Sub-Linear Memory: How to Make Performers SLiM

Anonymous Author(s)
Affiliation
Address
email

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 [35] has changed the landscape of deep learning for sequential data. A computational advantage of Transformers over conventional methods such as recurrent neural networks (RNNs) [14, 8] 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 [7], truncated back-propagation [10], clustering [17, 28] or linear attention methods [15, 9, 30, 20]. For a detailed overview of efficient Transformers, see [34]. We refer to the family of linear attention architectures as *Performers* (also known as *Linear Transformers*), following [9], 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 [9], demonstrating strong performance in a systematic comparison of efficient Transformers [33].

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 [39], 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 [17, 15, 9]. Aiming to use larger batch sizes, practitioners introduced various tricks – e.g. [25] introduced *gradient accumulation* (included in the popular Fairseq library, [24]), 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-

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

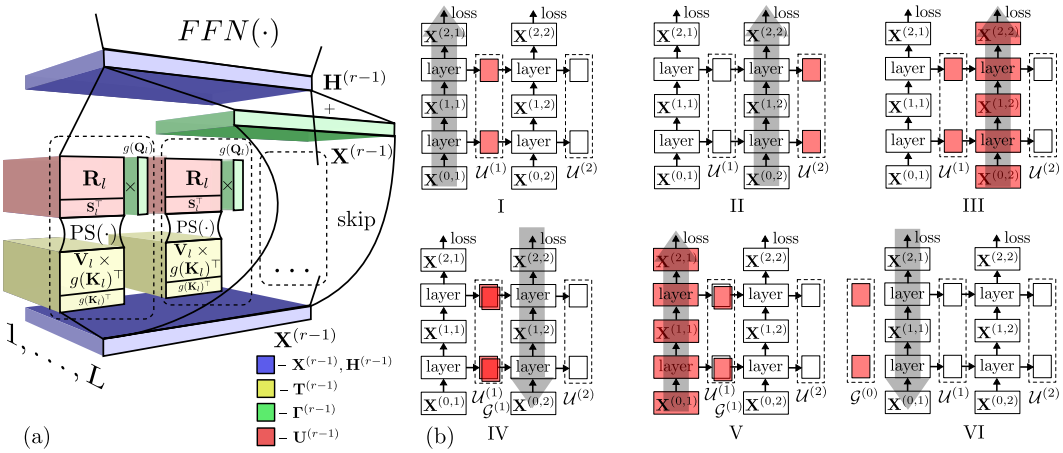

Figure 1: **(a)** rth layer and its decomposition into $\mathbf{T}^{(r-1)}, \mathbf{\Gamma}^{(r-1)}, \mathbf{U}^{(r-1)}$. **(b)** Illustration of Algorithm 1 when $r = n = 2$. Red color indicates objects stored in memory. I-II) forward passes for $n = 1, 2$ respectively, only the loss value and $\mathcal{U}^{(n)}$ are stored. III) backward pass start, forward computation through the slice $n = 2$ to build symbolic $\Phi^{(2)}$ and update $\mathcal{U}^{(2)} \to \mathcal{U}^{(1)}$. IV) back-propagation through $\Phi^{(2)}$ to find $\nabla_{\theta^{(2)}}\mathcal{L}$ and $\mathcal{G}^{(1)}$. V,VI) the same backward iteration for $n = 1$.

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 [10], 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 [22] and Enwik8 [21] datasets.

2 Background: linear self-attention and Performer

We commence by defining *linear self-attention* [15, 9, 30, 20]. Consider a sequence of length L and three matrices: *queries* $\mathbf{Q} \in \mathbb{R}^{L \times d}$, *keys* $\mathbf{K} \in \mathbb{R}^{L \times d}$ and *values* $\mathbf{V} \in \mathbb{R}^{L \times d}$. Then linear self-attention is defined as a functional producing $\mathbf{Y} = \mathrm{Att}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \in \mathbb{R}^{L \times d}$,

$$\forall l \in \{1, \ldots, L\} : \mathbf{Y}_l = \frac{\sum_{l'=1}^{l} \mathbf{V}_{l'} \cdot (g(\mathbf{K}_{l'})^\top g(\mathbf{Q}_l))}{\sum_{l'=1}^{l} g(\mathbf{K}_{l'})^\top g(\mathbf{Q}_l)} = \frac{(\sum_{l'=1}^{l} \mathbf{V}_{l'} \times g(\mathbf{K}_{l'})^\top) \times g(\mathbf{Q}_l)}{(\sum_{l'=1}^{l} g(\mathbf{K}_{l'}))^\top g(\mathbf{Q}_l)}, \quad (1)$$

where by $\mathbf{Z}_l \in \mathbb{R}^{d_2 \times \cdots}$ we denote slice $\mathbf{Z}_{l,:,\ldots,:}$ of $\mathbf{Z} \in \mathbb{R}^{d_1 \times d_2 \times \cdots}$. $g : \mathbb{R}^d \to \mathbb{R}^M_+$ s 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 [35], $\mathbf{Y}_l$ 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 [35] 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 subqudratic time.

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

$$\mathbf{R} = \mathrm{PS}((\mathbf{V}_l \times g(\mathbf{K}_l)^\top)_{l=1}^L), \ \mathbf{S} = \mathrm{PS}((g(\mathbf{K}_l))_{l=1}^L), \tag{2}$$

where $\mathrm{PS}(\mathbf{Z}) = (\sum_{i'=1}^i \mathbf{Z}_{i'})_{i=1}^n$ is a *prefix sum* along the first dimension of $\mathbf{Z}$. Next, compute

$$\forall 1 \le l \le L : \mathbf{Y}_l = (\mathbf{R}_l \times g(\mathbf{Q}_l))/(\mathbf{S}_l^\top g(\mathbf{Q}_l)). \tag{3}$$

Depending on the prefix-sum algorithm used in (2), we can obtain different complexity estimates for linear self-attention. [15] propose to iterate through $l = 1, \dots, L$ maintaining only current $\mathbf{R}_l, \mathbf{S}_l$, and compute and store the result $\mathbf{Y}_l$. This way, tensors $\mathbf{R}, \mathrm{PS}(\mathbf{R}) \in \mathbb{R}^{L \times d \times M}$ are not stored in memory, resulting in $O(L)$ time complexity and $O(L(d+M)+dM)$ memory complexity. [15] also propose a similar computation of gradients through (2-3); see Appendix D for more discussion.

Alternatively, [9] employ a parallel prefix-sum algorithm [18, 36], which, for a tensor $\mathbf{Z} \in \mathbb{R}^{L \times \cdots}$, finds $\mathrm{PS}(\mathbf{Z})$ in $O(\log L)$ parallel time and $O(L)$ memory. Applying this algorithm for $\mathrm{PS}(\mathbf{R}), \mathrm{PS}(\mathbf{S})$ 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 [23] .

Linear self-attention is a part of *Performers* [9] – efficient Transformers. Below we outline Performer architecture [9] for language modelling, while our analysis can be applicable in broader setups.

Let $\mathbf{p} \in \Sigma^L$ be an input sequence of length L, where Σ is a finite alphabet. By $\mathrm{emb}(\mathbf{p}_l, l) \in \mathbb{R}^{d_{model}}$, $1 \le l \le L$, denote a linear combination of the $\mathbf{p}_l$ token's learned embedding and positional embedding of l's position (sinusoids with different frequencies, as in [35]). Then Performer is defined as a parametrized mapping from $\mathbf{X}^{(0)} = (\mathrm{emb}(\mathbf{p}_l, l))_{l=1}^L \in \mathbb{R}^{L \times d_{model}}$ into $\mathbf{X}^{(out)} \in \mathbb{R}^{L \times |\Sigma|}$ through a sequence of hidden representations $\mathbf{X}^{(1)}, \dots, \mathbf{X}^{(s)} \in \mathbb{R}^{L \times d_{model}}$. For each $1 \le r \le s$, $\mathbf{X}^{(r)}$ is obtained from $\mathbf{X}^{(r-1)}$ 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 ($d_{model} = kd$) plus a skip connection. The second part is a feedforward network applied independently to each element of the sequence. See a formal definition of the architecture in Appendix A.

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

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

3 Low-memory back-propagation algorithm

3.1 Compact notation for Performer

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

$$\mathbf{T}^{(r-1)}, \mathbf{\Gamma}^{(r-1)} = F^{(r)}(\mathbf{X}^{(r-1)}; \theta), \ \mathbf{U}^{(r-1)} = \mathrm{PS}(\mathbf{T}^{(r-1)}), \ \mathbf{X}^{(r)} = G^{(r)}(\mathbf{U}^{(r-1)}, \mathbf{\Gamma}^{(r-1)}; \theta). \tag{5}$$

Here $\theta \in \mathbb{R}^{n_{param}}$ is a set of all parameters, $\mathbf{T}^{(r-1)}, \mathbf{U}^{(r-1)} \in \mathbf{R}^{L \times D_1}$ and $\mathbf{\Gamma}^{(r-1)} \in \mathbf{R}^{L \times D_2}$ are the following matrices (see Figure 1a for an illustration). **(a)** $\mathbf{T}^{(r-1)}$ is a matrix of representations which are passed into the prefix-sum operator. That is, for each $1 \le l \le L$, $\mathbf{T}_l^{(r-1)}$ is a concatenation of $g(\mathbf{K}_l)$ and flattened $\mathbf{V}_l \times g(\mathbf{K}_l)^\top$ for all attention heads computed at the rth step. Consequently, $D_1 = M(d+1)k$. **(b)** For each $1 \le l \le L$, $\mathbf{U}_l^{(r-1)}$ is a concatenation of all corresponding $\mathbf{S}_l$ and flattened $\mathbf{R}_l$ – results of the prefix sum (Eq. 2) inside each self-attention head at rth layer. **(c)** $\mathbf{\Gamma}^{(r-1)}$ is a matrix of representations which skip the prefix sum. For each $1 \le l \le L$, $\mathbf{\Gamma}_l^{(r-1)}$ is a concatenation of $\mathbf{X}_l^{(r-1)}$ and $g(\mathbf{Q}_l)$ for each attention head $1 \le j \le k$ (3). Therefore, $D_2 = Mk + d_{model}$.

$F^{(r)}$ and $G^{(r)}$ are functionals depending on a set of model's parameters θ. That is, they take subsets of θ corresponding to rth layer weights. $F^{(r)}$ is responsible for constructing $\mathbf{T}^{(r-1)}$ and $\mathbf{\Gamma}^{(r-1)}$ –

Algorithm 1 Low-memory forward-backward pass. See Algorithm 2 for updateProc. Compared to notation from the text, redundant indices are dropped and tensor names are reused here and in Algorithm 2.

Input: $\mathbf{p} \in \Sigma^L$,
$\theta \in \mathbb{R}^{n_{param}}$,
$C \in \mathbb{N}$.
Output: loss $\mathcal{L}$,
gradient $\nabla_\theta \mathcal{L}$.
Set $\mathcal{L} := 0$;
Set $\mathcal{U} := \mathbf{0}_{s \times D_1}$;
for $n = 1$ **to** N **do**
  updateProc(n, False);
end for
Set $\nabla_\theta \mathcal{L} := \mathbf{0}_{n_{param}}$;
Set $\mathcal{G} := \mathbf{0}_{s \times D_1}$;
for $n = N$ **to** 1 **do**
  updateProc(n, True);
end for
Return $\mathcal{L}$, $\nabla_\theta \mathcal{L}$.

Algorithm 2 updateProc procedure.

Input: $n \in \mathbb{N}$, binary flag onBP .
if onBP **then** Initialize $\Phi := 0$; **end if**
$\mathbf{X} := (\text{emb}(\mathbf{p}_{A_n+l}, A_n + l))_{l=1}^{B_n}$;
for $r = 1$ **to** s **do**
  Compute $\mathbf{T}, \mathbf{\Gamma} := F^{(r)}(\mathbf{X}; \theta)$;
  if onBP **then** Update $\mathcal{U}_r -= \sum_{l=1}^{B_n} \mathbf{T}_l$; **end if**
  Set $\mathbf{U} := \mathbf{1}_{B_n} \mathcal{U}_r^\top + \text{PS}(\mathbf{T})$, $\mathbf{X} := G^{(r)}(\mathbf{U}, \mathbf{\Gamma}; \theta)$;
  if onBP **then**
    Update $\Phi += \mathcal{G}_r^\top \mathbf{U}_{B_n}$;
  else
    Update $\mathcal{U}_r := \mathbf{U}_{B_n}$;
  end if
end for
Set $\mathcal{L}^{(upd)} := \mathcal{L}^{(n)}(\mathbf{X}\mathbf{W}^{(out)} + \mathbf{b}^{(out)})$;
if onBP **then**
  Update $\Phi += \mathcal{L}^{(upd)}$;
  Compute $\nabla_\theta \Phi, \nabla_{\mathcal{U}} \Phi$ through auto-differentiation;
  Update $\nabla_\theta \mathcal{L} += \nabla_\theta \Phi, \quad \mathcal{G} := \nabla_{\mathcal{U}} \Phi$;
else
  Set $\mathcal{L} += \mathcal{L}^{(upd)}$;
end if

representations preceding prefix sum, while $G^{(r)}$ finalizes multi-head self-attention and includes the feed-forward block. Importantly, $F^{(r)}$ and $G^{(r)}$ are applied **rowwise**, i.e. (5) can be rewritten as

$$\forall 1 \leq l \leq L : \mathbf{T}_l^{(r-1)}, \mathbf{\Gamma}_l^{(r-1)} = F^{(r)}(\mathbf{X}_l^{(r-1)}; \theta), \quad \mathbf{X}_l^{(r)} = G^{(r)}(\mathbf{U}_l^{(r-1)}, \mathbf{\Gamma}_l^{(r-1)}; \theta). \tag{6}$$

Hence, **the only place where the signal is propagated across the sequence is prefix sum in (5)**.

The representation (5) encapsulates architecture details of Performer inside $\{F^{(1)}, G^{(1)}, \ldots, F^{(s)}, G^{(s)}\}$. 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 [38, 35], adding a stabilizing gating mechanism [26], weight sharing across layers [19] or reversible Transformer layers [17].

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

$$\mathbf{X}^{(r,n)} = (\mathbf{X}_{A_n+l}^{(r)})_{l=1}^{B_n} \in \mathbb{R}^{B_n \times d_{model}}, \quad \mathbf{T}^{(r,n)} = (\mathbf{T}_{A_n+l}^{(r)})_{l=1}^{B_n},$$
$$\mathbf{U}^{(r,n)} = (\mathbf{U}_{A_n+l}^{(r)})_{l=1}^{B_n} \in \mathbb{R}^{B_n \times D_1}, \quad \mathbf{\Gamma}^{(r,n)} = (\mathbf{\Gamma}_{A_n+l}^{(r)})_{l=1}^{B_n} \in \mathbb{R}^{B_n \times D_2},$$

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

$$\mathbf{T}^{(r-1,n)}, \mathbf{\Gamma}^{(r-1,n)} = F^{(r)}(\mathbf{X}^{(r,n)}; \theta), \quad \mathbf{U}^{(r-1,n)} = \mathbf{1}_{B_n} \times (\mathbf{U}_{B_{n-1}}^{(r-1,n-1)})^\top \tag{7}$$

$$+\text{PS}(\mathbf{T}^{(r-1,n)}), \quad \mathbf{X}^{(r,n)} = G^{(r)}(\mathbf{U}^{(r-1,n)}, \mathbf{\Gamma}^{(r-1,n)}; \theta). \tag{8}$$

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

Now, instead of iterating over $r = 1, \ldots s$ and computing (5) 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 (7-8).

As can be deduced from the (7-8), we only need to maintain the current value of $(\mathbf{U}_{B_{n-1}}^{(r-1,n-1)})_{r=1}^{s} \in \mathbb{R}^{s\times D_1}$ in the outer iteration over n.

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

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 $\mathbf{p} \in \Sigma^L$ storage, which is $O(L)$ in principle, however the constant is negligibly small. For instance, if $\mathbf{p}$ 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 $\nabla_\theta \mathcal{L}$ of the loss function with respect to parameters θ. One can just perform automatic differentiation [11] (implemented in Tensorflow [1] and Pytorch [27]) 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 $\theta^{(1)} = \dots = \theta^{(N)} = \theta$ be results of a symbolic "identity operation" performed on θ, so that for all $1 \le n \le N$, $\theta^{(n)}$ is used instead of θ in (7-8). Then the total gradient of θ has the form $\nabla_\theta \mathcal{L} = \nabla_{\theta^{(1)}}\mathcal{L} + \dots + \nabla_{\theta^{(N)}}\mathcal{L}$. In Appendix C we derive an expression for $\nabla_{\theta^{(n)}}\mathcal{L}$, $1 \le n \le N$. Namely, denote $\mathcal{G}^{(n)} = \nabla_{\mathcal{U}^{(n)}}\mathcal{L}$, then $\nabla_{\theta^{(n)}}\mathcal{L} = \nabla_{\theta^{(n)}}\Phi^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}, \mathcal{G}^{(n)})$, where $\Phi^{(n)} : \mathbb{R}^{n_{param}} \times \mathbb{R}^{s\times D_1} \times \mathbb{R}^{s\times D_1} \to \mathbb{R}$,

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

In $\Phi^{(n)}$'s definition, $\mathbf{X}^{(out,n)}$ and $\mathcal{U}^{(n)} = (\mathbf{U}_{B_n}^{(r-1,n)})_{r=1}^{s}$ are results of (7-8) iteration over $r = 1, \dots, s$ with parameters $\theta = \theta^{(n)}$ and $(\mathbf{U}_{B_{n-1}}^{(r-1,n-1)})_{r=1}^{s}$ equal to $\Phi^{(n)}$'s second argument $\mathcal{U}^{(n-1)}$. Gradient $\nabla_{\theta^{(n)}}\Phi^{(n)}$ 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 $n = N, \dots, 1$ and to maintain values of $\mathcal{U}^{(n)}, \mathcal{G}^{(n)}$. $\mathcal{U}^{(N)}$ is known after the end of the forward pass, and

$$\forall 1 \le n \le N: \quad \mathcal{U}^{(n-1)} = \mathcal{U}^{(n)} - \sum_{l=1}^{B_n} (\mathbf{T}_l^{(r-1,n)})_{r=1}^{s}. \tag{9}$$

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

$$\mathcal{G}^{(n-1)} = \nabla_{\mathcal{U}^{(n-1)}}\Phi^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}, \mathcal{G}^{(n)}). \tag{10}$$

By a single auto-differentiation through $\Phi^{(n)}$ we can compute $\nabla_{\theta^{(n)}}\mathcal{L} = \nabla_{\theta^{(n)}}\Phi^{(n)}$ and the update (10). If $\mathbf{w}$ is some vector of length B_n and h is some scalar function of $\mathbf{v} = \mathrm{PS}(\mathbf{w})$, then for all $1 \le l \le B_n : \nabla_h \mathbf{w}_l = \sum_{l'=t}^{B_n} \nabla_h \mathbf{v}_{l'}$. In other words, the gradient through $\mathrm{PS}(\cdot)$ is another prefix sum computed backwards. Hence, auto-differentiation through $\Phi^{(n)}$ takes the same parallel time $O(\log C)$, serial time $O()$ 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 $\mathbf{p}$ storage. A full description of the forward-backward pass is presented in Algorithm 1. Figure 1b 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 $\mathbf{p}$ 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 (7-8). 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 $B_n sM(d+1)k$ "add" operations. Hence, we conclude that **Algorithm 1 requires as many FLOPs as two forward and one backward pass through (7-8) for the whole sequence** $\mathbf{p}$ plus $LsM(d+1)k = LsMd_{model} + LsMk$ 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 (11-15, Appendix A) alone requires $3Lsd_{model}^2 + 2Lsd_{model}d_{ff} = 11Lsd_{model}^2$ FLOPs. This is much bigger than $LsMd_{model} + LsMk$, since M is much smaller than d_{model} in practice [9, 15].

Since the back-propagation takes roughly 5 times more FLOPs than the forward pass [11], we conclude that **memory efficiency of Algorithm 1 results in a small constant-time increase in FLOPs**. FLOPs affect energy consumption [37], 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 $2sd_{model}(M+1)$ floats for storing $\mathcal{U}, \mathcal{G}$. For comparison, the subset of θ corresponding to dense layers in self-attention and feed-forward blocks (11-15), occupies $3sd_{model}^2 + 2sd_{model}d_{ff} = 11sd_{model}^2$ floats. Again, this is much bigger than $2sd_{model}(M+1)$, since M is much smaller than d_{model} in practice.

For discussion of related work, including compatibility with other memory-optimization techniques, comparison with [15] and extension to Transformers with dropout, please see Appendix B.

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 [9, 15, 33]. 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 $g(\mathbf{x}) = (\mathbf{x}_i^2)_{i=1}^d$ 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 [27]. 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 Σ^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

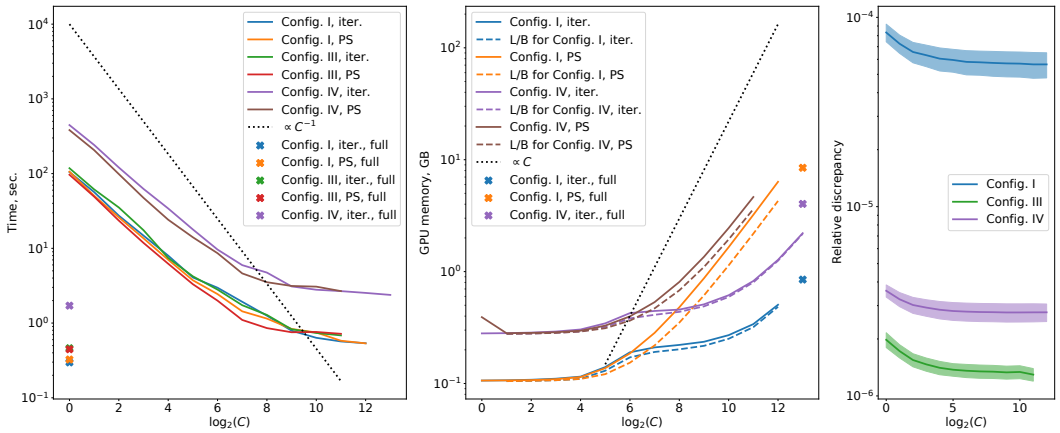

Figure 2: Benchmarks of Algorithm 1. All plots are averaged over 10 seeds. "iter." stands for iterative computation of (2-3), while "PS" is for explicit prefix sum computation in (2). We omit time and memory for big values of C in "Config. IV, PS" and "Config. IV, full" setups, because these led to memory overflow. Analogous results for configuration II can be found in Appendix E. **(Left)** Time dependence on C. Crosses indicate horizontal time levels for corresponding full memory-inefficient methods. The dotted line indicates $\propto C^{-1}$ tangent in logarithmic scale. **(Middle)** Memory dependence on C. Again, crosses are for horizontal levels of full-sequence methods and the dotted line indicates $\propto C$ tangent. We do not report curves for config. III, because they completely match curves for config. IV, which is natural, since s, d_{model} are the same for both configurations. "L/B" stands for a memory lower bound computed by processing input of length C. **(Right)** Relative gradient discrepancy as a function of C and standard errors. Evaluated on random inputs over Σ^L.

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 $\mathbf{p}$ storage. The serial time complexity for Performer is reported for the version with iterative $\mathrm{PS}(\cdot)$ computation (as in [15]), while the parallel time is reported for the parallel prefix sum (as in [9]). For both methods, memory complexity is the same, but the constant is smaller for the iterative version.

Model	Serial time	Parallel time	Memo -ry
RNN	$O(L)$	$O(L)$	$O(L)$
Residual NN	$O(L)$	$O(L)$	$O(L)$
Transformer	$O(L^2)$	$O(\log L)$	$O(L^2)$
Performer	$O(L)$	$O(\log L)$	$O(L)$
Our alg.	$O(L)$	$O(\frac{L}{C}\log C)$	$O(C)$
Ours, $C=1$	$O(L)$	$O(L)$	$\boldsymbol{O(1)}$

Table 2: Time per iteration (sec., averaged over 1000 iterations) and peak GPU memory (Gb). When using small values of C, we relate the remaining memory to a storage of parameters θ and optimizer's state.

Setup, L, C	Time	Memory
CT, 8192, full	0.3008	0.938
CT, 8192, 4096	0.5372	0.595
CT, 8192, 2048	0.6002	**0.436**
PTB, 1024, full	0.1377	0.300
PTB, 1024, 512	0.2526	0.257
PTB, 1024, 256	0.3060	**0.231**
ENW, 4096, full	0.4598	1.513
ENW, 4096, 2048	0.7922	1.085
ENW, 4096, 1366	0.8654	**0.909**

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

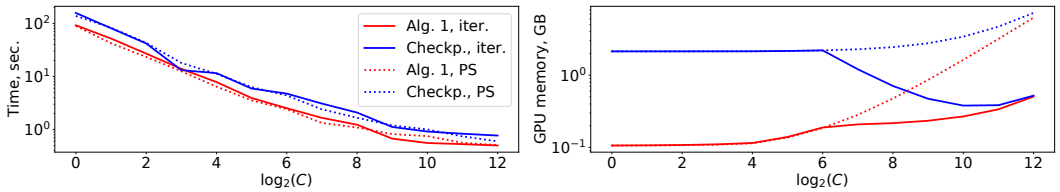

Figure 3: Algorithm 1 compared to checkpointing of $\{\mathcal{U}^{(n)}\}_{1\leq n\leq N}$. Time and memory plots.

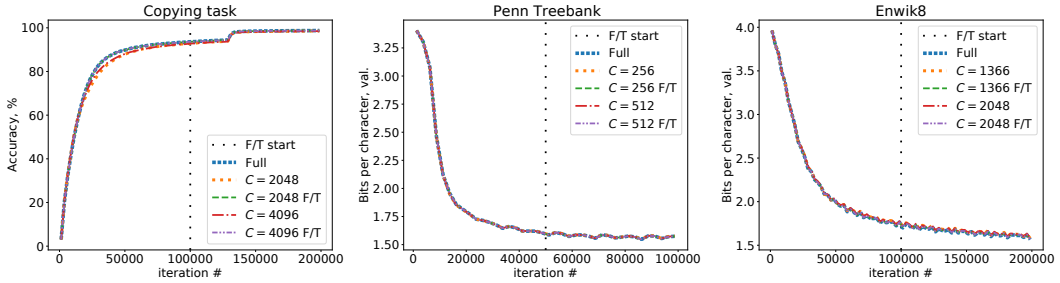

Figure 4: Learning curves for three language modelling setups. We report accuracy on a newly generated data samples for Copying task, and Bits Per Character (BPC) metric on validation examples for Penn Treebank and Enwik8. F/T stands for "fine-tuning". **All curves are roughly the same and result in 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.

D). 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 Σ^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** θ. 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 [12]. By checkpointing in this context we understand storing $\{\mathcal{U}^{(n)}\}_{1\leq n\leq N}$ during the forward pass and reusing them during the backward pass instead of recomputing. This results in a small FLOPs decrease since $\mathcal{U}^{(n)}$, while memory scales as $O(L/C)$. We use config. I 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* $\|\nabla_\theta^{(C)}\mathcal{L} - \nabla_\theta^{(full)}\mathcal{L}\|_2 / \|\nabla_\theta^{(full)}\mathcal{L}\|_2$ between the gradient $\nabla_\theta^{(C)}$ produced by Algorithm 1, and the gradient

Table 3: Time per iteration (sec., averaged over 1000 iterations) and peak GPU memory (Gb). When using small values of C, we relate the remaining memory to a storage of θ and optimizer's state.

Setup, L, C	BPC, F/T	BPC, no F/T	F/T mem., Alg. 1	F/T mem., full comp.	F/T time, Alg. 1	F/T time, full alg.
PTB, 1024, 16	**1.4263**	1.6544	**0.1077**	0.1834	0.8938	0.0426
ENW 4096, 64	**1.5642**	1.6141	**0.4276**	0.8049	1.1142	0.2045

$\nabla_\theta^{(full)}\mathcal{L}$ 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 [17, 15], sampling inputs as $0\omega 0\omega$, where ω 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 E.

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 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 works 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 [23].

Negative societal impacts. This work studies Performers – efficient Transformers. Transformers in general can have the following negative societal impacts: large carbon dioxide emissions [32], privacy and data leak vulnerabilities [5], bias and fairness issues and malicious misuse [4, 3].

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[3] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? . In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, page 610–623, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, 2020.

[5] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models. *CoRR*, abs/2012.07805, 2020.

[6] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174, 2016.

[7] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019.

[8] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, October 2014. Association for Computational Linguistics.

[9] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with Performers. In *International Conference on Learning Representations*, 2021.

[10] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

[11] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2008.

[12] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[13] Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *CoRR*, abs/1606.08415, 2016.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[15] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5156–5165. PMLR, 13–18 Jul 2020.

[16] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[17] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[18] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4):831–838, October 1980.

[19] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite BERT for self-supervised learning of language representations. In *International Conference on Learning Representations*, 2020.

[20] Rui Li, Chenxi Duan, and Shunyi Zheng. Linear attention mechanism: An efficient attention for semantic segmentation. *arXiv preprint arXiv:2007.14902*, 2020.

[21] Matt Mahoney. Large text compression benchmark, 2009.

[22] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[23] Eric Martin and Chris Cundy. Parallelizing linear recurrent neural nets over sequence length. In *International Conference on Learning Representations*, 2018.

[24] 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 the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 48–53, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[25] Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 1–9, Brussels, Belgium, October 2018. Association for Computational Linguistics.

[26] Emilio Parisotto, Francis Song, Jack Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant Jayakumar, Max Jaderberg, Raphaël Lopez Kaufman, Aidan Clark, Seb Noury, Matthew Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing transformers for reinforcement learning. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 7487–7498. PMLR, 13–18 Jul 2020.

[27] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[28] Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *arXiv*, 2003.05997, 2020.

[29] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv*, 1910.01108, 2020.

[30] Zhuoran Shen, Mingyuan Zhang, Shuai Yi, Junjie Yan, and Haiyu Zhao. Factorized attention: Self-attention with linear complexities. *CoRR*, abs/1812.01243, 2018.

[31] 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):1929–1958, 2014.

[32] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in NLP. *CoRR*, abs/1906.02243, 2019.

[33] Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena : A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2021.

[34] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *arXiv*, 9.20006732, 2020.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc., 2017.

[36] Uzi Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques. 2010.

[37] Zhanghao Wu*, Zhijian Liu*, Ji Lin, Yujun Lin, and Song Han. Lite transformer with long-short range attention. In *International Conference on Learning Representations*, 2020.

[38] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture, 2020.

[39] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations*, 2020.

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] See Conclusion.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Conclusion.
   (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] The linear self-attention and Performer are well defined in Section 2.
   (b) Did you include complete proofs of all theoretical results? [Yes] The correctness and complexity of the proposed algorithm are justified in Section 3.
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)? [Yes] See supplemental materials.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 4.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] E.g. see Figure 4 (right)
   (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 Section 4.

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] We use PyTorch, Penn Treebank dataset and Enwik8 dataset, all these assets are cited.
   (b) Did you mention the license of the assets? [N/A] The software we use is standard, open-source and can be freely used for research purposes.
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A] There are no new assets used in this work.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A] We only use open-source, publicly available sortware.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A] We only use standard datasets used many times in previous literature.
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] We didn't use crowdsourcing or experiments with human subjects.
   (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]