You Only Sample (Almost) Once: Appendix

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 τ 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 ℓ_2 norms of Q_i, K_j are bounded by some $\sqrt{\tau}$ for all $i, j = 1, \cdots, n$. Then we can construct new queries and keys as follows:

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

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

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 τ hyperplane hashes to boost the decay of collision probability as similarity decreases.

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

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 τ-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

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

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 $j = 1, ..., n$ 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 $\nabla_{Q_i} L$:

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

Then we use

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

and plug it into (1),

$$\frac{\partial L}{\partial Q_{i,q}}=\sum_{l=1}^{d}\frac{\partial L}{\partial Y_{i,l}}\sum_{j=1}^{n}\frac{\tau\left(1-\frac{\arccos(Q_i^TK_j)}{\pi}\right)^{\tau-1}}{\pi\sqrt{1-(Q_i^TK_j)^2}}K_{j,q}V_{j,l}$$

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

$$\frac{\partial L}{\partial Q_{i,q}}=\sum_{j=1}^{n}(\nabla_{Y_i}L)^TV_j\frac{\tau\left(1-\frac{\arccos(Q_i^TK_j)}{\pi}\right)^{\tau-1}}{\pi\sqrt{1-(Q_i^TV_j)^2}}K_{j,q}$$

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

$$\nabla_{Q_i}L=\sum_{j=1}^{n}(\nabla_{Y_i}L)^TV_j\frac{\tau\left(1-\frac{\arccos(Q_i^TK_j)}{\pi}\right)^{\tau-1}}{\pi\sqrt{1-(Q_i^TV_j)^2}}K_j$$

The term $\nabla_Q L$ in the main text is the matrix form of above expression for $i=1,\cdots n$

$$\nabla_Q L=[\left((\nabla_{\text{YOSO}}L)V^T\right)\odot\left(\tau(1-\frac{\arccos(QK^T)}{\pi})^{\tau-1}\right)\oslash\left(\pi\sqrt{1-(QK^T)^2}\right)]K \tag{2}$$

where $\odot,\oslash$ are element-wise multiplication and division respectively. Note that $\pi\sqrt{1-(QK^T)^2}$ 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

$$\hat{\nabla}_Q L=\left(\left((\nabla_{\text{YOSO}}L)V^T\right)\odot\frac{\tau}{2}(1-\frac{\arccos(QK^T)}{\pi})^{\tau}\right)K \tag{3}$$

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

$$\hat{\nabla}_{Q_i}L=\sum_{j=1}^{n}(\nabla_{Y_i}L)^TV_j\mathcal{B}(Q,K)_{i,j}\frac{\tau}{2}K_j$$

Note that if $\mathcal{B}(Q,K)_{i,j}$ 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 $\mathcal{B}(Q,K)_{i,j}$ and only computes the summation term when $\mathcal{B}(Q,K)_{i,j}=1$.

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 $O(\text{nnz}(\mathcal{B}(Q,K))(m+d))$ (counting number of success + computing nonzero terms). In the worst case, $\text{nnz}(\mathcal{B}(Q,K))=n^2$, 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, $\mathcal{B}(Q,K)$ is generally sparse if τ 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 $O(\min(nmd^2, \text{nnz}(\mathcal{B}(Q,K))(m+d)))$.

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 $\mathcal{B}(Q,K)$, so the quantity computed is as follows,

$$\nabla_Q L = [((\nabla_{\text{YOSO}} L)V^T) \odot \left(\tau(1 - \frac{\arccos(QK^T)}{\pi})^{\tau-1}\right) \oslash \left(\pi\sqrt{1-(QK^T)^2}\right) \odot \mathbb{1}_{\mathcal{B}(Q,K)}]K$$

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

$$w_{ij} = ((\nabla_{\text{YOSO}} L)_i^T V_j) \frac{(\tau(1 - \frac{\arccos(Q_i^T K_j)}{\pi})^{\tau-1}}{\pi\sqrt{1-(Q_i^T K_j)^2 + \epsilon}}$$

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

SEQ LENGTH	128			256			512			1024			2048			4096		
	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM
SOFTMAX	512	0.5/0.00	17	128	1.1/0.00	43	64	3.0/0.01	126	16	8.9/0.05	423	4	29.2/0.20	1529	1	100.8/0.72	5794
NYSTRÖMFORMER	256	1.1/0.01	32	128	1.8/0.01	49	64	3.1/0.03	83	64	5.6/0.01	149	32	11.0/0.07	283	16	21.9/0.14	551
LONGFORMER	-	-	-	-	-	-	32	11.3/0.03	169	16	24.4/0.04	348	8	50.4/0.10	699	4	102.1/0.14	1404
LINFORMER	256	0.7/0.00	27	128	1.3/0.00	51	64	2.5/0.00	99	32	5.2/0.01	197	16	10.1/0.03	391	8	20.9/0.08	782
REFORMER	256	1.1/0.00	28	128	2.2/0.00	57	64	4.3/0.01	114	32	8.7/0.01	228	16	17.5/0.05	457	8	35.3/0.04	916
PERFORMER	256	1.1/0.00	32	128	2.1/0.00	60	64	4.2/0.01	116	32	8.5/0.01	229	16	16.9/0.02	453	8	34.3/0.04	903
YOSO-32	512	1.0/0.00	15	256	2.0/0.01	31	128	4.0/0.01	63	64	8.5/0.01	127	32	16.5/0.01	255	16	34.8/0.19	510
YOSO-C-32	512	1.2/0.00	16	256	2.4/0.01	33	128	5.0/0.01	67	64	10.0/0.06	134	32	20.1/0.03	269	16	42.9/0.04	538
YOSO-C-16	512	1.0/0.00	15	256	1.9/0.00	31	128	3.9/0.02	62	64	7.9/0.01	125	32	15.9/0.02	250	16	32.1/0.04	501
*YOSO-32	512	1.0/0.00	15	256	2.0/0.01	31	128	4.5/0.01	63	64	9.6/0.02	127	32	18.7/0.05	254	16	41.3/0.14	509
*YOSO-16	512	0.8/0.00	14	256	1.6/0.00	29	128	3.4/0.01	59	64	7.5/0.11	119	32	15.4/0.08	238	16	30.4/0.05	476
*YOSO-C-32	512	1.2/0.00	16	256	2.5/0.00	33	128	5.0/0.00	67	64	11.5/0.04	134	32	23.1/0.09	268	16	48.3/0.10	537
*YOSO-C-16	512	1.0/0.00	15	256	2.0/0.02	31	128	4.1/0.01	62	64	8.9/0.01	125	32	18.1/0.12	250	16	36.6/0.05	500

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

Seq Length	128			256			512			1024			2048			4096		
	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM	BS	TIME/STD	MEM
Softmax	256	0.5/0.00	17	128	1.2/0.00	43	32	3.1/0.01	127	16	8.9/0.03	423	4	29.1/0.11	1529	1	101.6/1.96	5794
Nystromformer	256	1.1/0.00	32	128	1.8/0.01	49	32	3.3/0.05	83	16	6.2/0.09	151	4	17.6/0.24	292	1	70.9/1.15	592
Longformer	-	-	-	-	-	-	32	11.2/0.04	169	16	24.3/0.05	348	4	55.6/0.25	706	1	134.9/1.25	1335
Linformer	256	0.7/0.00	27	128	1.3/0.00	51	32	2.7/0.01	100	16	5.3/0.01	197	4	12.0/0.30	398	1	36.9/0.84	642
Reformer	256	1.1/0.00	28	128	2.1/0.01	57	32	4.5/0.02	114	16	9.0/0.03	229	4	19.3/0.15	464	1	53.2/1.07	935
Performer	256	1.1/0.00	32	128	2.1/0.00	60	32	4.4/0.02	117	16	8.7/0.03	230	4	18.5/0.11	461	1	48.1/0.66	942
YOSO-32	256	1.0/0.00	16	128	2.0/0.00	32	32	4.3/0.01	64	16	8.8/0.01	129	4	18.0/0.07	263	1	46.6/1.50	549
YOSO-16	256	0.8/0.00	14	128	1.6/0.00	29	32	3.3/0.01	60	16	6.8/0.02	121	4	14.6/0.15	247	1	41.5/0.89	518
YOSO-C-32	256	1.2/0.00	16	128	2.5/0.00	33	32	5.0/0.01	68	16	10.2/0.01	136	4	22.6/0.06	277	1	63.7/0.33	577
*YOSO-32	256	1.0/0.00	16	128	2.1/0.00	32	32	4.3/0.02	64	16	10.0/0.03	129	4	21.6/0.17	263	1	52.9/3.70	549
*YOSO-16	256	0.8/0.00	14	128	1.6/0.00	29	32	3.5/0.01	60	16	7.6/0.02	121	4	16.5/0.21	247	1	43.5/0.55	518
*YOSO-C-32	256	1.3/0.00	16	128	2.5/0.00	33	32	5.1/0.01	68	16	11.7/0.02	136	4	25.6/0.13	277	1	69.7/1.11	577
*YOSO-C-16	256	1.0/0.00	15	128	1.9/0.00	31	32	4.1/0.01	63	16	9.1/0.02	127	4	20.1/0.10	259	1	62.9/1.21	542

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 Reformer. 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.

Method	MLM	SOP	QNLI	QQP	MNLI-m/mm
Softmax	7.05	91.3	87.7	86.0	79.1/79.5
nyströmformer	7.60	90.2	84.3	84.0	75.5/76.0
linformer	8.21	90.9	85.8	85.0	77.4/77.5
reformer	8.57	89.0	86.9	86.2	78.2/78.6
performer	11.59	88.9	82.8	83.7	73.8/74.6
YOSO-32	8.55	89.5	84.7	83.1	75.0/75.3
YOSO-C-32	7.72	89.7	85.1	83.9	75.8/75.8
*YOSO-32	8.43	89.1	84.9	84.4	76.7/77.0
*YOSO-C-32	7.34	89.6	84.9	84.7	77.2/77.2

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.

Experiment	BERT-base			BERT-small		LRA				
	MLM/SOP	MRPC/SST-2	QNLI/QQP/MNLI	MLM/SOP	QNLI/QQP/MNLI	ListOps	Text	Retrieval	Image	Pathfinder
Input Length	512	512	512	512	512	2048	4096	4096	1024	1024
Num of Layers	12	12	12	4	4	2	2	2	2	2
Embedding Dim	768	768	768	512	512	64	64	64	64	64
Hidden Dim	3072	3072	3072	2048	2048	128	128	128	128	128
Num of Heads	12	12	12	8	8	2	2	2	2	2
Head Dim	64	64	64	64	64	32	32	32	32	32
Dropout					0.1					
Pooling	CLS	CLS	CLS	CLS	CLS	MEAN	MEAN	MEAN	MEAN	MEAN
Optimizer					Adam					
Batch Size	256	{8, 16, 32}	32	256	32	32	32	32	256	256
Learning Rate	1e-4	{2e-5, 3e-5, 4e-5, 5e-5}	3e-5	1e-4	3e-5	1-e4	1-e4	1-e4	1-e4	1-e4
Num of Steps	500K	4 epochs		500K	4 epochs	5K	20K	30K	35K	62.4K
Warmup Steps	10K	-	-	10K	-	1K	8K	800	175	312
Weight Decay	0.01	0.01	0.01	0.01	0.01	0	0	0	0	0

Table 4. Hyperparameters for all experiments. Pooling represents how token outputs from Transformers are processed to represent the entire 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.

References

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. Practical and optimal lsh for angular distance. In Cortes, C., Lawrence, N., Lee, D., Sugiyama, M., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf.

Charikar, M. S. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pp. 380–388, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581134959. doi: 10.1145/509907.509965. URL https://doi.org/10.1145/509907.509965.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pp. 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1423. URL https://www.aclweb.org/anthology/N19-1423.

Neyshabur, B. and Srebro, N. On symmetric and asymmetric lshs for inner product search. In Bach, F. and Blei, D. (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1926–1934, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/neyshabur15.html.

Tay, Y., Dehghani, M., Abnar, S., Shen, Y., Bahri, D., Pham, P., Rao, J., Yang, L., Ruder, S., and Metzler, D. Long range arena : A benchmark for efficient transformers. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=qVyeW-grC2k.