Sub-Linear Memory: How to Make Performers SLiM

Valerii Likhosherstov
University of Cambridge
vl304@cam.ac.uk

Krzysztof Choromanski
Google Brain & Columbia University

Jared Davis
DeepMind & Stanford University

Xingyou Song
Google Brain

Adrian Weller
University of Cambridge & Alan Turing Institute

Abstract

Transformer architectures have become very popular yet the original implementation requires $O(L^2)$ in serial time and memory as functions of input length $L$. Recent works proposed various linear self-attention mechanisms, scaling only as $O(L)$ for serial computation. We conduct a thorough complexity analysis of Performers, a class which includes most recent linear Transformer mechanisms. We note a remarkable computational flexibility: the gradient computation can be performed with no approximations using sublinear memory as a function of $L$ (in addition to negligible storage for the input sequence), at a cost of greater time complexity in the parallel setting. In the extreme case, a Performer consumes only $O(1)$ memory, and still requires $O(L)$ time. Due to complete backward-compatibility, this discovered time-memory tradeoff can be used for fine-tuning on low-memory devices in a decentralized fashion without any server computations.

1 Introduction

The Transformer architecture [38] has changed the landscape of deep learning for sequential data. A computational advantage of Transformers over conventional methods such as recurrent neural networks (RNNs) [17,9] is parallelization over the sequence dimension, meaning that the training speed can be increased by simply using more compute resources. However, this parallel-friendly structure of self-attention comes at a cost of quadratic $\Theta(L^2)$ time and memory complexity, where $L$ is the length of the Transformer’s input sequence.

A recent line of work aimed to address this restriction, using either structured sparsity [8], truncated back-propagation [12], clustering [20,31] or linear attention methods [18,10,11,33,23]. For a detailed overview of efficient Transformers, see [37]. We refer to the family of linear attention architectures as Performers (also known as Linear Transformers), following [11], since their generic kernel formulation covers all the aforementioned linear attention methods. Performers reduce time and memory complexity to linear $O(L)$ and can provably approximate conventional quadratic Transformers [11], demonstrating strong performance in a systematic comparison of efficient Transformers [36].

This recent trend of feeding longer sequences into Transformers, coupled with the use of deeper models, introduces new challenges for researchers and practitioners. Whereas conventional Transformer setups benefit from large-batch optimization [42], long sequence modelling necessitates smaller batch sizes in order to fit the model into memory. For example, recently proposed efficient Transformers

operating on long sequences used moderately small batch sizes of 1-8 instances \cite{20, 18, 11}. Aiming to use larger batch sizes, practitioners introduced various tricks – e.g. \cite{28} introduced gradient accumulation (included in the popular Fairseq library, \cite{27}), which splits the batch into smaller chunks which are evaluated sequentially, then the resulting batch gradient is accumulated.

Gradient accumulation allows to decrease memory usage at the cost of longer time, but it can only be applied when the batch size is bigger than 1. In this paper, we are discussing a situation when even a batch size of 1 is prohibitive, while longer processing times are affordable e.g. when fine-tuning a pretrained Transformer on low-memory devices (e.g. smartphones, embedded devices or microcontrollers) on the client-generated data without additional server computations. Heuristics, such as chunking the input into independent subsegments or truncated back-propagation \cite{12}, limit gradient propagation across the whole input, and, consequently, impair long-context learning.

We propose a solution based on the analysis of Performers. We discover a remarkable property: even for batch size of 1, a user can decrease memory consumption at the cost of smaller parallel bandwidth of the model. Notably, no approximations are introduced, so the obtained gradient is correct and backward-compatible. Our proposed long-sequence training algorithm can be used for training or fine-tuning on a low-memory device, thus contributing towards decentralized and democratized deep learning. The algorithm has the following advantages:

1. The integer parameter $C$, $1 \leq C \leq L$, controls a tradeoff between the memory, scaling as $O(C)$ in addition to a negligible input sequence storage, and parallel running time, scaling as $O((L/C) \log C)$. When $C = 1$, the algorithm consumes as much memory as if a single token were fed into Performer, plus a small fully characterized addition.

2. For any $C$, the algorithm requires as many floating point operations (FLOPs) as two standard forward and one backward pass plus a small addition.

We evaluate the proposed tradeoff empirically, and confirm backward-compatibility for the synthetic Copying Task and language modelling on Penn Treebank \cite{25} and Enwik8 \cite{24} datasets\footnote{Code: https://github.com/google-research/google-research/tree/master/performer/models/slim_performer}.
2 Background: linear self-attention and Performer

We commence by defining linear self-attention [18][11][33][23]. Consider a sequence of length $L$ and three matrices: queries $Q \in \mathbb{R}^{L \times d}$, keys $K \in \mathbb{R}^{L \times d}$ and values $V \in \mathbb{R}^{L \times d}$. Then linear self-attention is defined as a functional producing $Y = \text{Att}(Q, K, V) \in \mathbb{R}^{L \times d}$.

$$\forall l \in \{1, \ldots, L\}: Y_l = \frac{\sum_{v=1}^{l} V_v \cdot (g(K_v)^\top g(Q_l))}{\sum_{v=1}^{l} g(K_v)^\top g(Q_l)}, \quad (1)$$

where by $Z_l \in \mathbb{R}^{d \times \cdots}$ we denote slice $Z_{l_1, \ldots, l_d}$ of $Z \in \mathbb{R}^{d_1 \times d_2 \times \cdots}$ $g : \mathbb{R}^d \to \mathbb{R}^M$ as a positive mapping for a stable division in $1$. $M$ is typically much smaller than $L$, e.g. $M = d$.

For needs of autoregressive generative modelling, when each element depends only on previous elements of the sequence [38], $Y_l$ only depends on inputs at indices $\{1, \ldots, l\}$. Self-attention of type (1) was proposed for processing sequences of the long length $L$, since the original self-attention from [18] scales as $O(L^2)$. The second transition in (1), which is due to associativity of matrix multiplication, suggests an algorithm to compute linear self-attention efficiently in subquadratic time.

For tensors $Z^{(1)}, \ldots, Z^{(n)}$ of the same shape, let $Z = (Z^{(i)})_{i=1}^{n}$ be a tensor such that for all $1 \leq i \leq n$, $Z_{l_1, \ldots, l_d} = Z^{(i)}(l)$. By $R \in \mathbb{R}^{L \times d \times M}$, $S \in \mathbb{R}^{L \times M}$ denote a tensor and a matrix such that

$$R = \text{PS}((V_l \times g(K_l)^\top)_{l=1}^{L}), \quad S = \text{PS}((g(K_l))_{l=1}^{L}), \quad (2)$$

where $\text{PS}(Z) = (\sum_{v=1}^{l} Z_{v,l})_{l=1}^{L}$ is a prefix sum along the first dimension of $Z$. Next, compute

$$\forall l \leq l \leq L : Y_l = (R_l \times g(Q_l))/S_l^\top g(Q_l)). \quad (3)$$

Depending on the prefix-sum algorithm used in (2), we can obtain different complexity estimates for linear self-attention. [18] propose to iterate through $l = 1, \ldots, L$ maintaining only current $R_l, S_l$, and compute and store the result $Y_l$. This way, tensors $R, \text{PS}(R) \in \mathbb{R}^{L \times d \times M}$ are not stored in memory, resulting in $O(L)$ time complexity and $O(L(d + M) + dM)$ memory complexity. [18] also propose a similar computation of gradients through (2)[23]; see Appendix C for more discussion.

Alternatively, [11] employ a parallel prefix-sum algorithm [21][39], which, for a tensor $Z \in \mathbb{R}^{L \times \cdots}$, finds $\text{PS}(Z)$ in $O(\log L)$ parallel time and $O(L)$ memory. Applying this algorithm for $\text{PS}(R), \text{PS}(S)$ and then computing (3) results in only $O(\log L)$ parallel time complexity and $O(LdM)$ memory consumption. A similar approach was proposed in the context of parallel RNNs [26].


Let $p \in \Sigma^L$ be an input sequence of length $L$, where $\Sigma$ is a finite alphabet. By $\text{emb}(p_l, l) \in \mathbb{R}^{d_{\text{model}}}$, $1 \leq l \leq L$, denote a linear combination of the $p_l$ token’s learned embedding and positional embedding of $l$’s position (sinusoids with different frequencies, as in [38]). Then Performer is defined as a parametrized mapping from $X^{(0)} = (\text{emb}(p_l, l))_{l=1}^{L} \in \mathbb{R}^{L \times d_{\text{model}}}$ into $X^{(out)} \in \mathbb{R}^{L \times \Sigma}$ through a sequence of hidden representations $X^{(1)}, \ldots, X^{(s)} \in \mathbb{R}^{L \times d_{\text{model}}}$. For each $1 \leq r \leq s$, $X^{(r)}$ is obtained from $X^{(r-1)}$ by applying a Performer layer consisting of two parts. The first part is a multi-head self-attention which comprises of $k$ linear self-attentions applied in parallel ($d_{\text{model}} = kd$) plus a skip connection. The second part is a feedforward network independently to each element of the sequence. See a formal definition of the architecture in Appendix A.

For each $l, 1 \leq l \leq L - 1$, $X^{(out)}_l$ are predicted logits of the next token $p_{l+1}$. Let $\mathcal{L}_l(X^{(out)}_l)$ denote the cross-entropy loss with respect to $p_{l+1}$, or zero when $l = L$. The loss is defined as

$$\mathcal{L} = (L - 1)^{-1} \cdot (\mathcal{L}_1(X^{(out)}_1) + \cdots + \mathcal{L}_L(X^{(out)}_L)). \quad (4)$$

3 Low-memory back-propagation algorithm

3.1 Compact notation for Performer

We rewrite transformations $X^{(0)} \to X^{(1)} \to \cdots \to X^{(s)}$ in the following form. For each $1 \leq r \leq s$,

$$T^{(r-1)}, G^{(r-1)} = F^{(r)}(X^{(r-1)}; \theta), \quad U^{(r-1)} = \text{PS}(T^{(r-1)}), \quad X^{(r)} = G^{(r)}(U^{(r-1)}, G^{(r-1)}; \theta). \quad (5)$$
The representation (5) encapsulates architecture details of Performer inside \( \mathbf{F}^{(r)} \) and \( \mathbf{G}^{(r)} \). In fact, the representation (5) holds for various possible modifications, proposed in the literature. This includes, but is not limited by the different positioning of layer normalization \([21, 38]\), adding a stabilizing gating mechanism \([29]\), weight sharing across layers \([22]\) or reversible Transformer layers \([20]\).

High-level description of the algorithm. The algorithm, proposed in the remainder of the section, iterates over the sequence 1, \ldots, \( L \) and only maintains a front of current prefix sum values, thus allowing a substantial memory improvement. The backward pass is implemented similarly: current prefix sums along with their gradients are maintained in a dynamic programming fashion, but the iteration proceeds in a backward direction.

3.2 Forward computation

Suppose the memory budget is not enough to perform a complete forward pass through Performer (Equation 5 for \( r = 1, \ldots, s \)), because the input sequence length \( L \) is too big. We show that instead...
we can emulate the full forward computation under the memory needed for a forward pass through the input of length \( C \leq L \), plus a small addition. \( 1 \leq C \leq L \) is arbitrary and user-defined.

Split matrices \( X(r), T(r), \Gamma(r), U(r) \), into \( N \) slices of size at most \( C \) along the vertical axis (\( N = \lceil L/C \rceil \)): for each \( n, 1 \leq n \leq N \),

\[
X^{(r,n)} = (X^{(r)})_{n+1}^{B_n} \in \mathbb{R}^{B_n \times \text{model}}, \quad T^{(r,n)} = (T^{(r)})_{n+1}^{B_n} \in \mathbb{R}^{B_n \times D_1}, \quad \Gamma^{(r,n)} = (\Gamma^{(r)})_{n+1}^{B_n} \in \mathbb{R}^{B_n \times D_2},
\]

where \( A_n = (n-1)C \) and by \( B_n, 1 \leq n \leq N \), we denote the size of the \( n \)th slice: \( B_n = C \) for \( u < N, B_N = C \). Based on (5), we conclude that for each \( 1 \leq n \leq N \) and \( 1 \leq r \leq s \) it holds that

\[
\nabla \Gamma^{(r-1,n)} = F^{(r)}(X^{(r,n)}; \theta), \quad \nabla U^{(r-1,n)} = 1_{B_N} \times (U_{B_{n-1}}^{(r-1,n-1)})^T
\]

+ \( \Phi^T \nabla \Gamma^{(r-1,n)} \),

\[
X^{(r,n)} = G^{(r)}(U^{(r-1,n)}, \Gamma^{(r-1,n)}; \theta).
\]

Here, \( 1_{B_n} \in \mathbb{R}^{B_n} \) is a vector of \( B_n \) ones and we denote \( U_{B_0}^{(r-1,0)} = 0_{D_1} \) (a vector of \( D_1 \) zeros).

Now, instead of iterating over \( r = 1, \ldots, s \) and computing (7) for the whole sequence at once, we first iterate over \( n = 1, \ldots, N \) and then iterate over \( r = 1, \ldots, s \) in a nested loop to compute (8). As can be deduced from (7,8), we only need to maintain the current value of \( (U_{B_{n-1}}^{(r-1,n-1)})_r^{s} \in \mathbb{R}^{s \times D_1} \) in the outer iteration over \( n \).

Denote \( \mathcal{U}^{(n)} = (U_{B_n}^{(r-1,n)})_r^{s} \in \mathbb{R}^{s \times D_1}, 0 \leq n \leq N \). The memory-efficient algorithm for the forward pass is as follows. First, initialize \( \mathcal{L} = 0 \) and \( \mathcal{U}^{(0)} = 0_{s \times D_1} \). Then, iterate over \( n = 1, \ldots, N \) and maintain the current value of \( \mathcal{U}^{(n-1)} \). During each iteration, compute \( X^{(0,n)} = (\text{emb}(p_{A_n+1}, A_n + l))_{l=1}^{B_n} \). Then iterate over \( r = 1, \ldots, s \), where compute (7,8) and update \( \mathcal{U}^{(n)} = U^{(r-1,n)} \). Finally, compute \( X^{(out,n)} = X^{(s,n)} \mathcal{W}^{(out)} + b^{(out)} \) and update \( \mathcal{L} += \mathcal{L}^{(n)}(X^{(out,n)}) \), where we denote \( \mathcal{L}^{(n)}(X^{(out,n)}) = (L - 1)^{-1} \sum_{i=1}^{B_n} \mathcal{L}^{(n)}(X_i^{(out,n)}) \).

By the end of the iteration over \( n \), the correct loss value (4) is computed. As a result, the forward pass takes \( O(L) \) serial time or \( O((L/C) \log C) \) parallel time and consumes only \( O(C) \) memory. This is in addition to the input sequence \( p \in \Sigma^L \) storage, which is \( O(L) \) in principle, however the constant is negligibly small. For instance, if \( p \) is a flattened image or an ASCII text string, then it occupies precisely \( L \) bytes in memory. The log \( C \) term in the parallel time complexity is due to the parallel prefix-sum algorithm taking logarithmic time, as discussed in Subsection 2.

### 3.3 Back-propagation and the final algorithm

The goal of a backward pass is to compute the gradient \( \nabla \mathcal{L} \) of the loss function with respect to parameters \( \theta \). One can just perform automatic differentiation (implemented in Tensorflow and Pytorch) through the computation graph induced by the memory-efficient forward pass algorithm from Subsection 3.2. However, such a backward pass would need to store all intermediate tensors produced during the forward pass, resulting in \( O(L) \) memory complexity as a function of \( L \) and \( C \). Instead, we propose a back-propagation algorithm which has the same time and memory complexity as the efficient forward pass.

Let \( \theta^{(1)} = \cdots = \theta^{(N)} = \theta \) be results of a symbolic “identity operation” performed on \( \theta \), so that for \( 1 \leq n \leq N \), \( \theta^{(n)} \) is used instead of \( \theta \) in (7,8). Then the total gradient of \( \theta \) has the form

\[
\nabla \mathcal{L} = \nabla_{\theta^{(n)}} \mathcal{L} + \cdots + \nabla_{\theta^{(1)}} \mathcal{L}.
\]

In Appendix B, we derive an expression for \( \nabla \theta^{(n)} \mathcal{L}, 1 \leq n \leq N \). Namely, denote \( \mathcal{G}^{(n)} = \nabla_{\mathcal{U}} \mathcal{L}, \) then \( \nabla \theta^{(n)} \mathcal{L} = \nabla_{\theta^{(n)}} \mathcal{G}^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}, \mathcal{G}^{(n)}) \), where

\[
\Phi^{(n)} : \mathbb{R}^{n \text{param}} \times \mathbb{R}^{s \times D_1} \times \mathbb{R}^{s \times D_1} \to \mathbb{R},
\]

\[
\Phi^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}, \mathcal{Z}) = \mathcal{L}^{(n)}(X^{(out,n)}) + \sum_{r=1}^{s} \mathcal{Z}_r \mathcal{U}_r^{(n)}.
\]

In \( \Phi^{(n)} \)'s definition, \( X^{(out,n)} \) and \( \mathcal{U}^{(n)} = (U_{B_n}^{(r-1,n)})_r^{s} \) are results of (7,8) iteration over \( r = 1, \ldots, s \) with parameters \( \theta = \theta^{(n)} \) and \( (U_{B_n}^{(r-1,n-1)})_r^{s} \) equal to \( \Phi^{(n)} \)'s second argument \( \mathcal{U}^{(n-1)} \). Gradient \( \nabla \theta^{(n)} \mathcal{G}^{(n)} \) can be computed by automatic differentiation through \( \Phi^{(n)} \).
An efficient way to compute and sum up all $\nabla_{\theta(n)} \mathcal{L}$ is to iterate in a backward direction $n = N, \ldots, 1$ and to maintain values of $\mathcal{U}^{(n)}, \mathcal{G}^{(n)}, \mathcal{U}^{(N)}$ is known after the end of the forward pass, and

$$\forall 1 \leq n \leq N : \quad \mathcal{U}^{(n-1)} = \mathcal{U}^{(n)} - \sum_{l=1}^{B_n} (T_l^{(r-1,n)}) g_r.$$  \hspace{1cm} (9)

Further, in Appendix \[A\] we show that $\mathcal{G}^{(N)} = 0_{r \times D_1}$ and, for each $1 \leq n \leq N$,

$$\mathcal{G}^{(n-1)} = \nabla_{\mathcal{U}^{(n-1)}} \Phi^{(n)}(\rho^{(n)}, \mathcal{U}^{(n-1)}, \mathcal{G}^{(n)}).$$  \hspace{1cm} (10)

By a single auto-differentiation through $\Phi^{(n)}$ we can compute $\nabla_{\theta(n)} \mathcal{L} = \nabla_{\theta(n)} \Phi^{(n)}$ and the update \[\eqref{10}\]. If $w$ is some vector of length $B_n$ and $h$ is some scalar function of $v = PS(w)$, then for all $1 \leq l \leq B_n$ : $\nabla_h w_l = \sum_{u=1}^{B_n} \nabla_v v_l'$. In other words, the gradient through $PS(\cdot)$ is another prefix sum computed backwards. Hence, auto-differentiation through $\Phi^{(n)}$ takes the same parallel time $O(\log C)$, serial time $O(C)$ and memory $O(C)$, as the forward computation of $\Phi^{(n)}$. Since during the whole back-propagation algorithm, we only store and update tensors $\mathcal{U}^{(n)}, \mathcal{G}^{(n)}$, whose size doesn’t depend on $L$ and $C$, this results in total $O((L/C) \log C)$ parallel time, $O(L)$ serial time and $O(C)$ memory in addition to $p$ storage. A full description of the forward-backward pass is presented in Algorithm \[4\]. Figure \[4\] is an illustration of the algorithm.

### 3.4 Analysis of the running time and memory

As we have shown, Performer can be trained in parallel time $O((L/C) \log C)$ and $O(C)$ memory in addition to the input $p$ storage. Hence, $C$ is a tradeoff parameter: when $C$ is maximal ($C = L$), the model is fully-parallelized, therefore resulting in the fastest execution. Whereas minimal $C = 1$ corresponds to step-by-step processing, i.e. a fully-sequential regime which doesn’t benefit from parallelized computations on GPU or TPU, but consumes $O(1)$ memory as a function of $L$.

During the forward pass, Algorithm \[1\] requires as many total FLOPs as the naive forward pass through \[7\]. As for the backward pass, for each $1 \leq n \leq N$, the forward pass through $n$’s slice is repeated for symbolic construction of $\Phi^{(n)}$ (see Algorithm \[2\]), and then back-propagation is run through $\Phi^{(n)}$. In addition, a backward update of $\mathcal{U}^{(n)}$ \[9\] is computed, taking precisely $B_n s M (d + 1) k$ “add” operations. Hence, we conclude that Algorithm \[1\] requires as many FLOPs as two forward and one backward pass through \[7\] for the whole sequence $p$ plus $L s M (d + 1) k = L s M d_{model} + L s M k$ FLOPs. To characterize this addition, assuming that typically $d_{ff}$ (dimension of the feedforward block) is $4d_{model}$ in practice, observe that applying dense Performer layers \[1\] Appendix \[A\] alone requires $3 L s d_{model}^2 + 2 L s d_{model} d_{ff} = 11 L s d_{model}^2$ FLOPs. This is much bigger than $L s M d_{model} + L s M k$, since $M$ is much smaller than $d_{model}$ in practice \[1\] \[18\].

Since the back-propagation takes roughly 5 times more FLOPs than the forward pass \[13\], we conclude that memory efficiency of Algorithm \[1\] results in a small constant-time increase in FLOPs. FLOPs affect energy consumption \[40\], a crucial factor for on-device applications.

Further analysis of Algorithm \[1\] reveals that the $C = 1$ regime requires as much memory as if Transformer were applied to a sequence of length 1 plus exactly $2 s d_{model} (M + 1)$ floats for storing $\mathcal{U}, \mathcal{G}$. For comparison, the subset of $\theta$ corresponding to dense layers in self-attention and feed-forward blocks \[1\] \[15\], occupies $3 s d_{model}^2 + 2 s d_{model} d_{ff} = 11 s d_{model}^2$ floats. Again, this is much bigger than $2 s d_{model} (M + 1)$, since $M$ is much smaller than $d_{model}$ in practice.

Table \[7\] shows comparison of the proposed algorithm with other architectures such as a conventional Transformer, recurrent neural networks (RNNs, \[17\] \[9\]) and residual networks (e.g. Neural ODEs, \[6\]).

### 4 Experiments

Our main contribution is a new low-memory gradient computation algorithm for the existing Performer architecture. Performers have very competitive performance among other methods for long sequence modelling \[11\] \[13\] \[26\]. Hence, in the experimental section, we aim to answer the following questions about using this algorithm in practice. (a) Does the theoretical time-memory tradeoff, controlled by $C$, agree with benchmarks of time-memory for varied $C$? (b) In precise arithmetic, different values of $C$ lead to the same correct gradient $\nabla_{\theta} \mathcal{L}$. Does this hold in practice, when
finite-precision arithmetic is employed? (c) Can a model, pre-trained with a bigger $C$ (e.g. on a server), be fine-tuned with a smaller $C$ (e.g. on an embedded device)?

We address these questions below. We analyse 4 model configurations ($L, s, d_{model}$): $I = (8192, 1, 1024)$, $II = (1024, 3, 512)$, $III = (4096, 3, 1024)$, $IV = (16384, 3, 1024)$. In all configurations, we set $d_{ff} = 4d_{model}$, $k = d_{model}/64$ (number of heads). We set $M = d$ and employ $g(x) = (x_i^2)_{i=1}^d$, elementwise-quadratic feature mapping in (1), which we find to work well. In all experiments $\Sigma = \{0, \ldots, 255\}$ and batch size is set to 1. i.e. we analyse a setup where gradient accumulation cannot be used to decrease memory, and therefore our algorithm is crucial. Our code is in PyTorch 1.7 [30]. To ensure that reproduction of experiments is accessible for a wider audience, we use a single NVIDIA Tesla P100 GPU with 16 GB memory for each experiment.

4.1 Empirical benchmarking of the tradeoff

We run Algorithm 1 for configurations I, III, IV and different powers of 2 as $C$. We use input strings sampled randomly from $\Sigma^L$. In order to characterize the time-memory tradeoff, we measure wall-clock time and peak GPU memory for a single gradient evaluation. We use the torch.cuda.max_memory_allocated function to report peak GPU memory.

As discussed in Section 2, there are two methods to compute (2-3): the first (iterative) method doesn’t compute and store tensors (3) explicitly, resulting in smaller memory consumption at a cost of less parallelization, while the second one computes tensors (3) using the parallel prefix sum algorithm, therefore operating faster, but using more memory. The same methods can be applied for the memory-efficient algorithm when computing (7-8) updates. We implement and benchmark both methods as part of the algorithm. For the explicit prefix-sum method, we find that the torch.cumsum function works faster and consumes less memory than our custom implementation of the parallel prefix sum algorithm. We attribute this to hardware-optimized implementation of the native function, and use it in experiments. As for the iterative algorithm, we implement its “block” version, when, instead of iterating $l$ one-by-one, we iterate through blocks of small size (see details in Appendix D).
Table 1: Complexity for the back-propagation as a function of sequence length $L$ and the tradeoff parameter $C \leq L$. The indicated memory complexity is in addition to the input sequence $p$ storage. The serial time complexity for Performer is reported for the version with iterative PS($\cdot$) computation (as in [13]), while the parallel time is reported for the parallel prefix sum (as in [11]). For both methods, memory complexity is the same, but the constant is smaller for the iterative version.

<table>
<thead>
<tr>
<th>Model</th>
<th>Serial time</th>
<th>Parallel time</th>
<th>Memo</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN</td>
<td>$O(L)$</td>
<td>$O(L)$</td>
<td>$O(L)$</td>
</tr>
<tr>
<td>Residual NN</td>
<td>$O(L)$</td>
<td>$O(L)$</td>
<td>$O(L)$</td>
</tr>
<tr>
<td>Transformer</td>
<td>$O(L^2)$</td>
<td>$O(\log L)$</td>
<td>$O(L^2)$</td>
</tr>
<tr>
<td>Performer</td>
<td>$O(L)$</td>
<td>$O(\log L)$</td>
<td>$O(L)$</td>
</tr>
<tr>
<td>Our alg.</td>
<td>$O(L)$</td>
<td>$O(\log C)$</td>
<td>$O(C)$</td>
</tr>
<tr>
<td>Ours, $C = 1$</td>
<td>$O(L)$</td>
<td>$O(L)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Figure 3: Algorithm 1 compared to checkpointing of $\{U^{(n)}\}_{1 \leq n \leq N}$. Time and memory plots.

This way, the algorithm has a smaller constant in $O(L)$ time complexity and bigger constant in a “small” $O(dM)$ term of the memory complexity (assuming that $d, M \ll L$).

For a fixed $C$, in addition to reporting memory of Algorithm 1 we also report memory of the naive gradient computation run on a string of length $C$, sampled uniformly from $\Sigma^C$. This is to confirm that memory use of Algorithm 1 is just slightly above the full computation on the input of length $C$.

Results are reported in Figure 2 (left, middle). We observe significant improvements in memory compared to the full computation, as $C$ decreases. As $C$ converges to $2^0 = 1$, the remaining memory can be attributed to storage of the model’s parameters $\theta$. Time follows two regimes: declining fast as $C$ grows (meaning that prefix sums are parallelized) and declining slower for big values of $C$ (meaning that the practical limit of parallelization is reached). Memory scales slower than $O(C)$, as $C$ increases. We attribute this to details of the PyTorch internal code. We find that iterative version of (2-3) works only slightly slower than prefix-sum version, while consuming much less memory. Finally, Algorithm 1 consumes only slightly more memory in practice than the full method run on the input of length $C < L$ (“L/B” plots on Figure 2 (middle)).

In addition, we compare Algorithm 1 with checkpointing [15]. By checkpointing in this context we understand storing $\{U^{(n)}\}_{1 \leq n \leq N}$ during the forward pass and reusing them during the backward pass instead of recomputing. This results in a small FLOPs decrease since $U^{(n)}$, while memory scales as $O(L/C)$. We use config. 1 for comparison (Figure 3). While not faster in practice, checkpointing consumes much more memory as $C$ decreases for both iterative and prefix-sum computation of 2-3.

### 4.2 Effects of finite-precision arithmetic

Since the iterative version of (2-3) results in a good balance between time and memory of Algorithm 1, we use it further. To quantify finite-precision effects, we plot relative discrepancy $\|\nabla^{(C)}_\theta L - \nabla^{(full)}_\theta L\|_2/\|\nabla^{(full)}_\theta L\|_2$ between the gradient $\nabla^{(C)}_\theta$ produced by Algorithm 1 and the gradient...
Figure 4: Learning curves for three language modelling setups. We report accuracy on a newly generated data samples for Copying task, and Bits Per Character (BPC) metric on validation examples for Penn Treebank and Enwik8. F/T stands for “fine-tuning”. All curves are roughly the same and result in nearly the same performance, confirming correctness and backward-compatibility of gradients computed via memory-efficient Algorithm 1. Differences can be attributed to finite-precision arithmetic effects, accumulating over many iterations.

Table 3: Time per iteration (sec., averaged over 1000 iterations) and peak GPU memory (Gb). We also report memory required to store parameters $\theta$ only without any computations.

<table>
<thead>
<tr>
<th>Setup, $L, C$</th>
<th>BPC, F/T</th>
<th>BPC, no F/T</th>
<th>Memory, $\theta$</th>
<th>F/T mem., Alg. 1</th>
<th>F/T mem., full comp.</th>
<th>F/T time, Alg. 1</th>
<th>F/T time, full alg.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTB, 1024, 16</td>
<td>1.4263</td>
<td>1.6544</td>
<td>0.0661</td>
<td>0.1077</td>
<td>0.1834</td>
<td>0.8938</td>
<td>0.0426</td>
</tr>
<tr>
<td>ENW 4096, 64</td>
<td>1.5642</td>
<td>1.6141</td>
<td>0.2610</td>
<td>0.4276</td>
<td>0.8049</td>
<td>1.1142</td>
<td>0.2045</td>
</tr>
</tbody>
</table>

$\nabla_\theta L^{(\text{full})}$ produced by full computation. Figure 2 shows results for randomly initialized models. We observe a small discrepancy of order $10^{-6}$–$10^{-4}$, confirming the correctness of Algorithm 1.

4.3 Training from scratch and fine-tuning

To confirm backward compatibility of Algorithm 1 during training, we run three language modelling setups: Copying Task (CT), symbol-level Penn Treebank (PTB) and Enwik8 (ENW).

For the CT, we follow the setup from [20, 18], sampling inputs as $0 \omega_0 \omega$, where $\omega$ is drawn uniformly from $(\Sigma \setminus \{0\})^{L/2-1}$. In this setup, we only aggregate cross-entropy loss from the second half of the input, so the task is to reproduce the first half. We include the CT as a task where long-range signal is crucial, and the heuristic of “chunking” the input into segments would fail to solve the task.

We use model configurations I, II, III for the CT, PTB and ENW, resulting in sequence lengths $L = 8192, 1024, 4096$ respectively. For each setup, we compare training with full gradient computation, “fine-tuning” regime, when the first half of iterations is run using the full algorithm, and the second half is run using Algorithm 1 using various values of $C$. In addition, we include training from scratch equipped with memory-efficient gradient computation via Algorithm 1. Figure 4 demonstrates results: all methods result in almost same performance. This confirms that memory-efficient gradient computation is backward-compatible during training. Table 2 quantifies the memory savings and time tradeoff in all setups. Additional details and results, including bigger version of Figure 4, BPC for CT, train set performance, and curve differences in Figure 4, can be found in Appendix D.

4.4 One-shot fine-tuning under low memory

To analyze the scenario when model is pretrained on server and then fine-tuned (F/T) with a small $C$ on a low-memory device, we add the following experiment. We take a pretrained model from either PTB or ENW setup from Section 4.3 and subsample randomly 5000 examples from the corresponding validation set. We perform a one-step gradient descent with 0.01 learning rate (tuned on other random subset) to minimize the loss computed on the first half of each sequence and evaluate Bits Per Character (BPC) on the second half. In this experiment, the first half of the sequence represents the data generated by user on the device, and the second half is a new data to be predicted. Fine-tuning
procedure, therefore, represents “personalization” of the model to the specific user (see Table 3). We observe BPC improvement without any server compute and a memory improvement compared to the full computation, while time (≈ 1 sec.) is less crucial, since fine-tuning can run in the background.

5 Related Work and Extensions

Compatibility with other memory-optimization techniques. Observe that the specification (5) is compatible with the reversible layer design from [20], when the sparse self-attention is replaced with the linear self-attention. This can bring more memory savings, since one doesn’t need to store the whole symbolic \( \Phi(n) \) during the backward pass. Checkpointing along the layer dimension [15] can also be used to reduce the memory consumption for storing \( \Phi(n) \)’s graph, though at the cost of a longer execution time. The gradient accumulation technique [28] is also compatible with Algorithm 1, i.e. one can combine both methods to “collapse” batch and sequence dimensions simultaneously. Moreover, our algorithm is compatible with distillation [32], since it can be run on a distilled model.

Comparison with [18]. In [18], authors mention that a single self-attention block can be evaluated in \( O(1) \) additional memory. However, during back-propagation, one still needs to store \( L \) intermediate states, e.g. in the feedforward block. Hence, the full memory complexity is still \( O(L) \). In contrast, our method optimizes memory consumption along the sequence dimension for the whole multilayer model.

Extension to Transformers with dropout. Dropout [34] is a popular regularization technique. It is used with Transformers when the train dataset is small enough to cause overfitting (e.g. it wasn’t used with GPT-2, trained on a massive dataset). Our algorithm can be extended to stochastic computation graphs with dropout. For that, use separate random seeds to generate dropout masks for each slice \( 1 \leq n \leq N \), and reuse these seeds two times during the forward and backward pass through the \( n \)th slice.

6 Conclusion

We proposed an algorithm for memory-efficient back-propagation through a Performer. The algorithm reduces memory consumption along the sequence dimension, and can, therefore, be used for long-sequence training. The algorithm: (1) is completely backward-compatible, since it computes precise gradients and does not involve approximation, (2) does not require many additional computations, and (3) enables user control over the tradeoff between time and memory consumption.

Limitations. One limitation of this work is that the proposed memory improvements are traded off by longer running time. Another one is that these improvements rely on a specific feature of prefix sum signal propagation appearing in Performers and also, in parallelized versions of RNNs [26]. Finally, we don’t see an immediate extension of the proposed construction to the bidirectional self-attention case, used in Transformer encoders [38] and for masked language modelling [13]. The reason for that is that no notion of the prefix sum “front” can be introduced in this case.

Negative societal impacts. This work studies Performers – efficient Transformers. Transformers in general can have the following negative societal impacts: large carbon dioxide emissions [35], privacy and data leak vulnerabilities [5], bias and fairness issues and malicious misuse [4, 3].

Acknowledgments

We thank Tom Weingarten and Tamas Sarlos for many fruitful discussions.

Funding disclosure. V. L. acknowledges support from the Cambridge Trust and DeepMind. V. L. was part-time employed by Google while a PhD student. A.W. acknowledges support from a Turing AI Fellowship under grant EP/V025379/1, The Alan Turing Institute, and the Leverhulme Trust via CFI.

2See e.g. CausalFavor class in https://github.com/google/trax/blob/master/trax/layers/research/sparsity.py, which is compatible with the official Reformer code.
References


