A Performer architecture details

We define the Performer architecture formally as follows. \( X^{(out)} = X^{(s)}W^{(out)} + b^{(out)} \) and for each \( 1 \leq r \leq s \):

\[
\begin{align*}
H^{(r-1)} &= \text{LN}(\text{MultiHead-Att}(X^{(r-1)}) + X^{(r-1)}), \quad (11) \\
X^{(r)} &= \text{LN}(\text{FFN}(H^{(r-1)})) + H^{(r-1)}, \quad \text{where} \\
\text{MultiHead-Att}(X) &= [H^{(1)} \ldots H^{(k)}], \\
\forall j \leq k : H^{(j)} &= \text{Att}(\overline{X}W_{Q}^{(j)}, \overline{X}W_{K}^{(j)}, \overline{X}W_{V}^{(j)}), \\
\text{FFN}(H) &= \text{GeLU}(\overline{H}W^{(1)} + b^{(1)})W^{(2)} + b^{(2)}. \quad (15)
\end{align*}
\]

Here \( k \) is the number of attention heads (\( d_{\text{model}} = kd \)). \( W^{(out)} \in \mathbb{R}^{d_{\text{model}} \times |\Sigma|}, b^{(out)} \in \mathbb{R}^{1 \times |\Sigma|}, W^{(1)} \in \mathbb{R}^{d_{\text{model}} \times d}, b^{(1)} \in \mathbb{R}^{1 \times d}, W^{(2)} \in \mathbb{R}^{d \times d_{\text{model}}}, b^{(2)} \in \mathbb{R}^{1 \times d_{\text{model}}}, W_{Q}^{(j)}, W_{K}^{(j)}, W_{V}^{(j)} \in \mathbb{R}^{d_{\text{model}} \times d} \) are trainable parameters (separate for each instance of MultiHead-Att, FFN), “+” is broadcast rowwise when biases are added and LN is layer normalization [2], which is applied rowwise and depends on additional trainable parameters. GeLU denotes Gaussian error Linear Unit [10], which is applied elementwise.

B Derivation of Gradient Expressions

\( \theta^{(n)} \) doesn’t affect terms \( \mathcal{L}^{(1)}(X^{(out,1)}), \ldots, \mathcal{L}^{(n-1)}(X^{(out,n)}) \), so corresponding gradients are zero:

\[
\nabla_{\theta^{(n)}} \mathcal{L} = \nabla_{\theta^{(n)}} \sum_{n'=n}^{N} \mathcal{L}^{(n')}(X^{(out,n')}).
\]

Similarly, \( \mathcal{U}^{(n)} \) does not affect \( \mathcal{L}^{(1)}, \ldots, \mathcal{L}^{(n)} \), so

\[
\mathcal{G}^{(n)} = \nabla_{\mathcal{U}^{(n)}} \mathcal{L} = \nabla_{\mathcal{U}^{(n)}} \sum_{n'=n+1}^{N} \mathcal{L}^{(n')}(X^{(out,n')}),
\]

In particular,

\[
\mathcal{G}^{(N)} = \nabla_{\mathcal{U}^{(N)}} \mathcal{L} = 0 \times D_4.
\]

For all \( 1 \leq n < n' \leq N \), \( \theta^{(n)} \) and \( \mathcal{U}^{(n-1)} \) affect \( \mathcal{L}^{(n')} \) only through \( \mathcal{U}^{(n)} \), so according to the chain rule

\[
\nabla_{\theta^{(n)}} \sum_{n'=n+1}^{N} \mathcal{L}^{(n')}(X^{(out,n')}) = \sum_{r=1}^{s} \frac{\partial \mathcal{U}^{(n)}}{\partial \theta^{(n)}} \top \times \nabla_{\mathcal{U}^{(n)}} \sum_{n'=n+1}^{N} \mathcal{L}^{(n')}(X^{(out,n')})
\]

\[
= \sum_{r=1}^{s} \frac{\partial \mathcal{U}^{(n)}}{\partial \theta^{(n)}} \top \times \nabla_{\mathcal{U}^{(n)}} \mathcal{L},
\]

\[
\forall 1 \leq r' \leq s : \nabla_{\mathcal{U}^{(r'-1)}} \sum_{n'=n+1}^{N} \mathcal{L}^{(n')}(X^{(out,n')}) = \sum_{r=1}^{s} \frac{\partial \mathcal{U}^{(n)}}{\partial \mathcal{U}^{(r'-1)}} \top \times \nabla_{\mathcal{U}^{(r'-1)}} \sum_{n'=n+1}^{N} \mathcal{L}^{(n')}(X^{(out,n')})
\]

\[
= \sum_{r=1}^{s} \frac{\partial \mathcal{U}^{(n)}}{\partial \mathcal{U}^{(r'-1)}} \top \times \nabla_{\mathcal{U}^{(r'-1)}} \mathcal{L},
\]

where \( \frac{\partial}{\partial \theta} \) denotes Jacobian matrices. Further, for all \( 1 \leq r \leq s \):

\[
\frac{\partial \mathcal{U}^{(n)}}{\partial \square} \times \nabla_{\mathcal{U}^{(n)}} \mathcal{L} = \nabla_{\square} \left( \left\langle \mathcal{U}^{(n)} \top \langle \nabla_{\mathcal{U}^{(n)}} \mathcal{L} \rangle \right\rangle \right),
\]

where \( \square \in \{ \theta^{(n)} \cup \{ \mathcal{U}^{(r'-1)} \}_{1 \leq r' \leq s} \} \) denotes a stop-gradient operator, i.e. gradients are not propagated inside brackets and the argument is considered as constant.

14
We conclude that
\[
\nabla_{\theta(n)} \mathcal{L} = \nabla_{\theta(n)} \mathcal{L}^{(n)}(X^{(out,n)}) + \nabla_{\theta(n)} \sum_{n' = n+1}^N \mathcal{L}^{(n')}(X^{(out,n')}) = \nabla_{\theta(n)} \mathcal{L}^{(n)}(X^{(out,n)}) \\
+ \sum_{r=1}^s \frac{\partial \mathcal{U}^{(n)}_r}{\partial \theta(n)} \times \nabla_{\mathcal{U}^{(n)}_r} \mathcal{L}
\]
\[
= \nabla_{\theta(n)} \left( \mathcal{L}^{(n)}(X^{(out,n)}) + \sum_{r=1}^s [\mathcal{U}^{(n)}_r]^\top \langle \nabla_{\mathcal{U}^{(n)}_r} \mathcal{L} \rangle \right) = \nabla_{\theta(n)} \Phi^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}_n, \nabla_{\mathcal{U}^{(n)}_r} \mathcal{L}) \\
= \nabla_{\theta(n)} \Phi^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}_n, \mathcal{G}^{(n)}),
\]
\[
\forall 1 \leq r' \leq s \; : \; \mathcal{G}^{(n-1)}_r = \nabla_{\mathcal{U}^{(n-1)}_{r'}} \mathcal{L} = \nabla_{\mathcal{U}^{(n-1)}_{r'}} \mathcal{L}^{(n)}(X^{(out,n)}) + \nabla_{\mathcal{U}^{(n-1)}_{r'}} \sum_{n'=n+1}^N \mathcal{L}^{(n')}(X^{(out,n')}) \\
= \nabla_{\mathcal{U}^{(n-1)}_{r'}} \mathcal{L}^{(n)}(X^{(out,n)}) + \sum_{r=1}^s \frac{\partial \mathcal{U}^{(n)}_r}{\partial \mathcal{U}^{(n-1)}_{r'}} \times \nabla_{\mathcal{U}^{(n)}_r} \mathcal{L}
\]
\[
= \nabla_{\mathcal{U}^{(n-1)}_{r'}} \left( \mathcal{L}^{(n)}(X^{(out,n)}) + \sum_{r=1}^s [\mathcal{U}^{(n)}_r]^\top \langle \nabla_{\mathcal{U}^{(n)}_r} \mathcal{L} \rangle \right) \\
= \nabla_{\mathcal{U}^{(n-1)}_{r'}} \Phi^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}_n, \nabla_{\mathcal{U}^{(n)}_r} \mathcal{L}) \\
= \nabla_{\mathcal{U}^{(n-1)}_{r'}} \Phi^{(n)}(\theta^{(n)}, \mathcal{U}^{(n-1)}_n, \mathcal{G}^{(n)}),
\]
where the second chain of equalities is equivalent to (10).

C  Efficient “Block” Computation of (2-3)

Denote \( \bar{Q} = (g(Q_i))_{i=1}^L \), \( \bar{K} = (g(K_i))_{i=1}^L \), \( N = (R_l \times \bar{Q}_l)_{i=1}^L \), \( D = (S_l \times \bar{Q}_l)_{i=1}^L \). [13] propose the following algorithm for computation of (2-3). Initialize buffers \( \text{curR} = 0_{d \times M}, \text{curS} = 0_{M} \), iterate over \( l = 1, \ldots, L \) and compute

\[
\text{curR} := \text{curR} + V_l \times \bar{K}_l^\top; \\
\text{curS} := \text{curS} + \bar{K}_l; \\
N_l := \text{curR} \times \bar{Q}_l; \\
D_l := \text{curS}^\top \times \bar{Q}_l; \\
Y_l := N_l / D_l.
\]

This way, the 3D tensor \( R \in \mathbb{R}^{L \times d \times M} \) is not stored in memory explicitly, resulting in \( O(L) \) time and \( O(L(d + M) + dM) \) memory complexity. In order to have the same memory consumption during back-propagation, [13] propose the following routine. Keep buffers \( \text{curR}, \text{curS} \) as the result of forward pass, and initialize gradient buffers \( \text{gradR} = 0_{d \times M}, \text{gradS} = 0_{M} \). Assuming that \( \nabla_N \mathcal{L} \in \mathbb{R}^{L \times d}, \nabla_D \mathcal{L} \in \mathbb{R}^L \) are computed using automatic differentiation, iterate in a backward direction \( l = L, \ldots, 1 \) and compute

\[
\nabla_{\bar{Q}_l} \mathcal{L} := (\nabla_D \mathcal{L}) \cdot \text{curS} + \text{curR}^\top \times \nabla_N \mathcal{L}; \\
\text{curR} := \text{curR} - V_l \times \bar{K}_l^\top; \\
\text{curS} := \text{curS} - \bar{K}_l; \\
\text{gradR} := \text{gradR} + (\nabla_N \mathcal{L}) \times \bar{Q}_l^\top; \\
\text{gradS} := \text{gradS} + (\nabla_D \mathcal{L}) \cdot \bar{Q}_l; \\
\nabla_{\bar{V}_l} \mathcal{L} := \text{gradR} \times \bar{K}_l; \\
\nabla_{\bar{K}_l} \mathcal{L} := \text{gradR}^\top \times \bar{V}_l.
\]
In practice, the described algorithm works slowly when implemented in pure PyTorch, because \( l \) is iterated one-by-one: \cite{13} use low-level CUDA extensions to make the algorithm practical. Instead, we propose a “block” version, when we iterate through blocks of \( l \) of a small size \( C \) (we use \( C = 64 \)). In each block we use explicit prefix sums on inputs of length \( C \) to find \( \mathbf{Y}_{t+l+C-1} \), using the maintained front \( \mathbf{curR}, \mathbf{curS} \). The formal algorithm is as follows. Initialize buffers \( \mathbf{curR} = 0_{l \times M} \), \( \mathbf{curS} = 0_{M} \). For simplicity assuming that \( C \) divides \( L \) (extension for an opposite case is straightforward), iterate over \( l = 1, C + 1, \ldots, L - C + 1 \) and compute

\[
\text{blockR} := \text{PS}((\mathbf{V}_{l+t-1} \times \mathbf{K}_{t+t-1}^\top)^C_{l=1});
\]
\[
\text{blockR} := (\mathbf{curR} + \text{blockR}_{l'})^C_{l'=1};
\]
\[
\text{blockS} := \text{PS}((\mathbf{K}_{l+t-1})^C_{l=1});
\]
\[
\text{blockS} := (\mathbf{curS} + \text{blockS}_{l'})^C_{l'=1};
\]
\[
\mathbf{curR} := \text{blockR} C;
\]
\[
\mathbf{curS} := \text{blockS} C;
\]
\[
\mathbf{N}_{l+t+C-1} := (\text{blockR}_{l'} \times \mathbf{Q}_{l+t-1}^C)_{l'=1};
\]
\[
\mathbf{D}_{l+t+C-1} := (\text{blockS}_{l'} \times \mathbf{Q}_{l+t-1}^C)_{l'=1};
\]
\[
\mathbf{Y}_{t+l+C-1} := (\mathbf{N}_{l+t-1} / \mathbf{D}_{l+t-1})^C_{l'=1}.
\]

In the “block” version, the number of outer sequential iterations is reduced to \( L / C \), resulting in \( O((L / C) \log C) \) parallel time complexity, when the logarithmic parallel algorithm is used to compute prefix sums \cite{16,17}. In our experiments, we use torch.cumsum, which works fast in practice. The memory complexity of the algorithm is \( O(L(d + M) + CdM) \), where the second term is for storing \( \text{blockR} \). Assuming that \( C \) is a small constant (\( C = O(1) \)), we conclude that the “block” version has \( O(L(d + M) + dM) \) memory and \( O(L) \) time complexity – same as the algorithm of \cite{18}. As for hidden constants in complexity estimates, the constant inside \( O(L) \) time complexity is reduced at the cost of increasing constant of the “small” \( dM \) term in the memory complexity (when \( d, M \ll L \)), making the “block” iterative algorithm a practical choice for computing \( \mathbf{23} \).

We further show how to back-propagate through \( \mathbf{23} \) in \( O((L / C) \log C) \) time and \( O(L(d + M) + CdM) \) memory. Again, keep buffers \( \mathbf{curR}, \mathbf{curS} \) as the result of forward pass, and initialize gradient buffers \( \mathbf{gradR} = 0_{d \times M} \), \( \mathbf{gradS} = 0_{M} \). Assuming that \( \nabla_{\mathbf{N}} \mathcal{L} \in \mathbb{R}^{L \times d}, \nabla_{\mathbf{D}} \mathcal{L} \in \mathbb{R}^{L} \) are computed using automatic differentiation, iterate in a backward direction \( l = L - C + 1, L - 2C + 1, \ldots, 1 \) and compute

\[
\mathbf{gradR} := \mathbf{gradR} + \sum_{l'=l}^{l+C-1} (\nabla_{\mathbf{N}_{l'}} \mathcal{L}) \times \mathbf{Q}_{l'}^\top;
\]
\[
\mathbf{gradS} := \mathbf{gradS} + \sum_{l'=l}^{l+C-1} (\nabla_{\mathbf{D}_{l'}} \mathcal{L}) \cdot \mathbf{Q}_{l'};
\]
\[
\text{blockgradR} := \text{PS}((\nabla_{\mathbf{N}_{l+t-1}} \mathcal{L}) \times \mathbf{Q}_{l+t-1}^\top)^C_{l'=1};
\]
\[
\text{blockgradS} := (\mathbf{gradR} - \text{blockgradR}_{l'})^C_{l'=1};
\]
Figure 5: Version of Figure 2 for configuration I.

\[
\text{blockgrad} S := \text{PS}(\{(\nabla \mathcal{L}_{l'} - \mathcal{L}) \cdot \tilde{Q}_{l'}\}_{l'=1}^C); \\
\text{blockgrad} S := (\text{grad} S - \text{grad} S_{l'}{l'}=1; \\
\nabla \mathcal{L}_{l+\epsilon-1} := (\text{blockgrad} R_{l'} \times \mathbf{K}_{l+\epsilon-1})_{l'=1}^C; \\
\nabla \mathbf{K}_{l+\epsilon-1} := (\text{blockgrad} R_{l'}^T \times \mathbf{V}_{l+\epsilon-1})_{l'=1}^C.
\]

Finally, it’s easy to see how to use both one-to-one and “block” iterative computation as part of Algorithm 1 to compute the update (7-8). For that, when doing a forward computation for some \(n, r\), initialize \(\text{cur} R, \text{cur} S\) from corresponding subvectors of \(U_{(r-1,n-1)}\), with the rest of the algorithm unchanged. Similarly, during a backward pass for some \(n, r\), initialize \(\text{grad} R, \text{grad} S\) from corresponding subvectors of \(G^{(n)}\) and leave the rest of the iterative back-propagation algorithm unchanged.

**D Additional Experimental Details**

We use 200K, 100K, 200K SGD iterations in the Copying Task, Penn Treebank, Enwik8 setups respectively. We use Adam optimizer [19] with \(\beta_1 = 0.9, \beta_2 = 0.999\) (default configuration used in PyTorch). For the Copying task, we train with a learning rate \(10^{-3}\) for 130K iterations and then decrease the learning rate to \(10^{-4}\). We use a fixed learning rate of \(10^{-4}\) and \(2 \times 10^{-4}\) in Penn Treebank and Enwik8 experiments respectively.

Figure 5 is a version of Figure 2 for the configuration II. We draw the same conclusions to those reported in the main text. Figure 6 is a bigger version of Figure 4 from the main text. Figure 7 reports additional experimental results: Bits Per Character for the Copying Task and train set learning curves for Penn Treebank and Enwik8. Figure 8 is a version of Figure 4, showing a difference (\(\bullet\)) between curves and the “Full” curve. We observe that memory-efficient algorithms result in a negligible loss of performance, which we attribute to numerical effects, accumulating with many thousands of iterations.

![Figure 5: Version of Figure 2 for configuration I.](image_url)
Figure 6: Bigger version of Figure 4.
Figure 7: Bits-per-character learning curve for the Copying task and train-set learning curves for language modelling on Penn Treebank and Enwik8 respectively.
Figure 8: A difference (−) between curves and the “Full” curve from Figure 4.