The appendix includes additional details regarding the algorithm and experiments.

1. Algorithm Details

1.1. Normalizing Queries and Keys

Recall our assumption that the vector norms of $Q_i$ and $K_j$ are unit length. We used a parameter $\tau$ to control the decay rate of attention weights as the similarities change. In this subsection, we provide additional details about how this can be achieved using a slightly modified idea from Neyshabur & Srebro (2015).

Assume that the $\ell_2$ norms of $Q_i$, $K_j$ are bounded by some $\sqrt{\tau}$ for all $i,j = 1, \ldots, n$. Then we can construct new queries and keys as follows:

$$
\hat{Q}_i = \frac{1}{\sqrt{\tau}}[Q_{i,1}, \ldots, Q_{i,d}, \sqrt{\tau - \|Q_i\|_2^2}, 0]; \quad \hat{K}_j = \frac{1}{\sqrt{\tau}}[K_{j,1}, \ldots, K_{j,d}, \sqrt{\tau - \|K_j\|_2^2}]
$$

Note that $\tau \hat{Q}_i^T \hat{K}_j = Q_i^T K_j$ while $\|\hat{Q}_i\|_2 = \|\hat{K}_j\|_2 = 1$.

1.2. Approximating Random Projections in LSH

The main paper describes estimating self-attention using Bernoulli sampling via LSH. To do so, the first step of LSH is computing the hash code using random projections. To compute hash codes of a vector $x$, the hyperplane hash (Charikar, 2002) is used. We use a concatenation of $\tau$ hyperplane hashes to boost the decay of collision probability as similarity decreases.

$$
F : \mathbb{R}^d \rightarrow \{0, 1\}^{m\tau} \quad F(x) = \text{sign}(Px)
$$

where $P \in \mathbb{R}^{(m\tau) \times d}$, $P_{ij} \sim \mathcal{N}(0, 1)$). Here, $m\tau$ hashes are computed at once, then the output vector is partitioned to $m\tau$-dimensional binary hash codes. The time complexity for random project is $O(nm\tau d)$. To efficiently approximate random projections, we follow the construction used in (Andoni et al., 2015). The output of the $m\tau$-dimensional vector is divided into $\frac{m\tau}{d}$ $d$-dimensional vectors and then the hash codes are estimated by

$$
F(x) = \text{concat} \left( \text{sign}(HD_3^1 HD_2^1 HD_1^1 x), \ldots, \text{sign}(HD_3^{m\tau} HD_2^{m\tau} HD_1^{m\tau} x) \right),
$$

where $D_p^{q}$ are diagonal matrices with diagonal entries uniformly sampled from $\{-1, +1\}$, and $H$ is the Hadamard matrix. This approximation reduces the time complexity to $O(nm\tau \log_2(d))$ via the Fast Hadamard Transform.

1.3. Derivation of the Backpropagation scheme

When using expectation of LSH collision as attention weights, the attention output of one query $Q_i$ to keys $K_j$ and associated values $V_j$ for all $j = 1, \ldots, n$ is defined as

$$
Y_i = \sum_{j=1}^{n} \left( 1 - \frac{\arccos(Q_i^T K_j)}{\pi} \right)^\tau V_j
$$

Then, given the gradient of the loss w.r.t. $Y_i$, denoted $\nabla_{Y_i} L$, the goal is to compute the gradient of the loss w.r.t. $Q_i$, denoted $\nabla_{Q_i} L$. We start by computing the $q$-th entry of $\nabla_{Q_i} L$:

$$
\frac{\partial L}{\partial Q_{i,q}} = \sum_{l=1}^{d} \frac{\partial L}{\partial Y_{i,l}} \frac{\partial Y_{i,l}}{\partial Q_{i,q}} = \sum_{l=1}^{d} \frac{\partial L}{\partial Y_{i,l}} \frac{\partial}{\partial Q_{i,l}} \sum_{j=1}^{n} \left( 1 - \frac{\arccos(Q_i^T K_j)}{\pi} \right)^\tau V_{j,l}
$$

(1)
After swapping the order of the two summations, (1) becomes

\[ \frac{d}{dx} \left( 1 - \frac{\arccos(x)}{\pi} \right)^\tau = \frac{\tau (1 - \frac{\arccos(x)}{\pi})^{\tau - 1}}{\pi \sqrt{1 - x^2}} \]

and plug it into (1),

\[ \frac{\partial L}{\partial Q_{i,q}} = \sum_{l=1}^{d} \frac{\partial L}{\partial Y_{i,l}} \sum_{j=1}^{n} \frac{\tau \left( 1 - \frac{\arccos(Q_l^T K_j)}{\pi} \right)^{\tau - 1}}{\pi \sqrt{1 - (Q_l^T K_j)^2}} K_{j,q} V_{j,l} \]

After swapping the order of the two summations, (1) becomes

\[ \frac{\partial L}{\partial Q_{i,q}} = \sum_{j=1}^{n} (\nabla Y_i L)^T V_j \frac{\tau \left( 1 - \frac{\arccos(Q_l^T K_j)}{\pi} \right)^{\tau - 1}}{\pi \sqrt{1 - (Q_l^T K_j)^2}} K_{j,q} \]

Note that only \( K_{j,q} \) is different for different entries of \( \nabla Q_i L \), so we can write it as

\[ \nabla Q_i L = \sum_{j=1}^{n} (\nabla Y_i L)^T V_j \frac{\tau \left( 1 - \frac{\arccos(Q_l^T K_j)}{\pi} \right)^{\tau - 1}}{\pi \sqrt{1 - (Q_l^T K_j)^2}} K_j \]

The term \( \nabla Q L \) in the main text is the matrix form of above expression for \( i = 1, \cdots n \)

\[ \nabla Q L = \left( \left( \nabla YOSO L \right)^T V \right) \odot \left( \left( 1 - \frac{\arccos(Q K^T)}{\pi} \right)^{\tau - 1} \right) \odot \left( \frac{\pi \sqrt{1 - (Q K^T)^2}}{\pi} \right) K \]

where \( \odot \) are element-wise multiplication and division respectively. Note that \( \pi \sqrt{1 - (Q K^T)^2} \) approaches 0 as the similarity score between the query and the key approaches 1 – so to avoid numerical and stability issues, we use the fact that

\[ \frac{1}{2} \left( 1 - \frac{\arccos(x)}{\pi} \right) \leq \frac{1}{\pi \sqrt{1 - x^2}} \text{ for } x \in [-1, 1] \]

and define a lower bound to replace the actual gradient

\[ \hat{\nabla} Q L = \left( \left( \nabla YOSO L \right)^T V \right) \odot \frac{\tau}{2} \left( 1 - \frac{\arccos(Q K^T)}{\pi} \right) K \]

\[ (3) \]

1.4. Alternative Procedure for Approximating Backpropagation

The main paper described a procedure to estimate (3), which uses LSH-based Bernoulli Sampling \( d \) times as a subroutine. The complexity of this procedure is linear w.r.t. sequence length \( n \), which is desirable but the runtime can be large if \( d \) is set to be very large. An alternative described here based on additional assumptions, is linear with respect to \( d \).

The gradient of \( L \) w.r.t. the \( i \)-th row of \( Q \) is written as

\[ \hat{\nabla} Q_i L = \sum_{j=1}^{n} (\nabla Y_i L)^T V_j B(Q, K)_{i,j} \frac{\tau}{2} K_j \]

Note that if \( B(Q, K)_{i,j} \) is zero then the corresponding summation term does not need to be computed. The alternative procedure relies on the sparsity of attention matrices (which we have not exploited so far) to reduce the workload. The procedure is simple: it checks the value of \( B(Q, K)_{i,j} \) and only computes the summation term when \( B(Q, K)_{i,j} = 1 \).

For \( m \) samples, the procedure counts the number of times \( Q_i \) and \( K_j \) collide, and only computes the summation term corresponding to \( Q_i \) and \( K_j \) when at least one collides occurs. Therefore, the runtime is \( O(\text{nnz}(B(Q, K)) (m+d)) \) (counting number of success + computing nonzero terms). In the worst case, \( \text{nnz}(B(Q, K)) = n^2 \), it would be as expensive as dense matrix multiplications in complexity and even worse in practice due to a large memory latency resulting from indirect memory access. However, in practice, \( B(Q, K) \) is generally sparse if \( \tau \) is set sensibly. Further, the first procedure guarantees
a linear complexity scaling of our method for extremely long sequences. As an improvement, one can dynamically select one from these two method based on runtime, then the time complexity is $O(\min\{\text{and}^2, \text{nnz}(B(Q,K))(m + d)\})$.

**Estimating backpropagation based on (2).** To test the effect of using (3) instead of (2) for backpropagation, we developed a similar procedure to estimate (2). We only consider the gradient backpropagation through the non-zero entries of $B(Q,K)$, so the quantity computed is as follows,

$$
\nabla_QL = \left(\left(\tau(1 - \frac{\arccos(QK^T)}{\pi})\right)^{r-1}\right) \odot \left(\frac{\pi}{\sqrt{1 - (QK)^2}}\right) \odot 1_{B(Q,K)}K
$$

where $1_{B(Q,K)}$ is an $n \times m$ matrix whose $(i,j)$ entry is 1 if $B(Q,K)_{ij} \neq 0$ else it is 0. Then, we can use the sparsity of $B(Q,K)$ to save compute steps. The procedure initializes $\nabla_{YOSO}Q_i$ to zero and checks if $Q_i$ and $K_j$ collide in any of $m$ hashes. If so, it computes the weight

$$
w_{ij} = \frac{\left(\left(\tau(1 - \frac{\arccos(Q^T_{ij}K_j)}{\pi})\right)^{r-1}\right) \odot \left(\frac{\pi}{\sqrt{1 - (Q^T_{ij}K_j)^2}}\right) + \epsilon}{B(Q,K)}
$$

where a small $\epsilon$ is introduced to avoid a divide by zero error, and then accumulates $w_{ij}K_j$ to $\nabla_{YOSO}Q_i$. This procedure has the same runtime complexity as the procedure above. The models trained using this procedure for backpropagation computation is denoted as *YOSO-C in the experimental section.

## 2. Experiment Details

### 2.1. Efficiency

We provide a table of runtime and memory consumption for each efficient Transformer method in Table 1. The runtime difference between YOSO and *YOSO is generally less than 20%. In addition to testing the efficiency when the largest batch size for each methods is used, we tested the efficiency when using the same batch size that can fit in memory for all methods. The results are summarized in Table 2. Compared to baselines, the batch size setting appears to have a weaker impact on the efficiency of our method. This indicates that YOSO could potentially be used on memory constrained settings where only small batch sizes are possible.

<table>
<thead>
<tr>
<th>SEQ LENGTH</th>
<th>BS</th>
<th>128 TIME/STD</th>
<th>MEM</th>
<th>BS</th>
<th>256 TIME/STD</th>
<th>MEM</th>
<th>BS</th>
<th>512 TIME/STD</th>
<th>MEM</th>
<th>BS</th>
<th>1024 TIME/STD</th>
<th>MEM</th>
<th>BS</th>
<th>2048 TIME/STD</th>
<th>MEM</th>
<th>BS</th>
<th>4096 TIME/STD</th>
<th>MEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sofmax</td>
<td>512</td>
<td>0.50.000.17</td>
<td>128</td>
<td>1.10.000.43</td>
<td></td>
<td>64</td>
<td>3.00.01.126</td>
<td></td>
<td>16</td>
<td>8.90.05.423</td>
<td></td>
<td>4</td>
<td>29.20.20.1529</td>
<td></td>
<td>1</td>
<td>100.80.72</td>
<td></td>
<td>5794</td>
</tr>
<tr>
<td>Nystromformer</td>
<td>256</td>
<td>1.10.001.32</td>
<td>128</td>
<td>1.80.01.49</td>
<td></td>
<td>64</td>
<td>3.10.03.83</td>
<td></td>
<td>32</td>
<td>5.60.01.149</td>
<td></td>
<td>32</td>
<td>11.00.07.283</td>
<td></td>
<td>32</td>
<td>50.40.10.699</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Longformer</td>
<td>256</td>
<td>1.10.005.27</td>
<td>128</td>
<td>1.30.005.51</td>
<td></td>
<td>64</td>
<td>2.50.00.99</td>
<td></td>
<td>32</td>
<td>8.50.01.197</td>
<td></td>
<td>32</td>
<td>10.10.03.391</td>
<td></td>
<td>32</td>
<td>20.80.08.782</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>Reformer</td>
<td>256</td>
<td>1.10.000.28</td>
<td>128</td>
<td>2.20.005.57</td>
<td></td>
<td>64</td>
<td>4.30.01.118</td>
<td></td>
<td>32</td>
<td>8.70.01.228</td>
<td></td>
<td>32</td>
<td>1.50.05.457</td>
<td></td>
<td>32</td>
<td>33.50.04.916</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>Performer</td>
<td>256</td>
<td>1.10.000.32</td>
<td>128</td>
<td>2.10.006.60</td>
<td></td>
<td>64</td>
<td>4.20.01.116</td>
<td></td>
<td>32</td>
<td>8.50.01.229</td>
<td></td>
<td>32</td>
<td>16.50.02.453</td>
<td></td>
<td>32</td>
<td>34.80.19.510</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>YOSO-C-32</td>
<td>512</td>
<td>1.00.000.15</td>
<td>256</td>
<td>2.00.01.31</td>
<td></td>
<td>256</td>
<td>4.00.01.63</td>
<td></td>
<td>64</td>
<td>8.50.01.127</td>
<td></td>
<td>64</td>
<td>16.30.01.255</td>
<td></td>
<td>64</td>
<td>34.80.19.510</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>YOSO-C-32</td>
<td>512</td>
<td>1.20.001.31</td>
<td>256</td>
<td>2.40.01.33</td>
<td></td>
<td>256</td>
<td>5.00.01.67</td>
<td></td>
<td>64</td>
<td>10.00.06.134</td>
<td></td>
<td>32</td>
<td>20.10.03.269</td>
<td></td>
<td>32</td>
<td>15.90.02.250</td>
<td></td>
<td>32</td>
</tr>
<tr>
<td>YOSO-C-16</td>
<td>512</td>
<td>1.00.000.15</td>
<td>256</td>
<td>1.90.000.31</td>
<td></td>
<td>256</td>
<td>3.90.02.62</td>
<td></td>
<td>64</td>
<td>7.90.01.125</td>
<td></td>
<td>32</td>
<td>15.90.02.250</td>
<td></td>
<td>32</td>
<td>10.10.04.501</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>*YOSO-C-32</td>
<td>512</td>
<td>1.00.000.15</td>
<td>256</td>
<td>2.00.01.31</td>
<td></td>
<td>256</td>
<td>4.50.01.63</td>
<td></td>
<td>64</td>
<td>9.60.02.127</td>
<td></td>
<td>32</td>
<td>18.70.05.254</td>
<td></td>
<td>32</td>
<td>41.30.14.509</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>*YOSO-C-16</td>
<td>512</td>
<td>0.80.000.14</td>
<td>256</td>
<td>1.60.000.29</td>
<td></td>
<td>256</td>
<td>3.40.01.59</td>
<td></td>
<td>64</td>
<td>7.50.11.119</td>
<td></td>
<td>32</td>
<td>15.40.08.238</td>
<td></td>
<td>32</td>
<td>30.40.05.476</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>*YOSO-C-16</td>
<td>512</td>
<td>1.20.000.16</td>
<td>256</td>
<td>2.50.000.33</td>
<td></td>
<td>256</td>
<td>5.00.00.67</td>
<td></td>
<td>64</td>
<td>11.50.04.134</td>
<td></td>
<td>32</td>
<td>23.10.09.268</td>
<td></td>
<td>32</td>
<td>48.30.10.537</td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

*Table 1. Runtime and memory efficiency for each method when the largest batch size is used for each method on each sequence length. BS: batch size, TIME/STD: the mean and standard deviation of runtime in ms, MEM: memory in MB*

### 2.2. BERT-Small for Baseline Comparisons

In this section, we provide a table of experimental results of BERT-small pretraining and GLUE downstream finetuning results, see Table 3. YOSO-C is comparable to softmax self-attention and Nyströmformer in MLM and SOP tasks while it outperforms Reformer and Performer. We noticed that Reformer performs very well on GLUE tasks despite a modest performance on MLM and SOP. Via further investigation, we found that 99% of instances in QNLI, QQP, and MNLI have sequence lengths less than 112. Reformer sorts and chunks the input sequence and then calculates attention within chunked sequences and consecutive chunks. The default size of each chunk is 64, so the equivalent attention window is 128 if the number of hashes is set to 1 and is at least 128 if the number of hashes is larger than 1. Therefore, Reformer can capture the full attention across all tokens in GLUE tasks. This full attention might be the reason for the better performance on
Table 2. Runtime and memory efficiency for each method when the same batch size is used for each method on each sequence length. BS: batch size, TIME/STD: the mean and standard deviation of runtime in ms, MEM: memory in MB.

GLUE of Nyströmformer. Further, this short sequence length might also explain the reason for the performance on GLUE of Nyströmformer: when most tokens in the input sequence are padding tokens, the outputted landmarks of the segmented mean computation are zero vectors.

<table>
<thead>
<tr>
<th>METHOD</th>
<th>MLM</th>
<th>SOP</th>
<th>QNLI</th>
<th>QQP</th>
<th>MNLI-M/MM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOFTMAX</td>
<td>7.05</td>
<td>91.3</td>
<td>87.7</td>
<td>86.0</td>
<td>79.1/79.5</td>
</tr>
<tr>
<td>NYSTRÖMFORMER</td>
<td>7.60</td>
<td>90.2</td>
<td>84.3</td>
<td>84.0</td>
<td>75.5/76.0</td>
</tr>
<tr>
<td>LINFORMER</td>
<td>8.21</td>
<td>90.9</td>
<td>85.8</td>
<td>85.0</td>
<td>77.4/77.5</td>
</tr>
<tr>
<td>REFORMER</td>
<td>8.57</td>
<td>89.0</td>
<td>86.9</td>
<td>86.2</td>
<td>78.2/78.6</td>
</tr>
<tr>
<td>PERFORMER</td>
<td>11.59</td>
<td>88.9</td>
<td>82.8</td>
<td>83.7</td>
<td>73.8/74.6</td>
</tr>
<tr>
<td>YOSO-32</td>
<td>8.55</td>
<td>89.5</td>
<td>84.7</td>
<td>83.1</td>
<td>75.0/75.3</td>
</tr>
<tr>
<td>YOSO-C-32</td>
<td>7.72</td>
<td>89.7</td>
<td>85.1</td>
<td>83.9</td>
<td>75.8/75.8</td>
</tr>
<tr>
<td>YOSO-32</td>
<td>8.43</td>
<td>89.1</td>
<td>84.9</td>
<td>84.4</td>
<td>76.7/77.0</td>
</tr>
<tr>
<td>YOSO-C-32</td>
<td>7.34</td>
<td>89.6</td>
<td>84.9</td>
<td>84.7</td>
<td>77.2/77.2</td>
</tr>
</tbody>
</table>

Table 3. Dev set results on MLM and SOP pretraining and GLUE tasks for baseline comparisons.

2.3. Experiment Summary

We provide a summary of all experiment details in Table 4. Note that since the Transformer model implementation in (Tay et al., 2021) for LRA tasks is slightly different than the Transformer model used in BERT (Devlin et al., 2019), to follow the exact setting, we have two variations of Transformers: one for LRA and one for BERT. For compute resources, since BERT-base pretraining requires a significant amount of computation, we used AWS’s p3dn.24xlarge for experiments related to BERT (MLM & SOP pretraining and GLUE tasks). Then, we run the rest of our experiments on a 4x2080TI server.

Table 4. Hyperparameters for all experiments. Pooling represents how token outputs from Transformers are processed to represent the input sequence for sequence classification. CLS means we append one fixed token in input and then use the model output of this token as a representation of the sequence. MEAN means the output vectors are averaged as a representation of the sequence. For MRPC and SST-2 in GLUE, a hyperparameter search is performed to select the best batch size and learning rate from the candidate set.
Appendix

References


