LLM RAG Agent lesson plan

Ver 2.0

Kang T

Table of Contents

1. 개요

본 교재는 회차 주제별 주요 학습 내용을 요약 기술한 것으로, 실제 강의에는 교재 이외 저자 블로그, github, huggingface, 논문 등이 활용된다. 전체 주제는 실라버스 문서에 기술된다.

- 실라버스 - https://github.com/mac999/LLM-RAG-Agent-Tutorial/blob/main/1-1.prepare/syllabus-llm-rag-agent.docx

2. 트랜스포머 구조와 동작 메커니즘

2.1 서론

트랜스포머는 자연어 처리 및 생성 AI 분야에서 필수적으로 사용되는 구조이며, 특히 인코더는 입력 시퀀스를 이해하고 문맥 정보를 포착하는 데 핵심적인 역할을 한다. 본 교안에서는 트랜스포머 작동 원리를 중심으로, 핵심 구성요소 및 PyTorch 기반 구현 코드를 분석하고 설명한다.

2.2 트랜스포머 어텐션 동작 메커니즘

이 장은 트랜스포머에서 가장 중요한 셀프 어텐션(Self-Attention), 크로스 어텐션(Cross-Attention) 메커니즘에 대해 설명하고 있다. 크로스 어텐션은 주로 시퀀스-투-시퀀스(Seq2Seq) 모델의 디코더(Decoder) 부분에서 사용되며, 인코더(Encoder)가 학습한 원본 시퀀스의 컨텍스트 정보와 디코더가 현재 생성 중인 대상 시퀀스의 정보를 효과적으로 결합하는 역할을 한다. 특히, 영어-한국어 번역과 같이 서로 다른 언어 간의 복잡한 관계를 이해하고 번역의 품질을 높이는 데 필수적인 메커니즘이다

이 장은 영어-한국어 번역을 할 때 트랜스포머가 어떤 식으로 동작하는 지를 한단계식 묘사할 것이다.

2.2.1 학습 데이터셋 주요 변수 설명

영어-한국어 번역을 위한 트랜스포머 학습 데이터셋은 모델이 언어 간의 매핑을 학습하는 데 필요한 다양한 구성 요소를 포함하고 있다. 주요 변수와 그 예시는 다음과 같다.

2.2.2 입력 시퀀스 (Source Sequence) - 영어 문장

모델이 학습해야 할 원본 언어(영어) 문장이다. 이 문장들은 토큰화(tokenization) 과정을 거쳐 단어 또는 서브워드(subword) 단위의 시퀀스로 변환된다.

- **변수명 예시:** english_sentences 또는 source_texts
- **데이터 타입:** 문자열 리스트(List of Strings)이다.
- **예시 데이터:**

```
english_sentences = [
    "I love machine learning.",
    "How are you doing today?",
    "The cat is sleeping on the sofa.",
    "Artificial intelligence is transforming the world.",
    "Please translate this sentence."
]
```

2.2.3 대상 시퀀스 (Target Sequence) - 한국어 문장

모델이 번역해야 할 목표 언어(한국어) 문장이다. 이 문장들 역시 토큰화 과정을 거치며, 학습 시 모델의 출력을 비교하고 손실(loss)을 계산하는 데 사용된다.

- **변수명 예시:** korean_sentences 또는 target_texts
- **데이터 타입:** 문자열 리스트(List of Strings)이다.
- **예시 데이터:**

```
korean_sentences = [
    "저는 머신러닝을 사랑합니다.",
    "오늘 어떻게 지내세요?",
    "고양이가 소파 위에서 자고 있습니다.",
    "인공지능이 세상을 변화시키고 있습니다.",
```

"이 문장을 번역해 주세요."

]

2.2.4 토큰화된 입력 시퀀스 (Tokenized Source Sequence)

원본 영어 문장들을 토큰화하여 얻은 정수 ID(integer ID) 시퀀스이다. 각 정수는 어휘집(vocabulary)의 특정 단어 또는 서브워드에 매핑된다.

- **변수명 예시:** tokenized_source_ids 또는 encoder_input_ids
- **데이터 타입:** 정수 리스트의 리스트 (List of Lists of Integers) 또는 텐서(Tensor)이다.
- **예시 데이터 (가상의 토큰 ID):**

```
# 가상의 한국어 어휘집 매핑 예시
# {"<pad>": 0, "<bos>": 1, "<eos>": 2, "저는": 100, "머신러닝을": 101, "사랑합니다": 102, ...}

# 디코더 입력 (Shifted Right)
# 학습 시 디코더는 이전 출력(shifted right)을 입력으로 사용하여 다음 토큰을 예측한다.
decoder_input_ids = [
    [1, 100, 101, 102, 2],      # "<bos> 저는 머신러닝을 사랑합니다." -> [<bos>, 저는, 머신러닝을, 사랑합니다, <eos>]
    [1, 103, 104, 105, 106, 2], # "<bos> 오늘 어떻게 지내세요?" -> [<bos>, 오늘, 어떻게, 지내세요, ?, <eos>]
    # ...
]

# 디코더 레이블 (Expected Output)
# 모델이 예측해야 할 실제 다음 토큰이다.
decoder_label_ids = [
    [100, 101, 102, 2, 0],     # "저는 머신러닝을 사랑합니다.<eos><pad>" (패딩 고려)
    [103, 104, 105, 106, 2, 0], # "오늘 어떻게 지내세요?<eos><pad>"
    # ...
```

]

- <bos> (Beginning-of-Sentence): 문장의 시작을 나타내는 특수 토큰이다. 디코더 입력의 첫 번째 토큰으로 사용되어 번역 생성을 시작한다.
- <eos> (End-of-Sentence): 문장의 끝을 나타내는 특수 토큰이다. 디코더가 이 토큰을 생성하면 번역이 완료된 것으로 간주한다.
- 시프트된 대상 시퀀스 (Shifted Target Sequence): 트랜스포머의 디코더는 자기회귀적(autoregressive)으로 작동한다. 즉, 각 타임스텝에서 이전까지 생성된 토큰들을 바탕으로 다음 토큰을 예측한다. 따라서 학습 시 디코더의 입력은 실제 대상 시퀀스보다 한 칸 오른쪽으로 시프트된 형태(tokenized_target_ids 의 <bos> 포함)가 되고, 디코더의 목표 출력(레이블)은 실제 대상 시퀀스(tokenized_target_ids 의 <bos> 제외하고 <eos> 포함)가 된다.

2.2.5 토큰화된 대상 시퀀스 (Tokenized Target Sequence)

목표 한국어 문장들을 토큰화하여 얻은 정수 ID 시퀀스이다. 이 시퀀스는 디코더의 입력과 출력 레이블로 사용된다.

- **변수명 예시:** tokenized_target_ids 또는 decoder_input_ids (디코더 입력), decoder_label_ids (디코더 출력 레이블)
- **데이터 타입:** 정수 리스트의 리스트 (List of Lists of Integers) 또는 텐서(Tensor)이다.
- **예시 데이터 (가상의 토큰 ID):**

  # 가상의 한국어 어휘집 매핑 예시

  # {"<pad>": 0, "<bos>": 1, "<eos>": 2, "저는": 100, "머신러닝을": 101, "사랑합니다": 102, ...}

  # 디코더 입력 (Shifted Right)

```
# 학습 시 디코더는 이전 출력(shifted right)을 입력으로 사용하여 다음 토큰을 예측한다.
decoder_input_ids = [
    [1, 100, 101, 102, 2],      # "<bos> 저는 머신러닝을 사랑합니다." -> [<bos>, 저는, 머신러닝을, 사랑합니다, <eos>]
    [1, 103, 104, 105, 106, 2], # "<bos> 오늘 어떻게 지내세요?" -> [<bos>, 오늘, 어떻게, 지내세요, ?, <eos>]
    # ...
]

# 디코더 레이블 (Expected Output)
# 모델이 예측해야 할 실제 다음 토큰이다.
decoder_label_ids = [
    [100, 101, 102, 2, 0],     # "저는 머신러닝을 사랑합니다.<eos><pad>" (패딩 고려)
    [103, 104, 105, 106, 2, 0], # "오늘 어떻게 지내세요?<eos><pad>"
    # ...
]
```

- **<bos> (Beginning-of-Sentence)**: 문장의 시작을 나타내는 특수 토큰이다. 디코더 입력의 첫 번째 토큰으로 사용되어 번역 생성을 시작한다.
- **<eos> (End-of-Sentence)**: 문장의 끝을 나타내는 특수 토큰이다. 디코더가 이 토큰을 생성하면 번역이 완료된 것으로 간주한다.
- **시프트된 대상 시퀀스 (Shifted Target Sequence):** 트랜스포머의 디코더는 자기회귀적(autoregressive)으로 작동한다. 즉, 각 타임스텝에서 이전까지 생성된 토큰들을 바탕으로 다음 토큰을 예측한다. 따라서 학습 시 디코더의 입력은 실제

대상 시퀀스보다 한 칸 오른쪽으로 시프트된 형태(tokenized_target_ids 의 <bos> 포함)가 되고, 디코더의 목표 출력(레이블)은 실제 대상 시퀀스(tokenized_target_ids 의 <bos> 제외하고 <eos> 포함)가 된다.

2.2.6 패딩 마스크 (Padding Mask)

시퀀스 길이를 맞춰주기 위해 추가된 패딩 토큰(<pad>)이 어텐션 계산에 영향을 미치지 않도록 마스킹하는 데 사용된다. 패딩 토큰은 실제 내용이 아니므로, 모델이 이 토큰에 어텐션하지 않도록 한다.

- **변수명 예시:** source_padding_mask, target_padding_mask
- **데이터 타입:** 부울(Boolean) 텐서 또는 이진(Binary) 텐서 (0 또는 1)이다.
- **예시 데이터 (가상의 토큰 ID 에 기반):**

```
# 최대 시퀀스 길이 10 으로 패딩되었다고 가정
# `tokenized_source_ids` 예시:
# [3, 4, 5, 6, 2, 0, 0, 0, 0, 0]  (original length 5)

source_padding_mask = [
    [1, 1, 1, 1, 1, 0, 0, 0, 0, 0], # 1: 유효한 토큰, 0: 패딩 토큰
    [1, 1, 1, 1, 1, 1, 0, 0, 0, 0], # "How are you doing today?" (length 6)
    # ...
]

# `decoder_input_ids` 예시:
# [1, 100, 101, 102, 2, 0, 0, 0, 0, 0] (original length 5)

target_padding_mask = [
    [1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
```

[1, 1, 1, 1, 1, 1, 0, 0, 0, 0],

...

]

- **역할:** 어텐션 메커니즘에서 Softmax 를 계산하기 전에 패딩 위치에 해당하는 점수(logits)를 -inf (음의 무한대)로 설정하여 해당 위치의 가중치가 0 이 되도록 한다.

2.2.7 룩-어헤드 마스크 (Look-Ahead Mask) - 디코더 셀프 어텐션용

디코더의 셀프 어텐션 레이어에서 사용된다. 디코더는 자기회귀적으로 작동하므로, 특정 타임스텝에서 다음 토큰을 예측할 때 미래의 토큰들을 "엿보지(look ahead)" 않도록 마스킹해야 한다. 이는 번역 생성이 순차적으로 이루어져야 한다는 원칙을 지키기 위함이다.

- **변수명 예시:** look_ahead_mask
- **데이터 타입:** 부울(Boolean) 텐서 또는 이진(Binary) 텐서 (상삼각 행렬 형태)이다.
- 예시:

  시퀀스 [A, B, C, D]가 있다고 가정했을 때, 룩-어헤드 마스크는 다음과 같이 생성된다.

  [[1, 0, 0, 0], # A 는 A 만 볼 수 있다.

  [1, 1, 0, 0], # B 는 A, B 를 볼 수 있다.

  [1, 1, 1, 0], # C 는 A, B, C 를 볼 수 있다.

  [1, 1, 1, 1]] # D 는 A, B, C, D 를 볼 수 있다.

- **역할:** 디코더의 각 위치가 현재 위치 이후의 토큰에 어텐션하는 것을 방지한다.

2.3 트랜스포머 어텐션 계산학습 과정 (영어 한국어 번역)

크로스 어텐션은 트랜스포머 디코더의 두 번째 어텐션 계층에서 발생한다. 이 과정은 디코더가 인코더로부터 영어 원문의 컨텍스트 정보(즉, 원본 문장의 의미)를 가져와 현재 예측하고 있는 한국어 번역의 문맥과 결합하는 핵심 단계이다.

가정:

- **입력 (Source):** 영어 문장 "I love machine learning."
- **출력 (Target):** 한국어 문장 "저는 머신러닝을 사랑합니다."
- 이 문장들은 이미 토큰화되고 임베딩(embedding)을 거쳐 벡터 표현으로 변환되었다고 가정한다.
- 설명의 편의를 위해 임베딩 차원 d_model 을 4 로, 시퀀스 길이를 짧게 가정한다.

2.3.1 1 단계: 인코더 출력 및 디코더 입력 준비

1. **인코더의 최종 출력 (Key & Value Source)은 이러하다:**
    - 영어 문장 "I love machine learning."은 트랜스포머 인코더의 여러 레이어를 통과한다.
    - 인코더의 최종 레이어 출력은 각 입력 토큰(I, love, machine, learning, <eos>)에 대한 풍부한 문맥 정보를 담은 벡터 시퀀스(예: [vec_I, vec_love, vec_machine, vec_learning, vec_eos])가 된다.
    - 이 인코더 출력은 크로스 어텐션 레이어에서 **Key (K)**와 **Value (V)**의 소스(source)로 사용된다. 즉, K = Encoder_Output 이고 V = Encoder_Output 이다.
    - **예시:** Encoder_Output 은 영어 문장 "I love machine learning."의 각 단어에 해당하는 문맥이 반영된 벡터들이다.
2. **디코더의 현재 입력 (Query Source)은 이러하다:**
    - 디코더는 한국어 번역 문장을 생성한다. 학습 시에는 **이전 타임스텝까지의 정답 토큰(또는 <bos> 토큰)**을 입력으로 받는다. 이를 "Shifted Right"

입력이라고 한다.

- 예를 들어, "저는 머신러닝을 사랑합니다."를 생성하는 과정에서, 디코더가 이미 " <bos> 저는 머신러닝을"까지 생성했고 다음 단어인 "사랑합니다"를 예측해야 한다면, 디코더의 현재 입력은 " <bos> 저는 머신러닝을"이다.
- 이 디코더의 입력 시퀀스는 디코더 내부의 첫 번째 셀프 어텐션 레이어와 피드포워드 네트워크를 거쳐 새로운 벡터 시퀀스(예: [dec_vec_bos, dec_vec_저는, dec_vec_머신러닝을])로 변환된다.
- 이 디코더 중간 출력은 크로스 어텐션 레이어에서 **Query (Q)**의 소스(source)로 사용된다. 즉, Q = Decoder_Self_Attention_Output 이다.
- **예시:** Decoder_Self_Attention_Output 은 한국어 문장 "저는 머신러닝을"의 각 토큰에 해당하는 문맥이 반영된 벡터들이다.

2.3.2 2 단계: Q, K, V 선형 변환 (Linear Projections) 과정이다

크로스 어텐션 레이어는 학습 가능한 세 개의 가중치 행렬 W_Q, W_K, W_V를 가진다. 이들은 embed_dim 크기의 벡터를 같은 크기의 다른 표현 공간으로 투영(project)한다.

- **Query (Q) 생성은 이러하다:**
  - Q = Decoder_Self_Attention_Output @ W_Q 이다.
  - **예시:** [dec_vec_bos, dec_vec_저는, dec_vec_머신러닝을] 각각이 W_Q 와 곱해져 새로운 쿼리 벡터 [q_bos, q_저는, q_머신러닝을]를 형성한다. 이 쿼리들은 "나는 어떤 영어 단어에 집중해야 다음 한국어 단어를 잘 예측할 수 있을까?"라는 질문을 담고 있다.
- **Key (K) 생성은 이러하다:**
  - K = Encoder_Output @ W_K 이다.
  - **예시:** [vec_I, vec_love, vec_machine, vec_learning, vec_eos] 각각이 W_K 와 곱해져 새로운 키 벡터 [k_I, k_love, k_machine, k_learning, k_eos]를 형성한다. 이 키들은 "나는 이런 정보(영어 단어들)를 가지고 있어!"라고 말하는 셈이다.

- **Value (V) 생성은 이러하다:**
  - V = Encoder_Output @ W_V 이다.
  - **예시:** [vec_I, vec_love, vec_machine, vec_learning, vec_eos] 각각이 W_V 와 곱해져 새로운 값 벡터 [v_I, v_love, v_machine, v_learning, v_eos]를 형성한다. 이 값들은 키들이 제공하는 정보의 "실제 내용"이다.

2.3.3 3 단계: 어텐션 스코어 계산 (Attention Scores) 과정이다

각 쿼리 벡터(한국어 생성에 필요한 정보)가 모든 키 벡터(영어 원문 정보)와 얼마나 "관련되어" 있는지를 측정한다. 내적(dot product)을 사용하여 유사도를 계산한다.

- **계산:** Scores = Q @ K.T (쿼리 행렬과 키 행렬의 전치 곱)이다.
- **예시 (개념적):**
  - score_저는_I = q_저는 . k_I ("저는"이 영어 "I"와 얼마나 관련 깊은가?)이다.
  - score_저는_love = q_저는 . k_love ("저는"이 영어 "love"와 얼마나 관련 깊은가?)이다.
  - ...
  - score_머신러닝을_machine = q_머신러닝을 . k_machine ("머신러닝을"이 영어 "machine"과 얼마나 관련 깊은가?)이다.
  - score_머신러닝을_learning = q_머신러닝을 . k_learning ("머신러닝을"이 영어 "learning"과 얼마나 관련 깊은가?)이다.

결과는 (디코더_시퀀스_길이 x 인코더_시퀀스_길이) 크기의 점수 행렬이 된다.

2.3.4 4 단계: 스케일링 (Scaling) 과정이다

내적 값(Scores)을 키 벡터의 차원(d_k) 제곱근으로 나눈다. 이는 내적 값이 너무 커져 소프트맥스 함수의 기울기가 매우 작아지는 현상(vanishing gradients)을 방지하고 학습의 안정성을 높이는 방법이다.

- **계산:** Scaled_Scores = Scores / sqrt(d_k)이다.

2.3.5 5 단계: 마스킹 (Masking) - (선택적, 인코더 패딩 마스크) 과정이다

원본 영어 문장에 패딩 토큰(pad_token)이 포함되어 있다면, 해당 패딩 토큰은 실제 의미가 없으므로 어텐션 계산에 영향을 미치지 않도록 마스킹한다.

- **적용:** Scaled_Scores 행렬에서 패딩 토큰에 해당하는 열의 값을 -inf (음의 무한대)로 설정한다.
- **예시:** 인코더 입력이 "I love machine learning. <pad> <pad>"이고, vec_pad 가 있다면, k_pad 에 해당하는 점수 열은 -inf 로 바뀌는 것이다.
  - Scaled_Scores[:, index_of_pad_token] = -inf 이다.

2.3.6 6 단계: 소프트맥스 (Softmax) 및 어텐션 가중치 (Attention Weights) 계산 과정이다

스케일링된 점수에 소프트맥스 함수를 적용하여 각 키에 대한 확률 분포 형태의 가중치를 얻는다. 이 가중치 합은 1 이 된다.

- **계산:** Attention_Weights = Softmax(Scaled_Scores, dim=-1)이다.
- **의미:** 각 디코더 토큰(쿼리)이 인코더의 각 토큰(키)에 얼마나 집중해야 하는지를 나타내는 확률이다. 즉, "저는"이라는 한국어 토큰을 생성할 때 영어 원문의 "I"에 가장 높은 가중치를 부여하는 식이다.
- **예시:**
  - 한국어 쿼리 q_저는은 영어 키 k_I 에 가장 높은 어텐션 가중치를 가질 것이다.
  - 한국어 쿼리 q_머신러닝을은 영어 키 k_machine, k_learning 에 높은 어텐션 가중치를 가질 것이다.

2.3.7 7 단계: 가중합 (Weighted Sum) 및 컨텍스트 벡터 생성 과정이다

계산된 어텐션 가중치를 값(Value) 벡터에 곱한 후 모두 더하여 최종 출력을 생성한다.

- **계산:** Context_Vector = Attention_Weights @ V 이다.
- **의미:** 각 디코더 토큰(쿼리)에 대해, 인코더의 모든 토큰(값)으로부터 가중합된 컨텍스트 벡터가 생성된다. 이 벡터는 해당 디코더 토큰을 생성하는 데 필요한 인코더 측의 관련 정보를 요약한 것이다. 모델은 이 컨텍스트 벡터를 통해 영어

원문의 정보를 한국어 번역에 효과적으로 반영한다.

- **예시:**
  - context_for_저는 = (attn_weight_저는_I * v_I) + (attn_weight_저는_love * v_love) + ...이다.
  - context_for_머신러닝을 = (attn_weight_머신러닝을_I * v_I) + (attn_weight_머신러닝을_love * v_love) + ...이다.

이 Context_Vector는 (디코더_시퀀스_길이 x embed_dim) 크기의 행렬이 된다.

2.3.8 8 단계: 최종 선형 변환 (Output Projection) - (Multi-Head Attention 시) 과정이다

만약 Multi-Head Attention을 사용한다면 (대부분의 트랜스포머는 이를 사용한다), 각 헤드에서 생성된 Context_Vector들은 연결(concatenate)된 후, 최종적으로 하나의 선형 레이어 W_O를 통과하여 원래의 embed_dim 차원으로 다시 투영된다.

- **계산:** Final_Output = Concatenate(Head1_Context, Head2_Context, ...) @ W_O 이다.
- 이 Final_Output 은 디코더의 다음 서브 레이어(일반적으로 피드포워드 네트워크)로 전달된다.

2.3.9 9 단계: 손실 계산 및 역전파 (Loss Calculation & Backpropagation) 과정이다

- 디코더의 최종 출력은 선형 레이어(W_proj)와 소프트맥스 함수를 거쳐 한국어 어휘집 크기(vocab_size)의 확률 분포(logits)로 변환된다.
- 이 확률 분포는 디코더가 예측한 다음 한국어 토큰의 분포를 나타낸다.
- 모델의 예측(predicted_probabilities)은 실제 정답 한국어 토큰(decoder_label_ids)과 비교된다.
- **손실 함수 (Loss Function):** 주로 Cross-Entropy Loss 가 사용되어 예측과 실제 정답 사이의 차이를 측정한다.
  - Loss = CrossEntropyLoss(predicted_probabilities, actual_target_token_ids)이다.

- **역전파 (Backpropagation):** 계산된 손실은 신경망의 모든 가중치(임베딩 가중치, Q/K/V/O 행렬, 피드포워드 네트워크 가중치 등)를 업데이트하는 데 사용된다. 이는 옵티마이저(예: Adam)를 통해 수행된다.
    - Loss.backward() (기울기 계산)이다.
    - optimizer.step() (가중치 업데이트)이다.

2.4 크로스 어텐션 파이썬 의사코드 구현

다음은 트랜스포머의 크로스 어텐션 레이어를 파이썬(PyTorch)으로 구현한 의사코드 예시이다. 실제 트랜스포머 모델에서는 Multi-Head Attention의 일부로 구현된다.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class CrossAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        """
        크로스 어텐션 레이어를 초기화한다.
        Args:
            embed_dim (int): 입력 임베딩의 차원이다.
            num_heads (int): 어텐션 헤드의 개수이다.
        """
        super(CrossAttention, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads # 각 헤드의 차원이다.
        assert (
            self.head_dim * num_heads == embed_dim
        ), "embed_dim은 num_heads로 나누어 떨어져야 한다."

        # Query, Key, Value를 위한 선형 투영 레이어이다.
        # 디코더의 출력을 위한 Query 투영이다.
```

```
        self.query_proj = nn.Linear(embed_dim, embed_dim)
        # 인코더의 출력을 위한 Key 투영이다.
        self.key_proj = nn.Linear(embed_dim, embed_dim)
        # 인코더의 출력을 위한 Value 투영이다.
        self.value_proj = nn.Linear(embed_dim, embed_dim)
        # 모든 헤드의 출력을 결합하기 위한 최종 선형 투영 레이어이다.
        self.out_proj = nn.Linear(embed_dim, embed_dim)

    def forward(self, query, key, value, mask=None):
        """
        크로스 어텐션의 순방향 전파를 수행한다.
        Args:
            query (torch.Tensor): 디코더의 현재 입력 또는 이전 레이어의 출력이다.
                                  Shape: (batch_size, query_seq_len, embed_dim)
            key (torch.Tensor): 인코더의 최종 출력이다.
                                Shape: (batch_size, key_value_seq_len, embed_dim)
            value (torch.Tensor): 인코더의 최종 출력이다.
                                  Shape: (batch_size, key_value_seq_len, embed_dim)
            mask (torch.Tensor, optional): 어텐션 마스크이다 (패딩 토큰 무시 등).
                                           Shape: (batch_size, 1, 1, key_value_seq_len) 또는 (batch_size, 1,
query_seq_len, key_value_seq_len)
        Returns:
            tuple:
                - output (torch.Tensor): 크로스 어텐션 레이어의 출력이다.
                                         Shape: (batch_size, query_seq_len, embed_dim)
                - attention_weights (torch.Tensor): 계산된 어텐션 가중치이다.
                                                    Shape: (batch_size, num_heads, query_seq_len, key_value_seq_len)
        """
        batch_size = query.size(0)

        # 1. Q, K, V를 위한 선형 투영이다.
        # Q: (batch_size, query_seq_len, embed_dim) -> (batch_size, query_seq_len, embed_dim)
        # K, V: (batch_size, key_value_seq_len, embed_dim) -> (batch_size, key_value_seq_len, embed_dim)
        Q = self.query_proj(query)
        K = self.key_proj(key)
```

```
        V = self.value_proj(value)

        # 2. 헤드 분할이다.
        # Q, K, V: (batch_size, seq_len, embed_dim) -> (batch_size, seq_len, num_heads, head_dim)
        # 배치 행렬 곱셈을 위해 (batch_size, num_heads, seq_len, head_dim)으로 차원을 변경한다.
        Q = Q.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)
        K = K.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)
        V = V.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)

        # 3. 스케일드 닷-프로덕트 어텐션이다.
        # (Q @ K_T) / sqrt(head_dim)
        # (batch_size, num_heads, query_seq_len, head_dim) @ (batch_size, num_heads, head_dim,
key_value_seq_len)
        # -> (batch_size, num_heads, query_seq_len, key_value_seq_len)
        scores = torch.matmul(Q, K.transpose(-2, -1)) / (self.head_dim ** 0.5)

        # 마스크가 제공되면 적용한다.
        if mask is not None:
            # 마스크는 (batch_size, num_heads, query_seq_len, key_value_seq_len)로 브로드캐스트 가능해야
한다.
            # 일반적으로 마스크는 (batch_size, 1, 1, key_value_seq_len)과 같은 형태이다.
            scores = scores.masked_fill(mask == 0, float("-inf"))

        attention_weights = F.softmax(scores, dim=-1)

        # 4. Value와 곱한다.
        # (batch_size, num_heads, query_seq_len, key_value_seq_len) @ (batch_size, num_heads,
key_value_seq_len, head_dim)
        # -> (batch_size, num_heads, query_seq_len, head_dim)
        weighted_values = torch.matmul(attention_weights, V)

        # 5. 헤드를 연결하고 최종 선형 투영을 수행한다.
        # (batch_size, num_heads, query_seq_len, head_dim) -> (batch_size, query_seq_len, num_heads,
head_dim)
        # -> (batch_size, query_seq_len, embed_dim)
```

```
        weighted_values = weighted_values.transpose(1, 2).contiguous().view(batch_size, -1, self.embed_dim)

        output = self.out_proj(weighted_values)

        return output, attention_weights # attention_weights는 시각화 등을 위해 반환할 수 있다.

# --- 사용 예시 ---
if __name__ == "__main__":
    # 인코더 출력 (키, 값) 및 디코더 입력 (쿼리) 예시이다.
    batch_size = 2
    encoder_seq_len = 10 # 영어 문장의 길이이다.
    decoder_seq_len = 5  # 한국어 번역의 현재 길이이다.
    embed_dim = 512
    num_heads = 8

    # 인코더의 최종 출력 (key, value로 사용될 것이다).
    # 이는 영어 문장 "I love machine learning."의 임베딩이다.
    encoder_output = torch.randn(batch_size, encoder_seq_len, embed_dim)

    # 디코더의 현재 입력 또는 이전 디코더 레이어의 출력 (query로 사용될 것이다).
    # 이는 한국어 번역 "저는 머신러닝을"의 임베딩이다.
    decoder_input = torch.randn(batch_size, decoder_seq_len, embed_dim)

    # 크로스 어텐션 레이어를 초기화한다.
    cross_attention_layer = CrossAttention(embed_dim, num_heads)

    # 마스크 예시 (패딩 토큰을 무시하는 경우)이다.
    # encoder_output의 5 번째 토큰이 패딩이라고 가정한다.
    encoder_padding_mask = torch.ones(batch_size, 1, 1, encoder_seq_len, dtype=torch.bool)
    encoder_padding_mask[:, :, :, 4] = False # 5 번째 토큰에 해당하는 위치를 False로 설정한다.

    # 크로스 어텐션을 수행한다.
    # Query: decoder_input (한국어 번역의 현재 상태)
    # Key, Value: encoder_output (영어 원문)
    output, attn_weights = cross_attention_layer(
```

```
        query=decoder_input,
        key=encoder_output,
        value=encoder_output,
        mask=encoder_padding_mask # 마스크를 적용한다.
    )

    print("입력 쿼리(디코더 입력) shape:", decoder_input.shape)
    print("입력 키/값(인코더 출력) shape:", encoder_output.shape)
    print("크로스 어텐션 출력 shape:", output.shape)
    print("어텐션 가중치 shape (batch_size, num_heads, query_seq_len, key_value_seq_len):",
attn_weights.shape)

    # 출력 Shape: (batch_size, decoder_seq_len, embed_dim)
    # 어텐션 가중치 Shape: (batch_size, num_heads, decoder_seq_len, encoder_seq_len)
```

2.5 트랜스포머 구성요소 요약

2.5.1 Scaled Dot Product Attention

이 모듈은 어텐션의 기본 동작을 수행하는 핵심 요소이다. 입력으로 주어진 Q(Query), K(Key), V(Value) 벡터를 이용하여 다음 연산을 수행한다:

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

- Q, K 의 내적은 현재 토큰이 문맥 내 다른 토큰과 얼마나 관련 있는지를 나타낸다.
- 정규화를 위해 sqrt{d_k} 로 나눈다.
- softmax 를 통해 가중치를 확률화하고, V 에 곱해 최종 어텐션 출력을 생성한다.

이 모듈은 PyTorch에서 torch.matmul, F.softmax 등을 활용하여 구현된다.

2.5.2 *Multi-Head Attention*

어텐션을 단일 벡터로 계산하면 정보 손실이 크기 때문에, 여러 개의 어텐션 "헤드"를 병렬로 실행한 후 결과를 concat하여 사용한다. 각 헤드는 독립적으로 Q, K, V를 선형 변환한 후 Scaled Dot Product Attention을 적용한다.

이 모듈은 다음과 같은 단계를 포함한다:

- Q, K, V 를 헤드 수에 따라 분리하고,
- 각 헤드별 어텐션을 수행한 뒤,
- 결과를 concat 하여 최종 출력으로 만든다.

2.5.3 *Positional Encoding*

트랜스포머는 순서를 고려하지 않기 때문에 각 단어의 위치 정보를 인코딩해야 한다. 이를 위해 사인(sin), 코사인(cos) 함수를 기반으로 위치 인코딩 벡터를 생성한다.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right), \quad PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

해당 인코딩은 임베딩 벡터에 더해져 입력으로 사용된다.

2.5.4 *Position-wise Feed Forward Network*

각 토큰에 대해 독립적으로 적용되는 MLP 구조이다. 일반적으로 두 개의 Linear Layer 사이에 ReLU 활성함수가 들어간다. 형태는 다음과 같다:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

이는 토큰별로 표현력을 확장해주는 역할을 한다.

2.5.5 *Layer Normalization & Residual Connection*

트랜스포머는 각 모듈에서 residual connection(잔차 연결)과 layer normalization을 함께 사용한다.

- Residual Connection: 입력을 그대로 더하여 정보 손실을 줄임
- LayerNorm: 분포 정규화를 통해 학습 안정성 확보

이 두 요소는 모델의 깊이가 깊어질수록 성능 하락을 방지하는 데 기여한다.

2.6 트랜스포머 인코더 구조 요약

트랜스포머 인코더는 다음과 같은 순서로 구성된다:

1. 임베딩 + 포지셔널 인코딩 추가
2. Multi-Head Attention
3. Residual + LayerNorm
4. Feed Forward Network
5. Residual + LayerNorm

이 전체 블록을 여러 번 반복하여 문장의 문맥 정보를 점차 풍부하게 만든다.

2.7 PyTorch 기반 핵심 코드 분석

다음은 핵심 모듈별로 구조를 단순화한 의사 코드이다.

2.7.1 Scaled Dot Product Attention

```
class ScaledDotProductAttention(nn.Module):
    def forward(self, Q, K, V):
        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)
        attn = F.softmax(scores, dim=-1)
        return torch.matmul(attn, V)
```

2.7.2 MultiHeadAttention

```
class MultiHeadAttention(nn.Module):
    def forward(self, Q, K, V):
        # Q, K, V 선형 변환 및 split
        # 각 헤드별 attention 계산
        # concat 후 출력 선형 변환
        return output
```

2.7.3 Positional Encoding

```
class PositionalEncoding(nn.Module):
```

```
    def forward(self, x):
        # 위치별 sin, cos 벡터 계산 후 입력에 더함
        return x + self.pe[:, :x.size(1)]
```

2.7.4 EncoderLayer

```
class EncoderLayer(nn.Module):
    def forward(self, x):
        x = x + self_attn(x, x, x)  # Residual
        x = LayerNorm(x)
        x = x + FFN(x)  # Residual
        x = LayerNorm(x)
        return x
```

3. 자연어 처리 (NLP)

3.1 서론

자연어 처리(Natural Language Processing, NLP)는 인간 언어를 컴퓨터가 이해하고 해석할 수 있도록 하는 연구 분야이다. NLP는 텍스트 데이터나 음성 데이터와 같은 비정형 데이터를 구조화하고 분석하여 의미를 추출하고, 이를 기반으로 다양한 작업을 수행하는 기술을 다룬다. 대표적인 응용 분야로는 기계 번역, 질의응답 시스템, 문서 요약, 감정 분석, 문서 분류 등이 있다.

3.2 NLP 기본 개념

단어는 의미를 가지는 가장 작은 텍스트 단위이다. 사람이 언어를 통해 의사를 전달할 때 사용하는 기본 단위이며, 자연어 처리에서는 텍스트를 분석할 때 문장을 구성하는 단어 단위로 나누어 작업하는 것이 일반적이다.

토큰은 자연어 처리에서 입력 텍스트를 일정 단위로 나눈 결과물이다. 토큰은 꼭 단어 단위일 필요는 없으며, 경우에 따라서는 부분 단어(subword)나 심지어 문자 단위로 쪼갤 수도 있다. 예를 들어 "unbelievable"이라는 단어를 "un", "believe", "able"로 나누는 경우처럼, 의미적 조각으로 분리하는 것도 가능하다. 토크나이저(tokenizer)라는 도구를 이용하여 이 작업을 수행하며, 이를 통해 모델이 더 잘 일반화할 수 있도록 돕는다.

임베딩은 단어, 문장, 혹은 문서와 같은 이산적(discrete) 텍스트 정보를 연속적인 실수 공간(continuous vector space)에 매핑하는 과정을 의미한다. 자연어는 본래 기호적(symbolic)이고 이산적인 특성을 가지므로, 기계 학습 모델에 효과적으로 입력하기 위해서는 고정 차원의 연속적인 수치 벡터로 변환해야 한다. 이때 생성된 벡터를 임베딩(embedding)이라고 부른다.

3.3 임베딩 모델 및 학습 방법

대표적인 임베딩 모델에는 Word2Vec, GloVe, FastText가 있다.

이 중에 Word2Vec은 구글이 2013 년에 제안한 임베딩 방법으로, Skip-gram과 CBOW(Continuous Bag of Words) 두 가지 학습 구조를 가진다. Skip-gram은 중심 단어를 입력으로 주고 주변 단어를 예측하는 방식이다. 예를 들어 "The quick brown fox jumps"라는 문장에서 중심 단어를 "fox"로 두고 윈도 크기를 2 로 설정하면 주변 단어는 "brown"과 "jumps"가 된다. 모델은 "fox"를 보고 "brown", "jumps"를 맞추도록 학습한다. 수식으로 표현하면 다음과 같다.

$$\text{maximize} \prod_{-c \le j \le c, j \neq 0} P(w_{t+j} | w_t)$$

여기서 ccc는 윈도 크기, Wt는 중심 단어, Wt+j는 주변 단어이다.

임베딩 모델들은 주로 대규모 텍스트 데이터에서 비지도 학습 방식으로 학습된다. Word2Vec은 주변 단어를 예측하는 loss를 최소화한다.

대표적인 임베딩 학습 코드 예시는 다음과 같다.

```
# Word2Vec Skip-gram 간단 의사코드
for center_word, context_words in dataset:
    center_vec = word_embedding(center_word)
    for context in context_words:
        context_vec = word_embedding(context)
        loss += negative_sampling_loss(center_vec, context_vec)
    loss.backward()
    optimizer.step()
```

3.4 주요 NLP 기술

개체명 인식(Named Entity Recognition, NER)은 문장에서 특정한 개체를 찾아내고, 이를 사람(Person), 조직(Organization), 장소(Location) 등의 범주로 분류하는 기술이다. 예를 들어 "Apple Inc. was founded by Steve Jobs"라는 문장에서 "Apple Inc."는 조직, "Steve Jobs"는 사람으로 분류된다.

문장 분류(Text Classification)는 주어진 문장이 어떤 범주에 속하는지 예측하는 작업이다. 감정 분석(sentiment analysis)이나 뉴스 기사 분류(news classification)가 대표적인 예이다.

기계 번역(Machine Translation)은 한 언어로 작성된 문장을 다른 언어로 변환하는 기술이다. 예전에는 통계적 기계 번역(Statistical Machine Translation)이 주류였으나, 현재는 Transformer 기반 뉴럴 기계 번역(Neural Machine Translation)이 대세이다.

텍스트 요약(Text Summarization)은 긴 문서에서 핵심만을 추출하거나, 문장을 새롭게 생성하여 요약하는 기술이다. 추출적 요약(extractive summarization)과 생성적 요약(abstractive summarization)으로 나뉜다.

질의응답 시스템(Question Answering)은 입력된 질문에 대해 정확한 답변을 제공하는 시스템을 의미한다. 단순 정보 검색 기반 시스템부터, 복잡한 언어 이해를 통한 생성형 응답 시스템까지 다양한 형태가 존재한다.

3.5 N-gram

3.5.1 개념

N-gram은 연속된 **N개의 단어 또는 문자 단위 묶음**을 의미하는 개념이다. 자연어 처리(NLP)에서 주로 **문맥을 모델링**하거나 **언어 모델을 구성**할 때 사용된다.
예를 들어, 2-gram은 두 단어씩 묶어서 분석하고, 3-gram은 세 단어씩 묶는 방식이다.

3.5.2 종류

Unigram (1-gram): 단어 단위
예: ["나는", "밥을", "먹었다"]

Bigram (2-gram): 연속된 2 개 단어
예: [("나는", "밥을"), ("밥을", "먹었다")]

Trigram (3-gram): 연속된 3 개 단어
예: [("나는", "밥을", "먹었다")]

3.5.3 활용 예시

언어 모델링: 다음 단어를 예측하거나 문장의 확률을 계산할 때 사용된다. 예를 들어 Bigram 모델에서는 "밥을" 다음에 "먹었다"가 나올 확률을 학습한다.

스팸 필터링: 특정 n-gram 조합이 자주 스팸에 등장하면 그 확률을 기반으로 필터링 가능하다.

자동 완성 / 오타 수정: 앞의 n-1 단어 기반으로 다음 단어를 예측한다.

텍스트 유사도 계산: 문장을 n-gram으로 분해해 겹치는 비율로 유사도를 계산할 수 있다.

3.6 정확도 지표 및 계산 방법

모델 성능을 정량적으로 평가하기 위해 여러 지표를 사용한다. 각 지표는 다음과 같은 수식으로 정의된다.

Accuracy는 전체 데이터 중 올바르게 예측한 비율이다.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

여기서,

TP(True Positive): 참을 참으로 예측

TN(True Negative): 거짓을 거짓으로 예측

FP(False Positive): 거짓을 참으로 예측

FN(False Negative): 참을 거짓으로 예측

예를 들어, 전체 100 개 중 90 개를 맞췄다면 Accuracy는 0.9 이다.

Precision은 모델이 참이라고 예측한 것 중 실제 참인 비율이다.

$$\text{Precision} = \frac{TP}{TP + FP}$$

예를 들어, 50 개를 참이라고 예측했는데 이 중 45 개가 실제 참이라면 Precision은 0.9 이다.

Recall은 실제 참인 데이터 중 모델이 맞춘 비율이다.

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1 Score는 Precision과 Recall의 조화 평균이다.

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

간단한 예를 들면 다음과 같다.

전체 데이터가 100 개일 때,
TP = 40, TN = 50, FP = 5, FN = 5 라면,

Accuracy는 (40+50)/100=0.9(40+50)/100 = 0.9,
Precision은 40/(40+5)=0.88840/(40+5) = 0.888,
Recall은 40/(40+5)=0.88840/(40+5) = 0.888,
F1 Score는 약 0.888 이다.

코드로 계산하면 다음과 같다.

간단한 Precision, Recall, F1 계산 예제

precision = TP / (TP + FP)

recall = TP / (TP + FN)

f1 = 2 * (precision * recall) / (precision + recall)

3.7 BLEU와 ROUGE 지표

BLEU(Bilingual Evaluation Understudy)는 기계 번역 결과를 평가하는 지표이다. 번역 결과와 기준(reference) 번역 간의 n-gram 일치 정도를 기반으로 한다.

BLEU 점수는 다음 수식으로 계산된다.

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

여기서 BP는 번역이 기준 번역보다 짧을 경우 부여하는 패널티(Brevity Penalty)이다.

예를 들어, 기준 문장이 "the cat is on the mat", 기계 번역이 "the cat is on mat"이라면, 1-gram precision은 5 개 중 4 개가 일치하므로 0.8 이다.

ROUGE(Recall-Oriented Understudy for Gisting Evaluation)는 요약 성능을 평가하는 지표이다. 기준 요약과 모델 요약 간의 n-gram 겹침 정도를 기반으로 하며, 주로 Recall 중심으로 평가한다.

ROUGE-1 은 다음 수식으로 계산된다.

$$\text{ROUGE-1} = \frac{\text{overlapping unigrams}}{\text{total unigrams in reference}}$$

기준 요약이 "the cat sat on the mat"이고, 모델 요약이 "the cat sat mat"일 경우, 일치하는 단어는 "the", "cat", "sat", "mat"로 총 4 개이며, 기준은 6 단어이므로 ROUGE-1 은 $4/6 \approx 0.666$ 이다.

BLEU, ROUGE 지표를 계산하는 예제 코드 형태는 다음과 같다.

```
# 간단한 BLEU 점수 계산 예시

import nltk

reference = [['the', 'cat', 'is', 'on', 'the', 'mat']]

candidate = ['the', 'cat', 'is', 'on', 'mat']

score = nltk.translate.bleu_score.sentence_bleu(reference, candidate)
```

3.8 BLUE 함수 의사코드

3.8.1 개요

입력: candidate 문장, reference 문장

1. candidate, reference 를 토큰화한다.

2. 1-gram, 2-gram, 3-gram, 4-gram 각각에 대해:
   - candidate 의 n-gram 들을 만든다.
   - reference 의 n-gram 들을 만든다.
   - candidate n-gram 과 reference n-gram 사이에서 일치하는 개수를 센다.
   - precision = 일치 개수 / candidate 의 전체 n-gram 개수

3. 모든 n-gram precision 에 대해 log 를 취하고 평균한다.

4. 기하 평균 = exp(로그 평균)

5. Brevity Penalty (BP) 계산:
   - candidate 길이 < reference 길이면: BP = exp(1 - reference_len / candidate_len)
   - candidate 길이 >= reference 길이면: BP = 1

6. BLEU = BP × 기하 평균

출력: BLEU 점수

3.8.2 예시를 통한 단계별 BLEU 계산 과정

예시 문장

- Reference: "the cat is on the mat"
- Candidate: "the cat is on mat"

토큰화하면:

- reference = ["the", "cat", "is", "on", "the", "mat"]
- candidate = ["the", "cat", "is", "on", "mat"]

3.8.3 1-gram precision 계산

candidate의 1-grams:

"the", "cat", "is", "on", "mat"

reference의 1-grams:

"the", "cat", "is", "on", "the", "mat"

일치하는 것:

"the", "cat", "is", "on", "mat"
→ 5 개 모두 일치 (주의: "the"는 reference에 2 번 있지만 candidate에 1 번만 있으므로 통과)

precision (1-gram) = 5 / 5 = 1.0

3.8.4 2-gram precision 계산

candidate의 2-grams:

"the cat", "cat is", "is on", "on mat"

reference의 2-grams:

"the cat", "cat is", "is on", "on the", "the mat"

일치하는 것:

"the cat", "cat is", "is on" (총 3 개)

precision (2-gram) = 3 / 4 = 0.75

3.8.5 3-gram precision 계산

candidate의 3-grams:

"the cat is", "cat is on", "is on mat"

reference의 3-grams:

"the cat is", "cat is on", "is on the", "on the mat"

일치하는 것:

"the cat is", "cat is on" (총 2 개)

precision (3-gram) = 2 / 3 ≈ 0.6667

3.8.6 4-gram precision 계산

candidate의 4-grams:

"the cat is on", "cat is on mat"

reference의 4-grams:

"the cat is on", "cat is on the", "is on the mat"

일치하는 것:

"the cat is on" (총 1 개)

precision (4-gram) = 1 / 2 = 0.5

3.8.7 log 평균 및 exp 계산

각 precision의 log를 구한다:

log(1.0) = 0

log(0.75) ≈ -0.28768

log(0.6667) ≈ -0.40547

log(0.5) ≈ -0.69315

이들의 평균:

$$\frac{0 - 0.28768 - 0.40547 - 0.69315}{4} \approx -0.34658$$

exp(평균):

exp(−0.34658)≈0.707

3.8.8 Brevity Penalty (BP) 계산

candidate 길이 = 5
reference 길이 = 6

candidate가 더 짧기 때문에 BP를 계산한다:

$$BP = \exp(1 - 6/5) = \exp(1 - 1.2) = \exp(-0.2) \approx 0.8187$$

3.8.9 최종 BLEU 점수 계산

BLEU=BP×기하 평균=0.8187×0.707≈0.579

즉, 이 예제의 BLEU 점수는 약 **0.579** 이다.

3.8.10 간단한 코드 예시 (Python + Numpy)

```
import numpy as np
from collections import Counter

def bleu_score(candidate, reference):
    candidate = candidate.split()
    reference = reference.split()

    precisions = []
    for n in range(1, 5):
        cand_ngrams = Counter([tuple(candidate[i:i+n]) for i in range(len(candidate)-n+1)])
        ref_ngrams = Counter([tuple(reference[i:i+n]) for i in range(len(reference)-n+1)])

        overlap = sum(min(count, ref_ngrams[ngram]) for ngram, count in
cand_ngrams.items())
        total = sum(cand_ngrams.values())
        precisions.append(overlap / total if total > 0 else 0)

    if min(precisions) > 0:
        log_precisions = np.mean([np.log(p) for p in precisions])
        geo_mean = np.exp(log_precisions)
    else:
        geo_mean = 0

    c, r = len(candidate), len(reference)
    bp = np.exp(1 - r/c) if c < r else 1.0

    return bp * geo_mean

# 예제
candidate = "the cat is on mat"
reference = "the cat is on the mat"
```

```
print(f"BLEU score: {bleu_score(candidate, reference):.4f}")
```

출력:

```
BLEU score: 0.5790
```

3.9 결론

자연어 처리는 단어를 수치화하고, 그 수치 표현을 기반으로 다양한 작업을 수행하는 기술 체계이다. 임베딩 모델은 텍스트를 연속 벡터로 변환하고, 다양한 신경망 모델이 문맥과 의미를 추론하며, 정확도 지표를 통해 성능을 정량적으로 평가한다. NLP에서는 Word2Vec, GloVe, FastText와 같은 임베딩 기법은 물론, BERT나 GPT 같은 대규모 사전 학습 언어 모델이 문맥 기반 처리를 가능하게 하였다. BLEU, ROUGE, F1 Score 등 평가 지표들은 모델의 품질을 객관적으로 측정하는 데 역할을 한다.

4. 정규표현식

4.1 개요

정규표현식(Regular Expression)은 문자열에서 특정 패턴을 찾거나, 치환하거나, 분리하기 위한 문자열 검색 도구이다. 정규식은 다양한 프로그래밍 언어에서 사용되며, 복잡한 텍스트 처리 작업을 간단하게 해결할 수 있도록 해준다. 주로 데이터 검증, 로그 분석, 텍스트 파싱 등에서 많이 활용된다.

4.2 역사

정규식의 개념은 1950 년대 미국 수학자 스티븐 클리(Stephen Kleene)에 의해 처음 소개되었다. 그는 유한 오토마타 이론과 관련된 수학적 표현식을 통해 정규 언어(regular language)의 이론적 기초를 마련하였다. 이후 Unix 운영체제의 텍스트 처리 도구(예: grep, sed 등)에 채택되면서 실용적인 도구로 발전하였다. 오늘날에는 Python, JavaScript, Java, Perl, Ruby 등의 다양한 언어에서 표준적인 정규식 기능을 제공하고 있다.

4.3 주요 정규식 문법과 예시

정규식은 다양한 기호와 메타문자를 통해 패턴을 구성한다. 아래는 주요 문법과 간단한 예시이다.

문법	설명	예시	설명
.	임의의 한 문자	a.c	'abc', 'axc'에 매칭
^	문자열의 시작	^a	'apple'에 매칭, 'banana'에는 X
$	문자열의 끝	e$	'apple', 'title'에 매칭
[]	문자 집합	[aeiou]	모음에 매칭
[^]	제외 문자 집합	[^0-9]	숫자가 아닌 문자
*****	0 개 이상의 반복	a*	'', 'a', 'aa' 모두 매칭
+	1 개 이상의 반복	a+	'a', 'aa'는 매칭, ''은 X
?	0 또는 1 개	a?	'', 'a'는 매칭
{n}	정확히 n번 반복	a{3}	'aaa'에만 매칭
{n,}	n번 이상 반복	a{2,}	'aa', 'aaa', ...에 매칭
{n,m}	n~m번 반복	a{1,3}	'a', 'aa', 'aaa'에 매칭
₩d	숫자 문자	₩d+	'123', '5' 등에 매칭
₩w	단어 문자	₩w+	'abc123', '_' 등에 매칭
₩s	공백 문자	₩s+	' ', '₩t', '₩n' 등에 매칭
()	그룹화	(abc)+	'abc', 'abcabc'에 매칭

4.4 정규식으로 할 수 있는 작업 예시

전화번호 추출: ₩d{2,3}-₩d{3,4}-₩d{4}

특정 단어 포함 여부 확인: ₩bpython₩b

HTML 태그 제거: <[^>]+>

4.5 Python의 re 라이브러리 사용법

Python에서는 re 모듈을 사용하여 정규식을 처리할 수 있다. 주요 함수는 다음과 같다.

```
import re

# 검색
re.search(r"apple", "I like apple")  # 매칭 객체 반환

# 매칭
re.match(r"apple", "apple pie")  # 문자열 처음부터 매칭 시도

# 전부 검색
re.findall(r"₩d+", "There are 24 cats and 3 dogs.")  # ['24', '3'] 반환

# 치환
re.sub(r"cat", "dog", "I have a cat")  # 'I have a dog'

# 분할
re.split(r"₩s+", "split by space or tab")  # ['split', 'by', 'space', 'or', 'tab']
```

4.6 결론

정규식은 복잡한 문자열 처리 작업을 간결하게 수행할 수 있게 해주는 강력한 도구이다.
정규식 문법을 잘 이해하고 실습을 통해 익숙해지면, 데이터 분석과 웹 크롤링, 로그 처리 등 다양한 분야에서 효율적인 텍스트 처리가 가능하다.

5. OpenAI CLIP

5.1 CLIP 개요

CLIP(Contrastive Language–Image Pre-training)은 OpenAI에서 개발한 모델로, 텍스트와 이미지를 함께 학습하여 다양한 멀티모달 작업에서 강력한 성능을 발휘한다. CLIP은 자연어 설명과 이미지를 연결하는 방식으로 학습되었으며, 이미지 분류, 검색, 캡션 생성 등 여러 작업에 사용될 수 있다.

5.2 주요 기능

- **텍스트-이미지 매칭**: 텍스트 설명과 이미지 간의 유사도를 계산.
- **제로샷 학습**: 추가적인 학습 없이도 새로운 데이터셋에 적용 가능.
- **멀티모달 검색**: 텍스트를 사용하여 이미지 검색, 또는 이미지를 사용하여 텍스트 검색 가능.
- **이미지 분류**: 기존의 레이블이 아닌 자연어를 사용하여 분류.

5.3 설치

CLIP은 Python 환경에서 PyTorch 라이브러리를 기반으로 작동한다. 설치는 다음과 같이 진행된다.

5.3.1 설치 방법

pip install git+https://github.com/openai/CLIP.git

pip install torch torchvision

5.3.2 의존성

Python 3.7+

PyTorch 1.7.1+

torchvision

5.4 모델 로드

CLIP은 다양한 사전 학습된 모델을 제공한다. 다음은 모델을 로드하는 코드이다.

예제 코드:

```
import clip
```

```
import torch

# CLIP 모델과 토크나이저 로드
model, preprocess = clip.load("ViT-B/32", device="cuda" if torch.cuda.is_available() else "cpu")
```

5.5 이미지와 텍스트 유사도 계산

CLIP은 이미지와 텍스트의 표현을 동일한 임베딩 공간으로 변환하여 유사도를 계산한다.

5.5.1 예제 코드:

```
from PIL import Image

# 이미지와 텍스트 준비
image = preprocess(Image.open("example.jpg")).unsqueeze(0).to(device)
text = clip.tokenize(["A dog", "A cat"]).to(device)

# 모델 예측
with torch.no_grad():
    image_features = model.encode_image(image)
    text_features = model.encode_text(text)

# 유사도 계산
logits_per_image, logits_per_text = model(image, text)
probs = logits_per_image.softmax(dim=-1).cpu().numpy()

print("Label probabilities:", probs)
```

5.6 제로샷 이미지 분류

CLIP은 추가적인 학습 없이도 다양한 이미지 분류 작업에 활용될 수 있다.

예제 코드:

```
# 분류 레이블 정의
labels = ["a photo of a dog", "a photo of a cat"]
text = clip.tokenize(labels).to(device)
```

```
# 이미지 처리
image = preprocess(Image.open("example.jpg")).unsqueeze(0).to(device)

# 예측
with torch.no_grad():
    image_features = model.encode_image(image)
    text_features = model.encode_text(text)

    # 유사도 계산
    logits_per_image = (image_features @ text_features.T).softmax(dim=-1)

# 결과 출력
print("Predicted label:", labels[logits_per_image.argmax()])
```

5.7 멀티모달 검색

CLIP은 텍스트 기반 이미지 검색과 이미지 기반 텍스트 검색을 지원한다.

예제 코드

```
# 데이터 준비
images = [preprocess(Image.open(path)).unsqueeze(0).to(device) for path in ["image1.jpg",
"image2.jpg"]]
texts = clip.tokenize(["A beach", "A forest"]).to(device)

# 이미지 및 텍스트 특징 추출
with torch.no_grad():
    image_features = torch.cat([model.encode_image(img) for img in images])
    text_features = model.encode_text(texts)

# 유사도 계산
similarity = image_features @ text_features.T

print("Similarity matrix:", similarity.cpu().numpy())
```

5.8 모델 확장

CLIP은 다양한 작업에서 확장 가능하며, 사용자 정의 데이터셋에 미세 조정(Fine-tuning)하여 활용할 수 있다. 다만, 미세 조정은 기본적인 제로샷 성능을 저하시킬 수 있으므로 주의가 필요하다.

예제 코드:

```
# 사용자 정의 데이터셋 준비
from torch.utils.data import DataLoader

custom_dataset = CustomDataset("path/to/data")
data_loader = DataLoader(custom_dataset, batch_size=32, shuffle=True)

# 모델 미세 조정
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)
criterion = torch.nn.CrossEntropyLoss()

for epoch in range(epochs):
    for images, texts, labels in data_loader:
        images = preprocess(images).to(device)
        texts = clip.tokenize(texts).to(device)

        optimizer.zero_grad()

        image_features = model.encode_image(images)
        text_features = model.encode_text(texts)

        loss = criterion(image_features, labels)
        loss.backward()
        optimizer.step()
```

5.9 결론

CLIP은 텍스트와 이미지를 통합적으로 처리할 수 있는 강력한 모델로, 다양한 멀티모달 작업에서 우수한 성능을 제공한다. 이를 활용하면 기존의 딥러닝 모델로는 어려웠던 작업을 손쉽게 수행할 수 있다.

6. Variational Autoencoders (VAE)

6.1 서론

Variational Autoencoders(이하 VAE)는 딥러닝 분야에서 데이터의 압축 및 생성에 사용되는 확률적 생성 모델이다. 이는 Autoencoder의 확장된 형태로, 데이터를 잠재 공간(latent space)에서 확률 분포로 표현하고 이를 활용하여 새로운 데이터를 생성할 수 있는 기능을 갖추고 있다.

입력 데이터를 잠재 공간(latent space)으로 인코딩하고, 다시 디코딩하여 원본 데이터를 재구성하는 방식으로 학습된다. VAE의 목적은 주어진 데이터를 잠재 공간의 확률 분포로부터 샘플링하고 이를 통해 새로운 데이터를 생성하는 것이다.

VAE는 인코더와 디코더로 구성된다. 인코더는 입력 데이터를 잠재 공간의 분포로 매핑하며, 디코더는 이 잠재 벡터를 다시 원본 데이터로 변환한다. VAE는 변분 추정(variational inference)을 활용하여 잠재 변수의 분포를 추정, 학습한다.

6.2 VAE의 주요 개념

VAE는 일반적인 Autoencoder와 유사한 구조를 가지며, 입력 데이터를 압축하고 이를 복원하는 과정을 수행한다. 그러나 Autoencoder가 단순히 입력 데이터를 재구성하는 데 초점을 맞추는 반면, VAE는 잠재 공간에서 확률 분포를 학습하고 이를 활용하여 새로운 데이터를 생성할 수 있도록 설계되었다.

Autoencoder

Autoencoder는 입력 데이터를 저차원 공간으로 압축하는 인코더(Encoder)와 이를 다시 원래 데이터로 복원하는 디코더(Decoder)로 구성된다. 입력 데이터를 효율적으로 압축하는 데 사용되며, 입력과 출력 간의 차이를 최소화하는 방식으로 학습된다.

Variational Autoencoder

VAE는 Autoencoder의 한계를 극복하기 위해 확률적 접근 방식을 도입하였다. 인코더는 입력 데이터를 평균(μ)과 표준편차(σ)로 나타내는 확률 분포로 변환하며, 디코더는 이 분포에서 샘플링한 값을 기반으로 데이터를 복원한다. 이 과정은 잠재 공간에서 확률적으로 데이터를 샘플링할 수 있도록 설계되어 새로운 데이터 생성이 가능하다.

6.3 VAE의 구조

VAE는 인코더, 리파라메터라이제이션 트릭(Reparameterization Trick), 디코더의 세 가지 주요 구성 요소로 이루어져 있다.

인코더(Encoder)
인코더는 입력 데이터를 잠재 공간의 확률 분포로 매핑한다. 구체적으로, 입력 데이터를 평균(μ)과 표준편차(σ)로 표현하며, 이를 통해 잠재 공간 상의 데이터가 정규 분포를 따르도록 학습한다.

리파라메터라이제이션 트릭(Reparameterization Trick)
잠재 변수 zz를 샘플링하는 과정에서 역전파를 가능하게 하기 위해 사용되는 기술이다. 이는 잠재 변수를 다음과 같이 표현한다:

$$z = \mu(x) + \sigma(x) \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0,1)$$

이 방법은 확률적 샘플링을 가능하게 한다.

디코더(Decoder)
디코더는 잠재 공간에서 샘플링된 데이터를 기반으로 입력 데이터를 복원한다. 이 과정은 원본 데이터와 최대한 유사한 출력을 생성하도록 학습된다.

6.4 VAE 학습 목표

6.4.1 KL-발산과 VAE에서의 역할

VAE의 학습 과정에서 KL-발산(Kullback-Leibler Divergence)은 중요한 역할을 한다. KL-발산은 두 확률 분포 간의 차이를 측정하는 지표로, VAE에서는 잠재 분포(q(z|x))와 표준 정규 분포(p(z)) 간의 차이를 최소화하려는 목적을 가진다.
KL-발산은 다음과 같이 정의된다:
D_{KL}(q(z|x) || p(z)) = ∫ q(z|x) log (q(z|x) / p(z)) dz
VAE에서는 q(z|x)가 입력 x에 대한 잠재 변수 z의 조건부 분포로, 평균 μ와 분산 σ²를 가지는 정규분포 N(μ, σ²)로 표현된다. 반면, p(z)는 표준 정규분포 N(0, 1)이다. VAE는 이 두 분포 간의 차이를 줄이도록 학습된다.

6.4.2 KL-발산의 수학적 전개

KL-발산을 계산하는 과정에서 q(z|x)와 p(z)가 모두 정규분포이기 때문에, 이 두 분포 간의 KL-발산은 다음과 같이 간단히 전개된다.

정규분포 q(z|x)는 평균 μ와 분산 σ^2를 가지고, 표준 정규분포 p(z)는 평균 0 과 분산 1 을 가진다. KL-발산을 계산하면 다음과 같은 식으로 전개된다:

$D_{KL}(q(z|x) \| p(z)) = \int N(\mu, \sigma^2) \log (N(\mu, \sigma^2) / N(0, 1)) dz$

결과적으로 KL-발산은 다음과 같이 간단한 수식으로 나타날 수 있다:

$D_{KL}(q(z|x) \| p(z)) = 0.5 [\log(1/\sigma^2) + \sigma^2 + \mu^2 - 1]$

6.4.3 KL-발산 항목별 해석

KL-발산을 구성하는 각 항목은 다음과 같은 의미를 가진다:

1. $\log(1/\sigma^2)$: 분산 σ^2에 대한 상대적인 정보량을 측정한다. σ^2가 클수록 분포가 더 넓어지며, 그만큼 정보가 덜 밀집되어 있다는 의미이다.
2. σ^2: q(z|x)의 분산이 1(표준 정규분포)의 분산에 얼마나 가까운지, 또는 얼마나 멀리 떨어져 있는지를 측정한다.
3. μ^2: q(z|x)의 평균이 표준 정규분포의 평균(0)과 얼마나 차이가 나는지를 측정한다. μ^2가 클수록 q(z|x)의 평균은 표준 정규분포와 거리가 멀다는 의미이다.
4. -1: 표준 정규분포의 분산이 1 이므로, KL-발산의 마지막 항목은 이 값에 대한 보정이다.

6.4.4 VAE의 학습 과정과 KL-발산의 역할

VAE는 손실 함수로 재구성 오류(reconstruction loss)와 KL-발산을 결합하여 최적화한다. 손실 함수는 다음과 같다:

$L = E_{q(z|x)}[\log p(x|z)] - D_{KL}(q(z|x) \| p(z))$

첫 번째 항목인 $E_{q(z|x)}[\log p(x|z)]$는 재구성 오류로, 모델이 입력을 얼마나 잘 재구성하는지 측정한다. 두 번째 항목인 $D_{KL}(q(z|x) \| p(z))$는 KL-발산으로, 잠재 공간의 분포가 표준 정규분포에 얼마나 가까운지를 측정한다.

VAE는 이 두 항목을 동시에 최소화하면서 학습된다. KL-발산 항목은 잠재 공간의 분포가 표준 정규분포에 가까워지도록 유도하며, 재구성 오류는 입력 데이터를 잘 재구성할 수 있도록 유도한다.

6.4.5 VAE 학습에 대한 의사코드

VAE의 학습 과정에서 KL-발산을 계산하고 이를 손실 함수에 반영하는 의사코드를 다음과 같이 작성할 수 있다.

```
import torch
import torch.nn as nn

# VAE 모델 정의 (인코더, 디코더)
class VAE(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(VAE, self).__init__()
        self.encoder = Encoder(input_dim, latent_dim)  # 인코더
        self.decoder = Decoder(latent_dim, input_dim)  # 디코더

    def forward(self, x):
        # 인코딩
        mu, log_var = self.encoder(x)
        z = self.reparameterize(mu, log_var)

        # 디코딩
        reconstructed_x = self.decoder(z)
        return reconstructed_x, mu, log_var

    def reparameterize(self, mu, log_var):
        # reparameterization trick
        std = torch.exp(0.5 * log_var)
        eps = torch.randn_like(std)
        return mu + eps * std

# KL-발산 계산 함수
def kl_divergence(mu, log_var):
    # KL-발산: 0.5 * (mu^2 + exp(log_var) - log(var) - 1)
    return -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

# 손실 함수
def loss_function(reconstructed_x, x, mu, log_var):
    # 재구성 손실
    reconstruction_loss = nn.MSELoss(reduction='sum')(reconstructed_x, x)
```

```
    # KL-발산 손실
    kl_loss = kl_divergence(mu, log_var)

    # 총 손실
    return reconstruction_loss + kl_loss

# 모델 학습 예시
vae = VAE(input_dim=784, latent_dim=128)
optimizer = torch.optim.Adam(vae.parameters(), lr=1e-3)

for epoch in range(epochs):
    for batch_idx, (data, _) in enumerate(train_loader):
        optimizer.zero_grad()

        # 순전파
        reconstructed_x, mu, log_var = vae(data)

        # 손실 계산
        loss = loss_function(reconstructed_x, data, mu, log_var)

        # 역전파
        loss.backward()
        optimizer.step()
```

6.5 VAE의 주요 특징

VAE는 다음과 같은 특징을 가진다.

생성 모델

잠재 공간에서 샘플링하여 새로운 데이터를 생성할 수 있다. 이는 기존의 Autoencoder가 가지지 못한 중요한 특징이다.

잠재 공간 학습

데이터의 저차원 표현을 학습하며, 이 표현을 활용하여 데이터의 특성을 분석하거나 변형할 수 있다.

확률적 접근

확률론적 접근 방식을 통해 모델이 더욱 유연하게 데이터를 학습하고 일반화할 수 있다.

6.6 VAE의 응용 분야

VAE는 다음과 같은 다양한 분야에서 활용되고 있다.

이미지 생성

MNIST, CIFAR-10 등 데이터셋에서 새로운 이미지를 생성하는 데 사용된다.

데이터 클러스터링

잠재 공간에서 데이터의 군집을 분석하거나 군집 간 차이를 확인할 수 있다.

노이즈 제거

손상된 데이터에서 노이즈를 제거하거나 결함 데이터를 복원하는 데 사용된다.

시계열 데이터 분석

시계열 데이터를 변형하거나 미래 데이터를 예측하는 데 활용된다.

6.7 결론

VAE는 확률적 접근 방식을 통해 데이터를 효율적으로 압축할 뿐만 아니라, 새로운 데이터를 생성할 수 있는 강력한 생성 모델이다. 이러한 특징은 다양한 데이터 분석 및 생성 응용 분야에서 VAE를 중요한 도구로 자리 잡게 한다. 앞으로 VAE의 발전은 더욱 복잡한 데이터 구조를 이해하고 활용하는 데 기여할 것으로 기대된다.

7. Stable Diffusion

7.1 개요

Stable Diffusion은 텍스트 설명을 기반으로 고품질 이미지를 생성하는 혁신적인 생성 모델로, 최근 많은 관심을 받고 있다. 이 모델은 확산 모델(Diffusion Model)을 활용하여 점진적으로 이미지를 생성하며, 텍스트에서 이미지로 변환하는 과정에서 뛰어난 성능을 보인다. Stable Diffusion은 오픈 소스로 공개되어 있으며, 다양한 연구자와 개발자들이 이를 활용할 수 있도록 되어 있다.

Stable Diffusion은 입력된 텍스트 프롬프트를 기반으로 고해상도 이미지를 생성하는 Latent Diffusion Model (LDM) 계열의 생성 모델이다. 이 모델은 직접 고해상도 이미지 공간에서 Diffusion을 수행하는 것이 아니라, 이미지의 latent 표현 공간에서 효율적으로 노이즈 제거 과정을 진행한다. Stable Diffusion은 크게 다음 세 가지 주요 구성요소로 이루어진다:

1. VAE (Variational Autoencoder): 이미지 ↔ latent 공간 변환

2. U-Net: 노이즈가 섞인 latent를 복원하는 denoising 네트워크

3. Text Encoder (CLIP): 텍스트를 임베딩하여 조건으로 사용

추론 과정은 "완전한 노이즈"에서 시작하여 점진적으로 노이즈를 제거하며 텍스트 조건에 부합하는 이미지를 생성하는 방식으로 진행된다.

7.2 확산 모델 (Diffusion Model) 개요

확산 모델은 이미지 생성 과정에서 **노이즈 추가와 제거**를 통해 이미지를 생성하는 방식이다. 이 과정은 크게 두 가지로 나뉜다:

노이즈 첨가(Forward Process): 원본 이미지에서 점진적으로 노이즈를 추가하여 최종적으로 완전한 노이즈 이미지를 생성하는 과정이다. 이 과정은 이미지에서 세부 정보를 점차적으로 제거한다.

노이즈 제거(Reverse Process): 학습된 모델을 사용하여 노이즈 이미지를 원본 이미지로 복원하는 과정이다. 모델은 이미지의 복원 과정을 통해 노이즈를 제거하며 최종적으로 고품질의 이미지를 생성한다.

7.3 Stable Diffusion의 작동 원리

Stable Diffusion의 작동 원리는 텍스트 설명을 기반으로 이미지를 생성하는 데 필요한 **세 가지 주요 구성 요소**로 나눠진다:

노이즈 첨가 및 제거 과정: Stable Diffusion은 확산 모델의 원리를 확장하여 사용한다. 모델은 이미지를 점진적으로 생성하기 위해 **DDPM(Denoising Diffusion Probabilistic Models)** 방식을 개선하여 사용하며, 텍스트에서 이미지를 생성하는 데 필요한 정보를 점진적으로 복원한다.

텍스트 임베딩: 텍스트 설명을 입력받아 벡터 형태로 변환하는 과정이다. Stable Diffusion은 이를 위해 **CLIP 모델**을 사용하여 텍스트를 효과적으로 임베딩한다. CLIP 모델은 텍스트와 이미지를 연결하는 강력한 도구로, 이를 통해 텍스트 설명을 정확하게 반영하는 이미지를 생성할 수 있다.

U-Net 아키텍처: Stable Diffusion은 이미지 생성에서 U-Net 아키텍처를 활용한다. 이 아키텍처는 다운샘플링(Downsampling)과 업샘플링(Upsampling)을 통해 정보를 효과적으로 전달하며, 이미지의 복원 과정을 효율적으로 수행한다.

7.4 Stable Diffusion의 학습 과정

Stable Diffusion은 대규모 **텍스트-이미지 데이터셋**을 기반으로 학습된다. 이 데이터셋은 이미지와 해당하는 텍스트 설명으로 구성되며, 모델은 텍스트 설명을 임베딩 벡터로 변환한 후, 노이즈를 제거하는 과정을 통해 고품질의 이미지를 생성하는 방법을 학습한다. 학습 과정에서 모델은 텍스트와 이미지 간의 관계를 학습하고, 이를 기반으로 다양한 텍스트 입력에 대해 적절한 이미지를 생성할 수 있다.

7.5 텍스트-이미지 매핑

Stable Diffusion은 텍스트를 이미지로 변환하기 위해 **CLIP 모델**을 사용한다. CLIP은 텍스트와 이미지를 동시에 처리할 수 있는 모델로, 텍스트 설명을 이미지와 일치하도록 학습한다. Stable Diffusion은 이 모델을 활용하여 텍스트와 이미지 간의 관계를 정확하게 파악하고, 텍스트에 적합한 이미지를 생성한다. 이 과정에서 텍스트의 의미를 정확히 반영하는 이미지를 점진적으로 생성한다.

7.6 Stable Diffusion 동작 과정

Stable Diffusion의 동작은 크게 다음과 같다.

7.6.1 전체 동작 과정 요약

입력 텍스트를 임베딩한다.

latent 공간에서 시작점을 노이즈로 설정한다.

U-Net을 이용해 노이즈를 점진적으로 제거한다.

최종 latent를 VAE를 통해 디코딩하여 이미지를 복원한다.

7.6.2 주요 전체 과정 의사코드

```
# 1. 텍스트 임베딩
text_embedding = TextEncoder(prompt)

# 2. 초기 노이즈 latent 생성
```

```
latent = sample_normal_distribution(shape=(batch_size, latent_dim))

# 3. 노이즈 제거 반복
for timestep in reversed(timesteps):
    noise_pred = UNet(latent, timestep, text_embedding)
    latent = update_latent(latent, noise_pred, timestep)

# 4. latent를 디코딩하여 이미지 복원
image = VAE_Decoder(latent)
```

7.6.3 VAE (Variational Autoencoder)

VAE는 **이미지**를 **latent 공간**으로 인코딩하고, latent를 다시 **이미지**로 복원하는 역할을 한다. Stable Diffusion에서는 고해상도 직접 처리를 피하고, 작은 크기의 latent 공간(예: 64x64x4)에서 연산을 수행하도록 한다.

주요 동작 의사코드:

```
# Encoder: 이미지 → latent
mu, sigma = Encoder(image)
z = mu + sigma * random_normal_like(mu)  # reparameterization trick 사용

# Decoder: latent → 이미지
reconstructed_image = Decoder(z)
```

설명:

- **Encoder** 는 입력 이미지를 압축하여 μ(x), σ(x) 를 생성한다.
- **Reparameterization Trick** 을 이용하여 latent zz 를 샘플링한다.
- **Decoder** 는 latent zz 를 받아 이미지를 복원한다.

7.6.4 U-Net

U-Net은 **노이즈가 낀 latent**를 입력받아,
"어떤 노이즈가 섞였는지"를 예측하여 노이즈를 제거하는 네트워크이다.

Stable Diffusion에서는 일반 U-Net에 **Cross-Attention** 구조를 추가하여, 텍스트 조건을 이미지 복원 과정에 통합한다.

7.6.5 주요 동작 의사코드

```
def UNet(latent, timestep, text_embedding):
    x = initial_conv(latent)

    for down_block in downsampling_path:
        x = down_block(x)
        x = cross_attention(x, text_embedding)

    x = bottleneck(x)

    for up_block in upsampling_path:
        x = up_block(x)
        x = cross_attention(x, text_embedding)

    output = final_conv(x)
    return output  # 예측한 노이즈
```

7.6.6 설명

- 다운샘플링 → 업샘플링 경로를 통해 특징을 추출하고 복원한다.
- 각 주요 블록에서는 **Cross-Attention** 을 수행하여 텍스트 정보를 반영한다.
- 최종적으로 현재 latent 에 섞여 있는 **노이즈**를 예측하여 반환한다.

7.6.7 Text Encoder (CLIP Text Encoder)

Text Encoder는 텍스트 프롬프트를 받아, U-Net의 Attention 모듈에서 사용 가능한 고정 크기의 임베딩 벡터로 변환하는 역할을 한다.
Stable Diffusion에서는 OpenAI CLIP의 Text Encoder를 변형하여 사용한다.

주요 동작 의사코드

```
def TextEncoder(prompt):
    tokens = tokenize(prompt)
    text_embedding = transformer_encoder(tokens)
    return text_embedding
```

설명

- 입력된 텍스트를 먼저 토큰화(tokenization)한다.
- Transformer 를 통해 각 토큰에 대해 임베딩을 얻는다.
- 최종적으로 이 텍스트 임베딩이 U-Net 의 Attention 모듈에 주입된다.

7.6.8 최종 요약

Stable Diffusion은 다음과 같은 구조로 동작한다:

1. 텍스트를 임베딩한다.
2. 랜덤 노이즈 latent 에서 시작한다.
3. U-Net 을 통해 노이즈를 점진적으로 제거하며 latent 를 정제한다.
4. 최종 latent 를 VAE 를 통해 고해상도 이미지로 복원한다.

이 과정은 학습 시에는 정방향으로 (노이즈 추가 → 노이즈 제거 예측) 학습되고,
생성 시에는 역방향으로 (노이즈 제거 → 이미지 복원) 진행된다.

7.7 Stable Diffusion 설치 및 실행 방법

Stable Diffusion을 실행하려면, 먼저 **필수 라이브러리 및 의존성**을 설치해야 한다. 다음은 설치 과정이다:

시스템 요구 사항:

Python 3.8 이상

PyTorch 1.7 이상

CUDA 11.1 이상 (GPU 사용 시)

레포지토리 클론: GitHub에서 Stable Diffusion의 소스를 복사한다:

git clone https://github.com/Stability-AI/stablediffusion

cd stablediffusion

필수 패키지 설치: 필요한 Python 라이브러리를 설치한다:

pip install -r requirements.txt

모델 파일 다운로드: 모델 파일은 Stable Diffusion의 공식 페이지에서 다운로드 할 수 있다.

모델 실행: 모델을 실행하려면 다음 명령어를 사용한다:

python scripts/txt2img.py --prompt "a beautiful landscape" --plms

이 명령어는 텍스트 설명에 맞는 이미지를 생성한다.

7.8 Stable Diffusion의 활용

Stable Diffusion은 주로 **예술적 작업**이나 **디지털 아트**에서 널리 사용된다. 텍스트 프롬프트를 사용하여 고품질 이미지를 자동으로 생성할 수 있으며, 이로 인해 디자이너나 예술가들은 창작 과정에서 더 많은 시간을 절약할 수 있다. 또한, **게임 개발**이나 **광고 디자인**, **패션 디자인** 등 다양한 산업 분야에서 활용될 수 있다.

7.9 Stable Diffusion의 장점

Stable Diffusion의 가장 큰 장점은 **개방성**과 **효율성**이다. 기존의 GAN 모델에 비해 더 안정적이고, 텍스트에서 이미지로의 변환을 매우 효과적으로 수행할 수 있다. 또한, 오픈 소스로 제공되어 누구나 이를 활용하여 다양한 프로젝트를 진행할 수 있다는 점에서 매우 유용하다.

7.10 결론

Stable Diffusion은 이미지 생성 기술에서 중요한 진전을 이룬 모델이다. 확산 모델을 활용하여 안정적으로 고품질 이미지를 생성할 수 있으며, CLIP 모델을 사용하여 텍스트와 이미지를 정확히 매핑하는 방식은 매우 효과적이다.

7.11 참고 문헌

Stable Diffusion GitHub: https://github.com/Stability-AI/stablediffusion

블로그에서 제공된 링크: https://daddynkidsmakers.blogspot.com/2024/02/stable-diffusion.html

8. 전체 미세조정(FFT) 기반 모델 학습

8.1 소형 언어모델(sLLM)

sLLM(Small Language Model)은 파라미터 수가 비교적 적고, 계산 자원이 제한된 환경에서도 실행 가능한 소형 자연어 처리 모델을 의미한다. GPT-2 small, DistilBERT, TinyLLaMA, Phi-2 등은 sLLM의 대표적인 예이다. sLLM은 경량화되어 IoT, 엣지 장치, 모바일 등에서도 동작 가능하며, 응답 속도 및 추론 비용 측면에서도 우수하다.

8.2 전체 미세조정 (Full Fine-Tuning, FFT)

FFT는 사전학습된 언어모델의 모든 파라미터를 대상으로 다시 학습을 진행하는 방식이다. 기존 LLM의 모든 가중치를 업데이트함으로써 특정 도메인에 완전히 최적화된 모델을 만들 수 있다.

장점:

도메인 전용 성능 최대화

자유로운 모델 커스터마이징 가능

단점:

메모리, GPU 등 자원 요구가 매우 큼.

예) LLaMA 7B 이상 대형 모델: 최소 24–80GB 필요 (최적화 여부에 따라 다름)

학습 시간이 길고 비용이 큼

과적합(overfitting) 우려 존재

8.3 PEFT(Parameter-Efficient Fine-Tuning)

PEFT는 전체 모델이 아닌 일부 파라미터(모듈)만 미세조정하는 경량화된 학습 기법이다. 대표적으로 LoRA, Prefix Tuning, Adapter Tuning, BitFit 등이 있다.

장점:

학습 파라미터 수가 적어 자원 효율적임

기존 모델의 원본 가중치를 보존

빠른 학습 및 재사용 가능성 높음

단점:

극단적인 도메인 전이에는 한계

원본 모델 품질에 따라 성능 제한 가능성

8.4 분산 학습 전략: DP, DDP, FSDP, DeepSpeed

8.4.1 Data Parallelism (DP)

여러 GPU에서 동일한 모델을 복사하고 각기 다른 배치 데이터를 분산 학습하는 구조이다. PyTorch의 DataParallel API로 구현되며, 학습 도중 한 GPU가 다른 GPU보다 늦게 처리되면 전체 속도가 느려진다. 또한 파라미터 동기화를 위해 매 step마다 메인 GPU로 통신이 집중되어 병목이 발생한다.

```
import torch

import torch.nn as nn


model = MyModel()

model = nn.DataParallel(model)  # 단일 GPU 메모리에 비해 확장성 낮음
```

8.5 DistributedDataParallel (DDP)

PyTorch의 공식 분산 병렬 처리 기법으로, 각 GPU가 독립적으로 파라미터를 계산하고, backward 단계에서 통신하여 그래디언트를 평균낸다. DDP는 효율적이며, 노드 간 GPU 사용도 가능하다.

```
# DDP 예시 (단일 노드, 다중 GPU)

import os

import torch

import torch.distributed as dist

from torch.nn.parallel import DistributedDataParallel as DDP


os.environ['MASTER_ADDR'] = 'localhost'

os.environ['MASTER_PORT'] = '12355'

dist.init_process_group(backend='nccl', rank=0, world_size=1)
```

```
model = MyModel().cuda()
model = DDP(model)
```

8.6 Fully Sharded Data Parallel (FSDP)

모델 파라미터, 그라디언트, 옵티마이저 상태까지 모두 GPU 간에 분산시켜 메모리 사용량을 줄이는 기법이다. 매우 큰 모델도 적은 메모리로 학습할 수 있다. 각 GPU는 전체 모델의 일부분만 보유하므로 메모리 병목을 해소할 수 있다.

```
from torch.distributed.fsdp import FullyShardedDataParallel as FSDP
from torch.distributed.fsdp.wrap import default_auto_wrap_policy

model = MyModel().cuda()
fsdp_model = FSDP(model, auto_wrap_policy=default_auto_wrap_policy)
```

8.7 DeepSpeed

DeepSpeed는 Microsoft에서 개발한 분산 학습 프레임워크로, ZeRO(Zero Redundancy Optimizer) 알고리즘을 이용하여 optimizer state, gradient, parameter를 분산 저장한다. 또한 offloading, mixed precision, pipeline parallelism, activation checkpointing 등을 통해 메모리 절약 및 속도 향상을 지원한다.

사용 예시:

deepspeed_config.json 설정 파일 작성 (학습 단계, ZeRO stage, batch size 등 지정)

다음 코드로 모델 실행:

```
import deepspeed
model = MyModel()
model_engine, optimizer, _, _ = deepspeed.initialize(
    model=model,
    config="deepspeed_config.json",
    model_parameters=model.parameters()
```

)

예시 설정 파일 (deepspeed_config.json):

```
{
  "train_batch_size": 32,
  "fp16": {"enabled": true},
  "zero_optimization": {
    "stage": 2,
    "allgather_partitions": true,
    "allgather_bucket_size": 2e8,
    "overlap_comm": true,
    "reduce_scatter": true
  }
}
```

8.8 결론

sLLM은 경량화된 언어모델로 다양한 환경에 적합하며, 이를 효율적으로 미세조정하기 위해 FFT보다 PEFT 기법이 주로 사용된다. 대규모 학습 시에는 DDP, FSDP, DeepSpeed와 같은 분산 학습 프레임워크를 활용하여 자원을 효율적으로 사용해야 한다.

9. PEFT기반 Gemma-2B 의학 질의응답 파인튜닝

9.1 서론

대규모 언어 모델(LLM, Large Language Model)은 막대한 연산 자원을 요구하며, 전체 파라미터를 파인튜닝하는 방식은 메모리 사용량이 높고 계산 비용도 크다. 이에 따라, 효율적으로 도메인 특화 파인튜닝을 수행할 수 있는 방식으로 PEFT(Parameter-Efficient Fine-Tuning)가 주목받고 있다. 본 보고서에서는 PEFT의 대표 기법 중 하나인 LoRA(Low-Rank Adaptation)를 포함한 이론적 개요와 함께, Hugging Face의 Gemma-2B 모델을 의료 질의응답 태스크에 파인튜닝하는 실제 코드를 분석하여 소개한다.

9.2 PEFT(Parameter-Efficient Fine-Tuning)

PEFT는 사전 학습된 거대 언어 모델의 모든 가중치를 조정하지 않고, 특정 모듈 혹은 계층만을 학습 대상으로 설정함으로써 효율적인 파인튜닝을 가능하게 하는 접근 방식이다. 대표적인 장점은 다음과 같다.

자원 절약: 전체 모델이 아닌 일부만 학습하므로 GPU 메모리 사용량이 적다.

빠른 수렴: 학습 대상이 적어 빠르게 수렴하며 적은 데이터로도 효과적이다.

모듈화 및 재사용: 동일 모델에 대해 여러 도메인 전용 LoRA adapter를 따로 저장해 재사용할 수 있다.

대표적인 PEFT 기법에는 다음이 있다:

LoRA: 저랭크 행렬을 삽입하여 Linear 연산을 보완함.

Adapter Tuning: Transformer 블록에 작은 네트워크를 덧붙여 학습함.

Prefix/Prompt Tuning: 입력 시퀀스에 도메인 특화 토큰을 삽입함.

9.3 LoRA(Low-Rank Adaptation) 이론과 적용 구조

LoRA는 self-attention이나 feedforward 블록의 선형 계층에 대해 고정된 원본 가중치에 저랭크 행렬을 추가함으로써 파라미터 수를 줄이고 효율적인 학습을 가능하게 한다.

기존 연산:

$y = Wx$ LoRA

적용 후:

$y = (W + BA)x$

여기서 A, B는 학습 가능한 행렬이다.

$A \in \mathbb{R}^{r \times d}, B \in \mathbb{R}^{d \times r}$

r 값은 학습 능력과 파라미터 수를 조절하는 핵심 하이퍼파라미터이다.

일반적으로 r=8~64 사이에서 설정한다.

```
class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        std_dev = 1 / torch.sqrt(torch.tensor(rank).float())
        self.A = torch.nn.Parameter(torch.randn(in_dim, rank) * std_dev)
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

LoRAConfig 파라미터 설명:

```
LoraConfig(
    r=64,
    lora_alpha=32,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj",
"down_proj"],
    lora_dropout=0.05,
```

```
    bias="none",
    task_type="CAUSAL_LM"
)
```

r: 저랭크 행렬의 차원, 작을수록 연산량 감소, 클수록 표현력 증가

lora_alpha: 학습 중 scaling factor로 작동하며, 학습 안정성과 관련됨

target_modules: LoRA가 삽입될 Transformer 내 Linear 레이어 이름

lora_dropout: 학습 중 적용될 dropout 확률

bias: bias 항을 LoRA와 함께 학습할지 여부 (none, all, lora_only 등)

task_type: 파인튜닝 과제 종류 (CAUSAL_LM, SEQ_CLS 등). CAUSAL_LM: 다음 토큰을 순차적으로 예측하는 언어 생성 모델 (예: GPT류)

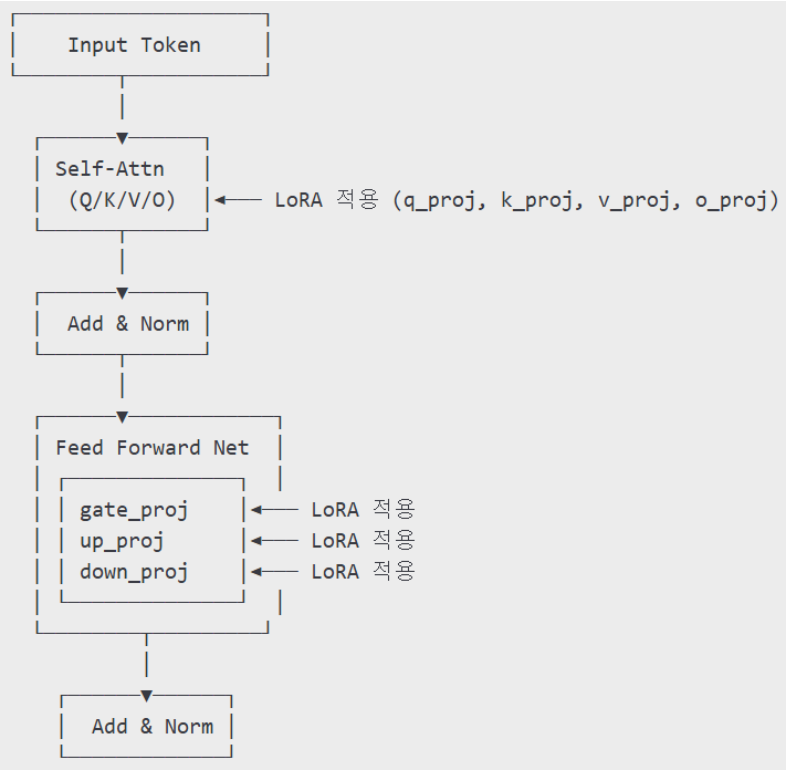
SEQ_CLS: 입력 전체에 대해 단일 클래스를 예측하는 분류 모델 (예: 감정 분석, 문장 분류)

적용 대상 모듈:

q_proj, k_proj, v_proj, o_proj: Self-Attention의 쿼리, 키, 밸류, 출력 projection에 대응함.

gate_proj, up_proj, down_proj: Transformer 블록의 Feedforward Network(FFN)에서 사용되는 세 개의 핵심 선형 계층이다.

- gate_proj: gating mechanism 적용 전 입력을 처리
- up_proj: hidden dimension을 확장함 (ex. 4096 → 11008)
- down_proj: 다시 차원을 줄이며 원래 출력으로 복원



LoRA는 이러한 projection 계층에서 학습 가능한 저랭크 행렬을 덧붙이는 방식으로 삽입되며, 실제 forward 계산 시 기존 weight는 고정된 채 BA 행렬만을 통해 미세 조정된다.

9.4 양자화(Quantization) 및 BitsAndBytesConfig 설명

양자화는 모델 파라미터를 float 대신 int4/8 로 표현하여 자원 사용을 줄이는 기술이다. bitsandbytes는 Hugging Face에서 채택된 양자화 라이브러리이며, 특히 4bit 양자화에서 매우 효과적이다.

설정 예시:

```
BitsAndBytesConfig(
  load_in_4bit=True,
  bnb_4bit_use_double_quant=True,
  bnb_4bit_quant_type="nf4",
  bnb_4bit_compute_dtype=torch.bfloat16
)
```

옵션 설명:

- load_in_4bit: 모델을 4bit 로 로드할지 여부
- bnb_4bit_use_double_quant: 이중 양자화 수행으로 표현력 향상
- bnb_4bit_quant_type:
    - nf4: NormalFloat4, 정규화된 4 비트 부동소수점
    - fp4: 기본 부동소수점 4 비트 양자화, 정밀도는 낮음
- bnb_4bit_compute_dtype: 계산 dtype (bfloat16, float16, float32) 지정. 일반적으로 bfloat16 사용

9.5 데이터셋 구조 및 프롬프트 템플릿

{

"instruction": "문제",

"input": "(선택적) 추가정보",

"output": "정답"

}

프롬프트 구성:

<start_of_turn>user

Below is an instruction that describes a task. Write a response that appropriately completes the request.

{instruction + input}

<end_of_turn>

<start_of_turn>model

{output}

<end_of_turn>

학습 데이터 크기 권장:

1,000~10,000 개 → 소규모 LoRA 파인튜닝에 적절

10,000 개 이상 → 안정적인 수렴 및 일반화 성능 확보 가능

9.6 전체 파인튜닝 코드 상세 설명

*9.6.1 **모델 및 양자화 설정***

bnb_config = BitsAndBytesConfig(...)

model = AutoModelForCausalLM.from_pretrained(model_id,
quantization_config=bnb_config, device_map="auto")

사전 학습된 모델을 4bit 형식으로 로드하며, GPU에 자동 할당

*9.6.2 **토크나이저 준비***

tokenizer = AutoTokenizer.from_pretrained(model_id, add_eos_token=True)

tokenizer.pad_token = tokenizer.eos_token

tokenizer.padding_side = 'right'

*9.6.3 **데이터셋 처리***

text_column = [generate_prompt(dp) for dp in dataset['train']]

dataset = dataset['train'].add_column("prompt", text_column)

dataset = dataset.map(lambda samples: tokenizer(samples["prompt"]), batched=True)

instruction과 input을 조합해 프롬프트 생성 후 tokenizer로 변환

*9.6.4 **데이터 분리***

dataset = dataset.train_test_split(test_size=0.1)

train_data = dataset["train"]

test_data = dataset["test"]

*9.6.5 **LoRA 설정 및 적용***

model = prepare_model_for_kbit_training(model)

lora_config = LoraConfig(...)

model = get_peft_model(model, lora_config)

학습 가능한 부분만 활성화하고 LoRA 어댑터 적용

*9.6.6 **학습 준비 및 수행:***

```
trainer = SFTTrainer(
    model=model,
    train_dataset=train_data,
    eval_dataset=test_data,
    peft_config=lora_config,
    args=transformers.TrainingArguments(
        per_device_train_batch_size=1,
        gradient_accumulation_steps=4,
        max_steps=100,
        learning_rate=2e-4,
        output_dir="outputs",
        save_strategy="epoch"
    ),
    data_collator=transformers.DataCollatorForLanguageModeling(tokenizer, mlm=False),
)
trainer.train()
```

Supervised Fine-Tuning 은 사전 학습된 LLM을 다음과 같이 (prompt, answer) 쌍 데이터로 지도 학습하는 Trainer이다.

input (prompt)	**output (정답)**
"서울은 어디에 있나요?"	"서울은 대한민국에 있습니다."

- per_device_train_batch_size: GPU 1 개당 처리할 미니배치 크기를 설정한다. 메모리가 적을 경우 1 로 설정.

- gradient_accumulation_steps: 여러 스텝 동안 gradient를 누적한 뒤 한 번만 backward/update 수행. VRAM이 부족할 때 효과적이며, 위 설정은 batch size=4 와 유사한 효과.
- max_steps: optimizer step 횟수. 스텝 수 기반 학습 종료 조건
- learning_rate: 옵티마이저 학습률. LoRA에서는 2e-4~5e-4 가 일반적인 범위.
- output_dir: 학습 결과, 체크포인트, 로그 등을 저장할 디렉토리 경로.
- save_strategy: 모델을 저장하는 주기를 정의한다. epoch이면 한 epoch마다 저장.

9.6.7 모델 병합 및 저장

```
merged_model = PeftModel.from_pretrained(base_model, new_model).merge_and_unload()
merged_model.save_pretrained("merged_model")
tokenizer.save_pretrained("merged_model")
```

최종 모델을 base 모델과 병합하여 추론에 적합한 형태로 저장함

9.7 레퍼런스

https://lightning.ai/lightning-ai/studios/code-lora-from-scratch?section=featured

9.8 결론

Gemma-2B 모델에 대해 LoRA 기반 PEFT를 적용하여 의료 분야 질의응답 학습을 수행하는 방법을 다루었다. LoRA의 구조와 어댑터 삽입 위치, 랭크 설정 효과 등을 분석하였고, bitsandbytes를 활용한 4bit 양자화가 어떻게 리소스를 절약하는지를 설명하였다. 아울러, 사용자 데이터셋의 예시로 의료 데이터셋 구조와 적정 학습 데이터 크기, 전체 학습 코드 흐름을 구체적으로 정리하였다.

10. 라마 3 파인튜닝

이 문서는 Meta LLaMA 3 모델을 기반으로 한 로컬 환경 파인튜닝(fine-tuning) 워크플로우를 다룬다. 특히 QLoRA를 활용한 모델 양자화와 LoRA 기반의 파라미터 효율적 미세조정(PEFT)을 통해 저사양 환경에서도 대형 언어 모델을 미세조정하는 실용적인 접근 방식을 설명한다. HuggingFace의 학습 파이프라인을 기반으로, 실제 의료 도메인 데이터셋을 예시로 하여 데이터 전처리부터 학습, 평가 및 배포까지 전체 과정을 단계별로 해설한다.

10.1 주요 라이브러리 설명 및 로딩

import os, pandas as pd, json, torch, wandb

from transformers import AutoTokenizer, AutoModelForCausalLM

from huggingface_hub import login

from trl import SFTTrainer, setup_chat_format

from datasets import Dataset, load_dataset

▸ 핵심 라이브러리 설명

transformers: 다양한 사전 학습된 언어 모델을 로딩하고, 학습 파이프라인 및 토크나이저 기능을 제공.

wandb: Weights & Biases는 실험 추적과 시각화를 통해 반복 실험 시 성능 향상을 유도.

trl: transformers에 기반한 RLHF 및 SFT 툴킷으로, LLM 학습을 더욱 구조화된 방식으로 수행 가능.

datasets: 고성능 데이터셋 처리 도구로, 대규모 텍스트 데이터셋을 효율적으로 로딩 및 병렬 처리 가능.

huggingface_hub: 모델과 토크나이저를 저장하고 공유하기 위한 API. 학습된 결과를 업로드하거나 다운로드하는 데 사용됨.

10.2 사용자 정의 데이터 로딩 함수: load_dataset_from_folder()

def load_dataset_from_folder(folder_path):

▸ 기능 설명

로컬 디렉토리에 존재하는 JSON 파일들을 읽어들여 학습에 사용할 수 있는 구조로 가공.

responses라는 키를 가진 JSON 구조를 기반으로, 리스트 형태의 답변(answer) 필드를 하나의 문자열로 병합.

데이터프레임으로 변환한 뒤 HuggingFace Dataset 객체로 래핑하여 후속 학습 파이프라인에 통합.

10.3 파인튜닝 전체 파이프라인: train()

설정에 따라 학습을 시작함.

10.4 베이스 모델 설정 및 인증

base_model = "Undi95/Meta-Llama-3-8B-hf"

login(token='...')

wandb.login(key='...')

HuggingFace Hub에서 LLaMA 3 기반 모델을 로드한다. 현재 모델은 8B 파라미터를 가진 GPT 스타일의 Causal Language Model.

HuggingFace 토큰과 wandb API 키를 이용해 외부 서비스에 접근 권한을 설정함.

실험 로그는 wandb 대시보드에서 실시간으로 확인 가능하며, 실험 비교에도 유용하다.

10.5 사용자 데이터셋 로딩 및 셔플링

custom_dataset = load_dataset_from_folder('./dataset')

custom_dataset = custom_dataset.shuffle(seed=65)

사용자 정의 데이터셋은 로컬의 JSON 파일로 구성되며, shuffle(seed=65)를 통해 학습 순서를 무작위화함.

seed 고정은 결과의 재현성(reproducibility)을 보장하여 학술적/산업적 실험 기준에 부합함.

10.6 QLoRA 기반 4-bit 양자화 설정

bnb_config = BitsAndBytesConfig(...)

▸ QLoRA 개념 및 파라미터 해설

QLoRA는 Q-formatted LoRA로, 양자화된 모델에 LoRA 기법을 결합한 방식이다.

load_in_4bit=True: 4 비트 precision으로 모델을 로딩함으로써 GPU 메모리 사용량을 약 3~4 배 절감 가능.

bnb_4bit_quant_type="nf4": NormalFloat4 는 학습 가능성이 높은 정규 분포 중심의 표현력 향상된 포맷.

bnb_4bit_use_double_quant=True: 2 차 양자화를 통해 정보 손실 최소화와 압축 효율을 향상시킴.

bnb_4bit_compute_dtype=torch.float16: 연산은 float16 으로 진행되어 연산 성능을 확보.

10.7 모델 및 토크나이저 로딩 + 대화 포맷 구성

model = AutoModelForCausalLM.from_pretrained(...)

tokenizer = AutoTokenizer.from_pretrained(...)

model, tokenizer = setup_chat_format(model, tokenizer)

▸ 설명

AutoModelForCausalLM은 GPT 스타일의 토큰 순서 예측 모델을 의미하며, LLaMA는 이에 해당한다.

setup_chat_format()은 대화형 프롬프트 템플릿을 모델과 토크나이저에 적용하여 학습과 추론에서 일관된 프롬프트 구조를 사용 가능케 한다.

특히 role-based template을 사용하는 경우 “user”, “assistant” 등의 역할 구분이 학습 성능에 큰 영향을 준다.

10.8 LoRA 기반 파라미터 효율적 미세조정

peft_config = LoraConfig(...)

model = get_peft_model(model, peft_config)

▸ LoRA 핵심 요소

r=16: LoRA rank 값으로, 작은 값일수록 파라미터 수가 줄어 메모리 효율적.

lora_alpha=32: scaling 계수로서 모델의 표현력 보정을 위한 계수.

target_modules: q_proj, v_proj 등 Attention projection 레이어만을 학습 대상으로 설정.

▸ 이점

전체 모델 파라미터의 수 퍼센트만 학습되므로 메모리 사용량, 학습 시간, 저장 공간이 대폭 감소함.

10.9 데이터셋 전처리 및 통합

HuggingFace 데이터셋(ai-medical-chatbot)과 커스텀 데이터셋을 각각 채팅 템플릿 형태로 전처리.

사용자 질문과 응답을 role: user, role: assistant 구조로 텍스트로 변환하여 통일된 학습 구조 구성.

concatenate_datasets()로 병합 후, train_test_split(test_size=0.1)을 통해 9:1 비율로 학습/검증 데이터 분리.

10.10 학습 하이퍼파라미터 구성

training_arguments = TrainingArguments(...)

per_device_train_batch_size=1: 4bit 양자화로도 VRAM 16GB 이상이 요구되므로 소형 배치 사용.

gradient_accumulation_steps=2: 실제 배치를 두 배로 증가시키는 효과로, 안정적 학습을 유도함.

optim="paged_adamw_32bit": 메모리 최적화를 위해 페이징된 옵티마이저 사용.

eval_steps=0.2: epoch 중 20% 간격으로 검증 수행하여 빠른 이상 감지 가능.

group_by_length=True: 유사한 길이 샘플끼리 묶어 학습 효율을 높이는 기법.

10.11 지도학습 기반 트레이닝: SFTTrainer

trainer = SFTTrainer(...)

trainer.train()

SFTTrainer는 HuggingFace Trainer를 확장하여 LLM의 텍스트 미세조정을 위해 설계됨.

packing=False 설정은 각 샘플을 독립된 개별 문장으로 학습함을 의미하며, 학습의 안정성을 높인다.

내부적으로 gradient clipping, learning rate scheduling 등의 기능이 포함됨.

10.12 평가 및 모델 배포

파인튜닝 전후로 동일한 질문에 대해 모델의 응답을 비교함으로써 성능 향상을 정성적으로 확인.

model.save_pretrained()로 로컬에 저장하고, push_to_hub()로 HuggingFace Hub에 업로드 가능.

학습된 모델은 추후 Inference API 또는 Gradio, FastAPI 등 프레임워크로 쉽게 활용 가능.

10.13 결론 및 확장

이 교안은 LLaMA 3 기반 모델을 저비용 환경에서 효과적으로 파인튜닝하기 위한 실무적 기법을 담고 있으며, 최신 연구 흐름(QLoRA, PEFT, W&B 로깅 등)을 실제 코드에 통합하는 방식을 익힐 수 있다. 이후 Vision-Language 모델이나 Instruction Tuning, Reinforcement Learning with Human Feedback(RLHF) 등의 고도화 모델링으로의 확장도 고려해볼 수 있다.

11. RAG (Retrieval-Augmented Generation)

11.1 개요

RAG(Retrieval-Augmented Generation)는 대규모 언어 모델(LLM: Large Language Model)의 한계를 극복하기 위해 외부 지식 기반을 검색해 생성 과정에 반영하는 방식이다. RAG는 외부 문서나 데이터베이스에서 관련 정보를 검색(Retrieval)하고, 이를 입력에 증강(Augmentation)하여, 최종적으로 텍스트를 생성(Generation)하는 세 단계를 거친다.

기존 LLM은 파라미터 내에 내재된 정보만으로 응답을 생성하지만, RAG는 검색된 외부 정보를 활용함으로써 최신성과 정확성, 사실성(factuality)을 향상시킨다.

11.2 역사

RAG는 2020 년 Facebook AI Research(FAIR)에 의해 처음 제안되었다. 이 기술은 Open-domain Question Answering 및 Fact-based Generation에서 LLM의 한계를 극복하고자 하는 목적으로 개발되었으며, DPR(Dense Passage Retrieval)을 검색기로 사용하고, BART와 같은 사전학습 언어모델을 생성기로 사용하는 구조였다.

학습은 metric learning 접근을 따르며, 주어진 질문에 대해 정답 문서와 오답 문서 집합을 구성하고, 정답 문서의 유사도를 최대화하고 오답의 유사도를 최소화하는 softmax 기반 loss를 최적화한다:

DPR은 질문과 문서를 각각 독립된 BERT 인코더(EQ, EP)를 통해 임베딩하고, 질문 벡터와 문서 벡터의 내적(dot product)을 유사도 함수로 사용한다. 이 방식은 효율적인 유사도 계산과 문서 임베딩 사전 인덱싱이 가능하다는 장점이 있으며, FAISS(Facebook AI Similarity Search)와 같은 고속 유사도 검색 라이브러리를 이용해 수십억 개의 문서에서 빠르게 유사도를 비교할 수 있다.

이후 다양한 오픈소스 프레임워크(HuggingFace Transformers, LangChain, Haystack 등)에서 RAG 개념을 구현하였고, 벡터 DB와 결합하여 PDF 문서 질의응답, 기술 검색, 법률 문서 요약 등 실제 적용 사례가 넓어졌다.

11.3 구성요소

11.3.1 검색(Retrieval)과 인덱싱

11.3.1.1 개요

사용자의 질문에 대해 관련성이 높은 외부 문서를 검색하는 과정이다. 보통 벡터 데이터베이스(예: FAISS, Chroma, Pinecone 등)와 임베딩 모델(OpenAI, HuggingFace 등)을 활용한다. 대표적 검색 알고리즘은 다음과 같다:

11.3.1.2 Cosine Similarity 기반 Top-K 검색

가장 기본적인 벡터 유사도 기반 검색이다.

문서 임베딩과 질문 임베딩 간의 코사인 유사도를 계산하여, 유사도가 높은 문서 K개를 선택한다.

문제점 예시: 사용자가 "AI의 장점은?"이라고 질문했을 때, 동일한 내용을 반복하거나 매우 유사한 문서들만 선택되어 다양성이 부족할 수 있다. 예를 들어 "AI의 효율성 향상"이라는 주제를 동일하게 반복하는 문서들이 중복 선택됨.

개발 배경: 검색 품질은 높지만 다양성이 낮아 후속 생성 모델의 응답 품질에 부정적 영향을 줄 수 있다.

의사코드:

Input: query_embedding, document_embeddings, K

scores = [cosine_similarity(query_embedding, doc_emb) for doc_emb in document_embeddings]

ranked_docs = sort_by_score_descending(document_embeddings, scores)

Return top K ranked_docs

11.3.1.3 MMR (Maximal Marginal Relevance)

Cosine Top-K의 중복성과 다양성 부족 문제를 해결하기 위해 도입되었다.

문제점 예시: 사용자가 "자율주행의 한계는?"이라는 질문을 했을 때, 기존 cosine similarity만 사용하면 "센서 정확도 문제" 관련 문서만 연속해서 선택됨. 이는 다른 관점(법률적, 윤리적, 기술적)을 놓치게 된다.

해결 방식: 질문과의 관련성(relevance)을 유지하면서 이미 선택된 문서와의 중복성(diversity)을 줄이는 방식으로 문서를 선택한다.

λ 값(0 ≤ λ ≤ 1)을 통해 관련성과 중복성의 중요도를 조절할 수 있다. 예를 들어 λ=0.9 는 관련성을 중심으로, λ=0.3 은 다양성을 중심으로 검색한다.

λ=1.0: 기존 cosine similarity와 동일, 중복성 고려하지 않음.

λ=0.0: 오직 다양성만 고려함 (질문과의 관련성 무시).

λ=0.5: 균형 있게 고려함.

MMR 의사코드:

```
Input: query_embedding, candidate_docs, λ (0 <= λ <= 1), K
Initialize selected_docs = []
While len(selected_docs) < K:
    best_score = -inf
    best_doc = None
    For doc in candidate_docs:
        relevance = cosine_similarity(query_embedding, doc.embedding)
        diversity = max([cosine_similarity(doc.embedding, s.embedding) for s in selected_docs], default=0)
        score = λ * relevance - (1 - λ) * diversity
        If score > best_score:
            best_score = score
            best_doc = doc
    Add best_doc to selected_docs
    Remove best_doc from candidate_docs
Return selected_docs
```

11.3.1.4 Hybrid Search (BM25 + Embedding 기반)

전통적인 키워드 기반 BM25 검색과 벡터 임베딩 기반 검색을 결합한다.

BM25: "Best Matching 25"의 약어로, TF-IDF의 발전형이며, 질의어와 문서 간의 키워드 일치 정도를 평가하는 전통적인 정보 검색 알고리즘이다.

TF(term frequency): 단어가 문서 내 얼마나 자주 등장하는가

IDF(inverse document frequency): 단어가 전체 문서에서 얼마나 희귀한가

BM25 는 TF와 IDF 외에도 문서 길이 정규화를 반영하여 검색 순위를 계산한다.

문제점 예시: "암 치료 기술"이라는 질문에 대해 BM25 는 "암"이라는 키워드가 반복된 문서만을 상위에 배치할 수 있으며, 반대로 임베딩만 사용하면 관련성은 높지만 정작 키워드가 포함되지 않은 문서가 선택될 수도 있다.

해결 방식: BM25 로 우선 필터링한 후, 벡터 임베딩으로 재정렬함으로써 키워드 기반 정합성과 의미적 관련성을 동시에 확보한다.

의사코드:

Input: query, document_texts, document_embeddings, N, K

Step 1: top_n_docs = BM25(query, document_texts, top=N)

Step 2: embed_query = embed(query)

scores = [cosine_similarity(embed_query, embed(doc)) for doc in top_n_docs]

ranked_docs = sort_by_score_descending(top_n_docs, scores)

Return top K ranked_docs

11.3.1.5 Rerank 기법

리랭크(Rerank)는 초기 검색 결과의 정확도를 높이기 위해 수행하는 후처리 단계이다

BM25 는 단순히 단어 빈도와 일치 정도로 문서를 평가한다.

예) 문맥, 의미, 어순, 동의어 등을 이해하지 못함 → 엉뚱한 결과가 상위에 올 수 있음

BAAI의 bge-reranker 같은 모델은 쿼리와 문서 간 의미적 관련성을 평가한다.

예) 문맥적 일치, 의도 파악, 표현 다양성을 고려 → 더욱 적절한 문서를 상위에 위치시킴

다음은 1 차 검색 후 리랭크 처리를 하는 예시를 보여준다.

pip install rank_bm25 transformers torch langchain

전체코드:

```
from langchain.vectorstores import BM25Retriever
from langchain.schema import Document
from transformers import AutoTokenizer, AutoModelForSequenceClassification
import torch
import torch.nn.functional as F

# 문서 정의
documents = [
    Document(page_content="LangChain is a framework for developing LLM applications."),
    Document(page_content="BM25 is a ranking function used by search engines."),
    Document(page_content="Transformers library provides pretrained models."),
]

# 초기 BM25 기반 retriever 설정
retriever = BM25Retriever.from_documents(documents)
retriever.k = 5  # 상위 5 개 검색

# 리랭커 모델 로딩
tokenizer = AutoTokenizer.from_pretrained("BAAI/bge-reranker-v2-m3")
model = AutoModelForSequenceClassification.from_pretrained("BAAI/bge-reranker-v2-m3")
model.eval()  # 추론 모드
```

```
def rerank(query: str, docs: list[Document]) -> list[Document]:
    pairs = [(query, doc.page_content) for doc in docs]
    inputs = tokenizer(pairs, padding=True, truncation=True, return_tensors="pt")

    with torch.no_grad():
        scores = model(**inputs).logits.squeeze(-1)  # (batch_size,)
        probs = F.softmax(scores, dim=0)  # 확률화하여 soft score 사용 가능

    # 스코어와 문서 결합
    reranked = sorted(zip(probs.tolist(), docs), key=lambda x: x[0], reverse=True)
    return [doc for _, doc in reranked]

# 검색 및 리랭크
query = "How do LLM frameworks work?"
initial_results = retriever.get_relevant_documents(query)
final_results = rerank(query, initial_results)

# 결과 출력
for i, doc in enumerate(final_results, 1):
    print(f"[{i}] {doc.page_content}")
```

11.3.1.6 HyDE (Hypothetical Document Embeddings)

질문을 먼저 가상의 답변(hypothetical answer)으로 변환한 뒤, 그 답변의 임베딩을 기반으로 문서를 검색하는 기법이다.

"질문"만으로 벡터 검색하면 의미가 부족할 수 있다. LLM을 이용해 "이 질문에 대한 답변을 써봐"라고 한 뒤, 그 가상의 답변을 벡터로 변환하여 검색에 사용한다. 이를 통해, 답변의 문맥이 풍부해 지게 할 수 있다.

11.3.1.7 멀티 쿼리(Multi-Query)

하나의 질문에서 다양한 의미 표현의 쿼리들을 생성한 뒤, 각 쿼리에 대해 개별 검색을 수행하고, 결과를 통합하는 방식이다.

입력 질문: "지구 온난화의 원인은?"

LLM을 이용해 쿼리 다양화:

"기후 변화의 주요 원인은 무엇인가?"

"지구의 평균 기온이 상승하는 이유는?"

"온실가스와 지구 온난화의 관계는?"

각각의 쿼리를 벡터로 임베딩하여 **벡터 검색**

검색 결과를 **통합(RAG 문맥으로)** 하여 응답 생성

11.3.1.8 임베딩 벡터 인덱싱

고속 검색을 위해 인덱싱을 해야 한다. 예를 들어, FAISS(Facebook AI Similarity Search)는 대규모 벡터에 대해 고속 근사 최근접 이웃 검색을 제공한다. 이는 DPR과 같은 dense retriever가 생성한 벡터를 저장하고 검색하는 데 사용된다.

임베딩 벡터는 일반적으로 768 차원(BERT 기준) 이상의 고차원 공간에 위치하며, 의미적으로 유사한 텍스트는 이 공간에서 가까운 방향을 가진다. 이 벡터들은 FAISS와 같은 벡터 DB에 인덱싱되어 저장되며, 질의가 입력되면 동일한 방식으로 임베딩된 벡터를 사용하여 근접 벡터를 탐색한다. 이는 전통적인 공간 인덱싱과 매우 유사한 원리이다.

FAISS는 다양한 인덱싱 구조를 지원하며, 질의 시 빠르게 유사한 벡터를 찾기 위해 다음과 같은 방법을 사용한다:

1. Flat Index

모든 벡터를 그대로 저장하고 전체를 비교 (정확하나 느림)

IndexFlatL2, IndexFlatIP 등

2. IVF (Inverted File Index)

벡터들을 K개의 클러스터(centroid)로 분할하고, 각 클러스터에 속하는 벡터만 탐색

IndexIVFFlat, IndexIVFPQ 등

3. HNSW (Hierarchical Navigable Small World)

그래프 기반 탐색 구조로, 근접한 이웃만 따라가며 빠르게 탐색

IndexHNSWFlat

11.3.2 증강(Augmentation)

검색된 문서를 사용자의 원 질문과 결합하여 LLM 입력 프롬프트를 구성한다. 일반적으로 다음과 같은 프롬프트 템플릿을 사용한다:

프롬프트 템플릿 예시:

You are a helpful assistant. Use the following context to answer the question.

Context:

{retrieved_context}

Question:

{user_question}

Answer:

이때 문맥이 너무 길면 문서를 청킹(chunking)하여 일부만 선택해야 한다.

11.3.3 생성(Generation)

프롬프트를 LLM(OpenAI GPT-4, Claude, LLaMA 등)에 전달하여 응답을 생성한다. 생성 과정은 다음 단계를 따른다:

Augmented Prompt를 형성한다 (사용자 질문 + 검색된 문서).

해당 Prompt를 LLM의 입력으로 전달한다.

LLM은 사전 학습된 확률 기반 언어 모델링을 통해 다음 토큰을 반복 생성하며 응답을 생성한다.

필요 시, temperature, top-k, top-p 같은 샘플링 파라미터를 조정하여 생성 다양성과 품질을 조절한다.

생성 의사코드:

```
Input: prompt_text
Initialize output = ""
while not end_of_sequence:
    next_token = LLM.generate_next_token(prompt_text + output)
    output += next_token
Return output
```

11.3.4 임베딩 및 청킹(Embedding and Chunking)

11.3.4.1 개요

RAG 시스템에서 문서를 검색 가능하게 만들기 위해서는 먼저 문서를 처리 가능한 단위로 나눈 뒤, 각 단위를 벡터로 변환해야 한다. 이 과정을 "청킹(chunking)"과 "임베딩(embedding)"이라 한다.

문서를 청킹하는 이유는 다음과 같다:

- LLM이나 임베딩 모델은 입력 토큰 수에 제한이 있으며, 긴 문서를 한 번에 처리할 수 없다.

- 의미 있는 문단 또는 문장 단위로 분할하면 정보가 손실되지 않고 검색 품질이 높아진다.
- 검색 단계에서 세부적인 단위(청크)로 비교함으로써, 질문과 가장 관련 있는 문맥을 찾아낼 수 있다.
- 증강 생성 시 과도한 정보가 입력되면 노이즈가 발생할 수 있으므로, 적절한 단위로 잘라 사용하는 것이 성능 향상에 기여한다.

11.3.4.2 임베딩 (Embedding)

각 청크는 임베딩 모델을 통해 고차원 벡터로 변환된다.

이 벡터는 해당 텍스트의 의미론적 특성을 수치화한 것으로, 벡터 간 거리를 통해 의미 유사도를 계산할 수 있다.

벡터는 이후 FAISS와 같은 검색 인덱스에 저장되며, 질의 입력 시 같은 임베딩 모델로 생성된 벡터와 비교된다.

임베딩 및 청킹 과정 요약 의사코드:

```
# 청킹
chunks = split_document(doc, chunk_size=500, overlap=100)


# 임베딩
for chunk in chunks:
    v = embed_model(chunk)
    vector_store.append(v)
```

임베딩 품질은 검색 정확도에 직접 영향을 주며, 청킹 전략은 정보 손실 없이 적절한 문맥을 유지하는 데 영향을 준다.

11.3.4.3 청킹 (Chunking)

전체 문서는 일반적으로 너무 길어서 직접 임베딩하거나 LLM에 전달할 수 없다. 따라서 문서를 일정 길이(예: 300~1000 자 단위)로 분할한다.

오버랩(overlap)을 적용하여 문맥이 단절되지 않도록 조절한다 (예: 500 자 청크, 100 자 중첩).

LangChain에서는 RecursiveCharacterTextSplitter와 같은 클래스를 사용해 단락, 문장, 단어 기준으로 유연하게 청킹 가능하다.

일반적으로 청킹된 텍스트는 임베딩 모델의 최대 토큰 제한 이내여야 하며, 각 임베딩 모델마다 다르지만 보통 다음과 같다.

임베딩 차원 (예: 512, 768, 1024)

- 텍스트가 임베딩된 후 생성되는 고정된 벡터의 차원 수이다.
- 예: BERT-base는 768 차원

입력 텍스트 최대 길이

- 보통 512 토큰 (WordPiece, SentencePiece 기준)
- 이 제한을 초과하면 텍스트는 잘리거나 오류 발생

11.4 증강 프롬프트 체인 내부 구조

LangChain 등 프레임워크에서는 다음과 같은 체인 구조를 사용한다. 이를 통해 문서 → 질문 응답까지 자동화된 흐름을 구현한다.

전체 체인 구성 요소:

Document Loader → Splitter: PDF 또는 텍스트 문서를 청킹(chunking)

Embedder: 각 청크에 대해 임베딩 생성

Retriever: 질문 임베딩과 유사한 문서 검색 (Top-K 또는 MMR)

Prompt Formatter: 검색된 문서와 질문을 프롬프트 템플릿에 삽입

LLM Generator: 프롬프트를 기반으로 응답 생성

LangChain 체인 의사코드:

Input: user_query

Step 1: embedded_query = embed(user_query)

Step 2: retrieved_chunks = retriever.retrieve(embedded_query)

Step 3: formatted_prompt = prompt_template.format(context=retrieved_chunks, question=user_query)

Step 4: answer = llm.generate(formatted_prompt)

Return answer

LangChain에서는 ConversationalRetrievalChain 또는 RetrievalQA 클래스를 통해 이 과정을 묶어 사용할 수 있다.

11.5 LangChain 기반 RAG 예시 (PDF 문서 기반)

다음은 LangChain과 OpenAI API를 이용해 PDF를 처리하고 질문에 답하는 전체 예시이다.

PDF 읽기 및 청킹

```
from langchain.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter

loader = PyPDFLoader("my_doc.pdf")
docs = loader.load()
splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=100)
chunks = splitter.split_documents(docs)
```

임베딩 및 벡터 저장

```
from langchain.vectorstores import FAISS
from langchain.embeddings import OpenAIEmbeddings

embedding_model = OpenAIEmbeddings()
vectorstore = FAISS.from_documents(chunks, embedding_model)
```

질문-응답 체인 구성

```
from langchain.chains import RetrievalQA
```

```
from langchain.chat_models import ChatOpenAI


retriever = vectorstore.as_retriever(search_type="mmr")
qa_chain = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(),
    chain_type="stuff",
    retriever=retriever
)
```

질문에 대한 응답 생성

```
query = "이 문서에서 저자의 주요 주장은 무엇인가요?"
result = qa_chain.run(query)
print(result)
```

11.6 결론

RAG는 LLM의 한계를 보완하고, 외부 지식과 실시간 정보 반영을 통해 신뢰성 높은 결과 생성을 가능하게 한다. 특히 PDF 등 구조화되지 않은 문서에 대해 검색 기반 QA 시스템을 구현할 때 유용하다. LangChain은 이러한 체인을 모듈화하여 쉽게 구성할 수 있는 도구를 제공한다.

12. RAG 기반 LLM 모델 학습 데이터 생성

12.1 RAG 학습 데이터 생성 방법

RAG 기반 LLM 학습을 위해 필요한 데이터는 질의(Query), 문맥(Context), 정답(Answer)으로 구성된 쌍이며, 다음과 같은 방식으로 생성할 수 있다:

12.1.1 공개 데이터셋 활용

Natural Questions, TriviaQA, HotpotQA 등 공개 QA 데이터셋은 질의-문맥-정답 구조를 갖추고 있어 RAG 학습에 적합하다.

Wikipedia 기반 Knowledge-intensive QA 태스크에서 자주 활용된다.

12.1.2 웹 크롤링 및 문서 파싱

도메인 특화 정보가 필요한 경우, 관련 웹사이트 또는 문서를 크롤링하여 데이터로 변환

HTML, PDF 문서를 정제하여 벡터 인덱싱이 가능한 문단 단위로 분할하고, 질문을 수작업 또는 LLM을 통해 생성 가능

12.1.3 합성 질의 응답 생성 (Synthetic QA Pairing)

기존 문서를 기반으로 LLM을 통해 다양한 질의를 생성하고, 해당 문서 일부를 Context로 활용하여 정답을 구성

HyDE(Hypothetical Document Embeddings) 방식으로 생성된 질문-문서 쌍을 RAG 학습에 사용

12.1.4 Annotation 기반 수동 작성

정밀한 학습을 위해 전문가 또는 Annotator가 직접 질의-문맥-정답 쌍을 작성

도메인 응용 시스템 구축 시 활용도 높음

12.1.5 문서-정답 기반 역 질의 생성

문서와 정답만 존재할 경우, 역으로 질의를 생성하는 방식 (LLM 기반 역질문 생성)

QA generation 모델이나 Prompt Tuning 활용 가능

12.2 Retrieval-Augmented Fine-Tuning (RAFT)

RAFT는 기존 RAG(Retrieval-Augmented Generation)의 한계를 보완하는 훈련 방식으로, 검색된 문서를 단순히 LLM 입력에 포함하는 것이 아니라, **정답을 포함한 예시로서 supervised fine-tuning**을 수행한다. 즉, 검색 결과를 바탕으로 모델이 실제로 정답을 학습한다.

주요 특징:

질문과 검색 문서를 함께 모델에 입력하고 정답 출력을 학습

검색 결과와 정답 간 alignment 학습

Retriever가 불완전할 경우를 감안한 fallback 학습 효과

예시 흐름:

```
for example in dataset:
    question = example["query"]
    gold_answer = example["answer"]
    context_docs = retriever.search(question)  # RAG 데이터를 학습에 사용

    input_text = format_prompt(question, context_docs)
    output_text = gold_answer

    loss = model(input_text, labels=output_text) # LLM 모델 학습
    loss.backward()
    optimizer.step()
```

RAFT는 정답이 명확히 주어지는 QA 태스크나 지식 기반 응답 생성에서 정확도를 크게 향상시킬 수 있으며, RAG의 한계인 generation의 비일관성과 정보 왜곡을 줄이는 데 효과적이다.

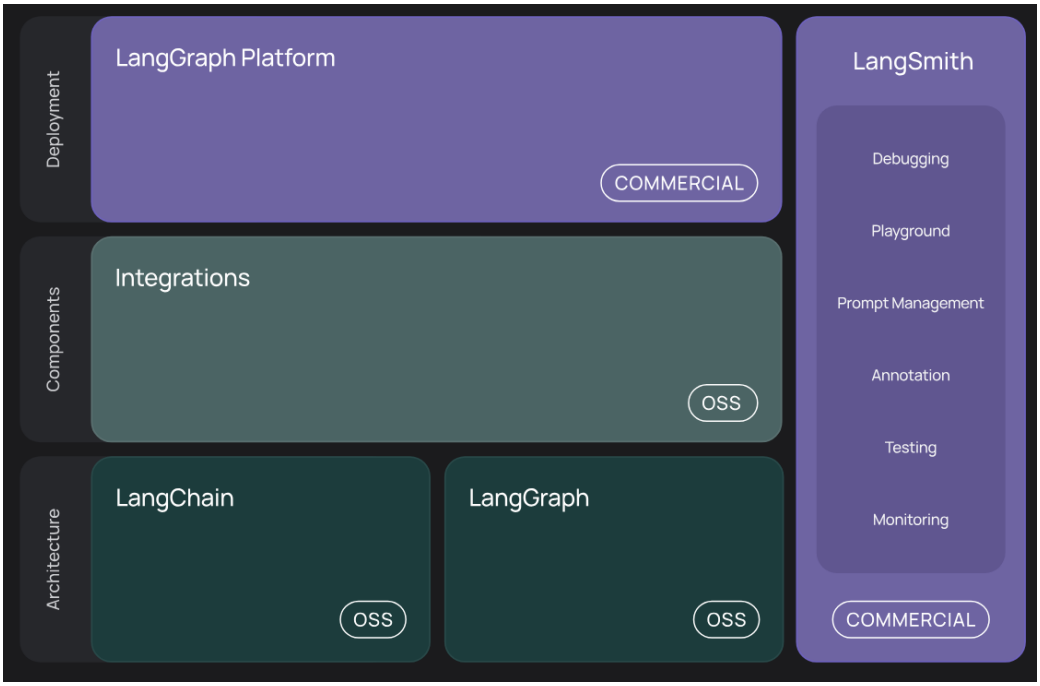
12.3 결론

sLLM은 경량화된 언어모델로 다양한 환경에 적합하며, 이를 효율적으로 미세조정하기 위해 FFT보다 PEFT 기법이 주로 사용된다. 대규모 학습 시에는 DDP, FSDP, DeepSpeed와 같은 분산 학습 프레임워크를 활용하여 자원을 효율적으로 사용해야 한다. 또한, RAG 기반 응용에서는 단순 retrieval 기반 응답 대신 RAFT를 통해 fine-tuning을 수행하고, 질의-문맥-정답 구조의 데이터를 다양한 방식으로 생성함으로써 모델의 정확성과 일반화를 동시에 향상시킬 수 있다.

13. LangChain

13.1 랭체인 개념

LangChain은 대규모 언어 모델(LLM)과 다양한 도구 및 데이터 소스를 통합하여 복잡한 작업을 수행하도록 설계된 프레임워크이다. 랭체인은 LLM의 강력한 자연어 처리 능력을 활용하며, 사용자 정의 가능한 체인과 에이전트를 통해 문제를 해결한다.



Introduction LangChain

13.2 설치 및 예시 코드

터미널에서 다음을 설치한다.

pip install langchain openai faiss-cpu tiktoken pypdf

예제 코드:

```
from langchain.chat_models import ChatOpenAI

# OpenAI LLM 인스턴스 생성
llm = ChatOpenAI(model="gpt-4", temperature=0.7, openai_api_key="your-api-key")
```

```
query = "What is LangChain?"
response = llm.run(query)
print(response)
```

13.3 구조

LangChain의 구조는 다음과 같은 주요 구성 요소로 나뉜다:

- **LLM**: 언어 모델 자체.
- **Prompt Templates**: 입력 데이터를 포맷팅하는 템플릿.
- **Chains**: 여러 구성 요소를 순차적으로 연결하는 작업 흐름.
- **Agents**: 동적으로 도구를 선택하고 실행하는 역할.

13.4 프롬프트 템플릿

프롬프트 템플릿은 LangChain에서 중요한 구성 요소로, 입력 데이터를 구조화하여 LLM의 성능을 극대화한다.

예제 코드:

```
from langchain.prompts import PromptTemplate

# 프롬프트 템플릿 생성
prompt_template = PromptTemplate(
    template="Provide a summary for the following topic: {topic}",
    input_variables=["topic"]
)

query = prompt_template.format(topic="LangChain's architecture")
print(query)
```

13.5 LCEL (LangChain Expression Language)

LCEL은 LangChain의 구성 요소를 연결하는 간단하고 표현적인 방식이다. 파이프라인을 '|' 연산자를 사용해 연결하여 직관적으로 작업 흐름을 정의할 수 있다.

LCEL 예제: '|' 연산자를 활용한 간단한 체인

```
from langchain.prompts import PromptTemplate
from langchain.chat_models import ChatOpenAI
from langchain.output_parsers import StructuredOutputParser, ResponseSchema

# OpenAI LLM
llm = ChatOpenAI(model="gpt-4", temperature=0.7, openai_api_key="your-api-key")

# Prompt Template
prompt_template = PromptTemplate(
    template="{question}",
    input_variables=["question"]
)

# Output Parser
response_schema = ResponseSchema(name="answer", description="The final answer to the question.")
parser = StructuredOutputParser.from_response_schemas([response_schema])

# LCEL Pipeline
query = "What are the key components of LangChain?"
result = prompt_template | llm | parser

# Execute
output = result.invoke({"question": query})
print(output)
```

13.6 에이전트 방식

LangChain은 다양한 에이전트 방식을 지원하며, 다음과 같은 주요 에이전트 유형이 있다:

13.6.1 Zero-shot ReAct

LLM이 단일 프롬프트를 통해 직접 작업을 수행한다.

예제 코드:

```
from langchain.agents import initialize_agent
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(model="gpt-4", openai_api_key="your-api-key")
agent = initialize_agent(llm, tools=[])  # 기본 에이전트 생성
query = "What is the weather today?"
response = agent.run(query)
print(response)
```

13.6.2 Conversational ReAct

대화형 에이전트로, 이전 대화 내용을 기반으로 적응한다.

예제 코드:

```
from langchain.agents import ConversationalAgent
from langchain.memory import ConversationBufferMemory
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(model="gpt-4", openai_api_key="your-api-key")
conversation_memory = ConversationBufferMemory()
conversation_memory.save_context({"input": "What is LangChain?"}, {"output": "LangChain
is a framework that simplifies LLM tasks."})
agent = ConversationalAgent(llm=llm, memory=conversation_memory)

query = "Can you explain more about LangChain's components?"
response = agent.run(query)
print(response)
```

13.6.3 ReAct Docstore

문서 저장소에서 정보를 검색한 후 작업을 수행한다.

예제 코드:

```
from langchain.agents import DocstoreAgent
```

```
from langchain.document_loaders import LocalDocumentLoader
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(model="gpt-4", openai_api_key="your-api-key")
document_loader = LocalDocumentLoader(directory="./docs")
agent = DocstoreAgent(llm=llm, document_loader=document_loader)

query = "Summarize the content of the document."
response = agent.run(query)
print(response)
```

13.6.4 Self-ask with Search

질문을 세분화하여 자체 검색 및 응답을 생성한다. Self-Ask with Search는 Google DeepMind 논문(Lewis et al., 2022)에서 제안된 기법, 복잡한 질문을 단순한 질문들로 나눈 뒤 외부 지식(예: 검색엔진)에서 정보를 검색해 조합하여 답한다.

LangChain은 이 패턴을 agent tool + LLM + 검색 retriever 조합으로 구현함.

예제 코드:

```
from langchain.agents import SelfAskWithSearchAgent
from langchain.tools import TavilySearchAPI
from langchain.chat_models import ChatOpenAI

llm = ChatOpenAI(model="gpt-4", openai_api_key="your-api-key")
search_tool = TavilySearchAPI(api_key="your-tavily-api-key")
agent = SelfAskWithSearchAgent(llm=llm, search_tool=search_tool)

query = "What are the latest advancements in AI?"
response = agent.run(query)
print(response)
```

13.7 에이전트 동작 방식

13.7.1 개요

chat-conversational-react-description은 LangChain에서 제공하는 대표적인 에이전트 타입이다. 이 에이전트는 대화 히스토리를 기억하면서, 스스로 추론하고 도구를 선택하거나 직접 최종 답변을 생성하는 구조를 가지고 있다. LangChain의 표준화된 ReAct(Reasoning + Acting) 패턴을 기반으로 작동하며, 자연스러운 대화 흐름을 지원하는 데 최적화되어 있다.

13.7.2 동작 흐름

에이전트는 다음과 같은 순서로 작동한다.

1. 사용자의 질문(input)을 수신한다.
2. 대화 히스토리(chat_history)와 현재 질문을 통합하여 프롬프트를 구성한다.
3. LLM을 호출하여 다음 행동을 결정한다. 응답 형태는 다음 중 하나이다:
   A. 특정 도구를 사용할 것 (Action: tool name, Action Input: input)
   B. 직접 최종 답변을 할 것 (Final Answer: answer text)
4. 응답을 JSON 형태로 파싱한다.
5. Action이 도구 사용이면 해당 도구를 실행하고, 결과를 다시 LLM에 전달하여 다음 행동을 결정한다.
6. Action이 Final Answer이면, 그 내용을 사용자에게 반환하고 대화를 종료한다.
7. 이번 질문과 답변을 대화 히스토리에 기록하여 다음 질문에 반영한다.

에이전트는 Final Answer가 나오기 전까지 필요하면 여러 번 도구를 사용할 수 있으며, 그 과정에서 지속적으로 reasoning을 수행한다.

13.7.3 LangChain 내부 프롬프트 구성

LangChain은 chat-conversational-react-description 에이전트를 생성할 때, 내부적으로 특수한 프롬프트 템플릿을 자동으로 생성한다. 이 프롬프트는 다음과 같은 요소로 구성된다.

- 시스템 메세지: "You are a helpful AI assistant."

- 대화 히스토리(chat_history): 과거 사용자와의 대화 내용.
- 현재 질문(input): 사용자가 새로 입력한 질문.
- 도구 이름 목록(tool_names): 사용 가능한 도구 목록.

또한 답변 포맷을 강제한다. 포맷은 다음과 같다.

Thought: (이 상황에서 무엇을 해야 할지 고민)
Action: (사용할 도구 이름)
Action Input: (도구에 입력할 값)

또는

Final Answer: (최종 답변)

이 포맷을 따르도록 LLM에게 명시적으로 지시하여, 에이전트의 일관된 동작을 유지한다.

13.7.4 프롬프트 생성 위치

LangChain 코드 기준으로 프롬프트는 langchain/agents/conversational_chat/base.py 파일의 ConversationalChatAgent.create_prompt() 함수에서 생성된다.

- https://github.com/langchain-ai/langchain/blob/master/libs/langchain/langchain/agents/conversational_chat/prompt.py

이 함수는 시스템 메세지, 대화 히스토리, 현재 질문, 도구 목록을 종합하여 최종 프롬프트를 구성하고 LLM에 전달한다.

13.7.5 결론

chat-conversational-react-description은 LangChain 에이전트 설계 중 가장 범용성과 대화 자연스러움을 겸비한 구조이다. ReAct 패턴을 기반으로 행동(도구 사용)과 사고(Thought)를 반복하며, 최종적으로 Final Answer를 생성한다. LangChain은 이 에이전트를 위해 내부적으로 체계화된 프롬프트 템플릿을 자동 구성하여, 일관된 reasoning과 action selection 과정을 보장한다.

13.8 기본 도구

LangChain은 검색, 수학 계산, 파이썬 코드 실행 등 다양한 내장 도구를 제공한다. 예를 들어, 검색 도구는 다음과 같이 사용할 수 있다:

예제 코드:

```
from langchain.tools import TavilySearchAPI

# Tavily Search Tool
search = TavilySearchAPI(api_key="your-tavily-api-key")

query = "Latest advancements in AI"
results = search.run(query)
print(results)
```

13.9 벡터DB 및 검색

LangChain은 대규모 텍스트 데이터의 인덱싱과 검색을 위해 벡터 데이터베이스와 통합된다. 이를 통해 사용자는 임베딩된 벡터를 효율적으로 검색할 수 있다.

예제 코드:

```
from langchain.vectorstores import FAISS

# 벡터 스토어 생성 및 검색
texts = ["LangChain is powerful.", "VectorDB is efficient."]
vectorstore = FAISS.from_texts(texts)
query = "What is LangChain?"
results = vectorstore.similarity_search(query)
print(results)
```

13.9.1 최대 마진 관련성(MMR, Maximal Marginal Relevance)

MMR 알고리즘은 검색 결과의 다양성을 높이면서도 관련성을 유지하도록 설계된 알고리즘이다.

- **주요 특징:**
  - 결과 집합의 다양성을 보장하기 위해 기존에 선택된 결과와의 중복을 최소화한다.
  - 사용자가 요청한 질의에 대한 관련성을 최대화한다.
- **동작 원리:**
  - 검색된 문서 집합에서 가장 관련성이 높은 문서를 선택한다.
  - 이후 반복적으로 관련성 점수와 다양성을 고려해 새로운 문서를 선택한다.
  - λ₩lambda 매개변수는 관련성과 다양성 간의 균형을 조절한다.
- **사용 사례:**

```
docsearch.as_retriever(
    search_type="mmr",
    search_kwargs={'k': 6, 'lambda_mult': 0.25}
)
```

이 설정은 질의와 관련성 높은 문서 6 개를 반환하되, λ 값을 통해 다양성을 강조한다.

13.9.2 유사도 점수 임계치(Similarity Score Threshold)

이 알고리즘은 문서와 질의 간의 유사도 점수가 특정 임계값을 초과하는 경우에만 문서를 검색한다.

- **주요 특징:**
  - 사용자 정의 임계값을 통해 결과의 품질을 제어한다.
  - 유사도가 낮은 문서를 필터링하여 검색 효율성을 높인다.
- **동작 원리:**
  - 벡터 공간에서 질의와 각 문서 간의 유사도 점수를 계산한다.
  - 설정된 임계값 이상인 문서만 선택한다.
- **사용 사례:**

```
docsearch.as_retriever(
    search_type="similarity_score_threshold",
```

```
    search_kwargs={'score_threshold': 0.8}
)
```

이 설정은 유사도 점수가 0.8 이상인 문서만 반환한다.

13.9.3 단일 문서 검색(Single Document Retrieval)

가장 유사한 단일 문서를 검색하는 알고리즘이다.

- **주요 특징:**
  - 빠르고 간단한 검색이 가능하다.
  - 단일 결과만 필요한 간단한 응용에 적합하다.
- **동작 원리:**
  - 전체 데이터셋에서 질의와 가장 유사한 문서를 계산한다.
  - 최고 점수의 문서 하나만 반환한다.
- **사용 사례:**

```
docsearch.as_retriever(search_kwargs={'k': 1})
```

13.9.4 필터 기반 검색(Filter-based Retrieval)

사용자가 지정한 필터 조건에 맞는 문서만 검색한다.

- **주요 특징:**
  - 특정 조건(예: 문서 속성)을 만족하는 결과만 반환한다.
  - 대규모 데이터셋에서 특정 그룹을 대상으로 검색할 수 있다.
- **동작 원리:**
  - 검색 요청 시 사용자 정의 필터를 적용한다.
  - 필터 조건에 부합하는 문서만 검색 대상에 포함한다.
- **사용 사례:**

```
docsearch.as_retriever(
    search_kwargs={'filter': {'paper_title': 'GPT-4 Technical Report'}}
)
```

13.9.5 RAG 기반 Retrieval QA 체인 생성

RAG(Retrieval-Augmented Generation)는 검색 기반 QA 시스템에서 자주 활용된다. 아래는 RAG QA 체인을 생성하는 예제이다.

- **과정:**
    1. **Retrieval:** 벡터 DB 에서 관련 문서를 검색한다.
    2. **QA Chain:** 검색된 문서를 LLM(Language Model)에 전달하여 질의에 대한 응답을 생성한다.
- **코드 구현:**

```
def create_retrieval_qa(vectordb):
    retriever = vectordb.as_retriever(
        search_type="mmr",
        search_kwargs={'k': 6, 'lambda_mult': 0.25}
    )
    qa_chain = RetrievalQA.from_chain_type(
        llm=llm_model,
        retriever=retriever,
        chain_type="stuff"
    )
```

- return qa_chain

위 코드는 MMR 알고리즘을 사용하여 검색된 문서를 기반으로 QA 체인을 생성한다.

- **구성 요소:**
    - search_type="mmr": MMR 알고리즘을 사용한다.
    - k=6: 검색된 문서 중 상위 6 개 문서를 반환한다.
    - lambda_mult=0.25: 관련성과 다양성의 균형을 조정한다.

13.9.6 결론

벡터 DB의 다양한 검색 알고리즘은 각각의 특성과 동작 방식에 따라 다양한 응용 분야에 적합하게 사용할 수 있다. MMR은 다양성과 관련성을 모두 중시할 때 유용하며, 유사도 점수 임계치는 품질이 보장된 결과를 반환하고, 필터 기반 검색은 특정 조건에 맞는 문서를 효과적으로 검색할 수 있다.

13.10 고려사항

LangChain을 활용한 RAG 시스템 구축 시 OpenAI LLM의 토큰 제한에 관련된 예외는 자주 발생하는 문제이다. 특히, 벡터 데이터베이스에서 검색된 문서가 너무 길거나, 대화 히스토리가 누적되어 OpenAI API의 최대 입력 토큰 수를 초과할 경우 openai.BadRequestError와 같은 오류가 발생한다.

13.10.1 모든 문서를 LLM에 일괄 전달할 때 발생하는 토큰 초과

chain_type="stuff" 방식은 검색된 모든 문서를 LLM에 그대로 전달하므로 토큰 수가 쉽게 초과된다.

해결 방안:

chain_type을 map_reduce 또는 refine으로 변경하여 문서를 분할하여 처리한다.

문서 청크 크기(chunk_size)를 줄이고 중첩(chunk_overlap)을 최소화한다.

13.10.2 대화 기록 누적으로 인한 토큰 증가

ConversationBufferMemory는 모든 대화 내역을 저장하므로 대화가 길어질수록 토큰 수가 증가한다.

해결 방안:

ConversationSummaryMemory를 사용하여 이전 대화를 요약한다.

max_token_limit을 설정하여 기억할 최대 토큰 수를 제한한다.

13.10.3 검색된 문서 수가 많아 전체 프롬프트 길이를 초과함

FAISS 등에서 검색된 청크가 많으면 LLM에 전달되는 전체 문서 길이가 길어져 오류가 발생한다.

해결 방안:

search_kwargs={"k": n}에서 n을 줄이거나, lambda_mult 등을 조정하여 중복을 줄인다.

ContextualCompressionRetriever 등을 활용해 검색된 문서를 요약해서 전달한다.

13.10.4 모델의 최대 토큰 수를 고려하지 않은 프롬프트 구성

GPT-3.5-turbo는 4,097 tokens, GPT-4 는 8,192~32,768 tokens의 입력 제한이 있다.

해결 방안:

LLM 생성 시 max_tokens를 명시적으로 지정한다.

입력 문서와 질문 토큰 길이를 사전에 계산하여 제한한다.

13.10.5 토큰 수 계산이 어려워 사전 조정이 어렵다

LLM 입력과 응답 토큰 수를 정확히 계산하기 어렵다는 피드백이 존재한다.

해결 방안:

tiktoken 라이브러리를 사용하여 프롬프트 토큰 수를 계산한다.

필요 시 토큰 계산 후 일정 기준 이상일 경우 청크를 필터링한다.

13.10.6 기타

LangChain을 통한 RAG 시스템 구현 시, OpenAI API의 토큰 제한을 고려한 설계는 필수적이다. 위 문제들은 토큰 길이 초과로 인한 예외 발생의 주된 원인이며, 적절한 체인 구성 및 전처리를 통해 이를 예방할 수 있다. 특히 map_reduce, memory 요약, 토큰 길이 측정, retriever 필터링과 같은 기능은 토큰 초과 문제의 핵심적인 해결책이 된다.

향후 모델 확장성이나 고비용 API 사용을 고려할 때, 이러한 문제들을 사전에 감지하고 구조적으로 관리하는 것이 RAG 아키텍처의 신뢰성과 유지보수성을 높이는 데 도움이 된다.

13.11 랭체인 이외 에이전트 도구

다음 표에 나열된 도구들은 각각 다른 방식으로 AI 에이전트를 개발하고 관리하는 데 도움을 줄 수 있으며, 각 도구의 특성에 따라 사용 사례가 다를 수 있다.

도구 이름	설명	링크
AutoGPT	텍스트 기반 입력을 통해 자동으로 여러 작업을 수행하는 AI 에이전트. GPT	AutoGPT

	모델을 활용하여 다양한 작업을 자동화할 수 있도록 설계됨.	GitHub
Pydantic	AI 에이전트 및 모델 구축을 위한 데이터 검증 및 설정 도구. Pydantic을 사용하여 AI 모델의 입력과 출력을 구조적으로 관리하고 검증할 수 있다.	PydanticAI
LlamaIndex	Llama 모델을 활용하여 효율적인 데이터 검색 및 에이전트 기능을 구현하는 도구. 특히 대규모 텍스트 데이터 처리 및 검색에 강점.	LlamaIndex GitHub
CrewAI	협업을 중심으로 AI 에이전트를 관리하는 도구. 여러 AI 에이전트를 연결하여 복잡한 문제를 해결하는데 유용.	CrewAI Website
LangGraph	그래프 데이터베이스 기반의 AI 에이전트 관리 도구로, 복잡한 데이터 관계를 효과적으로 처리하고 탐색할 수 있도록 돕는다.	LangGraph GitHub

13.12 결론

LangChain은 LLM을 활용한 강력한 프레임워크로, 복잡한 작업 흐름을 단순화하고 효율적으로 처리할 수 있는 다양한 기능을 제공한다. LCEL과 에이전트 방식은 사용자의 필요에 따라 유연하게 구성 및 실행할 수 있는 점에서 특히 유용하다.

13.13 레퍼런스

1. **LangChain Docs**: 체인에서 메모리, 에이전트에 이르기까지 모든 모듈에 대한 코드 예제가있는 매우 깨끗한 문서.
2. **LangChain Python GitHub**: 코드 아래에서 무슨 일이 일어나고 있는지 확인하거나 최신 업데이트를 추적하려는 경우.
3. **Hugging Face Transformers**: 임베딩 모델, 로컬 LLM 또는 Transformers 파이프라인으로의 전환에 대한 심층적인 이해.
4. **Facebook AI의 FAISS 가이드**:- 벡터 스토어를 조정하거나 대규모로 배포하려는 경우.
5. **LangChain + RAG 전체 스택 튜토리얼**: 팀에서 직접 제공하는 연습 코드.
6. **생산 분야의 RAG — Pinecone 블로그**: 과제 및 모범 사례를 포함하여 생산 컨텍스트에서 RAG 아키텍처를 설명.
7. **해리슨 체이스(LangChain)**: 지속적인 업데이트, 팁 및 향후 기능에 대한 엿보기.
8. **Retrieval-Augmented Generation 설명**:- RAG가 더 넓은 LLM 생태계에 어떻게 적합한지에 대한 입문서.

14. ollama

14.1 개요

Ollama는 로컬에서 대규모 언어 모델(LLM)을 실행하고 활용할 수 있도록 도와주는 경량화된 런타임이다. GPT, LLaMA, Mistral 등의 오픈소스 모델을 로컬 환경에서 직접 불러와 프롬프트 응답이나 RAG 검색 등에 사용할 수 있도록 설계되었다. Docker 없이도 작동하며, LangChain과의 통합도 매우 용이하다.

14.2 Windows에서 Ollama 설치 방법

https://ollama.com 에 접속한다.

운영체제(Windows)를 선택하고 설치 프로그램(.exe)을 다운로드한다.

설치 후, 커맨드 프롬프트(CMD)나 PowerShell을 열어 ollama 명령어가 인식되는지 확인한다.

14.3 주요 터미널 명령어

```
# 모델 다운로드 및 실행

ollama run llama3


# 설치된 모델 목록 확인

ollama list


# 실행 중인 모델 종료

ollama stop


# 로컬에 있는 모델 삭제

ollama remove llama3
```

모델은 처음 실행할 때 자동으로 다운로드되며, 이후에는 로컬에 캐시되어 반복 다운로드가 필요 없다.

14.4 LangChain에서 Ollama 사용하기

LangChain에서 Ollama를 호출하기 위해 다음과 같은 환경이 구성되어야 한다:

필수 패키지 설치

pip install langchain langchain-community

Python 예제 코드

```
from langchain.llms import Ollama

llm = Ollama(model="llama3")
prompt = "Ollama에서 실행된 LLaMA3 모델은 어떤 특징이 있나요?"
response = llm.invoke(prompt)
print(response)
```

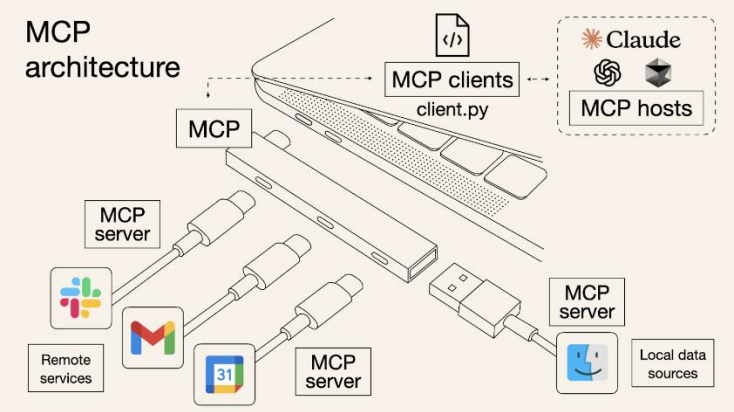
이 코드는 로컬에서 실행 중인 llama3 모델에게 프롬프트를 전달하고, 텍스트 응답을 출력한다. Ollama는 기본적으로 localhost:11434 포트를 통해 REST API 형태로 작동하며, LangChain은 이 API를 통해 프롬프트 요청을 수행한다.

Ollama는 GPU 없이도 CPU 환경에서 실행 가능하다. 단, 성능은 GPU 대비 크게 낮으며, 응답 속도나 메모리 사용량에 영향을 받을 수 있다. GPU가 없어도 llama2, mistral 등 경량 모델은 실행되지만, 7B 이상 모델은 로딩 및 추론 속도가 느려질 수 있다.

15. MCP(Model Context Protocol)와 AI 에이전트 개발

15.1 개요 및 개념

Model Context Protocol(MCP)은 다양한 애플리케이션이 LLM에 컨텍스트를 제공하는 방식을 표준화하기 위해 Claude 개발사인 Anthropic이 제안한 개방형 프로토콜이다. MCP는 USB-C 포트처럼 다양한 도구와 데이터를 AI 모델에 통합된 방식으로 연결하며, AI 에이전트가 도구를 인식하고 사용하는 데 필요한 표준 인터페이스를 제공한다.

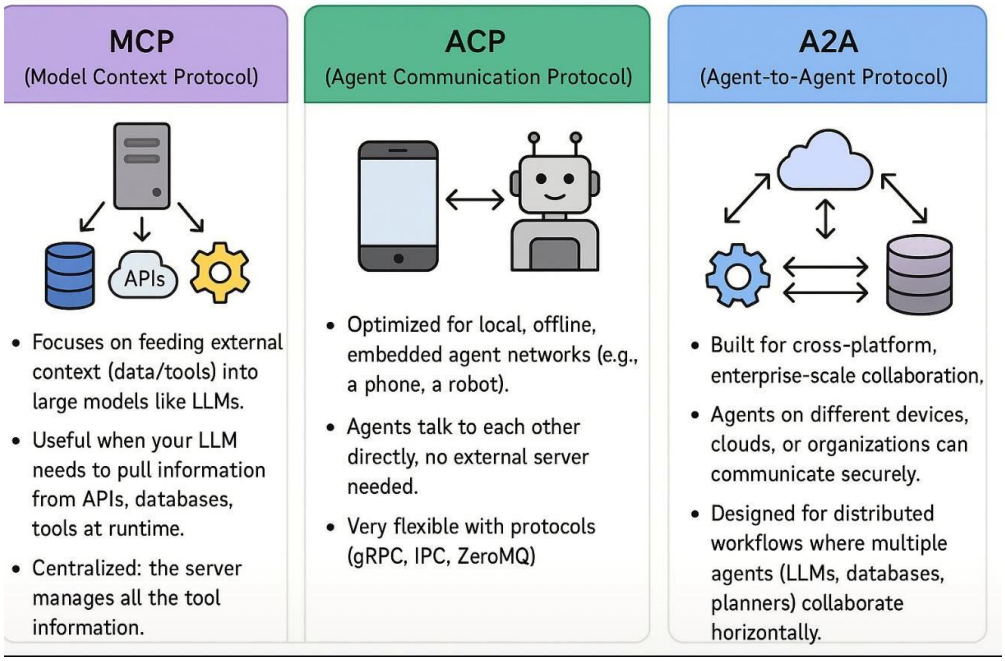


LLM 기반 애플리케이션에서의 MCP 도입은 단순한 질의응답 모델을 넘어서, 외부 시스템과 상호작용할 수 있는 진정한 "도우미(agent)" 역할로 확장되는 기반을 제공한다. 이를 통해 유연한 통합, 데이터 보호, 공급업체 간 전환성이 보장된다. 예를 들어, LLM은 MCP를 통해 로컬 파일을 열어 읽고, 외부 API를 통해 정보를 검색하고, 복잡한 사용자 요청을 처리할 수 있다.

15.2 MCP, ACP, A2A

15.2.1 서론

인공지능 에이전트 시스템의 발달은 더 이상 단일 모델의 능력에 의존하지 않는다. 복수의 도구와 데이터 소스, 서로 다른 플랫폼에 위치한 여러 에이전트가 유기적으로 협업하는 구조가 새로운 패러다임으로 부상하고 있다. 이와 같은 환경에서 효율적인 통신을 보장하기 위해 다양한 에이전트 프로토콜이 등장하게 되었으며, 그 중 대표적인 것이 Model Context Protocol(MCP), Agent Communication Protocol(ACP), 그리고 Agent-to-Agent Protocol(A2A)이다. 이들 프로토콜은 기능적으로는 겹치지만, 설계 목적과 적용 범위, 기술적 구현 방식에 있어 차이를 지닌다.



MCP, ACP, A2A 개념도

15.2.2 MCP (Model Context Protocol)

Model Context Protocol(MCP)은 2023 년 중반, 클로드(Claude) 개발사로 알려진 Anthropic에서 제안하였다. MCP는 LLM(Large Language Model) 중심의 아키텍처에서 외부 데이터 소스 또는 도구를 연결하는 데 최적화된 통신 프레임워크로, 주로 툴 호출 및 컨텍스트 주입을 위해 설계되었다. 본질적으로 MCP는 사용자가 입력한 질의에 대해 LLM이 적절한 응답을 생성할 수 있도록, 외부 데이터베이스나 API, 파일 시스템 등에서 실시간으로 정보를 가져오는 메커니즘을 제공한다.

기술 구조상 MCP는 중앙 집중형 아키텍처를 따른다. 즉, 모든 도구 호출과 자원 접근은 특정 MCP 서버를 거쳐 수행되며, 해당 서버는 도구 목록, 설명, 매개변수 스키마 등을 JSON-RPC 포맷을 통해 LLM과 주고받는다. 이 방식은 확장성과 신뢰성을 동시에 확보할 수 있으나, 서버 장애 또는 병목이 발생할 경우 전체 시스템의 성능 저하가 우려된다. Claude 데스크톱 클라이언트는 대표적인 MCP 호스트 환경 중 하나이며, 다양한 Python 기반 툴이 mcp SDK로 등록되어 동작한다.

15.2.3 ACP (Agent Communication Protocol)

Agent Communication Protocol(ACP)은 Anthropic 및 일부 커뮤니티 기반 프로젝트에서 MCP 이후의 보완 프로토콜로 제안된 것으로 알려져 있으며, 특히 로컬 에이전트 네트워크에서의 효율적인 통신을 목적으로 한다. ACP는 MCP와 달리 중앙 서버가 필요 없는 점에서 구조적 차이를 갖는다. 이를 통해 인터넷 연결 없이도 독립형 에이전트 간 상호작용이 가능하며, 스마트폰, 로봇, 임베디드 장치 같은 오프라인 환경에서도 작동한다.

기술적으로 ACP는 gRPC, IPC(Inter-Process Communication), ZeroMQ와 같은 고성능 경량 통신 프로토콜과 호환되며, 각 에이전트는 자신이 직접 메시지를 송수신할 수 있다. 따라서, 메시지 지연이 낮고, 보안적으로도 상대적으로 안전한 통신 경로를 구성할 수 있다. 이러한 구조는 IoT 기기나 로봇 간 협업, 모바일 환경에서의 비동기 AI 처리 등에 활용도가 높다.

15.2.4 A2A (Agent-to-Agent Protocol)

Agent-to-Agent Protocol(A2A)은 2024 년부터 Microsoft, Hugging Face, OpenAI 등 복수의 기업 및 오픈소스 커뮤니티에서 논의되기 시작한 개방형 표준이다. A2A는 대규모 분산 환경에서 복수의 에이전트가 서로 협력할 수 있는 수평적 통신 프레임워크를 지향한다. 특히 이 프로토콜은 기업 간 또는 클라우드-에지 간의 크로스 플랫폼 협업을 염두에 두고 설계되었으며, 보안성 및 인증, 작업 분산 구조가 핵심 기능으로 포함된다.

A2A는 서로 다른 네트워크 또는 조직 내에 분산되어 있는 LLM, 지식 DB, 계획자(planner) 등이 함께 워크플로우를 수행하는 데 적합하다. OAuth2 기반 인증, TLS 암호화, 멀티노드 네임스페이스 협상 등을 포함하며, 엔터프라이즈급 시스템 통합과 디지털 트윈, 실시간 협력 AI 시스템에 사용될 수 있다.

15.2.5 비교

MCP, ACP, A2A는 각각의 목적과 기술 요구사항에 따라 설계된 에이전트 통신 프로토콜이며, 단순히 대체 관계가 아니라 계층적 또는 협력적인 방식으로 병행 사용이 가능하다. MCP는 단일 LLM 중심의 외부 도구 활용에 최적화되어 있으며, ACP는 네트워크 독립적인 로컬 에이전트 협업을 지원한다. 반면 A2A는 대규모 분산 환경에서의 복수 에이전트 간 보안 협업을 가능하게 한다.

MCP는 단일 LLM에 툴과 데이터를 연결하는 "입력 인터페이스"이다.

ACP는 로컬 환경에서 "에이전트끼리 직접 대화하는 회선" 역할을 한다.

A2A는 복잡한 조직간 작업을 위한 "수평 분산 협업 인프라"이다.

이러한 프로토콜들은 향후 멀티 에이전트 AI 시스템의 실시간 협업, 자동화된 도구 오케스트레이션, 다양한 컴퓨팅 환경 간의 통합을 위한 플랫폼으로 사용될 수 있다.

15.3 MCP 아키텍처 구조 및 통신 방식 (Stdio vs SSE)

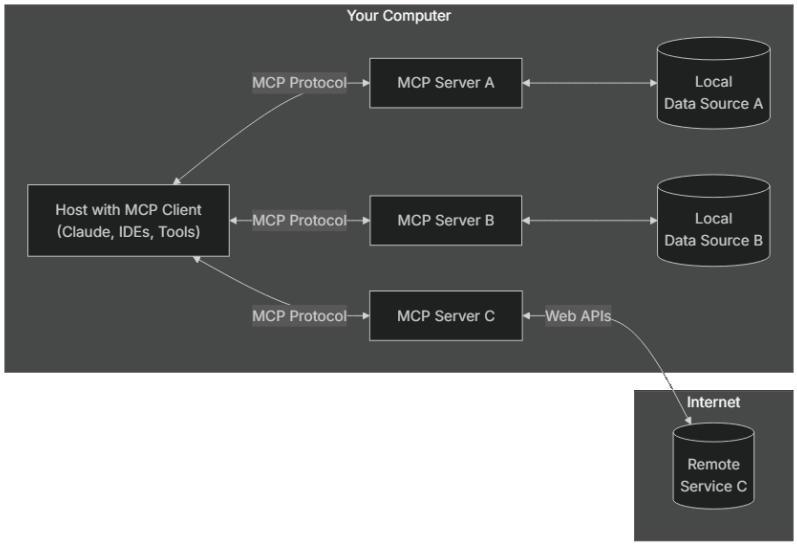
MCP는 클라이언트-서버 구조를 따르며 다음과 같은 요소로 구성된다.

MCP 호스트: Claude 데스크톱, AI 에이전트, 커서 IDE와 같은 MCP 클라이언트 통합 환경

MCP 서버: 기능을 노출하는 프로그램으로, 도구 등록 및 실행을 처리. Python, Node.js 등 다양한 런타임 환경에서 동작

로컬 데이터 소스: 파일 시스템, DB 등 로컬 자원

원격 서비스: GitHub, Brave Search 등 API 기반 외부 서비스



15.4 MCP 서버 실행 모드

stdio (표준입출력 기반):

MCP 서버는 stdin을 통해 명령을 받고 stdout으로 응답

로컬 단일 프로세스 통신에 적합하며 디버깅과 단순 실행에 유리

SSE (Server-Sent Events):

HTTP 기반 비동기 이벤트 스트리밍

/sse 엔드포인트에서 지속 연결을 통해 서버가 클라이언트로 데이터를 push

멀티클라이언트 및 장기 실행 애플리케이션에 적합

항목	stdio 모드	SSE 모드
통신 방식	stdin/stdout	HTTP + EventStream
장점	단순 구현, 로컬 적합	실시간 스트리밍, 확장성
활용 사례	테스트, 로컬 툴 실행	실서비스, LLM 연동

15.5 프로토콜 메시지 구조 (JSON-RPC)

MCP는 JSON-RPC 2.0 표준을 따른다. 다음은 MCP 도구 호출 예시이다:

요청(request)

```
{
  "jsonrpc": "2.0",
  "id": "call-1",
  "method": "callTool",
  "params": {
    "name": "create_record",
    "arguments": {
      "title": "New record",
      "content": "Record content"
    }
  }
}
```

응답(response)

```
{
  "jsonrpc": "2.0",
  "id": "call-1",
```

```
  "result": {
    "content": [
      {
        "type": "text",
        "text": "Created New Record"
      }
    ]
  }
}
```

이 구조는 함수 호출, 리소스 응답, 오류 처리 등을 표준화한다.

15.6 MCP 도구 설계 및 호출

도구는 MCP 서버 내에 JSON Schema를 기반으로 정의된다.

- 참고: Model Context Protocol (MCP) Anthropic 개발 방법 - WikiDocs

예시:

```
@app.list_tools()
async def list_tools():
    return [
        types.Tool(
            name="calculate_sum",
            description="두 수를 더하는 도구",
            inputSchema={
                "type": "object",
                "properties": {
                    "a": {"type": "number"},
                    "b": {"type": "number"}
```

```
                },
                "required": ["a", "b"]
            }
        )
    ]
```

도구 호출:

```
@app.call_tool()
async def call_tool(name: str, arguments: dict):
    if name == "calculate_sum":
        return [types.TextContent(type="text", text=str(arguments["a"] + arguments["b"]))]
```

15.7 MCP 설치 및 개발 실습 (Python 기반)

기본 설치:

```
pip install mcp pydantic-ai fastmcp tavily-python
```

계산기 도구 예제:

```
from mcp.server.fastmcp import FastMCP

mcp = FastMCP("SimpleCalc")

@mcp.tool()
def add(a: int, b: int) -> int:
    return a + b

if __name__ == "__main__":
    mcp.run()
```

서버 실행:

mcp dev mcp_simple_calc.py

15.8 클라이언트 코드 예시

Stdio 방식:

```
from mcp import ClientSession, StdioServerParameters
from mcp.client.stdio import stdio_client

server_params = StdioServerParameters(
    command="python",
    args=["./mcp_simple_calc_server.py"]
)

async def run():
    async with stdio_client(server_params) as streams:
        async with ClientSession(*streams) as session:
            await session.initialize()
            tools = await session.get_tools()
            print("도구 목록:", tools)
            result = await session.run("2 와 4 를 더해줘")
            print("응답 결과:", result.data)
```

SSE 방식:

```
from mcp.client.sse import sse_client
from mcp import ClientSession

async def run():
    async with sse_client(url="http://localhost:5173/sse") as streams:
        async with ClientSession(*streams) as session:
```

```
await session.initialize()

tools = await session.get_tools()

print("도구 목록:", tools)

result = await session.run("calculate power 2, 4?")

print("응답 결과:", result.data)
```

15.9 Ollama + MCP

저장소 클론:

git clone https://github.com/chrishayuk/mcp-cli

cd mcp-cli

의존성 설치:

pip install uv

uv sync --reinstall

Ollama 실행:

ollama run llama3.2

MCP CLI 실행:

uv run mcp-cli chat --server filesystem --provider ollama --model llama3.2

Ollama는 MCP 도구 검색 및 호출을 로컬 LLM 환경에서 가능하게 하며, 인터넷 없이도 도구 기반 AI 에이전트를 실행할 수 있다.

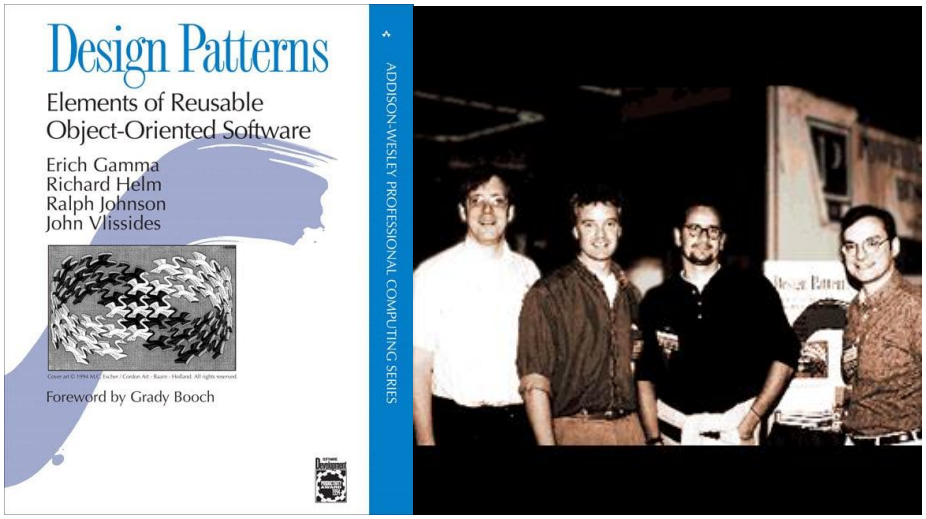
15.10 결론

MCP는 LLM 기반 시스템의 기능적 확장을 위해 핵심적인 역할을 수행하는 프로토콜이다. 표준화된 도구 호출, 다양한 런타임 지원, JSON-RPC 포맷 기반 통신, 로컬 및 원격 도구 통합 등은 향후 에이전트 기반 인공지능 시스템 설계의 기반이 될 것이다.

16. AI 에이전트 디자인 패턴

16.1 AI 에이전트에 디자인 패턴

디자인 패턴의 개념은 1994 년 "Gang of Four (GoF)"라고 불리는 에리히 감마(Erich Gamma), 리처드 헬름(Richard Helm), 랄프 존슨(Ralph Johnson), 존 블리시디스(John Vlissides) 네 명의 저자가『Design Patterns: Elements of Reusable Object-Oriented Software』에서 처음 정립한 것으로, 객체지향 소프트웨어 설계에서 반복적으로 등장하는 문제에 대한 해법을 패턴화한 것이 그 출발점이다. 이들은 OOA/D(Object-Oriented Analysis/Design)에서 각 요소들의 관계와 책임을 분리하여 유지보수가 용이하고 확장 가능한 아키텍처를 구성하는 기반이 되었다.



이와 같은 철학은 AI 에이전트 설계에도 그대로 적용된다. 특히 최근의 LLM 기반 에이전트는 단일 모델 호출을 넘어서, 문서 검색(RAG), 외부 도구 연동(MCP), 다중 LLM 협력, 멀티모달 처리 등 복합적인 시스템으로 진화하고 있다. 따라서 이를 체계적으로 조합하기 위한 "에이전트 디자인 패턴"의 정립은 DX 컨설턴트나 AX(Agent Experience) 설계자가 이해해야 할 역량이며, 독창적인 브랜드 에이전트를 기획하고 실현하는 데 기반이 될 수 있다.

16.2 AI 에이전트 디자인 패턴의 분류 및 원리

다음은 실제 구현 사례와 구조적 특징을 기반으로 정리한 주요 AI 에이전트 디자인 패턴 예시이다. 조직이나 개인의 목적에 따라 이런 방식으로 브랭딩하면 된다.

16.2.1 싱글 스킬 패턴 (Single Skill Agent)

이 패턴은 하나의 태스크에 특화된 에이전트를 구축할 때 사용된다. 예컨대, 사내 문서 기반 FAQ 검색, 날씨 응답, 이메일 요약 등 단일 기능에 초점을 맞추어, 파인튜닝된 LLM 혹은 RAG 기반 구조로 구성된다. 입력은 간단한 질의이며, 임베딩 검색기(retriever)와 하나의 LLM으로 구성된다. 이는 에이전트 설계의 가장 기본적인 형태이며, 단일 기능을 빠르게 서비스화하기에 적합하다.

16.2.2 멀티 스킬 라우팅 패턴 (Multi-Tool Routing Agent)

이 패턴은 입력에 따라 다양한 도구(tool) 또는 스킬을 선택적으로 실행하는 구조이다. 예를 들어 "계산해줘"라는 명령은 파이썬 실행 도구를, "날씨 알려줘"는 외부 API 도구를, "문서 요약해줘"는 LLM을 호출하는 식이다. MCP 기반 도구 시스템, OpenAI Function/Tool Calling, LangChain ReAct 등에서 구현 가능하며, 클라이언트는 입력을 분류하는 선택기(Classifier 또는 Planner)를 내장한다.

16.2.3 전문가 집단 패턴 (Expert Ensemble Agent)

이 패턴은 서로 다른 도메인에 특화된 LLM 또는 에이전트를 조합해 협력적인 응답을 생성한다. 예를 들어, 의학 질문은 Med-PaLM, 법률 질문은 GPT-Legal로 라우팅하거나, 모든 전문가의 답변을 받아 다수결 또는 점수 기반으로 결합하는 방식이다. 다중 에이전트를 조합하는 고급 전략으로, 전문성 기반 서비스에 적합하다.

16.2.4 멀티모달 에이전트 패턴 (Multimodal Retrieval Agent)

텍스트 외에도 이미지, 음성, 동영상, PDF, PPT, 음악 등 다양한 포맷의 데이터를 동시에 검색하고 분석하는 구조이다. 이를 위해 멀티모달 임베딩 모델(CLIP, BLIP, Whisper, MusicLM 등)이 사용되며, 통합된 벡터 검색 인덱스에서 유사도를 기준으로 문서를 탐색한다.
시각자료와 음성 기반 회의 요약 등 복합 입력 기반 서비스에 필수적인 패턴이다.

16.2.5 메타 에이전트 패턴 (Meta Agent)

에이전트가 다른 에이전트들을 통제하거나 조합할 수 있는 구조이다. 사용자의 복합 지시문("문서를 요약하고 PPT 만들어줘")을 받아, 먼저 요약 도구를 호출하고, 결과를 기반으로 생성 도구를 다시 호출하는 일련의 워크플로우를 구성한다. Self-RAG, Planner + Tool Executor, ReAct 기반 구조 등에서 이 패턴을 구현할 수 있다.

16.2.6 문맥 보강 패턴 (Context Enrichment Agent)

이 패턴은 LLM의 입력 컨텍스트를 외부 실시간 정보(API, DB, 클라우드 데이터 등)로 보강하는 방식이다. 주로 MCP(Model Context Protocol) 구조를 기반으로 하며, 서버가 제공하는 다양한 도구에서 실시간 정보를 얻어, 이를 LLM 입력에 삽입한다. 예를 들어, 환율 API를 호출한 후 결과를 사용자 질의와 함께 LLM에 전달하는 방식이다.

16.3 개발 전략과 브랜딩 관점 적용

AI 에이전트는 단순히 고성능 LLM을 사용하는 것으로 완성되지 않는다. 에이전트는 사용자의 니즈, 산업의 특성, 정보 구조에 맞는 설계 패턴을 바탕으로 구축되어야 하며, 이를 통해 브랜드의 신뢰성과 전문성을 확보할 수 있다.

AX(Agent Experience) 설계자는 사용자 시나리오와 인터랙션 흐름에 따라 위의 패턴들을 선택적으로 조합하여 UX 설계를 구체화해야 한다. 반면, DX 컨설턴트는 비즈니스 아키텍처와 기술 요소 사이의 구조를 해석하고, 최적화된 데이터 흐름과 모듈 구성을 설계할 수 있어야 한다. 단순한 API 연동이 아니라, 복수의 LLM과 도구가 어떻게 조화롭게 협력하며 각자의 강점을 발휘하게 할지를 브랜딩 관점에서 판단해야 한다.

예를 들어, '법률 전문 상담 에이전트'라는 브랜드를 기획한다면, 전문가 집단 패턴(법률, 계약, 노동법 등) + 문맥 보강 패턴(실시간 법령 데이터베이스) + 멀티모달 패턴(계약서 PDF 이해)을 조합한 구조가 필요하다.

만약, 사용하는 LLM 모델에 지식에 대한 토큰 임베딩이 없는 비즈니스 영역이라면 파인튜닝 등 모델 학습 개발이 별도로 필요하다.

16.4 결론

AI 에이전트 시스템은 본질적으로 다중 컴포넌트의 조합이며, 그 안에서 각 구성 요소들이 어떤 책임을 지니고 어떻게 협력할 것인지에 대한 설계가 핵심이다. 객체지향 디자인 패턴이 90 년대 소프트웨어 엔지니어링의 지형을 바꾼 것처럼, AI 에이전트 디자인 패턴은 AI 시대의 사용자 경험과 서비스 아키텍처를 규정하는 중심 원리가 될 것이다.

이러한 AI 에이전트 디자인 패턴은 향후 LLM 플랫폼 생태계의 발전과 함께 점점 더 구체화될 것이며, DX 전략가와 AX 설계자가 이 패턴들을 유연하게 조합하고 진화시키는 능력은 브랜드 차별화의 핵심 경쟁력이 될 수 있다.