Block-Recurrent Transformers

DeLesley Hutchins[*1], **Imanol Schlag**[*3†], **Yuhuai Wu**[1], **Ethan Dyer**[2], **Behnam Neyshabur**[2]

[1] Google Research [2] Google Research, Blueshift Team
[3] The Swiss AI Lab IDSIA, SUPSI & USI
{delesley, yuhuai, edyer, neyshabur}@google.com imanol@idsia.ch

Abstract

We introduce the Block-Recurrent Transformer, which applies a transformer layer in a recurrent fashion along a sequence, and has linear complexity with respect to sequence length. Our recurrent cell operates on blocks of tokens rather than single tokens during training, and leverages parallel computation within a block in order to make efficient use of accelerator hardware. The cell itself is strikingly simple. It is merely a transformer layer: it uses self-attention and cross-attention to efficiently compute a recurrent function over a large set of state vectors and tokens. Our design was inspired in part by LSTM cells, and it uses LSTM-style gates, but it scales the typical LSTM cell up by several orders of magnitude. Our implementation of recurrence has the same cost in both computation time and parameter count as a conventional transformer layer, but offers dramatically improved perplexity in language modeling tasks over very long sequences. Our model out-performs a long-range Transformer XL baseline by a wide margin, while running twice as fast. We demonstrate its effectiveness on PG19 (books), arXiv papers, and GitHub source code. Our code has been released as open source [1].

1 Introduction

Transformers have mostly replaced recurrent neural networks (RNNs), such as LSTMs [2], on tasks that involve sequential data, especially natural language. There are several reasons for their success. First, transformers process all elements of the sequence in parallel, and are thus faster to train on modern accelerator hardware. In contrast, an RNN must process tokens sequentially, which leads to slow step times during training, and large batch sizes in order to fully saturate GPUs or TPUs.

Second, an RNN must summarize and compress the entire previous sequence into a single state vector which is passed from one token to the next. The size of the state vector limits the amount of information that the RNN can encode about the previous tokens in the sequence. In contrast, a transformer can attend directly to past tokens, and does not suffer from this limitation.

Third, attention operates effectively over longer distances. The forget gate in an LSTM discards information moving forward, and causes vanishing gradients during backpropagation. In practice, this means that LSTMs struggle to send a clear signal over more than a few hundred tokens, far less than the typical size of the attention window in a transformer [3].

Despite these advantages, transformers also have a disadvantage. The computational complexity of self-attention is quadratic with respect to the sequence length, which is a limiting factor when attempting to process long documents, such as books, technical articles, or source code repositories. Moreover, a transformer has no memory of past context; any tokens that it cannot attend to are "invisible" to the model.

[*] Equal Contribution † Work done while interning at Google Research, Blueshift Team

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

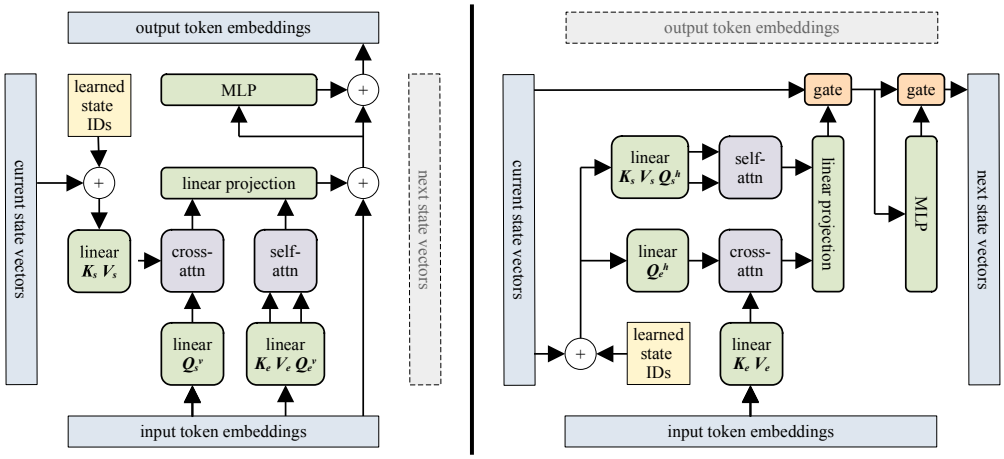

Figure 1: Illustration of our recurrent cell. The left side depicts the vertical direction (layers stacked in the usual way) and the right side depicts the horizontal direction (recurrence). Notice that the horizontal direction merely rotates a conventional transformer layer by 90°, and replaces the residual connections with gates.

In this work, we describe an architecture which combines the benefits of attention and recurrence. Like previous implementations of recurrence, our architecture constructs and maintains a fixed-size state, which summarizes the sequence that the model has seen thus far. However, our implementation of recurrence differs from previous work in several important aspects which together address the three limitations mentioned above.

Instead of processing the sequence one token at a time, **our recurrent cell operates on *blocks* of tokens**; see Figure 1. Within a block, all tokens are processed in parallel, at least during training. The recurrent cell likewise **operates on a *block* of state vectors rather than a single vector.** This means that the size of the recurrent state is orders of magnitude larger than in an LSTM, which dramatically improves the model's capacity to capture the past. Processing the sequence in blocks also helps propagate information and gradients over longer distances, because the number of recurrent steps (and thus the number of times that the forget gate is applied) is orders of magnitude smaller. We show that the Block-Recurrent Transformer can remember information over distances of 60k tokens or more.

The recurrent cell itself is strikingly simple. For the most part, it consists of an ordinary transformer layer applied in a recurrent fashion along the sequence length. There are **a few tricks that are necessary to stabilize training**; see Sections 3.2 and 3.4 for details. The cost of recurrence, in terms of both computation time and parameter count, is essentially the same as simply adding one more layer to our transformer baseline. We demonstrate empirically that adding a single recurrent layer results in a much larger improvement in perplexity on multiple datasets than adding a conventional transformer layer, while training time and memory use are equivalent. Moreover, our recurrent cell is very easy to implement because it largely makes use of existing transformer code. Thus, our technique is a cheap and cheerful way to improve language modeling perplexity on long sequences.

2 Related Work

The quadratic cost of attention is well known in the literature, and a great deal of work has been done on efficient long-range attention mechanisms; see [4, 5] for recent surveys. Sparse strategies such as Big Bird [6], Routing Transformers [7], and Reformer [8] select only a subset of tokens to attend to. Hierarchical mechanisms [9] combine multiple tokens into phrases or sentences to reduce sequence length. Expire-span [10] learns to prune far-away tokens that the model has labelled as "unimportant". Memorizing transformers [11] replace dense attention with k-nearest-neighbor lookup.

Yet another approach is to reduce the sequence length by pooling, averaging, or compressing it in some way. Hierarchical 1D attention [12], and Combiner [13] apply pooling or averaging over tokens at longer distances. Linformer [14] applies a linear transformation to the key and value matrices to reduce the sequence length. Compressive transformers [15] and funnel transformers [16] apply additional learned compression layers to compress the sequence.

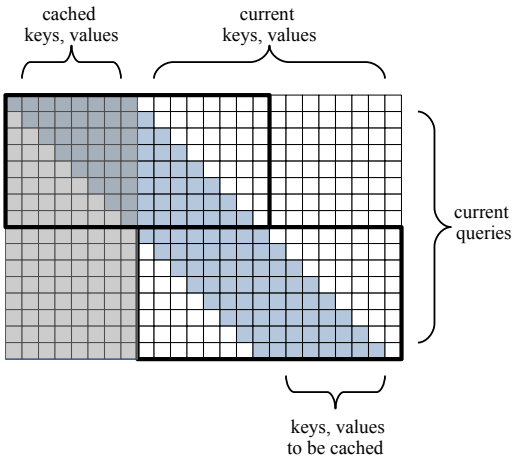

Figure 2: Sliding window, where segment length $N = 16$, window/block size $W = 8$. Keys and values for the first W shaded tokens were computed and cached on the previous training step; the remaining N unshaded tokens are the segment for the current training step. Instead of a single $N \times (W + N)$ attention matrix, attention is done in two tiles of size $W \times 2W$.

The equation for attention is (roughly) $\mathsf{softmax}(\boldsymbol{Q}\boldsymbol{K}^T)\boldsymbol{V}$ where $\boldsymbol{Q}$, $\boldsymbol{K}$, and $\boldsymbol{V}$ are the query, key, and value matrices of the attention layer. If the softmax operation is removed from this equation or somehow "linearized", the equation can be rearranged as $\boldsymbol{Q}(\boldsymbol{K}^T\boldsymbol{V})$, where $(\boldsymbol{K}^T\boldsymbol{V})$ can be computed incrementally (i.e., in a recurrent fashion) as a cumulative sum over the sequence [17]. Linearized attention thus has linear rather than quadratic complexity with respect to sequence length. Following this line of reasoning, there have been several proposals that approximate the softmax [18, 19] or replace it [20, 21]. Linear transformers are related to earlier work on fast weight programmers [20] [22], and can be extended with other forms of recurrence [23].

Our work differs from all of the above mechanisms, because we rely only on standard dense attention with softmax.

A few other lines of research have combined the transformer architecture with recurrence in some way. The feedback transformer [24] allows lower layers to attend to the output of the topmost layer. Feedback has minimal cost at inference time, but it is unfortunately very slow to train because tokens must be processed sequentially. Simple Recurrent Units [25, 26] use a recurrence function that does not involve matrix multiplication, and is consequently much faster. RNMT+ combines RNNs and transformers in an encoder/decoder architecture to improve on translation tasks [27]. "Sandwich models" alternate between transformer and RNN layers and out-perform both transformers and RNNs on tasks involving source code [28]. The R-Transformer introduces an additional local RNN which can be computed in parallel in order to better model sequential structure [29]. The Perceiver architecture [30] is somewhat similar to ours; it also applies a transformer layer in an iterative fashion.

To the best of our knowledge, the idea of performing recurrence on blocks of tokens is underexplored. In the context of translation, [31] operates on sentences rather than tokens. Staircase Attention [32] also operates on blocks of tokens; each layer takes, as input, the outputs of the same layer from the previous block.

3 Method

The Block-Recurrent Transformer is based on sliding-window attention [33], which is an extension of ideas from Transformer-XL [34].

A long document, such as a book, consists of a *sequence* of tokens. Due to memory limitations, it is usually not possible to fit the entire sequence into device memory. Thus, the sequence is divided into *segments* of length N ($N = 4096$ in our experiments), which are processed sequentially over a number of training steps. Each training step processes one segment.

The sliding window attention pattern is illustrated in Figure 2. Given a segment of N tokens, the sliding window applies a causal mask in which each token can only attend to the W previous tokens,

where W is the *window size* ($W = 512$ in our experiments). Because of the causal mask, most entries of the $N \times N$ attention matrix are masked out (assuming that $W << N$). Thus, the attention computation can be optimized by breaking it into smaller tiles along the diagonal. The segment of N tokens is subdivided into *blocks* of size W, and each block attends locally to itself and to the previous block, so the size of each local attention matrix is $W \times 2W$. Using this mechanism, attention is quadratic with respect to the window size W, but linear with respect to the segment length N.

Borrowing an idea from Transformer-XL, the keys and values from the last block in each segment are stored in a non-differentiable *cache* for use on the next training step. By using the cache, the first block in the next segment can attend to the last block in the previous segment, which extends the sliding window to cover the entire (book-length) sequence. The cache implements a form of truncated backpropagation through time [35] over long documents.

Note that if $N = W$, then sliding window attention will behave exactly like Transformer-XL; it will process and cache one segment (i.e. one block) per training step. Setting $N >> W$ does not change the context length of attention, but it allows gradients to backpropagate across multiple blocks during training; we show that the improved differentiability provides a modest benefit to perplexity over Transformer-XL. See Appendix A for more details.

3.1 Recurrent Cell

A Block-Recurrent Transformer layer extends the sliding-window attention mechanism by adding a set of recurrent states, which are updated at the end of each block of W tokens. Our design for the recurrent cell is illustrated in Figure 1, which depicts the operations done within a single block of the input sequence.

The recurrent cell receives two tensors as inputs: a set of W token embeddings, where W is the block/window size, and a set of S "current state" vectors. The cell produces two tensors as outputs: a set of W output embeddings, as well as a set of S "next state" vectors. We denote the function going from input token embeddings to output token embeddings as the *vertical* direction, and the function going from the current state vectors to the next state vectors as the *horizontal* direction. The number of state vectors S and the window size W are independent hyperparameters, but we set $S = W = 512$ in our experiments to simplify comparisons against baselines.

The **vertical direction** of the cell is an ordinary transformer layer with an additional cross-attention operation, much like a decoder layer in a standard encoder-decoder architecture [36]. It does self-attention over the input tokens, and cross-attends to the recurrent states. Unlike a typical decoder layer, we do self-attention and cross-attention in parallel. The results of both forms of attention are concatenated together and fed into a linear projection.

The **horizontal direction** of the cell mirrors the forward direction, except that it performs self-attention over the current state vectors, and cross-attends to the input tokens. The recurrent direction also replaces the residual connections with gates, which allows the model to "forget", an ability that is important for algorithmic tasks [37], or when processing long documents, where it has been central to the success of LSTMs [38].

Note that the presence of gates is the reason why self-attention and cross-attention are done in parallel. Doing them sequentially, as is standard practice, would introduce a third gate in the horizontal direction, which led to worse perplexity in our experiments.

Recurrence is integrated with the sliding window attention mechanism. Although not shown in Figure 1, each cell also receives keys and values from the previous block as input, these are concatenated with ($\boldsymbol{K}_e$, $\boldsymbol{V}_e$) from the current block in order to implement sliding-window attention.

A Block-Recurrent Transformer *layer* processes the blocks within a segment sequentially by stacking recurrent cells horizontally, with the "next states" output of the previous cell feeding into the "current states" input of the next cell. In code, this is implemented as a simple for-loop over blocks. Multiple layers can also be stacked vertically in the usual fashion. Our experiments use a single recurrent layer, sandwiched between a number of non-recurrent layers that use sliding-window attention.

The final set of state vectors from the last block in the segment are cached, along with the keys and values, and used as the initial state for the first block on the next training step. Every layer in the stack (both recurrent and non-recurrent) has its own cache.

Sharing of keys and values. Keys and values are shared between the vertical and horizontal directions. One set of keys and values ($\boldsymbol{K}_e, \boldsymbol{V}_e$) are computed from the input token embeddings, and another set of keys and values ($\boldsymbol{K}_s, \boldsymbol{V}_s$) are computed from the recurrent state vectors. Queries are not shared, so there are four separate sets of queries: $\boldsymbol{Q}_e^v$ and $\boldsymbol{Q}_s^v$ in the vertical direction, and $\boldsymbol{Q}_s^h$ and $\boldsymbol{Q}_e^h$ in the horizontal direction.

3.2 State IDs and Position Bias

With a large number of state vectors, the total size of the recurrent state is far larger than that of an LSTM. However, the same weights (projection matrices and MLP) are applied to each state vector. Without some way to differentiate the states, the model will compute the same result for each state vector, thus negating any advantage from having multiple states. To prevent this failure mode, we add a set of learned "state IDs" to the state vectors before computing the keys, values, and queries. These "state IDs" allow each state vector to consistently issue different queries against the input sequence, and against other states. State IDs are identical to learned position embeddings; we use a different name because there's no notion of "position" between states.

We do not add global position embeddings to the tokens, because global position embeddings don't work well for long sequences [34]. Instead, we add a T5-style relative position bias [39] to the self-attention matrix in the vertical direction. (Although similar, T5 relative positions differ slightly from the relative positions used in the Transformer-XL paper [34].) When the recurrent states cross-attend to input tokens, there is no position bias, because the relative distance between "state" and "token" is undefined.

We also normalize queries and keys as described in [40]; we found that normalization improved the stability of Transformer-XL when used with a relative position bias.

3.3 Gate Type

We experimented with two different gating mechanisms for the recurrent cell. Each state vector has its own gate, but all state vectors are updated in parallel, using the equations below.

Fixed gate. The fixed gate uses a learned convex combination, similar to highway networks [41].

$$\boldsymbol{z}_t = \boldsymbol{W}_z \boldsymbol{h}_t + \boldsymbol{b}_z \tag{1}$$

$$\boldsymbol{g} = \sigma(\boldsymbol{b}_g) \tag{2}$$

$$\boldsymbol{c}_{t+1} = \boldsymbol{c}_t \odot \boldsymbol{g} + \boldsymbol{z}_t \odot (1 - \boldsymbol{g}) \tag{3}$$

where $\boldsymbol{W}_z$ is a trainable weight matrix, $\boldsymbol{b}_z$ and $\boldsymbol{b}_g$ are trainable bias vectors, σ is the sigmoid function, $\boldsymbol{c}_t$ is the cell state for the current block (i.e., the state for the block at index t in the sequence of blocks), $\odot$ is the element-wise multiplication, and $\boldsymbol{h}_t$ is the current input to the gate. In our model, $\boldsymbol{h}_t$ is either the output of attention, in which case $\boldsymbol{W}_z$ is the linear projection that feeds into the gate, or $\boldsymbol{h}_t$ is the output of the hidden layer of the MLP, in which case $\boldsymbol{W}_z$ is the final layer of the MLP.

Unlike highway networks, the bias $\boldsymbol{b}_g$ is a simple learned vector of shape $\mathbb{R}^d$, which is broadcast over all state vectors, where d is the state embedding dimension. The value of $\boldsymbol{g}$ does ***not*** depend on either the current value of the state vector $\boldsymbol{c}_t$, or on the current input $\boldsymbol{h}_t$, and thus remains constant (i.e., fixed) after training. The fixed gate essentially implements an exponential moving average over previous blocks.

LSTM gate. The LSTM gate uses the standard combination of input and forget gates:

$$\boldsymbol{z}_t = \tanh(\boldsymbol{W}_z \boldsymbol{h}_t + \boldsymbol{b}_z) \tag{4}$$

$$\boldsymbol{i}_t = \sigma(\boldsymbol{W}_i \boldsymbol{h}_t + \boldsymbol{b}_i - 1) \tag{5}$$

$$\boldsymbol{f}_t = \sigma(\boldsymbol{W}_f \boldsymbol{h}_t + \boldsymbol{b}_f + 1) \tag{6}$$

$$\boldsymbol{c}_{t+1} = \boldsymbol{c}_t \odot \boldsymbol{f}_t + \boldsymbol{z}_t \odot \boldsymbol{i}_t \tag{7}$$

where $\boldsymbol{W}_z, \boldsymbol{W}_i, \boldsymbol{W}_f$ are trainable weight matrices, and $\boldsymbol{b}_z, \boldsymbol{b}_i, \boldsymbol{b}_f$ are trainable bias vectors. The LSTM gate is strictly more expressive, because the values of $\boldsymbol{f}_t$ and $\boldsymbol{i}_t$ depend on the current input $\boldsymbol{h}_t$. In our model, $\boldsymbol{h}_t$ depends on $\boldsymbol{c}_t$, so the LSTM gate also depends indirectly on $\boldsymbol{c}_t$. LSTM gate values are thus different for each state vector, and for each block index t.

3.4 Gate Initialization and Training Stability

We observed that training stability is quite sensitive to how the gates are initialized. Recurrence has a failure mode where the model learns to completely ignore the recurrent state, in which case its performance reverts to that of the non-recurrent transformer. Moreover, this situation appears to be a local optimum; once the model has reached this point, it does not recover. We stabilize training by initializing the weights and bias to small but non-zero values, and adding a constant -1 and +1 to the input and forget gates to bias them to "remember". See Appendix B for details.

3.5 Gate Configuration

We experimented with three different gate configurations.

Dual. The dual gate configuration is the one shown in Figure 1, in which both of the residual connections in the cell are replaced with gates. The disadvantage of this configuration is that there are two gates, both of which can forget.

Single. The single gate configuration removes the linear projection and the gate that is attached to it. Instead, the concatenation of self-attention and cross-attention is fed directly into the MLP.

Skip. The skip configuration removes the MLP and the gate that is attached to it. This configuration is similar to the single-gate version, except that it is strictly weaker. Instead of a two layer MLP with a very large hidden layer, it uses a linear projection with no nonlinearity.

3.6 Placement of Recurrence and Computation Cost

Single recurrent layer. The basic version of the Block-Recurrent Transformer uses a single recurrent layer sandwiched between a number of non-recurrent transformer layers with sliding attention. We use a 12-layer model with recurrence on layer 10. All layers have a Transformer-XL-style cache.

Cost of recurrence. During training, the 12-layer Block-Recurrent Transformer has almost exactly the same computation cost, in both parameters and FLOPS, as a 13-layer Transformer-XL model without recurrence. The two are equivalent because the recurrent cell does almost the same operations as a conventional transformer layer, merely in the horizontal instead of the vertical direction.

The inference cost for autoregressive decoding is also nearly identical, for the same reason. Recurrence adds an additional attention operation per token, the cost of which is the same as self-attention in a 13th layer.

4 Results

We tested the Block-Recurrent Transformer on three different data sets of long documents: PG19, arXiv, and GitHub. The PG19 dataset [42] contains full-length books written prior to 1919 from project Gutenberg. The arXiv dataset [11] is a corpus of technical papers downloaded via the arXiv Bulk Data Access[1], and filtered to include only articles labeled as "Mathematics" and whose LaTeX source is available. The GitHub dataset [11] is a corpus of source code from different GitHub repositories with open-source licenses. All of the files in each GitHub repository are concatenated together to make one long document.

The task is auto-regressive language modeling, where the goal is to predict the next token in the sequence. We report bits-per-token numbers (i.e. $\log_2$ perplexity; lower is better) for all models. Further training details for each dataset can be found in Appendix C.

4.1 Baselines

We compare the Block-Recurrent Transformer to five different baselines. The first baseline, `XL:512`, establishes a reference point against which various other improvements can be compared. It's a

[1] `https://arxiv.com/help/bulk_data`

Table 1: Average bits-per-token ($\log_2$ perplexity) of each model. The recurrent models (named `Rec:gate:config`) have the same computational cost as the `Slide:13L` baseline, but much better perplexity. They even outperform the `XL:2048` baseline, **while running more than twice as fast.** Measured error bars on PG19 are low, between 0.002 and 0.007, but are rounded up to 0.01 to match the precision of results in the table. Step time is for a single training step (lower is better). For PG19, we train both character-level (bytes) and token-level models.

Model	segment length	window length	step time (relative)	PG19 bytes	PG19 tokens	arXiv tokens	GitHub tokens
`XL:512`	512	512	0.88	1.01	3.62 ± 0.01	1.45	1.21
`XL:1024`	1024	1024	1.20	0.997	3.59 ± 0.01	1.37	1.08
`XL:2048`	2048	2048	2.11	0.990	3.58 ± 0.01	1.31	1.01
`Slide:12L`	4096	512	0.93	0.989	3.60	1.43	1.19
`Slide:13L`			1.00	0.989	3.58 ± 0.01	1.42	1.17
`Rec:lstm:dual`	4096	512	1.06	0.985	3.54 ± 0.01	1.26	1.01
`Rec:lstm:single`			1.05	0.962	3.54 ± 0.01	1.29	1.03
`Rec:lstm:skip`			1.00	0.969	3.56 ± 0.01	1.31	1.10
`Rec:fixed:dual`			1.01	0.957	$\mathbf{3.52 \pm 0.01}$	1.27	0.991
`Rec:fixed:single`			1.02	0.966	3.58 ± 0.01	1.25	1.00
`Rec:fixed:skip`			**0.99**	**0.952**	3.53 ± 0.01	**1.24**	**0.976**
`Feedback:lstm:single`	4096	512	1.40	0.977	3.50	**1.22**	-
`Feedback:fixed:skip`			1.35	**0.935**	**3.49**	1.24	-
Memorizing Trans. 64k	512	512	1.94	0.950	3.53	**1.22**	-

12-layer Transformer-XL model with a window size of 512, and 150 million parameters. It has 8 heads of size 128, embedding vectors of size 1024, an MLP with a hidden layer of size 4096, and the relu nonlinearity. It uses a Transformer-XL style cache, but no sliding window, so the segment length is the same as the window size, i.e., it is trained on segments of 512 tokens.

`XL:1024` and `XL:2048` are similar, but have window sizes of 1024 and 2048, respectively. As expected, increasing the window size improves perplexity, especially on the arXiv data set. However, these two models still have worse perplexity than the recurrent model, as well as being much slower.

`Slide:12L` is a 12-layer transformer with a window size of 512, but uses a sliding window over a segment of 4096 tokens. This model is almost identical to `XL:512`; the only difference is that the sliding window is differentiable over multiple blocks, while the Transformer-XL cache is not.

`Slide:13L` adds a 13th layer, and is directly comparable to the recurrent models in terms of both computation cost (FLOPS or step-time), number of parameters, and segment length. Notice that adding another layer with more parameters yields a much smaller improvement than adding recurrence.

Relative cost. All five baselines, and all 6 recurrent models, have roughly the same number of parameters: between 151 million (12 layer) and 164 million (13 layer or recurrent). The training speed (i.e. step time) of each model is shown in Table 1 (lower is better). Because the raw step time depends on hardware and compiler, we report numbers relative to the `Slide:13L` baseline.

Batch Size. We adjust the batch size so that each model processes the same number of tokens (and thus the same amount of training data) per training step. Thus, `XL:512` (segment length 512) runs at a batch size of 256 (8 per replica), while `Slide:12L` (segment length 4096) runs at a batch size of 32 (1 per replica) on PG19.

4.2 Benefit of Recurrence

We compare the 5 baselines to all six gate configurations for the Block-Recurrent Transformer. The recurrent model reliably outperforms all five baselines. The best overall configuration is `Rec:fixed:skip`, which outperforms the others in 3 out of 4 cases, and comes within the margin of error in the remaining case. This is especially notable because it is also the fastest configuration, having a slightly *lower* step time and fewer parameters than `Slide:13L`, because it does not have

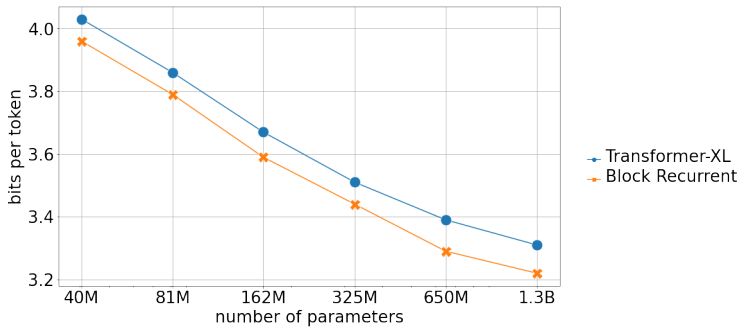

Figure 3: Scaling of the 12-layer Block-Recurrent Transformer vs 13-layer Transformer-XL on PG19. FLOPs are the same between the two models at a given parameter count. **At larger sizes, adding recurrence is equivalent to doubling the number of parameters.** Details in Appendix F.

the MLP. It is better than the 13-layer baseline by a wide margin, and it is even better than the Transformer-XL model with a window size of 2048, which runs over 2 times slower.

The other gate configurations also outperform the 13-layer baseline, but their relative ranking varies according to the dataset. Despite being theoretically more powerful, the `LSTM` gate tends to lag behind the `fixed` gate in all of our experiments.

Scaling up. Figure 3 shows the effect of adding recurrence as the transformer model is scaled up and down in size. We trained six different models on PG19, ranging in size from 40M parameters to 1.3B parameters. For the four smaller models, we compare a 12-layer Block-Recurrent Transformer against a 13-layer Transformer-XL baseline, while for the two larger models, we compare a 24-layer Block-Recurrent Transformer, with recurrence at layers 10 and 20, against a 26-layer Transformer-XL baseline. This experiment used a cosine-decay learning rate as described in [43], and a custom 32k SentencePiece vocabulary [44]. More details are in Appendix F.

Our experiments show that recurrence provides a consistent benefit across all scales. The relative improvement actually seems to increase with the number of parameters; at larger sizes **recurrence provides a benefit which is greater than doubling the number of parameters**.

4.3 Ablations

Multiple recurrent layers. Adding two recurrent layers right next to each other in the stack (layers 9 and 10) did not improve model perplexity. Adding two layers widely separated in the stack (layers 4 and 10) did provide an improvement, but the improvement was no better than simply adding another non-recurrent layer to the stack. Previous work on Memorizing Transformers [11] showed a similar effect. In our qualitative study, we saw that the model seems to use recurrence primary for long-range name lookups, much like memory. We conclude that one layer of recurrence is sufficient for the model to extract most of the benefits, although we did use two layers for our largest models.

Number of recurrent state vectors. We trained the model with differing numbers of state vectors, from 128 to 2048. Increasing the number of states makes a small but measurable improvement up to 1024, but the model does worse with 2048 (see Appendix D). We hypothesize that the model has trouble learning to use the recurrent state effectively if the state space grows too large.

Reducing window size. Reducing the size of the sliding window dramatically reduces perplexity for Transformer-XL, because it reduces the amount of context that the transformer is able to attend to. Reducing the size of the window in a recurrent transformer has a smaller effect, because the model can use recurrence to compensate (see Appendix D).

4.4 Block feedback

Inspired by the feedback transformer [24], which allows all layers to attend to the topmost layer, we implemented a variation in which every layer of the transformer (not just the recurrent one) can cross-attend to the state vectors in the recurrent layer. This variation further improves perplexity, but

Table 2: Comparison with other published work on PG19. Fields marked - are unknown.

Model	Layers	perplexity word-level	parameters	vocabulary size
Compressive Transformer [15]	36	33.6	-	32k
Routing Transformer [7]	22	33.2	490M[1]	98k
Perceiver AR [45]	60	28.9	974.6M[1]	32k
Block-Recurrent Transformer	24	28.46	650M	32k
Block-Recurrent Transformer	24	**26.50**	1.3B	32k

at a cost; step time increased by approximately 35-40%, and the additional queries also increase the number of parameters. Results are shown in Table 1, and further described in Appendix E.

4.5 Comparisons against prior published work

The PG19 test set contains 6,966,499 words [15], which are broken into 10,229,476 tokens using a SentencePiece vocabulary, trained on PG19. Our 24-layer 1.3B parameter model achieves 3.22 bits per token, and thus **achieves a new state of the art word-level perplexity of 26.50** (Table 2). However, we note that raw perplexity numbers are not necessarily a meaningful way to compare architectures, because they depend on numerous other factors, such as the number of parameters, vocabulary, learning rate schedule, batch size, etc.; a more detailed discussion is in Appendix C.3.

We were able to run a fair comparison (identical vocabulary, configuration, and hyperparameters) of the Block-Recurrent Transformer against the Memorizing Transformer [11], with a memory of size 64k (Table 1). The memorizing transformer is constructed similarly to our model; it has one layer which has been augmented with a mechanism that gives it the ability to attend over much longer distances. We find that Block-Recurrence does almost as well as the Memorizing Transformer on arXiv, and does just as well on PG19, but trains almost twice as fast. However, there are many ways of implementing approximate k-nearest-neighbor lookup, so relative speed will be highly implementation-dependent; our implementation runs on TPU, and does not use custom CUDA kernels.

4.6 Qualitative analysis

Prior work on long-context transformers [42, 11] has found that attention at long ranges is typically used to look up proper names, such as characters or places. We performed a qualitative analysis in an attempt to determine whether our model is using recurrence in the same way. We selected 5 books at random from the PG19 test set, ran both the Block-Recurrent Transformer and the 13-layer Transformer-XL on each book, and then compared the cross-entropy loss for all tokens. We sorted the results, and examined the top 4 tokens from each book with the greatest difference: the tokens for which the predictions of the recurrent model have the largest improvement over the baseline.

In 17/20 cases, the recurrent model predicted a proper name, usually with relatively high probability, that Transformer-XL was unable to predict. In 2 cases it predicted a chapter title (having previously seen the table of contents), and in the last case, it predicted a foreign-language word that was unique to that book. In 19/20 cases, the predicted word was nowhere within the attention window, so it must have been stored within the recurrent state (details in the appendix, Section G).

In a second study, we compared the recurrent model, running normally, against a variation in which the recurrent state is cleared at the end of each 4096-token segment, instead of being cached. Clearing the state degrades the model's ability to predict dependencies at a longer range than the segment length; typical mispredictions once again included proper names and chapter titles. Interestingly, this study also showed that the recurrent model is able to remember the title and author of a book (which is part of the Gutenberg boilerplate at the beginning and end of each book) **across the entire length of the book – more than 60,000 tokens.** See Appendix G.1.

A further quantitative comparison of the per-token cross-entropy between Transformer-XL and the Block-Recurrent Transformer is given in Appendix H.

[1]Personal communication.

5 Discussion

Our implementation of recurrence was inspired by the way that humans seem to process long sequences. When a human reads a novel, they do not attempt to remember every single word in the book. Instead, a human reader will construct a mental model, or knowledge graph, which summarizes the story thus far, i.e., the names of the main characters, the relationships between them, and any major plot points. When a human reads a paragraph of text, they will parse the information in the paragraph, process and interpret the information using background knowledge from their mental model, and finally update their mental model with new information. Our recurrent architecture loosely mimics this process. It takes a block of text, and parses it by running it through a conventional transformer stack. Tokens in the text attend to the recurrent states (i.e. the mental model), and the states, in turn, are updated by attending to the text.

Based on our qualitative analysis, it seems that the model is, in fact, using the recurrent state to summarize some of the information about frequently occurring characters and places. However, it does not seem to be doing much complex reasoning, as evidenced by the fact that our best performing model is the `fixed:skip` configuration. This configuration does not use a complex LSTM-style gate, which chooses to remember or forget based on its current state and inputs; instead, it simply computes an exponential moving average, not unlike some other forms of long-range approximate attention.

Moreover, the `skip` configuration cuts out the large MLP from the recurrent transformer layer. In a vanilla transformer, removing the MLP from all layers would severely degrade the model [46]; those large MLPs are computing something important. In a recurrent layer, removing the MLP makes little difference; it does not seem to be computing anything useful. We conclude that training the recurrent layer to make full use of its capabilities for knowledge extraction and summarization will require further advances.

5.1 Ethics

The potential negative social impacts from this work are similar to any other advance in language modelling. Large language models could potentially be used to create disinformation and fake news, power malicious chatbots, or generate spam. The Block-Recurrent Transformer can potentially create longer documents than was previously feasible, thus expanding the range of applications in which these negative impacts could occur. The best way to mitigate these risks is to train models that can reason about text, and flag misinformation or malicious content.

6 Conclusion

We have shown that when training language models on long documents, the Block-Recurrent Transformer provides a greater benefit at lower cost than scaling up the transformer model in other ways. Adding recurrence to a single layer has roughly the same cost as adding an additional non-recurrent layer, but results in a much larger improvement to perplexity. We have also shown that recurrence provides a larger benefit than simply increasing the window size of attention, or increasing the number of parameters. **Our medium-sized model has lower perplexity than a Transformer-XL model with 4 times the window size, but runs twice as fast, and our larger model outperforms a Transformer-XL model with twice the number of parameters**.

Furthermore, in contrast to some other recently proposed transformer variants, the Recurrent Transformer is very easy to implement, since it consists mostly of ordinary transformer components and RNN gates. No custom CUDA kernels are required. Our code has been released as open source [1].

Evaluating block-recurrent transformers on downstream tasks is an important direction for future work. We believe that the Block-Recurrent Transformer will be most useful in situations that require long-range context; examples of potential applications include writing book reports, summarizing long news articles, code completion, or question/answering over book-length works. There are are a number of new and emerging benchmarks that test long-range performance [47, 48, 4]. Previous studies have found a strong correlation between language modeling and diverse downstream tasks [49, 50].

Despite our initial successes, we also believe that the recurrent architecture that we present here has not yet achieved its full potential, and there are opportunities for future research and further improvements in this area.

References

[1] D. Hutchins, M. Rabe, Y. Wu, I. Schlag, and C. Staats, "Meliad." Github source code repository. `https://github.com/google-research/meliad`, 2022.

[2] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, 1997.

[3] U. Khandelwal, H. He, P. Qi, and D. Jurafsky, "Sharp nearby, fuzzy far away: How neural language models use context," in *Association for Computational Linguistics*, 2018.

[4] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, "Long range arena : A benchmark for efficient transformers," in *International Conference on Learning Representations*, 2021.

[5] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, "Efficient transformers: A survey," *arXiv preprint arXiv:2009.06732*, 2020.

[6] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontañón, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," in *NeurIPS*, 2020.

[7] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, "Efficient content-based sparse attention with routing transformers," *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 53–68, 2021.

[8] N. Kitaev, L. Kaiser, and A. Levskaya, "Reformer: The efficient transformer," in *International Conference on Learning Representations*, 2020.

[9] J. Ainslie, S. Ontañón, C. Alberti, V. Cvicek, Z. Fisher, P. Pham, A. Ravula, S. Sanghai, Q. Wang, and L. Yang, "ETC: encoding long and structured inputs in transformers," in *EMNLP*, 2020.

[10] S. Sukhbaatar, D. Ju, S. Poff, S. Roller, A. Szlam, J. Weston, and A. Fan, "Not all memories are created equal: Learning to forget by expiring," in *ICML*, 2021.

[11] Y. Wu, M. Rabe, D. Hutchins, and C. Szegedy, "Memorizing transformers," in *ICLR*, 2022.

[12] Z. Zhu and R. Soricut, "H-transformer-1d: Fast one-dimensional hierarchical attention for sequences," in *ACL* (C. Zong, F. Xia, W. Li, and R. Navigli, eds.), 2021.

[13] H. Ren, H. Dai, Z. Dai, M. Yang, J. Leskovec, D. Schuurmans, and B. Dai, "Combiner: Full attention transformer with sparse computation cost," in *Advances in Neural Information Processing Systems* (A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), 2021.

[14] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *CoRR*, vol. abs/2006.04768, 2020.

[15] J. W. Rae, A. Potapenko, S. M. Jayakumar, C. Hillier, and T. P. Lillicrap, "Compressive transformers for long-range sequence modelling," in *ICLR*, 2020.

[16] Z. Dai, G. Lai, Y. Yang, and Q. Le, "Funnel-transformer: Filtering out sequential redundancy for efficient language processing," in *NeurIPS*, 2020.

[17] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are RNNs: Fast autoregressive transformers with linear attention," in *ICML*, 2020.

[18] K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlós, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, D. B. Belanger, L. J. Colwell, and A. Weller, "Rethinking attention with performers," in *ICLR*, 2021.

[19] H. Peng, N. Pappas, D. Yogatama, R. Schwartz, N. A. Smith, and L. Kong, "Random feature attention," in *ICLR*, 2021.

[20] I. Schlag, K. Irie, and J. Schmidhuber, "Linear Transformers are secretly fast weight programmers," in *ICML*, 2021.

[21] W. Hua, Z. Dai, H. Liu, and Q. V. Le, "Transformer quality in linear time," *arXiv preprint arXiv:2202.10447*, 2022.

[22] J. Schmidhuber, "Reducing the ratio between learning complexity and number of time varying variables in fully recurrent nets," in *International Conference on Artificial Neural Networks (ICANN)*, 1993.

[23] K. Irie, I. Schlag, R. Csordás, and J. Schmidhuber, "Going beyond linear transformers with recurrent fast weight programmers," in *confNEU*, 2021.

[24] A. Fan, T. Lavril, E. Grave, A. Joulin, and S. Sukhbaatar, "Addressing some limitations of transformers with feedback memory," *arXiv preprint arXiv:2002.09402*, 2020.

[25] T. Lei, Y. Zhang, S. I. Wang, H. Dai, and Y. Artzi, "Simple recurrent units for highly parallelizable recurrence," in *EMNLP*, 2018.

[26] T. Lei, "When attention meets fast recurrence: Training language models with reduced compute," in *EMNLP*, 2021.

[27] M. X. Chen, O. Firat, A. Bapna, M. Johnson, W. Macherey, G. F. Foster, L. Jones, M. Schuster, N. Shazeer, N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, Z. Chen, Y. Wu, and M. Hughes, "The best of both worlds: Combining recent advances in neural machine translation," in *ACL*, 2018.

[28] V. J. Hellendoorn, P. Maniatis, R. Singh, C. Sutton, and D. Bieber, "Global relational models of source code," in *ICLR*, 2020.

[29] Z. Wang, Y. Ma, Z. Liu, and J. Tang, "R-transformer: Recurrent neural network enhanced transformer," *arXiv preprint arXiv:1907.05572*, 2019.

[30] A. Jaegle, F. Gimeno, A. Brock, O. Vinyals, A. Zisserman, and J. Carreira, "Perceiver: General perception with iterative attention," in *icml*, 2021.

[31] A. Al Adel and M. S. Burtsev, "Memory transformer with hierarchical attention for long document processing," in *2021 International Conference Engineering and Telecommunication (En T)*, 2021.

[32] D. Ju, S. Roller, S. Sukhbaatar, and J. Weston, "Staircase attention for recurrent processing of sequences," *arXiv preprint arXiv:2106.04279*, 2021.

[33] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.

[34] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive language models beyond a fixed-length context," in *ACL*, 2019.

[35] R. J. Williams and J. Peng, "An efficient gradient-based algorithm for on-line training of recurrent network trajectories," *Neural Computation*, 1990.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017.

[37] R. Csordás, K. Irie, and J. Schmidhuber, "The neural data router: Adaptive control flow in transformers improves systematic generalization," in *International Conference on Learning Representations*, 2022.

[38] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A search space odyssey," *IEEE transactions on neural networks and learning systems*, vol. 28, no. 10, 2016.

[39] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *Journal of Machine Learning Research*, 2020.

[40] A. Henry, P. R. Dachapally, S. S. Pawar, and Y. Chen, "Query-key normalization for transformers," in *EMNLP*, 2020.

[41] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Training very deep networks," in *NIPS* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), 2015.

[42] S. Sun, K. Krishna, A. Mattarella-Micke, and M. Iyyer, "Do long-range language models actually use long-range context?," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 807–822, Association for Computational Linguistics, Nov. 2021.

[43] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, *et al.*, "Training compute-optimal large language models," *arXiv preprint arXiv:2203.15556*, 2022.

[44] T. Kudo and J. Richardson, "Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing," in *EMNLP*, 2018.

[45] C. Hawthorne, A. Jaegle, C. Cangea, S. Borgeaud, C. Nash, M. Malinowski, S. Dieleman, O. Vinyals, M. Botvinick, I. Simon, *et al.*, "General-purpose, long-context autoregressive modeling with perceiver ar," *arXiv preprint arXiv:2202.07765*, 2022.

[46] Y. Dong, J. Cordonnier, and A. Loukas, "Attention is not all you need: pure attention loses rank doubly exponentially with depth," in *ICML* (M. Meila and T. Zhang, eds.), 2021.

[47] U. Shaham, E. Segal, M. Ivgi, A. Efrat, O. Yoran, A. Haviv, A. Gupta, W. Xiong, M. Geva, J. Berant, and O. Levy, "Scrolls: Standardized comparison over long language sequences," 2022.

[48] A. Wang, R. Y. Pang, A. Chen, J. Phang, and S. R. Bowman, "Squality: Building a long-document summarization dataset the hard way," *arXiv preprint arXiv:2205.11465*, 2022.

[49] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *NeurIPS*, 2020.

[50] A. Srivastava, A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch, A. R. Brown, A. Santoro, A. Gupta, A. Garriga-Alonso, *et al.*, "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models," *arXiv preprint arXiv:2206.04615*, 2022.

[51] N. Shazeer and M. Stern, "Adafactor: Adaptive learning rates with sublinear memory cost," in *ICML*, 2018.

[52] R. Thoppilan, D. De Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, *et al.*, "Lamda: Language models for dialog applications," *arXiv preprint arXiv:2201.08239*, 2022.

[53] O. Press, N. Smith, and M. Lewis, "Train short, test long: Attention with linear biases enables input length extrapolation," in *International Conference on Learning Representations*, 2022.

[54] D. Adiwardana, M.-T. Luong, D. R. So, J. Hall, N. Fiedel, R. Thoppilan, Z. Yang, A. Kulshreshtha, G. Nemade, Y. Lu, *et al.*, "Towards a human-like open-domain chatbot," *arXiv preprint arXiv:2001.09977*, 2020.

7 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]

(c) Did you discuss any potential negative societal impacts of your work?
[Yes] See Section 5.1.

(d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. 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] We plan to open-source the code itself, although we have not yet done so. We have given a detailed description of architecture, hyper-parameters, and training methodology. Our main results are for PG19, which is a publicly available dataset.

(b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix C.

(c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)?
[Yes] Given the large number of experiments, we did not have the computational resources to run every experiment multiple times, but we do run experiments over multiple datasets. For the main headline numbers on PG19-tokens, we ran the primary experiments three times each with different initial seeds and with dataset shuffling. Error bars are given in Table 1. Note that actual measured error bars are somewhat lower than reported in Table 1; we round results to the nearest 0.01, and thus round the error *up* to match the precision of the reported results. See Appendix C for details.

(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 Appendix C.

3. 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]

(b) Did you mention the license of the assets? [Yes] See Appendix C.1.

(c) Did you include any new assets either in the supplemental material or as a URL? [N/A]

(d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [Yes] See Appendix C.1.

(e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] See Appendix C.1.

Appendices

Appendix A Further Analysis.

The *context length* of an autoregressive language model refers to the number of previous tokens that the model can make use of when predicting the next token. A vanilla transformer operating on segments of length N has a context length of 0 for the first token, and $N - 1$ for the last token, and thus has an average context length of $N/2$. The prediction quality for a vanilla transformer is not uniform across the segment; it makes poor predictions for tokens near the beginning of the segment due to lack of context, and better predictions near the end.

With sliding window attention, each layer attends to the previous layer within a window of W tokens from the current position. The context length for a single layer is thus W, no matter where in the segment the token occurs. The simple case where $W = N$ corresponds to Transformer-XL. A Transformer-XL model achieves a large improvement in average perplexity over a vanilla model, simply because it can make much better predictions for the tokens at the beginning of each segment.

In a model with multiple layers, the *theoretical receptive field* (TRF) is defined as the maximum distance that information could potentially propagate through the model. This definition is similar to "context length", but the TRF is "theoretical", because the model may have a hard time actually learning to use that much context in practice. For example, the TRF of an LSTM is infinite, but in practice an LSTM has difficulty transmitting information over more than a few hundred tokens. The *effective context length* of an LSTM is much less than the TRF might suggest.

For a sliding-window model (or Transformer-XL), the TRF is $W \cdot L$, where L is the number of layers. However, the model can still only attend directly to the previous W tokens. Although the TRF is much higher, making use of the additional context requires multiple "hops" of attention, which is more difficult for the model to learn. This problem is especially acute in Transformer-XL ($N = W$), because the very first "hop" of attention will be to keys and values in the cache, which is not differentiable. Our sliding window model ($N >> W$) has an identical TRF to Transformer-XL, but it can differentiate across multiple blocks, which gives it a higher effective context length in practice, and thus better perplexity.

The TRF of the block-recurrent transformer is infinite. We also show that the effective context length also seems to be quite large in practice, since we observe cases in which the model is able to accurately predict information across distances of more than 60k tokens.

A.1 Computational Complexity

The computational complexity of attention in a recurrent layer is $\mathcal{O}((W^2 + S^2 + 2SW) \cdot N/W)$, where N is the segment length, W is the window length, and S is the number of states. N/W is the number of blocks, and each block does self attention W^2, state self-attention S^2, and attention between tokens/states and states/tokens $2SW$.

The complexity of sliding-window attention is $\mathcal{O}(W^2 \cdot N/W) = \mathcal{O}(W \cdot N)$.

A.2 Comparison between context, recurrence, and memory

The arXiv data set contains latex code, with lots of complicated syntax that can benefit from long-range attention (e.g. theorems, jargon, citations). As a result, adding memory, recurrence, or just increasing the window size of the Transformer-XL baseline yields a larger benefit on arXiv than it does for PG19.

There also seems to be a qualitative difference in how the models are using attention on these datasets. We hypothesize that dealing with complicated syntax requires direct (single-"hop") attention, which is why the `XL:2048` model does well on arXiv. In contrast, natural language novels don't have complicated syntax, but may benefit from a better understanding of subtle relationships between words (e.g. which character the word "her" refers to). These subtle interactions require multi-layer, multi-"hop" attention, which is easier to train in the more differentiable `Slide:13L` model. On PG19, `Slide:13L` actually does better than `XL:2048`, despite having a much shorter window size, while on arXiv, the situation is reversed.

There may be a similar relationship between recurrence and kNN memory. The Memorizing Transformer does direct, single-"hop" attention, using kNN lookup. Recurrence, in contrast, summarizes and compresses text into a set of recurrent states.

Both the Memorizing Transformer and the Block-Recurrent Transformer achieve very similar perplexity, and based on qualitative studies, they seem to use the additional memory or states primarily for long-range name lookups. However, recurrence may be better at capturing more subtle long-range information, like writing style, while memory is better at precision lookups of facts, like citations. The fact that the Memorizing Transformer does better than the Block-Recurrent transformer on arXiv, but not PG19, would seem to support this hypothesis, but further experiments on downstream tasks are necessary to confirm this hypothesis.

A.3 Comparison against Longformer

The idea of sliding window attention was popularized by the Longformer [33], but the full Longformer is a much more complicated model than the `Slide` model that we present here.

The full Longformer uses several different attention patterns, and sliding-window attention is only one of them. Longformer also uses dilated attention and sparse global attention, both of which are implemented with custom CUDA kernels. Moreover, LongFormer uses different window sizes in each layer, and it uses a multi-phase training regimen of pre-training and fine-tuning, following a curriculum that gradually increases window size and sequence length. We do none of these things.

Appendix B Gate Initialization and Training Stability

We observed that training stability is quite sensitive to how the gates are initialized. Recurrence has a failure mode where the model learns to completely ignore the recurrent state, in which case its performance reverts to that of the non-recurrent transformer. Moreover, this situation appears to be a local optimum; once the model has reached this point, it does not recover.

Our hypothesis is that learning the recurrent transition function is a much more difficult task than learning to attend directly to the input tokens. As a result, the vertical direction trains much faster than the horizontal direction, especially early in training. This may lead to a situation in which the recurrent states are much less informative than the input tokens, and the model learns to ignore them.

To avoid this failure mode, gate initialization requires special care. Moreover, proper gate initialization depends on the optimizer. We use the Adafactor optimizer [51], which normalizes gradients with respect to their variance, and then multiplies them by the native scale of the underlying parameter. Thus, if a bias term is initialized to 0, its native scale will be 0, the gradient updates will be very small, and the bias will tend to remain small over the course of training. If a bias term is initialized to 1 (which tells the forget gate to "remember", and is standard practice in LSTMs) then the initial updates will be large, and the model will learn to ignore the recurrent states before they have the chance to learn anything useful.

We compromise by initializing the bias terms of the gates to small but non-zero values, using a normal distribution with mean 0 and a standard deviation of 0.1. The weight matrices of the gates are also initialized to small values, using a truncated normal distribution with a standard deviation of $\sqrt{(\frac{0.1}{f_{\text{in}}})}$ where f_{in} is the dimension of $\boldsymbol{h}_t$.

We add a constant of -1 and +1 to the input and forget gates (see Eq. 4) to initially bias the gate to "remember" without affecting the size of the updates that Adafactor will apply. Using this initialization trick, the recurrent cell reliably learns to make use of the recurrent state.

Appendix C Training Details

For PG19, we do both token-level modeling, and character-level modeling. In our initial experiments, we use a pre-trained 32k sentencepiece vocabulary from T5 [39] for the token-level modeling. We use the Adafactor optimizer [51], a learning rate schedule with inverse square root decay, 1000 warmup steps, and a dropout rate of 0.05. The learning rate is 1.0; when combined with warmup and the decay schedule, this yields an applied learning rate of 0.03, decaying to 0.0014. This learning rate

and schedule were borrowed from other language models; we did not attempt to do a hyperparameter sweep to identify the optimum learning rate and schedule. We train for 500k steps with 32 replicas on Google V4 TPUs; training takes approximately 48 hours. Reported results are for the "test" split.

For the later scaling experiment on PG19, we switched the learning rate schedule to cosine decay, as recommended in [43], with a maximum rate of 0.01, and a minimum of 0.001. We did a brief experiment with a learning rate of 0.02 and 0.005, before settling on 0.01. The change in learning rate schedule resulted in a significant improvement. We also switched from the 32k T5 vocabulary to a 32k custom sentencepiece vocabulary trained on PG19. Our custom vocabulary has higher bits-per-token, but fewer tokens, and thus has slightly better word-level perplexity.

For arXiv, we use a pre-trained 32k vocabulary from LaMDA [52]. Due to the large number of mathematical symbols in LaTeX, many tokens are only one character, so the bits-per-token numbers are lower than for PG19. We dropped the learning rate to 0.5 after observing some instabilities when training on longer (4096) segment lengths. Reported results are for the "test" split.

The GitHub dataset is much larger than PG19, and has very high variance, due to the fact that it contains code written in many different programming languages and coding styles. Consequently, there was a lot of noise in the results, which made it difficult to accurately compare models. We reduced the noise by using a batch size that is 4x larger than for PG19. As with the PG19 scaling study, we use a cosine decay learning rate schedule, but the maximum LR is 0.005; this is half the LR used for PG19. Because of the increased batch size, these models ran for only 250k steps. The GitHub experiment uses a pre-trained 32k vocabulary from LaMDA. Reported results are for the "validation" split.

Due to the large number of experiments, we did not have the computational resources to run all experiments multiple times, and consequently do not provide error bars for all experiments. However, for the headline numbers on PG19-tokens, we ran each experiment 3 times, with both different random seeds and with dataset shuffling. Actual measured error bars on PG19 were very low, between 0.002 and 0.007. The numbers in Table 1 are rounded to the nearest 0.01, which means that the error bars must be rounded up to match the precision of the reported results. E.g. for `Rec:fixed:skip` on PG19-tokens, an average of 3 runs has a mean of 3.525 and a standard deviation of 0.0047; we round this *up* to 3.53 ± 0.01. Note that it is not possible to obtain truly accurate error bars from such a small number of runs; by rounding the error up, we provide a conservative estimate of the actual error.

C.1 Dataset licensing and other issues.

PG19 consists of works in the public domain, and consequently it is a public dataset that is freely available to other researchers. Due to the age of the texts, some of the books do contain potentially offensive material.

The arXiv dataset consists of documents for which the author has given express permission for their work to be distributed by `arxiv.org`. However, because the author still retains copyright, these articles cannot necessarily be redistributed in the form of a public dataset, nor will we publish a model that has been pre-trained on this data. We obtained access to this dataset via private channels.

The Github dataset consists of code with open-source licenses, which permit the code to be downloaded, compiled, or modified. Similar to arXiv, however, because the authors retain copyright, this code cannot necessarily be redistributed as a public dataset, nor will we publish a model that has been pre-trained on this data. We obtained access to this dataset via private channels.

C.2 Selection of data sets

Having a standardized data set is import for the purpose of comparing published results, and historically, papers on long-context language modeling have used enwik8 or wiki-103 as benchmarks [34][33]. However, these datasets are not particularly good benchmarks for our purposes.

The purpose of our experiments is to see whether block recurrence can transmit information over very long distances: we show retrieval over 60k+ tokens. We chose PG19 specifically because we believe it to be a good dataset for these sorts of experiments. It consists only of long, book-length works, it is much larger than enwik8 or wiki-103, it is publicly available, and has been cited in other published work. Arxiv and github are (sadly) not public, but they similarly have long documents in the 50k+ token range.

Enwik8 is not a corpus of long articles. In fact, it doesn't even split the text into separate articles at all; it's just a single text dump that concatenates a bunch of short unrelated articles together. Attempting to split it on article boundaries yields a data set in which the majority of "articles" are merely stubs, with HTML boilerplate and no actual text. Enwik8 is a fine benchmark for data compression, which was the purpose for which it was originally intended, but it is less than ideal for long-range language modeling.

Wiki-103 is significantly better, because it does break the text into articles, and it eliminates the boilerplate, but the average length is still only 3.6k tokens per article, which is less than the segment length used in our experiments, and much shorter than the 50k-100k tokens of PG19.

C.3 Comparisons with previously published results.

It is well-known that transformer perplexity scales with the number of parameters. However, the choice of vocabulary, learning rate schedule, batch size, number of training steps, and optimizer also make a large difference to the final headline perplexity numbers. The Chinchilla scaling study [43] demonstrates that a change to the learning rate schedule can have a large effect; we also observed improvements from a change to vocabulary as well. Not all vocabularies are created equal, even for vocabularies which have the same size, and are trained primarily on English-language text. Published perplexity numbers between different models cannot be meaningfully compared unless all other variables are strictly controlled.

Appendix D Window Size and Number of Recurrent States

Ablation results for the window size is is shown in Table 3 (a). Decreasing window size leads to worse perplexity in both the recurrent model and Transformer-XL, but the penalty is smaller for the recurrent model.

Ablation results for the number of recurrent states is shown in Table 3 (b). Increasing the number of recurrent states makes a small but measurable improvement up to 1024 states, but is worse at 2048 states. The window size was 512 for this experiment.

Table 3: Changing the window size (a) and number of recurrent states (b) on PG19.

Window size	`Rec:fixed:skip`	`Slide:12L`
128	3.58	3.69
256	3.54	3.64
512	3.53	3.60

Number of states	`Rec:fixed:skip`
128	3.54
256	3.535
512	3.53
1024	3.51
2048	3.55

Appendix E Block-Feedback

In the block-feedback variation of our model, the entire stack of 12 layers is applied to the first block of tokens. The recurrent state for that block is then extracted from the recurrent layer, and the state is broadcast to all other layers when processing the next block. Because the recurrent layer is placed high in the stack, this means that the lower layers of the transformer can cross-attend to a higher layer, which is computationally more powerful, much like the feedback transformer [24]. Results are shown in Table 1.

Recurrence with feedback is significantly more expensive than the non-feedback version, because all 12 layers now have a cross-attention module, instead of just the recurrent layer. In our experiments, feedback increased the step time by approximately 35-40%, and the additional queries also increase the number of parameters.

Adding feedback improves perplexity in most cases, but the improvement seems to depend on the data set. The effect of feedback also depends on the gate configuration. In particular, block feedback dramatically improves the performance of the LSTM gate. This could be because the recurrent states, and thus the gate, get a gradient from all all layers of the transformer, instead of just one.

Appendix F Scaling Plot Details

Table 4: Bits per token on PG19 at various model scales. The data in this table was used for the scaling plot in Figure 3.

	40.6M	81.2M	162M	325M	650M	1.3B
`Rec:fixed:skip`	3.96	3.79	3.59	3.44	3.29	3.22
`XL:512:13-layer`	4.03	3.86	3.67	3.51	3.39	3.31

Performance on large text datasets, such as PG-19, is highly correlated with the number of trainable parameters; larger models tend to perform better. However, training large models can be expensive, and not all researchers have access to the necessary amount of compute to beat our new state of the art, or even to reproduce our results.

Table 4 provides the numeric results from our scaling study, which covers scales from small 40M parameter models that can be easily trained on a single machine, to our largest 1.3B parameter model. Configuration files for all scales are provided in the open source release. Our scaling strategy is to increase the dimensions of the various parts of the transformer: embedding size, MLP hidden layer size, number of heads, head size, and number of layers. Each factor of 2 increase in the number of parameters scales either one or two of these dimensions.

Appendix G Qualitative Analysis Results

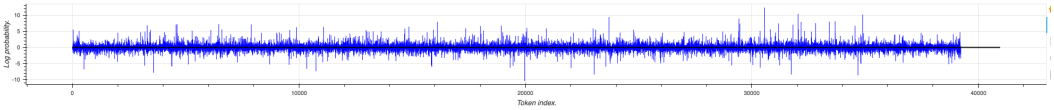

Figure 4: Difference in per-token cross-entropy loss.

The following are excerpts from our qualitative study. We selected five books at random from PG19 test set, and ran two different models on each book. The first model is a 13-layer Transformer-XL, and the second is the `Rec:fixed:skip` configuration of the Block-Recurrent Transformer. For each token, we compute the difference between the cross-entropy loss (i.e., the negative log likelihood (NLL)) output by both models, and and then sort the results.

Figure 4 shows an example of the per-token difference in NLL between the two models on the first book; the x-axis is the index of the token. On average, the recurrent model does slightly better than Transformer-XL, but it does not necessarily make a better prediction for any individual token.

The following excerpts show the top 4 tokens where the Block-Recurrent Transformer made a better prediction than Transformer-XL; these tokens correspond to spikes in Figure 4. We show the token number, the NLL returned by the recurrent model, the NLL returned by Transformer-XL, and an excerpt of text, with the token itself marked with `|token|`. Almost all of the top tokens are proper names of characters and places. In all cases except one, the mispredicted name does not appear within the attention window of the previous 512 tokens. These names are thus invisible to Transformer-XL, but visible to the recurrent model.

Note that these are not cherry picked examples; the five books are chosen at random. Moreover, the same pattern still holds if the search is expanded to the top 40 tokens for each book. In fact, even the names are often the same; Transformer-XL often seems to mispredict the same names over and over again; these are likely the names of main characters.

Sorting the other way, to show the top tokens where Transformer-XL does better than the recurrent model, does not show the same pattern. There are still plenty of proper names, but it is usually cases where both models fail to predict the name. Moreover, the names are mixed with more common words as well.

Memoirs of Marie Antoinette Queen Of France Vol. 7.

`(30555, 0.3696011, 12.688916)`

```
physician's house to make inquiries as to  the cause of so long an absence.
G|omin| and Larne had not yet ventured to  follow this advice, when next

(32043, 0.20177796, 10.513703)
with the autopsy arrived at the  outer gate of the Temple.  These were
Dum|ang|in, head physician of the  Hospice de l'Unite; Pelletan,

(34896, 1.3752508, 11.534926)
who had acted as courier to Louis  XVI. during the flight to Varennes,
and Tur|gi|, who had waited on the  Princesses in the Temple.  It was

(34896, 1.3752508, 11.534926)
.    In all the evidence there appeared but two serious facts, attested
by  Latour-|du|-Pin and Valaze, who deposed to them because they could
```

Notes: "Gomin" appears multiple times in the top 40 results.

An Unsentimental Journey through Cornwall

```
(983, 0.80441666, 11.626528)
OUGH CORNWALL          [Illustration: FALMOUTH, FROM FLUSHING.]
|DAY| THE FIRST      I believe in holidays. Not in a frantic rushing

(23606, 0.655162, 11.265186)
there was not the slightest use  in getting up, I turned round and took
another sleep.          |DAY| THE FIFTH      "Hope for the best, and be

(11297, 0.405902, 10.426135)
ball.    "Ma'am, if you go slow and steady, with me before and Cur|gen|ven
behind,  you'll _not_ fall."    Nor did I. I record it

(38021, 1.3457009, 11.167879)
our journey; going over the same ground which  we had traversed already,
and finding Praden|ack| Down as bleak and  beautiful as ever. Our first
```

Notes: The word "DAY" appears earlier in the table of contents, but not within the previous 512 tokens. "Curgenven" appears multiple times in the top 40.

The Good Hope **by Herman Heijermans Jr.**

```
(3975, 0.7425603, 15.8969145)
to us."    It matters nothing that this gospel of Life has often been
preached. He|ij|ermans has caught the spirit of it as well as the  letter.

(7385, 0.26241615, 15.000762)
must scratch the stones.    CLEM. Tomorrow afternoon, then.    COB. T|ja|!
I'll be here, then. Good day, Miss. [To Barend.] Good

(42638, 0.73628587, 15.407666)
hundred guilders. Bejour. [Rings off;  at the last words Kne|irt|je has
entered.]    KNEIRTJE. [Absently.] I----[

(25798, 0.12310324, 14.402218)
They exeunt, dragging Barend.]    KNEIR. Oh, oh----    TRU|US|. [With
anxious curiosity, at side door.] What was the matter,  Kneir?
```

Notes: The word "Tja" is not a proper name, but is a Dutch word that is likely unique to this particular book. It appears multiple times in the top 40 results. The name "Truus" appears multiple times in the top 40.

***Travels in Morocco Volume 2* by James Richardson**

Notes: The second example in this list is not a good one, because both models mispredict the name. We thus include a fifth example as well. The word "Toser" appears frequently in the top 40, and is also the only case where the proper name appears within the attention window of 512 tokens.

```
(61049, 2.0328524, 18.696453)
, some of them as black as [Redacted]s. Many people in  T|oser| have sore
eyes, and several with the loss of one eye, or nearly  so; opthal

(63453, 9.254061, 24.413633)
Tunis;" but the restrictive system established by the Turks  during late
years at Ghad|umes|, has greatly damaged the trade between the  Jereed and

(66149, 0.8768588, 14.105822)
enterprizing fellow,  worthy of imitation. He calculated the distance from
Ghabs to T|oser| at  200 miles. There are a number of towns in the

(64161, 1.0309317, 14.152167)
and ourselves went to Wedyen, a town and  date-wood about eight miles
from T|oser|, to the left. The date-grove is  extensive, and there are

(27282, 0.04982663, 12.551973)
, is a very ancient city, situate upon the right bank of the  river
Boura|gra|g, and near its mouth. This place was captured in 1263, by
```

India's Love Lyrics by Laurence Hope

```
(15694, 0.07477246, 11.22447)
rieving            A wasted chance.  Fate knows no tears.       Verses:
Fa|iz|Ulla       Just in the hush before dawn     A little wistful wind is

(3135, 0.0126608405, 9.759871)
afloat     In the little noontide of love's delights           Between
two Nights.            Val|go|vind's Boat Song       Waters glisten and
sunbeams quiver,

(23183, 0.1419289, 9.481476)
sleep, the touch of your lips on my mouth.            His Rubies: Told by
Val|go|vind       Along the hot and endless road,       Calm and erect,
with haggard eyes,'

(26886, 0.13428347, 9.46193)
off sheath,       And find there is no Rival like the Past.     Verse
by T|aj|Mahomed       When first I loved, I gave my very soul    Utterly
```

G.1 Clearing the recurrent state

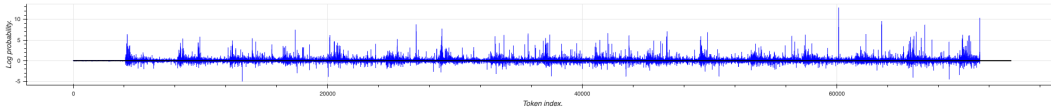

Figure 5: Difference in per-token cross-entropy loss with state clearing.

Our second qualitative study is structured similarly to the first, except that instead of comparing two different models, we compare two different runs of the same model: the `Rec:fixed:skip` configuration. The first run processes the book normally, while the second run clears the recurrent states at the beginning of each 4096-token segment. In the second run, the model can use recurrence within a segment to look beyond the local attention window of 512 tokens, but it cannot use recurrence to carry information from one segment to the next.

This experiment is somewhat cleaner than the first, because both runs are done with the same pre-trained model, which has the same parameters. Figure 5 shows the difference in per-token cross-entropy loss for the first book. There is no difference between the two runs for the first segment, and the biggest differences in subsequent segments are often clustered near the start of each new segment.

The overall pattern is very similar to the first qualitative experiment: most of the tokens involve proper names. We verified that in most cases, the mispredicted name not only does not occur within the 512-token attention window, but does not occur within the 4096-token segment. In addition to proper names, chapter titles and illustration captions occur frequently within the top 40 results; the recurrent model seems to be remembering these from a previous occurrence in the table of contents.

Perhaps most interestingly, in two of the books, one of the highest ranked mispredictions was the title and author of the book itself. The Gutenberg project inserts boilerplate at both the beginning and end of each book; the title and author are listed multiple times at the beginning, and once at the end. This experiment thus shows that the model is able to "remember" this information in the recurrent state, across a distance of 60,000 tokens or more.

Baby Mine, by Margaret Mayo

Note that the 2nd misprediction **is the title of the book itself**, at token number 71,244.

```
(60153, 0.00438364, 12.798895)
nerves, but you  needn't worry, I've got everything fixed.
Donneg|hey| sent a special  officer over with me. He's outside

(71244, 2.3727102, 12.660636)
sunlight and shadows of his  ample, well kept grounds.      End of the
Project Gutenberg EBook of| Baby| Mine, by Margaret Mayo    ***

(63536, 4.132567, 13.688847)
Alfred, with the air of a  connoisseur.    "She sure is," admitted
Don|neg|hey, more and more disgruntled as he felt  his reputation for

(26947, 3.7521973, 12.516711)
with a sigh of thanksgiving he hurried upstairs to his unanswered mail.
CHAPTER XIII    When Alfred| Hardy| found himself on the train bound for
```

The 'Patriotes' of '37 by Alfred D. Decelles

```
(38759, 0.85374373, 11.140007)
24, 125,  126.    Cote, Dr Cyri|le|, 89, 108, 118, 120;

(40181, 0.35475963, 10.291676)
17-26, 129-30.    Nelson, Dr Wolf|red|, a follower of Papineau, 37, 60

(28756, 0.0010796335, 9.256147)
on a well-reasoned plan of  action.  Most of the leaders--Wolf|red| Nelson,
Thomas Storrow Brown,  Robert Bouchette, and Amury Girod--were

(28772, 1.7782648, 10.611834)
leaders--Wolfred Nelson, Thomas Storrow Brown,  Robert Bouchette, and
Am|ury| Girod--were strangers to the men under  their command; and none of
```

Another Brownie Book, by Palmer Cox

In this case, **the top 4 results include the author of the book**. This book has a lot of illustrations which are listed in the table of contents, and make up many of the other results in the top 40.

```
(16442, 0.087840006, 8.706101)
```

```
[Illustration]    [Illustration]          [Illustration]    THE BROWNIES
AND THE TUG|BO|AT.      [Illustration]    While Brownies strayed along a

(24225, 0.04784866, 8.266858)
[Illustration]            End of the Project Gutenberg EBook of Another
Brownie Book, by Palmer| Cox|   ***

(24224, 5.651471, 11.840027)
Illustration]    [Illustration]            End of the Project Gutenberg
EBook of Another Brownie Book, by| Palmer| Cox    ***

(4460, 1.0682098, 6.6397715)
To secret haunts without delay.    [Illustration]          [Illustration]
THE BROWNIES AT| AR|CHERY.      [Illustration]    [Illustration]
```

The Life Of Thomas Paine Vol. II. **(of II) by Moncure Daniel**

By the end of the book, the recurrent model is quite sure that "Paine" is a main character, but not if its memory keeps getting erased.

```
(170457, 0.98941976, 12.209277)
they could. The scandal branched into  variants. Twenty-five years later
pious Grant| Thor|burn promulgated that  Paine had run off from Paris

(120592, 0.6974209, 10.396527)
my friends and accept the same to yourself."    As the Commissioners did
not leave when they expected,| Paine| added  several other letters to

(169152, 0.18432029, 9.516743)
whose composition the farrier no doubt supposed  he had paid the editor
with stories borrowed from "Old|ys|," or not  actionable. Cheetham

(139870, 0.53406477, 9.646917)
nor was any letter received from  him. This was probably the most important
allusion in a letter of| Paine|,  dated New York, March 1, 1804, to "C
```

Appendix H Token level cross-entropy

In addition to qualitative studies comparing a single document, we also compared the average bits-per-token over all documents in the PG19 test set. It has been observed that in vanilla transformer architectures, this token-wise cross-entropy often diverges at long segment lengths [53].

In Figure 6 we plot the cumulative cross-entropy, which is the average bits-per-token (i.e. $\log_2$ perplexity) averaged up to the given length, and we compare the Block Recurrent Transformer against the Transformer-XL baseline. Performance of the two architectures is comparable for the first few thousand tokens, but the recurrent architecture clearly outperforms Transformer-XL at longer document lengths.

Appendix I Vocabulary Ablation Experiments

The authors of the PG-19 dataset propose *word-level perplexity* (WLP) as a measure of model performance [15]. However, it is only possible to compute WLP if there is an accurate count of the number of words. A published word count is provided for the PG19 data set [15], but a word count is not available for the other data sets in this paper, such as arXiv and github. There is not even a clear definition of what constitutes a "word" in source code or LaTeX documents, which have a lot of symbols. Consequently, we do not provide WLP for these data sets.

WLP also glosses over some details of tokenization; in particular, the word count ignores whitespace. SentencePiece [44] is often advertised as being a subword tokenizer that preserves whitespace, but in

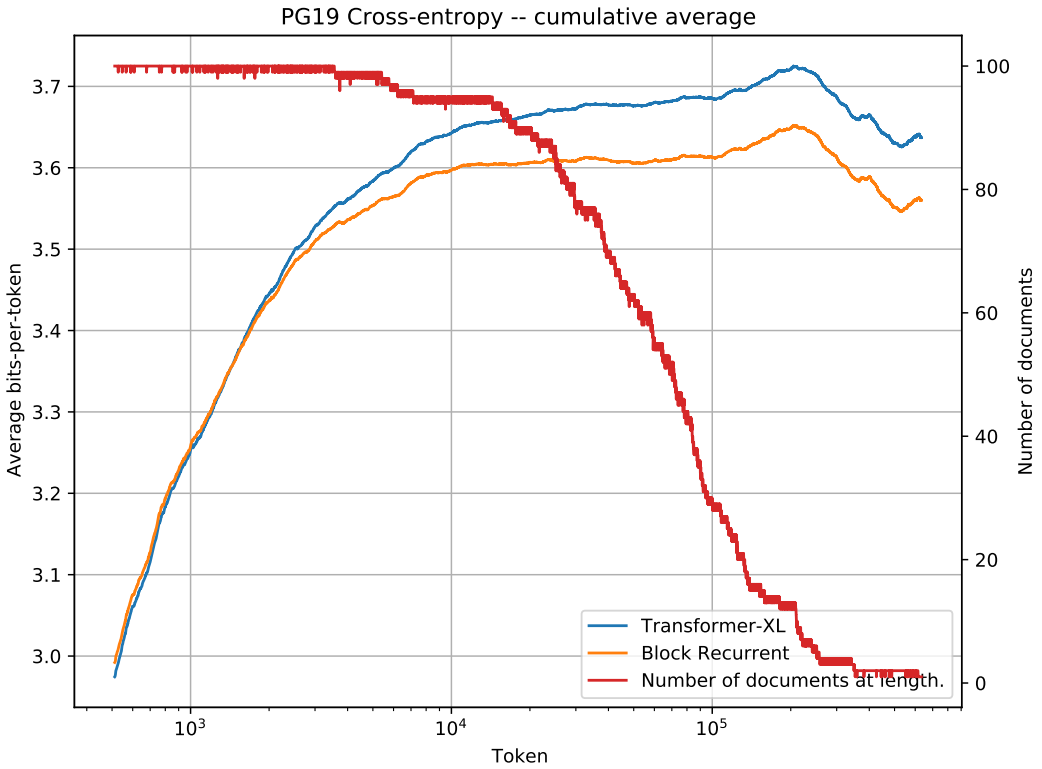

Figure 6: Cumulative cross-entropy on PG19 of a 13-layer Transforemer-XL and Block Recurrent model. Though comparable at the first few thousand tokens, the recurrent model performs better at longer sequences. In red we show the number of documents at a given token length.

fact that is only partially true. Depending on how it is configured, SentencePiece may normalize text by mapping multiple whitespace characters (e.g. tab) to space, merge internal whitespace characters, or strip all newlines from the text.

Thus, depending on vocabulary, a model trained with a SentencePiece tokenizer may or may not be predicting whitespace and newlines. If it does predict them, then that whitespace is not captured by the word count, which can affect the WLP numbers.

Measuring bits-per-character (BPC) rather than WLP is a potentially simpler and fairer comparison that works with any data set and tokenizer, but it runs into the same problem with whitespace. If the tokenizer merges or strips whitespace from the input text, as many SentencePiece vocabularies do, then it is not possible to get a consistent character count across different vocabularies.

I.1 Effect of vocabulary on model perplexity

The vocabulary itself can also make a significant difference. We argue that the tokenizer and vocabulary should be considered as part of the model, and comparisons using either WLP or BPC are only valid between models that use the same tokenizer and vocabulary. Using WLP to judge model innovations without taking into account other differences can lead to incorrect conclusions. We give an empirical example of how using different vocabularies can lead to significant differences in WLP.

The total cross-entropy loss $L_S = -\sum_{t=0}^{N_S} \log(p_t|p_{<t})$ where S is the chosen tokenization scheme and N_S is the number of tokens in the data when tokenized by S. Models using the same tokenization scheme can be compared by their *token-level perplexity* $\text{PPL}_S = \exp(\frac{L_S}{N_S})$ where $\frac{L_S}{N_S}$ is the *average loss*. Lower perplexity is better.

WLP is defined to be $\exp(\frac{L_S}{N_W})$ where N_W is the number of "words" as defined by the authors of the PG-19 datasets. For the PG-19 test set, $N_W = 6,966,499$ [15]. Thus, WLP is simply a rescaling of the average loss, i.e. WLP = $\exp(\frac{L_S}{N_S}\frac{N_S}{N_W})$, where we'll refer to $\frac{N_S}{N_W}$ as the *scale factor* – the number of tokens per word.

I.1.1 Vocabulary size

The size of the vocabulary has three obvious effects on the model. First, a larger vocabulary has more parameters, and thus can store more information in the embedding table.

Second, a larger vocabulary will typically have longer and rarer words, so the average length of each token (characters-per-token) will also be longer, and the *scale factor* (tokens-per-word) will be smaller. The context length of the model *in tokens* is usually fixed, so having longer tokens means that the model can attend over longer distances in the input text.

Third, having longer tokens also means that the model will process more of the training data on each training step (again because the number of tokens per step is fixed), and thus complete more epochs over the data set.

I.1.2 Vocabulary quality

Even if two vocabularies are the same size, there may be a difference in vocabulary quality. For example, if a multi-lingual vocabulary is used on English-language text, then only a portion of its total capacity will be used to capture English words, and it will behave much like a smaller vocabulary.

A more subtle issue is that there are many ways in which text can be tokenized. A tokenizer which does a good job of capturing natural semantic segmentation into subwords can be expected to perform better than one that doesn't, e.g. "`walk|ed`" vs. "`wal|ked`."

I.2 Experimental results

We train the `Rec:fixed:skip` model with a segment length of 4096 for 500k steps employing an inverse square root decay schedule while only varying the choice of vocabulary (all other hyperparameters are as in Section 4).

The byte-level vocabulary does not use a tokenizer, and operates on the raw ASCII text. The T5 vocabulary [39] was trained on multi-lingual data from Common Crawl (commoncrawl.org), consists of 32k tokens, and does not preserve whitespace. We also test on a 32k sized version of the LaMDA vocabulary [54]. To test the effect of vocabulary size, we also train a number of SentencePiece models on the PG19 training set with sizes ranging from 512 to 128k.

Results are shown in Table 5 and Figure 7. For SentencePiece vocabularies trained on PG19, we observe that a larger vocabulary has a significant effect on the scale factor, although the effect starts to diminish with more than 32k entries (see Figure 7). The average bits per token increases as the tokens become longer.

Table 5: Test perplexity of the `Rec:bias:skip` model when using different tokenizers. Results are given in bits-per-token, so WLP is $2^{(\text{bits-per-token} \cdot \text{scale factor})}$. STE stands for Tensorflow's deprecated SubwordTextEncoder as used by other baselines; SPM stands for SentencePiece model [44].

vocab	size	token count	scale factor	avg bits-per-token	WPL
T5 [39]	32k	10,523,460	1.5106	3.525	40.08
LaMDA [52]	32k	12,053,681	1.7302	3.121	42.23
PG19 STE	32k	11,097,364	1.5930	3.352	40.49
PG19 SPM	32k	10,229,476	1.4684	3.647	40.93
PG19 SPM	128k	9,638,472	1.3835	3.830	39.37
PG19 SPM	96k	9,711,259	1.3940	3.801	**39.36**
PG19 SPM	64k	9,856,687	1.4149	3.766	40.18
PG19 SPM	32k	10,229,476	1.4684	3.647	40.93
PG19 SPM	8k	11,559,455	1.6593	3.289	43.94
PG19 SPM	4k	12,622,101	1.8118	3.021	44.43
PG19 SPM	1k	15,865,499	2.2774	2.477	49.91
PG19 SPM	512	18,124,332	2.6016	2.125	46.16
bytes	256	41,288,269	5.9267	0.960	51.61

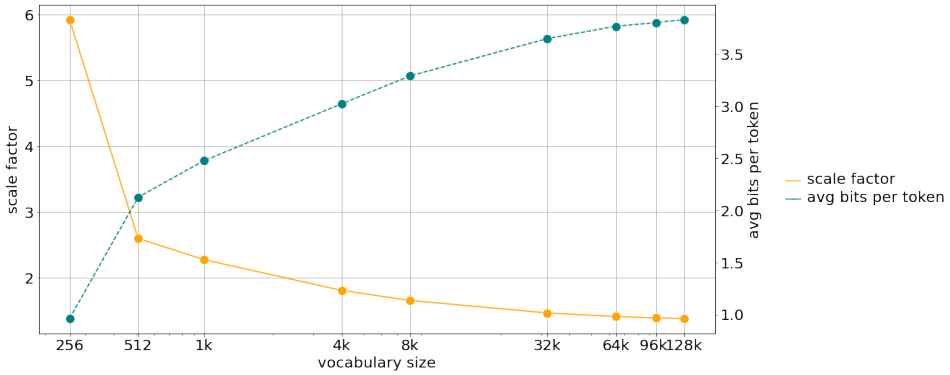

Figure 7: Word-perplexity scale factor and average bits per token for sentencepiece vocabularies trained on PG19 with different sizes.