LM2: Large Memory Models

태그	구현코드 작성중
논문	https://arxiv.org/abs/2502.06049
짧은 소개	보조 메모리 모듈 추가하여, 광범위한 문서에서 필수 정보를 효과적으로 추출.

전체 코드

tigerak/lm2: We replicated the model described in the LM2 paper.

소개

아주 긴 컨텍스트에서 추론할 때는, 건초더미에서 바늘을 찾는 것처럼, 문서에 넓게 흩어진 필수 정보를 정확히 뽑아내는 능력이 중요하다.

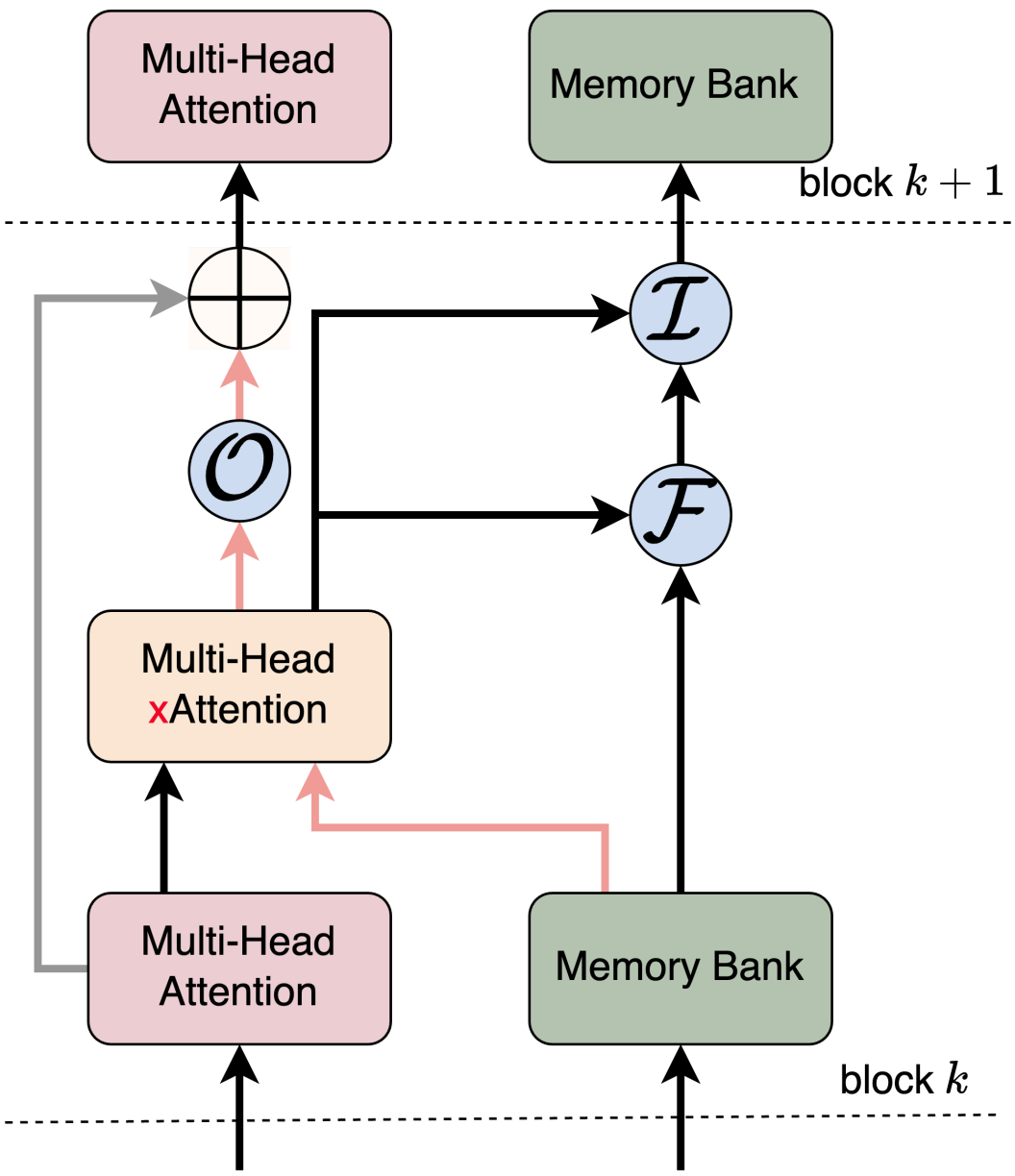
기존의 모델은 이전 문맥을 요약한 프롬프트를 만들거나,
하나의 토큰에 이전 정보를 저장(
https://arxiv.org/abs/2207.06881)하는 등의 방법을 이용하였으나,
광대한 문맥을 처리함에 있어서 누락되는 데이터가 존재할 수 밖에 없었다.

LM2는 아래 그림에서 볼 수 있듯 LLaMa 3.2의 각 어텐션 블록에 메모리 모듈을 추가하여 장기 기억력을 더욱 증가시킨 모델이다.

디코더 모델에 LSTM의 게이트 구조를 적용하였기 때문에 각 어텐션 블록은 Forget Gate, Input Gate, Output Gate를 갖고 있다.

▼ LSTM 참고

https://dgkim5360.tistory.com/entry/understanding-long-short-term-memory-lstm-kr

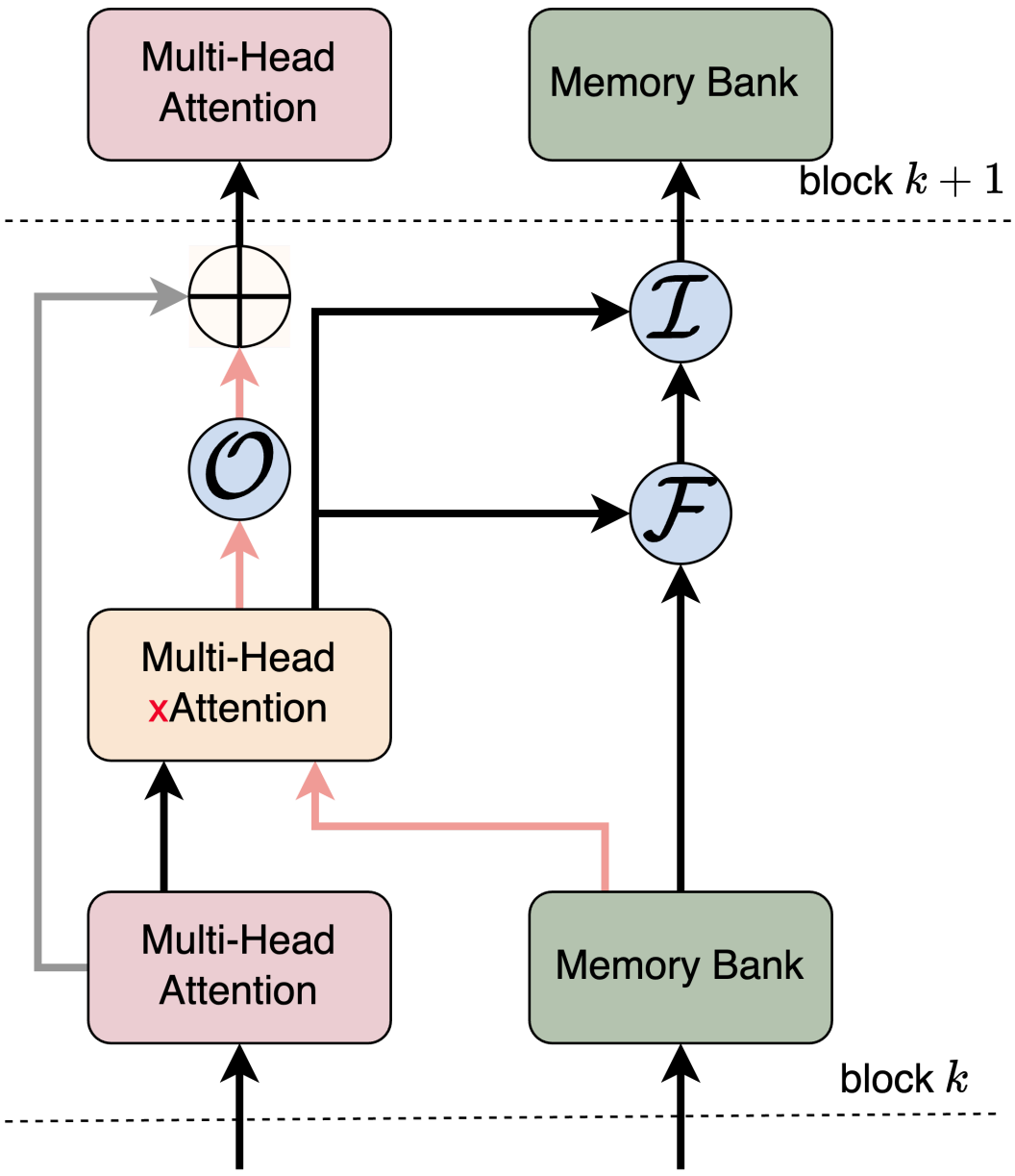



```
class LlamaDecoderLayerWithMemory(LlamaDecoderLayer):
    def __init__(self,
                 config,
                 layer_idx: int,
                 memory_module: LM2MemoryModule):
```

```
    super().__init__(config=config,
                     layer_idx=layer_idx)
    self.memory_module = memory_module

def forword(self,
            hidden_states: torch.Tensor,
            position_embeddings: Optional[Tuple[torch.Tensor, torch.Tensor]]=None, # (cos, sin) for RoPE
            output_attentions: Optional[bool]=False,
            memory_states: Optional[torch.Tensor]=None,
            **kwargs):

    # Llama self-attention
    residual = hidden_states # (B, S, d)
    hidden_states  = self.input_layernorm(hidden_states)
    attn_output, attn_weights = self.self_attn(
                                hidden_states=hidden_states,
                                position_embeddings=position_embeddings,
                                attention_mask=attention_mask,
                                past_key_value=past_key_value,
                                cache_position=cache_position,
                                output_attentions=output_attentions, #
                                use_cache=use_cache, #
                                **kwargs
                                ) # output: attn_output, attn_weights
hidden_states = residual + attn_output

# LM2 memory module 추가
residual2 = hidden_states
hidden_states = self.post_attention_layernorm(hidden_states)
E_out, updated_mem = self.memory_module(hidden_states, memory_states) # memory_module ⇒ (E_out, updated_m
hidden_states = residual2 + E_out

# Llama MLP
residual3 = hidden_states
hidden_states = self.post_attention_layernorm(hidden_states)
hidden_states = self.mlp(hidden_states)
hidden_states = residual3 + hidden_states

outputs = (hidden_states,)
if output_attentions:
    outputs += (attn_weights,)
outputs += (updated_mem,)

return outputs
```

모델 구조

1. 메모리 정보 흐름



1.1. 모델 모듈 초기화

1.1.1. 메모리 뱅크 초기화

위 그림에서 메모리 뱅크의 크기는 $\mathbf{M} \in \mathbb{R}^{N \times d \times d}$의 3차원 형식이다.
여기서 N은 메모리 슬롯 수를 나타내고, d는 각 슬롯의 차원을 나타내며 model의 hidden_dim과 같다.

$\mathbf{M}_r = \mathbf{I}_{d \times d}$, where $r \in \{1, ..., N\}$ ← 단위 행렬로 초기화.
그리고
$\mathbf{M}$은 시퀀스가 입력되면서 계속해서 학습 가능해야하므로 requires_grad=True로 설정.

```
class LM2MemoryModule(nn.Module):
    """
    논문 2.1, 2.2절 구조.
    - 메모리 모듈 shape: (N, d, d)
      각 슬롯 M_r = I_{dxd}로 초기화
    - time_step마다 cross_attention & gate_update
    """
    def __init__(self,
                 d_model: int,
                 num_slots: int):
        super().__init__()
        self.d_model = d_model
        self.num_slots = num_slots

        # Memory 파라미터(슬롯), 논문에서 "M_r = I_{d×d}"로 초기화.
        mem_init = torch.eye(d_model).unsqueeze(0).repeat(num_slots,1,1) # eye(d_model) = (d, d)
        self.memory = nn.Parameter(mem_init, requires_grad=True) # 학습 가능하게 등록
```

1.1.2. Q, K, V 가중치 파라미터 초기화

이제 $\mathbf{E}_t$와 $\mathbf{M}_t$가 cross_attn으로 들어가야 하는데 ⇒ $Q = E_t W^Q, K = M_t W^K, V = M_t W^V$ (eq. 1)

$\mathbf{E} = \mathbb{R}^{T \times d}$이고 $\mathbf{M} \in \mathbb{R}^{N \times d \times d}$이기 때문에 이를 맞춰주어야 한다.
(이 논문에서 T는 시퀀스 길이를 뜻하고, t는 디코더 블록의 t번째를 뜻한다. (코드에서는 T→Seq_len, t→t 로 사용하였다.))
따라서
$\mathbf{M}_r$을 d*d로 flat한 후 → d크기로 선형 변환해주어야 한다.
d*d를 d로 줄여서 사용하는 이유는 효율적인 메모리 사용 때문이다. (그대로 사용하기에는 너무 큼.)
이를 위해 key와 value의 weight를 만들어준다.
여기서
$W^Q, W^K, W^V = \mathbb{R}^{d \times d}$ 는 학습 가능한 투영 행렬이다

```
        # 크로스 어텐션 Q, K, V 투영 행렬 <- eq.(1)
    self.W_Q = nn.Linear(d_model, d_model)
    # 메모리 (d,d)를 flatten하여 (d*d) -> d 로 매핑
    self.W_K = nn.Linear(d_model*d_model, d_model)
    self.W_V = nn.Linear(d_model*d_model, d_model)
```

1.1.3. 게이트 파라미터도 초기화

```
        # 게이트 파라미터터 <- eq.(2)
        self.W_out = nn.Linear(d_model, d_model)
        # 게이트 파라미터터 <- eq.(4), (5)
        #     - (d→d) 만 사용 => 게이트가 (B, d)
        #       => broadcast ⇒ (B, N, d, d)
        self.W_forget = nn.Linear(d_model, d_model)
        self.W_in = nn.Linear(d_model, d_model)
```

1.2. 메모리 모듈에 데이터 입력

self_attn을 통과하면, 각 토큰이 다른 각 토큰들과 어떤 연관성을 갖는 지에 대한 정보가 나온다.
여기에 skip_connection을 하면 그림의
$\mathbf{E}_t$가 된다.

$\mathbf{E}_t$와 $\mathbf{M}_t$을 받아온다.

```
    def forward(self,
                E_t: torch.Tensor,
                M_t: torch.Tensor=None):
        """
        E_t: (Batch_size, Seq_len, d_model) ⇒ time-step loop로 Seq_len개 토큰 각각 처리
        M_t: (Batch_size, Num_slots, d_model, d_model) ⇒ 이전 memory 상태
        return:
          E_out: (Batch_size, Seq_len, d_model)
          M_out: (Batch_size, Num_slots, d_model, d_model)
        """
        batch_size, seq_len, d_model = E_t.shape
        B, S, d = batch_size, seq_len, d_model
        N = self.num_slots

            # 출력 버퍼 설정
        if M_t is None:
            # batch마다 독립된 텐서로 복제 (B, N, d, d)
            M_out = self.memory.unsqueeze(0) \
                .expand(batch_size, N, d_model, d_model) \
                .clone()
        else:
            M_out = M_t.clone()

        E_out = torch.zeros_like(E_t) # (B, S, d)
```

1.3. Cross Attention

1.3.1. Query

시퀀스 순서대로 각 토큰을 Qurey로 넣어준다.
(설명의 편의를 위해 for문으로 구성했다.)

```
# time_step loop (각 토큰마다 메모리 업데이트)
for t in range(seq_len):  # t만 있다면 각 개별 토큰이라는 뜻.
    ### Cross Attention ###
    # 현재 토큰 (batch) => Q
    e_t = E_t[:, t, :] # (B, d)
    Q = self.W_Q(e_t) # (B, d)
```

1.3.2. Key & Value

$\mathbf{M} \in \mathbb{R}^{N \times d}$라고 했으므로 $\mathbf{M}_r = \mathbb{R}^{d \times d}$를 2.에서 이야기 한 것 처럼 d*d로 flatten 시켜준 후
d로 선형 변환하여 Key와 Value로 넣어준다.

```
    # flatten(M_out) ⇒ K, V
    M_flat = M_out.view(batch_size, N, d_model*d_model) # (B, N, d*d)

    K_ = self.W_K(M_flat) # (B, N, d)
    V_ = self.W_V(M_flat) # (B, N, d)
```

1.3.3. 차원 맞춤

메모리 뱅크는 N개의 슬롯이 있기 때문에 (N, d)이고 Qurey로 들어오는 토큰은 (d)이므로,
cross_atten을 위해 Qurey의 차원을 늘려준다.

```
    # Q ⇒ (B, d) ⇒ reshape ⇒ (B,1,d) for batch matmul
    Q_3d = Q.unsqueeze(1) # (B, 1, d)
```

1.3.4. Attention Score

attention score $\mathbf{A} = \frac{Q \times K^\top}{\sqrt{d_{model}}}$
cross_attn을 거치면 Qurey로 들어온 토큰이, Key로 들어온 메모리 뱅크의 각 슬롯들과 어떤 연관이 있는지에 대한 정보(어떤 메모리 슬롯에 집중하는 지에 대한) attn_probs가 나온다.

```
    # attn_score ⇒ (B, 1, N)
    attn_score = (torch.bmm(Q_3d, K_.transpose(2, 1))
                  / (d_model**0.5))
    attn_probs = F.softmax(attn_score, dim=-1) # (B, 1, N)
    # NaN 체크
    if torch.isnan(attn_probs).any():
        print("경고: NaN detected in attention probabilities!")
```

1.4. $\mathbf{E}_{mem}$

Value는 메모리 뱅크의 값이므로 attn_probs에 Value를 가중합하면(1,N)⊙(N,d),
최종적으로 "메모리의 각 슬롯이 현재 Qurey 토큰과 얼마나 관련이 있는 지에 대한 정보"와 "각 슬롯의 임베딩 정보"가 d차원에 압축된다.
(for문으로 토큰을 하나씩 뽑아 계산 중이기에, qurey 차원을 추가했던 이유로, 1번째 차원을 squeeze하여 출력한다.)
논문에서는 이 값을
$\mathbf{E}_{mem}$이라고 한다.

```
    # E_mem_3d = attn_probs * V_ : (B, 1, d) ← resultant attention output
    E_mem_3d = torch.bmm(attn_probs, V_) # 배치 행렬 곱
    E_mem = E_mem_3d.squeeze(1)
```

▼ 개인 생각

- 논문에는 "To ensure temporal consistency, causal masking is applied, and optionally, top-k attention is used to retain only the most relevant memory interactions.” 라고 나와있다.

  self_attn에서 RoPE와 causal mask를 적용하여 현재 토큰을 예측하는데 어떤 토큰이 중요한 지를 이미 계산했다.
  따라서 self-attn의 output인
  $\mathbf{E}_t$에는(현재 시점의 토큰 임베딩에는) 과거 토큰의 중요도가 반영된 상태이다.
  그러므로 self-attn에서 중요도가 높은 토큰은 cross-attn에서 메모리를 업데이트 할 때, 더 의미있는 반영을 유발한다.

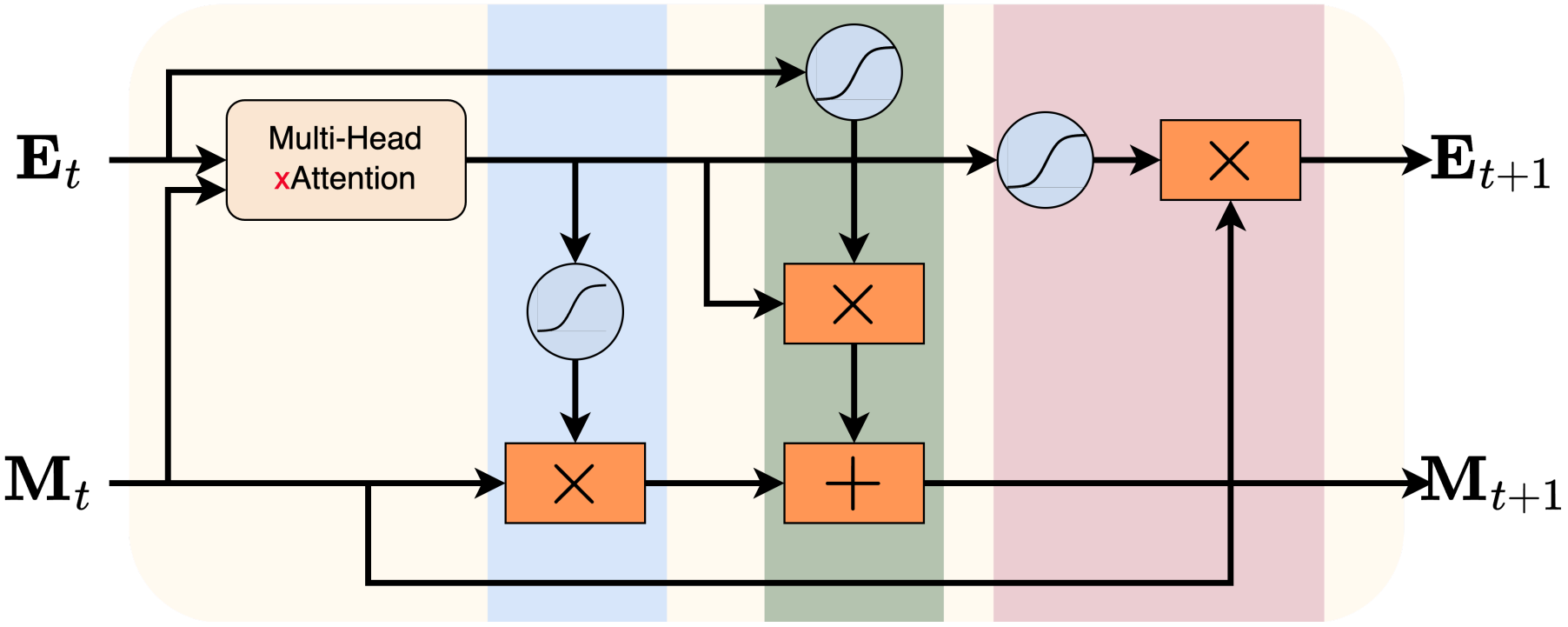
  이는 memory module에 어떤 형태의 정보를 담을 지의 문제로 보인다.
  따라서 Cross Attention에서는, self_attn과 달리, Causal Masking을 반드시 적용해야 한다고는 생각하지 않는다.
  왜냐하면 전체 메모리 슬롯을 DB처럼 활용해서 필요한 정보를 찾는 방식으로 학습되면 되기 때문이다.

  두 차이는 실험을 통해 알아봐야겠다.

- top-k개의 attention-prob만 반영하여 계산하는 옵션도 코드에 구현하지 않았다. 계산의 효율성을 위한 옵션이기 때문이다.
- RoPE는 self-attn에서 적용하였으므로 굳이 같은 블록의 Cross Attention에서 다시 적용할 필요는 없다.

1.5. Output Gate

아래 그림의 빨간 영역은 output gate이다.



1.5.1. g_out (sigmoid)

cross_attn의 결과인 $\mathbf{E}_{mem}$(현재 토큰이 각 메모리 슬롯에 대해 갖는 정보)을 선형 변환하여 sigmoid에 넣으면

$g_{out} = \sigma(E_{mem} W_{out})$ ⇒ 메모리 정보의 각 요소가 얼마나 반영될지를 조절한다.
- Sigmoid :각 원소가 다른 원소와 상관없이 독립적으로 확률 값을 가진다.
- SoftMax : 모든 원소의 출력값의 합이 1이 되도록 한다.)

다시 말해, 메모리 뱅크에서 검색된 정보가 출력
$\mathbf{E}_{t+1}$에 얼마나 영향을 미칠 지에 대한 정보로 바뀐다.
이 때,
$W_{out} \in \mathbb{R}^{d \times d}$이며 학습 가능한 파라미터이다.

```
### Output Gate ###
g_out = torch.sigmoid(self.W_out(E_mem))
```

1.5.2. $\mathbf{E}_{t+1}$

$\mathbf{E}_{gated} = g_{out} \cdot M_t$ (eq.3) 에 따르면 e_mem_gated = g_out * M_out 이어야한다.
하지만 M_out은 (B, N, d, d)로 g_out (B, d)와 차원이 맞지 않고 ← 맞춰주면 되긴 함.
E_mem 또한 메모리 슬롯에 대한 정보가 있으므로 코드에서는 e_mem_gated = g_out * E_mem로 계산하였다.

(단, E_mem만 이용할 경우, 메모리의 일부 정보가 계속 무시되어, 버려지는 슬롯이 생길 수도 있다.)

~~코드에서는 1.3.4. 의 attn_probs 값을 반영하기 위해 하이터 파라미터 λ를 이용해 E_mem과 M_t를 적절히 더한 값을 이용했다. M_t만 이용할 경우 메모리 전체를 조절하기 때문에, 실제 검색된 정보가 아닌 불필요한 슬롯까지 조정될 가능성이 있다.~~

~~따라서 attn_probs 값을 갖고 있는 E_mem을 적당한 비율로 함께 사용하여 불필요한 슬롯은 최대한 적게 업데이트 되도록 작성했다.~~

그 후, e_t를 더해서 skip connection해주었다.

```
e_mem_gated = g_out * E_mem
e_t_new = e_t + e_mem_gated  # skip connection
```

2. 메모리 업데이트

얼마나 많은 새로운 정보가 도입되고 얼마나 많은 오래된 정보가 버려지는 지를 gating함으로써
메모리 모듈은 중요한 장기 사실을 덮어쓰는 것을 방지하는 동시에 긴 컨텍스트 시퀀스를 처리할 때 관련이 없거나 오래된 콘텐츠를 제거하게 된다.

2.1. 망각

위 그림의 파란 영역은 Forget Gate이다.
메모리 모듈에이 기존에 저장하고 있던 데이터 중, 관련이 없거나 오래된 데이터를 제거하기 위한 게이트다.
현재 Query 토큰에 대한 각 메모리 슬롯의 관련도와 슬롯 임베딩 정보를 담고 있는 E_mem을 선형 변환하여 특징을 추출하고 sigmoid를 거쳐,
메모리에서 중요도가 떨어지는 부분들을 찾아낸다.

```
### Memory Update ###
#   g_forget   = sigma(E_mem * W_forget)⇒ (B, d)

g_forget_vec = torch.sigmoid(self.W_forget(E_mem))  # (B, d)
```

2.2. 입력

위 그림의 초록 영역은 Input Gate이다.
e_t는 현재 토큰이 다른 토큰들과 어떤 연관이 있는지에 대한 정보가 반영된 단어 정보를 갖고 있다.
W_in을 통해 e_t를 선형 변환하여 특징을 추출하고, sigmoid를 거쳐

$g_{in} = \sigma(\mathbf{E}_t \mathbf{W}_{in})$ (eq. 4)
현재 Query 토큰에 대해 계산한 정보를 메모리 모듈에 얼마나 반영시킬지 정한다.

```
#   g_in       = sigma(e_t * W_in)      ⇒ (B, d)
#   new_info   = tanh(E_mem)              ⇒ (B, d)

g_in_vec = torch.sigmoid(self.W_in(e_t))     # (B, d)
```

2.3. 메모리 업데이트

2.3.1. E_mem에 tanh를 적용하여 어떤 메모리 슬롯에 대한 정보를 가져갈지 정할 수 있다

```
new_info_vec = torch.tanh(E_mem)            # (B, d)
```

2.3.2. M_out의 차원에 맞게 모두 4차원으로 바꿔준다.

```
# reshape & expand
#   → reshape ⇒ (B,1,d,1) ⇒ expand ⇒ (B,N,d,d)
g_in_4d = g_in_vec.view(B, 1, d, 1).expand(B, N, d, d)
g_forget_4d = g_forget_vec.view(B, 1, d, 1).expand(B, N, d, d)
new_info_4d = new_info_vec.view(B, 1, d, 1).expand(B, N, d, d)
```

2.3.3. 최종 업데이트

입력 : g_in_4d * new_info_4d
g_in으로 현재 토큰에 대한 정보를 메모리에 얼마나 반영할지 정하고,
new_info로 현재 토큰이 각 슬롯에 대해 갖는 정보와 이를 얼마나 반영할 지를 계산하여
이 둘을 곱해 입력값으로 사용한다.

망각 : g_forget_4d * M_out
g_forget으로 메모리 슬롯의 중요도가 떨어지는 부분을 찾아내고,
 M_out을 곱하여 메모리에 불필요한 정보를 제거한다.

```
        # 최종 업데이트
        #   -> M_{t+1} = g_in * new_info + g_forget * M_t
        M_out = g_in_4d * new_info_4d + g_forget_4d * M_out

        E_out[:, t, :] = e_t_new

    return E_out, M_out
```

사전학습

1. 베이스 모델

   Llama-3 모델을 베이스로 하였다.

   16개의 디코더 블록을 쌓았으며, hidden_dim은 2024이다.

   각 블록의 feed-forword dim은 8192이다.

   32개의 attention_heads를 갖고있으며 8개의 key_value_heads를 설정했다.

2. 메모리 모듈

   메모리 모듈은 16개 모든 블록에 통합하였다. 모든 블록에 통합했을 때 가장 좋은 성능을 보였기 때문이다.

3. 크기

   베이스 모델의 파라미터는 1.2B 였으며, 메모리 모듈 통합으로 0.5B가 추가되어, 총 1.7B 크기이다.

4. 데이터

   사전 교육을 위해 SmolLM-Corpus에서 가져온 고품질 데이터 세트 중 합성 테스트북 및 스토리, 교육용 웹 콘텐츠 만 사용하였다.

실험

실험은 다음 다섯 가지 질문에 대해 답하기 위해 설계되었다.

질문 1: LM2는 메모리 작업을 어떻게 수행합니까?

질문 2: LM2가 일반 작업의 성능에 해를 끼칩니까?

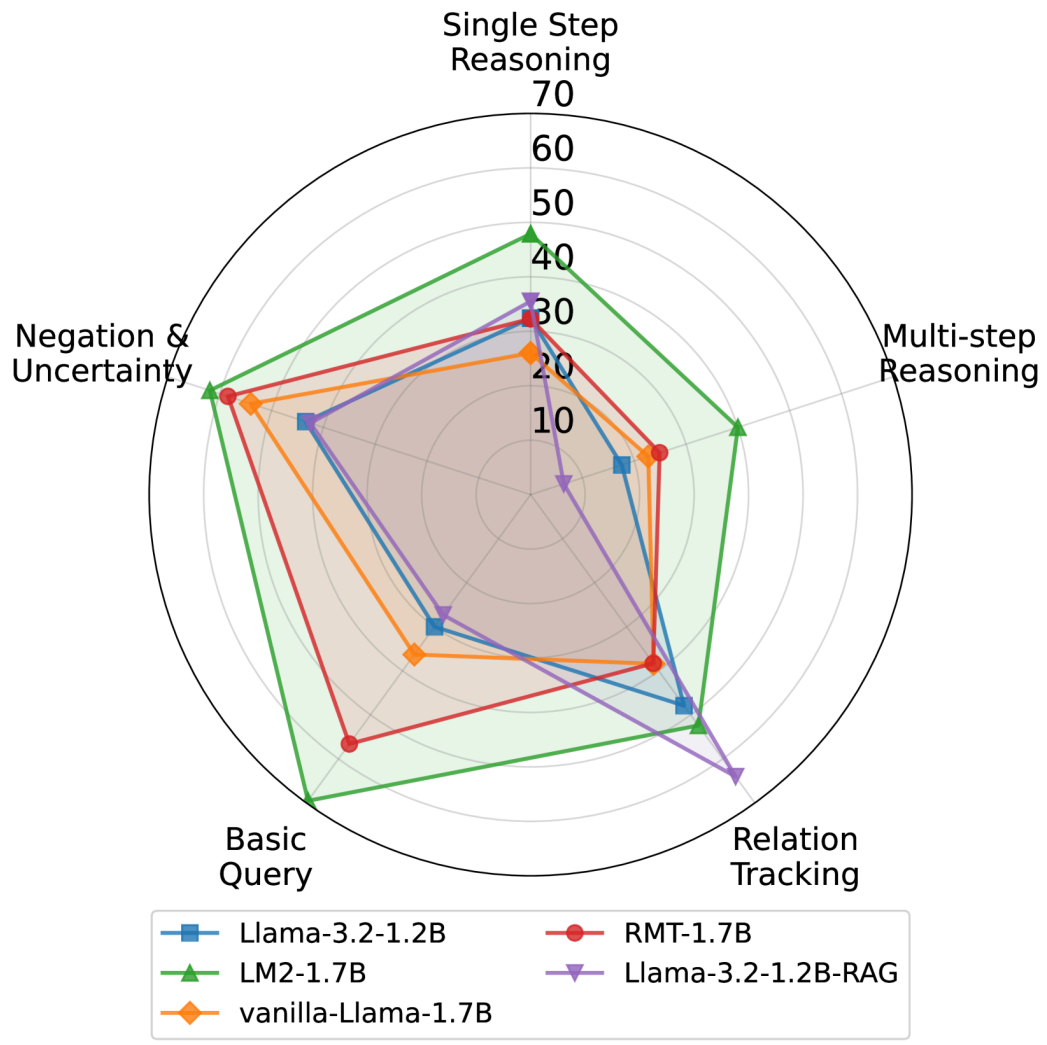
질문 3: 모든 디코더 블록에 메모리 모듈을 포함해야 합니까?

질문 4: 메모리 뱅크에는 무엇이 저장됩니까?

질문 5: 테스트 시 메모리 모듈은 어떻게 업데이트됩니까?

1. 메모리 작업에 대한 성능

BABILong 데이터셋에서 입력 텍스트의 크기를 0k ~ 128k까지 키워가며 다양한 질문에 대한 답변을 평가함.



2. 일반 벤치마크에 대한 성능

제안된 메모리 메커니즘이 일반적인 적용 가능성을 방해하지 않음을 나타낸다.

		vanilla Llama	RMT	LM2
Subject Category	**STEM**	27.2	25.7	**28.1**
	Humanities	28.7	26.7	**32.2**
	Social Sciences	29.2	27.0	**31.6**
	Others	27.7	27.1	**28.0**
Difficulty Level	**High School**	28.8	26.5	**30.4**
	College	27.7	27.1	**29.0**
	Professional	27.5	26.6	**27.6**
	General Knowledge	27.2	25.6	**28.5**
Average		28.0	26.5	**29.4**

3. 메모리 모듈의 영향

Few-Shot Examples

Example 1:

Question: What is the capital of France?

Options: A) Berlin, B) Madrid, C) Paris, D) Rome

Answer:

- First, I know that the capital of France is a well-known fact.
- France is a country in Western Europe, and its capital city is Paris.

Example 2:

Question: Which of the following is required for the process of photosynthesis to occur?

Options: A) Oxygen and glucose B) Sunlight, water, and carbon dioxide

C) Carbon monoxide and nitrogen D) Chlorophyll and methane

Answer:

- Photosynthesis is a process that plants use to convert light energy into chemical energy.
- It requires sunlight as the energy source, water as a reactant, and carbon dioxide from the air.

Example X ...

Target Question:

Question: Which of the following statements is true about the process of photosynthesis?

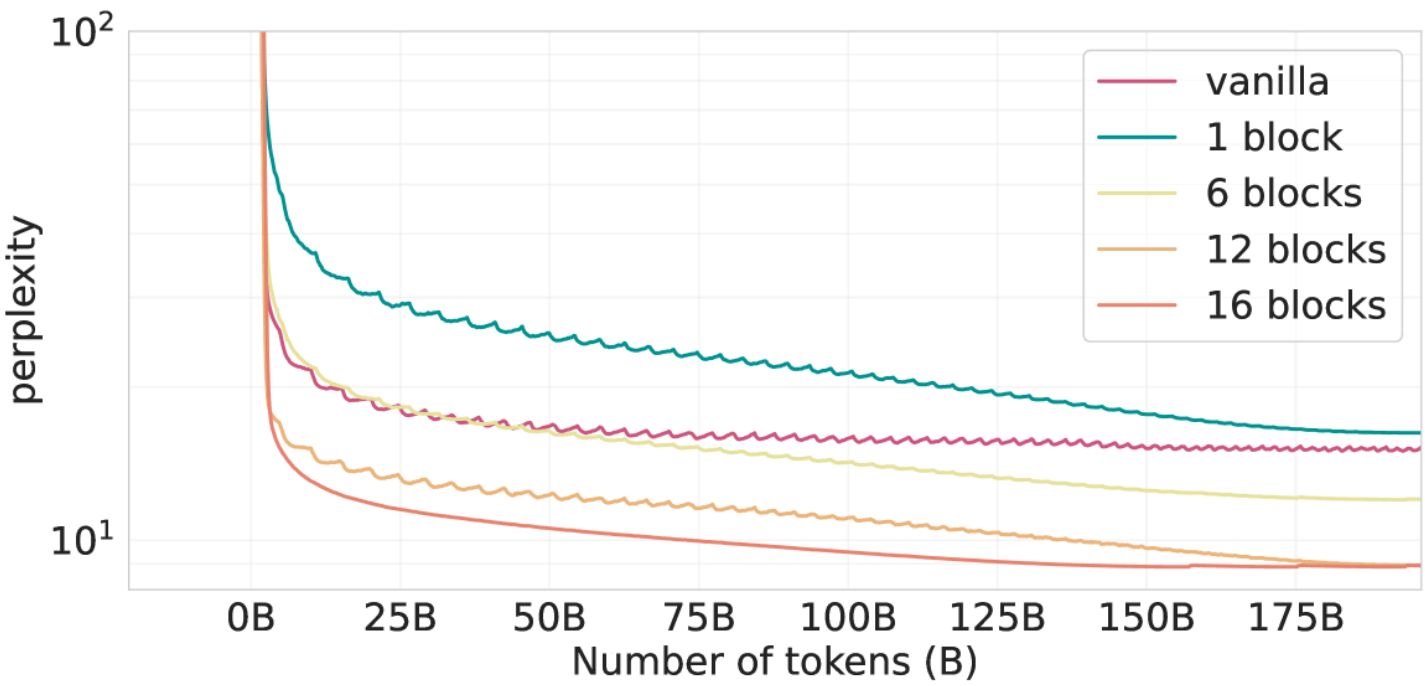
Options: A) It produces oxygen as a byproduct. B) It occurs in animal cells.

C) It uses carbon monoxide as a reactant. D) It does not require sunlight.

LM2 Answer:

- Photosynthesis is a process that plants use to convert light energy into chemical energy.
- It produces oxygen as a byproduct.

위와 같은 Few-shot 질문을 하였을 때



16개의 모든 블록에 메모리 모듈을 추가 한 경우 가장 성능이 좋았다.

4. 메모리 표현 분석

https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html 을 이용해 위의 fow-shot 질문을 하였다.

그런 다음 가장 관련성이 높은 메모리 슬롯 두 개(슬롯 1679 및 1684)와 가장 관련 없는 슬롯 한 개(슬롯 1)를 뽑아 위 방법으로 분석하였다.

메모리 슬롯 1679 : 대상 질문에 대한 사실 정보를 검색하고 합성하는 데 특화

메모리 슬롯 1684 : 입력 텍스트 내의 구조적 요소("Options:" 또는 "Answer:” 등)에 초점.

메모리 슬롯 1 : 입력 텍스트 전반에 걸쳐 주로 부정적인 활성화를 보였는데, 이는 작업별 콘텐츠에 대한 최소한의 참여를 나타냄.

5. 테스트 시간 메모리 조정

위의 fow-shot 질문을 하였을 때,

메모리 업데이트 전에는 각 메모리가 질문의 구조에 초점을 맞추고 있는 반면

추론 업데이트 이후에는 각 메모리가 질문과 관계된 토큰으로 이동하고 있다.