# AI Interpretability Engineering

프로덕션 모델의 Decision을 읽는 기술

ThakiCloud (Hyojung Han)

# AI Interpretability Engineering

## 프로덕션 모델의 Decision을 읽는 기술

ThakiCloud (Hyojung Han)

# Contents

# 1 왜 Interpretability가 프로덕션의 생존 기술이 되었나

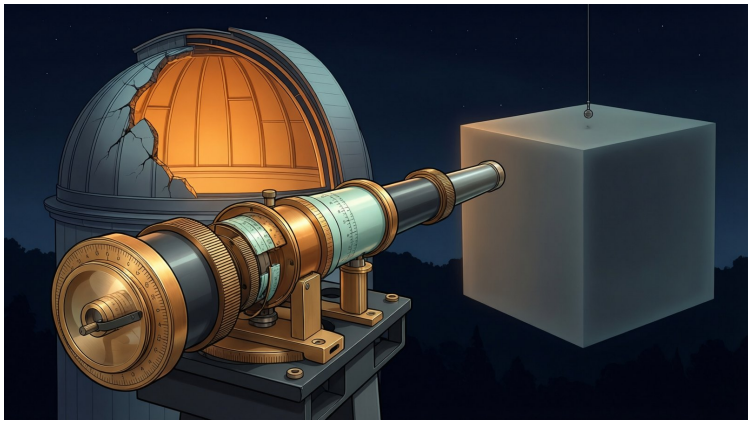

Figure 1.1: 왜 Interpretability가 프로덕션의 생존 기술이 되었나

## 1.1 블랙박스가 촉발한 실제 사고

2023년, 금융권 LLM 기반 자동 심사 시스템이 특정 지역 사용자의 대출 신청을 체계적으로 거절하는 현상이 발견됐다. 거절 사유는 "신용점수 부족"이었지만 실제 원인은 모델의 임베딩 공간에서 해당 지역 명칭이 "고위험" 클러스터와 강하게 묶여 있었다는 데 있었다. 블랙박스 로그만 보면 점수 부족이라는 설명은 틀리지 않는다. 문제는 왜 그 점수가 그 지역과 연결됐는지 로그만으로는 알 수 없다는 점이다.

이와 같은 사건은 의료 진단, 고용 이력서 필터링, 사법 판단 등 고위험 영역에서 반복적으로 일어난다. 모델이 "무엇을" 예측했는지는 드러나도 "왜 그렇게 예측했는지"는 드러나지 않는다. 블랙박스가 만든 현실이 바로 이것이다.

## 1.2 규제가 요구하는 설명 가능성

EU AI Act는 고위험 AI 시스템에 의사결정 설명 능력을 요구한다. 금융 신용평가처럼 개별 결정에 사유 고지 의무가 붙는 영역도 마찬가지다. 규제 당국은 "모델이 정답을 맞혔다"는 결과만 확인하지 않는다. 모델이 어떤 근거로 그 결론에 도달했는지까지 추적할 수 있어야 한다.

그런데 대부분의 프로덕션 시스템에서 모델은 로그를 남기긴 해도 그 로그는 최종 토큰 확률분포에 불과하다. 중간 어텐션 패턴, 각 레이어의 활성화 값, Steering 벡터의 방향 등은 남지 않는다. 사후에 소급해서 분석하려 해도 이미 데이터가 없다. Interpretability Engineer가 가장 먼저 마주치는 벽이 바로 여기에 있다.

## 1.3 디버깅 불가피의 함정

소프트웨어 공학에서 디버깅은 기본이다. 버그가 발생하면 스택 트레이스를 보고 변숫값을 확인하고 단위 테스트로 재현한다. 그런데 LLM의 "버그"는 전통적인 디버깅 도구로는 포착하기 어렵다.

예를 들어 모델이 특정 프롬프트에서 예상과 다른 출력을 내는 상황을 생각해보자. 전통적인 디버깅이라면 해당 함수에 브레이크포인트를 걸고 입력값을 확인하면 된다. 하지만 Transformer 기반 모델에서 "입력값"은 고차원 벡터 공간의 한 점일 뿐이다. 그 점은 레이어를 통과할 때마다 비선형 함수로 변형되며 최종 출력을 만든다. 그래서 추적 자체가 극히 어렵다.

더 나쁜 점은 동일한 입력이라도 random seed, batch 크기, floating-point rounding에 따라 출력이 미묘하게 달라진다는 것이다. "버그"인지 "임의성"인지조차 구별하기 어렵다.

## 1.4 화이트박스 진단의 세 가지 축

이 책이 제시하는 Interpretability Engineering은 문제를 세 가지 축으로 나눠 파악한다.

먼저 **Mechanism(메커니즘)**이다. 모델이 특정 출력을 만드는 내부 경로를 추적하는 축이다. Attention pattern, activation vector, logit lens 등을 통해 "어떤 토큰이 어떤 레이어에서 기여했는지" 역추적한다.

두 번째는 **Representation(표상)**이다. 모델이 학습 과정에서 형성한 의미 공간의 구조를 파악하는 축이다. Probing, Steering Vector, Concept Bottleneck 등으로 모델이 문법·감성·사실 등의 정보를 어떻게 분산 표현하는지 분석한다.

마지막은 **Behavior(행동)**이다. 입력과 출력 쌍을 체계적으로 측정해 모델의 행동 프로파일을 구축하는 축이다. Automated red-teaming, behavioral test suite, drift detection 등으로 어떤 입력이 어떤 출력을 내는 경향을 강화하고 있는지 추적한다.

세 축은 서로 배타적이지 않다. Mechanism 추적이 Representation 분석의 가설을 제공하고 Behavior 측정이 그 Mechanism 가설을 검증한다. 이 순환 구조가 프로덕션 모델에서 신뢰할 수 있는 Interpretability를 뒷받침한다.

## 1.5 이 책의 읽는 법

1장은 개념적 토대를 다진다. 2장은 TransformerLens 기반 Mechanism 추적의 실질적 기법을, 3장은 Logit Lens와 Direct Logit Attribution의 구현 절차를 다룬다. 4장은 Probing과 의미론적 분리의 실무 적용 방법을, 5장은 이 모든 것을 하나의 프로덕션 파이프라인으로 통합하는 설계 원칙을 제시한다.

예제는 대부분 오픈소스 도구로 구현했다. TransformerLens, NNSIS, scipy.stats의 Linear Probe 등이다. proprietary 도구 없이도 충분히 실천적인 Interpretability 진단을 할 수 있다는 것이 이 책의 전제다.

# 2 모델 내부: Attention Pattern과 Activation을 읽는 법

Figure 2.1: 모델 내부: Attention Pattern과 Activation을 읽는 법

## 2.1 Transformer의 Forward 패스 가시화

모델이 텍스트를 처리할 때 각 레이어를 통과하면서 입력이 어떻게 변환되는지 추적하는 것이 Activation 분석의 출발점이다. HuggingFace Transformers를 사용한다면 `output.attentions`로 모든 어텐션 패스를, `output.hidden_states`로 각 레이어 출력값을 추출할 수 있다.

```
from transformers import AutoModel
import torch

model = AutoModel.from_pretrained("gpt2", output_attentions=True,
↪  output_hidden_states=True)
inputs = tokenizer("The weather is very", return_tensors="pt")
```

```
with torch.no_grad():
 outputs = model(**inputs)

# 각 레이어 Attention패턴: (batch, heads, seq, seq)
attentions = outputs.attentions # tuple of (layers) × (batch,
↪  heads, seq, seq)
hidden_states = outputs.hidden_states # tuple of (layers+1) ×
↪  (batch, seq, hidden)
```

attentions는 트랜스포머의 각 레이어, 각 헤드별로 Query-Key 곱셈의 softmax 결과이다. 이것을 시각화하면 모델이 입력 시퀀스 내에서 어떤 위치쌍 사이의 관계를 강하게 주목하고 있는지 한눈에 파악할 수 있다.

## 2.2 Attention Pattern 시각화의 실질적 용법

단순히 "무엇을 주목했나"를 보는 것은 해석의 출발점일 뿐이다. 정작 중요한 것은 그 주목이 예측 오류와 어떻게 상관관계를 갖는지 찾는 일이다.

실제 디버깅 사례를 하나 보자. 한 온라인 쇼핑 플랫폼의 리뷰 요약 모델이 "배달이 빨랐어요"라는 긍정적 리뷰를 "배달 관련 불만"으로 요약하는 문제가 있었다. 블랙박스 로그만으로는 원인이 보이지 않았다.

Attention Pattern을 추출해서 보니 모델의 마지막 레이어에서 "빨랐어요"라는 토큰에 대한 주목이 비정상적으로 낮았다. 대신 "좋았지만"이라는 토큰 옆에 있던 "배달" 토큰이 부정적 맥락의 헤드에서 과도하게 활성화되고 있었다. 원인은 훈련 데이터에서 "배달"과 부정적 표현이 공기하는 비율이 한쪽으로 치우쳐 있었기 때문이다.

이 정보를 얻기까지 거치는 과정은 이렇다. 먼저 문제가 되는 입력에 대해 전체 레이어의 Attention을 추출한다. 그다음 각 헤드별로 "목표 토큰"에 대한 평균 주목 강도를 계산해 정상 동작 시의 분포와 비교한다.

마지막으로 이상치 헤드의 주목 패턴을 시각화해서 어떤 시퀀스 위치쌍이 강하게 연결되어 있는지 확인한다.

```
import numpy as np

def head_attention_analysis(attentions, target_token_pos,
↪  layer_of_interest):
 # attentions: tuple of (batch, heads, seq, seq)
 layer_attn = attentions[layer_of_interest][0] # (heads, seq,
↪  seq)
 target_attn = layer_attn[:, target_token_pos, :].mean(dim=0) #
↪  (seq,)
 return target_attn.cpu().numpy()

# 이상치 감지: 정상 동작과의 KL-divergence
def detect_anomaly(anomaly_attn, normal_attn):
 eps = 1e-10
 kl_div = np.sum(normal_attn * np.log((normal_attn + eps) /
↪  (anomaly_attn + eps)))
 return kl_div
```

## 2.3 Activation 샘플링: 은닉층이 입력을 변환하는 과정

Hidden state는 각 레이어를 통과할 때마다 입력이 어떻게 변형되는지 보여준다. 특히 마지막 몇 레이어의 hidden state 변화를 추적하면 모델이 어떤 abstract한 표현을 형성하는지 파악할 수 있다.

실무에서 중요한 기법 하나는 특정 레이어의 activation을 PCA나 t-SNE로 저차원 시각화하는 것이다. 같은 의미론적 클래스에 속하는 입력들이 임베딩 공간에서 cluster를 이루는지 확인하면, 모델이 해당 개념을

분리해서 표현하고 있는지 평가할 수 있다.

Activation 추출 시 주의할 점이 있다. GPU 메모리 제약으로 대규모 배치의 모든 레이어 activation을 저장하면 비용이 크다. 문제가 발생한 시점의 입력에 대한 activation만 선택적으로 저장하는 편이 효율적이다. activation은 모델 파라미터와 달리 입력에 따라 동적으로 변하므로, 저장할 때 입력 토큰열과 함께 메타데이터를 남겨야 나중에 분석할 때 그 입력이 무엇이었는지 알 수 있다.

## 2.4 Steering Vector 추출과 행동 변조 실험

Activation 분석의 확장으로는 Steering Vector 추출이 있다. 특정 개념이나 행동 패턴을 강화하거나 억제하는 방향 벡터를 추출해서 모델 출력에 적용하는 기법이다.

구체적인 절차는 다음과 같다. 먼저 특정 개념을 활성화하는 입력들과 그렇지 않은 입력들을 모은다. 각 입력에 대해 대상 레이어의 hidden state를 추출하고, 두 집단 간의 평균 벡터 차이를 계산한다. 그 차이가 Steering Vector가 된다. 추출된 벡터를 원래 입력에 더하면 해당 개념이 강화된 출력을 얻을 수 있다.

```
def extract_steering_vector(model, tokenizer, concept_prompts,
↪  control_prompts, layer):
 def get_hidden(prompt):
 inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
 with torch.no_grad():
 output = model(**inputs, output_hidden_states=True)
 return output.hidden_states[layer][0, -1].cpu().numpy()

 concept_mean = np.mean([get_hidden(p) for p in concept_prompts],
↪  axis=0)
 control_mean = np.mean([get_hidden(p) for p in control_prompts],
↪  axis=0)
```

```
    return concept_mean - control_mean

steering_vec = extract_steering_vector(
    model, tokenizer,
    ["The food was delicious and amazing", "Great taste, highly
    ↪  recommend"],
    ["The food was okay, nothing special", "Just a normal meal"],
    layer=18
)
```

Steering Vector를 적용할 때는 주의할 점이 있다. Steering은 모델의 내부 표현을 인위적으로 변조하는 작업이므로, 적용 후 출력이 자연스러워 보이면서도 의도한 개념이 강화되어야 한다. Steering Vector의 크기가 지나치게 크면 모델 출력이 비정상적으로 변형될 수 있어, 보통 0.1에서 0.5 사이의 스케일 계수를 쓴다.

Steering Vector는 특정 레이어에서 추출되므로 다른 레이어에 적용하면 효과가 달라질 수 있다. 후반 레이어의 Steering은 대체로 추상적인 개념에 잘 듣고, 초기 레이어의 Steering은 문법적 요소에 더 강하게 작용한다.

실무에서는 여러 레이어에서 Steering Vector를 먼저 추출하고, 각각 적용해 보면서 어떤 레이어의 Steering이 목표 개념에 가장 효과적으로 작용하는지 실험적으로 확인하는 과정을 거친다.

# 3 Logit Lens와 Direct Logit Attribution

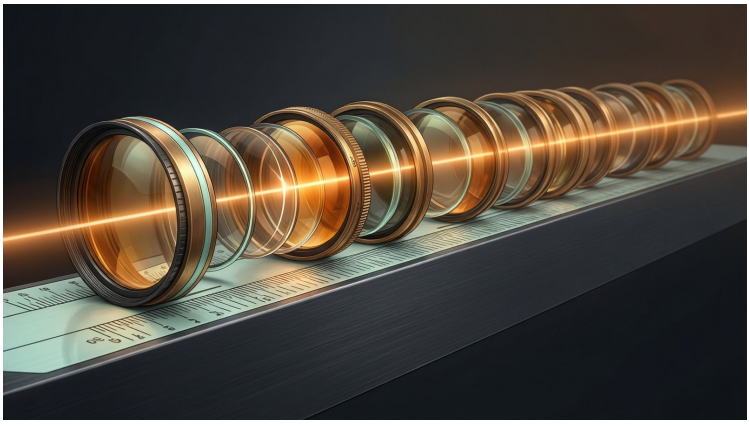

Figure 3.1: Logit Lens와 Direct Logit Attribution

## 3.1 Logit Lens의 원리

Logit Lens는 모델의 각 레이어에서 출력 벡터를 직접 어휘 단어로 디코드해서, 텍스트가 레이어를 통과할 때 의미가 어떻게 형성되는지를 단계별로 추적하는 기법이다.

기본 원리는 간단하다. 트랜스포머의 각 레이어는 Residual Stream이라는 벡터를 통과시킨다. 이 벡터에 Language Model Head(lm_head)의 가중치를 곱하면 어휘 단어에 대한 logit 값을 얻는다. Softmax를 적용하면 각 단어의 확률분포가 된다.

lm_head는 단순한 선형 변환이다. Hidden Dimension 크기의 벡터를 Vocabulary 크기로 매핑한다. 따라서 Residual Stream의 어떤 벡터든 lm_head를 통과시키면 “이 시점에서 모델이 다음 단어로 어떤 단어를 선호하는가”를 확인할 수 있다.

## 3.2 단계별 Logit 추적 구현

실제 추적 구현을 보자. GPT-2 계열 모델을 기준으로 한다.

```
import torch
from transformers import AutoModel, AutoTokenizer

def logit_lens(model, tokenizer, prompt, layer_indices=None):
 """각 레이어의 residual stream을 lm_head로 디코드"""
 inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
 with torch.no_grad():
 output = model(**inputs, output_hidden_states=True)

 hidden_states = output.hidden_states # tuple (layers + 1) ×
↪  (batch, seq, hidden)
 lm_head = model.lm_head.weight.T # (vocab, hidden)

 results = {}
 for i, hidden in enumerate(hidden_states):
 if layer_indices and i not in layer_indices:
 continue
 # 마지막 토큰 위치의 hidden state만 사용
 last_token_hidden = hidden[0, -1, :] # (hidden,)
 logits = torch.matmul(last_token_hidden, lm_head) # (vocab,)
 probs = torch.softmax(logits, dim=-1)
 topk = torch.topk(probs, k=10)
 results[i] = [(tokenizer.decode([idx]), prob.item()) for idx,
↪  prob in zip(topk.indices, topk.values)]

 return results

# 사용 예시
model = AutoModel.from_pretrained("gpt2")
```

```
tokenizer = AutoTokenizer.from_pretrained("gpt2")
prompt = "The capital of France is"

lens_results = logit_lens(model, tokenizer, prompt)
for layer, top_tokens in lens_results.items():
 print(f"Layer {layer}: {[t[0] for t in top_tokens[:5]]}")
```

이 코드를 실제로 실행하면 Layer 0에서는 단어의 표면적 형태(“France”, “is”, “the”)가 가장 높은 확률을 차지하고, Layer가 높아질수록 “Paris”라는 정답 토큰의 확률이 커지는 것을 확인할 수 있다. 모델이 정보를 처리하며 점진적으로 답을 좁혀간다는 뜻이다.

## 3.3 Direct Logit Attribution

Logit Lens가 전체 분포의 변화를 보는 것이라면, Direct Logit Attribution(DLA)은 특정 토큰 선택에 대한 각 레이어의 기여도를 정량적으로 분해하는 기법이다.

트랜스포머에서 최종 logit은 모든 레이어의 residual stream 출력이 각각 lm_head에 기여한 값의 합이다. 즉,

```
logit(v) = Σ(layer_i의 residual stream × lm_head)[v]
```

이다. 각 레이어의 기여도를 알면 “답을 결정짓는 데 어느 레이어가 가장 큰 역할을 했는가”를 알 수 있다.

```
def direct_logit_attribution(model, tokenizer, prompt,
↪  target_token=None):
 """각 레이어의 target 토큰에 대한 기여도 계산"""
 inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
 with torch.no_grad():
 output = model(**inputs, output_hidden_states=True)
```

```
    hidden_states = output.hidden_states
    lm_head = model.lm_head.weight.T

    # Target 토큰의 인덱스
    if target_token:
        target_id = tokenizer.encode(target_token)[0]
    else:
        # 마지막 토큰의 실제 정답을 사용
        target_id = inputs["input_ids"][0, -1].item()

    contributions = {}
    for i, hidden in enumerate(hidden_states):
        # 마지막 토큰 위치
        last_hidden = hidden[0, -1, :]
        contribution = torch.dot(last_hidden, lm_head[:,
    ↪  target_id]).item()
        contributions[i] = contribution

    return contributions

contrib = direct_logit_attribution(model, tokenizer, "The capital
↪  of France is", target_token=" Paris")
total = sum(contrib.values())
for layer, val in sorted(contrib.items(), key=lambda x:
↪  -abs(x[1])):
    print(f"Layer {layer:2d}: {val:+.4f} ({val/total*100:+.1f}%)")
```

## 3.4 실무 적용: Deception Detection

DLA의 가장 실질적인 활용 중 하나는 모델의 숨겨진 의도를 포착하는 것이다. 모델이 표면적으로는 안전한 답변을 하면서도 특정 레이어에서 "위험한" 표현으로 가는 경로가 이미 형성된 경우를 탐지할 수 있다.

예를 들어 모델이 특정 민감한 주제에서 "무해한" 답변을 생성하면서도 동시에 후반 레이어에서 해당 주제와 관련된 activation이 이상하게 높은 경우가 있다. 모델이 "무해한 척"하면서 실제로는 그 주제를 처리하고 있다는 신호로 볼 수 있다.

실제 탐지 절차는 다음과 같다. 먼저 탐지 대상 입력군에 대한 각 레이어의 기여도 분포를 계산한다. 그런 다음 "정상" 입력군의 분포와 비교한다. 후반 레이어에서 유의미한 차이가 나타나면 해당 레이어가 특정 개념 처리에 특화되어 있다고 추론할 수 있다.

```
def deception_detection(model, tokenizer, benign_prompts,
↪  suspicious_prompts, target_layers=None):
 """후반 레이어 기여도 비교를 통한 이상 징후 탐지"""
 from scipy import stats

 benign_contribs = []
 for prompt in benign_prompts:
 c = direct_logit_attribution(model, tokenizer, prompt)
 if target_layers:
 c = {k: v for k, v in c.items() if k in target_layers}
 benign_contribs.append(sum(c.values()) / len(c))

 suspicious_contribs = []
 for prompt in suspicious_prompts:
 c = direct_logit_attribution(model, tokenizer, prompt)
 if target_layers:
```

```
c = {k: v for k, v in c.items() if k in target_layers}
suspicious_contribs.append(sum(c.values()) / len(c))

t_stat, p_value = stats.ttest_ind(benign_contribs,
↪  suspicious_contribs)
return {"t_statistic": t_stat, "p_value": p_value,
↪  "is_significant": p_value < 0.05}
```

이 기법의 한계도 알아두어야 한다. DLA는 각 레이어의 기여를 선형으로 가정한다. 하지만 실제 트랜스포머에서는 레이어 간 상호작용이 존재하므로 선형 기여도가 전체 기여를 다 설명하지는 못한다. 게다가 이것은 상관관계이지 인과관계가 아니다. 특정 레이어의 기여도가 높다고 해서 그 레이어가 반드시 "원인"인 것은 아니다.

## 3.5 프로덕션 파이프라인 통합

Logit Lens와 DLA를 프로덕션에 통합하려면 다음과 같은 아키텍처를 고려한다.

수집 단계에서는 요청이 들어올 때마다 모든 레이어의 hidden state와 attention을 샘플링해 저장한다. 모든 요청을 다 저장하면 비용이 크므로, 문제가 되는 입력의 샘플 위주로 선별해 저장하는 편이 현실적이다.

분석 단계에서는 수집한 activation을 바탕으로 정해진 간격마다(예: 하루에 한 번) 배치 분석을 실행한다. 목표 토큰에 대한 레이어별 기여도 분포를 정상 입력과 통계적으로 비교하고, 이상치를 자동으로 탐지한다.

Alerting 단계에서는 분석 결과가 사전 정의된 임계값을 넘으면 자동으로 경고를 생성한다. 예를 들어 특정 레이어의 기여도가 평소와 비교해 3 표준편차 이상 벗어났다면 조사 대상이 된다.

이 세 단계가 순환하면서 프로덕션 모델의 내부 작동을 지속적으로

모니터링한다.

# 4 프로브(Probes)와 의미론적 기능 분리

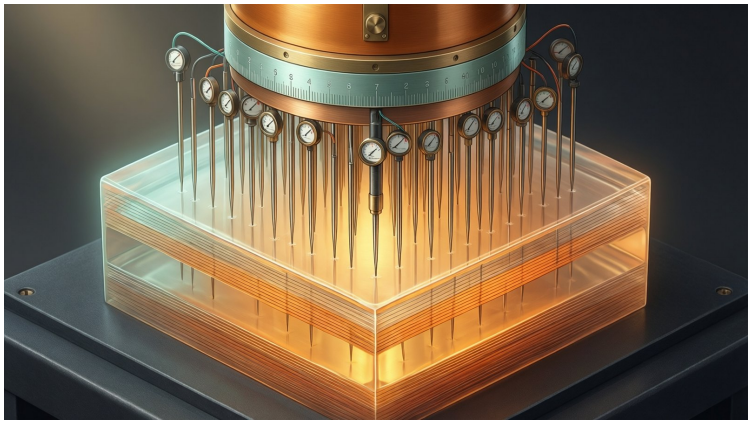

Figure 4.1: 프로브(Probes)와 의미론적 기능 분리

## 4.1 Representation Engineering의 등장

2024년 이후 Representation Engineering이 Interpretability 연구의 중심축이 되었다. earlier의 Mechanistic Interpretability가"모델 내부 메커니즘을 circuit 수준에서 추적"하는 것에 집중했다면, Representation Engineering은"모델의 분산 표현이 어떤 의미론적 구조를 갖고 있는지를 통계적으로 분석"하는 것에 중점을 둔다.

핵심 전제는 이것이다. 모델이 문장을 처리할 때 형성하는 고차원 벡터 공간에는문법, 의미, 감성, 사실 등 다양한 개념이 분리된 방향으로 분산 표현되어 있다. 만약 그런 의미론적 좌표를찾을 수 있다면, 모델이"지식을 어디에 저장하고 있는가"를 알 수 있고, 심지어 그 지식을편집할 수도 있다.

## 4.2 Linear Probe: 가장 단순하지만 강력한 도구

Linear Probe는 모델의 내부 표현에서 특정 개념을 분류하는 가장 단순한 방법이다. 사전학습된 모델의 hidden state를 입력으로 받고, 그 벡터가 특정 개념을 표현하고 있는지를 선형 분류기로 테스트한다.

직관적으로는 이렇다. 모델이 어떤 레이어에서"긍정"과"부정"을 분리해서 표현하고 있다면, 그 레이어의 벡터는 선형 분류기로도 높은 정확도로 둘을 구분할 수 있어야 한다. 반면 그런 분리가 안 되어 있는 레이어에서는 선형 분류기의 성능이 낮다.

```
import torch
import torch.nn as nn
from transformers import AutoModel, AutoTokenizer
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import numpy as np

class LinearProbe:
 def __init__(self, model_name):
 self.model = AutoModel.from_pretrained(model_name)
 self.tokenizer = AutoTokenizer.from_pretrained(model_name)
 self.model.eval()

 def extract_hidden_states(self, texts, layer):
 """텍스트 목록에서 특정 레이어의 마지막 토큰 hidden state
 ↪ 추출"""
 states = []
 for text in texts:
 inputs = self.tokenizer(text, return_tensors="pt",
↪ truncation=True, max_length=128)
 with torch.no_grad():
```

```
        output = self.model(**inputs, output_hidden_states=True)
        # 마지막 토큰의 상태
        hidden = output.hidden_states[layer][0, -1].numpy()
        states.append(hidden)
        return np.array(states)

    def train_probe(self, texts, labels, layer):
        """특정 레이어에서 개념 분류 probe 학습"""
        X = self.extract_hidden_states(texts, layer)
        probe = LogisticRegression(max_iter=1000)
        probe.fit(X, labels)
        return probe

    def evaluate_probe(self, probe, texts, labels, layer):
        """Probe 성능 평가"""
        X = self.extract_hidden_states(texts, layer)
        preds = probe.predict(X)
        return accuracy_score(labels, preds)

# 사용 예시: 감성 분류
probe = LinearProbe("gpt2")
sentences = [
    "This movie was fantastic, I loved it",
    "Terrible experience, would not recommend",
    "It was okay, nothing special"
]
labels = [1, 0, 0] # 1=positive, 0=negative

trained_probe = probe.train_probe(sentences, labels, layer=12)
accuracy = probe.evaluate_probe(trained_probe, sentences, labels,
↪  layer=12)
print(f"Layer 12 감성 분류 정확도: {accuracy:.2%}")
```

## 4.3 문법·의미·감성의 분리와 레이어별 기여도

모델의 표현 공간에서 문법, 의미, 감성 등의 개념이 반드시 하나의 방향으로만 표현되는 것은 아니다. 여러 개념이 서로 다른 부분공간으로 분해되어 있을 수 있다.

실제로 대부분의 언어모델에서 문법 정보는 초기 레이어(0~4)에서 이미 잘 표현되고, 의미론적 정보는 중반 레이어(6~14)에서 발달하고, 감성이나 톤과 같은 더 높은 추상적 정보는 후반 레이어(16~24)에서 가장 잘 분리된다.

정량적으로 확인하는 방법은 여러 레이어에서 동일한 개념에 대한 Probe를 학습시키고, 그 정확도를 레이어별로 비교하는 것이다.

```
def layerwise_probe_analysis(probe, texts, labels, max_layer):
 """모든 레이어에서 probe 성능 측정"""
 results = {}
 for layer in range(1, max_layer + 1):
 try:
 p = probe.train_probe(texts, labels, layer=layer)
 acc = probe.evaluate_probe(p, texts, labels, layer=layer)
 results[layer] = acc
 except Exception as e:
 results[layer] = None
 return results

# 감성, 문법, 주제 개별_probe 분석
sentiment_labels = [1, 0, 1, 0, 1, 0] * 50 # 300개 샘플
grammar_labels = [1, 1, 0, 0, 1, 1] * 50 # 조동사 일치, 수 일치 등
topic_labels = [0, 0, 1, 1, 0, 0] * 50 # 정치 vs 스포츠

sentiment_by_layer = layerwise_probe_analysis(probe, sentences,
↪  sentiment_labels, max_layer=24)
```

```
grammar_by_layer = layerwise_probe_analysis(probe, sentences,
↪  grammar_labels, max_layer=24)
topic_by_layer = layerwise_probe_analysis(probe, sentences,
↪  topic_labels, max_layer=24)
```

이 결과를 시각화하면 모델의 레이어가 어떻게 서로 다른 종류의 정보를 처리하고 있는지를 한눈에 파악할 수 있다.

##oppy-probe와 Intervention Test

Linear Probe는 분류 성능을 중심으로 하지만, 더 강력한 분석 방법으로 oppy-probe와 Intervention Test가 있다.

oppy-probe는 Probe 분류기의 결정 경계에 해당하는 방향 벡터 자체를 추출한다. LogisticRegression의 경우 coef_ 속성이 바로 결정 경계에 수직인 방향 벡터이다. 이 벡터를 원래의 hidden state에 더하면 Probe가"긍정적"으로 분류하는 방향으로 표현을 이동시킬 수 있다.

```
defoppy_intervention(probe, hidden_state, strength=0.5):
 """oppy-probe 방향으로 hidden state 이동"""
 direction = probe.coef_[0] # (hidden_dim,)
 direction_norm = direction / np.linalg.norm(direction)
 # 원래 방향에서_probe 방향으로 이동
 new_state = hidden_state + strength * direction_norm *
↪  np.linalg.norm(hidden_state)
 return new_state
```

Intervention Test의 논리는 더 간결하다. 먼저 특정 레이어의 hidden state를 추출한다. 그 다음 해당 hidden state의 특정 좌표 값을 강제로 변경하고(예: 해당 개념의 평균값으로 설정) 모델의 최종 출력이 어떻게 변하는지를 확인한다. 출력이 변화했다면 그 좌표가 해당 개념의 처리에 관여하고 있는 것이고, 변화가 없다면 관련이 희박한 것이다.

## 4.4 프로덕션 적용: 모델의 지식 지도 구축

Probe 분석의 가장 실질적인 프로덕션 적용은 모델의 지식 지도를 구축하는 것이다. 조직 내부 데이터를 학습한 모델이 특정 사실을"어디에" 저장하고 있는지를 추적하면, 모델이 잘못된 정보를 학습했거나 편향된 표현을 형성했을 때 정확히 어느 레이어, 어느 표현 좌표에서 문제가 발생했는지를 알 수 있다.

구체적인 절차는 다음과 같다. 먼저 모델에게 질문할 사실 목록을 정의한다. 예: "파리는 프랑스의 수도이다", "스위스는 EU 회원국이 아니다" 등. 각 질문에 대한 모델의 답변이나 참답변과 연결된 입력을 수집한다. 그리고 여러 레이어에서 Probe를 학습해서 각 사실이 어떤 레이어에서 가장 잘 분리되는지를 분석한다.

이 정보가 유용하면, 모델이 잘못된 사실을 출력할 때 어느 레이어를 확인하고 수정해야 하는지를 추론할 수 있다. 또한 훈련 데이터의 특정 영역에서만 발생하는 편향이 특정 레이어에 집중되어 있다면, 그 레이어의 표현만 선택적으로 debiasing하는 것이 가능해진다.

## 4.5 한계와 주의사항

Probe 분석에도 분명한 한계가 있다. 높은 분류 정확도가 그 개념이 분명하게 분해되어 있다는 직접적인 증거가 되지는 않는다. 분류기가 표현의 선형 조합을 학습했을 가능성도 있기 때문이다. 또한 Probe로 찾은 좌표와 실제 모델 내부의 계산 메커니즘 사이의 관계를 단정할 수 없다.

그래서 Probe 결과를 해석할 때는 항상"상관관계이지 인과관계가 아니다"는점을 명심해야 한다. 특정 레이어의Probe가 높은 성능을 보인다고해서 반드시 그 레이어가 해당 개념을"처리"하고 있다는 것은 아니다. 다른 레이어에서 먼저 처리가 이루어지고, 그 결과가 해당 레이어에 반영되었을 수도 있다.

Interpretation 결과를 바탕으로 모델을 직접 수정할 때는 반드시Intervention Test로 가설을 검증한 뒤에 실행해야 한다.

# 5 프로덕션 Interpretability 파이프라인 설계

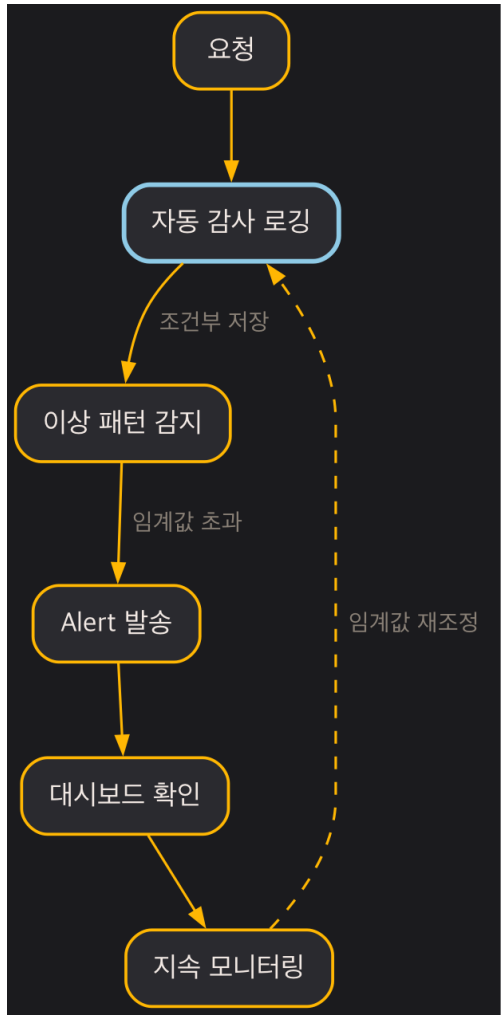


Figure 5.1: 프로덕션 Interpretability 파이프라인 설계

## 5.1 요구사항 정의: 무엇을 Interpretability 해야 하는가

Interpretability Engineering에 들어가기 전에 먼저 무엇을 알고 싶은지부터 분명히 정해야 한다. 목적은 대체로 세 가지 중 하나로 좁혀진다.

먼저 **디버깅**이다. 모델 출력이 기대와 다를 때 그 이유를 내부에서 추적하는 작업으로, 인시던트 대응 단계에서 가장 급하게 필요해진다.

**규제 대응**도 있다. 특정 의사결정이 어떤 이유로 내려졌는지를 외부에 설명해야 하는 경우로, 금융이나 의료, 고용처럼 위험도가 높은 영역에서 주로 요구된다.

마지막은 **신뢰 구축**이다. 모델이 왜 그런 출력을 내놓는지 내부 작동 메커니즘을 이해함으로써 운영자가 모델을 믿고 적절히 쓸 수 있게 만드는 것이다.

이 세 가지 목적은 상호 배타적이지 않다. 같은 도구와 파이프라인으로 동시에 달성할 수 있지만, 우선순위를 어디에 두느냐에 따라 수집하는 데이터의 종류와 분석 깊이가 달라진다.

## 5.2 도구 선택: 오픈소스 생태계 활용

Interpretability 도구 생태계는 빠르게 성숙하고 있으며, 크게 세 가지 카테고리로 나뉜다.

**Circuit-level 도구**는 모델 내부의 계산 그래프를 직접 추적한다. TransformerLens, NNSIS, ELLVM이 대표적이다. 매우 낮은 수준에서 작동하지만 대규모 모델에서는 계산량이 엄청나고 분석 결과를 해석하기 어려운 경우가 많다.

**Representation 도구**는 모델의 분산 표현을 통계적으로 분석한다. TransformerLens의 Logit Lens, Linear Probe, Steering Vector 추출 기법이 여기 속한다. Circuit-level 도구보다 실무에 적용하기 쉽고 결과 해석도 수월한 편이다.

**Behavioral 도구**는 입력과 출력 쌍을 체계적으로 측정해 모델의 행동 프로파일을 구축한다. Automated red-teaming, behavioral test suite, drift detection이 대표적이다. 구현이 가장 쉬우면서도 프로덕션 모니터링과 바로 통합할 수 있다는 장점이 있다.

실무에서는 세 도구를 모두 활용하되 목적에 따라 접근 방식을 달리한다. 긴급한 디버깅이라면 Behavioral 도구로 문제를 재현하고 대략적인 위치를 잡은 뒤, Representation 도구로 근본 원인을 파고든다. Circuit-level 도구는 메커니즘 이해가 꼭 필요할 때만 제한적으로 쓴다.

## 5.3 자동 감사 로깅 구조

프로덕션 Interpretability의 핵심 인프라 중 하나는 자동 감사 로깅이다. 요청이 들어올 때마다 모델의 내부 상태를 선택적으로 저장하는 구조가 필요하다.

저장해야 할 최소 정보는 요청의 입력 텍스트와 토큰화 결과, 요청 시각과 요청자 식별 정보(필요한 경우), 최종 출력 토큰열과 그 확률분포, 그리고 핵심 attention pattern과 hidden state다.

모든 요청마다 모든 내부 상태를 저장하면 비용이 막대해지므로, 필요한 정보만 골라 저장하는 편이 현실적이다. 목적은 문제가 발생한 시점을 분석하는 데 있으므로 평소에는 핵심 메트릭만 남기고, 출력 확률분포가 평소와 크게 다르거나 특정 단어가 포함된 입력처럼 특정 조건을 만족할 때만 상세 내부 상태를 저장하는 조건부 수집 방식이 효과적이다.

```
import json
from datetime import datetime
from collections import deque
import threading

class InterpretabilityLogger:
 def __init__(self, storage_path,
 ↪ attention_storage_threshold=0.3):
 self.storage_path = storage_path
 self.attention_threshold = attention_storage_threshold
 self.recent_summaries = deque(maxlen=1000) # 경량 메타데이터만
 ↪ 저장
 self.lock = threading.Lock()

 def log_request(self, request_id, prompt, output,
 ↪ attention_patterns=None,
 hidden_states=None, log_probs=None):
 """감사 로그 기록"""
 summary = {
 "request_id": request_id,
 "timestamp": datetime.utcnow().isoformat(),
 "prompt_hash": hash(prompt) % 10**6,
 "output_length": len(output),
 "avg_log_prob": sum(log_probs) / len(log_probs) if log_probs
 ↪ else None,
 "max_prob": max(log_probs) if log_probs else None,
 "min_prob": min(log_probs) if log_probs else None,
 }

 # 비정상적으로 낮은 확률의 토큰이 있으면 상세 로그 저장
 is_anomalous = log_probs and min(log_probs) < -5.0
```

```
with self.lock:
self.recent_summaries.append(summary)

if is_anomalous or attention_patterns is not None:
detailed_log = {
**summary,
"prompt": prompt,
"output": output,
"attention_patterns": attention_patterns,
"log_probs": log_probs,
}
self._write_detailed_log(detailed_log)

def _write_detailed_log(self, log_data):
"""상세 로그를 파일에 기록"""
filepath = f"{self.storage_path}/{log_data['request_id']}.json"
with open(filepath, "w") as f:
json.dump(log_data, f, ensure_ascii=False)

def get_anomaly_summary(self, lookback=100):
"""최근 로그에서 비정상 패턴 통계"""
recent = list(self.recent_summaries)[-lookback:]
anomaly_count = sum(1 for s in recent if s.get("min_prob", 0) <
↪  -5.0)
return {
"total_requests": len(recent),
"anomalous_requests": anomaly_count,
"anomaly_rate": anomaly_count / lookback if lookback > 0 else 0
}
```

이 구조의 핵심은 비정상 패턴을 자동으로 감지해 상세 로그를 선별적으로 남기는 데 있다. 매 요청마다 모든 내부 상태를 저장하지 않고도 문제가

발생한 시점의 상세 데이터는 항상 확보해 둘 수 있다.

## 5.4 임계값 설정과 Alert 설계

Interpretability 데이터가 쌓이고 나면 다음 단계는 이상을 감지해 Alert을 보내는 일이다. Alert 설계에서 가장 중요한 부분은 임계값을 어떻게 설정하느냐다.

임계값을 정할 때 고려해야 할 변수가 몇 가지 있다. 먼저 모델의 기본 동작 분포다. 정상 동작에서 attention entropy 분포나 logit 차이의 분포를 먼저 파악해야 이상 감지의 기준선을 세울 수 있다. 또 하나는 Alert 빈도와 정밀도 사이의 트레이드오프다. 임계값을 낮게 잡으면 Alert이 더 많이 발생하지만 그중 진짜 문제인 비율은 낮아지고, 임계값을 높게 잡으면 Alert 수는 줄어드는 대신 진짜 문제를 놓칠 가능성이 커진다.

실무에서는 이렇게 접근하면 효과적이다. 먼저 최소 2주 분량의 정상 동작 데이터를 모아 각 메트릭의 분포를 파악한다. 그다음 평균에서 표준편차 3 이상 벗어났을 때 Alert가 발생하도록 기준을 잡는다. 처음에는 범위를 넉넉하게 잡고 시작해, 실제 Alert을 검토하면서 점진적으로 조정해 나가는 편이 좋다.

Alert이 발생했을 때 확인해야 할 항목도 미리 정해두어야 한다. 문제가 된 입력이 무엇이었는지, 정상 동작과의 차이가 어느 레이어에서 가장 큰지, 이전에 비슷한 패턴이 있었는지 등을 빠르게 확인할 수 있는 대시보드를 구성해두면 대응 시간을 크게 줄일 수 있다.

## 5.5 지속적 모니터링과 개선 사이클

Interpretability 파이프라인은 한번 구축한다고 끝나는 일이 아니다. 모델이 업데이트되거나 시간이 지나며 훈련 데이터 분포가 바뀔 때마다 내부 표현의 특성도 함께 변한다. 이런 변화는 지속적으로 관찰해야 한다.

모니터링할 항목으로는 레이어별 attention entropy의 추세, 정상 입력군에서 이상치로 감지되는 입력 비율의 변화, 특정 개념에 대한 Probe 정확도가 시간에 따라 어떻게 바뀌는지, 그리고 Steering Vector를 적용했을 때 출력 변화율이 얼마나 안정적인지를 들 수 있다.

이런 메트릭이 평소와 크게 달라졌다면 모델 업데이트나 훈련 데이터 파이프라인에 변화가 생겼을 가능성이 크다. 그런 변화를 미리 포착해 적절히 대응하도록 만드는 것이 프로덕션 Interpretability의 최종 목표다.