adaTT
기술 참조서

Gradient-Based Task Affinity · Transfer Loss · Multi-Phase Training

프로젝트

AIOps 추천 시스템

PLE-adaTT + Cluster Head

버전

v1.0

2026-03-05

이 문서는 PLE-Cluster-adaTT 아키텍처의 핵심 모듈인 adaTT (Adaptive Task Transfer)의 이론적 기초, 수학적 구조, 구현 세부사항, 학습 스케줄, 설계 선택의 근거를 상세히 기술한다. 버그 발생 시 작동 원리에 대한 이해를 통해 문제 상황을 유추할 수 있도록 돕는 것이 핵심 목표이다.

목차

0. 심층 기초 해설 — adaTT의 근본 원리와 직관

이 장에서는 adaTT(Adaptive Task-aware Transfer)의 설계를 뒷받침하는 근본적인 아이디어, 배경 지식, 수학적 메커니즘을 깊이 있게 해설한다. 이후 장에서 다루는 구현 세부사항을 왜 그렇게 만들었는가라는 관점에서 이해하기 위한 토대이다.

0.1 왜 "적응형 타워"인가 — 고정 타워의 한계에서 출발하는 이야기

Multi-Task Learning(MTL)에서 가장 단순한 아키텍처는 하나의 공유 백본 위에 태스크별 고정 타워(Task-specific Tower)를 올리는 구조이다. 이 구조는 명쾌하지만 근본적인 약점을 안고 있다.

> 고정 타워의 세 가지 한계
>
> 1. 일방적 공유: 공유 백본의 파라미터가 모든 태스크에 동일하게 영향을 미친다. CTR 최적화가 Churn 예측을 악화시켜도 이를 감지하거나 조절할 메커니즘이 없다.
> 2. 태스크 간 상호작용 무시: 16개 태스크가 공유 파라미터를 두고 암묵적 경쟁을 벌이지만, 어떤 태스크 쌍이 서로 돕고 어떤 쌍이 해치는지 전혀 측정하지 않는다.
> 3. 학습 단계별 변화 미반영: 학습 초기에는 CTR과 CVR이 비슷한 방향으로 학습되다가, 후반에는 전혀 다른 특성을 잡아낼 수 있다. 고정 가중치로는 이 변화를 추적할 수 없다.

adaTT는 이 세 가지 한계를 각각 다음과 같이 해결한다:

1. 선택적 전이: gradient 방향이 일치하는 태스크 쌍만 지식을 공유하고, 반대 방향이면 차단한다.
2. 태스크 친화도 측정: gradient cosine similarity로 태스크 간 관계를 정량적으로 측정한다.
3. 동적 적응: 3-Phase 스케줄로 학습 단계에 따라 전이 강도를 조절한다.

핵심 내러티브를 한 문장으로 요약하면: "고정된 타워는 태스크 간 간섭을 방관하지만, 적응형 타워는 간섭을 측정하고 제어한다."

0.2 Transformer와 Attention — 왜 이 메커니즘이 태스크 적응에 적합한가

adaTT가 "Adaptive Task-aware **Transfer**"라는 이름을 쓰지만, 내부적으로 활용하는 핵심 원리는 Attention 메커니즘의 사상(⌧⌧)과 깊이 연결되어 있다. 여기서 그 연결고리를 명확히 짚는다.

0.2.1 Self-Attention의 Query-Key-Value 원리

Transformer의 Self-Attention은 세 가지 역할로 입력을 분리한다:

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

$\boldsymbol{Q}$: Query — "나는 무엇을 찾고 있는가"
$\boldsymbol{K}$: Key — "나는 어떤 정보를 제공할 수 있는가"

V: Value — "실제로 전달할 정보"
d_k: Key 차원 (스케일링 팩터)

이 메커니즘의 핵심 통찰은 **"관련 있는 것에 집중하고, 관련 없는 것은 무시한다"**이다. $QK^\top$는 쿼리와 키 사이의 유사도를 계산하고, softmax는 이 유사도를 확률 분포로 변환하여 Value의 가중 합을 만든다.

> **학부 수학**
>
> 내적**(dot product)**과 사잇각의 관계. 두 벡터 $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$의 내적은 $\boldsymbol{a} \cdot \boldsymbol{b} = \sum_{k=1}^{n} a_k b_k$이다. 고등학교에서 배운 $\boldsymbol{a} \cdot \boldsymbol{b} = \|\boldsymbol{a}\|\,\|\boldsymbol{b}\| \cos\theta$와 연결하면, 내적 값이 크다는 것은 (1) 벡터가 길거나 (2) 사잇각 θ가 작다(방향이 비슷하다)는 뜻이다. Attention에서 $QK^\top$의 (i, j) 원소는 곧 i번 Query와 j번 Key의 내적이므로, 방향이 비슷한 **Key**에 높은 점수를 부여하는 구조이다. $\sqrt{d_k}$로 나누는 이유는 차원 d_k가 커질수록 내적 값의 분산이 d_k에 비례하여 증가하기 때문에, 이를 정규화하지 않으면 softmax 출력이 극단값에 몰리는 **gradient vanishing** 문제가 생기기 때문이다.

0.2.2 adaTT에서의 Attention 유비(類比)

adaTT는 토큰 간의 Self-Attention이 아니라 태스크 간의 **Attention**을 수행한다. 비유하면:

역할	Transformer Self-Attention	adaTT Task Transfer
Query	현재 토큰의 질의	현재 태스크의 gradient 방향
Key	다른 토큰의 응답 가능성	다른 태스크의 gradient 방향
유사도	$Q\frac{K^\top}{\sqrt{d_k}}$	gradient cosine similarity
확률화	softmax	softmax (temperature T)
Value	다른 토큰의 실제 정보	다른 태스크의 loss 값
출력	가중 합산된 context	전이 손실 (transfer loss)

즉, adaTT의 전이 가중치 계산은 본질적으로 태스크 공간에서의 **Attention**이다. "내 태스크(Query)와 gradient 방향이 비슷한(Key) 다른 태스크로부터 손실(Value)을 가져와 가중 합산한다."

0.3 조건부 계산과 Hypernetwork — adaTT의 계보

adaTT의 아이디어는 더 넓은 **Conditional Computation** 패러다임의 일부이다. 이 패러다임은 "입력이나 상황에 따라 네트워크의 동작을 동적으로 변경한다"는 사상이다.

0.3.1 Conditional Computation의 핵심 아이디어

> **역사적 배경**
>
> **Conditional Computation**의 기원. "입력에 따라 네트워크의 일부만 활성화한다"는 아이디어는 Bengio et al. (2013, *"Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation"*)에서 본격적으

로 제안되었다. 당시 목표는 계산 비용 절감이었다 — 모든 뉴런을 매번 활성화하는 대신, 게이트 함수로 필요한 부분만 선택하여 연산량을 줄이는 것이었다. 이 아이디어는 이후 Mixture of Experts (MoE, Shazeer et al., 2017)로 발전하여 Google의 Switch Transformer (Fedus et al., 2022)에서 1조 파라미터 규모로 확장되었다. adaTT는 "어떤 뉴런을 활성화할 것인가"가 아니라 "어떤 태스크의 지식을 전이할 것인가"를 조건부로 결정한다는 점에서, Conditional Computation의 태스크 수준 확장이라고 할 수 있다.

전통적 신경망은 모든 입력에 대해 동일한 가중치 $\boldsymbol{W}$를 적용한다:

$$\boldsymbol{y} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

Conditional Computation은 조건 $\boldsymbol{c}$에 따라 가중치 자체가 변한다:

$$\boldsymbol{y} = \boldsymbol{W}(\boldsymbol{c})\boldsymbol{x} + \boldsymbol{b}(\boldsymbol{c})$$

여기서 조건 $\boldsymbol{c}$는 태스크 ID, 입력 특성, 학습 단계 등 다양한 신호가 될 수 있다.

0.3.2 Hypernetwork와의 관계

Ha et al., 2017 의 Hypernetwork는 "네트워크가 다른 네트워크의 가중치를 생성한다"는 개념이다. adaTT는 이 아이디어의 경량화된 변형으로 볼 수 있다:

> 역사적 배경
>
> **Hypernetwork**의 탄생. Ha, Dai & Le (ICLR 2017, *"HyperNetworks"*)는 "작은 네트워크(hypernetwork)가 큰 네트워크(main network)의 가중치를 직접 생성한다"는 파격적인 아이디어를 제안했다. 영감의 원천은 생물학이었다 — 유전자(genotype)가 직접 행동을 결정하는 것이 아니라, 단백질 합성 경로를 거쳐 표현형(phenotype)을 만들어내는 것처럼, hypernetwork가 main network의 파라미터를 간접적으로 결정한다. 이 패러다임은 이후 Task-Conditioned HyperNetworks (von Oswald et al., NeurIPS 2020), LoRA (Hu et al., 2022)의 저랭크 적응 등으로 이어졌다. adaTT는 hypernetwork의 풀 가중치 생성 대신 gradient 유사도라는 관측 신호로 전이 가중치를 결정하여 파라미터 효율성을 극대화한 변형이다.

측면	**Hypernetwork**	**adaTT**
가중치 생성	별도 네트워크가 전체 가중치 생성	gradient 유사도가 전이 가중치 결정
조건 입력	태스크 임베딩 벡터	태스크별 gradient 벡터
파라미터 수	큰 (생성 네트워크 자체가 큰)	작은 (n^2 전이 행렬 + prior)
적응 속도	학습에 의존	EMA로 빠른 적응

adaTT의 핵심 장점은 **gradient** 자체를 조건 신호로 사용한다는 것이다. 태스크 임베딩 같은 별도의 학습 가능한 표현이 아니라, 현재 학습 상태에서 직접 관측되는 gradient 방향으로 태스크 관계를 판단한다. 이는 학습 단계에 따른 태스크 관계 변화를 지연 없이 반영할 수 있다.

0.3.3 태스크 임베딩이 파라미터 공간을 조절하는 메커니즘

일반적인 태스크 적응형 모델에서 태스크 임베딩 $\boldsymbol{e}_{\text{task}}$는 파라미터 공간에서 작동 영역을 선택하는 역할을 한다:

$$\boldsymbol{W}_{\text{effective}} = \boldsymbol{W}_{\text{shared}} + \boldsymbol{\Delta}(e_{\text{task}})$$

adaTT에서 이 역할을 하는 것이 전이 가중치 행렬 $\boldsymbol{R}$이다. 각 태스크 i에 대해 행 $\boldsymbol{R}_{i,:}$가 곧 "태스크 i의 관점에서 바라본 다른 태스크와의 관계"를 인코딩하며, 이것이 loss landscape에서의 이동 방향을 조절한다.

> 최신 동향
>
> **Task-specific Adaptation 최신 연구 (2024–2025).** 태스크 임베딩으로 파라미터 공간을 조절하는 아이디어는 최근 더욱 정교해지고 있다. (1) **Task Arithmetic (Ilharco et al., ICLR 2023)**: 파인튜닝된 모델들의 가중치 차이 벡터(task vector)를 산술적으로 더하고 빼서 모델 능력을 조합하는 방법을 제안했다. 예를 들어 "감정 분석 능력 + 번역 능력 - 유해 출력 능력" 같은 연산이 가능하다. (2) **TIES-Merging (Yadav et al., NeurIPS 2023)**: 다수의 task vector를 병합할 때 부호 충돌(sign conflict)을 해결하는 알고리즘을 제안했다. (3) **AdapterSoup (Chronopoulou et al., EACL 2023)**: 다수의 LoRA 어댑터를 가중 평균하여 새로운 태스크에 적응한다. adaTT의 전이 가중치 행렬 $\boldsymbol{R}$은 이러한 task vector 조합의 연속적이고 동적인 버전으로 해석할 수 있다.

수학적으로 볼 때, 전이 가중치가 loss에 미치는 영향은:

$$\nabla_\theta \mathcal{L}_i^{\text{adaTT}} = \nabla_\theta \mathcal{L}_i + \lambda \sum_{j \neq i} w_{i \to j} \nabla_\theta \mathcal{L}_j$$

이 식에서 $\lambda \sum_{j \neq i} w_{i \to j} \nabla_\theta \mathcal{L}_j$는 태스크 i의 파라미터 업데이트 방향을 수정하는 보정 벡터이다. 친화도가 높은 태스크 j의 gradient가 큰 가중치로 합산되어, 공유 파라미터가 양쪽 태스크 모두에게 유리한 방향으로 이동하게 만든다.

0.4 핵심 수식의 직관적 해석

이 절에서는 본 문서에 등장하는 핵심 수식들을 왜 이 형태인가의 관점에서 해설한다.

0.4.1 코사인 유사도 — 방향만 비교하는 이유

$$\cos(\theta_{i,j}) = \frac{\boldsymbol{g}_i \cdot \boldsymbol{g}_j}{\|\boldsymbol{g}_i\| \cdot \|\boldsymbol{g}_j\|}$$

직관: 두 태스크의 gradient를 고차원 공간의 화살표로 생각하자.

- 크기는 loss의 절대적 민감도를 나타낸다. CTR loss가 0.01이고 LTV loss가 1000이면 gradient 크기가 수만 배 다르다.
- 방향은 "어느 쪽으로 파라미터를 바꾸면 loss가 줄어드는가"를 나타낸다.

adaTT가 관심 있는 것은 방향이다. "두 태스크가 파라미터를 같은 방향으로 바꾸고 싶어하는가?" 코사인 유사도는 벡터의 크기를 정규화하여 순수한 방향 비교만 수행한다. 만약 유클리드 거리를 사용하면, gradient가 큰 태스크가 지배적으로 작용하여 실제 방향이 같은데도 "거리가 멀다"고 판단할 수 있다.

> 학부 수학

코사인 유사도의 기하학적 의미. 2차원 평면에서 벡터 $\boldsymbol{a} = (3, 4)$, $\boldsymbol{b} = (6, 8)$을 생각하자. 유클리드 거리는 $\|\boldsymbol{a} - \boldsymbol{b}\| = \sqrt{9 + 16} = 5$로 "멀다"고 판정한다. 그러나 두 벡터는 같은 방향이다. 코사인 유사도는 $\cos\theta = \frac{3 \cdot 6 + 4 \cdot 8}{5 \cdot 10} = \frac{50}{50} = 1.0$으로 "완전히 같은 방향"이라고 정확하게 판정한다. 기하학적으로 코사인 유사도는 두 벡터를 단위원(또는 고차원 단위구)에 사영(projection)한 뒤 사잇각을 측정하는 것과 같다. $\cos\theta = 1$이면 $\theta = 0°$ (같은 방향), $\cos\theta = 0$이면 $\theta = 90°$ (직교, 무관), $\cos\theta = -1$이면 $\theta = 180°$ (정반대 방향)이다. gradient의 크기는 loss의 절대적 스케일에 의존하지만, 방향은 "파라미터를 어느 쪽으로 바꿔야 loss가 줄어드는가"라는 본질적 정보를 담고 있으므로, 방향만 비교하는 코사인 유사도가 적합하다.

0.4.2 Softmax 정규화 — 확률 분포를 형성하는 이유

$$w_{i \to j} = \frac{\exp\left(\frac{\boldsymbol{R}_{i,j}}{T}\right)}{\sum_{k \neq i} \exp\left(\frac{\boldsymbol{R}_{i,k}}{T}\right)}$$

직관: 전이 가중치를 확률 분포로 만드는 것에는 세 가지 이유가 있다.

1. 합이 **1**: 전이 가중치의 합이 항상 1이므로, 태스크 수가 늘어나도 전이 손실의 스케일이 일정하다. 16개 태스크든 32개 태스크든, $\lambda = 0.1$의 의미가 변하지 않는다.
2. 경쟁적 선택: softmax의 특성상 하나의 가중치가 올라가면 나머지가 내려간다. 이는 가장 도움이 되는 태스크에 집중하는 효과를 낸다.
3. 미분 가능성: hard argmax와 달리 softmax는 연속적이고 미분 가능하여, 전이 가중치 $\boldsymbol{W}$ (학습 가능 파라미터)의 gradient-based 최적화가 가능하다.

Temperature $\boldsymbol{T}$의 역할을 온도계에 비유하면:

- T가 낮으면 (차가우면) 물질이 결정화되듯 하나의 태스크에 집중한다.
- T가 높으면 (뜨거우면) 물질이 기체가 되듯 모든 태스크에 고르게 분산된다.
- $T = 1.0$은 액체 상태 — 적당히 유동적이면서도 구조를 유지한다.

학부 수학

Softmax 함수의 수학적 해부. 입력 벡터 $\boldsymbol{z} = (z_1, z_2, z_3)$에 대해 softmax는 $\sigma(z_i) = \frac{e^{z_i}}{\sum_k} e^{z_k}$이다. 예를 들어 $\boldsymbol{z} = (2, 1, 0)$이면 $\sigma = \frac{e^2, e^1, e^0}{e^2 + e + 1} \approx \frac{7.39, 2.72, 1.0}{11.1} \approx (0.67, 0.24, 0.09)$이다. 이제 temperature T를 적용하면 $\sigma(\frac{z_i}{T})$가 된다. $T = 0.5$이면 입력이 $(4, 2, 0)$으로 **2**배 확대되어 $\approx (0.87, 0.12, 0.02)$으로 최대값에 더 집중하고, $T = 2.0$이면 입력이 $(1, 0.5, 0)$으로 축소되어 $\approx (0.42, 0.34, 0.24)$으로 거의 균등해진다. 수학적으로 $T \to 0^+$이면 softmax는 one-hot 벡터(argmax)에 수렴하고, $T \to \infty$이면 균등 분포 $\frac{1}{n}$에 수렴한다. 이 성질은 통계역학의 Boltzmann 분포 $p_i \propto e^{-\frac{E_i}{k_B T}}$에서 유래했으며, 여기서 T는 실제 물리적 온도이다.

0.4.3 EMA 평활화 — 기억과 망각의 균형

$$\boldsymbol{A}_t = \alpha \cdot \boldsymbol{A}_{t-1} + (1 - \alpha) \cdot \boldsymbol{C}_t$$

직관: EMA를 일종의 기억 시스템으로 생각하자.

- $\alpha = 0.9$는 "과거 기억의 90%를 유지하고, 새 관측의 10%만 반영한다"는 뜻이다.
- 이는 효과적 관측 창 **(effective window)** $\approx \frac{1}{1-\alpha} = 10$에 해당한다.
- 즉, 최근 10번의 관측을 가중 평균한 것과 비슷한 효과이다.

왜 단순 평균이 아닌 EMA인가?

- 단순 평균: 모든 과거 관측에 동일한 가중치를 준다. 학습 초기의 (의미 없는) gradient도 후반까지 영향을 미친다.
- **EMA**: 최근 관측에 더 큰 가중치를 준다. 태스크 간 관계가 학습 중에 변하면 이를 빠르게 반영한다.
- 이동 평균 **(sliding window)**: 정확한 윈도우 관리가 필요하고 메모리 비용이 크다. EMA는 스칼라 하나(α)만으로 동일한 효과를 달성한다.

> 학부 수학
>
> **EMA의 재귀 필터 해석.** EMA 공식 $A_t = \alpha A_{t-1} + (1-\alpha) C_t$를 반복 대입하면 $A_t = (1-\alpha)\sum_{k=0}^{t-1} \alpha^k C_{t-k} + \alpha^t A_0$ 이 된다. 각 과거 관측 C_{t-k}에 붙는 가중치는 $(1-\alpha)\alpha^k$로, k가 커질수록(과거로 갈수록) 기하급수적으로 감소한다. $\alpha = 0.9$이면 1 step 전의 가중치는 0.09, 5 step 전은 0.059, 20 step 전은 0.012, 50 step 전은 0.0005로 사실상 무시된다. 이는 신호처리에서 **IIR (Infinite Impulse Response) 1**차 저역통과 필터와 정확히 같은 구조이다. 전달 함수는 $H(z) = \frac{1-\alpha}{1-\alpha z^{-1}}$이며, 차단 주파수 $f_c \approx \frac{1-\alpha}{2\pi}$로 고주파 노이즈(배치별 gradient 변동)를 제거하고 저주파 추세(진짜 태스크 관계)만 통과시킨다. 메모리 관점에서도, sliding window 평균이 $O(W)$ 메모리가 필요한 반면 EMA는 현재 상태 A_t 하나만 저장하면 되어 $O(1)$ 메모리로 동작한다.

0.4.4 Transfer-Enhanced Loss — 다른 태스크로부터 배우기

$$\mathcal{L}_i^{\text{adaTT}} = \mathcal{L}_i + \lambda \cdot \sum_{j \neq i} w_{i \to j} \cdot \mathcal{L}_j$$

직관: 이 수식을 조언을 듣는 과정으로 비유하자.

- $\mathcal{L}_i$: 태스크 i 자신의 판단 (원본 loss)
- $\sum_{j \neq i} w_{i \to j} \cdot \mathcal{L}_j$: 다른 태스크들의 조언 (가중 합산)
- $\lambda = 0.1$: 조언을 얼마나 심각하게 받아들일지 (10% 반영)
- $w_{i \to j}$: 누구의 조언을 더 신뢰할지 (친화도 기반 가중치)

$\lambda = 0.1$이라는 보수적인 값은 "자기 판단의 **90%**를 유지하되, 동료의 조언을 **10%**만 반영한다"는 것이다. `max_transfer_ratio = 0.5`는 "아무리 좋은 조언이라도 자기 판단의 50%를 초과하여 따르지 않는다"는 안전장치이다.

0.4.5 Prior Blend — 경험과 데이터의 균형

$$\boldsymbol{R}_{\text{blended}} = (\boldsymbol{W} + \boldsymbol{A}) \cdot (1 - r) + \boldsymbol{P} \cdot r$$

직관: 이 수식은 경험 많은 선배**(Prior)**와 실제 데이터**(Affinity)**의 의견을 혼합하는 과정이다.

- 학습 초기 ($r = 0.5$): "데이터가 아직 부족하니, 선배의 의견(도메인 지식)을 절반 반영하자"
- 학습 후반 ($r = 0.1$): "이제 데이터가 충분히 쌓였으니, 실제 관측 결과를 90% 신뢰하자"

이는 Bayesian 추론의 핵심 원리와 정확히 일치한다: **prior**가 강할 때는 **prior**에 의존하고, 데이터가 충분해지면 **likelihood(관측)**를 따른다. Appendix A.2에서 이 대응 관계를 수학적으로 증명한다.

> 학부 수학
>
> **Bayesian** 추론의 기초 — 사전분포와 사후분포. Bayes 정리는 $P(\theta \mid D) = P(D \mid \theta) \cdot \frac{P(\theta)}{P(D)}$이다. 여기서 $P(\theta)$는 사전분포**(prior)** — 데이터를 보기 전의 믿음이고, $P(D \mid \theta)$는 우도**(likelihood)** — 파라미터 θ가 주어졌을 때 데이터가 관측될 확률이며, $P(\theta \mid D)$는 사후분포**(posterior)** — 데이터를 본 후 업데이트된 믿음이다. 동전 던지기를 예로 들자. 처음에 "앞면 확률이 0.5 근처"라는 prior를 갖고 있다가, 10번 던져 7번 앞면이 나오면 posterior가 0.5보다 높은 쪽으로 이동한다. 데이터가 1000번이면 posterior는 거의 관측 비율(0.7)에 수렴한다. adaTT에서 Group Prior $\boldsymbol{P}$가 "사전 믿음"이고, gradient 코사인 유사도가 "관측 데이터"이며, blend ratio r의 감소가 "데이터가 쌓이면 prior 의존도를 줄인다"는 Bayesian updating을 구현한다.

0.5 전체 내러티브 — "측정하고, 선택하고, 조절한다"

adaTT의 전체 작동 원리를 세 단어로 요약하면 측정**(Measure)**, 선택**(Select)**, 조절**(Regulate)**이다.

1. 측정: 매 N step마다 각 태스크의 gradient를 추출하고, 코사인 유사도로 태스크 간 친화도를 계산한다. EMA로 노이즈를 걸러내어 안정적인 친화도 행렬 $\boldsymbol{A}$를 유지한다.
2. 선택: 친화도 행렬과 Group Prior를 혼합하고, negative transfer를 차단한 뒤, softmax로 정규화하여 전이 가중치 $\boldsymbol{w}$를 결정한다. "누구에게 배울 것인가"를 데이터 기반으로 결정하는 것이다.
3. 조절: 3-Phase 스케줄로 전이의 강도와 시점을 제어한다. 학습 초기에는 관찰만 하고(Warmup), 중반에는 동적으로 전이하며(Dynamic), 후반에는 안정성을 위해 고정한다(Frozen).

이 세 단계가 결합되어, **16**개 태스크가 서로의 학습을 방해하지 않으면서도 상호 보완적으로 성장하는 생태계를 형성한다.

> 고정 타워 **vs** 적응형 타워**:** 핵심 차이
>
> 고정 타워: "모든 태스크가 같은 백본을 공유한다. 충돌이 나면 어쩔 수 없다."
> 적응형 타워**(adaTT)**: "모든 태스크가 같은 백본을 공유하되, 누가 누구에게 도움이 되는지 실시간으로 측정하고, 해로운 간섭은 차단하며, 유익한 지식만 선별적으로 전달한다."

1. adaTT 개요 및 설계 철학

핵심 문제: **Multi-Task Learning**의 **Negative Transfer**

다수의 태스크를 하나의 네트워크로 동시 학습하면 **Negative Transfer**가 발생할 수 있다. 태스크 A의 gradient가 태스크 B의 gradient와 반대 방향일 때, 공유 파라미터 업데이트가 한쪽 태스크의 성능을 향상시키면서 다른 쪽을 악화시키는 현상이다. 16개 태스크를 동시 학습하는 본 시스템에서는 이 문제가 특히 심각하다.

1.1 왜 Adaptive Task Transfer인가

본 프로젝트의 Multi-Task Learning (MTL) 아키텍처는 16개 활성 태스크를 동시에 학습한다. CTR, CVR, Churn, NBA 등 서로 다른 비즈니스 목표를 가진 태스크들이 Shared Expert 파라미터를 공유하기 때문에, 태스크 간 상호작용을 제어하지 않으면 학습이 불안정해진다.

전통적인 접근법은 다음과 같다:

- 고정 가중치 (Fixed Weighting): 각 태스크에 수동으로 가중치를 설정 Kendall et al., 2018
- **GradNorm**: gradient 크기 기반 동적 가중치 Chen et al., 2018
- **PCGrad**: gradient 충돌 시 사영(projection) Yu et al., 2020

이들은 태스크 간 유사도를 직접 측정하지 않는다. adaTT는 gradient cosine similarity로 태스크 간 친화도를 측정하고, 이를 기반으로 선택적 지식 전이를 수행한다.

역사적 배경

MTL에서 **Task-specific Adaptation**의 발전사. Multi-Task Learning의 개념은 Caruana (1997, *"Multitask Learning"*)로 거슬러 올라간다. 초기에는 단순 Hard Parameter Sharing(공유 백본 + 태스크별 헤드)이 주류였으나, 태스크 간 간섭 문제가 지속적으로 보고되었다. 이를 해결하기 위해 (1) Cross-Stitch Networks (Misra et al., CVPR 2016) — 태스크별 네트워크 출력을 선형 결합, (2) MTAN (Liu et al., CVPR 2019) — Attention으로 공유 피처에서 태스크별 피처 선택, (3) PLE (Tang et al., RecSys 2020) — Expert 단위로 공유/전용 분리 등이 발전해왔다. adaTT는 이 계보에서 **gradient** 자체를 태스크 관계 신호로 사용한다는 점에서 차별화된다.

최신 동향

2024–2025 Task-aware MTL 아키텍처 트렌드. 최근 MTL 연구는 크게 세 방향으로 진화하고 있다. (1) **Gradient Manipulation**: Nash-MTL (Navon et al., ICML 2022), Aligned-MTL (Senushkin et al., CVPR 2023) 등이 gradient를 파레토 최적 방향으로 사영하는 방법을 제안하며, CAGrad (Liu et al., NeurIPS 2021)는 최악 태스크의 gradient 방향을 보장한다. (2) **Task Routing**: Mod-Squad (Chen et al., NeurIPS 2023)은 각 태스크가 Expert 서브셋을 동적 선택하는 구조를 제안했고, adaTT의 친화도 기반 전이와 상보적이다. (3) **Foundation Model** 시대의 **MTL**: LoRA 기반 태스크별 어댑터 (Hu et al., 2022), MTLoRA (Agiza et al., 2024) 등은 거대 모델에서 파라미터 효율적 태스크 적응을 수행한다. adaTT의 gradient cosine similarity 기반 접근은 이러한 어댑터 라우팅에도 확장 가능하다.

1.2 핵심 아이디어

adaTT의 핵심은 세 가지로 요약된다:

1. **Gradient Cosine Similarity**: 두 태스크의 gradient 방향이 같으면 positive transfer, 반대면 negative transfer로 판정
2. **Group Prior**: 도메인 지식 기반의 태스크 그룹 구조를 prior로 활용하여 학습 초기 안정성 확보
3. **3-Phase Schedule**: Warmup → Dynamic → Frozen 단계로 친화도 측정 → 적용 → 고정

소스: `adatt.py` 전체 (477 lines) — `AdaptiveTaskTransfer` 클래스

그림 1: adaTT 3-Phase Schedule 개요

1.3 시스템 내 위치

adaTT는 PLE-Cluster-adaTT 아키텍처의 **forward pass** 후반에 위치한다. Shared Experts → CGC Gating → Task Experts → Task Towers 순서로 예측값이 생성된 후, 각 태스크의 손실이 계산되면 adaTT가 태스크 간 전이 손실을 추가한다.

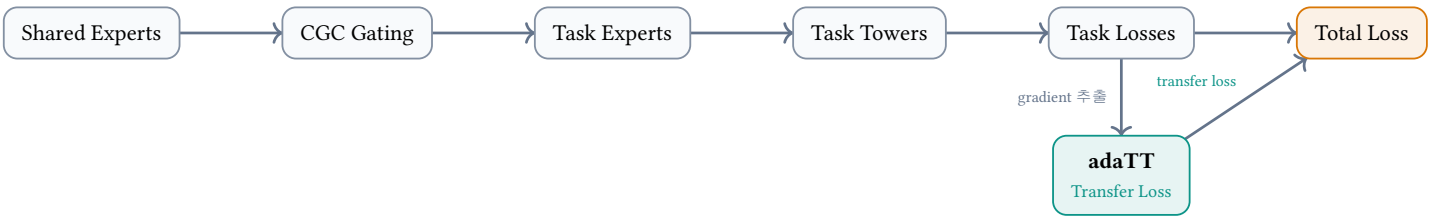

그림 2: adaTT의 forward pass 내 위치 (`ple_cluster_adatt.py:1290-1319`)

2. TaskAffinityComputer — 핵심 친화도 엔진

`TaskAffinityComputer`는 adaTT의 기반 모듈로, 태스크 간 gradient cosine similarity를 계산하고 EMA (Exponential Moving Average)로 안정화된 친화도 행렬을 유지한다.

소스: `adatt.py:58-143` — `TaskAffinityComputer` 클래스

2.1 클래스 구조와 초기화

```
# adatt.py:58-84
class TaskAffinityComputer(nn.Module):
    def __init__(self, task_names: List[str], ema_decay: float = 0.9):
        super().__init__()
        self.task_names = task_names
        self.n_tasks = len(task_names)
        self.ema_decay = ema_decay

        # EMA 친화도 행렬 (학습 안정성)
        self.register_buffer("affinity_matrix", torch.eye(self.n_tasks))
        self.register_buffer("update_count", torch.tensor(0))
```

> **설계 선택: register_buffer**
>
> `affinity_matrix`를 `register_buffer`로 등록한 이유는 두 가지다: (1) `state_dict`에 포함되어 체크포인트 저장/복원 시 자동 관리, (2) `.to(device)` 호출 시 자동으로 GPU/CPU 이동. `nn.Parameter`가 아닌 buffer로 등록하여 optimizer가 이 행렬을 학습 대상으로 취급하지 않는다.

핵심 상태:

버퍼	초기값	용도
`affinity_matrix`	`eye(n_tasks)`	EMA 평활된 태스크 간 친화도 행렬 [n×n]
`update_count`	`0`	EMA 업데이트 횟수 (첫 업데이트 시 EMA 스킵 판정)

2.2 compute_affinity() 메서드

이 메서드는 각 태스크의 flattened gradient 벡터를 받아 코사인 유사도 행렬을 계산하고, EMA로 기존 친화도와 혼합한다.

```
# adatt.py:86-138
def compute_affinity(self, task_gradients: Dict[str, torch.Tensor]) -> torch.Tensor:
    # 1. gradient 수집 (누락 태스크는 zero padding)
    grad_list = []
    for name in self.task_names:
```

```
        if name in task_gradients:
            grad_list.append(task_gradients[name])
        else:
            grad_list.append(torch.zeros_like(reference_grad))

    # 2. Stack: [n_tasks, grad_dim]
    grad_matrix = torch.stack(grad_list, dim=0)

    # 3. 코사인 유사도 행렬
    norms = grad_matrix.norm(dim=1, keepdim=True).clamp(min=1e-8)
    normalized = grad_matrix / norms
    affinity = torch.mm(normalized, normalized.t())

    # 4. EMA 업데이트
    with torch.no_grad():
        if self.update_count > 0:
            new_affinity = self.ema_decay * self.affinity_matrix \
                         + (1 - self.ema_decay) * affinity
            self.affinity_matrix.copy_(new_affinity.clamp(-1.0, 1.0))
        else:
            self.affinity_matrix.copy_(affinity.clamp(-1.0, 1.0))
        self.update_count.add_(1)
```

학부 수학

행렬 곱으로 모든 쌍의 코사인 유사도를 한 번에 계산하기. n개 태스크의 gradient를 행으로 쌓은 행렬 $\boldsymbol{G} \in \mathbb{R}^{n \times d}$를 생각하자. 각 행 $\boldsymbol{g}_i$를 L2 norm으로 나누면 정규화 행렬 $\hat{\boldsymbol{G}}$를 얻는다 ($\hat{\boldsymbol{g}}_i = \frac{\boldsymbol{g}_i}{\|\boldsymbol{g}_i\|}$). 이때 $\hat{\boldsymbol{G}}\hat{\boldsymbol{G}}^\top$의 (i, j) 원소는 $\sum_{k=1}^{d} \hat{g}_{i,k}\hat{g}_{j,k} = \hat{\boldsymbol{g}}_i \cdot \hat{\boldsymbol{g}}_j = \cos\theta_{i,j}$이다. 즉, 단 한 번의 행렬 곱으로 n^2개 코사인 유사도를 동시에 구할 수 있다. 만약 이중 for 루프로 쌍마다 계산하면 $O(n^2 d)$ 연산이 n^2번의 Python 호출로 분산되어 느리지만, 행렬 곱 `torch.mm`은 GPU의 병렬 연산 유닛(CUDA cores)을 활용하여 수백 배 빠르게 처리한다. 16개 태스크 기준으로 $16 \times 16 = 256$개의 유사도를 단일 GEMM (General Matrix Multiplication) 커널로 계산한다.

N-03 FIX: 부동소수점 오차 누적 방지

EMA 업데이트 후 반드시 `.clamp(-1.0, 1.0)`을 적용한다 (`adatt.py:132,135`). 수천 번의 EMA 누적 과정에서 부동소수점 오차가 쌓여 코사인 유사도가 $[-1, 1]$ 범위를 벗어나는 것을 방지한다. 이 clamping 없이 `arccos` 등 후속 연산을 수행하면 NaN이 발생할 수 있다.

2.3 친화도 행렬 해석

친화도 행렬 $\boldsymbol{A} \in [-1, 1]^{n \times n}$는 대칭 행렬이다 ($\boldsymbol{A}_{i,j} = \boldsymbol{A}_{j,i}$).

범위	의미	adaTT 행동
$\boldsymbol{A}_{i,j} \approx 1$	강한 양의 친화도	태스크 i, j의 gradient가 같은 방향 → 적극 전이
$\boldsymbol{A}_{i,j} \approx 0$	중립	태스크 간 연관 없음 → 약한 전이

$A_{i,j} < -0.1$	음의 친화도	Negative transfer 감지 → 전이 차단

대각 원소 $A_{i,i} = 1$은 자기 자신과의 코사인 유사도이므로 항상 1이다.

3. Gradient Cosine Similarity 수학적 기초

3.1 수학적 정의

두 태스크 i, j의 Shared Expert 파라미터 θ에 대한 gradient를 각각 $\boldsymbol{g}_i = \nabla_\theta \mathcal{L}_i$, $\boldsymbol{g}_j = \nabla_\theta \mathcal{L}_j$라 하면:

$$\cos(\theta_{i,j}) = \frac{\boldsymbol{g}_i \cdot \boldsymbol{g}_j}{\|\boldsymbol{g}_i\| \cdot \|\boldsymbol{g}_j\|}$$

$\boldsymbol{g}_i \in \mathbb{R}^d$: 태스크 i의 flattened gradient 벡터
d: Shared Expert 전체 파라미터 수
$\|\cdot\|$: L2 norm

구현에서는 행렬 연산으로 모든 태스크 쌍의 유사도를 한 번에 계산한다:

```
# adatt.py:122-125
norms = grad_matrix.norm(dim=1, keepdim=True).clamp(min=1e-8)
normalized = grad_matrix / norms
affinity = torch.mm(normalized, normalized.t())  # [n_tasks, n_tasks]
```

`clamp(min=1e-8)` : zero gradient 태스크의 0-division 방지

3.2 EMA 평활화

단일 배치의 gradient는 노이즈가 크므로, 시간에 걸쳐 안정화된 친화도를 얻기 위해 Exponential Moving Average를 적용한다:

$$\boldsymbol{A}_t = \alpha \cdot \boldsymbol{A}_{t-1} + (1 - \alpha) \cdot \cos(\theta_t)$$

$\alpha = 0.9$ (기본값, `ema_decay` 파라미터)
$\boldsymbol{A}_0$: 첫 번째 관측값을 그대로 사용 (EMA 초기화, `adatt.py:134-135`)

> 왜 $\alpha = 0.9$인가
>
> $\alpha = 0.9$이면 과거 10개 관측의 가중 평균에 근사한다 (effective window $\approx \frac{1}{1-\alpha} = 10$). 학습 초기에는 gradient가 불안정하므로 빠른 수렴보다 안정성을 우선하되, 태스크 관계가 epoch에 따라 변할 수 있으므로 너무 보수적이지 않은 값을 선택한다. `model_config.yaml:601` 에서 설정: `ema_decay: 0.9` .

3.3 왜 코사인 유사도인가 (vs. 유클리드 거리)

코사인 유사도를 사용하는 이유는 세 가지다:

1. 스케일 불변성: 태스크별 loss 규모가 다르면 gradient 크기도 다르다. 코사인 유사도는 방향만 비교하므로 이 차이에 영향받지 않는다.
2. 해석 용이성: [−1, 1] 범위로 정규화되어 "같은 방향 = positive transfer, 반대 방향 = negative transfer"라는 직관적 해석이 가능하다.
3. 효율적 계산: 정규화 후 행렬 곱 한 번으로 $O(n^2 d)$에 계산 완료.

Fifty et al., 2021 의 Task Affinity 연구에서도 gradient cosine similarity를 태스크 유사도 측정의 표준 지표로 사용한다.

> 최신 동향
>
> **Gradient** 기반 태스크 유사도 측정의 최신 연구 **(2023–2025).** 코사인 유사도 외에도 다양한 gradient 기반 지표가 연구되고 있다. (1) **TAG (Task Affinity Grouping, Fifty et al., ICML 2021)**: gradient 내적의 부호 변화 패턴으로 태스크 그룹핑을 자동화한다. (2) **Gradient Vaccine (Wang et al., ICLR 2021)**: gradient 충돌 시 코사인 유사도에 기반한 부분 사영을 적용하여 PCGrad보다 세밀한 제어를 달성한다. (3) **Conflict-Averse Gradient Descent (CAGrad, Liu et al., NeurIPS 2021)**: 코사인 유사도를 최소화 대상으로 삼아 모든 태스크의 gradient와 최소 각도를 최대화하는 공통 방향을 찾는다. (4) **Nash-MTL (Navon et al., ICML 2022)**: 태스크 간 경쟁을 Nash 협상 문제로 공식화하여 파레토 최적 gradient를 구한다. adaTT는 이 중에서 유사도 측정 + 선택적 전이를 결합한 접근으로, 측정과 활용을 분리한 모듈러 설계가 강점이다.

3.4 Gradient 추출 경로

gradient는 `ple_cluster_adatt.py`의 `_extract_task_gradients` 메서드에서 추출된다. 핵심은 **Shared Expert** 파라미터에 대해서만 gradient를 계산한다는 것이다.

```
# ple_cluster_adatt.py:1871-1903
shared_params = list(self.shared_experts.parameters())

for task_name, loss in eligible:
    grads = torch.autograd.grad(
        loss,
        shared_params,
        retain_graph=True,
        create_graph=False,
        allow_unused=True,
    )
    grad_flat = torch.cat([
        g.flatten() if g is not None else torch.zeros_like(p).flatten()
        for g, p in zip(grads, shared_params)
    ])
    task_gradients[task_name] = grad_flat
```

> **retain_graph=True**의 필수성
>
> `retain_graph=True`는 아키텍처상 제거 불가하다 (`ple_cluster_adatt.py:1865-1868`). `_extract_task_gradients`는 forward pass 도중 (loss 계산 후, `backward()` 전)에 호출되며, 동일한 computation

graph를 Trainer의 `loss.backward()`에서 재사용해야 한다. 여기서 graph를 해제하면 `backward()` 시 `RuntimeError: Trying to backward through the graph a second time`이 발생한다.

3.5 torch.compiler.disable 데코레이터

```python
# ple_cluster_adatt.py:1847
@torch.compiler.disable
def _extract_task_gradients(self, task_losses):
```

왜 **torch.compile**에서 제외하는가

`torch.autograd.grad`는 컴파일된 그래프 내에서 `requires_grad` 추적이 불완전하다. `torch.compiler.disable` 데코레이터로 이 메서드를 컴파일 그래프 밖에서 실행하여 gradient 추출이 정상 작동하도록 보장한다. 실제로는 `torch.compile` 자체가 비활성화되어 있지만 (`trainer.py:177`), 향후 활성화 시에도 안전하도록 방어적으로 적용한 것이다.

4. Transfer Loss 계산 메커니즘

`compute_transfer_loss()`는 adaTT의 핵심 메서드로, 각 태스크의 원본 손실에 다른 태스크로부터의 전이 손실을 가산한다.

소스: `adatt.py:283-353` — `compute_transfer_loss()` 메서드

4.1 전체 공식

각 태스크 i에 대한 Transfer-Enhanced Loss:

$$\mathcal{L}_i^{\text{adaTT}} = \mathcal{L}_i + \lambda \cdot \sum_{j \neq i} w_{i \to j} \cdot \mathcal{L}_j$$

$\mathcal{L}_i$: 태스크 i의 원본 손실 (focal, huber, MSE 등)
$\lambda = 0.1$ (기본값, `transfer_lambda` 파라미터)
$w_{i \to j}$: 태스크 i에 대한 태스크 j의 전이 가중치 (softmax 정규화)

> **수식 직관**
>
> 이 수식은 "각 태스크가 자기 자신의 loss만 보는 것이 아니라, 친화도가 높은 다른 태스크의 loss도 일부 참고한다"는 것을 말한다. $\lambda = 0.1$은 다른 태스크의 의견을 10%만 반영하겠다는 뜻이고, $w_{i \to j}$는 "누구의 의견을 더 들을 것인가"를 결정한다. 직관적으로, gradient 방향이 비슷한 태스크일수록 가중치가 높아져 서로의 학습을 가속시킨다.

4.2 전이 가중치 계산 상세

전이 가중치 $w_{i \to j}$는 여러 요소의 조합이다:

```python
# adatt.py:355-396 — _compute_transfer_weights()
raw_weights = self.transfer_weights + affinity  # 학습 가능 + 친화도

# Group Prior 결합 (annealing)
raw_weights = raw_weights * (1 - r) + self.group_prior * r

# Negative Transfer 차단
raw_weights = torch.where(
    affinity > self.neg_threshold,  # -0.1
    raw_weights,
    torch.zeros_like(raw_weights),
)

# 대각선 0 (자기 전이 제외)
raw_weights = raw_weights.masked_fill(self.diag_mask, 0.0)
```

```
# Softmax 정규화
weights = F.softmax(raw_weights / max(self.temperature, 1e-6), dim=-1)
```

수학적으로:

$$\boldsymbol{R} = (\boldsymbol{W} + \boldsymbol{A}) \cdot (1 - r) + \boldsymbol{P} \cdot r$$
$$\boldsymbol{R}_{i,j} \leftarrow 0 \quad \text{if } \boldsymbol{A}_{i,j} < \tau_{\text{neg}}$$
$$\boldsymbol{R}_{i,i} = 0$$
$$w_{i \to j} = \text{softmax}\left(\frac{\boldsymbol{R}_{i,j}}{T}\right)$$

$\boldsymbol{W}$: 학습 가능한 전이 가중치 (`nn.Parameter` , 초기값 0)
$\boldsymbol{A}$: EMA 친화도 행렬
$\boldsymbol{P}$: Group Prior 행렬
r: Prior blend ratio (Phase에 따라 변화)
$\tau_{\text{neg}} = -0.1$: Negative transfer 차단 임계값
$T = 1.0$: Softmax temperature

수식 직관

이 수식은 전이 가중치를 결정하는 4단계 파이프라인을 말한다. 먼저 학습 가능한 가중치 $\boldsymbol{W}$와 측정된 친화도 $\boldsymbol{A}$를 합산하고, 도메인 지식(Prior $\boldsymbol{P}$)과 혼합한다. 그런 다음 "해로운 전이"를 0으로 차단하고, 자기 자신은 제외한 뒤, softmax로 확률 분포를 만든다. 직관적으로, "데이터에서 관측한 태스크 관계 + 도메인 전문가의 사전 지식"을 결합하되, 해로운 경로는 끊어버리는 과정이다.

4.3 G-01 FIX: Transfer Loss Clamp

Transfer loss가 원본 loss를 지배하지 않도록 비율 제한이 적용된다:

```
# adatt.py:346-351
raw_transfer = self.transfer_lambda * transfer_loss
if self.max_transfer_ratio > 0:
    max_val = original_loss.detach() * self.max_transfer_ratio
    raw_transfer = torch.clamp(raw_transfer, max=max_val)
enhanced_losses[task_name] = original_loss + raw_transfer
```

max_transfer_ratio = 0.5

Transfer loss가 원본 loss의 **50%**를 초과할 수 없다 (`adatt.py:191`). 이 제한 없이 학습하면, 특정 태스크의 loss가 매우 작을 때 transfer loss가 상대적으로 과도하게 커져 학습 방향이 왜곡된다. `original_loss.detach()` 를 사용하여 clamp 경계값이 gradient에 영향을 주지 않는다.

4.4 Target 미존재 태스크 마스킹

모든 배치에 모든 태스크의 target이 있는 것은 아니다. Target이 없는 태스크는 전이 가중치에서 제외된다:

```
# adatt.py:321-334
loss_list = []
loss_mask = []
for name in self.task_names:
    if name in task_losses:
        loss_list.append(task_losses[name])
        loss_mask.append(1.0)
    else:
        loss_list.append(torch.tensor(0.0, device=affinity.device, requires_grad=False))
        loss_mask.append(0.0)

# ...
masked_transfer_w = transfer_w[i] * loss_mask_tensor
transfer_loss = (masked_transfer_w * loss_tensor).sum()
```

> **왜 0이 아닌 마스크를 사용하는가**
>
> 단순히 0.0 loss를 넣으면 softmax 후 해당 가중치가 0이 되지 않는다. `loss_mask_tensor` 로 곱하여 해당 전이 경로를 완전히 차단한다. 이는 배치별로 target이 가변적인 실환경(특히 비활성 태스크 존재 시)에서 안전하다.

5. Group Prior 구조

Group Prior는 도메인 지식을 수학적 prior로 인코딩한 행렬이다. 학습 초기에 태스크 간 친화도가 충분히 측정되기 전까지, 합리적인 전이 방향을 제공한다.

소스: `adatt.py:256-281` — `_build_group_prior()` 메서드

5.1 태스크 그룹 정의

`model_config.yaml:611-628`에서 4개 그룹이 정의된다:

그룹	멤버	intra 강도	비즈니스 의미
engagement	ctr, cvr, engagement, uplift	0.8	고객 참여/전환 관련
lifecycle	churn, retention, life_stage, ltv	0.7	고객 생애주기 관련
value	balance_util, channel, timing	0.6	고객 가치/행동 패턴
consumption	nba, spending_category, consumption_cycle, spending_bucket, merchant_affinity, brand_prediction	0.7	소비 패턴 분석

`inter_group_strength: 0.3` — 그룹 간 전이 강도는 낮게 유지

5.2 Prior 행렬 구성

```python
# adatt.py:256-281
def _build_group_prior(self) -> torch.Tensor:
    # 1. 그룹 간 전이 강도로 초기화
    prior = torch.ones(self.n_tasks, self.n_tasks) * self.inter_group_strength  # 0.3

    # 2. 그룹 내 전이 강도 설정
    for group_name, members in self.task_groups.items():
        strength = self.intra_group_strength.get(group_name, 0.5)
        indices = [self.task_names.index(m) for m in members if m in self.task_names]
        for i in indices:
            for j in indices:
                if i != j:
                    prior[i, j] = strength

    # 3. 대각선 = 0 (자기 자신 전이 없음)
    prior.fill_diagonal_(0.0)

    # 4. 행 정규화
    row_sums = prior.sum(dim=1, keepdim=True).clamp(min=1e-8)
    prior = prior / row_sums
```

행 정규화의 의미

행 정규화는 각 태스크 i가 다른 태스크로부터 받는 전이 가중치의 합을 1로 만든다. 이는 softmax와 유사한 효과를 가지며, 태스크 수에 무관하게 전이 강도를 일관되게 유지한다.

5.3 Prior Blend Annealing

Prior blend ratio r은 학습 진행에 따라 선형적으로 감소한다:

$$r(e) = r_{\text{start}} - (r_{\text{start}} - r_{\text{end}}) \cdot \min\left(\frac{e - e_{\text{warmup}}}{e_{\text{freeze}} - e_{\text{warmup}}}, 1.0\right)$$

$r_{\text{start}} = 0.5$: 학습 초기 prior 비율 (`prior_blend_start` , `model_config.yaml:607`)
$r_{\text{end}} = 0.1$: 학습 후반 prior 비율 (`prior_blend_end` , `model_config.yaml:608`)
e_{warmup}: warmup 종료 에포크
e_{freeze}: freeze 시작 에포크

수식 직관

이 수식은 "학습이 진행될수록 도메인 전문가의 사전 지식($\boldsymbol{P}$)에 대한 의존도를 줄이고, 실제 데이터에서 관측한 친화도를 더 신뢰한다"는 것을 말한다. r이 0.5에서 0.1로 감소하면, Prior 비중이 50%에서 10%로 줄어들고 관측 데이터 비중이 50%에서 90%로 늘어난다. 직관적으로, 신입 사원이 처음에는 선배 의견(Prior)에 의존하다가 경험이 쌓이면 자기 판단(관측)을 더 믿는 것과 같다.

이 annealing은 **Bayesian** 관점에서 prior에서 posterior로의 전환으로 해석할 수 있다: 학습 초기에는 데이터가 부족하므로 domain knowledge (prior)에 의존하고, 데이터가 누적될수록 학습된 gradient 기반 친화도 (likelihood)를 신뢰한다.

역사적 배경

Bayesian Weight 개념의 기원. Prior와 데이터를 혼합하는 아이디어는 Thomas Bayes (1763, 사후 출판)와 Pierre-Simon Laplace (1812, *Theorie analytique des probabilites*)로 거슬러 올라간다. 신경망에 Bayesian 관점을 도입한 것은 MacKay (1992, *"A Practical Bayesian Framework for Backpropagation Networks"*)와 Neal (1996, *"Bayesian Learning for Neural Networks"*)이 선구자이다. 현대 딥러닝에서는 Dropout이 근사적 Bayesian inference로 해석되고 (Gal & Ghahramani, ICML 2016), Weight Uncertainty (Blundell et al., ICML 2015, *"Bayes by Backprop"*)이 가중치의 불확실성을 직접 학습한다. adaTT의 Prior Blend Annealing은 이러한 Bayesian 전통의 실용적 경량화이다 — 풀 Bayesian posterior를 추론하는 대신, blend ratio r 하나로 prior-to-posterior 전환을 모사한다.

그림 3: Prior Blend Annealing 스케줄

6. 3-Phase adaTT Schedule

adaTT는 학습을 세 단계로 구분하여 친화도 측정과 전이를 제어한다.

소스: `adatt.py:298-313` — `compute_transfer_loss()` 내 Phase 분기

> **역사적 배경**
>
> 학습 스케줄링의 계보. "학습을 단계별로 나누어 진행한다"는 아이디어는 Bengio et al. (2009, *"Curriculum Learning"*)에서 체계화되었다 — 쉬운 예제부터 어려운 예제 순으로 학습하면 최종 성능이 향상된다는 발견이다. 이 아이디어는 (1) Pre-training + Fine-tuning (Erhan et al., 2010), (2) Layer-wise Training (Hinton et al., 2006, Deep Belief Networks), (3) Warmup-then-Decay 학습률 스케줄 (Goyal et al., 2017) 등으로 확장되었다. adaTT의 3-Phase(Warmup → Dynamic → Frozen)는 이 전통을 태스크 간 전이에 적용한 것이다: Phase 1에서 태스크 관계를 관찰하고(curriculum의 "탐색" 단계), Phase 2에서 관계를 활용하며(학습 본체), Phase 3에서 안정화한다(fine-tuning의 "고정" 단계).

6.1 Phase 1: Warmup (친화도 측정만)

```python
# adatt.py:300-304
if epoch < self.warmup_epochs:
    if task_gradients is not None:
        self.affinity_computer.compute_affinity(task_gradients)
    return task_losses  # 원본 손실 그대로 반환
```

Phase 1에서는 gradient cosine similarity를 계산하여 친화도 행렬을 축적하되, 전이 손실은 추가하지 않는다. 원본 `task_losses`를 변경 없이 반환한다.

- 기간: epoch 0 `warmup_epochs` (프로덕션 기준 10, 테스트용 0)
- 목적: 충분한 친화도 데이터 축적 없이 전이를 시작하면 random transfer로 학습이 불안정해진다
- 설정: `model_config.yaml:598` — `warmup_epochs: 0` (테스트), 프로덕션 권장 10

6.2 Phase 2: Dynamic Transfer (동적 전이)

```python
# adatt.py:311-317
# Phase 2: 동적 전이
if task_gradients is not None:
    self.affinity_computer.compute_affinity(task_gradients)

affinity = self.affinity_computer.get_affinity_matrix()
transfer_w = self._compute_transfer_weights(affinity)
```

Phase 2에서는 매 step마다 친화도를 업데이트하면서 동시에 전이 손실을 적용한다. Prior blend ratio r이 `prior_blend_start`에서 `prior_blend_end`로 선형 감소한다.

- 기간: `warmup_epochs` ~ `freeze_epoch`
- 학습 가능 파라미터: `self.transfer_weights` (`nn.Parameter`, `adatt.py:229-231`)

- **Prior blend annealing**: r 값이 0.5에서 0.1로 감소 (`adatt.py:373-379`)

6.3 Phase 3: Frozen (가중치 고정)

```
# adatt.py:307-308
if self.is_frozen:
    return self._apply_frozen_transfer(task_losses)
```

Phase 3에서는 전이 가중치를 고정하고 더 이상 gradient를 계산하지 않는다. `_apply_frozen_transfer` 에서 `transfer_w[i].detach()` 를 사용한다 (`adatt.py:425`).

- 기간: `freeze_epoch` 학습 종료
- 설정: `model_config.yaml:599` — `freeze_epoch: 1` (테스트), 프로덕션 권장 28
- 효과: gradient 계산 오버헤드 제거, 학습 안정화

> **H-6 검증: freeze_epoch > warmup_epochs**
>
> `adatt.py:219-223` 에서 `freeze_epoch <= warmup_epochs` 이면 `ValueError` 를 발생시킨다. Phase 2가 완전히 스킵되면 학습된 친화도가 전혀 전이에 반영되지 않으므로, adaTT를 사용하는 의미가 없어진다. 이 검증은 설정 오류를 조기에 차단한다.

6.4 Phase 전환 트리거: on_epoch_end

```
# adatt.py:431-452
def on_epoch_end(self, epoch: int) -> None:
    self.current_epoch.fill_(epoch)

    if self.freeze_epoch is not None and epoch >= self.freeze_epoch:
        if not self.is_frozen.item():
            self.is_frozen.fill_(True)
            logger.info(f"adaTT: 전이 가중치 고정 (epoch {epoch})")
```

> **fill_() 사용 이유**
>
> `self.current_epoch = epoch` 처럼 plain tensor를 재할당하면 `register_buffer` 로 등록된 buffer와의 연결이 끊어진다. `fill_()` 은 in-place 업데이트로 `state_dict` , device 관리를 유지한다. `is_frozen` 도 마찬가지로 `fill_(True)` 를 사용한다 (`adatt.py:441`).

7. Negative Transfer 감지 및 차단

7.1 Negative Transfer란 무엇인가

두 태스크의 gradient가 반대 방향을 가리킬 때, 공유 파라미터의 업데이트가 한 태스크의 성능을 개선하면서 다른 태스크의 성능을 저하시키는 현상이다.

예를 들어 CTR (클릭률)과 Churn (이탈률)의 gradient가 반대 방향이라면, CTR을 개선하는 방향으로 학습할 때 Churn 예측 성능이 악화된다.

7.2 차단 메커니즘

`_compute_transfer_weights()`에서 친화도가 임계값 이하인 전이 경로를 0으로 차단한다:

```
# adatt.py:383-388
raw_weights = torch.where(
    affinity > self.neg_threshold,     # -0.1 (기본값)
    raw_weights,                       # 유지
    torch.zeros_like(raw_weights),     # 차단
)
```

$$R_{i,j} = \begin{cases} R_{i,j} & \text{if } A_{i,j} > \tau_{\text{neg}} \\ 0 & \text{otherwise} \end{cases}$$

$\tau_{\text{neg}} = -0.1$ (`negative_transfer_threshold` , `model_config.yaml:600`)

수식 직관

이 수식은 일종의 게이트이다. 친화도가 임계값보다 높으면 전이 가중치를 그대로 통과시키고, 낮으면 0으로 완전 차단한다. 직관적으로, "나와 반대 방향으로 가는 태스크의 조언은 아예 듣지 않는다"는 안전장치이다.

왜 **-0.1**이 임계값인가 **(0**이 아닌**)**

코사인 유사도 0은 "직교" (무관)를 의미하며, 약간의 음의 상관도 노이즈일 수 있다. −0.1로 설정하여 약한 음의 상관은 허용하고, 명확한 반대 방향 gradient만 차단한다. 임계값이 0이면 너무 많은 전이 경로가 차단되어 adaTT의 효과가 약화된다.

최신 동향

Negative Transfer 완화 기법의 최신 진화 **(2023–2025).** Negative transfer 문제는 MTL의 핵심 난제로 활발히 연구되고 있다. (1) **Aligned-MTL (Senushkin et al., CVPR 2023)**: gradient를 공통 하강 방향(common descent direction)

으로 정렬하되, 태스크별 기여도를 보존하는 정교한 사영 기법을 제안했다. (2) **ForkMerge (Ye et al., NeurIPS 2023)**: 학습 도중 태스크를 동적으로 분리(fork)했다가 병합(merge)하여 negative transfer 구간을 자동 감지하고 회피한다. (3) **Auto-Lambda (Liu et al., ICLR 2022)**: 검증 셋에서의 meta-gradient로 태스크 가중치를 자동 조절하여 negative transfer를 간접적으로 완화한다. adaTT의 임계값 기반 차단($\tau_{\text{neg}} = -0.1$)은 이들에 비해 계산 비용이 극히 낮으면서도 효과적이라는 점에서 실무 친화적이다. 향후 개선 방향으로는 적응적 임계값(τ_{neg}를 학습 단계에 따라 조절)이나 soft gating(binary 차단 대신 연속적 감쇠)을 고려할 수 있다.

7.3 Negative Transfer 진단 API

```python
# adatt.py:460-476
def detect_negative_transfer(self) -> Dict[str, List[str]]:
    affinity = self.affinity_computer.get_affinity_matrix()
    negative_pairs = {}
    for i, task_i in enumerate(self.task_names):
        neg_list = []
        for j, task_j in enumerate(self.task_names):
            if i != j and affinity[i, j] < self.neg_threshold:
                neg_list.append(task_j)
        if neg_list:
            negative_pairs[task_i] = neg_list
    return negative_pairs
```

반환 예시: `{"churn": ["ctr", "engagement"], "ltv": ["brand_prediction"]}` 형태로 어떤 태스크 쌍에서 negative transfer가 감지되었는지 확인할 수 있다.

7.4 차단이 학습에 미치는 영향

상황	효과
차단 미적용	Negative transfer가 있는 태스크 쌍이 서로의 loss를 증가시켜 학습 불안정
과도한 차단 ($\tau_{\text{neg}} = 0$)	대부분의 전이 경로 차단 → adaTT가 사실상 비활성화
적절한 차단 ($\tau_{\text{neg}} = -0.1$)	명확한 negative transfer만 차단, 중립/양성 전이는 유지

8. 2-Phase Training Loop

Trainer는 전체 학습을 두 개의 Phase로 구분한다. adaTT는 **Phase 1**에서만 활성화되며, Phase 2에서는 비활성화된다.

소스: `trainer.py:456-542` — `train()` 메서드, `trainer.py:580-654` — `_train_phase()`

8.1 Phase 1: Shared Expert Pretrain

- 기간: `shared_expert_epochs` (기본 15, `model_config.yaml:793`)
- 학습 대상: 전체 모델 — Shared Experts, CGC, Task Experts, Task Towers
- **adaTT**: 활성 상태 — gradient 추출 및 transfer loss 적용

```
# trainer.py:483-488
logger.info("=== Phase 1: Shared Expert 학습 ===")
self._train_phase(
    train_loader, val_loader,
    max_epochs=self.config.shared_expert_epochs,
    phase_name="phase1",
)
```

Phase 1이 끝나면 adaTT는 충분한 친화도 데이터를 축적한 상태이다. `freeze_epoch`이 Phase 1 내에 있으면 Phase 1 후반부에서 이미 가중치가 고정된다.

8.2 Phase 2: Cluster Finetune

- 기간: `cluster_finetune_epochs` (기본 8, `model_config.yaml:794`)
- 학습 대상: 클러스터별 Task Expert 서브헤드만
- **adaTT**: 비활성화 — Shared Expert가 frozen이므로 gradient 추출 무의미

```
# trainer.py:493-496
_adatt_backup = self.model.adatt
if self.model.adatt is not None:
    self.model.adatt = None
    logger.info("adaTT 비활성화: Phase 2 (Shared frozen으로 친화도 무효)")
```

> 왜 **Phase 2**에서 **adaTT**를 비활성화하는가
>
> adaTT의 gradient는 **Shared Expert** 파라미터에 대해 계산된다 (`ple_cluster_adatt.py:1872`). Phase 2에서 Shared Expert가 frozen이면 gradient가 0이 되므로 코사인 유사도 계산이 무의미하다. 또한 frozen 파라미터에 대한 `autograd.grad` 호출은 불필요한 계산 오버헤드이다.

8.3 Phase 전환 시 리셋

`_setup_phase2()` (`trainer.py:544-578`)에서 다음 항목이 리셋된다:

리셋 항목	이유
Optimizer	Shared Expert frozen → 모멘텀 초기화 필요 (stale momentum 방지)
Scheduler	Phase 2 전용 warmup (2 에포크, Phase 1의 5 에포크보다 짧음)
GradScaler	AMP 스케일러 상태 초기화 (Phase 전환 시 loss 스케일 변화)
Early stopping	best_val_loss, patience_counter 모두 초기화
CGC Attention	Shared Expert frozen → CGC gating도 함께 freeze

```
# trainer.py:544-578
def _setup_phase2(self):
    if self.config.freeze_shared_in_phase2:
        self.model.freeze_shared_experts()
    # Optimizer 리셋
    self.optimizer = self._create_optimizer()
    # Scheduler 리셋 (Phase 2 전용 warmup)
    self.config.warmup_steps = self.config.phase2_warmup_steps  # 2 에포크
    self.scheduler = self._create_scheduler()
    # Early stopping 리셋
    self.best_val_loss = float("inf")
    self.patience_counter = 0
```

8.4 adaTT 복원 보장

Phase 2 종료 후 adaTT는 반드시 복원된다 (체크포인트/추론 호환성):

```
# trainer.py:504-508
finally:
    self.model.adatt = _adatt_backup  # 예외 발생 시에도 복원
    if self.config.freeze_shared_in_phase2:
        self.model.unfreeze_shared_experts()
```

`finally` 블록으로 감싸 예외 발생 시에도 모델 상태가 일관되게 유지된다.

9. Loss Weighting 전략

adaTT의 transfer loss는 기존 loss weighting 전략 위에 추가적으로 동작한다. 즉, 태스크별 loss weight와 uncertainty weighting이 먼저 적용되고, 그 결과에 adaTT의 transfer loss가 가산된다.

소스: `ple_cluster_adatt.py:1607-1845` — `_compute_task_losses()`

9.1 Loss 계산 파이프라인

전체 loss 계산 경로:

1. 태스크별 **loss** 유형 결정: focal, huber, MSE, NLL, contrastive 등 (`ple_cluster_adatt.py:1656`)
2. **Focal Loss alpha/gamma** 적용: 태스크별 차별화된 양성 클래스 가중치 (`ple_cluster_adatt.py:1768-1780`)
3. **Loss weight** 적용: 태스크별 고정 가중치 또는 Uncertainty Weighting (`ple_cluster_adatt.py:1818-1830`)
4. **Evidential loss** 가산: 불확실성 추정 보조 손실 (`ple_cluster_adatt.py:1832-1841`)
5. **adaTT transfer loss**: gradient 기반 전이 손실 추가 (`ple_cluster_adatt.py:1310-1316`)
6. **CGC entropy regularization**: Expert collapse 방지 (`ple_cluster_adatt.py:1321-1329`)

9.2 Uncertainty Weighting

Kendall et al., 2018 의 방법을 구현한다:

> 역사적 배경
>
> **Uncertainty Weighting**의 탄생. Kendall, Gal & Cipolla (CVPR 2018, *"Multi-Task Learning Using Uncertainty to Weigh Losses for Scene Understanding"*)는 컴퓨터 비전에서 세만틱 분할 + 깊이 추정 + 인스턴스 분할을 동시 학습하는 문제에서, 태스크별 가중치를 수동 튜닝하는 비용을 **homoscedastic uncertainty**로 자동화하는 방법을 제안했다. 이 아이디어의 핵심은 가우시안 likelihood $p(y \mid f(x), \sigma) = \mathcal{N}(f(x), \sigma^2)$에서 $-\log p$를 취하면 자연스럽게 $\frac{1}{2\sigma^2} \cdot \|y - f(x)\|^2 + \log \sigma$ 형태가 되어, **loss**가 큰 태스크의 $\boldsymbol{\sigma}$가 커지고 가중치가 줄어드는 자기 조절 메커니즘이 만들어진다는 것이다. 이 방법은 이후 추천 시스템 MTL에서도 표준 기법으로 자리잡았다.

> 학부 수학
>
> 왜 $\frac{1}{2\sigma^2}$ 형태가 나오는가 — 가우시안 로그-우도에서의 유도. 관측값 y가 예측 $\hat{y}$를 중심으로 한 정규분포 $\mathcal{N}(\hat{y}, \sigma^2)$를 따른다고 가정하자. 확률밀도함수는 $p(y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y-\hat{y})^2}{2\sigma^2}\right)$이다. 음의 로그를 취하면 $-\log p(y) = \frac{(y-\hat{y})^2}{2\sigma^2} + \log \sigma + \text{const}$가 된다. 여기서 $(y-\hat{y})^2$이 loss $\mathcal{L}$에 해당하므로, $\mathcal{L}^{\text{weighted}} = \frac{\mathcal{L}}{2\sigma^2} + \log \sigma$가 자연스럽게 도출된다. $\sigma^2 = \exp(\log_\text{var})$로 재파라미터화하면 $\log \sigma = \frac{1}{2} \cdot \log_\text{var}$이므로 코드의 구현과 일치한다. 정밀도 $\frac{1}{\sigma^2}$가 곧 loss의 가중치가 되며, σ가 무한히 커져 loss를 0으로 만드는 것을 $\log \sigma$ 정규화 항이 방지한다.

$$\mathcal{L}_i^{\text{weighted}} = \frac{1}{2\sigma_i^2} \cdot \mathcal{L}_i + \frac{1}{2} \log \sigma_i^2$$

$\sigma_i^2 = \exp(\log\_\text{var}_i)$: 태스크 i의 학습 가능한 불확실성
`log_var` clamp: [−4.0, 4.0], precision clamp: [0.001, 100.0] (`ple_cluster_adatt.py:1822-1823`)

수식 직관

이 수식은 "불확실성이 높은 태스크의 loss는 낮은 가중치로, 불확실성이 낮은 태스크의 loss는 높은 가중치로 반영한다"는 것을 말한다. $\frac{1}{2\sigma_i^2}$는 정밀도(precision)로, σ_i^2가 크면 가중치가 작아진다. $\frac{1}{2}\log\sigma_i^2$ 항은 σ_i^2를 무한히 키워 loss를 0으로 만드는 치팅을 방지하는 정규화 페널티이다. 직관적으로, "잘 모르는 태스크의 실수에는 관대하되, 잘 아는 태스크의 실수에는 엄격하게" 학습하는 전략이다.

이 가중치는 adaTT 이전에 적용된다. 즉, adaTT의 `task_losses` 입력에는 이미 uncertainty weighting이 반영된 값이 들어온다.

9.3 태스크별 Loss Weight 현황

`model_config.yaml`에서 정의된 태스크별 고정 가중치:

태스크	weight	loss type	비고
ctr	1.0	focal (γ=2, α=0.25)	표준
cvr	1.5	focal (γ=2, α=0.20)	양성 비율 극소 → weight 상향
churn	1.2	focal (γ=2, α=0.60)	FN 비용 높음 → alpha 상향
retention	1.0	focal (γ=2, α=0.20)	양성 비율 높음
nba	2.0	CE	12 classes, 비즈니스 핵심
ltv	1.5	huber (δ=1.0)	regression, 이상치 대응
brand_prediction	2.0	contrastive	InfoNCE, 50K 브랜드
spending_category	1.2	CE	12 categories
나머지	0.8–1.0	다양	태스크별 상이

9.4 adaTT와 Loss Weight의 상호작용

adaTT의 `compute_transfer_loss`는 각 태스크의 loss weight가 이미 적용된 후의 `task_losses`를 입력으로 받는다. 따라서 높은 loss weight를 가진 태스크(nba: 2.0)가 다른 태스크에 더 큰 전이 효과를 미친다.

이는 의도된 동작이다: 비즈니스적으로 중요한 태스크의 학습 시그널이 다른 태스크에도 전파되어야 한다.

10. Optimizer 및 Scheduler 설정

소스: `trainer.py:220-334` — `_create_optimizer()`, `_create_scheduler()`

10.1 AdamW Optimizer

```
# trainer.py:236-242
trainable_params = [p for p in self.model.parameters() if p.requires_grad]
return torch.optim.AdamW(
    trainable_params,
    lr=self.config.learning_rate,      # 0.0005
    weight_decay=self.config.weight_decay,  # 0.01
)
```

파라미터	기본값	설명
`learning_rate`	0.0005	전역 학습률 (`model_config.yaml:752`)
`weight_decay`	0.01	L2 정규화 강도
`gradient_clip_norm`	5.0	Gradient 크기 제한 (`model_config.yaml:780`)
`gradient_accumulation_steps`	1	실질 배치 = 16384 × 1

10.2 Per-Expert Learning Rate

Shared Expert마다 다른 학습률을 설정할 수 있다:

```
# trainer.py:249-261
for expert_name, expert_module in self.model.shared_experts.items():
    expert_params = [p for p in expert_module.parameters() if p.requires_grad]
    cfg = expert_lr_config.get(expert_name, {})
    lr = cfg.get("lr", self.config.learning_rate)
    wd = cfg.get("weight_decay", self.config.weight_decay)
    param_groups.append({"params": expert_params, "lr": lr, "weight_decay": wd})
```

이 기능은 `model_config.yaml:756-770`에 주석 처리된 예시로 제공되며, 하이퍼볼릭 공간에서 학습하는 `unified_hgcn`은 보수적인 lr이 필요하고, `deepfm`은 상대적으로 높은 lr로 빠르게 수렴할 수 있다.

> **Phase 2에서의 자동 제외**
>
> Phase 2에서 Shared Expert가 frozen되면 `requires_grad=False` 가 되므로, `_create_optimizer` 에서 해당 파라미터가 자동으로 제외된다 (`trainer.py:250`). 이는 불필요한 optimizer state 메모리 할당을 방지한다.

10.3 Learning Rate Scheduler

SequentialLR: Linear Warmup → CosineAnnealingWarmRestarts

```
# trainer.py:296-318
warmup_scheduler = torch.optim.lr_scheduler.LinearLR(
    self.optimizer, start_factor=0.1, total_iters=warmup_steps  # 5 에포크
)
cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(
    self.optimizer, T_0=self.config.cosine_t0,    # 10
    T_mult=self.config.cosine_t_mult,              # 2
)
return torch.optim.lr_scheduler.SequentialLR(
    self.optimizer,
    schedulers=[warmup_scheduler, cosine_scheduler],
    milestones=[warmup_steps],  # 5 에포크 후 cosine으로 전환
)
```

파라미터	기본값	설명
`warmup_steps`	5	Linear warmup 기간 (에포크 단위)
`cosine_t0`	10	첫 cosine 주기 길이 (`model_config.yaml:773`)
`cosine_t_mult`	2	주기 배수 (10 → 20 → 40 에포크)
`start_factor`	0.1	Warmup 시작 LR = 0.0005 × 0.1 = 0.00005

10.4 Phase 2 전용 Scheduler

Phase 2 시작 시 scheduler가 리셋되며, warmup은 2 에포크로 짧아진다:

```
# trainer.py:562-566
original_warmup = self.config.warmup_steps
self.config.warmup_steps = self.config.phase2_warmup_steps  # 2
self.scheduler = self._create_scheduler()
self.config.warmup_steps = original_warmup  # 복원
```

11. CGC-adaTT 동기화

CGC (Customized Gate Control)와 adaTT는 서로 다른 메커니즘이지만, 동일한 Shared Expert 파라미터에 대해 작동하므로 동기화가 필수적이다.

소스: `ple_cluster_adatt.py:1921-1942` — `on_epoch_end()` CGC freeze 동기화

11.1 CGC의 역할

CGC는 각 태스크가 어떤 Shared Expert에 더 attention을 줄지 학습한다. adaTT가 태스크 간 지식 전이를 제어한다면, CGC는 **Expert** 선택을 제어한다. 두 메커니즘이 독립적으로 학습하면 상충하는 방향으로 파라미터를 업데이트할 수 있다.

11.2 동기화 전략: 동시 Freeze

adaTT의 `freeze_epoch`에서 CGC Attention도 함께 frozen된다:

```
# ple_cluster_adatt.py:1931-1942
# v2.3: CGC freeze -- adaTT freeze_epoch과 동기화
freeze_epoch = self.config.get("adatt", {}).get("freeze_epoch")
if (freeze_epoch is not None
        and epoch >= freeze_epoch
        and self.task_expert_attention is not None
        and not self._cgc_frozen.item()):
    for param in self.task_expert_attention.parameters():
        param.requires_grad = False
    self._cgc_frozen.fill_(True)
    logger.info(f"CGC Attention frozen at epoch {epoch}")
```

> 왜 동시에 **freeze**하는가
>
> adaTT가 전이 가중치를 고정했는데 CGC가 계속 학습하면, CGC가 Expert 가중치를 변경하여 adaTT가 측정한 친화도 관계가 무효화된다. 예를 들어 adaTT가 "CTR→CVR positive transfer"라고 판단했는데, CGC가 CTR의 Expert 선택을 변경하면 gradient 방향이 달라져 고정된 전이 가중치가 더 이상 유효하지 않게 된다.

11.3 Phase 2에서의 CGC Freeze

Phase 2 시작 시에도 CGC가 freeze된다 (`trainer.py:549-555`):

```
# trainer.py:549-555
if hasattr(self.model, '_cgc_frozen') and not self.model._cgc_frozen.item():
    if self.model.task_expert_attention is not None:
        for param in self.model.task_expert_attention.parameters():
            param.requires_grad = False
```

```
        self.model._cgc_frozen.fill_(True)
        logger.info("CGC Attention frozen in Phase 2")
```

Shared Expert가 frozen인 Phase 2에서 CGC gating을 학습하는 것은 무의미하다: 입력(Expert 출력)이 변하지 않으므로 gating 학습이 과적합을 유발한다.

11.4 _cgc_frozen 상태 추적

```
# ple_cluster_adatt.py:376
self.register_buffer("_cgc_frozen", torch.tensor(False))
```

`register_buffer`로 등록하여 체크포인트 저장/복원 시 freeze 상태가 유지된다. MLflow에서도 freeze 이벤트를 로깅한다 (`trainer.py:604-608`).

12. 메모리 및 성능 최적화

adaTT의 gradient 추출은 계산 비용이 높다. 16개 태스크 각각에 대해 Shared Expert 파라미터의 gradient를 계산하므로, 최적화 없이는 학습 속도가 크게 저하된다.

12.1 retain_graph=True의 메모리 영향

```
# ple_cluster_adatt.py:1865-1868
# G-04 NOTE: retain_graph=True는 아키텍처상 필수
# 메모리 영향: n_tasks x shared_param_size.
# 16개 태스크 기준 forward pass 대비 약 2x peak memory.
```

`retain_graph=True`는 각 태스크의 `autograd.grad` 호출 후에도 computation graph를 메모리에 유지한다. 16개 태스크에 대해 순차적으로 호출하므로, peak memory는 forward pass 대비 약 **2**배로 증가한다.

요소	메모리	비고
Forward pass graph	1x (기준)	일반적인 학습에서도 필요
retain_graph 추가	1x	graph가 해제되지 않아 추가 메모리
16 task gradients	0.3x	각 gradient는 shared_param_size
합계	**2.3x**	RTX 4070 12GB에서 batch_size 16384 가능

12.2 adatt_grad_interval

gradient 추출 빈도를 줄여 계산 비용을 낮추는 핵심 최적화:

```
# ple_cluster_adatt.py:373
self.adatt_grad_interval = adatt_config.get("grad_interval", 10)

# ple_cluster_adatt.py:1299-1304
if self.training and self.adatt is not None:
    if self.global_step % self.adatt_grad_interval == 0:
        task_gradients = self._extract_task_gradients(task_losses)
```

> **기본값 10의 근거**
>
> 매 step마다 gradient를 추출하면 16 × `autograd.grad` 호출이 발생한다. 친화도는 EMA로 평활화되므로, 10 step 간격으로 측정해도 충분히 안정적이다. `grad_interval=10` 으로 설정하면 gradient 계산 오버헤드가 **1/10**로 감소한다. 기존에는 warmup 중 매 step 추출하여 hang이 발생했었다 (`ple_cluster_adatt.py:1300-1301`).

12.3 torch.compiler.disable

```
# ple_cluster_adatt.py:1847
@torch.compiler.disable
def _extract_task_gradients(self, task_losses):
```

현재 프로젝트에서 `torch.compile`은 비활성화되어 있다 (`trainer.py:174-177`). 15-태스크 MTL + adaTT `retain_graph` + dynamic shape 조합으로 커널 컴파일 수가 수백 개에 달해 첫 epoch에 **30**분 이상 소요되기 때문이다. 대신 TF32 + cuDNN benchmark로 10 15% 속도를 확보한다:

```
# trainer.py:170-172
torch.backends.cuda.matmul.allow_tf32 = True
torch.backends.cudnn.allow_tf32 = True
torch.backends.cudnn.benchmark = True
```

12.4 AMP (Automatic Mixed Precision)

Forward pass는 `torch.amp.autocast` 하에서 fp16으로 실행된다:

```
# trainer.py:709-711
with autocast(device_type=self.device.type):
    outputs = self.model(inputs, compute_loss=True)
    loss = outputs.total_loss / self.config.gradient_accumulation_steps
```

adaTT의 gradient 추출은 autocast 내에서 이루어지지만, focal loss 계산은 float32로 명시적으로 캐스팅한다 (`ple_cluster_adatt.py:1774`):

```
# ple_cluster_adatt.py:1774
p_f = pred.squeeze().float().clamp(1e-7, 1 - 1e-7)
```

> **M-2 + M-3 FIX: fp16 focal loss 안전성**
>
> fp16에서 `log(1e-7) = -16.1`은 정상이지만, `focal_weight * bce`의 중간 결과가 subnormal 범위에 들어가면 NaN이 발생할 수 있다. 전체 focal loss 계산을 float32로 수행하여 이 문제를 방지한다.

12.5 Gradient Accumulation

```
# trainer.py:723-730
if (batch_idx + 1) % self.config.gradient_accumulation_steps == 0:
    self.scaler.unscale_(self.optimizer)
    torch.nn.utils.clip_grad_norm_(
        self.model.parameters(), self.config.gradient_clip_norm  # 5.0
    )
    self.scaler.step(self.optimizer)
```

```
    self.scaler.update()
    self.optimizer.zero_grad()
```

현재 `gradient_accumulation_steps=1`이므로 매 배치마다 업데이트된다. 실질 배치 크기 = $16384 \times 1 = 16384$.

13. 디버깅 가이드

13.1 일반적인 실패 모드

증상	원인	해결
NaN loss 발생	fp16 focal loss underflow	M-2/M-3: float32 캐스팅 확인
학습 초기 loss 발산	transfer loss가 원본 지배	G-01: `max_transfer_ratio` 확인 (0.5)
Phase 2에서 RuntimeError	adaTT 비활성화 누락	trainer.py에서 `model.adatt = None` 확인
`ValueError` 초기화 실패	`freeze_epoch <= warmup_epochs`	H-6: config 검증
학습 hang (무응답)	매 step gradient 추출	`adatt_grad_interval` 설정 확인 (기본 10)
체크포인트 불일치	`fill_()` 미사용	buffer 업데이트 시 in-place 연산 확인

13.2 친화도 행렬 해석

학습 중 친화도 행렬을 확인하는 방법:

```python
# 친화도 행렬 조회
affinity = model.adatt.affinity_computer.get_affinity_matrix()
print(affinity)  # [n_tasks, n_tasks]

# Negative transfer 감지
neg_pairs = model.adatt.detect_negative_transfer()
print(neg_pairs)  # {"churn": ["ctr"], ...}

# 현재 전이 가중치 행렬
transfer_w = model.adatt.get_transfer_matrix()
print(transfer_w)  # softmax 정규화된 [n_tasks, n_tasks]
```

건강한 친화도 행렬의 특성:

- 같은 그룹 내 태스크 간 양의 친화도 (> 0.3)
- 다른 그룹 간 약한 양 또는 중립 ($-0.1 \sim 0.3$)
- 대각선 원소 = 1.0
- 전체 행렬이 **-1** 또는 **+1**로 포화되지 않음 (포화 시 EMA 감쇠율 조정 필요)

13.3 Transfer Loss 모니터링

MLflow에서 로깅되는 핵심 메트릭:

```
# trainer.py:810-811 — step 로깅 시 outputs.task_losses 포함
self._log_step(outputs, phase_name)
```

모니터링 체크리스트:

- `task_losses/<task>` : 태스크별 enhanced loss (transfer loss 포함)
- `adatt_freeze_epoch` : freeze 시점 로깅
- `cgc_frozen_epoch` : CGC freeze 시점 로깅
- 총 loss = $\sum$ enhanced losses + CGC entropy loss + SAE loss + evidential loss

13.4 Phase 전환 디버깅

Phase 전환 시 확인 사항:

1. **Phase 1 → Freeze**: `adaTT: 전이 가중치 고정 (epoch N)` 로그 확인
2. **Phase 1 → Phase 2**: `adaTT 비활성화: Phase 2` 로그 확인
3. **CGC Freeze**: `CGC Attention frozen at epoch N` 로그 확인
4. **Optimizer** 리셋: `Optimizer 리셋: Phase 2 시작` 로그 확인

```
# Phase 전환 로그 예시
# INFO: adaTT: 전이 가중치 고정 (epoch 28)
# INFO: CGC Attention frozen at epoch 28 (synced with adaTT freeze_epoch)
# INFO: === Phase 2: Cluster Head Fine-tuning ===
# INFO: Shared Experts 동결
# INFO: adaTT 비활성화: Phase 2 (Shared frozen으로 친화도 무효)
# INFO: Optimizer 리셋: Phase 2 시작
# INFO: 스케줄러 리셋: Phase 2 시작 (warmup=2 에포크)
```

13.5 Loss 폭발 방지

Loss가 급격히 증가하는 경우의 진단 순서:

1. **NaN** 확인: `_train_epoch` 에서 `math.isfinite(loss_val)` 체크 (`trainer.py:781-786`)
2. **Transfer loss** 비율 확인: `max_transfer_ratio=0.5` 초과 여부
3. **Gradient norm** 확인: `gradient_clip_norm=5.0` 적용 여부
4. **VRAM OOM** 확인: `trainer.py:762-775` 에서 OOM 발생 시 배치 스킵 처리

```
# trainer.py:781-786
loss_val = outputs.total_loss.item()
if not math.isfinite(loss_val):
    logger.error(f"NaN/Inf loss at batch {batch_idx}!")
    self.optimizer.zero_grad()  # gradient 오염 방지
    continue
```

14. 설정 매개변수 총람

14.1 adaTT 핵심 파라미터

파라미터	기본값	범위	설명
`enabled`	`true`	bool	adaTT 활성화 여부
`transfer_lambda`	0.1	[0, 1]	전이 손실 가중치 λ
`temperature`	1.0	$(0, \infty)$	Softmax temperature T
`warmup_epochs`	0 (test) 10 (prod)	[0, max_epochs)	Phase 1 기간
`freeze_epoch`	1 (test) 28 (prod)	(warmup, max_epochs)	Phase 3 시작
`negative_transfer_threshold`	−0.1	[−1, 0]	Negative transfer 차단 임계값
`ema_decay`	0.9	[0, 1]	친화도 EMA 감쇠 계수
`prior_blend_start`	0.5	[0, 1]	학습 초기 group prior 비율
`prior_blend_end`	0.1	[0, 1]	학습 후반 group prior 비율
`transfer_strength`	0.5	[0, 1]	로짓 전이 강도 (CTR→CVR 등)

소스: `model_config.yaml:593-628`

14.2 태스크 그룹 파라미터

파라미터	기본값	범위	설명
`task_groups.<group>.members`	–	list	그룹 소속 태스크 이름 리스트
`task_groups.<group>.intra_strength`	0.5	[0, 1]	그룹 내 전이 강도
`inter_group_strength`	0.3	[0, 1]	그룹 간 전이 강도

14.3 Training 파라미터

파라미터	기본값	범위	설명
`batch_size`	16384	[1024, 65536]	학습 배치 크기
`learning_rate`	0.0005	[1e-5, 1e-2]	전역 학습률

`weight_decay`	0.01	[0, 0.1]	L2 정규화 강도
`shared_expert_epochs`	15	[5, 100]	Phase 1 기간
`cluster_finetune_epochs`	8	[3, 50]	Phase 2 기간
`freeze_shared_in_phase2`	`true`	bool	Phase 2에서 Shared Expert 동결
`early_stopping_patience`	7	[1, 20]	Early stopping patience
`gradient_clip_norm`	5.0	[0.1, 20]	Gradient clipping 임계값
`use_amp`	`true`	bool	Mixed Precision (fp16) 사용
`cosine_t0`	10	[5, 50]	CosineAnnealing 첫 주기 (에포크)
`cosine_t_mult`	2	[1, 4]	CosineAnnealing 주기 배수
`warmup_steps`	5	[0, 20]	LR warmup 에포크 수
`phase2_warmup_steps`	2	[0, 10]	Phase 2 LR warmup 에포크 수

소스: `model_config.yaml:750-805`, `trainer.py:50-96`

14.4 성능 최적화 파라미터

파라미터	기본값	설명
`adatt_grad_interval`	10	gradient 추출 간격 (step 단위). 작을수록 정확, 클수록 빠름
`gradient_accumulation_steps`	1	gradient 누적 횟수. 실질 배치 = batch_size × 이 값
`gradient_checkpointing`	`false`	9.25M 모델에 불필요 (끄면 10 20% 속도 향상)
`use_amp`	`true`	fp16 Mixed Precision. 메모리 40% 절감, 속도 20% 향상

14.5 adaTT 내부 상수

코드에 하드코딩된 상수로, config에서 변경할 수 없다:

상수	값	위치 및 설명
`max_transfer_ratio`	0.5	`adatt.py:191` — transfer loss의 원본 대비 최대 비율
`norm clamp min`	1e-8	`adatt.py:123` — gradient norm 0-division 방지
`diag_mask`	eye(n)	`adatt.py:239-242` — 자기 전이 제외용 마스크
`affinity_matrix 초기값`	eye(n)	`adatt.py:79` — 대각선 1, 나머지 0 (중립 시작)

Appendix A. 수학 증명 및 이론적 근거

A.1 EMA 친화도의 수렴 특성

EMA 업데이트 규칙:

$$\boldsymbol{A}_t = \alpha \boldsymbol{A}_{t-1} + (1-\alpha)\boldsymbol{C}_t$$

여기서 $\boldsymbol{C}_t$는 시점 t의 관측된 코사인 유사도 행렬이다.

> **수식 직관**
>
> 이 수식은 "과거의 기억($\boldsymbol{A}_{t-1}$)에 새 관측($\boldsymbol{C}_t$)을 조금씩 섞어 넣는다"는 것을 말한다. $\alpha = 0.9$이면 매번 기존 값의 90%를 유지하고 새 값의 10%만 반영한다. 직관적으로, 하루하루의 주가 변동이 아닌 이동 평균 추세선을 보는 것과 같다 – 노이즈를 걸러내고 진짜 경향을 포착한다.

> **학부 수학**
>
> **등비급수와 EMA 수렴.** 아래 증명에서 핵심이 되는 수학은 등비급수의 합이다. 고등학교에서 배운 공식을 떠올려 보자: $\sum_{k=0}^{n-1} \alpha^k = \frac{1-\alpha^n}{1-\alpha}$ (단, $|\alpha| < 1$). $n \to \infty$이면 $\alpha^n \to 0$이므로 $\sum_{k=0}^{\infty} \alpha^k = \frac{1}{1-\alpha}$이다. $\alpha = 0.9$를 대입하면 $\frac{1}{1-0.9} = 10$이다. 증명에서 $(1-\alpha)\sum_{k=0}^{t-1} \alpha^k$의 합은 $(1-\alpha) \cdot \frac{1-\alpha^t}{1-\alpha} = 1-\alpha^t$이 된다. $t \to \infty$이면 $\alpha^t \to 0$이므로 가중치의 합이 1로 수렴하고, EMA가 정확히 true mean $\boldsymbol{C}^*$에 수렴한다는 것이 증명의 핵심이다. 직관적으로, 무한히 많은 "조금씩 줄어드는 가중치"들의 총합이 정확히 1이 되므로, 가중 평균이 진짜 평균에 일치하는 것이다.

명제 **A.1**: $\boldsymbol{C}_t$가 정상 (stationary) 분포를 따르고 $\mathbb{E}[\boldsymbol{C}_t] = \boldsymbol{C}^*$이면,

$$\mathbb{E}[\boldsymbol{A}_t] \to \boldsymbol{C}^* \quad \text{as } t \to \infty$$

증명: $\boldsymbol{A}_t$를 전개하면

$$\boldsymbol{A}_t = (1-\alpha)\sum_{k=0}^{t-1} \alpha^k \boldsymbol{C}_{t-k} + \alpha^t \boldsymbol{A}_0$$

$t \to \infty$에서 $\alpha^t \boldsymbol{A}_0 \to 0$이고,

$$\mathbb{E}[\boldsymbol{A}_t] = (1-\alpha)\sum_{k=0}^{t-1} \alpha^k \mathbb{E}[\boldsymbol{C}_{t-k}] = (1-\alpha)\cdot \boldsymbol{C}^* \cdot \frac{1-\alpha^t}{1-\alpha} = \boldsymbol{C}^*(1-\alpha^t) \to \boldsymbol{C}^*$$

따라서 EMA 친화도는 **true affinity**에 수렴한다. □

분산: $\mathrm{Var}(\boldsymbol{A}_t) \approx \frac{1-\alpha}{1+\alpha}\,\mathrm{Var}(\boldsymbol{C}_t)$

$\alpha = 0.9$이면 분산이 원래의 $\approx 5.3\%$로 감소하여 높은 안정성을 제공한다.

> **수식 직관**

명제 A.1의 핵심 메시지는 두 가지이다. 첫째, EMA 친화도는 충분한 시간이 지나면 진짜 태스크 간 친화도에 수렴한다 – 초기값 $\boldsymbol{A}_0$의 영향은 α^t로 기하급수적으로 사라진다. 둘째, 분산이 원래의 약 5%로 줄어들므로, 배치마다 요동치는 gradient 노이즈가 대부분 제거된 안정적인 추정값을 얻는다.

A.2 Group Prior의 Bayesian 해석

adaTT의 Group Prior는 **Bayesian** 추론의 prior distribution으로 해석할 수 있다.

역사적 배경

Conjugate Prior의 역사. 아래에서 사용하는 Normal-Normal conjugacy(정규분포 prior + 정규분포 likelihood → 정규분포 posterior)는 Bayesian 통계학에서 가장 오래되고 기본적인 결과 중 하나이다. Raiffa & Schlaifer (1961, *"Applied Statistical Decision Theory"*)가 **conjugate family** 개념을 체계화했으며, prior와 posterior가 같은 분포 족에 속하면 posterior를 해석적(closed-form)으로 구할 수 있다는 이점이 있다. 머신러닝에서는 가우시안 프로세스(Rasmussen & Williams, 2006), Bayesian Linear Regression, Kalman Filter 등이 모두 이 conjugate 구조를 활용한다. adaTT의 blend 공식이 conjugate posterior mean과 일치한다는 것은, 도메인 지식과 데이터를 결합하는 가장 수학적으로 정당한 방법을 사용하고 있다는 것을 의미한다.

학부 수학

Normal-Normal Conjugacy 유도 **(간략판).** Prior: $\theta \sim \mathcal{N}(\mu_0, \sigma_0^2)$. 관측: $x \mid \theta \sim \mathcal{N}(\theta, \sigma^2)$. Posterior는 $\theta \mid x \sim \mathcal{N}(\mu_n, \sigma_n^2)$이며, $\mu_n = \frac{\sigma^2 \mu_0 + \sigma_0^2 x}{\sigma^2 + \sigma_0^2}$이다. 이를 정리하면 $\mu_n = r \cdot \mu_0 + (1 - r) \cdot x$, 여기서 $r = \frac{\sigma^2}{\sigma^2 + \sigma_0^2}$이다. 이 공식의 직관은 단순하다: 관측의 노이즈(σ^2)가 크면 r이 커져서 prior μ_0를 더 믿고, prior의 불확실성(σ_0^2)이 크면 r이 작아져서 관측 x를 더 믿는다. adaTT에서 $\mu_0 = \boldsymbol{P}$(Group Prior), $x = \boldsymbol{W} + \boldsymbol{A}$(학습된 가중치 + 친화도)이며, r이 0.5에서 0.1로 감소하는 것은 "관측 데이터의 양이 늘어나면서 σ^2의 영향이 줄어든다"는 것을 모사한다.

모델: 태스크 i, j의 true affinity $a_{i,j}$에 대해

$$a_{i,j} \mid \boldsymbol{P}, \sigma^2 \sim \mathcal{N}\left(\boldsymbol{P}_{i,j}, \sigma^2\right)$$

여기서 $\boldsymbol{P}$는 Group Prior 행렬이다.

수식 직관

이 수식은 "진짜 친화도 $a_{i,j}$는 도메인 전문가의 사전 추정($\boldsymbol{P}_{i,j}$) 근처에 있을 것이다"라는 사전 믿음을 말한다. σ^2가 작으면 사전 추정을 강하게 신뢰하는 것이고, 크면 "Prior가 맞는지 잘 모르겠다"는 뜻이다.

관측: 코사인 유사도 $c_{i,j} = \cos\left(\theta_{i,j}\right)$

$$c_{i,j} \mid a_{i,j}, \tau^2 \sim \mathcal{N}\left(a_{i,j}, \tau^2\right)$$

Posterior mean (conjugate normal):

$$\mathbb{E}\left[a_{i,j} \mid c_{i,j}\right] = \frac{\tau^2}{\sigma^2 + \tau^2} \boldsymbol{P}_{i,j} + \frac{\sigma^2}{\sigma^2 + \tau^2} c_{i,j}$$

이를 $r = \frac{\tau^2}{\sigma^2 + \tau^2}$로 정의하면:

$$\mathbb{E}[a_{i,j}] = r \cdot \boldsymbol{P}_{i,j} + (1 - r) \cdot c_{i,j}$$

> **수식 직관**
>
> 이 posterior mean 공식은 "Prior(도메인 지식)와 관측 데이터(gradient 유사도)를 신뢰도에 비례하여 가중 평균한다"는 것을 말한다. 관측의 노이즈(τ^2)가 크면 r이 커져 Prior를 더 신뢰하고, Prior의 불확실성(σ^2)이 크면 r이 작아져 데이터를 더 신뢰한다. 이것이 adaTT의 `raw_weights * (1-r) + group_prior * r` 코드와 수학적으로 동일하다는 것이 핵심 결론이다.

이것은 adaTT의 prior blend 공식 (`adatt.py:381`)과 정확히 동일하다:

```
raw_weights = raw_weights * (1 - r) + self.group_prior * r
```

r의 annealing ($r_{\text{start}} \to r_{\text{end}}$)은 관측 데이터가 누적될수록 prior의 분산 σ^2가 상대적으로 커지는 (즉, prior에 대한 확신이 줄어드는) Bayesian updating 과정을 반영한다.

A.3 Softmax Temperature의 역할

$$w_{i \to j} = \frac{\exp\left(\frac{\boldsymbol{R}_{i,j}}{T}\right)}{\sum_{k \neq i} \exp\left(\frac{\boldsymbol{R}_{i,k}}{T}\right)}$$

> **수식 직관**
>
> 이 수식은 "원점수 $\boldsymbol{R}_{i,j}$를 T로 나눈 뒤 softmax를 취한다"는 것이다. T로 나누면 점수 차이가 $\frac{1}{T}$배 확대되므로, T가 작을수록 최고 점수와 나머지의 격차가 벌어져 "승자 독식"에 가까워진다. 반대로 T가 크면 점수 차이가 축소되어 거의 균등한 분포가 된다.

- $T \to 0$: 가장 높은 가중치에 집중 (hard selection)
- $T \to \infty$: 균등 분포 (모든 태스크 동일 가중치)
- $T = 1.0$ (기본값): 중간 지점, 친화도 차이를 적절히 반영

본 시스템에서 $T = 1.0$을 사용하는 이유: 16개 태스크에서 너무 sharp한 선택 ($T < 0.5$)은 소수 태스크에만 전이가 집중되고, 너무 uniform한 선택 ($T > 2.0$)은 negative transfer를 충분히 차단하지 못한다.

> **최신 동향**
>
> **Softmax Temperature의 현대적 활용 (2023–2025).** Temperature scaling은 MTL을 넘어 다양한 영역에서 핵심 역할을 하고 있다. (1) **Knowledge Distillation**: Hinton et al. (2015)의 원래 KD 논문에서 T가 "dark knowledge"의 전달 정도를 제어했고, 최근에는 DKD (Zhao et al., CVPR 2022)가 target class와 non-target class의 logit을 분리하여 별도 temperature를 적용한다. (2) **LLM Decoding**: GPT-4, Claude 등의 대형 언어모델에서 generation temperature가 창의성 **vs** 정확성 트레이드오프를 제어한다. (3) **Contrastive Learning**: SimCLR (Chen et al., ICML 2020)에서 $T = 0.07$ 같은 낮은 temperature가 hard negative에 집중하게 하여 표현 학습 성능을 향상시킨다. (4) **Gumbel-Softmax**:

Jang et al. (ICLR 2017)은 T를 학습 중 점진적으로 낮춰(annealing) 이산적 선택을 미분 가능하게 근사한다. adaTT에서도 향후 T를 학습 가능한 파라미터로 만들거나 phase별로 annealing하는 확장을 고려할 수 있다.

A.4 Negative Transfer 차단의 이론적 근거

Yu et al., 2020 의 PCGrad에서 gradient conflict를 $\cos(\theta_{i,j}) < 0$으로 정의한다. adaTT는 이보다 완화된 임계값 $\tau_{\text{neg}} = -0.1$을 사용하는데, 이는 **noise margin**을 고려한 것이다.

학부 수학

벡터 사영(**projection**)과 **gradient conflict.** PCGrad의 핵심 연산은 벡터 사영이다. 벡터 $\boldsymbol{a}$를 $\boldsymbol{b}$ 위에 사영하면 $\text{proj}_{\boldsymbol{b}}\boldsymbol{a} = \frac{\boldsymbol{a}\cdot\boldsymbol{b}}{\boldsymbol{b}\cdot\boldsymbol{b}}\cdot\boldsymbol{b}$이다. PCGrad는 $\cos\theta < 0$(충돌)일 때 gradient $\boldsymbol{g}_i$에서 $\boldsymbol{g}_j$ 방향 성분을 제거한다: $\boldsymbol{g}_{i'} = \boldsymbol{g}_i - \text{proj}_{\boldsymbol{g}_j}\boldsymbol{g}_i$. 기하학적으로 이는 $\boldsymbol{g}_i$를 $\boldsymbol{g}_j$에 수직인 평면에 사영하는 것이다. 결과적으로 $\boldsymbol{g}_{i'}\cdot\boldsymbol{g}_j = 0$이 되어 충돌이 해소된다. adaTT는 이런 사영 대신 전이 자체를 차단하는 더 단순하고 보수적인 전략을 취한다 — gradient를 변형하지 않고, 해로운 태스크의 loss 기여를 0으로 만든다.

SGD의 stochastic noise로 인해 true affinity가 0에 가까운 태스크 쌍도 배치에 따라 $\cos(\theta_{i,j}) \approx -0.05$ 등의 약한 음의 상관을 보일 수 있다. $\tau_{\text{neg}} = -0.1$은 이러한 noise를 허용하면서 명확한 negative transfer만 차단하는 sweet spot이다.

A.5 Transfer-Enhanced Loss의 수렴 영향

명제 **A.5**: Transfer loss의 gradient는 positive transfer 태스크의 학습 방향으로 공유 파라미터를 편향시킨다.

$$\nabla_\theta \mathcal{L}_i^{\text{adaTT}} = \nabla_\theta \mathcal{L}_i + \lambda \sum_{j \neq i} w_{i\to j} \nabla_\theta \mathcal{L}_j$$

수식 직관

이 수식은 "태스크 i의 파라미터 업데이트 방향이 자기 자신의 gradient에 다른 태스크들의 gradient를 가중 합산한 보정 벡터를 더한 것"임을 말한다. 직관적으로, 나 혼자 걸어가던 방향에 동료들이 "이쪽이 더 좋다"고 제안하는 벡터를 $\lambda = 0.1$만큼 반영하여 경로를 미세 조정하는 것이다. 친화도가 높은 동료의 제안($w_{i\to j}$가 큰 경우)이 더 크게 반영된다.

$w_{i\to j} > 0$이고 $\cos(\theta_{i,j}) > 0$인 태스크 j의 gradient가 태스크 i의 gradient 방향으로 가산되어, 공유 파라미터가 양쪽 태스크 모두에 유리한 방향으로 업데이트된다.

$\lambda = 0.1$과 `max_transfer_ratio=0.5` 의 조합은 원본 loss의 학습 방향을 크게 왜곡하지 않으면서도 positive transfer의 이점을 얻을 수 있는 보수적 설정이다.

교차 참조 – **Logit Transfer**와의 관계

adaTT는 **backward pass**(gradient) 수준에서 태스크 간 전이를 조절하는 반면, 본 시스템에는 **forward pass**(logit) 수준에서 태스크 간 예측값을 직접 전달하는 **Logit Transfer** 메커니즘이 별도로 존재한다. Logit Transfer는 CTR→CVR→LTV 같은 비즈니스 로직상 순차적 의존성을 단방향 DAG로 명시적으로 설계하며, adaTT의 전방향

적응적 전이와 상호 보완적으로 작동한다. 상세 메커니즘은 *PLE* 기술 참조서 의 “Logit Transfer – 태스크 간 명시적 정보 전달” 절을 참조한다.