Linee Guida Tesi

Introduzione

Lavoriamo in modo *self/supervised* ossia senza etichette di patologia nel dataset e senza classificatore esterno, usando solo:

- immagine I_i
- suo referto T_i
- altri referti T_j e immagini I_j nel batch come "distrattori".

Variante del modello che fa esattamente questo: scegliere il minimo numero di patch tali che l'immagine "mascherata" riesca ancora a distinguere il proprio referto da tutti gli altri referti nel batch.

1 Setup: solo coppie (immagine, referto), zero etichette strutturate

Dataset:

$$\mathcal{D} = \{(I_i, T_i)\}_{i=1}^{N}$$

- Nessuna label "cardiomegalia = 1" ecc.
- Non vogliamo un classificatore esterno per pseudo-label → niente "silver labels".

Usiamo solo il fatto che: **il referto giusto di I_i è T_i e non è nessun altro T_j nel batch.** Questo è esattamente lo scenario del contrastive learning tipo CLIP/InfoNCE.

2 Encoder + maschera (come prima)

Encoder immagine f_θ (ViT):

$$f_\theta(I_i) = V_i = [v_{i1}, \ldots, v_{iM}], \quad v_{ij} \in \mathbb{R}^d$$

Maschera sui patch (parametrizzata dal modello):

$$z_{ij} = \sigma(w_z^\top v_{ij} + b_z) \in (0, 1)$$

Patch mascherati:

$$\tilde{v}_{ij} = z_{ij}\, v_{ij}$$

Pooling globale (su patch mascherati):

$$\tilde{v}_i = \text{pool}(\{\tilde{v}_{ij}\}_{j=1}^{M})$$

Encoder testo g_ϕ (BERT):

$$u_i = g_\phi(T_i) \in \mathbb{R}^d$$

(simbolo unico per il referto, es. [CLS]).

3 Similarità immagine–testo e InfoNCE "mascherato"

Definiamo la similarità tra immagine mascherata e testo:

$$s_{ij}(\mathbf{z}_i) = \frac{\tilde{v}_i(\mathbf{z}_i)^\top u_j}{\tau}$$

dove $\mathbf{z}_i = (z_{i1}, \ldots, z_{iM})$.

Per l'esempio i, InfoNCE standard:

$$\mathcal{L}^{(i)}_{\text{NCE}}(\mathbf{z}_i) = -\log \frac{\exp(s_{ii}(\mathbf{z}_i))}{\sum_{j=1}^{B} \exp(s_{ij}(\mathbf{z}_i))}$$

Con solo questa loss, il modello impara:

- a fare in modo che $I_i \odot \mathbf{z}_i$ sia più vicino al suo testo T_i,
- e più lontano dagli altri testi T_j nel batch.

Quindi stai già "usando gli altri esempi" per capire cosa distingue I_i, T_i da I_k, T_k.

4 Aggiungere l'idea "minimo numero di patch" in modo puramente self-supervised

Qui arriva il trucco per far sì che la maschera faccia esattamente ciò che chiedi: **scegliere il sottoinsieme minimo di patch che conserva il potere discriminativo rispetto agli altri testi del batch.**

Definiamo tre termini per ogni esempio i:

4.1 InfoNCE su immagine completa (baseline di riferimento)

Calcoliamo la NCE anche sulla immagine completa (senza maschera):

$$\bar{v}_i = \text{pool}(\{v_{ij}\})$$

$$\bar{s}_{ij} = \frac{\bar{v}_i^\top u_j}{\tau}$$

$$\mathcal{L}^{(i)}_{\text{NCE-full}} = -\log \frac{\exp(\bar{s}_{ii})}{\sum_{j=1}^{B} \exp(\bar{s}_{ij})}$$

Questa dice: "con tutte le informazioni disponibili, quanto bene distinguo T_i dagli altri T_j?".

4.2 InfoNCE su immagine mascherata

Questo è quello di prima:

$$\mathcal{L}^{(i)}_{\text{NCE-mask}}(\mathbf{z}_i) = -\log \frac{\exp(s_{ii}(\mathbf{z}_i))}{\sum_{j=1}^{B} \exp(s_{ij}(\mathbf{z}_i))}$$

Vuoi che, usando solo i patch selezionati da $\mathbf{z}_i$, il modello mantenga (quasi) la stessa capacità di distinguere T_i dagli altri T_j.

4.3 Vincolo di sparsità sulla maschera

$$\mathcal{L}^{(i)}_{\text{sparse}} = \|\mathbf{z}_i\|_1 = \sum_{j=1}^{M} z_{ij}$$

Questo spinge a diminuire il numero (o il peso) di patch attivi.

4.4 Consistenza "immagine piena vs mascherata"

Per evitare che la maschera "rovini" l'informazione, imponiamo che:

- la similarità col proprio testo resti alta,
- il ranking rispetto agli altri testi (negativi) cambi il meno possibile.

Un modo semplice:

$$\mathcal{L}_{\text{cons}}^{(i)}(\mathbf{z}_i) = \left(s_{ii}(\mathbf{z}_i) - \bar{s}_{ii}\right)^2$$

cioè la similarità mascherata con il proprio testo deve rimanere vicina a quella full.

Volendo puoi aggiungere anche: $\sum_{j \neq i} \left(s_{ij}(\mathbf{z}_i) - \bar{s}_{ij}\right)^2$, ma spesso basta concentrare il vincolo sul positivo.

5 Loss finale per ogni esempio (puramente self-supervised)

Metti tutto insieme:

$$\mathcal{L}_{\text{patch-IB}}^{(i)} = \underbrace{\mathcal{L}_{\text{NCE-full}}^{(i)}}_{\text{impara embedding buoni}} + \underbrace{\mathcal{L}_{\text{NCE-mask}}^{(i)}(\mathbf{z}_i)}_{\text{maschera deve mantenere discriminatività}} + \lambda \underbrace{\mathcal{L}_{\text{sparse}}^{(i)}}_{\text{pochi patch}} + \mu \underbrace{\mathcal{L}_{\text{cons}}^{(i)}(\mathbf{z}_i)}_{\text{non perdere info rilevante}}$$

Sommi su tutto il batch:

$$\mathcal{L}_{\text{patch-IB}} = \frac{1}{B} \sum_{i=1}^{B} \mathcal{L}_{\text{patch-IB}}^{(i)}$$

e la aggiungi alle altre (se le hai) tipo global contrastive, reconstruction, ecc.

6 Cosa sta succedendo davvero (intuitivamente)

- **NCE-full:** fa sì che $\bar{v}_i$ contenga tutta l'informazione che rende I_i diverso dagli altri, rispetto ai testi.
- **NCE-mask + consistenza:** obbliga la maschera a tenere patch sufficienti per ottenere quasi lo stesso score rispetto a T_i (e negativi) solo con $I_i \odot \mathbf{z}_i$.
- **Sparsità:** spinge a ridurre il numero di patch tenuti.
- **Batch:** fornisce il contesto di "altre immagini + referti" che rendono chiaro cosa rende unico I_i, T_i.

Quindi, senza etichette di patologia, la maschera impara a: **selezionare le regioni minime dell'immagine che mantengono il potere discriminativo di riconoscere il proprio referto e distinguerlo dagli altri nel batch.**

Questo è esattamente "fare leva su ciò che più caratterizza e distingue un'immagine e referto dagli altri scelti casualmente".

7 Ha senso anche senza "positivi multi-immagine"?

Sì:

- Nel caso precedente (con etichette di patologia) usavamo più immagini con la stessa patologia come multi-positivi.
- Qui non abbiamo quell'informazione, quindi assumiamo che ogni (I, T) sia il proprio "prototipo" (instance discrimination).

- Il modello, per minimizzare la NCE, deve:
  - estrarre ciò che nel referto è specifico e
  - trovare regioni visive che lo supportano, rispetto agli altri testi nel batch.

Nel tempo, la rete condivide i pesi su tutto il dataset, quindi non può "barare" su un singolo esempio: deve imparare regole generali che funzionino per molte coppie.

8 Varianti possibili (sempre self-supervised, nessuna label esterna)

Se vuoi spingerti oltre, restando unsupervised:

1. **Clustering dei testi:**
   - fai un clustering degli embedding dei referti (solo testo), totalmente self-supervised;
   - usi l'ID di cluster come "pseudo-patologia" per introdurre anche una componente tipo MICL multi-immagine;
   - è ancora self-supervised: niente classificatore esterno.
2. **Augmentazioni multi-view dell'immagine:**
   - crei 2–3 viste augmentate dell'immagine (crop, jitter, ecc.);
   - imponi che le maschere apprese sulle diverse viste producano embedding simili per lo stesso referto;
   - rinforza la robustezza dei patch selezionati.

Ma il nucleo fondamentale che risponde alla domanda è già nella loss $\mathcal{L}_{\text{patch-IB}}$.

9 Come determino i referti con la stessa patologia senza etichette?

Non li determini esplicitamente: usi solo la coppia (I_i, T_i) vs gli altri (I_j, T_j) nel batch, con una loss contrastiva.

La maschera viene ottimizzata per preservare il potere discriminativo di questa coppia rispetto alle altre, con un vincolo di sparsità: nessuna label richiesta.

Il local alignment patch–token è una parte essenziale della soluzione perché forza il modello a imparare *come* il contenuto testuale si mappa sulle regioni visive, mentre la loss Patch-IB (Information Bottleneck spaziale) forza *quali* patch sono necessari per distinguere l'immagine dal resto del batch.

Non era incluso nel primo pseudocodice per tenere la parte Patch-IB chiara e isolata, ma **segue l'integrazione nell'architettura finale**, con:

1. **allineamento globale immagine–testo (InfoNCE full)**
2. **allineamento locale patch–token (cross-attention supervision)**
3. **bottleneck spaziale con maschera sparsa (Patch-IB)**

In questo modo otteniamo un modello completo che soddisfa tutti i vincoli.

10 Architettura completa con Local Alignment incluso

IMMAGINE I

Image Encoder f_theta (ViT)

V \in R^{M x d} patch embeddings

(A) FULL EMBEDDING

v_full = pool(V)

(B) MASKED EMBEDDING

z = MaskHead(V)
V_mask = z \odot V
v_mask = pool(V_mask)

TESTO T

Text Encoder g_phi (BERT)

token embeddings U = [u_1 ... u_L], u_CLS

global u = u_CLS

LOCAL ALIGNMENT PATCH-TOKEN (CROSS ATTENTION)

CrossAttention(queries = U, keys = V, values = V)
-> produce attention map A \in R^{L x M}
-> produce text-aligned patch summary \tilde{v}_k
-> loss_locale = \Sigma_k || \tilde{v}_k - u_k ||^2

LOSS COMPLESSIVE

1) InfoNCE globale (v_full <-> u)
2) InfoNCE mascherata (v_mask <-> u)
3) Sparsità patch (||z||_1)
4) Consistenza full vs mask
5) Local alignment patch-token

11 Pseudo-codice completo con local alignment incluso

11.1 Step A — Cross-attention patch–token

Aggiungiamo un modulo per il local alignment:

```
class LocalAlign(nn.Module):
    def __init__(self, d_model, n_heads=4):
        super().__init__()
        self.attn = nn.MultiheadAttention(d_model, n_heads, batch_first=True)
        self.lin = nn.Linear(d_model, d_model)  # proiezione finale opzionale

    def forward(self, U, V):
        # U: (B, L, d)  tokens
        # V: (B, M, d)  patches
        # Query=U, Key=V, Value=V
        out, attn = self.attn(U, V, V)          # out: (B, L, d), attn: (B, L, M)
        out = self.lin(out)
        return out, attn
```

11.2 Step B — Loss locale patch–token

La loss locale implementa:

$$\tilde{v}_{ik} = \sum_j A_{kj} v_{ij} \quad , \quad L_{\text{local}} = \sum_k \|\tilde{v}_{ik} - u_{ik}\|^2$$

In codice:

```
def local_loss(V, U, attn, out):
    # V: (B, M, d)
    # U: (B, L, d)
    # out: text-aligned vectors (B, L, d)
    # attn: attention weights (B, L, M)

    # attn sono pesi -> v_tilde_k = sum_j alpha_kj * v_j
    V_expanded = V.unsqueeze(1)              # (B,1,M,d)
    attn_exp = attn.unsqueeze(-1)            # (B,L,M,1)
    V_tilde = (attn_exp * V_expanded).sum(dim=2)  # (B,L,d)

    return F.mse_loss(V_tilde, U)
```

11.3 Step C — Forward completo del modello

```
class FullPatchIBModel(nn.Module):
    def __init__(self, img_encoder, txt_encoder, d_model, temp=0.07):
        super().__init__()
        self.img_encoder = img_encoder
        self.txt_encoder = txt_encoder
        self.mask_head = MaskHead(d_model)
        self.local_align = LocalAlign(d_model)
        self.temp = temp

    def forward(self, images, input_ids, attention_mask):
        # --- 1. Encode image ---
```

```
        V = self.img_encoder(images)                    # (B, M, d)

        # --- 2. Encode text ---
        U_all = self.txt_encoder.bert(input_ids,
                                      attention_mask).last_hidden_state
        U_all = self.txt_encoder.proj(U_all)            # (B, L, d)
        u_cls = U_all[:,0]                              # (B, d)
        u_cls = F.normalize(u_cls, dim=-1)

        # --- 3. Full embedding ---
        v_full = patch_pool(V)
        v_full = F.normalize(v_full, dim=-1)

        # --- 4. Masked embedding ---
        z = self.mask_head(V)                           # (B, M, 1)
        V_mask = V * z
        v_mask = patch_pool(V_mask)
        v_mask = F.normalize(v_mask, dim=-1)

        # --- 5. Local alignment ---
        U_local, attn = self.local_align(U_all, V)  # (B, L, d), (B, L,
    M)

        return {
            "V": V,
            "U_all": U_all,
            "u_cls": u_cls,
            "v_full": v_full,
            "v_mask": v_mask,
            "z": z,
            "U_local": U_local,
            "attn": attn
        }
```

11.4 Step D — Loss totale con local alignment incluso

```
def full_loss(outputs, lambda_sparse=1e-3, mu_cons=1.0,
              lambda_local=1.0, temp=0.07):

    v_full = outputs["v_full"]
    v_mask = outputs["v_mask"]
    u_cls  = outputs["u_cls"]
    V      = outputs["V"]
    U_all  = outputs["U_all"]
    U_local= outputs["U_local"]
    attn   = outputs["attn"]
    z      = outputs["z"]

    # ----- 1) InfoNCE full -----
    logits_full = sim_matrix(v_full, u_cls, temp)
    L_nce_full  = info_nce_loss(logits_full)

    # ----- 2) InfoNCE mask -----
    logits_mask = sim_matrix(v_mask, u_cls, temp)
    L_nce_mask = info_nce_loss(logits_mask)

    # ----- 3) sparsit  -----
```

```
    L_sparse = z.abs().mean()

    # ----- 4) consistency -----
    pos_full = logits_full.diag()
    pos_mask = logits_mask.diag()
    L_cons = F.mse_loss(pos_mask, pos_full)

    # ----- 5) local alignment -----
    L_local = local_loss(V, U_all, attn, U_local)

    # ----- totale -----
    L = L_nce_full + L_nce_mask \
        + lambda_sparse * L_sparse \
        + mu_cons * L_cons \
        + lambda_local * L_local

    return {
        "L_total": L,
        "L_nce_full": L_nce_full,
        "L_nce_mask": L_nce_mask,
        "L_sparse": L_sparse,
        "L_cons": L_cons,
        "L_local": L_local
    }
```

12 Perché *deve* esserci il local alignment?

Per tre motivi:

12.1 A) L'Information Bottleneck spaziale individua il "minimo insieme di patch"

ma non dice **quali** patch corrispondono a quali parti del referto. È solo una pressione:

- tieni pochi patch,
- ma tieni quelli che mantengono discriminatività di immagine–referto.

Non riguarda la semantica "frase X → regione Y".

12.2 B) Il contrastivo globale non è sufficiente per grounding semantico

Il contrastivo distingue coppie (I,T) tra loro, non patch e token. Serve un vincolo di forma:

$$\tilde{v}_{ik} \approx u_{ik}$$

per ogni token (o frase) importante del referto.

12.3 C) Le mappe di attenzione patch–token diventano interpretative e clinicamente utili

Senza local alignment, le attention map sono:

- rumore,
- non robuste,

- non diagnostiche.

Con local alignment, diventano:

- localizzazioni consistenti,
- grounding cross-immagine,
- supervisione debole ma efficace.

13 Architettura finale

Patch-IB (per la parsimonia) + Local Alignment (per il grounding) + Global CLIP (per discriminazione)

Questa combinazione è ciò che eleva il modello:

- oltre ConVIRT (che ha solo global CLIP),
- oltre GLoRIA (che ha solo local alignment, senza bottleneck),
- oltre modelli VLM 2024–25 (che raramente integrano un IB spaziale vero).

14 Ottimizzazione GPU: Top-K e Token Dropping

Ora, sfruttiamo più efficientemente la GPU:

- usa la maschera per selezionare solo un sottoinsieme di patch (Top-K),
- applica le parti pesanti (local alignment, bottleneck) solo su quei patch,
- può anche, se vuoi, potare token dentro il ViT negli ultimi layer (stile TokenLearner / EViT).

Separiamo la cosa in due livelli:

1. Livello A – ottimizzi la testa multimodale (bottleneck + local alignment).
2. Livello B – opzionale, più aggressivo – fai token dropping dentro il backbone.

14.1 Idea di base: Top-K patch "salienti" per ogni immagine

Invece di usare tutti i M patch:

- calcoli una salienza per patch (i logits della mask head),
- scegli i K patch più importanti ($K \ll M$),
- usi solo quelli per:
  - v_mask,
  - local alignment patch–token,
  - eventuali altre teste.

Schema concettuale:

```
Immagine I


ViT → V \in R^{M x d}

    Full path (tutti i patch) → v_full              (serve solo per L_NCE_full)

    Mask head (leggera) → score s \in R^{M}


        Top-K selezione


       V_sel \in R^{K x d}, z_sel \in {0,1}^K

    pool(V_sel) → v_mask  (per Patch-IB)
    local alignment con token testo (solo su K patch)
```

14.2 Livello A – Versione ottimizzata della testa (senza toccare il ViT)

14.2.1 Modifica della MaskHead: produci score, poi Top-K

Invece di usare direttamente sigmoid sui logits, usi i logits per fare ranking:

```
class MaskHeadTopK(nn.Module):
    def __init__(self, d_model, k_ratio=0.25):
        super().__init__()
        self.linear = nn.Linear(d_model, 1)
        self.k_ratio = k_ratio   # es. 0.25 = tieni il 25% dei patch

    def forward(self, V):
        # V: (B, M, d)
        logits = self.linear(V).squeeze(-1)   # (B, M)

        B, M = logits.shape
        K = max(1, int(M * self.k_ratio))

        # Top-K per ogni immagine
        topk_vals, topk_idx = torch.topk(logits, K, dim=1)   # (B, K)

        # costruiamo una mask binaria (straight-through)
        hard_mask = torch.zeros_like(logits)                 # (B, M)
        hard_mask.scatter_(1, topk_idx, 1.0)                 # 1 sui top
    -K

        # opzionale: componente soft per gradienti (straight-through
    estimator)
        soft_mask = torch.sigmoid(logits)                    # (B, M)
        z = hard_mask + (soft_mask - soft_mask.detach())     # (B, M)

        return z, topk_idx
```

- `hard_mask` = dove davvero tieni (1) o butti (0) patch.
- `soft_mask` = per avere gradienti (straight-through): in forward si comporta come hard_mask, in backward come soft.

14.2.2 Selezione patch per i moduli pesanti

```
class PatchIBTopKModel(nn.Module):
    def __init__(self, img_encoder, txt_encoder, d_model, temp=0.07,
    k_ratio=0.25):
        super().__init__()
        self.img_encoder = img_encoder
        self.txt_encoder = txt_encoder
        self.mask_head = MaskHeadTopK(d_model, k_ratio=k_ratio)
        self.local_align = LocalAlign(d_model)
        self.temp = temp

    def forward(self, images, input_ids, attention_mask):
        # 1) encode image
        V = self.img_encoder(images)          # (B, M, d)

        # 2) encode text (tutti i token)
        bert_out = self.txt_encoder.bert(input_ids, attention_mask)
        U_all = self.txt_encoder.proj(bert_out.last_hidden_state)  # (B,
    L, d)
```

```
        u_cls = U_all[:, 0]                                          # (B, d)
        u_cls = F.normalize(u_cls, dim=-1)

        # 3) full embedding (tutti i patch)
        v_full = patch_pool(V)                                       # (B, d)
        v_full = F.normalize(v_full, dim=-1)

        # 4) mask + Top-K
        z, topk_idx = self.mask_head(V)            # z: (B, M), topk_idx: (B, K)
        z_exp = z.unsqueeze(-1)                    # (B, M, 1)
        V_mask = V * z_exp                         # (B, M, d)
        # pool sui patch selezionati (usa la mask, non tutti)
        # normalizzazione per non biasare con K:
        denom = z_exp.sum(dim=1).clamp(min=1e-6)   # (B,1,1)
        v_mask = (V_mask.sum(dim=1) / denom.squeeze(-1))  # (B, d)
        v_mask = F.normalize(v_mask, dim=-1)

        # 5) local alignment SOLO sui patch selezionati (V_sel)
        # gather: (B, K, d)
        B, M, d = V.shape
        K = topk_idx.shape[1]
        idx = topk_idx.unsqueeze(-1).expand(-1, -1, d)  # (B,K,d)
        V_sel = torch.gather(V, 1, idx)                 # (B, K, d)

        # local attention: U_all vs V_sel
        U_local, attn = self.local_align(U_all, V_sel) # attn: (B, L, K)

        return {
            "V": V,
            "V_sel": V_sel,
            "U_all": U_all,
            "u_cls": u_cls,
            "v_full": v_full,
            "v_mask": v_mask,
            "z": z,
            "U_local": U_local,
            "attn": attn
        }
```

Risparmio GPU:

- la MultiheadAttention ora lavora su K patch invece che M: complessità $\sim O(B \cdot L \cdot K \cdot d)$ vs $O(B \cdot L \cdot M \cdot d)$.
- per $K = M/4$, hai $\sim 4\times$ meno costi di local alignment.

La loss `full_loss` resta praticamente identica, solo che ora `local_loss` usa V_{sel} e attn (B, L, K) al posto di $V(B, M, d)$.

```
def local_loss(V_sel, U_all, attn, U_local):
    # V_sel: (B, K, d), attn: (B, L, K)
    B, K, d = V_sel.shape
    V_expanded = V_sel.unsqueeze(1)          # (B,1,K,d)
    attn_exp = attn.unsqueeze(-1)            # (B,L,K,1)
    V_tilde = (attn_exp * V_expanded).sum(dim=2)  # (B,L,d)

    return F.mse_loss(V_tilde, U_all)
```

14.3 Livello B – Token dropping dentro il ViT (più aggressivo)

Questa è la versione "pro" se vuoi davvero tagliare FLOPs, non solo sulla testa.

14.3.1 Idea tipo TokenLearner / EViT

Nel ViT:

- dopo alcuni layer (es. dopo il 4° o 6°),
- calcoli score per patch (con un piccolo MLP sulle feature correnti),
- usi Top-K per tenere solo qualche token per i layer successivi,
- ripeti magari più volte.

Struttura:

```
Input patches → ViT layer 1..L1 → V^(L1)

                    Mask head + Top-K

                        V_sel^(L1)

              ViT layer L1+1 .. Lend (solo su K token)
```

Questo riduce astrologicamente il costo:

- i layer più profondi (costosi) vedono solo K token,
- gli strati iniziali vedono tutti i patch (servono per bene l'estrazione).

14.3.2 Pseudo-scheletro ViT con dropping

```python
class TokenDroppingViT(nn.Module):
    def __init__(self, base_vit, d_model, drop_layer=6, k_ratio=0.25):
        super().__init__()
        self.blocks_before = base_vit.blocks[:drop_layer]
        self.blocks_after  = base_vit.blocks[drop_layer:]
        self.norm = base_vit.norm
        self.patch_embed = base_vit.patch_embed
        self.cls_token = base_vit.cls_token
        self.pos_embed = base_vit.pos_embed
        self.mask_head = MaskHeadTopK(d_model, k_ratio)

    def forward(self, x):
        # x: (B, C, H, W)
        B = x.size(0)
        x = self.patch_embed(x)             # (B, M, d)
        cls_tokens = self.cls_token.expand(B, -1, -1)  # (B,1,d)
        x = torch.cat((cls_tokens, x), dim=1)            # (B, M+1, d)
        x = x + self.pos_embed[:, :x.size(1), :]

        # primi layer con tutti i token
        for blk in self.blocks_before:
            x = blk(x)

        # separa CLS dai patch
```

```
        cls_tok, patches = x[:, :1, :], x[:, 1:, :]      # (B,1,d), (B,M,
    d)

        # MaskHead sui patch
        z, topk_idx = self.mask_head(patches)            # z: (B,M)
        # selezione top-K patch
        B, M, d = patches.shape
        K = topk_idx.shape[1]
        idx = topk_idx.unsqueeze(-1).expand(-1, -1, d) # (B,K,d)
        patches_sel = torch.gather(patches, 1, idx)    # (B,K,d)

        # ricomponi sequenza: [CLS] + patch selezionati
        x = torch.cat((cls_tok, patches_sel), dim=1)   # (B, K+1, d)

        # layer rimanenti solo su K+1 token
        for blk in self.blocks_after:
            x = blk(x)

        x = self.norm(x)
        cls_tok, patches = x[:, :1, :], x[:, 1:, :]
        return patches    # (B, K, d)
```

Poi questo `TokenDroppingViT` lo usi come `img_encoder` nel modello precedente.

Nota: i gradienti verso la MaskHead qui sono un po' più delicati, ma lo stesso trick di straight-through funziona (non perfetto teoricamente, ma funziona empiricamente).

14.4 Trade-off e consigli pratici

14.4.1 Cosa fare per iniziare (safe & easy)

1. Versione A solo testa (Top-K su V dopo il ViT, senza toccare il backbone):
   - molto più semplice da debuggare;
   - riduce già il costo della parte multimodale.
2. In questa fase:
   - gioca con k_ratio = 0.25, 0.5, 0.75;
   - annealing: parti con k_ratio più alto e poi riduci (stile pruning progressivo).

14.4.2 Se funziona e con più speed

3. Passa a token dropping dentro il ViT:
   - dopo 4–6 block,
   - usa la stessa MaskHeadTopK sulle feature intermedie,
   - tieni ad es. K = 1/4–1/2 dei token originali.

14.4.3 Aspetti per la GPU

- Mixed precision (FP16 / BF16) con `torch.cuda.amp.autocast()`.
- Gradient checkpointing sui blocchi del ViT.
- Batch size ragionevoli (es. 32 su 24/32GB, 16 su 12GB).
- Eventualmente riduci L (max tokens testo) con truncation.

14.5 Intuizione finale (in linea con l'idea)

Quello che stai costruendo è: un modello che impara a selezionare pochi patch stabili e ricorrenti, perché deve:

- preservare la discriminatività immagine–referto (Patch-IB),
- mantenere l'allineamento semantico patch–token,
- farlo in maniera efficiente grazie al Top-K.

La versione con subset di patch:

- non perde in efficacia se scegli K abbastanza grande e magari lo riduci gradualmente,
- guadagna molto in efficienza riducendo il costo del local alignment e, eventualmente, dei layer profondi del ViT.

15 Piano di esperimenti / Ablation Study

Facciamo:

1. Un piano di esperimenti / ablation study per verificare Patch-IB + local alignment + Top-K / token dropping.
2. Come monitorare se i patch selezionati hanno senso clinico, con e senza annotazioni di localizzazione.

15.1 Piano esperimenti (ablation + efficienza)

15.1.1 Dataset & setting di base

- Dataset radiologico con immagini + referti (es. MIMIC-CXR, CheXpert).
- Addestramento solo da immagini + referti, come da tuo scenario.
- Valutazioni:
  - Retrieval immagine→testo e testo→immagine (R@1, R@5, R@10).
  - Classification (AUC per patologia) con: linear probe su embedding immagine congelato, o fine-tuning leggero di una testa di classificazione.

Obiettivo: verificare che i vari blocchi non rovinino (o migliorino) la qualità globale del modello, mentre guadagni localizzazione ed efficienza.

15.1.2 Varianti del modello (ablation)

Ti suggerisco questa progressione:

Modello A — Baseline CLIP-like

- Sì: encoder immagine (ViT), encoder testo (BERT), InfoNCE globale (v_full ↔ u_cls).
- No: maschera, Patch-IB, local alignment, Top-K.

Serve per avere i numeri di riferimento: R@K, AUC, tempo per step, memoria occupata.

Modello B — + Local Alignment (senza maschera)

- Aggiungi solo il modulo di cross-attention patch–token + loss locale: $L_{\text{local}} = \|\tilde{v}_{ik} - u_{ik}\|^2$
- Niente maschera, niente Patch-IB.

Confronti con A: Retrieval / AUC: dovrebbero restare simili o migliorare. Già qui puoi vedere se l'attenzione patch–token produce mappe sensate (visivamente).

Modello C — + Patch-IB (maschera sparsificata, senza Top-K)

- Mantieni local alignment.
- Aggiungi: maschera z sui patch, v_mask, InfoNCE su v_mask, perdita di sparsità (L1), perdita di consistenza (sim_full vs sim_mask).

Loss globale del tipo:

$$L = L_{\text{NCE-full}} + L_{\text{NCE-mask}} + \lambda_{\text{sparse}} L_{\text{sparse}} + \mu L_{\text{cons}} + \lambda_{\text{local}} L_{\text{local}}$$

Qui misuri: Qualità globale (retrieval, AUC), Sparsità effettiva, Costo computazionale.

Modello D — + Top-K nella testa (ottimizzazione GPU "soft")

- Maschera come MaskHeadTopK: usi tutti i patch per v_full, ma solo Top-K patch per v_mask e local alignment.

Confronti C vs D: Retrieval / AUC non devono crollare. Misuri: tempo per batch, uso GPU, FLOPs. Fai sweep su k_ratio (es. 1.0, 0.75, 0.5, 0.25).

Modello E — Token Dropping nel ViT (ottimizzazione "hard") Se tutto sopra funziona e vuoi andare oltre:

- introduci il dropping di patch dentro il ViT: dopo N layer, calcoli score sui patch, tieni solo K token per i layer successivi.

Confronti D vs E: Se E mantiene quasi gli stessi risultati di D ma con ulteriore riduzione di tempo e memoria, allora hai dimostrato che il modello non solo localizza, ma lo fa in modo effettivamente più efficiente.

15.1.3 Hyper da esplorare (minimo set)

- λ_{sparse} (es. 1e-4, 1e-3, 1e-2).
- μ_{cons} (es. 0.1, 1.0, 5.0).
- λ_{local} (es. 0.1, 1.0).
- k_{ratio} (top-K): 1.0, 0.75, 0.5, 0.25.

15.1.4 Metriche di efficienza

Per ogni modello (A–E): Tempo per step, Immagini/s, Memoria GPU, (FLOPs). Plot: k_ratio vs R@1, k_ratio vs tempo/step, k_ratio vs memoria.

15.2 Come capire se i patch selezionati sono "clinicamente sensati"

Qui facciamo due casi: con e senza annotazioni di localizzazione.

15.2.1 Con annotazioni (anche su un sotto-dataset piccolo)

Se riesci ad avere per qualche centinaio/migliaio di immagini bounding box o maschere di lesioni.

a) IoU / Dice a livello di heatmap

- Prendi la maschera z, Upsampling a risoluzione immagine, Soglia (es. top 20% dei pixel).
- Metriche: IoU, Dice score.
- Confronti: Modello A vs B, C, D, E.

b) "Pointing game" Più semplice se hai bbox: Trovi il punto di massima attivazione della heatmap. Verifichi se cade dentro un bbox GT. Statistiche: % di "hit" per patologia.

c) Coverage vs precision Per ogni immagine: Copertura (% del bbox GT coperta), Precisione (% delle regioni con z alto che cadono fuori da regioni normali).

15.2.2 Senza annotazioni di localizzazione (scenario realistico)

Qui vai di proxy metrics + valutazione umana.

a) Deletion / insertion test (faithfulness) Per ogni immagine–referto:

1. Ordina i patch per z.

2. Deletion test: parti dall'immagine completa, rimuovi progressivamente i patch con z più alto, traccia come decresce la similarità.

3. Insertion test: parti da immagine vuota, inserisci progressivamente i patch con z più alto, traccia come cresce lo score.

Un buon sistema ha: curva deletion che scende velocemente, curva insertion che sale velocemente.

b) Sparsità e entropia della maschera Metriche semplici:

- Sparsità media: sparsità $= \frac{1}{BM} \sum_{i,j} \mathbb{1}[z_{ij} > \tau]$.
- Entropia spaziale: bassa entropia $\rightarrow$ pochi patch concentrano la massa $\rightarrow$ localizzazione netta.

c) Consistenza tra viste (robustezza) Per ogni immagine: Genera 2–3 augmentazioni, calcola la maschera z, riporta le maschere nello stesso sistema di coordinate, misura Jaccard o cosine similarity. Se il modello ha imparato qualcosa di reale: patch importanti resteranno simili tra viste diverse.

d) Consistenza tra pazienti con referti simili (indirettamente) Senza etichette:

1. Calcolare gli embedding testuali u per tutti i referti.

2. Fare un clustering o trovare k-NN in spazio testo.

3. Per ogni "gruppo" di referti simili, misurare la similarità tra maschere z delle immagini.

Se il modello ha imparato pattern ricorrenti: gruppi di referti simili $\rightarrow$ maschere simili.

16 Integrazione piano esperimenti + interpretabilità

In pratica puoi strutturare il lavoro così:

1. **Fase 1 – Performance globale:** Allenare A $\rightarrow$ E. Misurare retrieval + AUC + efficienza.

2. **Fase 2 – Qualità di localizzazione (senza labels):** Deletion / insertion, sparsità, consistenza cross-view e cross-referti. Mostrare qualche heatmap qualitativa.

3. **Fase 3 – Validazione con annotazioni (se disponibili):** IoU / pointing game su subset annotato.