

LLM RAG Agent

교안



Ver 1.3



Table of Contents

1. 개요 6

2. 트랜스포머 인코더 6

2.1 서론 6

2.2 트랜스포머 인코더의 구성 요소 및 역할 6

2.2.1 Scaled Dot Product Attention 6

2.2.2 Multi-Head Attention 7

2.2.3 Positional Encoding 7

2.2.4 Position-wise Feed Forward Network 7

2.2.5 Layer Normalization & Residual Connection 7

2.3 트랜스포머 인코더 구조 요약 8

2.4 PyTorch 기반 핵심 코드 분석 8

2.4.1 Scaled Dot Product Attention 8

2.4.2 MultiHeadAttention 8

2.4.3 Positional Encoding 8

2.4.4 EncoderLayer 9

2.5 결론 9

3. NLP 10

4. OpenAI CLIP 11

4.1 CLIP 개요 11

4.2 주요 기능 11

4.3 설치 11

4.3.1 설치 방법 11

4.3.2 의존성 11

4.4 모델 로드 11

4.5 이미지와 텍스트 유사도 계산 12

4.5.1 예제 코드: 12

4.6 제로샷 이미지 분류 12

4.7 멀티모달 검색 13

4.8 모델 확장 14

4.9 결론 15

5. Variational Autoencoders (VAE) 16

5.1 서론 16

5.2 VAE의 주요 개념 16

5.3 VAE의 구조 17

5.4 VAE 학습 목표 17

5.4.1 KL-발산과 VAE에서의 역할 17

5.4.2 KL-발산의 수학적 전개 18

5.4.3 KL-발산 항목별 해석 18

5.4.4 VAE의 학습 과정과 KL-발산의 역할 18

5.4.5 VAE 학습에 대한 의사코드 18

5.5 VAE의 주요 특징 20

5.6 VAE의 응용 분야 21

5.7 결론 21

6. Stable Diffusion 21

6.1 개요 21

6.2 확산 모델 (Diffusion Model) 개요 22

6.3 Stable Diffusion의 작동 원리 22

6.4 Stable Diffusion의 학습 과정 23

6.5 텍스트-이미지 매핑 23

6.6 Stable Diffusion 동작 과정 23

6.6.1 전체 동작 과정 요약 23

6.6.2 주요 전체 과정 의사코드 23

6.6.3 VAE (Variational Autoencoder) 24

6.6.4 U-Net 25

6.6.5 주요 동작 의사코드 25

6.6.6 설명 25

6.6.7 Text Encoder (CLIP Text Encoder) 26

6.6.8 최종 요약 26

6.7 Stable Diffusion 설치 및 실행 방법 26

6.8 Stable Diffusion의 활용 27

6.9 Stable Diffusion의 장점 27

6.10 결론 27

6.11 참고 문헌 28

7. 라마3 파인튜닝 29

7.1 주요 라이브러리 설명 및 로딩 29

7.2 사용자 정의 데이터 로딩 함수: load\_dataset\_from\_folder() 29

7.3 파인튜닝 전체 파이프라인: train() 30

7.4 베이스 모델 설정 및 인증 30

7.5 사용자 데이터셋 로딩 및 셔플링 30

7.6 QLoRA 기반 4-bit 양자화 설정 31

7.7 모델 및 토크나이저 로딩 + 대화 포맷 구성 31

7.8 LoRA 기반 파라미터 효율적 미세조정 31

7.9 데이터셋 전처리 및 통합 32

7.10 학습 하이퍼파라미터 구성 32

7.11 지도학습 기반 트레이닝: SFTTrainer 33

7.12 평가 및 모델 배포 33

7.13 결론 및 확장 33

8. LangChain 34

8.1 랭체인 개념 34

8.2 구조 34

8.3 프롬프트 템플릿 34

8.4 LCEL (LangChain Expression Language) 35

8.5 에이전트 방식 36

8.5.1 Zero-shot ReAct 36

8.5.2 Conversational ReAct 36

8.5.3 ReAct Docstore 37

8.5.4 Self-ask with Search 37

8.6 에이전트 동작 방식 38

8.6.1 개요 38

8.6.2 동작 흐름 38

8.6.3 LangChain 내부 프롬프트 구성 39

8.6.4 프롬프트 생성 위치 39

8.6.5 결론 40

8.7 기본 도구 40

8.8 벡터DB 및 검색 40

8.8.1 최대 마진 관련성(MMR, Maximal Marginal Relevance) 41

8.8.2 유사도 점수 임계치(Similarity Score Threshold) 41

8.8.3 단일 문서 검색(Single Document Retrieval) 42

8.8.4 필터 기반 검색(Filter-based Retrieval) 42

8.8.5 RAG 기반 Retrieval QA 체인 생성 42

8.8.6 결론 43

8.9 랭체인 이외 에이전트 도구 43

8.10 결론 44

# 개요

본 교안은 회차 주제별 주요 학습 내용을 기술한 것으로, 실제 강의에는 교안 이외 저자 블로그, github, huggingface, 논문 등이 활용된다.

교안이외에 과제는 별도 제출 요청할 것이며, 이 내용은 별도 문서에 기술한다. 전체 주제는 실라버스 문서에 기술된다.

# 트랜스포머 인코더

## 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 인코더의 작동 원리를 중심으로, 핵심 구성요소 및 PyTorch 기반 구현 코드를 분석하고 설명한다.

## 트랜스포머 인코더의 구성 요소 및 역할

### 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:



* Q, K의 내적은 현재 토큰이 문맥 내 다른 토큰과 얼마나 관련 있는지를 나타낸다.
* 정규화를 위해 dk\sqrt{d\_k} 로 나눈다.
* softmax를 통해 가중치를 확률화하고, V에 곱해 최종 어텐션 출력을 생성한다.

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

### 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, V를 헤드 수에 따라 분리하고,
* 각 헤드별 어텐션을 수행한 뒤,
* 결과를 concat하여 최종 출력으로 만든다.

### 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.



해당 인코딩은 임베딩 벡터에 더해져 입력으로 사용된다.

### Position-wise Feed Forward Network

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



이는 토큰별로 표현력을 확장해주는 역할을 한다.

### 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.

## 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.

## PyTorch 기반 핵심 코드 분석

다음은 핵심 모듈별로 구조를 단순화한 의사 코드이다.

### 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)

### 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

### 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)]

### 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

## 결론

본 교안은 트랜스포머 인코더의 핵심 개념과 구조, 그리고 PyTorch 구현 기반의 작동 원리를 상세히 분석하였다. 트랜스포머를 정확히 이해하는 것은 생성 AI, 챗봇, 자연어 이해 등 다양한 응용 분야에 있어 필수적인 지식이다. 본 교안을 통해 학생들은 단순한 모델 사용을 넘어 내부 동작 원리를 이해하고, 이를 실제 프로젝트에 응용할 수 있는 기반을 갖출 수 있다.

# NLP

TBD

# OpenAI CLIP

## 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.

## 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.

## 설치

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

### 설치 방법

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

pip install torch torchvision

### 의존성

Python 3.7+

PyTorch 1.7.1+

torchvision

## 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")

## 이미지와 텍스트 유사도 계산

CLIP은 이미지와 텍스트의 표현을 동일한 임베딩 공간으로 변환하여 유사도를 계산한다.

### 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)

## 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()])

## 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())

## 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()

## 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.

# Variational Autoencoders (VAE)

## 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.

## 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.

## 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:



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.

## VAE 학습 목표

### 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.

### 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 σ²를 가지고, 표준 정규분포 p(z)는 평균 0과 분산 1을 가진다. KL-발산을 계산하면 다음과 같은 식으로 전개된다:  
D\_{KL}(q(z|x) || p(z)) = ∫ N(μ, σ²) log (N(μ, σ²) / 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/σ²) + σ² + μ² - 1]

### KL-발산 항목별 해석

KL-발산을 구성하는 각 항목은 다음과 같은 의미를 가진다:  
1. log(1/σ²): 분산 σ²에 대한 상대적인 정보량을 측정한다. σ²가 클수록 분포가 더 넓어지며, 그만큼 정보가 덜 밀집되어 있다는 의미이다.  
2. σ²: q(z|x)의 분산이 1(표준 정규분포)의 분산에 얼마나 가까운지, 또는 얼마나 멀리 떨어져 있는지를 측정한다.  
3. μ²: q(z|x)의 평균이 표준 정규분포의 평균(0)과 얼마나 차이가 나는지를 측정한다. μ²가 클수록 q(z|x)의 평균은 표준 정규분포와 거리가 멀다는 의미이다.  
4. -1: 표준 정규분포의 분산이 1이므로, KL-발산의 마지막 항목은 이 값에 대한 보정이다.

### 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.

### 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()

## 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.

## 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.

## 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.

# Stable Diffusion

## 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.

## 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.

## 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.

## 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.

## 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.

## Stable Diffusion 동작 과정

Stable Diffusion의 동작은 크게 다음과 같다.

### 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.

### 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)

### 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 zz를 샘플링한다.
* **Decoder**는 latent zz를 받아 이미지를 복원한다.

### 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.

### 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

### 설명

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

### 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.
* Transformer를 통해 각 토큰에 대해 임베딩을 얻는다.
* 최종적으로 이 텍스트 임베딩이 U-Net의 Attention 모듈에 주입된다.

### 최종 요약

Stable Diffusion은 다음과 같은 구조로 동작한다:

1. 텍스트를 임베딩한다.
2. 랜덤 노이즈 latent에서 시작한다.
3. U-Net을 통해 노이즈를 점진적으로 제거하며 latent를 정제한다.
4. 최종 latent를 VAE를 통해 고해상도 이미지로 복원한다.

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

## 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 페이지](https://huggingface.co/CompVis/stable-diffusion-v-1-4-original)에서 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.

## 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.

## 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.

## 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.

## **참고 문헌**

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

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

# 라마3 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.

## 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.

## 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.

## 파인튜닝 전체 파이프라인: train()

설정에 따라 학습을 시작함.

## 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.

## 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.

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

bnb\_4bit\_quant\_type="nf4": NormalFloat4는 학습 가능성이 높은 정규 분포 중심의 표현력 향상된 포맷.

bnb\_4bit\_use\_double\_quant=True: 2차 양자화를 통해 정보 손실 최소화와 압축 효율을 향상시킴.

bnb\_4bit\_compute\_dtype=torch.float16: 연산은 float16으로 진행되어 연산 성능을 확보.

## 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.

## 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.

## 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.

## 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.

## 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.

## 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.

## 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.

# LangChain

## 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.

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)

## 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.

## 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)

## 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)

## 에이전트 방식

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

### 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)

### 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)

### 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)

### Self-ask with Search

질문을 세분화하여 자체 검색 및 응답을 생성한다.

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)

## 에이전트 동작 방식

### 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.

### 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:
   1. 특정 도구를 사용할 것 (Action: tool name, Action Input: input)
   2. 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.

### 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.

### 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.

### 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.

## 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)

## 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)

### 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 6개를 반환하되, λ\lambda 값을 통해 다양성을 강조한다.

### 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.

### 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})

### 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'}}

)

### 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 DB에서 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:**
  1. search\_type="mmr": MMR 알고리즘을 사용한다.
  2. k=6: 검색된 문서 중 상위 6개 문서를 반환한다.
  3. lambda\_mult=0.25: 관련성과 다양성의 균형을 조정한다.

### 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.

## 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 모델을 활용하여 다양한 작업을 자동화할 수 있도록 설계됨. | [AutoGPT GitHub](https://github.com/Significant-Gravitas/AutoGPT) |
| **Pydantic** | AI 에이전트 및 모델 구축을 위한 데이터 검증 및 설정 도구. Pydantic을 사용하여 AI 모델의 입력과 출력을 구조적으로 관리하고 검증할 수 있다**.** | [PydanticAI](https://ai.pydantic.dev/) |
| **LlamaIndex** | Llama 모델을 활용하여 효율적인 데이터 검색 및 에이전트 기능을 구현하는 도구. 특히 대규모 텍스트 데이터 처리 및 검색에 강점. | [LlamaIndex GitHub](https://github.com/jerryjliu/llama_index) |
| **CrewAI** | 협업을 중심으로 AI 에이전트를 관리하는 도구. 여러 AI 에이전트를 연결하여 복잡한 문제를 해결하는데 유용. | [CrewAI Website](https://github.com/crewAIInc/crewAI) |
| **LangGraph** | 그래프 데이터베이스 기반의 AI 에이전트 관리 도구로, 복잡한 데이터 관계를 효과적으로 처리하고 탐색할 수 있도록 돕는다. | [LangGraph GitHub](https://github.com/langchain-ai/langgraph) |

## 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.