Text Generation

KV cache

Autoregression

Autoregression – 생성한 토큰과 그 입력 시퀀스를 결합하여 새로운 입력 시퀀스를 만들어서 재입력

① 'next token'을 생성하려면 그 직전 토큰 시퀀스를 필요로 함(LM 학습원리)

② 처음 시작은 시스템 'prompt'와 사용자 'prompt'를 결합해서 입력하고,

③ 생성된 토큰을 이 전 시퀀스에 결합해서 다시 입력하고,

④ 생성이 끝났다는 토큰이 생성되거나, max-sequence-length 까지 이 과정을 반복함.

```
# Huggingface 예
for _ in range(100):
    next_token = generate_token(inputs)

    # 다음 입력에 샘플링된 토큰 추가
    inputs['input_ids'] = torch.cat([inputs['input_ids'], next_token], dim=-1).to(device)
    next_attention = torch.ones((inputs['attention_mask'].size(0), 1)).to(device)
    inputs['attention_mask'] = torch.cat([inputs['attention_mask'], next_attention], dim=-1).to(device)
```

(참고) Huggingface tokenizer output format

```
input_text = "Tell me a joke about chickens."
inputs = tokenizer(input_text, return_tensors="pt").to(device)
print(inputs)
print(tokenizer.decode(inputs['input_ids’][0]))
```

```
{
   'input_ids’:
          tensor([[24446,   502,   257,  9707,   546, 25972,    13]], device='cuda:0’),
    'attention_mask’:
          tensor([[1, 1, 1, 1, 1, 1, 1]], device='cuda:0’)
}
```

Autoregression – 중복 계산 피하기

Autoregression은 토큰 시퀀스를 반복해서 입력하기 때문에 같은 계산을 반복하는 비효율성 문제가 있음

① 새로이 생성된 토큰만 입력하여 이 토큰의 'next token'을 생성하되,

② 'next token'을 생성하기 위해 필요한 토큰 시퀀스 정보를 저장하고 복원하여 연산하는 방식을 취함

③ 문제는 어떤 정보를 저장해야 하는가? → 어떤 정보가 필요한가?

④ 트랜스포머 모델의 모든 블록의 출력을 저장할 필요 없고,

⑤ 기존 토큰과 새로이 입력된 토큰을 연계하여 연산해야 하는 블록에 필요한 벡터들을 저장하고 연산함

KV Cache 이해하기 ≈ 트랜스포머 모델 이해하기

- 어떤 정보를 저장하는 것이 필요할까? ≈ 무엇이 새로운 정보인가?

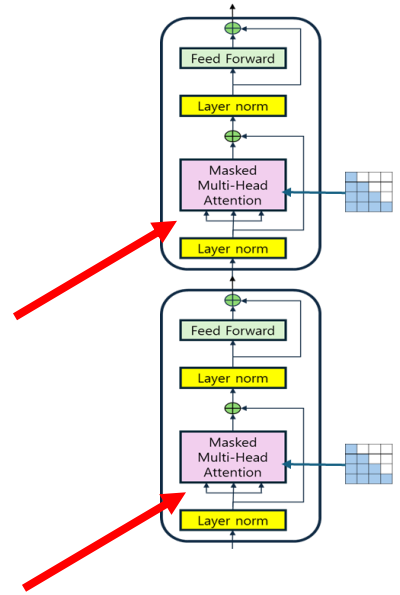
① 지금까지 생성된 토큰과 새로이 생성된 토큰을 입력하기 때문에 새로운 정보는? 새로이 생성된 토큰

② Scaled-dot-product-attention에서 지금까지 생성된 토큰과 새로이 생성된 토큰과의 연산만 필요

③ 사실상 causality가 무의미 해짐 – 마치 RNN에서 순차적으로 토큰을 하나씩 입력하는 것과 같은 이치

④ 상태벡터를 학습하는 RNN과 달리 트랜스포머 모델은 지금까지 생성된 토큰 정보를 저장하는 것이 필요

- 어디서 저장하여야 하는가? – Scaled-Dot-Product-Attention 입력
  - Layer norm, Feed Forward, Skip Connection & Add – (X)
  - Scaled-dot-product-attention의 입력인 Q, K, V에서 새로 지금까지 축적된
  - (K, V) 리스트와 새로이 생성된 토큰의 q, k, v 이면 next-token을 생성하기 충분하다.
  - 따라서, 다음 토큰을 생성하기 위한 입력은?
  - 새로이 생성된 토큰과 지금까지 저장된 (K, V) 리스트로 한다.



Masked Multi-Head Attention(MMHA)의 입력

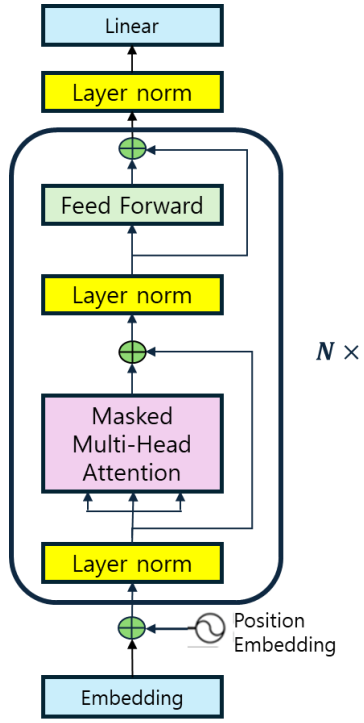
- **Masked Multi-Head Attention의 입력 – MMHA를 위한 K_{past} 와 V_{past} 를 caching함**
- 지금 스텝 T 까지의 K_{past}, V_{past} 저장하고, 새 토큰 1개에 대해서 $Q_{new}, K_{new}, V_{new}$를 계산
- $Q_{new} = [B \times n_{head} \times 1 \times \frac{d_{model}}{n_{head}}]$, $K_{new} = [B \times nh_{head} \times 1 \times \frac{d_{model}}{n_{head}}]$, $V_{new} = [B \times n_{head} \times 1 \times \frac{d_{model}}{n_{head}}]$
- $K_{total} = [K_{past}, K_{new}] = [B \times n_{head} \times (T+1) \times \frac{d_{model}}{n_{head}}]$
- $V_{total} = [V_{past}, V_{new}] = [B \times n_{head} \times (T+1) \times \frac{d_{model}}{n_{head}}]$

① 입력은 새로운 토큰 1개와 K_{past}, V_{past}

② 모델의 다른 블록(Add & Norm, Linear-Layer, Feed Forward)의 입력은 토큰 1개

③ Attention 계산할 때만 K_{past}, V_{past} 이용

④ 위 모든 과정은 트랜스포머 블록마다 적용됨 – 블록이 12개이면 12개의 (K_{past}, V_{past}) 를 저장



Scaled Dot Product Attention with K-V Caching

- Scaled-dot-product-attention의 입력인 K_{total} 와 V_{total} 의 shape는 $\left(B, n_{head}, T+1, \frac{d_{model}}{n_{head}}\right)$ 이고, Q_{new} 의 shape는 $\left(B, n_{head}, 1, \frac{d_{model}}{n_{head}}\right)$ 이다.

$$softmax\left(\frac{Q_{new} \cdot K_{total}^{T}}{\sqrt{d_{model}}}\right) \cdot V_{total} \rightarrow X.shape = \left(B, n_{head}, 1, \frac{d_{model}}{n_{head}}\right)$$

- Scaled Dot Product Attention의 출력인 X는 다음과 같이 재정렬되어 다음 블록으로 입력되고,

① $[B, 1, d_{model}]$

② $K_{total} = [K_{past}, K_{new}]$, $V_{total} = [V_{past}, V_{new}]$을 각각 K_{past}, V_{past}로 저장한다.

- (주의) K_past, V_past는 모델의 모든 트랜스포머 블록마다 저장된다. GPT-2 12-블록의 경우,
- past_kvs = [(k_layer1, v_layer1), (k_layer2, v_layer2), … (k_layer12, v_layer12)]

Text Generation with KV-Cache: Huggingface 예

- inputs의 포맷을 다음과 같이 한다.

```
inputs = {
"input_ids": next_token,
"attention_mask": torch.cat([inputs["attention_mask"], torch.tensor([[1]]).to(device)], dim=1),
"past_key_values": past_key_values,
}
```

  - past_key_values are a tuple of 12 (Ks, Vs) → 12개 layer일 때, len(past_key_vlaues) = 12
  - shape of each K and V is [batch, n_head, seq, n_dim//n_head]
  - K, V in the 3rd layer can be accessed : past_key_values[2][0], past_key_values[2][1]

- 모델이 반환한 outputs = model(inputs)으로부터 logits과 past_key_values를 얻은 후에 next_token을 샘플링하고, inputs을 위 포맷과 같이 업데이트 하여 model(inputs)를 재 호출한다.

① logits = outputs.logits

② past_key_values = outputs.past_key_values

past_key_values를 어떻게 저장하는가? (예) GPT2.py

① Block에서

- forward에서 (x, K_past, V_past)를 받는다.
① x → q, k, v 만들고,
② Q = [q], K = [K_past, k], V = [V_past, v] 만들고,
③ F.scaled_dot_product_attention(Q, K, V)를 호출하여 y를 반환 받는다.
- y, (K, V) 를 반환한다.

② GPT 클래스에서,

- forward에서 (x, past_kvs)를 받는다. past_kvs는 12개 길이의 tuple이고, 각 tuple은 K, V 쌍을 저장한다.
- 블록 개수만큼 루프를 돌면서 다음 과정을 반복한다.
① 입력을 x, K_past, V_past 로 주고 block을 호출한다. (K_past = past_kvs[i][0], V_past = past_kvs[i][1])
② y, (K, V)를 반환 받고,
③ (K, V)를 present_kvs에 append한다.
④ x ← y 하고, ①로 간다.
- 루프가 끝나면, x, present_kvs를 반환한다.

③ GPT클래스는 최종 logits과 present_kvs를 반환한다.

Text Generation Mechanism with KV-Caching

- 트랜스포머 모델 안에 KV 데이터를 저장하는 것이 아니라 프로세싱 하는 루틴을 추가함

→ 실제 저장은 모델 밖에서 이루어짐

① 트랜스포머 모델은 KV cache 데이터와 토큰 입력 받음

② 트랜스포머 모델은 Logits을 생성하고, 이전 KV cache와 생성된 KV를 결합하여 새로운 KV cache 생성

③ 반환된 logits과 KV cache를 이용하여 next-token을 샘플링

④ 트랜스포머 모델에 next-token과 KV cache를 입력