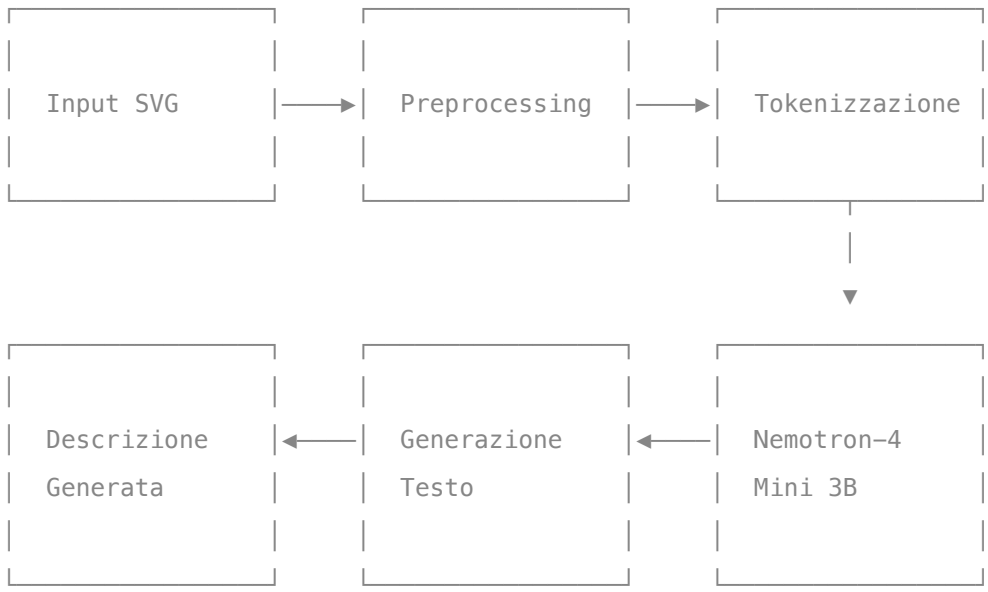
Architetture per il Captioning SVG

Questo documento illustra passo per passo le strutture da implementare per il sistema di captioning SVG, sia nella versione iniziale decoder-only che nella versione completa encoder-decoder che verrà sviluppata quando Leonardo fornirà l'encoder.

1. Architettura Decoder-Only (Fase Iniziale)



Componenti Principali:

1. **Input SVG**
   - File SVG da descrivere
   - Può essere in bianco e nero (dataset iniziale) o a colori (dataset avanzato)
2. **Preprocessing**
   - **Rasterizzazione**: Conversione dell'SVG in formato bitmap
   - **Normalizzazione**: Standardizzazione delle dimensioni e dei valori dei pixel
   - **Estrazione metadati**: Raccolta di informazioni strutturali dall'SVG
3. **Tokenizzazione**
   - **Token speciali**: Integrazione dei token speciali forniti da Leonardo
   - **Tokenizzazione gerarchica**: Rappresentazione della struttura dell'SVG

- **Token di stile**: Indicatori per guidare lo stile descrittivo

4. **Nemotron-4 Mini 3B**
   - **Modello decoder-only**: Architettura transformer con 3 miliardi di parametri
   - **Input condizionato**: Il modello riceve i token dell'SVG come contesto
   - **Generazione autoregressiva**: Genera un token alla volta basandosi sui precedenti
5. **Generazione Testo**
   - **Beam search**: Esplorazione di multiple sequenze candidate
   - **Temperatura di campionamento**: Controllo della creatività/diversità
   - **Lunghezza controllata**: Limitazione della lunghezza della descrizione
6. **Descrizione Generata**
   - Output finale del sistema: descrizione testuale dell'SVG

Flusso di Implementazione:

1. **Setup del modello base**

```python
# Esempio di codice per inizializzare Nemotron-4 Mini 3B
from transformers import AutoModelForCausalLM, AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("nvidia/nemotron-4-mini-3b")
model = AutoModelForCausalLM.from_pretrained("nvidia/nemotron-4-mini-3b")
```

2. **Preprocessing SVG**

```python
# Esempio di codice per preprocessare SVG
def preprocess_svg(svg_path):
    # Rasterizzazione
    raster_image = convert_svg_to_raster(svg_path)

    # Normalizzazione
    normalized_image = normalize_image(raster_image)

    # Estrazione metadati
    metadata = extract_svg_metadata(svg_path)
```

```
    return normalized_image, metadata
```

3. Tokenizzazione con token speciali

python

```python
# Esempio di codice per tokenizzazione
def tokenize_svg(image, metadata, tokenizer):
    # Aggiunta token speciali (da implementare con il codice di Leonardo)
    special_tokens = add_special_tokens(metadata)

    # Tokenizzazione dell'immagine
    tokens = tokenizer.encode_image(image)

    # Combinazione
    input_tokens = special_tokens + tokens

    return input_tokens
```

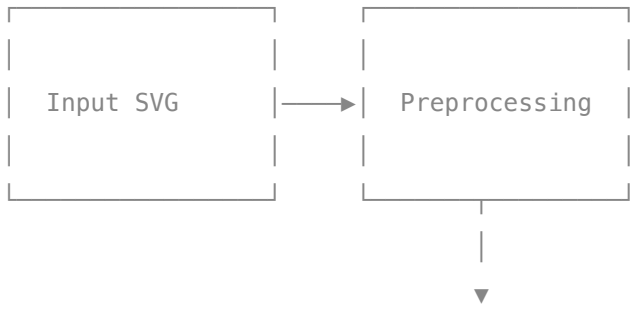
4. Generazione della descrizione

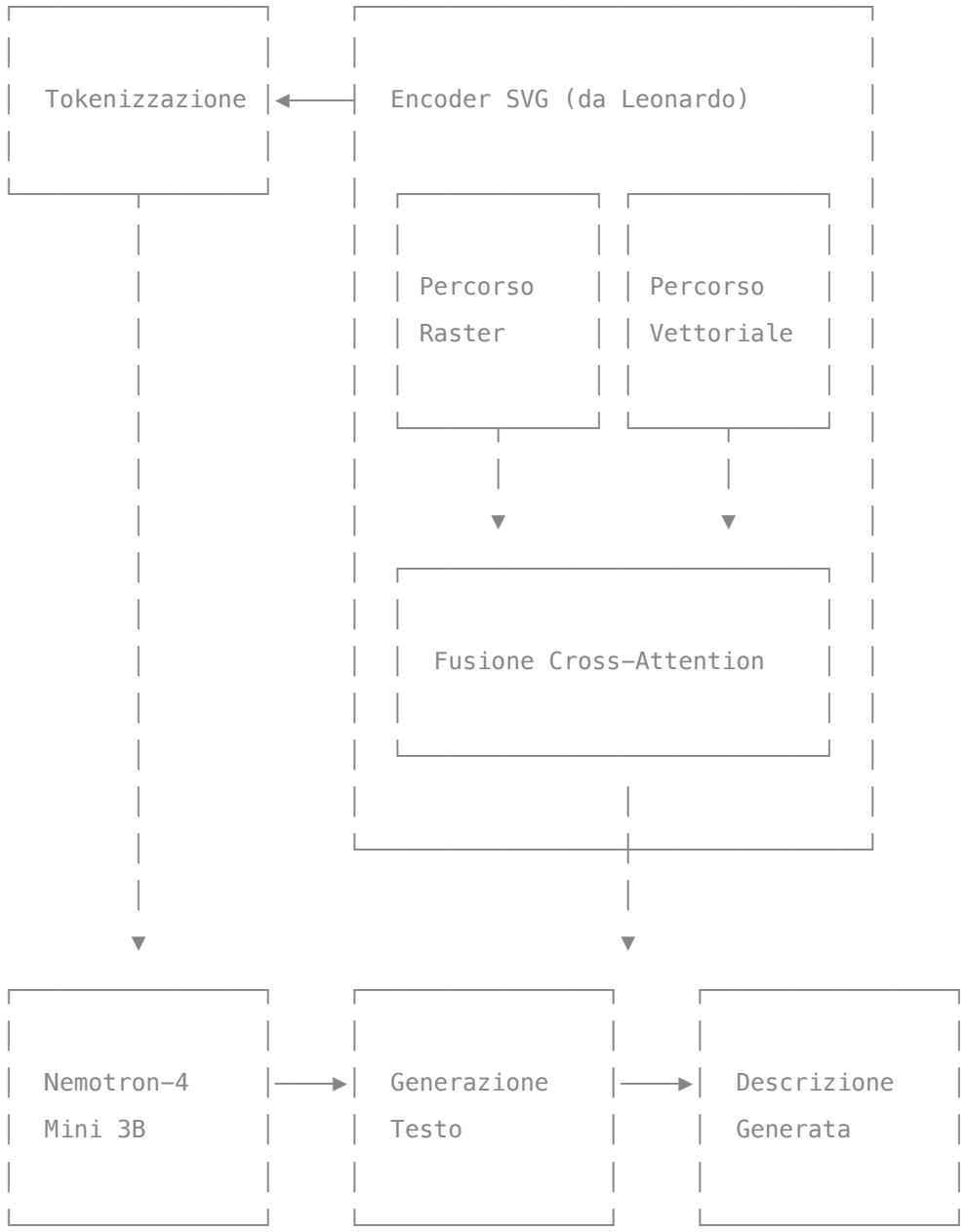
python

```python
# Esempio di codice per generazione
def generate_caption(input_tokens, model, tokenizer):
    outputs = model.generate(
        input_tokens,
        max_length=100,
        num_beams=5,
        temperature=0.7,
        no_repeat_ngram_size=2
    )

    caption = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return caption
```

2. Architettura Encoder-Decoder (Fase Avanzata)





Componenti Principali:

1. **Input SVG e Preprocessing**
   - Simili alla versione decoder-only, ma con preprocessing ottimizzato per l'encoder dual-path
2. **Encoder SVG (da Leonardo)**
   - **Percorso Raster**:
     - Elabora la versione rasterizzata dell'SVG
     - Cattura caratteristiche visive come colori, texture, forme
   - **Percorso Vettoriale**:

- Elabora la struttura vettoriale dell'SVG (comandi, path, elementi)
- Cattura relazioni gerarchiche e strutturali

- **Fusione Cross-Attention**:
  - Integra le rappresentazioni dei due percorsi
  - Permette lo scambio di informazioni tra le rappresentazioni raster e vettoriali

3. **Tokenizzazione**
   - Arricchita con le rappresentazioni dell'encoder
   - Incorpora token speciali e informazioni strutturali
4. **Nemotron-4 Mini 3B**
   - Funziona come decoder condizionato dalle rappresentazioni dell'encoder
   - Riceve sia i token dell'SVG che le rappresentazioni dell'encoder
5. **Generazione Testo e Descrizione Generata**
   - Simili alla versione decoder-only, ma con maggiore precisione grazie all'encoder specializzato

Flusso di Implementazione:

1. **Integrazione dell'encoder fornito da Leonardo**

```python
# Esempio di codice per integrare l'encoder
from svg_encoder import SVGEncoder  # Modulo fornito da Leonardo

# Inizializzazione dell'encoder
encoder = SVGEncoder()
```

2. **Preprocessing ottimizzato per dual-path**

```python
# Esempio di codice per preprocessing dual-path
def preprocess_svg_dual_path(svg_path):
    # Rasterizzazione per percorso raster
    raster_image = convert_svg_to_raster(svg_path)

    # Estrazione struttura vettoriale per percorso vettoriale
```

```
    vector_structure = extract_svg_structure(svg_path)

    return raster_image, vector_structure
```

3. Elaborazione con encoder dual-path

python

```
# Esempio di codice per elaborazione con encoder
def encode_svg(raster_image, vector_structure, encoder):
    # Elaborazione raster
    raster_features = encoder.encode_raster(raster_image)

    # Elaborazione vettoriale
    vector_features = encoder.encode_vector(vector_structure)

    # Fusione cross-attention
    fused_features = encoder.fuse_features(raster_features, vector_features)

    return fused_features
```

4. Generazione condizionata dalla rappresentazione dell'encoder

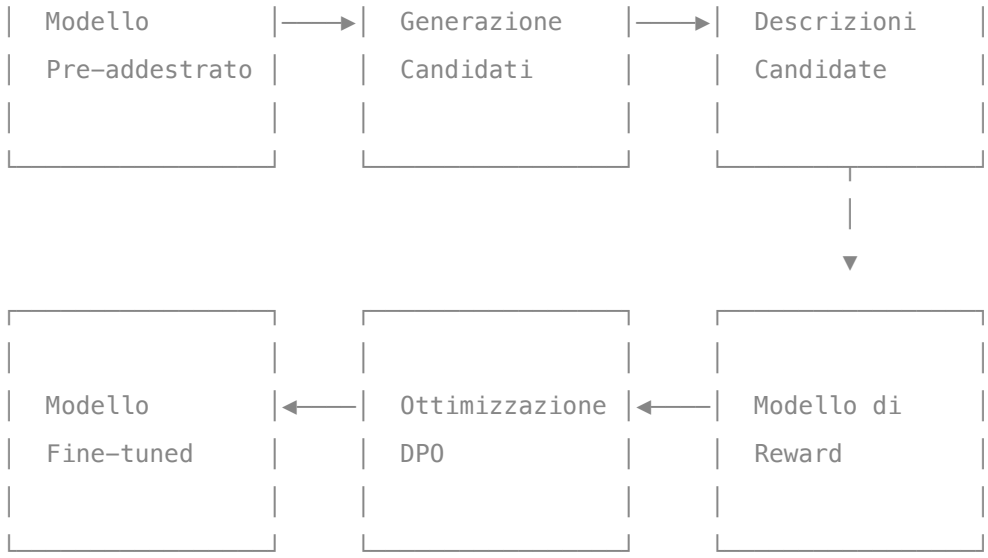
python

```
# Esempio di codice per generazione condizionata
def generate_caption_with_encoder(fused_features, model, tokenizer):
    # Preparazione input per il decoder
    decoder_input = prepare_decoder_input(fused_features)

    # Generazione con condizionamento dall'encoder
    outputs = model.generate(
        decoder_input,
        encoder_hidden_states=fused_features,
        max_length=100,
        num_beams=5,
        temperature=0.7
    )

    caption = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return caption
```

3. Fine-tuning con DPO (Fase Finale)



Componenti Principali:

1. **Modello Pre-addestrato**
   - Versione decoder-only o encoder-decoder addestrata sui dataset SVG
2. **Generazione Candidati**
   - Generazione di multiple descrizioni candidate per ogni SVG
3. **Descrizioni Candidate**
   - Set di descrizioni alternative per lo stesso SVG
4. **Modello di Reward**
   - Valuta la qualità delle descrizioni candidate
   - Basato sull'approccio Self-Cap (Moratelli et al.)
5. **Ottimizzazione DPO**
   - Direct Preference Optimization
   - Ottimizza il modello in base alle preferenze del modello di reward
6. **Modello Fine-tuned**
   - Versione finale del modello ottimizzata con DPO

Flusso di Implementazione:

1. **Creazione del modello di reward**

python

```python
# Esempio di codice per modello di reward
class SelfCapRewardModel(nn.Module):
    def __init__(self):
        super().__init__()
        # Inizializzazione del modello di reward
        self.clip_model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32")
        self.reward_head = nn.Linear(512, 1)

    def forward(self, images, captions):
        # Calcolo del reward
        image_features = self.clip_model.get_image_features(images)
        text_features = self.clip_model.get_text_features(captions)

        # Calcolo similarità
        similarity = F.cosine_similarity(image_features, text_features)

        # Reward finale
        reward = self.reward_head(similarity.unsqueeze(-1))
        return reward
```

2. Generazione di descrizioni candidate

python

```python
# Esempio di codice per generazione candidati
def generate_candidates(svg, model, tokenizer, num_candidates=16):
    candidates = []

    for _ in range(num_candidates):
        # Generazione con diversi parametri
        output = model.generate(
            tokenizer.encode(svg, return_tensors="pt"),
            max_length=100,
            temperature=0.9,
            top_p=0.9,
            do_sample=True
        )

        candidate = tokenizer.decode(output[0], skip_special_tokens=True)
        candidates.append(candidate)

    return candidates
```

3. Implementazione DPO

python

```
# Esempio di codice per DPO
def train_with_dpo(model, reward_model, dataset):
    # Configurazione DPO
    dpo_trainer = DPOTrainer(
        model=model,
        ref_model=copy.deepcopy(model),
        reward_model=reward_model,
        beta=0.1,  # Parametro di regolarizzazione
        dataset=dataset
    )

    # Addestramento
    dpo_trainer.train()

    return dpo_trainer.model
```

4. Confronto tra Architetture

Aspetto	Decoder-Only	Encoder-Decoder
Complessità	Minore	Maggiore
Comprensione SVG	Basica	Avanzata (dual-path)
Qualità descrizioni	Buona	Eccellente
Tempo di addestramento	Minore	Maggiore
Risorse richieste	Moderate	Elevate
Flessibilità	Alta	Molto alta

5. Passi di Implementazione Consigliati

1. **Fase Decoder-Only**:
    - Implementare il preprocessing base per SVG
    - Configurare Nemotron-4 Mini 3B
    - Integrare i token speciali (quando forniti da Leonardo)
    - Addestrare sul dataset in bianco e nero
    - Valutare le prestazioni iniziali
2. **Fase Encoder-Decoder**:

- Integrare l'encoder fornito da Leonardo
- Implementare il meccanismo di fusione cross-attention
- Adattare il decoder per ricevere le rappresentazioni dell'encoder
- Fine-tuning sul dataset completo
- Valutare il miglioramento rispetto alla versione decoder-only

3. **Fase DPO**:
   - Implementare il modello di reward Self-Cap
   - Generare descrizioni candidate
   - Configurare l'ottimizzazione DPO
   - Fine-tuning con DPO
   - Valutazione finale e confronto con baseline

Questa struttura modulare permette di sviluppare il sistema in fasi incrementali, aggiungendo complessità e funzionalità man mano che si procede con l'implementazione.