to get input. We find that we can use the intermediate normalized input to get what we want. The gradient derivations are listed as follows,

$$\frac{\partial y_{ij}}{\partial \gamma_j} = \hat{x}_{ij}, \quad \frac{\partial y_{ij}}{\partial \beta_j} = 1, \quad \frac{\partial y_{ij}}{\partial \hat{x}_{ij}} = \gamma_j,$$

$$
\begin{aligned}
\frac{\partial L}{\partial \gamma_j} &= \sum_{i=1}^{N} \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial \gamma_j} = \sum_{i=1}^{N} \frac{\partial L}{\partial y_{ij}} \hat{x}_{ij}, \\
\frac{\partial L}{\partial \beta_j} &= \sum_{i=1}^{N} \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial \beta_j} = \sum_{i=1}^{N} \frac{\partial L}{\partial y_{ij}}, \\
\frac{\partial L}{\partial \hat{x}_{ij}} &= \frac{\partial L}{\partial y_{ij}} \frac{\partial y_{ij}}{\partial \hat{x}_{ij}} = \frac{\partial L}{\partial y_{ij}} \gamma_j,
\end{aligned}
$$

Here we can the gradients with regard to γ, β and $\hat{x}$. We still need to derive the gradient to input further.

$$
\frac{\partial \hat{x}_{ij}}{\partial \sigma_i^2} = -\frac{\hat{x}_{ij}}{2(\sigma_i^2 + \epsilon)}, \quad \frac{\partial \hat{x}_{ij}}{\partial \mu_i^2} = -\frac{1}{\sqrt{\sigma_i^2 + \epsilon}},
$$

$$
\begin{aligned}
\frac{\partial L}{\partial \sigma_i^2} &= \sum_{p=1}^{N} \sum_{q=1}^{M} \frac{\partial L}{\partial \hat{x}_{pq}} \frac{\partial \hat{x}_{pq}}{\partial \sigma_i^2} \\
&= \sum_{q=1}^{M} \frac{\partial L}{\partial \hat{x}_{iq}} \frac{\partial \hat{x}_{iq}}{\partial \sigma_i^2} \quad (p = i) \\
&= \sum_{j=1}^{M} \frac{\partial L}{\partial \hat{x}_{ij}} \frac{\partial \hat{x}_{ij}}{\partial \sigma_i^2} \quad (let\ q = j) \\
&= \sum_{j=1}^{M} \frac{\partial L}{\partial y_{ij}} \gamma_j \cdot \left(-\frac{\hat{x}_{ij}}{2(\sigma_i^2 + \epsilon)} \right),
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial L}{\partial \mu_i} &= \sum_{p=1}^{N} \sum_{q=1}^{M} \frac{\partial L}{\partial \hat{x}_{pq}} \frac{\partial \hat{x}_{pq}}{\partial \mu_i} \\
&= \sum_{q=1}^{M} \frac{\partial L}{\partial \hat{x}_{iq}} \frac{\partial \hat{x}_{iq}}{\partial \mu_i} \quad (p = i) \\
&= \sum_{j=1}^{M} \frac{\partial L}{\partial \hat{x}_{ij}} \frac{\partial \hat{x}_{ij}}{\partial \mu_i} \quad (let\ q = j) \\
&= \sum_{j=1}^{M} \frac{\partial L}{\partial y_{ij}} \gamma_j \cdot \left(-\frac{1}{\sqrt{\sigma_i^2 + \epsilon}} \right),
\end{aligned}
$$

$$
\frac{\partial \sigma_i^2}{\partial x_{ij}} = \frac{2(x_{ij} - \mu_i)}{M}, \quad \frac{\partial \mu_i}{\partial x_{ij}} = \frac{1}{M}, \quad \frac{\partial \hat{x}_{ij}}{\partial x_{ij}} = \frac{1}{\sqrt{\sigma_i^2 + \epsilon}},
$$

Combining all the intermediate results above, we have

$$
\begin{aligned}
\frac{\partial L}{\partial x_{ij}} &= \sum_{p=1}^{N} \sum_{q=1}^{M} \left(\frac{\partial L}{\partial \hat{x}_{pq}} \frac{\partial \hat{x}_{pq}}{\partial x_{ij}} \right) + \sum_{p=1}^{N} \left(\frac{\partial L}{\partial \sigma_p^2} \frac{\partial \sigma_p^2}{\partial x_{ij}} + \frac{\partial L}{\partial \mu_p} \frac{\partial \mu_p}{\partial x_{ij}} \right) \\
&= \frac{\partial L}{\partial \hat{x}_{ij}} \frac{\partial \hat{x}_{ij}}{\partial x_{ij}} + \frac{\partial L}{\partial \sigma_i^2} \frac{\partial \sigma_i^2}{\partial x_{ij}} + \frac{\partial L}{\partial \mu_i} \frac{\partial \mu_i}{\partial x_{ij}} \\
&= \left[\frac{\partial L}{\partial y_{ij}} \gamma_j - \left(\sum_{j=1}^{m} \frac{\partial L}{\partial y_{ij}} \gamma_j \cdot \hat{x}_{ij} \right) \cdot \frac{\hat{x}_{ij}}{m} \right. \\
&\quad \left. - \left(\sum_{j=1}^{m} \frac{\partial L}{\partial y_{ij}} \gamma_j \right) \cdot \frac{1}{m} \right] \cdot \frac{1}{\sqrt{\sigma_i^2 + \epsilon}}
\end{aligned}
$$

where intermediate normalized input $\hat{x}$ can be computed as $\hat{x}_{ij} = (y_{ij} - \beta_j)/\gamma_j$. Therefore, by extra stashing weight (scaling factor) γ, bias β and variance of the input γ^2, we can get the gradients using output without recovering input.

E Implementation Details

E.1 In-Place GELU

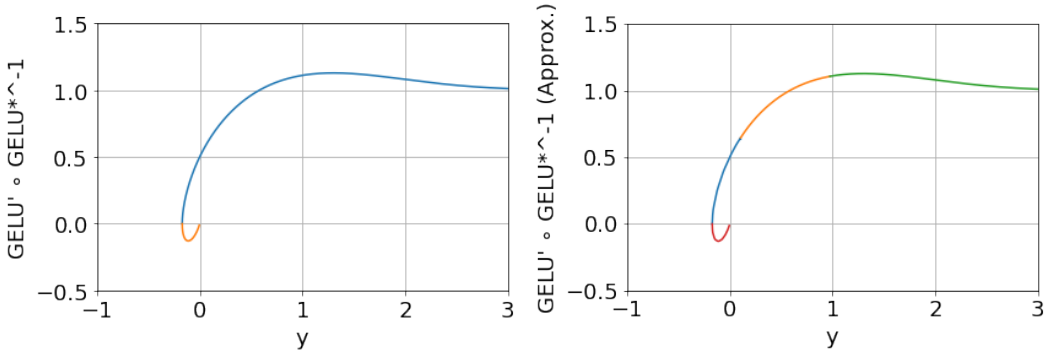
As we show in Section 3, it is possible to compute the inverse of the GELU function by knowing what side of the minimum the input originated from. This is shown in Equation 1 where GELU^* is a function that returns both the GELU output and m is the mask bit that denotes which side of the minimum the input originates from.

$$\text{GELU}^{*-1}(\text{GELU}(x), m) = x \tag{1}$$

Moreover, we observe that we do not need to compute the inverse and then compute the derivative with respect to the input in a two step process as a naïve approach would suggest. Instead, we can precompute this composition (see Equation (2)):

$$\frac{d\,\text{GELU}}{dx}(y) = \text{GELU}' \circ \text{GELU}^{*-1}(y, m) \tag{2}$$

in order to **compute the derivative with respect to the input** directly using the output value. A plot of this relation is shown in Figure 10a.



(a) GELU derivative from (2). The section corresponding to $x > -0.75179$ is in blue, $x \leq -0.75179$ – in orange.

(b) Our approximation of function (2) with a piece-wise polynomial approximation. Different sections of the approximation are shown in different colours.

A key observation about the GELU function is that it is transcendental, and hence there is no simple solution for the inverse of the GELU function in terms of elementary functions [58]. We therefore approximate sections of Equation (2) with a piece-wise polynomial with a degree up to 13.[5] A plot of this approximation is shown in Figure 10b.

E.2 In-Place LayerNorm

As stated in Section 3, we aim to reuse the output of the LayerNorm [4], which must be stored for the backpropagation of the successive fully-connected layers, while discarding the input. Although prior work has demonstrated the usability of In-Place Activated BatchNorm in the context of CNN networks [53], we note that this approach is not applicable in the Transformer case, which employs LayerNorm instead [63].

By employing an alternative derivation for the gradient of LayerNorm which stashes alternative parameters, we can compute the gradients with *negligible* performance overhead while achieving ideal memory footprint reduction for this operator. Following a similar approach as for In-Place

[5] Additional details of this implementation are covered in Appendix F.

GELU, we implement this operator as a Python PyTorch module, allowing it to be easily substituted in place of the existing LayerNorm layer in an implementation of the BERT model [41, 69]. See F for additional implementation details and the full derivation.

E.3 Sub-Layer Dropout Recomputation

Our basic implementation of Sub-Layer Dropout Recomputation follows the example of our other optimizations. Note that the way Dropout is implemented requires a randomly generated mask, where a portion of the inputs are set to zero according to a percentage p, which is also needed in the backward pass [46, 7]. We simply stash this mask, and discard the output. Then, in the backward pass, we recompute the output as shown in Figure 4. Storing a boolean mask of size N will take $N \times 1$ bytes, boolean tensors in PyTorch use 1 byte per value, while 32-bit floating points will use $N \times 4$ bytes [46]. Therefore, the total memory saved by discarding the output will be $4/5$ of the total pre-optimization dropout output memory cost.

In contrast to prior works which modify the framework and may not expose this optimization [48, 32, 27], we develop a PyTorch module which can be added in to reduce the memory pressure of the critical attention section of the Transformer-based models [63] with minimal overhead. Appendix F provides more in-depth comparison between prior work and our implementation.

E.4 Other Engineering Optimizations

We note that PyTorch uses a memory-inefficient implementation of the softmax function which retains both the input and output of the function for the backward pass [46]. Instead, only the output is necessary. This optimization has also previously been implemented as part of some models in the Huggingface library [18]. We use this optimization as well in our implementation of the attention mechanism.

F Additional Implementation Details

This section contains additional implementation details of our technique.

F.1 In-place GELU

We implement this optimization in PyTorch. In this case, we write the forward pass to return the GELU of the input function, as well as a boolean mask indicating whether the input is greater than or equal to the point at which the minimum value occurs, $x \approx -0.75179$. The backward pass is slightly more complicated. We take as inputs to the backward pass the incoming gradient, the saved mask, as well as the saved output values. The corresponding approximating polynomial shown in Figure 10b is determined and then computed. We use approximating polynomials of up to degree 13 in this case.

We implement the forward and backward pass using CUDA [37] which is wrapped in C++ [23]. Both of these are wrapped in a Python [62] PyTorch layer to be substituted easily for an existing implementation. While profiling our implementation we found that the memory latency of loading these inputs, as well as storing the output to be the bottleneck. We were thereby able to implement polynomials of degree 13, since the computation is hidden by the memory access latency, although better approximations may be possible.

F.2 In-place LayerNorm

In-place Layernorm is implemented as a custom PyTorch module [46]. This is done in 3 stages as per PyTorch's custom module implementation. To do this, we write a custom CUDA [37] implementation based on PyTorch's own implementation of the LayerNorm layer [46]. We write a forward and backward layer wrapper in C++ [23], which is then again wrapped as a Python [62] PyTorch module, allowing it to be easily substituted in place of the existing LayerNorm layer in an implementation of the BERT model such as the NVIDIA DeepLearningExamples BERT implementation or the Huggingface implementation [41, 69].

F.3 Sub-Layer Dropout Recomputation

We note that although the state-of-the-art recomputation/checkpointing papers [28, 33] consider such an abstract idea in theory, their practical implementations [27, 32] never treat the sub-layer granularity provided in the frameworks as the lowest granularity of those techniques' applicability. We concede that in the case of TensorFlow, the dropout layer has additional underlying granularity as a result of its implementation [1], and hence this would not be applicable in that case – and Checkmate [28] would consider this level of granularity. However, this is not the case in PyTorch's checkpointing implementation [16]. The tensor-level granularity of optimization is noted in Capuchin [48] as well. However, Capuchin is a technique that requires runtime level profiling, and modifications to the framework. It may or may not offload, recompute, or otherwise store any part of the dropout layer specified.

Our method uses in-built PyTorch operators at a C++ level to rewrite the attention mechanism, which is again wrapped as a Python PyTorch module to be substituted.

G Experimental Setup

Table 4 shows a more detailed view of our experimental platform.

GPU	# GPUs	GPU Mem. (GB)	CUDA Version	GPU Driver	Pytorch Version	# (v)CPUs	Sys. Mem. (GiB)
2080 Ti	4	11	11.2	460.27.04	1.9.0	64	126
V100	4	16	11.0	450.142.00	1.9.0	32	244
A100	1	40	11.2	460.32.03	1.9.0	64	250

Table 4: A short summary of our test setups, including CUDA [37], PyTorch [46], and driver [38] versions.

H Memory Footprint Reduction Ablation Study

We calculate the memory footprint reduction contributed by each optimization across different sequence lengths relative to the total memory footprint of each encoder layer, given a H to A ratio of 64 [63, 12]. This is shown in Figure 11 for the selected configurations. From the figure, it's clear that In-Place GELU and LayerNorm provide the bulk of the memory footprint reduction in the short sequence length regime, while the other two optimizations provide an improvement in the long sequence length case. Note that this is due to the fact that the latter's memory footprint reduction is $\mathcal{O}(S^2)$, while the former's memory footprint reduction goes as $\mathcal{O}(SH)$ where S is the sequence length. This allows *Tempo* to stay robust to model parameters, and provide consistent performance across sequence lengths.

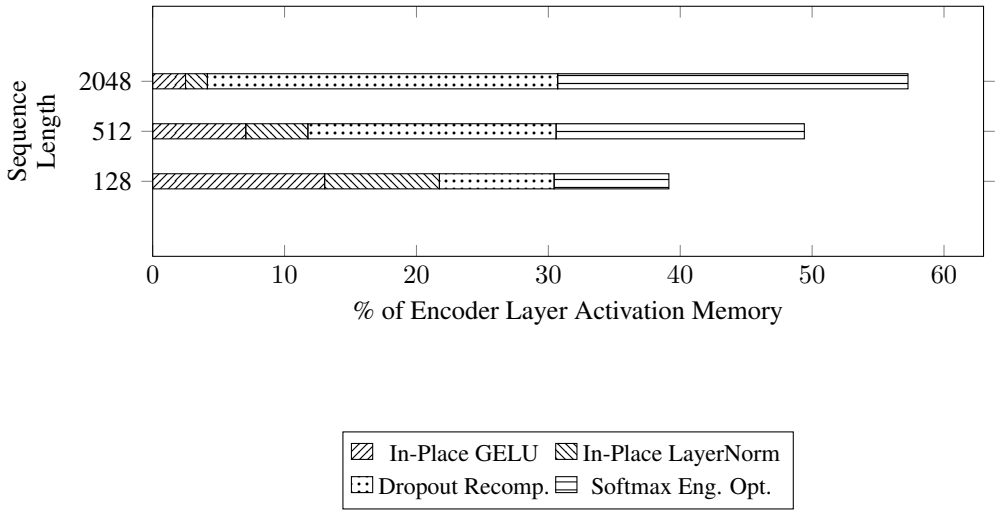


Figure 11: Per layer comparison of Tempo memory footprint reduction across different sequence lengths.