
BLOCK-RECURRENT TRANSFORMERS

A PREPRINT

DeLesley Hutchins^{*2}, Imanol Schlag^{*3†}, Yuhuai Wu², Ethan Dyer¹, Behnam Neyshabur¹

¹ Google Research, Blueshift Team

² Google Research

³ The Swiss AI Lab IDSIA, SUPSI & USI

{delesley, yuhuai, neyshabur, edyer}@google.com
imanol@idsia.ch

March 16, 2022

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, 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.

1 Introduction

Transformers have mostly replaced recurrent neural networks (RNNs), such as LSTMs [1], on tasks that involve sequential data, most notably in the domain of natural language processing. There are several reasons for their success.

First, transformers are feed-forward models that process all elements of the sequence in parallel, and are thus more efficient to train on modern accelerator hardware; this was one of the primary motivations of the original transformer paper [2]. A transformer can be trained efficiently even at a batch size of 1, since batching can be done along sequence length. 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, transformers have much higher bandwidth for transmitting information from past tokens to future ones. 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

^{*}Equal Contribution

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

can encode about the previous sequence, and that size cannot be easily increased, because the computational cost of vector-matrix multiplication is quadratic with respect to the size of the state vector. In contrast, a transformer can attend directly to past tokens, and does not suffer from this limitation.

Third, the attention mechanism operates over longer distances. The forget gate in an LSTM erases a small portion of the previous state after each token, which 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, which is usually at least 512 or 1024 [3].

Despite these advantages, transformers also have disadvantages. Although attention is more effective than an LSTM over long distances, it still has limits. The computational complexity of the standard self-attention mechanism is quadratic with respect to the size of the attention window, which is equal to the sequence length in a vanilla transformer. Therefore, the size of the window cannot be easily scaled up; this is a limiting factor when attempting to process long documents, such as books, technical articles, or source code repositories. This limitation is well-known in the literature, and a wide variety of mechanisms have been proposed which aim to replace the regular attention layer with a more efficient version that can be scaled to cover longer sequences; see Section 2 for a brief survey.

In this work, we describe an alternative approach to modeling long sequences. Instead of attending directly to all previous tokens, we use a recurrent architecture. 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. The block size can be quite large; we use blocks of 512 tokens in our experiments. Within a block, all tokens are processed in parallel, which resolves efficiency issues caused by a lack of parallelism. 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 now orders of magnitude smaller.

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.

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. The cell uses cross-attention to attend to both the recurrent state and the input tokens. There are **a few tweaks that are necessary to stabilize model training**; see Sections 3.3 and 3.5 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 usage remain unchanged.

Finally, 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.

The sliding window mechanism [6], which we use in this paper, performs attention only within the context of a local *window*, thus reducing complexity to be quadratic with respect to the window size, rather than total sequence length. Transformer XL [7] uses a somewhat similar idea, but caches keys and values from the current training step for use on the next step. Caching allows long documents that do not necessarily fit in GPU or TPU memory to be processed incrementally over multiple training steps. When using a cache, the theoretical context available to the model is longer than the sequence length, but the cache itself is not differentiable, which makes it more difficult for the model to learn long-range dependencies.

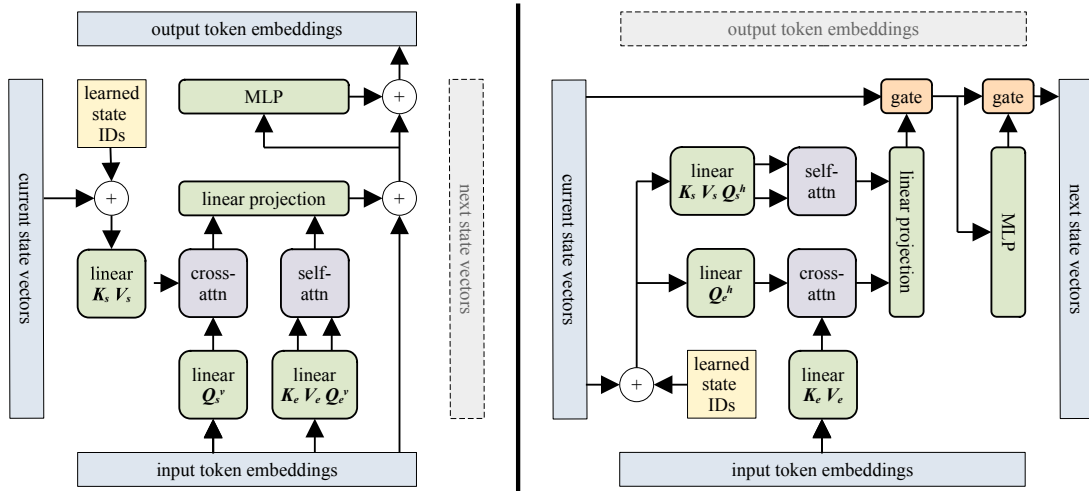


Figure 1: Illustration of our recurrent cell. The left side depicts the vertical direction (layers stacked in the usual way) and the right side depicts the horizontal direction (recurrence). Notice that the horizontal direction merely rotates a conventional transformer layer by 90° , and replaces the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

One way to reduce the cost of attention over long distances is to restructure it. The equation for attention is (roughly) $\text{softmax}(\mathbf{Q}\mathbf{K}^T)\mathbf{V}$ where \mathbf{Q} , \mathbf{K} , and \mathbf{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 $\mathbf{Q}(\mathbf{K}^T\mathbf{V})$, where $(\mathbf{K}^T\mathbf{V})$ can be computed incrementally via a cumulative sum over the sequence length. The complexity of “linearized” attention thus becomes $O(N)$, rather than $O(N^2)$, where N is the sequence length [8]. Following this line of reasoning, there have been several proposals that approximate the softmax [9, 10] or replace it [11]. However, on auto-regressive language modeling tasks, these so-called *linear transformers* often have reduced accuracy when compared to a classical transformer with softmax attention [11].

Other techniques attempt to exploit sparsity. Sparse strategies such as Big Bird [12] select only a subset of tokens to attend to. Routing Transformers [13] use clustering to select the subset, while Reformer [14] relies on hashing. Hierarchical mechanisms [15] combine multiple tokens into phrases or sentences to reduce sequence length. Expire-span [16] learns to prune far-away tokens that the model has labelled as “unimportant”. Memorizing transformers [17] replace dense attention with a non-differentiable 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 [18], and Combiner [19] apply pooling or averaging over tokens at longer distances. Linformer [20] applies a linear transformation to the key and value matrices to reduce the sequence length. Compressive transformers [21] and funnel transformers [22] apply additional learned compression layers to compress the sequence.

Recurrence is a relatively simple mechanism for dealing with sequences of arbitrary length, and it is attractive for several reasons. A recurrent function f is a function from $(\mathcal{S}, \mathcal{X}) \rightarrow (\mathcal{S}, \mathcal{Y})$, which takes a current state $s_t \in \mathcal{S}$ and an input $x_t \in \mathcal{X}$, and produces a next state $s_{t+1} \in \mathcal{S}$ and an output $y_t \in \mathcal{Y}$. Recurrence is $O(N)$ during both training and inference, and it runs in constant space during inference. The state s_t encodes a fixed-size summary of the sequence seen thus far. Most recurrent architectures have an inductive bias wherein the state s_t preferentially preserves more recently seen information, which is particularly useful on language modeling tasks.

A few lines of research have combined the transformer architecture with recurrence in some way. The feedback transformer [23] allows lower layers to attend to the output of the topmost layer. Feedback is a powerful mechanism because it combines the advantages of attention with a deep recurrent function: each token is run through many transformer layers, before the result is handed to the next token. Just like RNNs, Feedback has minimal cost at inference time, but it is unfortunately very slow to train because tokens must be processed sequentially.

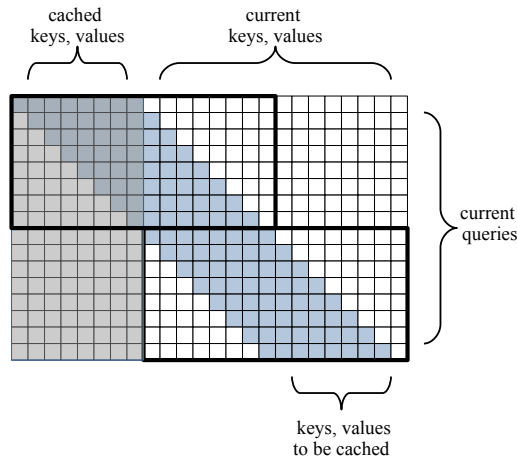


Figure 2: Sliding window, where sequence length $N = 16$, window size $W = 8$. Keys and values for the first W shaded tokens were computed and cached on the previous training step; the remaining N unshaded tokens are from the current input. Instead of a single $N \times (W + N)$ attention matrix, attention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

Simple Recurrent Units [24, 25] 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 [26]. “Sandwich models” alternate between transformer and RNN layers and out-perform both transformers and RNNs on tasks involving source code [27].

The Recurrent Fast Weight Programmer [28] builds on fast weight attention [29] and proposes various recurrent models derived from their Fast Weights inspired linear Transformer [11]. The R-Transformer introduces an additional local RNN which can be computed in parallel in order to better model sequential structure [30].

Recurrent architectures are usually slow because they process tokens sequentially, which is inefficient on parallel hardware. One way to improve parallelism is to process blocks of tokens in parallel. To the best of our knowledge, this idea is relatively underexplored. In the context of translation, [31] translate one sentence at a time while maintaining a shorter representation of all previously translated sentences in an encoder-decoder style mechanism based on the popular T5 transformer model [32]. Staircase Attention [33] also operates on blocks of tokens. Through a clever arrangement, a layer in the Staircase model receives so-called *backward* tokens as input, which are the outputs of the same layer from the previous block, and is thus block-wise recurrent.

3 Method

3.1 Sliding Attention

Our model is based on sliding-window attention [6], which is an extension of ideas from Transformer-XL [7], and is illustrated in Figure 2.

Given a long sequence of N tokens ($N = 4096$ in our experiments), 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 $W \ll N$). Thus, the attention computation can be optimized by breaking it into smaller tiles along the diagonal. The sequence 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 sequence length N .

As is done with Transformer-XL, when reading a long document, like a book, we divide the document up into segments of length N , and process them sequentially over a number of training steps. After each training step,

the last W key-value pairs are stored in a non-differentiable cache, and used as the “previous block” when processing the first block of the next segment on the next training step. Sliding-window attention, with a cache, implements a form of truncated backpropagation through time [34] over long documents.

Receptive field of sliding attention. 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 sliding window architecture has a context length of W for every token.

The difference between sliding-window attention and Transformer-XL is that the latter only processes one block per training step (i.e, the block length and segment length are the same), while the sliding window processes a longer segment that contains multiple blocks. Because the initial cached block is not differentiable, the sliding window gains some benefit from being able to backpropagate gradients over multiple blocks, a capability that becomes much more important when adding recurrence.

In a transformer with sliding-window attention, the topmost layer can attend to at most W tokens away to the previous layer. Thus, the theoretical *receptive field* (the maximum distance that information can propagate through the model) is $W \cdot L$, where L is the number of layers. In our experiments, sliding window transformers do not seem to utilize the full theoretical receptive field, most likely because attention focuses mainly on nearby tokens. With recurrence, the receptive field is unlimited, at least in theory. We observe that the Block-Recurrent Transformer seems to make much better use of long-range context, as evidenced by a substantial improvement in perplexity.

3.2 Recurrent Cell

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 size, and a set of S “current state” vectors. It also 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 [2]. 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 when processing very long documents and which has been central to the success of LSTMs [35].

Note that the presence of gates is why we do self-attention and cross-attention in parallel. Doing them sequentially, as is standard practice, would introduce a third gate in the horizontal direction, which reduced model accuracy in our experiments.

A Block-Recurrent Transformer *layer* breaks the N tokens of each segment into blocks, and processes the blocks 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. Multiple layers can also be stacked vertically in the usual fashion, either with other recurrent layers, or with conventional transformer layers.

Sharing of keys and values. Keys and values are shared between the vertical and horizontal directions. One set of keys and values (K_e, V_e) are computed from the input token embeddings, and are used for both self-attention in vertical direction, and cross-attention in the horizontal direction. Another set of keys and values (K_s, V_s) are computed from the recurrent state vectors, and are used for both self-attention in the horizontal direction, and cross-attention in the vertical direction.

Queries are not shared; the keys K_e (for tokens) and K_s (for states) live in different vector spaces, so we compute different queries for them. Thus, there are four separate sets of queries: Q_e^v and Q_s^v for self-attention and cross-attention in the vertical direction, and Q_s^h and Q_e^h for self-attention and cross-attention in the horizontal direction.

Recurrence is integrated with the sliding window mechanism described in Section 3.1. Thus, keys and values for the token embeddings (K_e , V_e) actually consist of keys and values for tokens from the current block, concatenated with keys and values from the previous block. In the horizontal direction, the cell will similarly cross-attend to both blocks, which means that any given input token will be seen twice by the recurrent cell.

Our hope is that using overlapping windows in this way minimizes any artifacts caused by the fact that recurrence occurs at artificial boundaries. In other words, the rigid 512-token window boundary is not necessarily aligned with the end of a sentence, or even the end of a word.

Finally, when processing a document that is longer than the segment length N , the final set of state vectors from the last block are stored in a non-differentiable cache and then used as the initial state for the first block on the next training step, just like the cached keys and values in Transformer-XL [7].

3.3 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. In other words, given two similar state vectors, the model will issue similar queries, and update both vectors in a similar way. When iterating the recurrent cell, there is a failure mode in which the initial state vectors become more and more similar until they are approximately identical.

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. Ultimately, 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 position embeddings to the tokens, because global position embeddings don’t work well for long sequences [7]. Instead, we add a T5-style relative position bias [32] to the self-attention matrix in the vertical direction; the bias is a learned function of the relative distance between query and key. When the recurrent states cross-attend to input tokens, there is no position bias, because the relative distance between “state” and “token” is undefined.

As is standard practice, we apply layernorm before projecting to keys, values, and queries, and before applying the final MLP. We also apply pre-attention and post-MLP dropout, as well as dropout on the self-attention matrix. We also normalize queries and keys as described in [36]; we found that normalization improved the stability of Transformer-XL when used with a relative position bias.

3.4 Gate Type

We experimented with two different gating mechanisms for the recurrent cell.

Fixed gate. The fixed gate uses a learned convex combination that is similar to highway networks [37].

$$z_t = W^{(z)} h_t + b^{(z)} \quad (1)$$

$$g = \sigma(b^{(g)}) \quad (2)$$

$$c_{t+1} = c_t \odot g + z_t \odot (1 - g) \quad (3)$$

where $W^{(z)}$ is a trainable weight matrix, $b^{(z)}$ and $b^{(g)}$ are trainable bias vectors, σ is the sigmoid function, c_t is the current cell state at time t , \odot is the element-wise multiplication, and h_t is the current input to the gate. In our model, h_t is either the output of attention, in which case $W^{(z)}$ is the linear projection that feeds into the gate, or h_t is the output of the hidden layer of the MLP, in which case $W^{(z)}$ is the final layer of the MLP.

Note that the bias $\mathbf{b}^{(g)}$ is a simple learned vector, so the value of \mathbf{g} does **not** depend on either the current state \mathbf{c}_t , nor the input \mathbf{h}_t and thus remains constant after training. The fixed gate thus implements an exponential moving average over previous blocks.

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

$$\mathbf{z}_t = \tanh(\mathbf{W}^{(z)}\mathbf{h}_t + \mathbf{b}^{(z)}) \quad (4)$$

$$\mathbf{i}_t = \sigma(\mathbf{W}^{(i)}\mathbf{h}_t + \mathbf{b}^{(i)} - 1) \quad (5)$$

$$\mathbf{f}_t = \sigma(\mathbf{W}^{(f)}\mathbf{h}_t + \mathbf{b}^{(f)} + 1) \quad (6)$$

$$\mathbf{c}_{t+1} = \mathbf{c}_t \odot \mathbf{f}_t + \mathbf{z}_t \odot \mathbf{i}_t \quad (7)$$

where $\mathbf{W}^{(z)}$, $\mathbf{W}^{(i)}$, $\mathbf{W}^{(f)}$ are trainable weight matrices, and $\mathbf{b}^{(z)}$, $\mathbf{b}^{(i)}$, $\mathbf{b}^{(f)}$ are trainable bias vectors. The LSTM gate is strictly more expressive, because the values of \mathbf{f}_t and \mathbf{i}_t depend on the current input \mathbf{h}_t . In our model, \mathbf{h}_t depends on \mathbf{c}_t , so the LSTM gate also depends indirectly on \mathbf{c}_t .

3.5 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.

Proper gate initialization also depends on the optimizer. We use the Adafactor optimizer [38], 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 \mathbf{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.

3.6 Gate Configuration

We also 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, which seems to reduce model accuracy in some cases.

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.

3.7 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 standard transformer layers with sliding attention, as described in Section 3.1. We use a 12-layer transformer with recurrence on layer 10. The first 9 layers do pre-processing of the input sequence without recurrence, and the final 2 layers do post-processing after the recurrence.

The 12-layer Block-Recurrent Transformer has almost exactly the same cost, in both parameters and FLOPS, as a 13-layer 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.

Feedback. Another variation, inspired by the Feedback Transformer [23], allows all layers to cross-attend to the recurrent state. Unlike the original Feedback Transformer, we implement feedback over blocks, rather than individual tokens.

In this variation, the entire stack of 12 layers is applied to one block of tokens, the recurrent state is extracted from the recurrent layer, and the state is then 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 [23].

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.

4 Results

We tested the Block-Recurrent Transformer on three different data sets of long documents: PG19, arXiv, and GitHub. The PG19 dataset [39] contains full-length books written prior to 1919 from project Gutenberg, and we do both byte-level and token-level language modeling. The arXiv dataset [17] is a corpus of technical papers downloaded via the arXiv Bulk Data Access¹, and filtered to include only articles labeled as “Mathematics” and whose \LaTeX source is available. The GitHub dataset [17] is a corpus of source code from different GitHub repositories with open-source licenses. All files in each 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. The baseline model is 12-layer transformer, with 8 heads of size 128, embedding vectors of size 1024, an MLP with a hidden layer of size 4096, and the relu nonlinearity. We use the Adafactor optimizer [38], a learning rate schedule with inverse square root decay, 1000 warmup steps, and a dropout rate of 0.05. Except where otherwise noted, we train each model for 500k steps. Results are shown in Table 1.

For PG19, we do both token-level modeling with a vocabulary size of 32k, and character-level modeling. We use a learning rate of 1.0, and train with 32 replicas on Google V4 TPUs; training takes approximately 24 hours.

For arXiv, we do token-level modeling with a vocabulary of 32k. 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) sequence lengths.

The GitHub dataset 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 learning rate of 0.25, and a batch size that is 8x larger than for PG19 and arXiv (see Section 4.2). Due to resource constraints, these models ran only for 250k steps and we did not run the full set of experiments.

¹https://arxiv.com/help/bulk_data

Model vocab size	sequence length	window length	step time (relative)	PG19		arXiv tokens 32k	GitHub tokens 32k
				bytes 256	tokens 32k		
XL:512	512	512	0.88	1.01	3.62	1.45	1.28
XL:1024	1024	1024	1.20	0.997	3.59	1.37	-
XL:2048	2048	2048	2.11	0.990	3.58	1.31	-
Slide:12L	4096	512	0.93	0.989	3.60	1.43	-
Slide:13L			1.00	0.989	3.57	1.42	1.19
Rec:lstm:dual	4096	512	1.06	0.985	3.56	1.26	1.07
Rec:lstm:single			1.05	0.962	3.55	1.29	1.05
Rec:lstm:skip			1.00	0.969	3.56	1.31	-
Rec:fixed:dual			1.01	0.957	3.54	1.27	1.03
Rec:fixed:single			1.02	0.966	3.58	1.25	-
Rec:fixed:skip			0.99	0.952	3.53	1.24	1.03
Feedback:lstm:single	4096	512	1.40	0.977	3.50	1.22	1.05
Feedback:fixed:skip			1.35	0.935	3.49	1.24	1.11
Memorizing Transformer 32k	512	512	-	0.950	3.50	1.24	-

Table 1: Average **bits-per-token** (\log_2 perplexity) of each model when trained for 500k steps. Notice that the recurrent models 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.

On the arXiv and PG19 datasets, we trained some models up to 1 million steps, and they continue to improve, but the relative rankings don't change. On PG19 byte-level, the `Feedback:fixed:skip` model achieves a **bits-per-character of 0.910** at 1 million steps.

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. `XL:512` is a 12-layer transformer with a window size of 512. 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 sequences of 512 tokens.

`XL:1024` and `XL:2048` are similar, but have window sizes of 1024 and 2048, respectively. These two models are more expensive than the recurrent model. As expected, increasing the window size improves perplexity, especially on the arXiv data set. This observation is in keeping with previous work using long-context memory on this dataset [17].

`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. Surprisingly enough, differentiability yields a significant improvement.

`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 sequence length. Notice that simply adding another layer with more parameters yields a much smaller improvement than adding recurrence.

Relative cost. All five baselines have roughly the same number of parameters: between 151 million (12 layer) and 164 million (13 layer). 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, which is very close in terms of both computational cost and memory use to our recurrent model.

4.2 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` runs at a batch size of 256 (8 per replica), while `Slide:12L` runs at a batch size of 32 (1 per replica) on PG19.

Training on longer segments has an advantage in that the model is able to backpropagate gradients over a longer distance, but it has a disadvantage in that it also decreases the batch size. Because long documents are processed sequentially over multiple training steps, if the batch size is too small, it is possible for the model to over-fit to one document before it starts processing the next one. The effect of reduced batch size shows up as additional noise in the plot of bits-per-token over time, which was a particular problem on the GitHub data set.

4.3 Recurrence

We compare the 5 baselines to all six gate configurations for the single-layer Recurrent Transformer (`Rec:gate:config`), and to two of the best-performing gate configurations for the Recurrent Transformer with feedback (`Feedback:gate:config`).

Our best-performing recurrent model on all data sets uses the `fixed` gate with the `skip` configuration, which reliably outperforms all five baselines and the other five gate configurations in terms of bits-per-token. This is especially notable because it is also the fastest configuration, having a slightly *lower* step time and fewer parameters than the 13-layer transformer-XL (because it does not have 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. On PG19 the `fixed:dual` configuration does almost as well as `fixed:skip`, with `fixed:single` lagging well behind the other two. This is perhaps due to the fact that the `dual` configuration can learn to mimic the `skip` configuration by setting the second gate to “remember”. However, the model doesn’t always learn this; on arXiv, `fixed:dual` is actually the worst of the three `fixed` gates.

Despite being theoretically more powerful, the LSTM gate tends to lag behind the `fixed` gate in all of our experiments, and there is no clear winner in terms of gate configuration. On PG19 and GitHub, `lstm:single` is the best performer, and `lstm:dual` is the worst, while on arXiv the situation is reversed.

4.4 Feedback

Adding feedback further improves model accuracy in most cases, but again, the improvement seems to depend on the data set. On PG19 the effect is significant, while on arXiv it is minimal, and on GitHub it’s actually worse than the non-feedback version. Because adding feedback comes at a significant computational cost, it is probably more worthwhile to scale the transformer in other directions, e.g., by adding more parameters.

The effect of feedback also depends on the gate configuration. In particular, feedback dramatically improves the performance of the LSTM gate, matching or even surpassing the `fixed:skip` model. 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.

4.5 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. We conclude that one layer of recurrence is sufficient for the model to extract most of the benefits of having a larger receptive field.

Reducing window size. Reducing the size of the sliding window in a conventional transformer dramatically reduces accuracy, 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 much smaller effect, because the model can use recurrence to compensate. Recurrence thus provides a larger benefit over the baseline at smaller window sizes.

Model	PG19 tokens	perplexity word-level	parameters	vocabulary size	vocabulary
Slide:13L:large	3.39	34.80	327M	32k	T5 [32]
Rec:lstm:single:large	3.36	33.72	375M		
Rec:fixed:skip:large	3.32	32.34	320M		
Compressive Trans. (36 layer)	-	33.6	?	32k	custom
Routing Trans. (22 layer)	-	33.2	?	98k	?

Table 2: Average bits-per-token and word-level perplexity of the larger model, trained for 200k steps. Fields marked “?” are unknown, and were not specified in other published work.

Changing the number of recurrent states. Reducing the number of recurrent states reduces the accuracy of the model. However, we also observed some training instability if the number of states was increased past a certain point. Finding the ideal number of recurrent states is a topic for future work.

Scaling up. Transformers are famously able to scale up to massive models containing billions of parameters [40]. We ran a brief scaling experiment, in which we doubled the number of heads, doubled the size of the MLP, and increased the number of replicas by a factor of 4, to see if recurrence still offers improvements at larger scales. We ran the larger model, which has 320 million parameters, for 200k steps on the PG19 token-level dataset; results are shown in Table 2.

4.6 Other Comparisons

The PG19 test set contains 6,966,499 words [21], which are broken into 10,523,460 tokens using our 32k vocabulary. Following the technique in [21], we convert bits-per-token to word-level perplexity in Table 2. Our 12-layer Rec:fixed:skip:large model compares favorably with the published results for the 36-layer Compressive Transformer [21], and the 22-layer Routing Transformer [13].

However, we wish to point out that on large data sets like PG19, such comparisons are not very meaningful. It is very easy to improve transformer performance by increasing the number of parameters [40], or the vocabulary size. Even with the same vocabulary size, the tokenizer is part of the model, so changing the vocabulary itself can make a significant difference. Large models are seldom trained to convergence, so the number of epochs and batch size matter a great deal, as do numerous hyperparameters, such as optimizer, learning rate, dropout, etc. Models which vary widely along multiple dimensions cannot be meaningfully compared, and published work sometimes does not include sufficient detail to allow precise replication.

We were able to run a completely fair comparison of the Block-Recurrent Transformer against the Memorizing Transformer [17], with a memory of size 32k. Number and size of layers, number of parameters, tokenizer and vocabulary, along with all hyperparameters were all set identically to the XL:512 baseline. 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. However, instead of using recurrence, the Memorizing Transformer does k -nearest-neighbor lookup into a large database of the previous 32k tokens.

We find that recurrence with feedback performs roughly on a par with the memorizing transformer, but trains considerably faster (Table 1). We do not report the exact speedup, because there are many ways of implementing approximate k -nearest-neighbor lookup, so relative speed is highly implementation-dependent.

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 that they have encountered thus far. 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.

Humans also process information in blocks, e.g. sentences or paragraphs, rather than individual characters or words. When a human reads a paragraph of text in a novel, they will first parse the information in the paragraph, then process and interpret the information using background knowledge from their mental model, and finally update their mental model with the new information.

Our recurrent architecture mimics this process. It takes a block of text, parses it by running it through a conventional transformer stack, and then uses cross-attention so that that tokens in the text can attend to recurrent states (i.e. the mental model). The recurrent states, in turn, are updated by cross-attending to the text.

Unfortunately, our current implementation of recurrence does not seem to be living up to its full potential. In theory, a transformer layer seems like a good candidate for a powerful recurrent function. The state vectors could potentially encode a set of facts that the model has extracted from the document that it is reading. The attention mechanism seems like a good way to query those facts, and a gate which can selectively choose to remember or forget can incrementally update the set of facts with new information from the input. Taken together, this mechanism would seem to have all of the elements needed to perform deep summarization and knowledge extraction from long texts.

Nevertheless, our model does not seem to be doing complex reasoning of this sort, 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. A recurrent cell with a `fixed` gate is thus mathematically somewhat similar to other forms of long-range attention, which compute an algorithmically tractable approximation of dense attention by averaging over many inputs at long ranges. The `fixed` gate is a bit more complex than that, because it can also attend to other states as well as the inputs, but it is still computing a moving average.

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 [41]; those large MLPs are computing something important. In a recurrent layer, removing the MLP actually improves the model. Not only is the MLP not computing anything useful, it seems to be adding unwanted noise that impedes learning.

The training instabilities that we observed provide further evidence that learning a good recurrent function is a difficult task. The `fixed:skip` configuration may perform better than the other configurations simply because it is the simplest one, and thus has an inductive bias to learning a simpler (and thus easier) function.

We conclude that training the recurrent layer to make full use of its capabilities for knowledge extraction and summarization will require further advances. Possible directions for future research are more sophisticated training tasks, auxilliary losses, or architectural improvements with a better inductive bias.

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; our model achieves a lower perplexity than a Transformer XL model with a window size of 2048, but it runs more than 2 times faster!

Furthermore, in contrast to other recently proposed transformer variants, the Recurrent Transformer is very easy to implement, since it consists mostly of ordinary transformer components and RNN gates.

Despite these successes, we 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] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Computation*, vol. 9, no. 8, 1997.
- [2] 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.

- [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] I. Beltagy, M. E. Peters, and A. Cohan, “Longformer: The long-document transformer,” *arXiv preprint arXiv:2004.05150*, 2020.
- [7] 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.
- [8] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, “Transformers are RNNs: Fast autoregressive transformers with linear attention,” in *ICML*, 2020.
- [9] K. M. Choromanski, V. Likhoshesterov, 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.
- [10] H. Peng, N. Pappas, D. Yogatama, R. Schwartz, N. A. Smith, and L. Kong, “Random feature attention,” in *ICLR*, 2021.
- [11] I. Schlag, K. Irie, and J. Schmidhuber, “Linear Transformers are secretly fast weight programmers,” in *ICML*, 2021.
- [12] 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.
- [13] 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.
- [14] N. Kitaev, L. Kaiser, and A. Levskaya, “Reformer: The efficient transformer,” in *International Conference on Learning Representations*, 2020.
- [15] 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.
- [16] 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.
- [17] Y. Wu, M. Rabe, D. Hutchins, and C. Szegedy, “Memorizing transformers,” in *ICLR*, 2022.
- [18] 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.
- [19] 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.
- [20] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, “Linformer: Self-attention with linear complexity,” *CoRR*, vol. abs/2006.04768, 2020.
- [21] J. W. Rae, A. Potapenko, S. M. Jayakumar, C. Hillier, and T. P. Lillicrap, “Compressive transformers for long-range sequence modelling,” in *ICLR*, 2020.
- [22] Z. Dai, G. Lai, Y. Yang, and Q. Le, “Funnel-transformer: Filtering out sequential redundancy for efficient language processing,” in *NeurIPS*, 2020.
- [23] 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.
- [24] T. Lei, Y. Zhang, S. I. Wang, H. Dai, and Y. Artzi, “Simple recurrent units for highly parallelizable recurrence,” in *EMNLP*, 2018.
- [25] T. Lei, “When attention meets fast recurrence: Training language models with reduced compute,” in *EMNLP*, 2021.

- [26] 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.
- [27] V. J. Hellendoorn, P. Maniatis, R. Singh, C. Sutton, and D. Bieber, “Global relational models of source code,” in *ICLR*, 2020.
- [28] K. Irie, I. Schlag, R. Csordás, and J. Schmidhuber, “Going beyond linear transformers with recurrent fast weight programmers,” in *confNEU*, 2021.
- [29] 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.
- [30] Z. Wang, Y. Ma, Z. Liu, and J. Tang, “R-transformer: Recurrent neural network enhanced transformer,” *arXiv preprint arXiv:1907.05572*, 2019.
- [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] 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.
- [33] D. Ju, S. Roller, S. Sukhbaatar, and J. Weston, “Staircase attention for recurrent processing of sequences,” *arXiv preprint arXiv:2106.04279*, 2021.
- [34] R. J. Williams and J. Peng, “An efficient gradient-based algorithm for on-line training of recurrent network trajectories,” *Neural Computation*, 1990.
- [35] 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.
- [36] A. Henry, P. R. Dachapally, S. S. Pawar, and Y. Chen, “Query-key normalization for transformers,” in *EMNLP*, 2020.
- [37] 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.
- [38] N. Shazeer and M. Stern, “Adafactor: Adaptive learning rates with sublinear memory cost,” in *ICML*, 2018.
- [39] 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.
- [40] 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.
- [41] 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.