Memformer: A Memory-Augmented Transformer for Sequence Modeling

Anonymous ACL submission

Abstract

Transformers have reached remarkable success in sequence modeling. However, these models have efficiency issues as they need to store all the history token-level representations as memory. We present Memformer, an efficient neural network for sequence modeling, that utilizes an external dynamic memory to encode and retrieve past information. Our model achieves linear time complexity and constant memory space complexity when processing long sequences. We also propose a new optimization scheme, memory replay back-propagation (MRBP), which promotes long-range back-propagation through time with a significantly reduced memory requirement. Experimental results show that Memformer has achieved comparable performance compared against the baselines by using 8.1x less memory space and 3.2x faster on inference. Analysis of the attention pattern shows that our external memory slots can encode and retain important information through timesteps.

1 Introduction

Memory plays a fundamental role in human cognition. Humans perceive and encode sensory information into a compressed representation stored in neurons, and later we effectively retrieve the stored information to accomplish various tasks. The formation of memory involves complex cognitive processes. Modeling and studying the behavior of human memory is still a challenging research problem in many areas.

Many researchers have attempted to incorporate memory systems in artificial neural networks. Early works like recurrent neural networks (RNN) (Rumelhart et al., 1988) including LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Chung et al., 2014) model temporal sequences with their internal compressed state vector as memory. However, they are limited in preserving the long-term information due to the memory bottleneck. To alleviate this limitation, more powerful memory network architectures such as Neural Turing Machine (NTM) (Graves et al., 2014), Differential Neural Computer (DNC) (Graves et al., 2016) have been proposed by leveraging a large external dynamic memory. Unfortunately, due to their complex memory interaction mechanism, they are not widely used for down-stream tasks at present.

More recently, Vaswani et al. (2017) propose Transformer by discarding the use of recurrence and memory. Instead, it computes all the $\mathcal{O}(N^2)$ paired dependencies in a sequence with self-attention (Bahdanau et al., 2015). Transformers have achieved great success in various natural language processing tasks. Nevertheless, the quadratic computation complexity can be costly. Some works try to address the limitations of self-attention, including Reformer, Sparse Transformer, Longformer, Linformer (Child et al., 2019; Kitaev et al., 2020; Wang et al., 2020), etc. They successfully reduce the complexity of self-attention and thus enable processing longer sequences. However, most of them still require linear memory space complexity.

Transformer-XL (Dai et al., 2019) re-introduces the concept of memory and recurrence. It caches each layer's hidden states of self-attention into a fixed-size queue and re-uses them in the later attention computation. However, the memory as raw hidden states cannot effectively compress high-level information. Thus, Transformer-XL in practice needs a massive memory size to perform well, and spends huge computation in using its memory. Compressive Transformer (Rae et al., 2020) improves upon Transformer-XL by further compressing its memories into fewer vectors via a compression network. However, as mentioned in the papers, both Transformer-XL and Compressive Transformer discard the information from the distant past, which causes a theoretical maximum temporal range given the fixed memory size.

Inspired by the previous external memory networks, we propose Memformer, which incorporates a fixed-size external dynamic memory combined with the recent Transformer architecture. Memformer interacts with its external dynamic memory through the memory reading and writing modules. Also, we introduce a forgetting mechanism to improve the effectiveness of memorizing new information. By utilizing recurrence and a fixed-size memory, our model has a theoretically infinite temporal range of memorization and implies a linear computation complexity and constant memory space complexity. As the traditional back-propagation through time (BPTT) has an unaffordable memory cost in our model, we introduce a new optimization scheme, memory replay back-propagation (MRBP), to significantly reduce the memory cost in training recurrent neural networks with large size of memory representations.

We evaluate Memformer on the autoregressive image generation and language modeling task. Experimental results show that Memformer performs on par with Transformer and Transformer XL with large memory size, while being much more efficient in terms of computation speed and memory space consumption. We also conduct an analysis showing that Memformer can retain information for an extended period.

2 Related Work

This section introduces some recent research directions that aim to alleviate the quadratic cost of self-attention. Moreover, we analyze their assumptions and limitations under the autoregressive setting to provide a broader view of these models.

2.1 Sparse Attention

One influential direction is to replace the full self-attention with sparse attention patterns to speed up the computation. Child et al. (2019) proposed Sparse Transformer, using a block sparse attention pattern to reduce the computation complexity to $\mathcal{O}(N\sqrt{N})$. Later, Longformer (Beltagy et al., 2020) and Big Bird (Zaheer et al., 2020) further explored this direction and proposed an even more sparse attention pattern to reduce the cost to $\mathcal{O}(N)$. They introduced global tokens to encode the information from the entire sequence and kept the self-attention to the closest k tokens and the global tokens to achieve linear complexity. Although linear sparse attention's theoretical soundness is proven for bidirectional encoders, it does not hold for the decoder. The main reason is that the global tokens cannot leak information to the future tokens in the autoregressive setting, where all the tokens can only see their previous tokens. Thus, linear sparse attention cannot guarantee a token to see its all past tokens. Only Sparse Transformer here with $\mathcal{O}(N\sqrt{N})$ complexity can theoretically cover all the past tokens for the sequence generation.

2.2 Linear Attention

Another direction is focusing on improving the softmax operation in the self-attention. Linformer (Wang et al., 2020) reduced the complexity to $\mathcal{O}(N)$ by projecting the entire sequence to a constant size of keys and values, but this method has not been applied to autoregressive decoding. Performer (Choromanski et al., 2020) and Linear Transformer (Katharopoulos et al., 2020) used a linear dot-product of kernel feature maps to replace softmax. However, for Linear Transformer under the autoregressive setting, it needs to compute the cumulative summation to aggregate the history information. This assumption is too strong if the input sequence is long and the length is not fixed. After thousands of steps, the numerical values can become very large due to the summation, causing overflow and gradient instability.

2.3 Recurrence and Memory

Applying recurrence and memory to Transformers is an orthogonal direction comparing to the efficient attention approaches. If the memory size is constant, recurrence enables the model to have constant memory complexity during inference. There are mainly two works exploring this direction. Transformer-XL (Dai et al., 2019) used relative positional encoding and consisted of a segment-level recurrence mechanism to encode beyond a fixed-length context. Compressive Transformer (Rae et al., 2020) extended from Transformer-XL by further compressing the previous cached hidden states to achieve a longer context. However, using past hidden states as memory would cause a theoretical maximum temporal range of context, meaning that a token is not guaranteed to see all the past tokens. Thus, in practice, Transformer-XL and Compressive Transformer need huge memory size to achieve good performance.

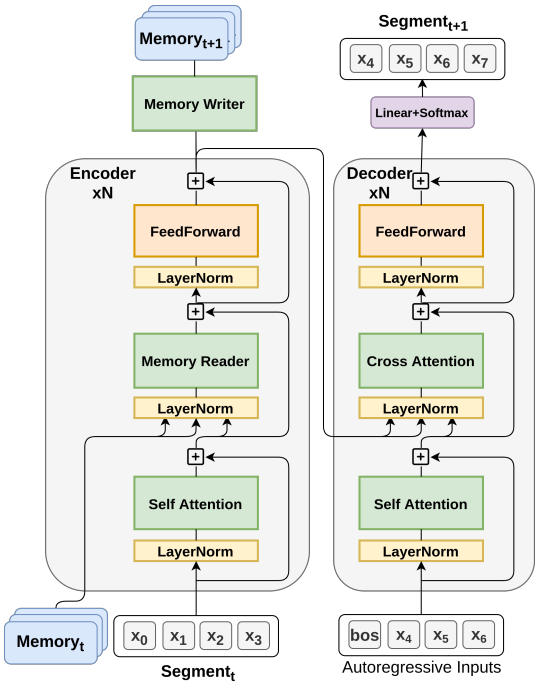


Figure 1: Memformer overall architecture for the encoder (left) and decoder (right). Transformer encoder is responsible to interact with the memory. Sequence modeling is achieved by predicting the next segment conditioned to the current segment and memory.

2.3.1 Dynamic Memorization

Within the scope of memory networks, there are dynamic memorization techniques. Different from Transformer-XL which stores the token-level history representations as memory, dynamic memorization does not have a theoretical upper bound for the temporal range. Neural Turing Machine (NTM) (Graves et al., 2014) and Differential Neural Computer (DNC) (Graves et al., 2016) are two early models that can control external memory resources to achieve long-lasting memory. However, their complex memory mechanisms cause them to be slow and unstable during training. In this work, we propose a dynamic memorization mechanism to achieve more efficient memory representations.

3 Methods

In this section, we first formalize the segment-level sequence modeling. Then, we present the memory reading and writing modules. Finally, we explain the memory replay back-propagation (MRBP) algorithm used for training.

3.1 Segment-level Sequence Modeling

Given a sequence of N tokens $x_1, x_2, \ldots, x_N$, an standard language model learns the joint probability of the sequence by taking the product of each token's probability conditioned to the previous tokens, which is defined as:

$$P(x) = \prod_t P(x_t | x_{<t})$$

When we have a large external memory system to store the history information, we cannot afford to interact with memory for every token. The workaround is to process a long sequence at the segment level. We can split a sequence into T segments and each segment has L tokens: $s_t = \{x_{t,1}, x_{t,2}, \ldots x_{t,L}\}$.

Because a bidirectional encoder is better at extracting word representations, we apply a Transformer encoder-decoder here. The encoder's role is to encode the segment s_t and inject the information into the memory M_t, while it also retrieves past information from the previous timestep's memory M_{t-1}. The encoder's final output will be fed into the decoder's cross attention layers to predict the token probabilities of the next timestep's segment s_{t+1} with standard language modeling.

$$M_t = \text{Encoder}(s_t, M_{t-1})$$

$$P(s_t | s_{<t}) = \prod_{n=1:L} P_{\text{Decoder}}(x_{t,n} \,|\, x_{t,<n}, M_{t-1})$$

$$P(x) = \prod_{t=1:T} P_{\text{Model}}(s_t | s_{<t})$$

At each timestep, given a segment as the input, the model needs to continue that segment by generating the next text segment, and the generated segment will be fed back into the model again. Since the memory stores all the past information, we can autoregressively generate all the token segments in a sequence. In this fashion, we can model the entire long sequence.

Figure 1 shows the overall architecture of Memformer. We will further explain each component and the implementation in the following sections.

3.2 External Dynamic Memory Slots

External dynamic memory (EDM) is a data structure that stores high-level representations of past inputs. "Dynamic" means that the model interactively encodes and retrieves the information from memory in a recurrent manner. This contrasts with static memory design, where the memory is stored statically and does not change during the inference.

In our design, we allocate a constant k number of vectors as the external dynamic memory. At each

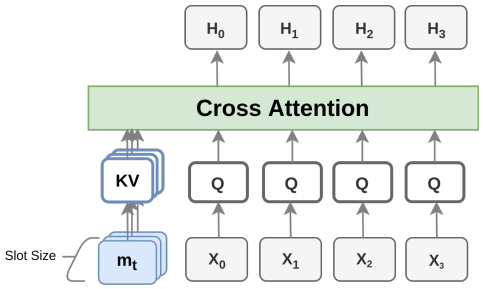


Figure 2: Memory Reading. The input sequence x attends over all the memory slots to retrieve the history information.

timestep t, we can have $M_t = [m_t^0, m_t^0, \ldots, m_t^k]$. For each sample in the batch, they have separate memory representations. Therefore, similar to RNN during inference, the memory consumption will be constant no matter how long the input sequence is. We name it memory slots because each slot is working individually to have different representations. The following sections will explain how the model manages to read and write this memory.

3.3 Memory Reading

For each input segment sequence, the model needs to read the memory to retrieve relevant past information. We leverage the cross attention to achieve this function:

$$Q_x, K_M, V_M = xW_Q, M_tW_K, M_tW_V \quad (1)$$

$$A_{x,M} = \text{MHAttn}(Q_x, K_M) \quad (2)$$

$$H_x = \text{Softmax}(A_{x,M})\, V_M \quad (3)$$

MHAttn refers to Multi-Head Attention. Memory slot vectors are projected into keys and values, and the input sequence x is projected into queries. Then the input sequence's queries attend over all the memory slots' key-value pairs to output the final hidden states. This enables the model to learn the complex association of the memory. Figure 2 shows the illustration.

Memory reading occurs multiple times as every encoder layer incorporates a memory reading module. This process ensures a higher chance of successfully retrieving the necessary information from a large memory.

3.4 Memory Writing

Memory writing involves a slot attention module to update memory information and a forgetting method to clean up unimportant memory information. Contrary to memory reading, memory writing only happens at the last layer of the encoder. This helps to store the high-level contextual representations into the memory. In practice, we append some classification tokens to the input sequence to better extract the sequence representations.

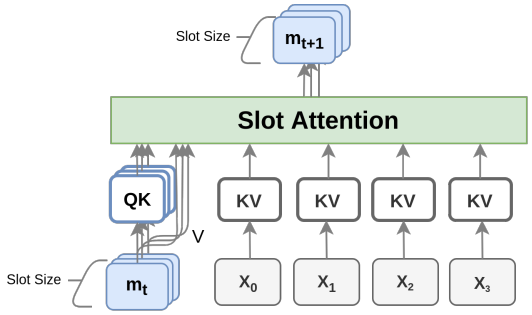


Figure 3: Memory Writing. Each memory slot attends over itself and the input sequence representations to produce the next timestep's memory slot.

3.4.1 Update via Memory Slot Attention

Figure 3 shows how memory is updated with the current segment's information. Each slot is separately projected into queries and keys. The segment token representations are projected into keys and values. Slot attention means that each memory slot can only attend to itself and the token representations. Thus, each memory slot cannot write its own information to other slots directly, as memory slots should not be interfering with each other.

$$Q_{m^i}, K_{m^i} = m^i W_Q, m^i W_K \quad (4)$$

$$K_x, V_x = xW_K, xW_V \quad (5)$$

$$A'_{m^i} = \text{MHAttn}(Q_{m^i}, [K_{m^i}; K_x]) \quad (6)$$

When we compute the final attention scores, we divide the raw attention logits with a temperature τ ($\tau < 1$). This operation sharpens the attention distribution, which makes the writing focusing on fewer slots or token outputs.

$$A_{m^i} = \frac{exp(A'_i/\tau)}{\sum_j exp(A'_j/\tau)} \quad (7)$$

Finally, the next timestep's memory is collected with by attention.

$$m_{t+1}^{i\,\prime} = \text{Softmax}(A_{x,M})\, [m_t^i; V_x] \quad (8)$$

The attention mechanism helps each memory slot to choose to whether preserve its old information or update with the new information.

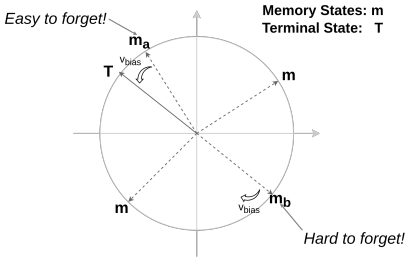


Figure 4: Illustration of forgetting. Memory slot m_a is easy to be forgotten, while m_b is hard to be forgotten.

3.4.2 Implementation of Memory Writer

Since each memory slot stores the information independently, we design a special type of sparse attention pattern. Each slot in the memory can only attend over itself and the encoder outputs. It aims to preserve the information in each slot longer over the time horizon. When a slot only attends itself during writing, the information will not be changed in the next timestep.

3.4.3 Forgetting Mechanism

Forgetting is crucial for learning as it helps to filter out trivial and temporary information to memorize more important information. LSTM introduces the forget gate (Gers et al., 2000) to reset its memory state, and the forget gate is proven to be the most important component in the LSTM (van der Westhuizen and Lasenby, 2018).

In this work, we introduce a forgetting mechanism called *Biased Memory Normalization* (BMN), specifically designed for our slot memory representations. We normalize the memory slots for every step to prevent memory weights from growing infinitely and maintain gradient stability over long timesteps. To help forget the previous information, we add a learnable vector v_{bias} to it. Also, naturally the initial state v^i_{bias} is after normalization.

$$m^i_{t+1} \leftarrow m^i_{t+1} + v^i_{\text{bias}}$$

$$m^i_{t+1} \leftarrow \frac{m^i_{t+1}}{||m^i_{t+1}||}$$

$$m^i_0 \leftarrow \frac{v^i_{\text{bias}}}{||v^i_{\text{bias}}||}$$

In Figure 4, we illustrate the forgetting mechanism with the learnable bias vector v_{bias}. Because of the normalization, all memory slots will be projected onto a sphere distribution. Here, we demonstrate with a 2D sphere for simplicity.

v_{bias} here controls the speed and the direction of forgetting. When adding v_{bias} to the memory slot, it would cause the memory to move along the sphere and forget part of its information. If a memory slot is not updated for many timesteps, it will eventually reach the terminal state T unless the new information is injected. The terminal state is also the initial state, and it is learnable.

The speed of forgetting is controlled by the magnitude of v_{bias} and the cosine distance between m'_{t+1} and v_{bias}. For example, m_b is nearly opposite to the terminal state, and thus would be hard to forget its information. m_a is closer to the terminal state and thus easier to forget.

Algorithm 1: Memformer Update

Input: rollout=$[x_t, x_{t+1}, \ldots, x_T]$: a list containing previous inputs
memories=$[M_t, M_{t+1}, \ldots, M_T]$: memory from the previous

▷ Initialize a list for back-propagation
replayBuffer = $[M_t]$
▷ Forward pass & no gradient
for $t = t, t+1, \ldots, T-1$ **do**
  M_{t+1}, _ = Model(x_t, M_t)
  replayBuffer.append(M_{t+1})
end
▷ Backward pass with gradient
$\nabla M_{t+1} = 0$
for $t = T, T-1, \ldots, t+1, t$ **do**
  ▷ Recompute
  M_{t+1}, O_t = Model(x_t, M_t)
  $loss$ = $f_{loss}(O_t)$
  $loss$.backward()
  M_{t+1}.backward(∇M_{t+1})
  ∇M_{t+1} = ∇M_t
end
▷ Update and pop the oldest memories
memories = replayBuffer
memories.pop()

3.5 Memory Replay Back-Propagation

Memformer relies on the external memory to process a sequence. At inference time, there is no additional memory cost because of the fixed-size memory design. Nevertheless, during training, it would require back-propagation through time (BPTT) so that the memory writer network can be trained to retain long-term information. The problem with

traditional BPTT is that it unrolls the entire computational graph during the forward pass and stores all the intermediate activations. This process would lead to impractically huge memory consumption for Memformer.

A favorable existing approach to eliminate this problem is gradient checkpointing (Chen et al., 2016). The algorithm can significantly reduce the memory cost of a large neural network. However, the standard gradient checkpointing still needs to compute all the nodes in the computational graph and store unnecessary activations during the forward pass. We propose Memory Replay Back-Propagation (MRBP), a more efficient variant of gradient checkpointing, by replaying the memory at each timestep to accomplish gradient back-propagation over long unrolls.

The algorithm takes an input with a rollout $x_t, x_{t+1}, \ldots, x_T$ and the previous memories $M_t, M_{t+1}, \ldots, M_T$ if already being computed. MRBP only traverses the critical path in the computational graph during the forward pass and recomputes the partial computational graph for the local timestep during the backward pass. It then obtains each timestep's memory and stores those memories in the replay buffer. The full algorithm is described in Algorithm 1. The experiments of memory cost reduction with MRBP is in the Appendix A.

4 Experiments

4.1 Computation and Memory Cost

We experimented the computation and memory cost of Vanilla Transformer, Transformer-XL, and Memformer. For Vanilla Transformer, it has to increase the input sequence length to encode more tokens. Its cost is $O(N^2)$ where N is the sequence length. Transformer-XL and Memformer use memory to store the history information, and the input sequence length is a constant value. Thus, their computation complexity is $O(N)$.

As a trade-off, for both Transformer-XL and Memformer, the memory size is then an important factor to affect the capacity of storing the history information. Transformer-XL stores the past hidden states for all layers as memory. If L is the number of layers, and K is the memory size, then the memory cost is $O(K \times L)$. Memformer only stores K vectors as memory with cost $O(K)$.

To better illustrate the difference, Figure 5 shows the number of FLOPs (floating-point operations) versus sequence length (left) and the GPU memory consumption versus memory size on the actual models (right). The sequence length is increased from 128 to $8,192$. Here, Memformer and Transformer-XL had the same number of parameters. From the figure, Vanilla Transformer has the largest computation cost growth. Memformer's costs grew linearly with the sequence length and achieved better efficiency than Transformer-XL. Then, we compared the GPU memory consumption. We tested the memory size ranging from 64 to $2,048$, with a batch size 16 for better visibility of memory cost difference. Transformer-XL's memory consumption grew rapidly with the memory size, while Memformer is more efficient with large memory size. In large memory size setting, Memformer uses 8.1x less memory space.

4.2 Autoregressive Image Generation

Model	#FLOPs (B)	Perplexity ↓
LSTM	52.5	1.698
Transformer Decoder	41.3	1.569
Transformer-XL		
memory=56	5.6	1.650
memory=224	15.6	1.618
memory=784	49.1	1.611
Memformer		
4 encoder+8 decoder	**5.0**	**1.555**
Memformer Ablation		
2 encoder+6 decoder		
memory=64	3.9	1.594
memory=32	3.9	1.600
memory=16	3.9	1.604
memory=1	3.9	1.627
4 encoder+4 decoder	3.6	1.628
w/o memory	1.8	1.745
temperature=1.0	3.9	1.612
w/o forgetting	3.9	1.630
w/o multi-head	3.9	1.626

Table 1: Results for autoregressive image generation. Our method only takes about 10% FLOPs of the best Transformer-XL model.

Recent research (Ramesh et al., 2021) demonstrates the approach of treating an image as a long sequence for image generation. Thus, we evaluated our model on the MNIST (LeCun and Cortes, 2010) image generation task with sequence modeling. Each image of size 28×28 was reshaped into a sequence of 784 tokens, and the 8-bit gray-scale was turned to a 256 vocabulary size.

For the baselines, LSTM had 4 layers and 512 hidden size. Transformer Decoder had 8 layers

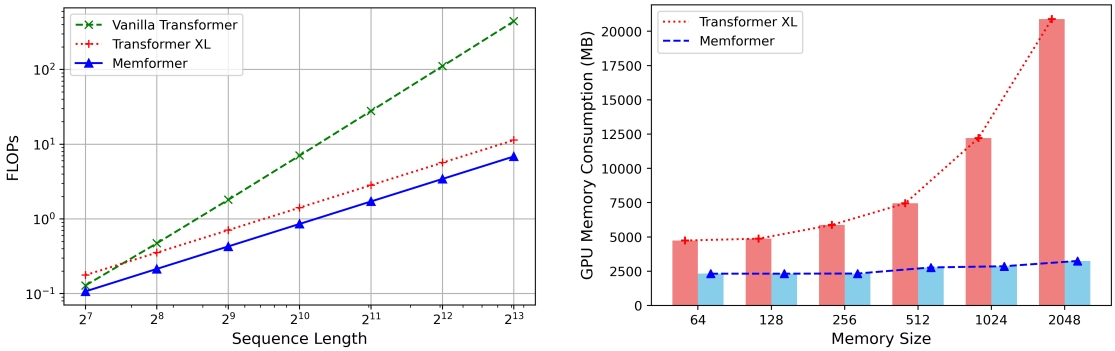


Figure 5: Comparison of the number of FLOPs and GPU memory consumption for Vanilla Transformer, Transformer-XL, and Memformer.

and could take all the 784 tokens as the input. Transformer-XL had 8 layers. All the models had the same 128 hidden size, 4 attention heads, 32 head size, and 256 feedforward size. Memformer was tested with default memory size 64. The default memory writer temperature was set to 0.25. We also conducted ablation studies to examine the contribution of various components.

Model	#FLOPs (B)	PPL ↓
Transformer-XL base		
memory=1600	250	23.95
memory=1024	168	23.67
memory=512	94	23.94
memory=256	58	25.39
memory=128	39	25.60
memory=32	26	27.22
Compressive Transformer		
memory= 512 compress=512	172	23.23
Memformer		
4 encoder + 16 decoder	**54**	**22.74**
Memformer Ablation		
4 encoder + 12 decoder	48	23.91
memory=512	35	23.30
w/o memory	31	25.57

Table 2: Experimental results on language modeling. Our method is 3.2 times faster here.

Table 1 shows the experimental results. We report median from three trials. Our Memformer with 4 layers of encoder and 8 layers of decoder achieved the best performance (1.555), while only using nearly 10% of FLOPs compared to the best Transformer XL baseline with memory size of 784 (1.611). Its performance was even better than the Transformer Decoder with the entire input sequence. We hypothesized that this observation was due to the extra parameters from the 4 layers of encoder. Therefore, we conducted an ablation study by having various numbers of encoder and decoder layers. If we reduce the number of decoder layers in Memformer (4 encoder+4 decoder), the performance dropped as shown (1.628). Results indicated that the number of decoder layers was important for the performance. Overall, Memformer outperformed Transformer-XL with a much lower computation cost.

The performance increased as the memory size increased. Moreover, when we completely removed the memory, Memformer performed terribly, signifying the importance of the encoded information in the memory. Other components such as forgetting mechanism, memory writer temperature, multi-head attention were proven to contribute to the final performance as well.

4.3 Language Modeling

We also conducted experiments on WikiText-103 (Merity et al., 2017), which is a long-range language modeling benchmark. It contains $28K$ articles with an average length of $3.6K$ tokens per article. Due to the limitation of computational resources, we are unable to experiment on the more recent PG19 (Rae et al., 2020) dataset. To study the computation cost and memory efficiency, we test with Transformer-XL base with 16 layers, 512 hidden size, 2,048 feedforward size, 64 head size, and 8 heads. The details are in the Appendix.

Memformer has the same hidden size, feedforward size, head size, and number of heads. We also re-implement a version of Compressive Transformer of the same size as there is no official implementation. The memory length is set to 512, and the compressive memory length is 512. The compression ratio is 4. The target sequence length for all models was set to 128. We test the performance under various memory sizes.

Table 2 summarizes the results on WikiText-103 test set. We report the number of inference FLOPs (billions) and perplexity median from three trials. As Transformer-XL's memory size increased, the perplexity dropped as expected, but the the number of FLOPs grew quickly because the attention length was also increased. The perplexity stopped decreasing after we increased the memory size to $1,600$. We suspect that since the average number of tokens in WikiText-103 is $3,600$, a larger memory size would bring noises and hence did not further improve the performance compared to a smaller memory size $(1,024)$. Compressive Transformer achieves slightly better performance with extra FLOPS compared to Transformer XL with memory size 1024.

Memformer with 4 encoders, 16 decoders, and $1,024$ memory size achieved the best performance. It required much less computation cost (54) and performed much better than Transformer-XL with $1,024$ memory size, supporting that Memformer has a more efficient memory representation.

In the ablation studies, to compensate for the extra number of encoder layers, we reduced the number of decoder layers to 12. The final performance was close to Transformer-XL, but Memformer used a much smaller number of FLOPs. Also, memory size was important for Memformer, as the performance dropped after the memory size is reduced to 512. When we completely removed the memory module by removing the memory writer and memory reading cross attention, the perplexity increased to 25.57, which is similar to Transformer-XL with a memory size of 128.

4.3.1 Memory Writer Analysis

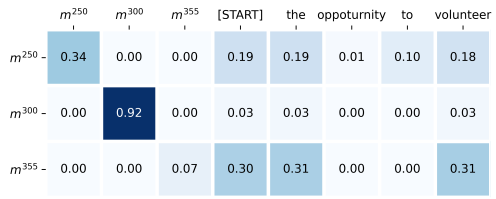


Figure 6: Visualization of three types of memory slots.

It is interesting to interpret how memory writer updates the memory slots. We analyzed the attention outputs from the memory writer. We roughly categorized the memory slots into three different types and visualized three examples with normalized attention values in Figure 6.

We picked the memory slot m^{250}, m^{300} , and m^{355}. During the middle of processing a document, around 60% to 80% of the memory slots are like m^{300}. Their attention focused on themselves, meaning that they were not updating for the current timestep. This suggests that the memory slots can carry information from the distant past.

For the second type, the memory slot m^{250} had some partial attention over itself and the rest of attention over other tokens. This type of memory slots is transformed from the first type of memory slots, and at the current timestep they aggregate information from other tokens.

The third type of memory slot looks like m^{355}. It completely attended to the input tokens. At the beginning, nearly all memory slots belong to this type, but later only 5% to 10% of the total memory slots account for this type. We also found that the forgetting vector's bias for m^{355} had a larger magnitude (3.20) compared to some other slots (1.15), suggesting that the information was changing rapidly for this memory slot.

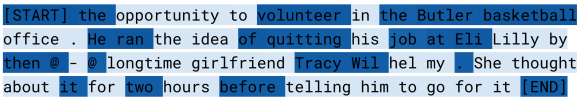


Figure 7: Visualization of the memory writer's attention.

To better understand how the slot m^{355} update its information, we visualized its attention on an example input sequence in Figure 7. It shows that this slot learned a compressed representation of the sentence by attending over some named entities and verbs, which is consistent with human cognition.

5 Conclusion

We presented Memformer, an autoregressive model which utilizes an external dynamic memory to efficiently process long sequences with a linear time complexity and constant memory complexity. Along with Memformer, we introduced a new optimization scheme, Memory Replay Backpropagation, which enables training recurrent neural networks with large memory. Experimental results showed that Memformer achieved comparable performance with great efficiency, and was able to preserve information from the distant past.

With the enhanced memory capacity, we believe that Memformer can spark interesting works that rely on recurrence and autoregressive modeling, which will benefit tasks such as dialog and interactive systems.

References

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *CoRR*, abs/2004.05150.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *CoRR*, abs/1604.06174.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. Generating long sequences with sparse transformers. *URL https://openai.com/blog/sparse-transformers*.

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamás Sarlós, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. 2020. Rethinking attention with performers. *CoRR*, abs/2009.14794.

Junyoung Chung, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc Viet Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Conference of the Association for Computational Linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, Volume 1: Long Papers*, pages 2978–2988. Association for Computational Linguistics.

Felix A. Gers, Jürgen Schmidhuber, and Fred A. Cummins. 2000. Learning to forget: Continual prediction with LSTM. *Neural Comput.*, 12(10):2451–2471.

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *CoRR*, abs/1410.5401.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John P. Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. 2016. Hybrid computing using a neural network with dynamic external memory. *Nat.*, 538(7626):471–476.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5156–5165. PMLR.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2017. Pointer sentinel mixture models. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. 2020. Compressive transformers for long-range sequence modelling. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net.

Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. 2021. Zero-shot text-to-image generation.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1988. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA.

Jos van der Westhuizen and Joan Lasenby. 2018. The unreasonable effectiveness of the forget gate. *CoRR*, abs/1804.04849.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 5998–6008.

Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. Linformer: Self-attention with linear complexity. *CoRR*, abs/2006.04768.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in Neural Information Processing Systems*, 33.

A MRBP Efficiency Test

In this section, we test MRBP's efficiency by comparing against the standard back-propagation through time (BPTT) and the standard gradient checkpointing (GC) algorithm. This algorithm is useful for Memformer to reduce memory requirement because of the back-propagation through several timesteps. We use the Memformer model and set all the hyper-parameters to be the same.

Method	GPU Memory (MB)	Speed (relative)
BPTT	16,177	x1.00
GC	9,885	x0.48
MRBP	7,229	x0.90

Table 3: Memory Replay Back-Propagation performance comparison. Evaluation speed is based on seconds per sample. BPTT means back-propagation through time. GC means gradient checkpointing.

The back-propagation through time (BPTT) approach is the fastest because it does not need re-computation. However, it costs the most amount of memory due to unrolling the entire computational graph. While gradient checkpointing can save huge amount of memory, it is much slower than the other two methods (x0.48). In contrast, our MRBP saves more GPU memory with only slight speed degeneration (x0.90).

B Training Details

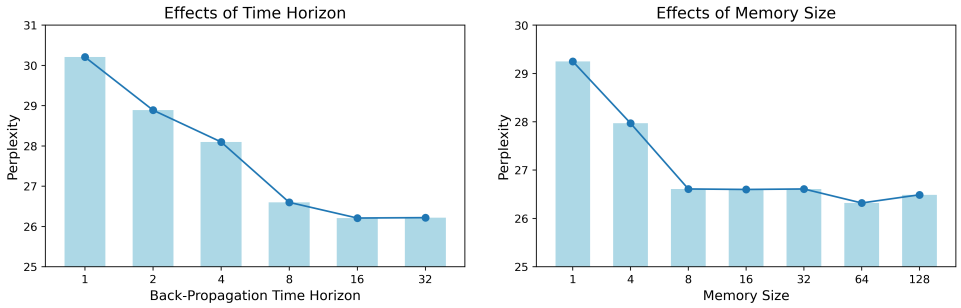
	Image Generation	Language Modeling
batch size	256	128
warm-up steps	1,000	1,0000
learning rate	1e-3	1e-3
dropout	0.1	0.1
memory length	8	1,024
temperature	0.25	0.125
time horizon	8	8
weight decay	0.01	0.01
max gradient norm	1.0	1.0
training steps	10,000	150,000

Table 4: Training Details

We trained our model on NVIDIA V100 16GB and 2080Ti 11GB. The training for image generation took about one day on one GPU. The training for language modeling took approximately four days on four GPUs.

C Effects of Time Horizon and Memory Size

We test how the time horizon for back-propagation affects the performance. We test on a smaller Memformer model for the efficiency. The results are shown in Figure 8a. We vary the back-propagation time horizon from 1 to 32. When the time horizon is set to 1, back-propagation cannot pass gradients through memory to the previous timestep. Thus, we observe the performance is the worst when the time horizon is 1. As we increase the time horizon, the model achieves better perplexity scores. When the time horizon is increased to 32, we observe the marginal improvement on perplexity is almost gone. A large memory size ideally helps to store more information. From Table 8b, we can see a huge improvement when increasing the memory size from 1 to 8. Furhter increasing the memory size has a smaller effects on the performance, and we suspect that this is due to the size of the model.



(a) Effects of different time horizons

(b) Effects of different memory sizes

Figure 8: Effects of different configurations. (a) shows the effects of changing time horizon. (b) shows the effects of changing memory size.

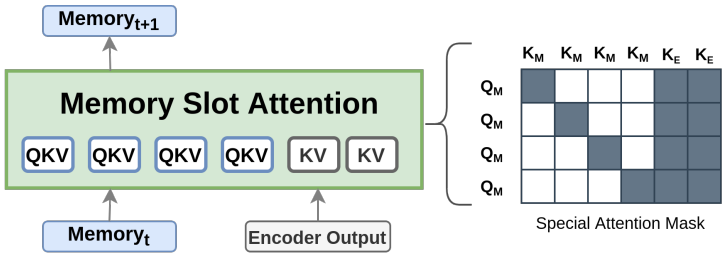


Figure 9: Memory Writer's Attention

D Implementation of Memory Writer

Memory Slot Attention in Figure 9 produces the next timestep's memory M_{t+1}. This module takes the inputs of the previous timestep's memory M_t and the encoder's final hidden states. It then projects the memory into queries, keys, and values, while the encoder outputs are into keys and values. Since each memory slot should not be interfering with other memory slots, we design a special type of sparse attention pattern. Thus, each slot in the memory can only attend over itself and the encoder outputs. This is to preserve the information in each slot longer over the time horizon. For example, if one slot only attends itself, then the information in that slot will not change in the next timestep.