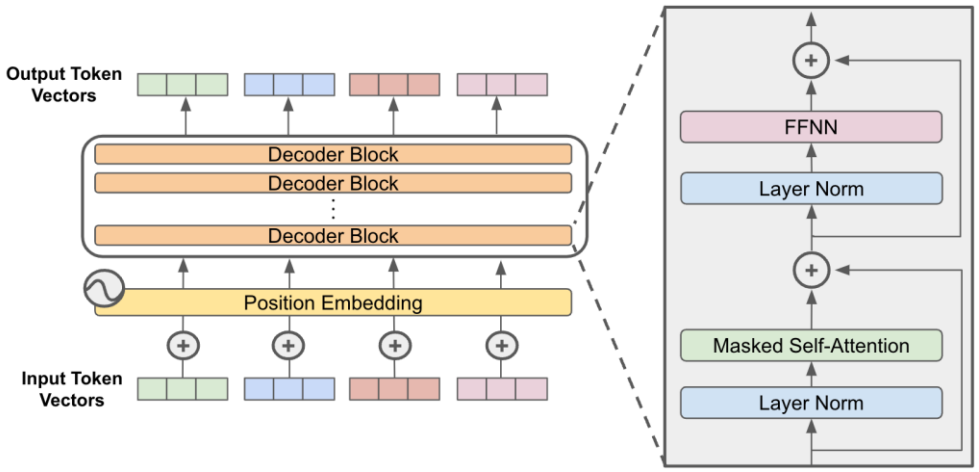
You Only Cache Once: Decoder-Decoder Architectures for Language Models

2025. 01. 02

임제원

Decoder Only Transformer VS YOCO



<KV Cache 처리 과정>

입력 문장: "I am a student"

Decoder Only Transformer는 토큰별로 처리

첫 번째 토큰: "I"
KV Cache = Key: ["I"], Value : ["I"]

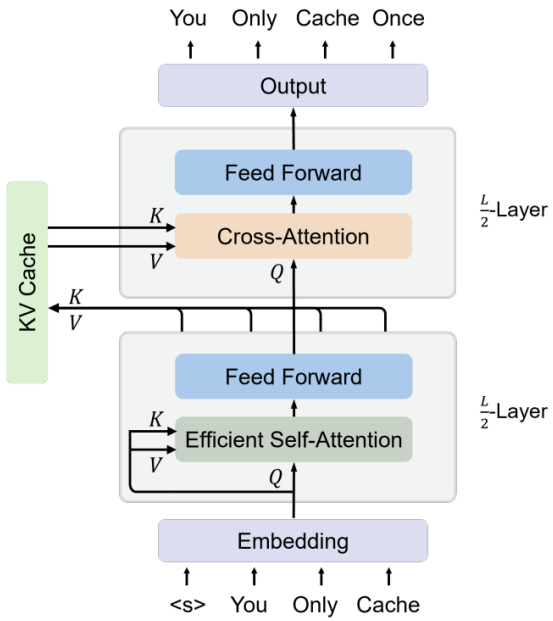
두 번째 토큰: "am"
KV Cache = Key: ["I", "am"], Value : ["I", "am"]

세 번째 토큰: "a"
KV Cache = Key: ["I", "am", "a"], Value : ["I", "am", "a"]

각 디코더 층마다 서로 다른 KV Cache 보유

L: 입력 시퀀스 길이 (Sequence Length)
N: 레이어 수 (Number of Layers)
D: 은닉 차원 (Hidden Dimension)

<KV Cache Memory>
O(LND)
= 4 x 6 x 512
= 12,288



<KV Cache 처리 과정>
YOCO는 KV 생성을 병렬 처리

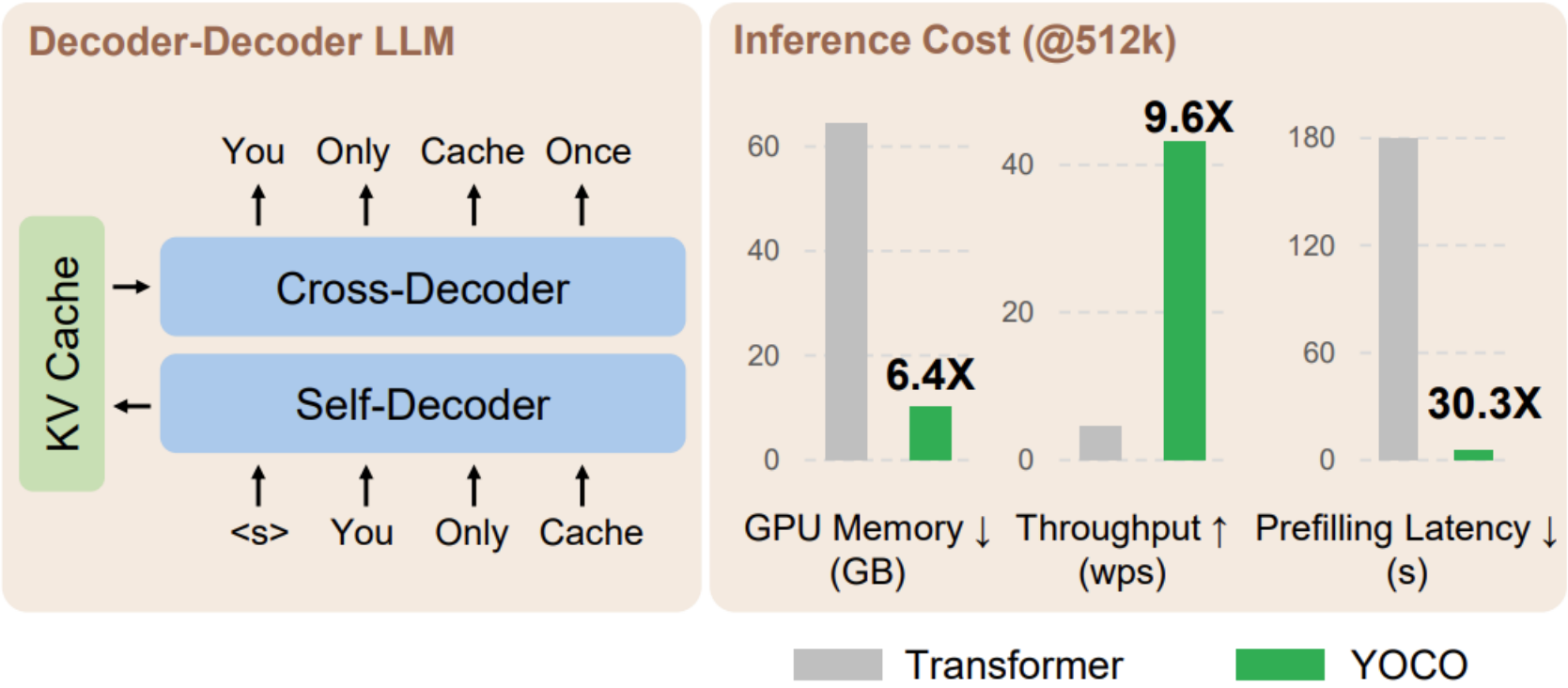
<Self Decoder>
입력 문장: "I am a student"
Q,K,V = 임베딩된 토큰들
이후 Self Attention을 통해 Q,K,V 생성

KV Cache = Key: ["I", "am", "a", " student"],
Value : ["I", "am", "a" , " student"]

한번에 만들어진 KV는 KV Cache에 저장
이후 모든 층의 디코더에서 KV Cache를 공유하며 재사용

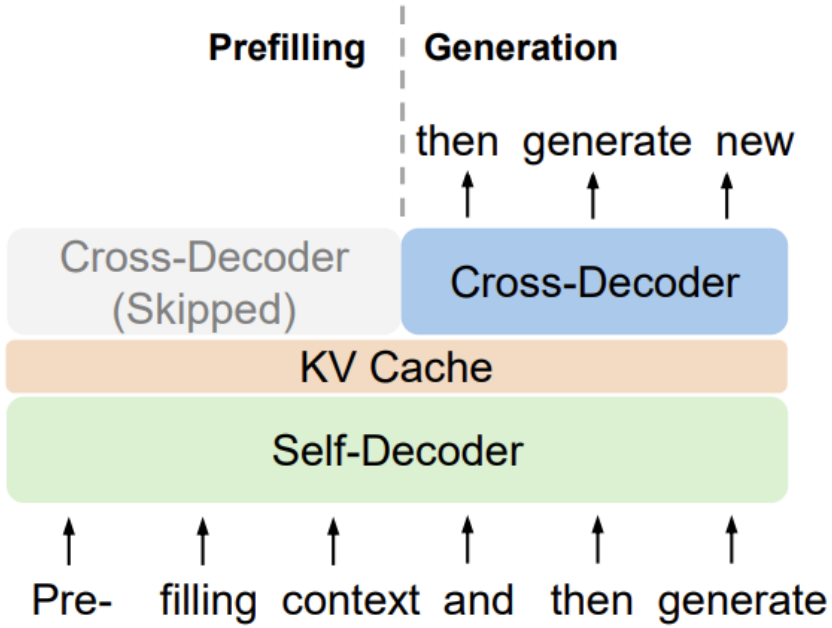
<KV Cache Memory>
O((L+N)D)
= (4 + 6) x 512
= 5120

Ⅱ Decoder-Decoder Structure in YOCO



- 번역 모델이 아닌 생성 모델
- Encoder-Decoder 대신 Self Decoder–Cross Decoder 구조 사용
- Encoder-Decoder Structure(Transformer) 대비 GPU Memory 감소, Throughput(처리량) 증가, Prefilling Latency 감소
- Prefilling Latency: 언어 모델이 입력 토큰을 처리하고 초기 디코딩 상태를 설정하는 데 걸리는 시간을 의미
  - 주로 입력 토큰을 처리하여 Decoder가 첫 번째 출력을 생성하기까지의 지연시간
  - 이 단계에서 모델이 모든 입력을 인코딩하고, 필요한 Attention 및 연산을 완료해야 하므로 상당한 계산 자원을 필요로 함

Decoder-Decoder Structure in YOCO



Cross Decoder→

Self Decoder→

	KV Cache Memory
Transformer	$\mathcal{O}(LND)$
YOCO	$\mathcal{O}((N+L)D)$

Table 1: Inference memory complexity of KV caches. N, L, D are the sequence length, number of layers, and hidden dimension.

	Prefilling Time
Transformer	$\mathcal{O}(LN^2D)$
YOCO	$\mathcal{O}(LND)$

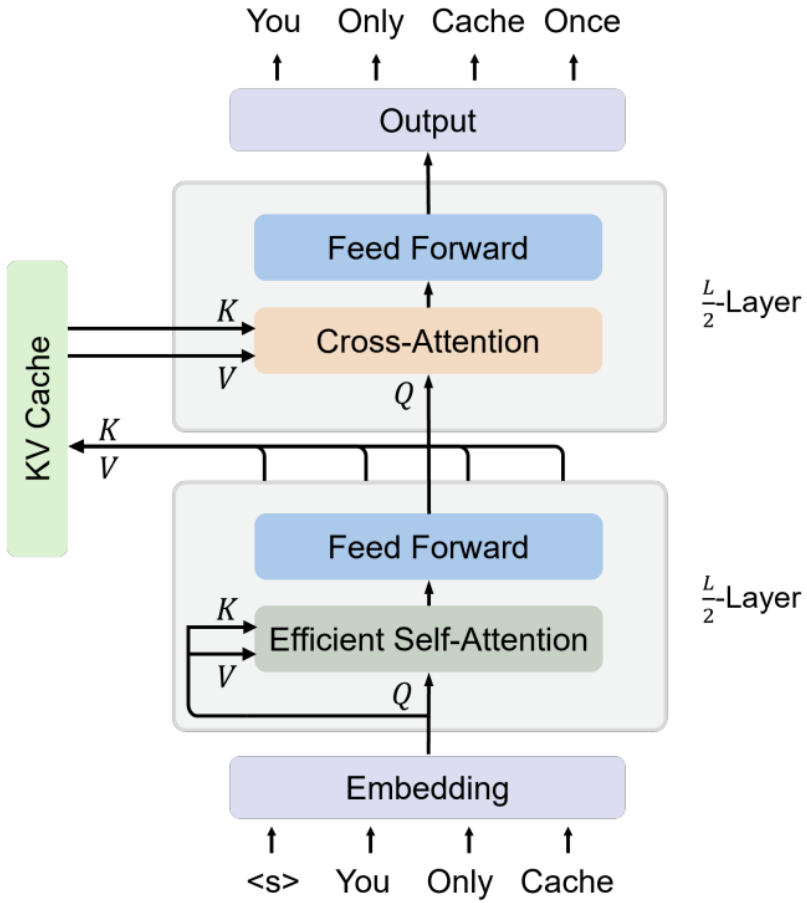
Table 2: Prefilling time complexity of attention modules. N, L, D are the same as above.

- Prefilling
  정의: 입력 토큰을 병렬로 처리하여 KV Cache를 생성하는 단계
  목적: 모델이 문장의 첫 번째 출력 토큰을 예측하기 위해 필요한 Context 설정

L: 입력 시퀀스 길이 (Sequence Length)
N: 레이어 수 (Number of Layers)
D: 은닉 차원 (Hidden Dimension)

- Generation
  정의: Decoder가 하나씩 출력 토큰을 생성하는 단계
  과정: KV Cache를 재사용하여 토큰을 순차적으로 예측
  YOCO는 Cross-Decoder에서 KV Cache를 한 번만 계산하여 계산량을 줄이고 효율성 극대화

Decoder-Decoder Structure in YOCO



<Self-Decoder>

- Self Decoder는 Global KV Cache를 생성
- Cross Decoder는 Cross Attention 과정에서 공유 KV 캐시를 재사용
- Self Decoder, Cross Decoder 모두 Causal Masking 사용

<Self Decoder>

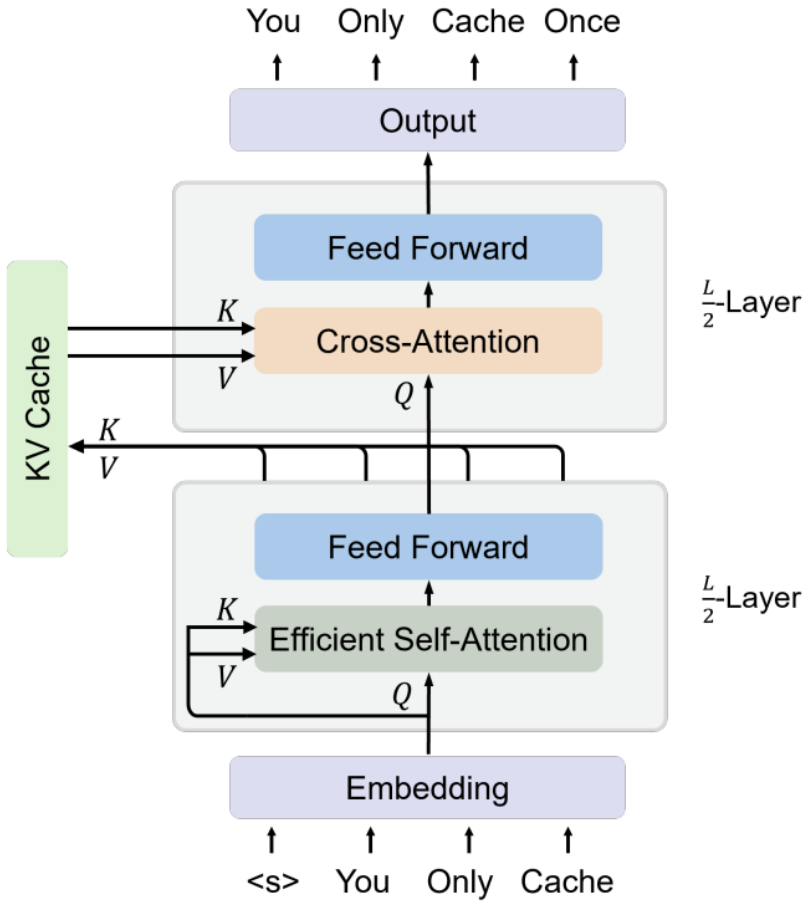
$$Y^l = ESA\left(LN\left(X^l\right)\right) + X^l$$

$$X^{l+1} = SwiGLU\left(LN\left(Y^l\right)\right) + Y^l$$

Y^l : 레이어 l의 출력 벡터
$LN(X^l)$: 입력 X^l에 Layer Normalization(LN) 적용
ESA : Efficient Self Attention, 효율적인 Self-Attention 메커니즘을 의미
Sliding-Window Attention 또는 Gated Retention 사용 가능
$SwiGLU(X)$: Swish-Gated Linear Unit, 활성화 함수의 종류

→ 정규화 된 X^l을 입력으로 받아 ESA를 수행하여 출력됨

Decoder-Decoder Structure in YOCO



<Self-Decoder>

- Self-Decoder의 출력은 KV Cache를 생성

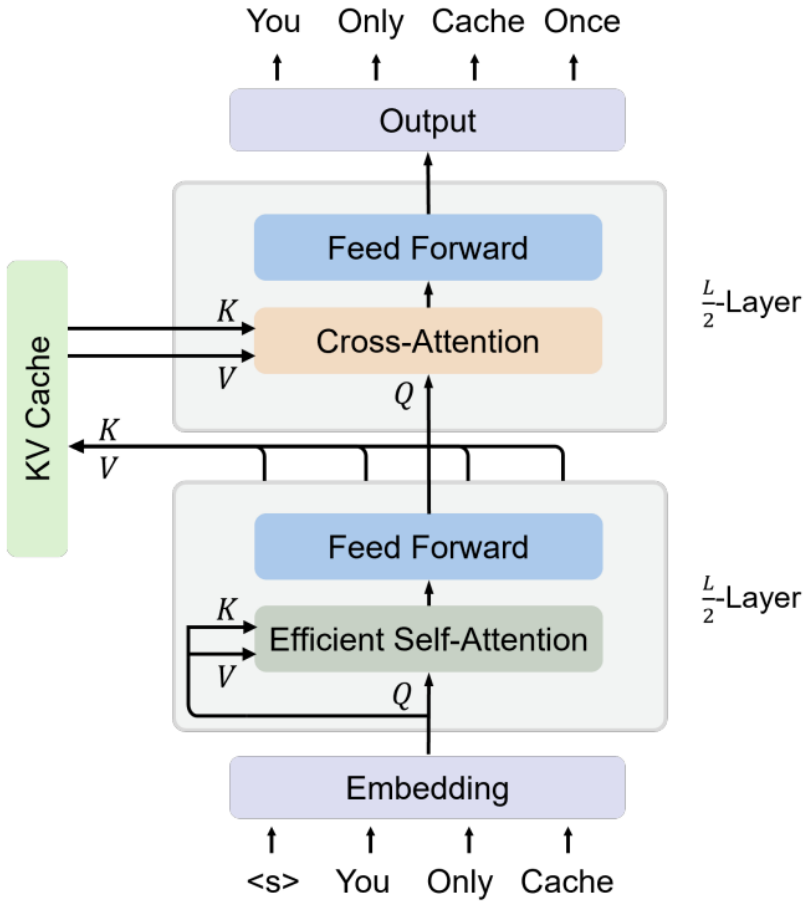
<KV Cache Generation>

$$\hat{K} = LN\left(X^{L/2}\right)W_k, \qquad \hat{V} = LN\left(X^{L/2}\right)W_v$$

$X^{L/2}$: Self-Decoder에서 나온 토큰의 중간 표현
$LN\left(X^{L/2}\right)$: 입력 X^l에 Layer Normalization(LN) 적용
W_k, W_v: Key, value의 가중치

→ Self-Decoder에서 나온 출력 값을 정규화 하여 KV Cache로 사용
→ 이후 모든 Cross-Decoder 레이어에서 재사용

Decoder-Decoder Structure in YOCO



<Cross-Decoder>

<Cross-Attention>

$$\widehat{Q^l} = LN(X^l)W_Q^l$$

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$$X^{l+1} = SwiGLU\left(LN(Y^l)\right) + Y^l$$

X^l : 이전 Cross-Decoder 레이어의 출력 값
W_Q^l : Query 가중치 행렬
$LN(X^l)$: 정규화

Decoder-Decoder Structure in YOCO

<Design Choices of Self-Decoder>

<Gated Retention>

Gated Retention(gRet)은 데이터 의존적 게이팅 메커니즘을 사용하여
병렬성, 성능, 낮은 추론 비용을 동시에 달성

1. The Parallel Representation(병렬 표현)

$$Q = (XW_Q) \odot \Theta, \quad K = (XW_K) \odot \overline{\Theta}, \quad V = XW_V, \quad \Theta_n = e^{in\theta}$$

$$\gamma = \text{sigmoid}(XW_\gamma)^{1/\tau}, \quad D_{nm} = \begin{cases} \prod_{i=m+1}^{n} \gamma_i, & n \geq m \\ 0, & n < m \end{cases}$$

$$\text{gRet}(X) = (QK^\intercal \odot D)V$$

2. The Recurrent Representation(재귀 표현)

$$S_n = \gamma_n S_{n-1} + K_n^\intercal V_n$$
$$\text{gRet}(X_n) = Q_n S_n, \quad n = 1, \cdots, |x|$$

3. The Chunkwise Recurrent Representation(청크 단위 재귀 표현)

chunk, i.e., $x_{[i]} = x_{(i-1)B+1}, \cdots, x_{iB}$, we compute the i-th chunk as:

$$\beta_{(i-1)B+j} = \prod_{k=(i-1)B+1}^{(i-1)B+j} \gamma_k, \quad D_{[i]}(j,k) = \frac{\beta_{(i-1)B+k}}{\beta_{(i-1)B+j}} \text{ if } j \leq k \text{ else } 0$$

$$R_i = K_{[i]}^\intercal (V_{[i]} \odot \frac{\beta_{iB}}{\beta_{[i]}}) + \beta_{iB} R_{i-1}, \quad \beta_{[i]}(j,k) = \beta_{(i-1)B+j}$$

$$\text{gRet}(X) = \underbrace{(Q_{[i]} K_{[i]}^\intercal \odot D_{[i]}) V_{[i]}}_{\text{Inner-Chunk}} + \underbrace{(Q_{[i]} R_{i-1}) \odot \beta_{[i]}}_{\text{Cross-Chunk}}$$

4. Multi-Head Gated Retention

$$\text{head}_i = \text{gRet}(X)$$
$$Y = \text{GroupNorm}_h(\text{Concat}(\text{head}_1, \cdots, \text{head}_n))$$
$$\text{MHGR}(X) = (\text{swish}(XW_G) \odot Y)W_O$$

Decoder-Decoder Structure in YOCO

<Design Choices of Self-Decoder>

<Sliding-Window Attention>

Attention 연산을 고정된 윈도우 크기 C로 제한하여 계산 효율성을 높이는 방식

→ 메모리 사용량을 O(N)에서 O(C)로 감소

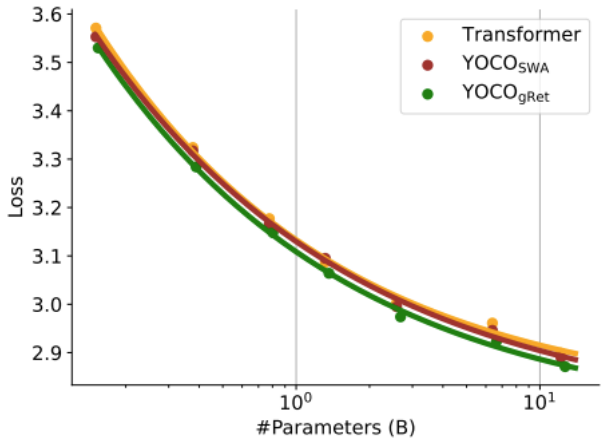
$$
\begin{aligned}
Q = XW_Q, \quad K = XW_K, \quad V = XW_V \\
\text{head}_i = \text{softmax}(Q_{[i]} K_{[i]}^{\intercal} + B)V \\
B_{ij} = \begin{cases} 0, & i - C < j \leq i \\ -\infty, & \text{otherwise} \end{cases} \\
Y = \text{Concat}(\text{head}_1, \cdots, \text{head}_h) \\
\text{SWA}(X) = YW_O
\end{aligned}
$$

Model	ARC-C	ARC-E	BoolQ	Hellaswag	OBQA	PIQA	Winogrande	SciQ	Avg
Training with 1T tokens									
OpenLLaMA-3B-v2	0.339	0.676	**0.657**	**0.700**	0.260	0.767	0.629	**0.924**	0.619
StableLM-base-alpha-3B-v2	0.324	0.673	0.646	0.686	0.264	0.760	0.621	0.921	0.612
StableLM-3B-4E1T	—	0.666	—	—	—	**0.768**	0.632	0.914	—
YOCO-3B	**0.379**	**0.731**	0.645	0.689	**0.298**	0.763	**0.639**	**0.924**	**0.634**
Training with 1.6T tokens									
StableLM-3B-4E1T	—	0.688	—	—	—	0.762	0.627	0.913	—
YOCO-3B	0.396	0.733	0.644	0.698	0.300	0.764	0.631	0.921	0.636
Extending context length to 1M tokens									
YOCO-3B-1M	0.413	0.747	0.638	0.705	0.300	0.773	0.651	0.932	0.645

Table 3: Eval Harness [GTA+23] results compared with previous well-trained Transformer language models [TBMR, Tow, GL23]. We scale the 3B model to 1.6 trillion training tokens. The 1T and 1.6T results of StableLM-3B-4E1T are taken from its technical report [TBMR]. YOCO-3B-1M is extended to the context length of 1M tokens.

- 벤치마크 테스트셋 비교 결과 언어모델에서 경쟁력 있는 성능을 보임

Ⅲ Experiments



- 다양한 매개변수에 따른 유효성 검사 손실 비교

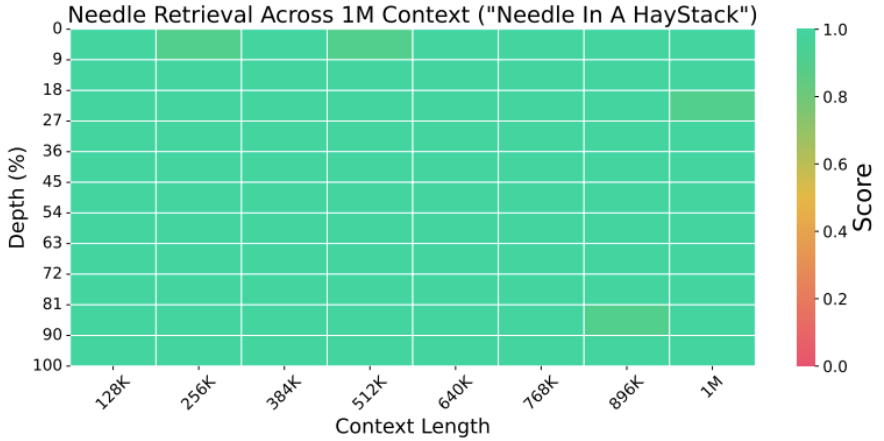


Figure 5: Needle-in-a-haystack results in 1M length.

- 바늘 찾기 테스트, 긴 문장에서도 "바늘" 단어를 잘 찾아내는 성능 확인

Model	Size	$N = 1$	$N = 2$	$N = 4$	$N = 8$
YaRN-Mistral-128K [PQFS23]	7B	0.02	0.12	0.08	0.20
LWM-1M-text [LYZA24]	7B	1.00	0.90	0.76	0.62
MiniCPM-128K [HTH+24]	2.4B	1.00	1.00	0.54	0.56
ChatGLM3-128K [ZLD+22]	6B	0.94	0.72	0.52	0.44
YOCO-3B-1M	3B	0.98	0.98	0.84	0.56

- 다중 바늘 찾기 테스트에서도 강한 성능을 보임
- LWM-1M-text의 절반의 모델 크기로도 비슷한 성능

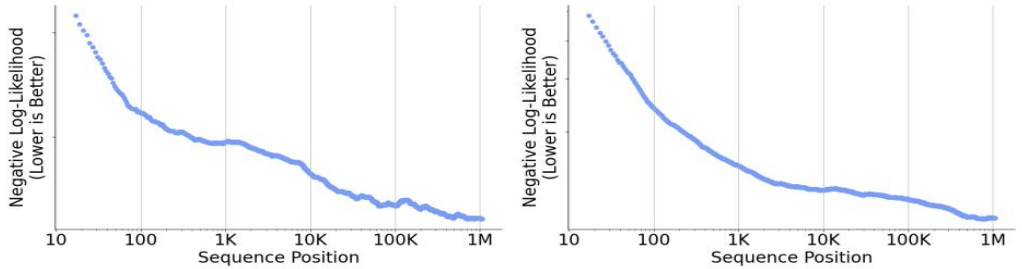
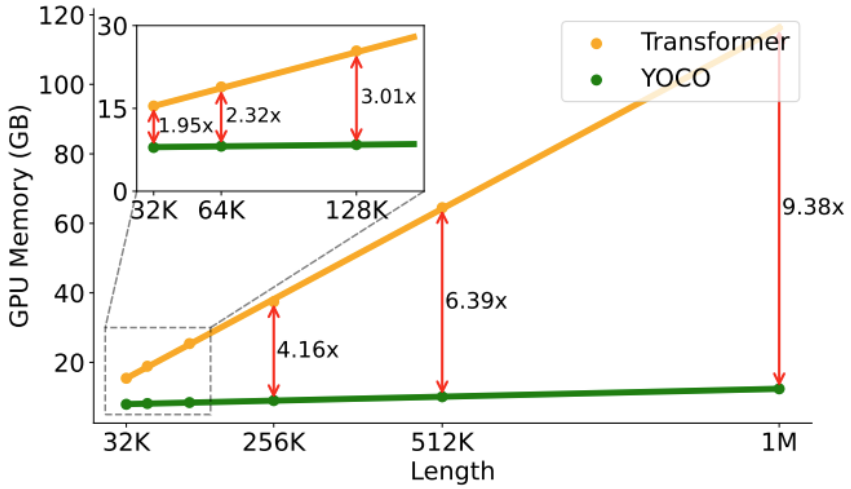


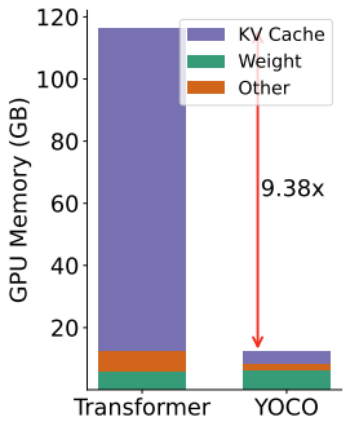
Figure 6: Cumulative average negative log-likelihood on book and repository-level code. We filter the validation examples that are longer than 1M tokens. YOCO achieves improved performance with longer context, i.e., utilizing long-distance information for language modeling.

- NLL이 시퀀스 길이가 길어질 수록 일관되게 감소함
- 장거리 의존성 해결을 보임

Experiments

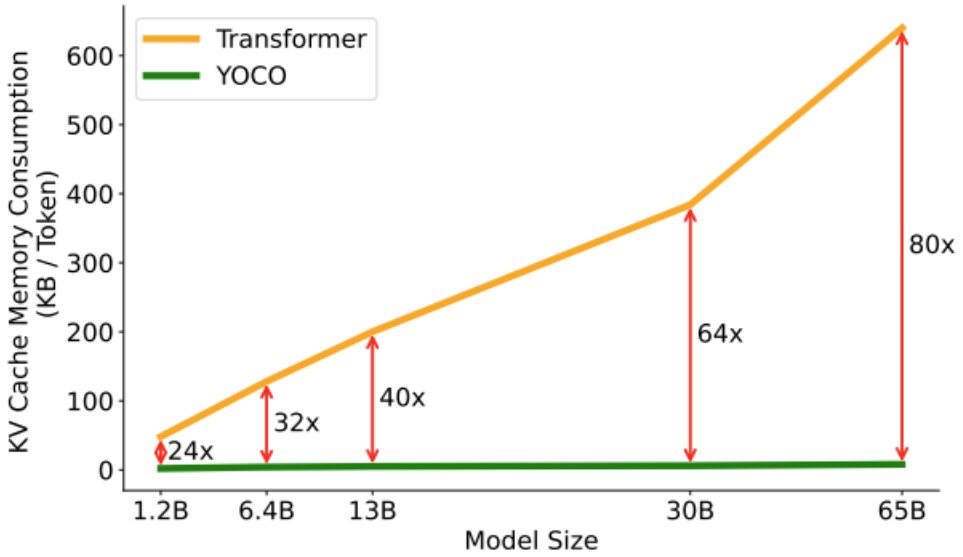


(a) Inference memory of Transformer and YOCO across various lengths.



(b) Breakdown memory consumption in 1M context length.

Figure 7: GPU memory consumption during inference.



- 입력 시퀀스의 길이가 길어져도 트랜스포머에 비해 GPU Memory가 훨씬 감소
- 모델의 크기가 클 수록 더 많이 절약할 수 있음

Experiments

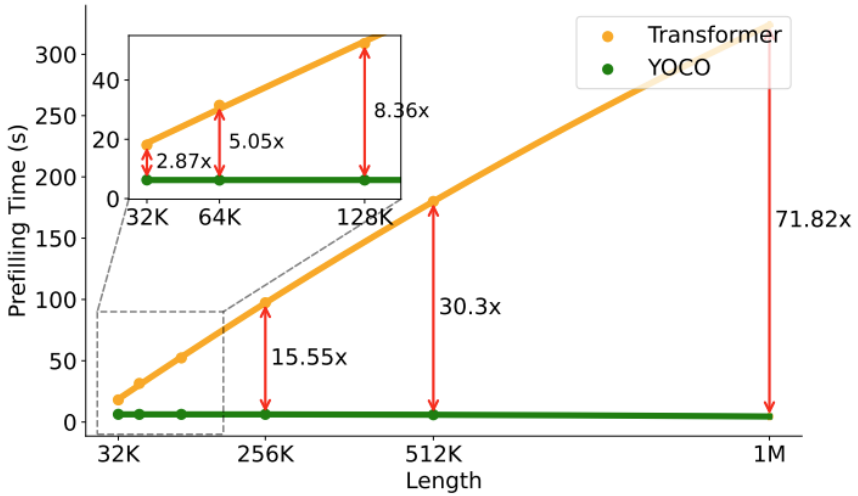


Figure 9: Prefilling latency for different length, i.e., the encoding time of given input prompt before generating the first token. Transformer's time grows quadratically while YOCO's grows linearly. Even for a short input length, such as 32K, YOCO can still accelerate 2.87×.

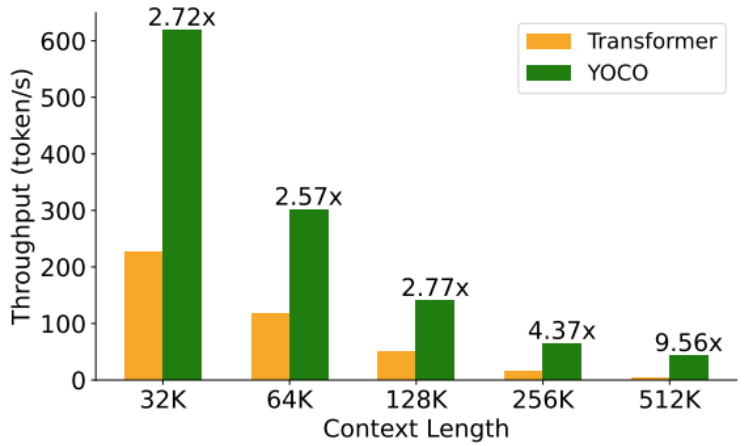


Figure 10: Inference throughput of Transformer and YOCO varying the context length.

- 입력 시퀀스의 길이가 길어져도 Prefilling Time이 Transformer대비 거의 증가하지 않는 모습
- 입력 시퀀스의 길이에 따른 처리량도 Transformer 대비 뛰어난 성능

IV Conclusion

- Decoder-Decoder Architecture YOCO는 트랜스포머에 비해 추론 효율과 경쟁력 있는 성능
- 대규모 언어 모델로 갈 수록 유리한 결과를 달성
- 특히 “긴 시퀀스 모델링 " 에서 추론 효율을 몇 배나 향상시키는 것으로 나타남