교수님 연구 미팅 - 11-27

태그	
날짜	@2025년 11월 14일
상태	진행 중
실험 제목	

https://arxiv.org/pdf/2302.07849

실험

실험 리스트

주요 문제점 및 개선을 위한 참고
▼ 0. 현재 구조 정리 (Best CCNF)

주요 구조

핵심 구조 (세미나 p.14–20 기준)

- **NF+GMM → 단일 NF(Conditional NF)** 로 전환하여 misalignment 문제 해결
- **Class Embedding + Affine Coupling (RealNVP)** 기반 구조
- 안정화 장치 5종:

  ① Spectral Norm (Jacobian 안정화)

  ② Clamp α (|det J| 상한 제한)

  ③ Warm-up (Additive → Affine gradual transition)

  ④ Gradient Clipping (overflow 방지)

  ⑤ Balance / Logdet / Z-regularization (Jacobian–latent ratio 균형 유지)

결과 (세미나 p.23–25)

- Pixel AUROC 최고 83.9%, Forgetting 거의 0
- Image AUROC 65~67% 수준으로 여전히 낮음
- Spectral Norm이 가장 결정적 안정화 요소 (없을 시 catastrophic forgetting 발생)
- Task 0에서는 Spectral Norm이 과도 제약 → 첫 Task만 off 하는 전략이 제안됨

장점 요약

항목	설명
안정성	학습 divergence 없음 (Jacobian 제어 완벽)
Continual robustness	Task 간 feature mapping alignment 유지
경량성	3 epoch 이내 수렴, replay/regularization 불필요
해석 가능성	latent separability 유지 (t-SNE 기반 class clustering 뚜렷)

문제점

(A) 성능 한계

- Image-level AUROC이 Pixel-level 대비 20%p 이상 낮음 → latent likelihood는 잘 학습되지만 global context representation이 부족.
- NF는 local feature의 bijective mapping만 다루기 때문에 spatial aggregation에 약함.

(B) 표현적 한계

- Class embedding이 고정적 (task adaptation 없음)

  → intra-class variation(조명·texture 등)에 대응하지 못함.
- Affine coupling의 linear한 transform 한계 → 고차원 feature 상관관계 반영 부족.

(C) 확장적 한계

- Stable하나 구조적 novelty 부족 (기존 RealNVP 변형 수준)
- Meta adaptation, MoE gating, multi-scale modeling 등의 확장성 미흡.
- Continual setting에서는 좋은데 “domain generalization / zero-shot transfer” 성질은 약함.

🧩 표현적 한계 : (1) Class embedding이 고정적이라 task adaptation이 어렵다

🔷 구조적으로 무슨 의미인가?

현재 CCNF에서 class embedding은 `nn.Embedding(n_classes, emb_dim)` 으로 정의되어 있습니다.

[

e_y = \text{Emb}(y), \quad e_y \in \mathbb{R}^{d_{emb}}

]

이 `e_y` 는 **class ID y마다 고정된 하나의 벡터**로 학습됩니다.

즉, 같은 클래스의 모든 샘플(feature, domain, illumination 등)은

하나의 정적인 condition vector로 flow transformation을 공유합니다.

NF는 이를 condition으로 사용하여 `s(x_a, e_y)` 와 `t(x_a, e_y)` 를 계산하죠.

🔷 왜 이것이 문제인가?

(1) Intra-class variation을 표현하지 못함

예를 들어, “bottle” 클래스 안에는

- 금속 vs 유리 재질
- 어두운 조명 vs 밝은 조명
- 회전된 각도, scale 등

  다양한 feature 분포가 존재합니다.

그런데 CCNF에서는 이 모든 샘플이 같은 (e_{\text{bottle}}) 로 condition 되기 때문에,

flow가 각 variation을 **하나의 transformation으로 맞춰야 함** → over-smoothing or overfitting 발생.

> 즉, embedding은 class-level condition은 제공하지만 instance-level or context-level adaptability 가 없습니다.

---

(2) Continual setting에서 “new class interference”

Continual learning에서는 새로운 class가 추가되면 embedding table이 확장되고,

기존 embedding은 고정된 latent reference에 맞춰져 있습니다.

하지만 새로운 class의 feature distribution이 global mean을 바꾸면

기존 embedding의 의미가 뒤틀리거나 scale mismatch가 생김.

→ 결과적으로 **이전 class의 flow mapping**이 약간씩 drift (catastrophic forgetting의 원인 중 하나).

---

(3) Embedding이 “discrete conditioning”이라 generalization이 약함

Flow의 transformation은 conditional parameterization:

[

y_b = x_b \cdot \exp(s(x_a, e_y)) + t(x_a, e_y)

]

이때 (e_y) 가 discrete embedding이면,

새로운 class나 unseen domain (e_{y^*}) 에 대해 interpolation이 불가능합니다.

즉, class ID가 학습된 집합 밖이면 condition space가 정의되지 않음.

→ Zero-shot / few-shot adaptation 불가능.

🔷 요약

관점	문제점
Intra-class diversity	조명, 질감, 각도 등 다양한 분포를 단일 embedding으로 설명 불가
Inter-class drift	새로운 class 등장 시 기존 embedding 간 관계 왜곡
Generalization	학습된 class 외 domain에 대응 불가 (discrete conditioning)

🔷 개선 방향 예시

- **Dynamic / Adaptive embedding**

  → instance feature나 batch 통계 기반으로 e_y 보정 (e.g., meta-conditioning, FiLM meta adapter)

- **Prototype embedding**

  → class 내 cluster별 embedding 다중화 (GMM-style or MoE conditioning)

- **Task embedding 추가**

  → continual task마다 별도 context vector를 주입 (domain-aware conditioning)

🧩 표현적 한계 : (2) Affine Coupling의 Linear Transform 한계

🔷 현재 coupling 구조의 수식적 형태

CCNF의 각 coupling layer는 RealNVP 계열의 **affine transformation** 을 사용합니다:

[

\begin{aligned}

x &= [x_a, x_b] \

y_a &= x_a \

y_b &= x_b \cdot \exp(s(x_a, e_y)) + t(x_a, e_y)

\end{aligned}

]

즉, half-split 구조에서 x_a는 그대로 통과하고,

x_b만 **scale-exp + translation** 형태로 변환됩니다.

이 변환은 **element-wise linear** 합니다.

🔷 왜 이것이 문제인가?

(1) 고차원 상관관계를 학습하지 못함

Affine transformation은 각 차원을 독립적으로 스케일링합니다.

즉, `s` 와 `t` 가 각각 feature-wise로 적용되므로

feature 간 상호작용(correlation) 이 제한됩니다.

> Coupling layer는 invertible이지만 linear →
>
> 복잡한 covariance structure를 모델링하지 못하고
>
> diagonal Gaussian에 가까운 형태로 수렴함.

결과적으로,

- latent z에서 클래스 간 separation은 뚜렷하지만,
- class 내부 구조(세부 패턴, fine-grained distribution)는 collapse됩니다.

→ Image AUROC (global context) 낮은 이유 중 하나가 이겁니다.

---

(2) 여러 layer를 쌓아도 correlation 학습이 느림

Permutation이 단순 channel-swap이기 때문에,

x_a ↔ x_b 간 정보 교환이 제한적입니다.

layer를 여러 개 쌓아야 feature 전체가 mixing되지만,

mixing efficiency가 낮아 학습이 느리고 expressive power 제한됨.

---

(3) Continuous domain에서는 비선형 decision boundary 형성 불가

Affine coupling은 결국 log $p(x|y)$ 를

log-volume + quadratic term (from z^2) 형태로 근사합니다.

이 형태는 **local Gaussian-like** behavior를 보이며

highly nonlinear boundary를 구성하기 어렵습니다.

→ texture-based anomaly에서는 잘 동작하지만,

shape-based / context anomaly에서는 실패 가능성이 큼.

🔷 요약

관점	문제점
Feature dependency	feature 간 correlation 반영 불가 (element-wise scaling 한계)
Mixing inefficiency	permutation 단순 → layer stacking 필요
Nonlinearity 부족	복잡한 decision boundary 생성 어려움 (global structure 표현력 낮음)

🔷 개선 방향 예시

개선 방향	개념	기대 효과
Nonlinear Coupling (Residual Flow, Neural Spline Flow 등)	s, t를 MLP → spline 기반 비선형으로 확장	local correlation 반영, expressive power 향상
1×1 Invertible Convolution (Glow-style)	linear mixing matrix로 feature 간 coupling	feature 간 correlation 즉시 반영
Conditioned MoE Coupling	class/domain별 expert coupling layer 선택	feature correlation + class 특성 동시 반영

🧩 종합 요약

구분	구조적 원인	결과	해결 방향
고정 class embedding	discrete table 기반 condition	intra-class diversity, generalization 한계	dynamic embedding, meta-conditioning
Affine coupling 한계	element-wise linear transform	feature correlation 반영 불가, global 표현 약함	nonlinear coupling, invertible conv, MoE coupling

▼ 1. DSM Loss 추가

2. multi scale noise

- MULDE의 방식으로 부터 motivation

실험 항목	내용	결과
Cross cond	다른 block의 feature를 condition으로 사용하는 방법	효과 없음
P6_NC_CCNF	MULDE의 multi noise 방법 motivation	image-auroc 성능 향상
P6_NC_CCNF_task0_spect_false	P6+task0에서 spect off 하고 task1부터 사용	

▼ 2. 코드에서 발견되는 치명적 구조적 문제들

▼ ~~(A) 학습 vs 추론에서 입력 분포가 서로 다름 (진짜 크리티컬)~~

`NFCAD.forward`를 보면, 학습 시에는 NF에 넣기 전에 feature를 배치 단위로 표준화합니다.

```
def forward(self, x, y=None, pos_embed=None, scale=0, epoch=0, g
lobal_step=None):
    y_labels, y_onehot = y
    y_labels = y_labels.reshape(-1)  # (N,)

    # ⚠ 여기: 배치 단위로 표준화
    with torch.no_grad():
        mean = x.mean(dim=0)
        std = x.std(dim=0) + 1e-5
    x = (x - mean) / std

    results = self.ccnfs[scale].log_prob_with_logdet(x, y_labels, pos_e
mbed, global_step)
    ...
```

그런데 **추론(inference)** 쪽은 이렇게 되어 있습니다:

```
def inference(self, features, lvl, final_class_id, task_identity, n_learne
d_classes):
    e = features[lvl].detach()
    bs, dim, h, w = e.size()
    e = e.permute(0, 2, 3, 1).reshape(-1, dim)  # (N, dim)

    pos_embed = positionalencoding2d(...)
```

```
# ⚠️ 여기: 아무런 표준화 없이 바로 NF에 넣음
logps = self.ccnfs[lvl].log_prob(e, y_temp, pos_embed)
...
```

즉,

- 학습: `x_norm = (x - batch_mean) / batch_std` 기준으로 NF를 학습
- 추론: **raw feature** `x` **를 그대로** NF에 넣어서 log p(x|y)를 계산

→ **학습한 분포와 추론에 쓰는 입력 분포가 아예 다릅니다.**

이건

- 첫 task에서 density가 망가져서 성능이 바닥을 치고,
- class마다 score 스케일이 들쭉날쭉해지고,
- pixel-level은 상대 비교라 그럭저럭 나오는데, image-level은 전체 스코어가 뒤틀리는

패턴을 만들기 딱 좋은 상황입니다.

> ✅ 이건 단순한 튜닝 문제가 아니라 모델 구조(전처리 파이프라인) 자체가 깨져 있는 거라
>
> **무조건 먼저 고쳐야 할 부분**입니다.

▼ ~~(B) `self.feat_dims` 와 `ccnfs` 개수 정의 방식~~

초기화 부분:

```
self.encoder = Encoder(args)
feat_dims = self.encoder.feat_dims
self.feat_dims = feat_dims * args.feature_levels

self.ccnfs = nn.ModuleList([
    load_ccnf_model(args, feat_dim, self.n_classes)
    for feat_dim in self.feat_dims
])
```

- ViT인 경우 `Encoder` 에서 이미 `self.feat_dims = [feat_dim] * args.feature_levels` 로 되어 있음.
- 여기서 다시 `args.feature_levels` 를 해서

→ `self.feat_dims` 길이가 `feature_levels^2`가 됩니다.

- 그런데 실제로는 `for lvl in range(self.args.feature_levels)`만 사용하니

  앞의 몇 개 NF만 쓰고 나머지는 완전히 버리는 구조예요.

기능상 치명적인 버그는 아닐 수 있지만,

- capacity 낭비
- 코드 읽기 어렵고 추후 버그 유발 가능
- logdet normalization에 쓰는 `self.feat_dims`도 직관적이지 않음

이라서 구조적으로 정리해 두는 게 좋습니다.

```
self.encoder = Encoder(args)
self.feat_dims = self.encoder.feat_dims  # 그대로 쓰기

self.ccnfs = nn.ModuleList([
    load_ccnf_model(args, feat_dim, self.n_classes)
    for feat_dim in self.feat_dims
])
```

▼ 4. class-feature 간 querying quality 개선

지금 구조를 “query–key–value” 시점에서 보면 **뭔가 개념적으로 허전한 구석이 있는 상태**가 맞습니다.

▼ 4-1. 현재 코드에서 class embedding은 실제로 뭘 하고 있냐?

지금 CCNF를 보면, class embedding은 이렇게 쓰이고 있어요:

```
def _get_conditioning(self, y, pos_embed=None, sigma=None):
    cond_list = [self.emb(y)]  # (B, cond_dim)
    if self.pos_cond_dim > 0 and pos_embed is not None:
        cond_list.append(pos_embed)
    ...
    return torch.cat(cond_list, dim=-1)
```

그리고 coupling layer에서는

```
h = torch.cat([x_a, cond], dim=-1)
s = self.net_s(h)
t = self.net_t(h)
```

즉:

- `x`(feature) ↦ `flow(x; θ)` 이 아니라
- `x`(feature) ↦ `flow(x; θ, cond(y))` 라서

  class embedding은 단순히 "조건(conditioning)"으로 들어가서 s,t 네트워크를 바꾸는 역할만 합니다.

그리고 loss는

- `log p(x|y_true)`를 최대화하려는 `L_g`
- `log p(x|y_true)` vs `log p(x|y_neg)` margin을 키우는 `L_mi`, `L_e`

정도로 class 간 구분을 유도하죠.

그러니까 수식으로 보면 이 모델은

> "x가 주어졌을 때, label y가 알려져 있다는 가정하에 p(x|y) 를 최대우도 추정하는 generative 모델"

이고,

> inference 때는 argmax_y log p(x|y) 로 class를 고르는 "naive generative classifier"

입니다.

여기에는 **어디에도 "query로서의 x가 key/value로서의 class embedding을 찾는 과정"은 명시되어 있지 않아요.**

▼ 4-2. 그래서 네가 말한 "query ↔ key 유사도 최대화"는 지금 구조에 없다

정리하면:

- 학습 시: y는 주어진 정답 label이고, 그 y에 해당하는 embedding을 그냥 가져다 씀.
- class embedding은 **"이 class일 때 flow의 parameter를 어떻게 바꿀까"**를 담는 벡터일 뿐,

- x와 embedding 사이에 **직접적인 similarity objective**(dot product, cosine, InfoNCE 같은 것)는 없습니다.
- L_mi, L_e는 "올바른 y에서의 log-likelihood가 더 크도록" 만들 뿐,

  그게 꼭 `sim(f(x), e_y)` 형태일 필요는 없죠.

그래서 네 intuition대로:

> "inference에서는 x가 query, class embedding이 key/value처럼 행동하는데
>
> 학습 때는 query–key alignment를 직접 안 걸어준다"

라는 느낌은 **정확한 지적**입니다.

현재 구조는 "flow weight 전체가 class를 구분하도록 바뀌는" 쪽에 더 가깝고,

class embedding은 안 써도 되는 보조 정보로 밀려날 위험이 있어요.

hidden_dim이 크면 더더욱 "embedding 안 쓰고도 버틴다" 쪽으로 가버릴 수 있고요.

▼ 4-3. 그게 왜 문제냐? (continual + AD 관점)

이게 특히 문제되는 지점이 몇 가지 있어요.

1. **class embedding이 "실질적인 class prototype" 역할을 하지 않음**
   - flow 네트워크가 복잡하다 보니, class 구분 정보를 **weight 안에 다 집어넣고**,

     embedding은 거의 안 써도 모델이 학습이 가능함.
   - 그러면 나중에 **새 class 추가(continual)** 할 때 embedding만 늘리는다고 해서

     잘 분리된 새로운 class 공간이 생기지 않음.
2. **feature–class 관계가 분명하지 않음**
   - z space에서 class별 클러스터가 잘 만들어졌는지 보장 X.
   - 그냥 "이 y를 conditioning으로 넣으면 loss가 줄어드는 방향으로"만 학습되니,

     embedding 자체가 "해석 가능하거나, 재사용 가능한 prototype"이 되지 않음.
3. **image-level decision에 활용하기 어렵다**

- query–key 스타일이면 “feature가 어느 class 쪽 prototype과 가깝냐”로 빠르게 score를 만들 수 있는데,

  지금은 **각 class마다 flow를 통과시키고 log p(x|y)**를 다시 계산해야 함.

- 계산 비용도 크고, score calibration도 더 어렵지.

결론적으로,

> 지금 설계는 “generative class-conditional flow”로는 맞는데,
>
> “query–key 기반 prototype 모델” 관점에서는 구조가 반쯤만 완성된 상태

라고 보면 됩니다.

▼ 4-4. 네가 원하는 쪽으로 바꾸려면: 구조적으로 뭘 바꿔야 하냐?

“x는 query, class embedding은 key/prototype”이라는 semantics를 **진짜로** 주고 싶다면,

모델을 아래 둘 중 하나 방향으로 바꾸면 됩니다.

4-1. Flow + Latent Gaussian Prior (embedding = μ_y)

이전에 살짝 언급했던 구조를 여기서 다시 꺼내보면 딱 이걸 해결해 줍니다.

1. **Flow는 unconditional**로 두고

   [

   z = f_\theta(x)

   ]

2. class embedding e_y를 **latent space의 Gaussian center**로 사용:

   [

   p(z|y) = \mathcal{N}(z; \mu_y = e_y,\ \Sigma_y)

   ]

   - μ_y = class embedding
   - Σ_y는 diagonal(σ_y²) 정도로 두고, 이것도 embedding에서 뽑든, 별도 파라미터로 두든.

3. 그러면

[

\log p(x|y) = \log p(z|y) + \log\left|\det\frac{\partial f}{\partial x}\right|

]

이고, `log p(z|y)` 항이 **완전히 선명한 "query–prototype similarity"** 구조가 됩니다:

```
# z: (B, D)
mu_y = emb_mu(y)       # (B, D)
logvar_y = emb_logvar(y)
var_y = torch.exp(logvar_y)

log_pz_y = -0.5 * (((z - mu_y)**2) / var_y + logvar_y + const).sum(dim=-1)
```

여기서 `(z - mu_y)**2`를 줄이도록 학습하는 게 바로

z를 자기 class embedding 근처로 몰아넣는 것이죠.

→ 이 구조에서는

- 학습 과정에서 **자동으로 "z와 e_y의 유사도 최대화"가 들어간 셈**이고,
- class embedding은 진짜로 **latent prototype** 역할을 하게 됩니다.
- continual 시에는 새 class 추가할 때 μ_new, σ_new만 추가하면 되니 훨씬 자연스럽고요.

이건 네가 말한 "query–key/max similarity"를

NF의 generative objective 안으로 통합한 버전이라고 보면 됩니다.

4-2. Flow + Contrastive / Classification Head (explicit similarity loss)

조금 더 직접적인 방법은:

1. 지금처럼 `z = f(x, y)` 이런 conditional flow 말고,

   아예 **unconditional flow** 또는 backbone feature h(x)를 하나 두고,

```
z = f(x)       # 또는 그냥 encoder feature z
q = proj(z)    # (B, d)  - query
```

2. class embedding을 key로 사용:

```
# 클래스별 embedding K: (C, d)
K = class_embed.weight        # (n_classes, d)
logits = q @ K.T              # (B, C)
```

3. 여기에 **cross-entropy / InfoNCE**를 걸어줌:

```
L_cls = F.cross_entropy(logits, y_true)
```

이러면 이건 완전히 CLIP/contrastive 스타일:

- q(x)는 **자기 class의 embedding에 가깝고, 다른 class와는 멀어지도록** 학습
- class embedding이 진짜로 "key/prototype" 역할을 하게 됨.

그리고 이걸 기존 NF loss에 **추가로 얹는 구조**로 만들 수 있어요:

[

L_{\text{total}} = L_{\text{NF}} + \lambda_{\text{cls}} L_{\text{cls}}

]

여기서 `L_NF`는 지금 쓰는 `L_g, L_mi, L_e, ...` 조합이고,

`L_cls`가 명시적인 query–key alignment.

장점:

- 구조적으로 간단함
- image-level classification/score에도 바로 쓸 수 있음 (logits 이용)
- continual에서도 class embedding을 prototype으로 쓸 수 있음

단점:

- "pure generative"에서 살짝 벗어나서 **hybrid (generative + discriminative)**가 됨.
- 하지만 지금도 L_mi, entropy 등 이미 "pseudo-discriminative" loss가 있으니,

  사실상 이미 hybrid인 셈이고, 여기에 제대로 된 head 하나 추가하는 수준이라고 보면 됩니다.

▼ 4-5. “지금 구조에서도 완전 엉터리는 아니다”라는 얘기도 잠깐

혹시 걱정할까 봐 덧붙이면,

- 지금 구조도 엄밀히 말하면 **x–y joint를 간접적으로 맞추고 있긴 해요.**

왜냐하면

- `L_g`는 `log p(x|y)`를 키우고
- `L_mi`는 `log p(x|y_true) > log p(x|y_neg)`를 만들려고 하기 때문에,

결국

> “x는 자기 label y에서 더 높은 log-likelihood를 가져야 한다”

라는 constraint는 존재합니다.

다만,

- 그 과정에서 **class embedding이 핵심이 되도록 강제하는 구조가 부족**하고,
- flow weight가 충분히 크면 **embedding은 거의 안 써도 class 구분이 가능**해서

  embedding이 “dead parameter”가 될 위험이 있습니다.

그래서 네가 말한 그 느낌:

> “query–key alignment가 loss에 없다”

→ **맞는 말이고,**

그걸 구조적으로 보완해 주면 지금 겪는 문제들

(첫 task 학습 불안정, class별 편차, image-level classification 애매함)에

꽤 직접적으로 도움이 될 가능성이 큽니다.

▼ 4-6. 현실적인 적용 플랜 (내가 하면 이렇게 함)

내가 네 코드에 손을 대야 한다면:

1. **NF를 “unconditional + latent prior” 구조로 단순화**
   - CCNF에서 y-conditioning을 제거하고 `z = flow(x)`만.
   - ClassEmbed는 `μ_y, log σ_y`를 출력하는 prior head로 바꿈.
   - Loss는 `log p(z|y) + log|det J|` + 지금 쓰는 regularization 일부 재조정.

2. (옵션) **z-class embedding contrastive head 추가**
    - `q = proj(z)` 만들고
    - `logits = q @ E.T`로 classification head 추가
    - `L_cls`를 작은 weight로 추가해서 query–prototype alignment를 강화.
3. inference는
    - pixel-level: 기존처럼 `log p(x|y)` 또는 `Mahalanobis(z, μ_y)` 기반 score
    - image-level: prototype 거리 기반 top-k / GeM pooling.

이렇게 가면

- class embedding이 진짜로 "prototype / key"가 되고,
- 네가 말한 "query–key similarity 최대화"가 **명시적인 수학적 목표**로 들어가고,
- continual 시나리오에서도 새 class를 추가하기 훨씬 자연스러워집니다.

진행사항 기반 주요 문제점 및 개선, 결과

▼ 1. Baseline (E0_after_Refactor)

1.1 Motivation

- HGAD → CCNF로 구조를 옮기면서,

  "NF 하나로 class-conditional density를 학습하는 베이스라인"이 필요했음.
- 어떤 추가적인 안정화 트릭(feature norm, prototype, MI 등)을 넣기 전에

  순수 CCNF + 기존 loss 세트가 CL에서 어느 정도까지 버티는지를 알고 싶었다.

1.2 목적

- **리팩터링된 코드가 제대로 작동하는지 검증**
- Continual setting(5 tasks)에서
    - Pixel AUROC, Image AUROC의 기본 성능
    - Task별/전체 forgetting 패턴
    - Loss 항들의 수렴 패턴

을 baseline으로 확보.

1.3 방법 / 설정

- Encoder: CNN backbone (timm), `features_only` 모드, multi-level feature.
- NF: CCNF (coupling 기반)
  - class embedding + positional encoding을 조건으로 사용.
- Loss:
  - Main: `L_g` ($-\log p(x|y) / D$)
  - 정규화: `L_logdet_reg`, `L_balance`, `L_z_reg`, `L_scale_penalty`
  - 판별 보조: `L_mi`, `L_e`
  - noise regularization, z-reg 등 기존 세트 유지.
- Continual 구조:
  - 5 tasks, 각 task마다 3개 class씩 추가
  - 각 task 1 epoch 학습
  - NF와 embedding 모두 업데이트 (freeze 없음)

1.4 결과

- **최종 평균 성능:**
  - Image AUROC: **0.6811** (68.11%)
  - Pixel AUROC: **0.8360** (83.60%)
- Forgetting (Image / Pixel):
  - Avg Forgetting(Image): **0.0629**
  - Avg Forgetting(Pixel): **−0.0605**

    (픽셀 기준으로는 오히려 평균적으로 과거 task가 더 좋아짐)
- Task별 image 성능:
  - Task0: ~0.65 → Task4 이후에도 0.6 이상 유지
  - 일부 task에서 약간의 감소는 있으나 **전체적으로 가장 안정적인 패턴**
- Loss curve:

- `L_g`, `L_logdet_reg`, `L_z_reg` 등 대부분의 loss가

  짧은 iteration 내에 안정적인 감소.

- `L_noise_reg`도 다른 실험 대비 낮은 수준에서 수렴.

1.5 해석 / 시사점

- 리팩터링된 CCNF 기반 baseline은 **생각보다 꽤 괜찮은 CL 성능**을 보인다.
  - Pixel AUROC는 후반 task까지 꾸준히 증가.
  - Image AUROC도 다른 변형 실험들보다 가장 높은 수준.
- Loss 안정성 역시 baseline이 가장 좋음.

  → 이후 실험에서 성능이 나빠지면 **"새로 넣은 기법이 NF density를 망쳤다"**고 봐야 할 정도로, baseline이 기준점 역할을 확실히 한다.

▼ 2. Feature Norm (on/off)

(관련 실험: `E1_feature_norm`, `E2_feature_norm_weakly_regularization_at_1st`)

2.1 Motivation

- 앞에서 발견한 문제:
  - 학습 시 forward에서 batch-wise feature standardization을 하고,
  - inference에서 동일한 정규화를 적용하지 않아 **train/test feature 분포 mismatch** 문제가 있었음.
- 이를 해결하기 위해,
  - **Feature normalization을 명시적으로 제어**해서

    NF가 안정된 입력 분포를 보도록 만들고자 했다.

2.2 목적

- Feature norm을 도입하면
  - Pixel AUROC가 더 잘 나오는지,
  - Image AUROC / CL 안정성이 개선되는지 확인.
- 특히 **첫 task에서 학습이 잘 안 되는 문제**를 완화할 수 있는지 체크.

2.3 방법 / 설정

- E1: `feature_norm` 적용
  - 입력 feature를 per-batch 혹은 train 시 통계 기반으로 정규화.
- E2: `feature_norm` + `weakly regularization at 1st`
  - 첫 task(epoch)에서 regularization 강도를 약하게 두어,

    norm과 NF 정규화 loss가 과도하게 충돌하지 않도록 조정.

2.4 결과

- 성능 (최종 평균):

Exp	Image AUROC	Pixel AUROC
E0_baseline	0.6811	0.8360
E1_feature_norm	**0.5689**	**0.8541**
E2_feature_norm_weak_reg	**0.5867**	**0.8583**

- 패턴:
  - Feature norm을 켜면 **Pixel AUROC는 +2~3% 정도 꾸준히 상승**.
  - 그러나 **Image AUROC는 baseline에 비해 10pt 이상 하락**.
- Forgetting 측면:
  - E1/E2 모두 Avg Forgetting(Image)이 **음수** 혹은 큰 폭 감소

    → 수치상 "forgetting이 줄어든 것처럼" 보이지만,
  - 절대적인 이미지 성능이 baseline보다 낮기 때문에 의미있는 개선이라 보기 어려움.

2.5 해석 / 시사점

- Feature norm은
  - patch-level representation 분포를 정제해서

    pixel anomaly 구분에는 도움이 되지만,
  - global density scale(logdet, z 분산)을 바꿔버려

    이미지 단위 likelihood calibration을 망가뜨리는 효과가 있다.
- 결국:

- “pixel ↑ / image ↓”라는 trade-off가 반복적으로 나타남.
- feature norm 자체로는 image-level 문제(특히 class-wise calibration)를 해결하지 못한다.

▼ 3. Weakly regularization at 1st

(관련 실험: `E2_feature_norm_weakly_regularization_at_1st`,

`E3_no_feature_norm_weakly_regularization_at_1st` 및 변형들)

3.1 Motivation

- 첫 task에서:
  - `L_logdet_reg`, `L_z_reg`, `L_balance` 등이 너무 강하면

    NF가 사실상 identity에서 잘 못 벗어나거나,
  - 반대로 Jacobian 쪽이 과도하게 눌려서 표현력이 제한될 수 있다.
- 따라서 **첫 task에서 정규화 loss를 “살짝 풀어주는” 전략**을 통해
  - 초기 geometry 형성을 더 잘 하게 하고,
  - 이후 task에서 embedding만으로 빠르게 적응시키는 것이 목적이었다.

3.2 목적

- Regularization 강도를 **task0에서만 약하게** 두었을 때:
  - 첫 task image/pixel 성능이 개선되는지,
  - 이후 continual 시 forgetting 패턴이 개선되는지 확인.

3.3 방법 / 설정

- E2: feature_norm + weak reg at task0.
- E3: **feature_norm을 끄고** weak reg at task0만 적용.
- 이후 E3 변형들(Sub_E_3, metaE_4 등)은

  regularization 스케줄링/epoch 튜닝이 결합된 버전 (다음 섹션과 겹침).

3.4 결과 (E3 계열 중심)

- E3 (no feature_norm + weak reg at 1st):

Exp	Image AUROC	Pixel AUROC
E3_no_feature_norm_weak_reg	0.5676	0.8639
E3-Sub_E_3	0.6172	0.8722
E3-metaE_4	0.6282	0.8758
E3-metaE_4-subE_3	**0.6507**	**0.8776**

- 패턴:
  - Regularization을 부드럽게 조정하면서 epoch 스케줄까지 같이 튜닝했을 때
    - Pixel AUROC는 계속 우상향하여 **최대 ~87.8%** 수준까지 도달.
    - Image AUROC도 baseline(68.11%)에는 못 미치지만
      65% 수준까지 회복.
- Forgetting:
  - 일부 실험은 Avg Forgetting(Image)이 거의 0에 가까움.
    (≈ 안정적인 수준)

3.5 해석 / 시사점

- 첫 task에서 regularization을 약하게 두고,
  이후 task에서 점진적으로 강화하는 전략은
  - NF가 초기에 geometry를 더 자유롭게 학습하도록 만들고,
  - Pixel 수준에서는 분명한 성능 향상을 가져왔다.
- 다만 image 측면에서는,
  - baseline의 68% 대비 65%로 근접하지만,
    여전히 **NF density의 global calibration 문제가 남아 있는 상태**.
  - 즉 “weak reg at 1st”는 **불안정성을 줄이는 데는 도움**이 되지만
    class-wise image score 정렬 문제를 근본적으로 해결해주지는 않는다.

▼ 4. Epochs Tuning (metaE_4 등)

(관련 실험: `E3_no_feature_norm_weak_reg_metaE_4` , `metaE_4-subE_3` 등)

4.1 Motivation

- Task당 epoch 수를 1로 제한한 상태에서,
  - 첫 task가 특히 어려운 이유 중 하나가

    NF가 수렴하기엔 step 수가 너무 적다는 점.
- 하지만 전체 학습 시간 제한 때문에

  task별 epoch를 늘리긴 어렵다.
- 따라서 **정규화 스케줄(warm-up, loss weight)을 epoch/iteration 관점에서 재조정(metaE_4)** 하여,
  - 실질적으로 effective epoch를 늘린 것과 같은 효과를 노렸다.

4.2 목적

- Loss 스케줄/시점 조정만으로
  - 첫 task의 수렴 안정성,
  - 이후 task의 빠른 적응,
  - 최종 image/pixel AUROC 개선

    을 동시에 달성할 수 있는지 확인.

4.3 방법 / 설정

- `metaE_4`:
  - 첫 task에서 4 “sub-epoch” 정도로

    regularization weight/learning rate warm-up를 더 부드럽게 나눠 적용.
- `subE_3`:
  - 일부 regularization/strategy를 첫 몇 sub-epoch에만 적용하는 식으로

    더 미세하게 스케줄.

4.4 결과

- 이미 3번 섹션에서 보았듯이,
  - `metaE_4` + `subE_3` 조합(`E3_metaE_4-subE_3`)이

    image/pixel 모두 가장 좋은 성능:
    - Image AUROC: 0.6507

        - Pixel AUROC: 0.8776
- 평균 성능 트렌드 그래프에서도
    - epoch 튜닝 후 실험군들이

      “After Task2~4” 단계에서 baseline과 유사한 혹은 약간 낮은 수준의 평균 image 성능을 유지.

4.5 해석 / 시사점

- Epoch/스케줄 튜닝은
    - **NF 수렴 안정성을 크게 개선**시켜주었다.
    - Pixel AUROC는 확실히 얻을 수 있는 이득.
- 하지만 여전히 baseline 보다 image AUROC가 낮다는 것은,
    - “얼마나 잘 수렴하느냐”보다
    - “**어떤 objective로 수렴하느냐**”가 더 중요하다는 걸 보여준다.
- 즉 control을 아무리 정교하게 해도

  density 모델 구조 자체의 calibration issue를 바꾸지 않으면

  image 측면에서 ceiling을 넘기 어렵다.

▼ 5. Prototype Head

(관련 실험: `E4_no_feature_norm_weak_reg_metaE_4-prototype_head_True`, `E4_metaE_4-subE_3-prototype_head_True`)

5.1 Motivation

- 기존 class embedding e_y는 단순히 CCNF coupling의 condition으로만 사용되고 있고,
    - z와 e_y 사이에 **직접적인 alignment objective**는 없었다.
- “입력 feature/latent z는 query, class embedding은 key/prototype” 관점에서,
    - 이미지 representation이 **자기 클래스의 prototype과 가깝도록** 만드는 것이

      class-aware representation과 CL 안정성에 도움이 될 수 있다고 판단.

5.2 목적

- Prototype head를 통해
  - class embedding이 **진짜 prototype 역할**을 하도록 만들면
  - image AUROC, 특히 class간 편차 및 forgetting이 줄어드는지 확인.

5.3 방법 / 설정

- Multi-level NF latent z에서 image-level representation 추출 (pooling).
- Projection:
  - (q_i = W_q z_i), (k_y = W_k e_y)
- Prototype loss (InfoNCE/CE):

  [

  L_{\text{proto}} = CE(\text{softmax}(q_i^\top K / \tau), y_i)

  ]
- 전체 loss에 λ_proto 가중치로 추가.

5.4 결과

Exp	Image AUROC	Pixel AUROC
E3_metaE_4-subE_3 (no proto)	0.6507	0.8776
E4_metaE_4-proto	0.6449	0.8750
E4_metaE_4-subE_3-proto	0.6413	0.8785

- 패턴:
  - Pixel AUROC: 거의 동일하거나 +0.5% 수준의 미세한 이득.
  - Image AUROC: base(E3_metaE_4-subE_3=0.6507) 대비
    조금 떨어지거나 비슷한 수준(0.641~0.645).
- Forgetting:
  - prototype 유무에 따른 큰 변화는 없음.
    이미지 forgetting은 약간 개선되는 경우도 있지만
    절대 성능 차이가 작음.

5.5 해석 / 시사점

- Prototype head는
  - representation 차원에서 “class identity”를 명시적으로 만들지만,
  - **NF density calibration 문제와는 직접적인 연결이 없다.**
- 그 결과,
  - pixel 구분에는 약간의 도움을 줄 수 있어도,
  - image-level likelihood가 왜곡된 상태에서는

    AUROC ceiling을 뚫지는 못한다.
- 또한 prototype loss가 latent z에 바로 작용하기 때문에,
  - NF의 log p(x|y) 학습과 약간 충돌하여

    소폭의 image 성능 감소를 유발하기도 한다.

▼ 6. Mutual Information

(관련 실험:

`E5_no_feature_norm_weak_reg_metaE_4-subE_3-use_MI`,

`...-partial_freeze`, `...-strong_MI`, `...-weak_MI`)

6.1 Motivation

- Prototype head 실험 뒤 든 생각:
  - “class embedding을 prototype/head에만 쓰지 말고,

    feature embedding과의 mutual information을 직접 최대화해서

    class-specific feature를 강화하면 어떨까?”
- 현재 `L_mi`는 density 수준(log p(x|y_true) vs log p(x|y_neg))의 MI surrogate인데,

  feature representation과 embedding 사이의 MI는 명시적으로 다루지 않는다.

6.2 목적

- Feature ↔ class embedding 간 MI를 올리면,
  - class-specific feature subspace가 더 뚜렷해지고,

    - image-level 구분 및 continual 성능이 좋아지는지 확인.
- 특히
    - pure MI,
    - partial freeze (NF 고정 + MI만 일부 계층에),
    - strong/weak λ 설정

      의 영향을 비교.

6.3 방법 / 설정 (개념)

- Image-level feature v_i와 class embedding e_y 사이에
    - projection 후 InfoNCE 스타일 `L_mi_feat`를 추가.
- baseline: no MI_feat
- 실험:
    - use_MI: NF까지 다 학습 + MI_feat
    - use_MI-strong/weak: λ 강도만 변경
    - use_MI-partial_freeze: NF 일부 freeze 후 MI 도입

6.4 결과

Exp	Image AUROC	Pixel AUROC
use_MI	**0.4868**	**0.7227**
use_MI-strong	0.4868	0.7227
use_MI-weak	0.4868	0.7227
use_MI-partial_freeze	0.6264	0.8748

- 핵심:
    - **순수 MI 도입 (λ 크기와 무관)** →

      image/pixel 전부 **대폭 성능 붕괴** (0.48 / 0.72 수준).
    - partial_freeze를 적용하면 성능이 baseline 수준까지 회복되지만,

      여전히 baseline(0.6811)에 못 미침.
- Forgetting:
    - use_MI 계열의 forgetting 값은 절대 성능이 너무 낮아서 의미가 크지 않음.

    - partial_freeze는 forgetting 측면에서 무난하지만,

      성능상 뚜렷한 우위는 없다.
- Loss curve:
    - `L_mi_feat`가 전체 loss 스케일에서 매우 큰 비중을 차지.
    - `L_logdet_reg`, `L_noise_reg`, `L_z_reg`의 증가 패턴이 MI 실험군에서 급격하게 변화.
    - NF density learning이 MI gradient에 의해 심각하게 교란된 모습.

6.5 해석 / 시사점

- 현재 방식의 MI 도입은 **완전히 실패**했다. (그냥 단호하게 말하자)
- 이유:
    1. MI_feat gradient가 NF input/latent에 직접 작용하면서

       `log p(z)` / `log|det J|` **균형을 깨뜨림 → density가 무너짐**.
    2. MI는 "class 분리"에만 신경 쓰지

       anomaly detection 특성(OOD, subtle defect)을 고려하지 않기 때문에

       normal manifold 주변의 미세 구조를 망가뜨린다.
- MI가 유용해지려면:
    - **NF보다 앞단(encoder + adapter)에만 적용**하고,
    - NF는 **pure density learner**로 분리해야 한다.
    - 그리고 MI를 anomaly-insensitive 하게 쓰거나,

      normal data에 대해서만 representation consistency를 주는 방향으로 재설계해야 한다.

전체 정리 한 줄씩만

1. **Baseline** – CCNF 순정 구조가 CL에서 surprisingly 괜찮은 수준의 이미지·픽셀 성능을 낸다. (Image 68%, Pixel 84%)
2. **Feature Norm** – 픽셀은 좋아지지만 global density scale이 바뀌어 image AUROC는 큰 폭으로 깎인다.
3. **Weak reg at 1st** – 초기 NF 수렴을 돕고 픽셀 성능을 크게 올리지만, image 문제는 근본 해결까지는 못 간다.

4. **Epoch tuning** – 수렴을 더 부드럽게 만들어 최선 실험에서 Image 65%, Pixel 88%까지 끌어올리지만, baseline ceiling(68%)은 못 넘는다.
5. **Prototype head** – class identity는 정리되지만 NF density calibration과 직접 연결되지 않아 pixel만 약간 이득, image는 거의 변화 없음.
6. **Mutual Information (현재 방식)** – NF에 직접 물려서 density를 망가뜨린다. partial freeze로 겨우 baseline 수준까지 복구되지만 명확한 이득은 없다.

▼ E6. 단계 1 – "안 건드리는 NF" 만들기 (Backbone 고정)

1-1. 목표

- 지금까지 결과가 말해주는 건 명확함:
  - E0 baseline이 **image AUROC 최고(0.68)**.
  - E3_metaE_4-subE_3가 **pixel 최고(0.878)**.
- 둘 다 공통점: **NF 자체 구조는 그대로** 쓰고 있음.
- 문제는 task가 쌓일수록 logdet / z_std가 조금씩 드리프트하면서 image score가 깨지는 것.

그래서 1단계 목표는:

> Task0에서 NF geometry를 최대한 좋게 만든 후,
> Task1~4에서는 NF 파라미터는 얼려버리고
> class embedding / 아주 작은 adapter만 학습하는 구조로 바꾸기.

1-2. 구체 플랜

1. **Task0 전용 "pretrain"**
   - 지금 제일 잘 나오는 설정(E3_metaE_4-subE_3) 기준으로

     Task0만 여러 epoch 돌려서 NF+embedding을 튼튼하게 만든다.
   - 여기까지만 해도 image/pixel 기준 best를 한번 찍어둘 수 있음.
2. **Task1~4부터 NF freeze**

```
for ccnf in self.ccnfs:
    for name, p in ccnf.named_parameters():
```

```
            if "emb.table" not in name:   # class embedding만 예외
                p.requires_grad = False
```

- Optimizer는 **embedding, (필요하면) 작은 adapter**만 포함.
- 지금 loss 로그처럼 logdet, z_std가 task가 올라갈수록 커지면 안 된다.

  그게 이 실험의 성공 기준.

3. **NF freeze 상태에서 CL 학습했을 때,**
   - Pixel AUROC가 얼마나 유지/하락하는지,
   - Image AUROC forgetting이 얼마나 줄어드는지 체크.

> 이게 안 되면, 사실 NF 기반 continual AD 자체가 구조적으로 안 맞는 거라, 그걸 먼저 확인해 보는 의미도 있다.

주요 변경사항

1. log_prob_with_logdet 메서드 (ccnf_model.py:464-503)

- 기존: log_pz = -0.5 * (z**2).sum(dim=-1) (모든 클래스를 원점으로)
- 변경: log_pz = -0.5 * ((z - target_center)**2).sum(dim=-1) (각 클래스를 해당
  임베딩으로)
- target_center = self.emb(y) 를 사용하여 순수 클래스 임베딩을 latent space의 중심점으로 설정

1. log_prob 메서드 (ccnf_model.py:383-423)

- inference 시에도 동일한 Gaussian Mixture Prior 적용
- 학습과 추론 간 일관성 보장

1. 차원 매칭 로직 (자동 생성)

- cond_dim (128)과 x_dim (예: 768)이 다를 경우 자동으로 projection layer 생성
- Scaled identity initialization (0.1×I)으로 부드러운 전환 보장
- 학습 가능한 projection으로 최적의 매핑 학습

기대 효과

1. Image-level Score 개선

이제 모든 픽셀/패치의 latent representation (z)이 해당 클래스 임베딩 (e_y)으로
모이므로:

새로운 Score 계산 방식 (inference 시)

distance_score = -((z - e_y)**2).sum(dim=-1).mean() # Query-Key 거리 기반

1. Query-Key 유사도 구조 강화

- Query: z (입력 feature의 latent representation)
- Key: e_y (클래스 고유의 prototype)
- 학습 Loss 자체가 $z - e_y^2$를 최소화하는 Contrastive Learning 효과

1. 클래스별 성능 편차 완화

각 클래스가 latent space에서 독립적인 영역 (e_y 근처)을 점유하여 간섭 감소

다음 단계 제안

Phase 1: 빠른 검증 (Task 0 단독)

Task 0만으로 짧은 학습 실행 (5-10 epochs)

python train_continual.py --task 0 --epochs 10

Phase 2: Score 비교 분석

Inference 시 두 가지 Score를 비교:

1. Original Likelihood: log_px (기존 방식)
2. Distance Score: -||z - e_y||^2 (제안 방식, latent 거리 기반)

특히 Distance Score를 Image-level Anomaly Score로 사용했을 때 AUROC 개선 여부 확인이
핵심입니다.

Phase 3: 전체 Task 실행

성능 개선이 확인되면 전체 continual learning scenario로 확장

기술적 세부사항

- Projection Layer: 첫 forward pass 시 자동 생성 (lazy initialization)
- 메모리 효율: Projection은 bias 없이 가벼운 Linear layer 사용

- 초기화: 0.1×identity로 기존 학습된 모델과의 호환성 유지
- Optimizer: Projection layer는 자동으로 optimizer에 포함됨 (nn.Module 구조)

E7. Latent Gaussian Mixture Prior 도입 (Query-Key 구조 강화)

E9 Image AUROC부터 올리기: “스코어 정의”부터 다시 손보기

CUDA_VISIBLE_DEVICES=0 python main.py --dataset mvtec --model ccnf --strategy naive --exp_name E9_baseline

E10 : 3. class embedding을 “진짜 prototype”으로 쓰기 위한 구조 개선

CUDA_VISIBLE_DEVICES=1 python main.py --dataset mvtec --model ccnf --strategy naive --exp_name E10_baseline

E11 분리된 prototype

▼ 내용

좋지, 이제 전략 1을 **진짜 구조 관점에서** 한 번 해부해 보자.

질문 요지:

> “전략 1이 정확히 어떤 컴포넌트로 구성돼 있고,
>
> 학습/추론 단계에서 데이터·gradient·score가 어떻게 흘러가는 구조냐?”

아예 **블록 다이어그램 + 수식 + 코드 mental model**까지 같이 설명해볼게.

---

1. 전체 구조를 한 줄로 요약하면

> NF는 z=f(x)만 잘 하게 놔두고,
> 그 위에 “prototype head”를 하나 더 얹어서
> z와 class embedding μ_y를 “일방적으로” 정렬시키는 구조야.

핵심은 “일방적”이라는 거.

z → μ_y 방향으로만 정보가 가고, μ_y → z 쪽 gradient는 차단(detach).

---

2. 모듈 관점에서 본 구조

당신 코드 기준으로 컴포넌트를 다시 써보면:

(1) 기존에 이미 있는 것들

- Encoder

  ```
  image → encoder → multi-level feature maps e_l ∈ ℝ^{B×C×H×W}
  ```

- CCNF (각 feature level마다 1개씩)

  ```
  e_l (flattened) → CCNF_l → z_l, logdet_l, log_px_y_l
  ```

- ClassEmbed (ccnf_model.ClassEmbed)

  ```
  y → emb.table[y] = class embedding ∈ ℝ^{d_emb}
  ```

여기까지는 기존 E3 구조 그대로.

---

(2) 전략 1에서 새로 추가되는 것들

(a) prototype projection layer

```
proto_proj: ℝ^{d_emb} → ℝ^{d_latent}
```

- 역할: embedding 공간(d_emb)을 latent 공간(d_latent = z_dim)으로 매핑
- 각 class embedding을 “latent space에서의 prototype μ_y”로 해석하게 만드는 맵

(b) prototype alignment loss L_center

```
L_center = || z_img_detached - μ_y ||^2
```

여기서

- z_img_detached: image-level latent representation (z에서 pooling 후 detach)
- μ_y: proto_proj(emb(y))

---

3. 학습(Training) 때 데이터/gradient 흐름

3-1. Forward 패스 순서

Task 하나의 batch 기준으로:

1. **Encoder**

   ```
   features = encoder(image)  # multi-level features
   ```

2. **각 level마다 CCNF 통과**

   ```
   e_l → flatten → CCNF_l.forward → z_l, logdet_l, log_px_y_l
   ```

3. **기존 NF loss 계산 (이미 있는 것들)**
   - L_g: -log p(x|y) / D
   - L_logdet_reg
   - L_balance
   - L_z_reg
   - L_noise_reg
   - L_mi, L_e (binary/contrastive MI surrogate)

     → 이건 지금 E3에서 잘 돌아가는 그 구조 그대로.

4. **Image-level latent z_img 만들기 (여기서부터 prototype head)**

   예시:

   ```
   # z_l: (N, D) where N = B * H * W
   z_l_2d = z_l.view(B, H, W, D)              # (B, H, W, D)
   ```

```
z_img = z_l_2d.mean(dim=(1, 2))            # (B, D)  spatial pooling
z_img_detached = z_img.detach()
```

5. **class embedding → prototype μ_y 만들기**

```
e_y = ccnf_l.emb.table[y_labels]   # (B, d_emb)

if not hasattr(self, 'proto_proj'):
    self.proto_proj = nn.Linear(d_emb, D, bias=False)
    # 0.1 * identity로 초기화

μ_y = self.proto_proj(e_y)        # (B, D)
```

6. **Prototype alignment loss 계산 (z는 detach 상태)**

```
L_center = ((z_img_detached - μ_y)**2).mean()
losses['L_center'] = L_center
```

7. **총 loss에 추가 (작은 weight로)**

```
total_loss = base_loss_from_E3 + λ_center * L_center
```

3-2. Backward에서 gradient가 어떻게 흐르는지

여기가 전략 1의 핵심이다.

- **z_img_detached**
  - detach를 했기 때문에 **flow CCNF 파라미터에는 gradient가 안 감**
  - encoder/flow weight에는 L_center의 영향 0
- gradient는 어디로 가냐?
  - `proto_proj`의 weight
  - `ClassEmbed.table` (각 class embedding 벡터)

즉,

```
∂L_center/∂z = 0   (detach)
∂L_center/∂μ_y ≠ 0
```

```
∂L_center/∂proto_proj ≠ 0
∂L_center/∂emb.table ≠ 0
```

⇒ 결과적으로:

- NF의 mapping f_θ(x) = z는 **오직 density-related loss(L_g, L_logdet_reg, L_z_reg, ...)** 에만 의해 update
- prototype μ_y는 **현재 z 분포를 따라가도록** 천천히 움직임

이게 E10과 180도 다른 점:

- E10: z와 μ_y 둘 다 서로를 향해 gradient → 둘이 줄다리기 → geometry 붕괴
- 전략1: z는 자기 갈 길 가고, μ_y만 z 위치에 맞춰 조정.

---

4. 추론(Inference) 때 어떻게 쓰는지

학습 때 prototype head로 **μ_y를 latent space 상의 “class center”**처럼 학습해 놨으니,

추론 때는 이걸 anomaly score에 활용한다.

4-1. Latent distance score

1. 이미지 x 입력 → encoder → feature → CCNF → z_l
2. pooling 해서 image-level z_img (detach 필요 없음, inference니까):

   ```
   z_img = mean_over_hw(z_l)  # (B, D)
   ```

3. 해당 class y (혹은 predicted class $\hat{y}$)의 prototype μ_y 추출:

   ```
   e_y = emb.table[y]          # (B, d_emb) or (C, d_emb) if per-class
   μ_y = proto_proj(e_y)       # (B, D) or (C, D)
   ```

4. distance score:

   ```
   dist2 = ((z_img - μ_y)**2).sum(dim=-1)   # (B,)
   ```

5. 최종 image anomaly score는 hybrid로:

```
# log_px_y_image: pixel log_px_y를 top-k / mean 등으로 pooling한 값
score = α * (-dist2) + (1-α) * (-log_px_y_image)
```

- α=0.3~0.5 정도로 시작하는 걸 추천.

---

5. 이 구조가 E10/이전 prototype 방식과 뭐가 다른지 "한 장짜리 비교"

E10 / 기존 prototype head

- prior 또는 loss 안에 prototype term을 직접 넣음
  - ex) log_pz = $-0.5||z-\mu_y||^2$
  - 또는 L_proto = $||z-\mu_y||^2$를 그대로 flow에 backprop
- 결과:
  - flow가 z를 prototype 방향으로 움직임
  - 동시에 L_z_reg, L_balance, L_noise_reg는 "정규 Gaussian prior"를 유지하려고 함

    → 서로 다른 prior 두 개를 섞어놓은 꼴

    → latent geometry 찢어짐

    → logdet, z_std, log_pz 불안정

    → E10처럼 성능 하락

전략 1

- prototype loss가 **z에 gradient를 보내지 않음(detach)**
- prior / NF loss는 E3와 **완전히 동일하게 유지**
- prototype은 "현재 z를 설명하는 center" 역할만 함 (flow는 방해받지 않음)
- anomaly score에서 prototype–z distance를 auxiliary term로 사용
- NF는 여전히 "log p(x|y)"를 최대화하는 density 모델로 기능

---

6. 직관적인 그림으로 요약

머릿속 그림은 이렇게 생각하면 됨:

예전(E10)

latent 공간에서

- z는 한쪽 loss가 “이쪽으로 와!” 또 다른 loss가 “아니다 저쪽이야!” 하면서

  사방에서 끌려다니는 점 구름

전략 1

- flow는 자기 하고 싶은 대로 z 구름을 배치
- 우리는 그 z 구름을 보고

  prototype μ_y를 “z 구름의 중심으로 천천히 옮겨두는 작업”만 함
- flow는 그 과정에 전혀 간섭받지 않음

그래서:

- NF geometry는 깨끗이 유지
- prototype은 “center-of-mass classifier” 같은 역할을 얻음
- 그 둘을 hybrid score로 섞으면 image-level decision이 훨씬 안정됨

---

7. 한 줄로 다시 정리

> **전략 1의 구조는 “flow 위에 얹은, detatched z 기반 prototype head”이고,
>
> 작동 방식은 “NF는 density만, prototype은 latent 분포를 따라가는 관찰자(observer)”**다.
>
> 관찰자가 공간을 바꾸는 게 아니라,
>
> 이미 만들어진 공간 위에 “좌표축(μ_y)”만 얹는 개념이라고 보면 된다.

원하면,

지금 너의 `model.CCNF` / `ccnf_model.CCNF` 코드 기준으로

정확히 어느 함수에 어떤 줄을 넣으면 되는지 patch 형식으로도 썰어 줄게.

CUDA_VISIBLE_DEVICES=0 python main.py --dataset mvtec --model ccnf --strategy naive --exp_name E11_baseline

E12 추론 시에만” anomaly score를 log p(x|y) + (옵션) prototype distance 기반으로 다시 정

의

▼ 내용

🔍 Step 2: 동작 구조를 단계별로 정리

여기서는 "Step1이 이미 적용되어 있어서 prototype μ_y가 latent space에서 의미를 가진다"는 버전부터 설명할게.

(뒤에서 "Step1 없이 쓸 수 있는가?"를 따로 다룰 거고.)

1) Training 쪽 – 아무것도 안 바꾼다

- `model.CCNF.compute_loss`, `forward`, `compute_base_loss`

  → 지금 E3에서 쓰는 NF loss 그대로 사용:

  - `L_g`, `L_logdet_reg`, `L_balance`, `L_z_reg`, `L_noise_reg`, `L_mi`, `L_e`, ...

Step 2는 **코드 상으로는 inference 함수만 건드리는 전략**이다.

---

2) Inference 파이프라인 (per image)

기존 코드 흐름 (약간 단순화):

```
features = encoder(image)            # multi-level features
for lvl in feature_levels:
    logps, entropy, meta = ccnf.inference(features, lvl, ...)
    # logps: (N,) = log p(x|y)/D, N = B*H*W
    # 이를 (B,H,W)로 reshape 해서 pixel score로 사용
    # image-level score는 max/mean/top-k 등으로 pooling
```

여기에 Step 2에서 하는 건 크게 세 가지:

(A) NF 기반 pixel score 계산 (그대로 유지)

```
# 기존처럼
log_px_y = ccnf[lvl].log_prob(e, y_fixed, pos_embed)  # (N,)
log_px_y_norm = log_px_y / feat_dim
```

```
pixel_nf_score = -log_px_y_norm  # 높을수록 anomaly
```

이게 **NF가 학습한 원래 anomaly score**다.

여길 건드리면 E9처럼 터지는 거고, Step 2는 이걸 **살려두고 쓴다**.

(B) prototype distance score 계산 (Step1이 있을 때)

Step1에서 이미 했던 설정:

- latent z: `z = f(x, y)`
- prototype μ_y: μ_y = proto_proj(emb(y))

추론에서:

```
# 1) latent 얻기 (forward_to_latent 같은 helper를 하나 두는 걸 추천)
z = ccnf[lvl].forward_to_latent(e, y_fixed, pos_embed)  # (N, D)

# 2) image-level pooling
z_img = z.view(B, H, W, D).mean(dim=(1, 2))          # (B, D)

# 3) class prototype
e_y = ccnf[lvl].emb.table[y_fixed_img]               # (B, d_emb) or (C, d_e
mb)
mu_y = proto_proj(e_y)                               # (B, D)

# 4) distance
dist2 = ((z_img - mu_y)**2).sum(dim=-1)              # (B,)
proto_score = -dist2                                 # 작을수록 normal, 클수록 ano
maly
```

이 `proto_score`는 **class prototype 기준 "이 이미지가 class manifold에서 얼마나 벗어났냐"**를 보는 지표다.

(C) hybrid image-level score 정의

이제 이미지별 anomaly score를 이렇게 정의:

```
# NF 기반 image score
img_nf_score = aggregate(pixel_nf_score, method='topk')  # ex. top 1~2% mean

# prototype 기반 image score
img_proto_score = proto_score                    # 위에서 만든 -dist2

# Hybrid
α = 0.3 ~ 0.5   # 추천 범위
img_score = α * img_proto_score + (1 - α) * img_nf_score
```

- α=0이면 → pure NF (현재 E3 형태)
- α>0이면 → prototype 기반 calibration이 섞임

이게 Step 2의 핵심 동작이다:

> NF가 준 log-likelihood 기반 score + latent prototype distance를 섞어서 image-level decision을 더 튼튼하게 만드는 것.

🧠 Step 2의 의도 (왜 이 짓을 하냐?)

1. NF score(log_px_y)는
   - local density, patch-level anomaly엔 강함
   - class별 global calibration에 약함
2. prototype distance(-||z−μ_y||²)는
   - class center에서 얼마나 벗어났는지 잘 잡지만
   - z noise, scale, feature 분포에 민감

이 둘을 섞으면:

- NF가 **local anomaly detection 전문가**,
- prototype이 **global class calibration 전문가** 역할을 하게 된다.

둘 다 버리지 않고 hybrid로 쓰는 게 Step 2의 설계 의도.

❓ 그럼 Step2는 Step1 위에 +로 얹는 거냐, 독립적으로도 되냐?

✅ 결론부터 말하면:

1. **"full 버전" Step2 (prototype distance까지 쓰는 hybrid)는 Step1 위에 얹는 게 정석 조합이다.**
2. **하지만 Step2의 일부 아이디어는 Step1 없이도 독립적으로 시도할 수 있다.**

조금 나눠보자.

1️⃣ Step1 + Step2 = 추천되는 "정석 조합"

- Step1:
  - proto_proj + detach(z) 기반 prototype alignment로 μ_y를 latent space에서 의미 있는 center로 만들어 둠.
- Step2:
  - 그 μ_y를 써서 $||z - \mu_y||^2$를 anomaly score로 혼합.

이 조합이 의도한 그림이고,

E10의 "망한 prototype 버전"을 수학적으로 깨끗하게 고쳐 쓴 구조라고 보면 된다.

> 즉, "prototype distance term을 쓰는 Step2"는
>
> **Step1로 prototype을 먼저 제대로 만들어 둔 상태에서** 쓰는 게 맞다.

CUDA_VISIBLE_DEVICES=1 python main.py --dataset mvtec --model ccnf --strategy naive --exp_name E12_baseline

E12-1 proto distance 조정

proto distance를 -proto distance에서 proto distance**2로 수정

CUDA_VISIBLE_DEVICES=1 python main.py --dataset mvtec --model ccnf --strategy naive --exp_name E12_proto_distance-metaE_5-subE_4 --meta_epochs 4 --sub_epochs 3

결론

- 지금까지의 실험 결과 실질적으로 유효했던 것은 다음과 같음
- DSM, feature_normalization, weakly_regularization_1st
- DSM 을 적용하는 경우 전반적인 성능이 향상되었음. feature_normalization을 사용하는 경우 pixel_level의 성능은 향상되나 image-level의 성능은 변화가 없거나 유사함
- 첫 task에서 weakly regularization을 적용하는 경우 전반적인 성능 저하
    - 아마 regularization term이 너무 많아 첫 task에서 학습을 거의 못 해서 성능이 저하되며, 이를 적절히 풀어줌으로써 성능이 향상 됨
- class 별 class embedding이 NF에 직접 주입되면서 $|\det J|$, likelihood가 급변히 변하게 되고, 이를 규제하기 위해 다수의 regularization term을 사용. 이를 통해 결과적으로 전반적인 성능은 향상 시키고, forgetting은 완화되었으나 첫 번째 task에서의 적응력이 매우 떨어짐
- 다만 현재 장치에서 class embedding에 class-specific feature를 학습 또는 저장하는 장치가 매우 부족. 현재는 최종 likelihood maximization에 의해 간접적으로 class embedding이 class-specific feature를 학습하도록 유도하는 방식이지, 직접 주입하는 방식은 아님
- 이러한 방식 때문에 inference하는 과정에서 class embedding을 선택하는 것에 있어서 문제가 생김. inference 과정에서 입력된 feature를 query로써 사용하여 class embedding 을 선택하게 되나, query로 사용되는 feature와 class embedding 간의 유사도 최대화와 같은 별도의 querying 장치가 없기 때문에 잘못 선택될 확률이 존재