1 Encodeur de la langue des signes

Le `TransformerEncoder` constitue le **cœur de l'analyse séquentielle** du système de traduction de langue des signes. Son rôle principal est de transformer une séquence brute de caractéristiques gestuelles (issues de MediaPipe) en représentations contextuelles riches qui capturent les dépendances temporelles complexes de la langue des signes.

1. Projection des Entrées (`input_projection`) - **Fonction** : Adapte la dimension des features d'entrée (126D pour MediaPipe) à la dimension du modèle (`d_model`) - **Opération mathématique** : Transformation linéaire $Wx + b$ où $W \in \mathbb{R}^{d_{model} \times input\_dim}$ - **Utilité** : Uniformise l'espace de représentation pour les étapes suivantes

2. Encodage Positionnel (`pos_encoding`) - **Problème résolu** : Les Transformers n'ont pas de notion innée de l'ordre séquentiel - **Solution** : l'Encodage Positionnel ajoute des informations temporelles via des fonctions sinus/cosinus. - **Formule** :

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

- **Importance** : Cela permet au modèle de distinguer la position temporelle de chaque frame dans la séquence vidéo.

3. Régularisation (`dropout`) - **But** : Prévenir le surapprentissage pendant l'entraînement. - **Mécanisme** : Désactive aléatoirement une fraction des neurones. - **Impact** : Améliore l'apprentissage du modèle.

4. Pile de Couches Encodeur (`layers`) - **Composition** : Empilement de `num_layers` couches identiques. - **Pour chaque couche** : nous avons - Mécanisme d'auto-attention multi-têtes. - Réseau feed-forward position-wise. - Connexions résiduelles et normalisation de couche. - **Évolution** : Chaque couche affine les représentations en capturant des dépendances de plus en plus complexes.

5. Normalisation Finale (`norm`) - **Fonction** : Stabilise les activations et accélère la convergence - **Bénéfice** : Réduit la sensibilité aux initialisations et aux hyperparamètres

```
import math
import random
import time as tm
import typing as _t
import logging
from dataclasses import dataclass

import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchinfo import summary
```

```
# Set up logging:
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("train_transformer.log"),
        logging.StreamHandler()
        ]
    )
LOGGER = logging.getLogger(__name__)
```

```
class TransformerEncoder(nn.Module):
    """
    Complete Transformer Encoder.

    This module implements the full encoder stack of the Transformer␣
 ↪model.
    It takes input sequences of landmark features and produces encoded
    representations that capture contextual information.

    :param input_dim: Dimension of input features (C from (B, T, C)).
    :param d_model: Dimension of the model (embedding size).
    :param num_heads: Number of attention heads.
    :param num_layers: Number of encoder layers.
    :param d_ff: Dimension of feed-forward network hidden layer.
    :param dropout: Dropout rate, defaults to 0.1.
    :param max_seq_len: Maximum sequence length for positional encoding,
      defaults to 5000.
    """
    def __init__(
        self,
        input_dim: int,
        d_model: int,
        num_heads: int,
        num_layers: int,
        d_ff: int,
        dropout: float=0.1,
        max_seq_len: int=250
    ) -> None:
        """Initialize TransformerEncoder module."""
        super().__init__()

        # Project input features to model dimension
```

```
        self.input_projection = nn.Linear(input_dim, d_model)

        # Positional encoding
        self.pos_encoding = PositionalEncoding(d_model, max_seq_len)

        # Dropout for input embeddings
        self.dropout = nn.Dropout(dropout)

        # Stack of encoder layers
        self.layers = nn.ModuleList([
            EncoderLayer(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)
            ])

        # Final layer norm
        self.norm = nn.LayerNorm(d_model)

    def forward(
        self,
        x: torch.Tensor,
        mask: torch.Tensor=None
    ) -> torch.Tensor:
        """
        Forward pass of Transformer encoder.

        :param x: Input tensor of shape (B, T, C) where B is batch size,
          T is sequence length (number of frames), C is feature dimension.
        :param mask: Optional mask tensor for attention, defaults to None.
        :returns: Encoded representations of shape (B, T, d_model).

        :raises ValueError: If input tensor does not have 3 dimensions.
        """
        # Validate input shape
        if len(x.shape) != 3:
            raise ValueError(
                f"Input must have 3 dimensions (B, T, C), got {len(x.
↪shape)}"
            )

        # Project input to model dimension
        x = self.input_projection(x)  # (B, T, d_model)
        # Add positional encoding and apply dropout
        x = self.pos_encoding(x)
        x = self.dropout(x)
```

```
        # Pass through each encoder layer
        for layer in self.layers:
            x = layer(x, mask)
        # Apply final layer norm
        x = self.norm(x)
        return x
```

1.1 Flux de Données (Forward Pass)

1. **Étape 1 : Validation et Préparation**

- Vérification que l'entrée a la forme `(batch_size, sequence_length, feature_dim)`
- Contrôle que la longueur de séquence ne dépasse pas `max_seq_len`

2. **Étape 2 : Projection Linéaire** Prenoons par exemple, la séquence "BONJOUR" en langue des signes

- **Frame 0** : Mains en position de départ
- **Frame 1** : Début du mouvement vers le front
- **Frame 2** : Mains au front (point culminant du geste)
- **Frame 3** : Retour vers la position neutre

Chaque frame a des coordonnées de mains :

```
Frame 0: [x1,y1,z1, x2,y2,z2, ...]  # 126 valeurs
Frame 1: [x1,y1,z1, x2,y2,z2, ...]  # 126 valeurs
Frame 2: [x1,y1,z1, x2,y2,z2, ...]  # 126 valeurs
Frame 3: [x1,y1,z1, x2,y2,z2, ...]  # 126 valeurs
```

La dimension de la matrice d'entrée est donc : $\dim(x) = B \times T \times 126$. La projection linéaire réalise l'opération de projection suivante:

$$Wx + b$$

où $W \in \mathbb{R}^{d_{model} \times input\_dim}$

Ce qui change la dimension de la matrice.

`(B, T, 126) ---> (B, T, d_model)`

Les coordonnées MediaPipe brutes sont projetées dans un espace de dimension supérieure plus expressif.

3. **Étape 3 : Enrichissement Temporel**

`Features projetées + Encodage positionnel → Représentations temporelles`

Chaque frame reçoit une "signature temporelle" unique.

4. **Étape 4 : Transformation Contextuelle**

`Représentations temporelles → [Couche 1] → [Couche 2] → ... → [Couche N]`

Chaque couche d'encodeur : 1. **Auto-attention** : Chaque position "regarde" toutes les autres positions 2. **Combinaison** : Agrège l'information contextuelle 3. **Transformation non-linéaire** : Applique des transformations complexes via le FFN

5. **Étape 5 : Normalisation et Sortie**

`Sortie finale normalisée de forme (B, T, d_model)`

1.2 Rôle Fondamental de l'Encodage Positionnel

Les Transformers, par leur nature, traitent toutes les positions de la séquence **simultanément** et **sans ordre prédéfini**. Contrairement aux RNN/LSTM qui traitent les séquences séquentiellement et donc "connaissent" naturellement l'ordre, les Transformers sont agnostiques à la position des éléments dans la séquence.

**Imaginer que vous regarder une vidéo de langue des signes où quelqu'un signe "JE T'AIME". Sans encodage positionnel, le Transformer verrait ça comme :

`[Frame1, Frame2, Frame3] = [JE, T', AIME]`

Mais il pourrait aussi penser que c'est :

`[Frame3, Frame1, Frame2] = [AIME, JE, T']`

Résultat : le modèle ne pourrait faire aucune distinction entre "AIME JE T"' et "JE T'AIME" !
C'est pourquoi on doit dire au modèle : "Frame1 vient en premier, Frame2 en deuxième, Frame3 en troisième".

```python
class PositionalEncoding(nn.Module):
    """
    Positional encoding for Transformer models.

    This module adds positional information to the input embeddings
    using sine and cosine functions of different frequencies.

    :param d_model: The dimension of the embeddings.
    :param max_len: Maximum sequence length, defaults to 5000.
    """
    def __init__(self, d_model: int, max_len: int=5000) -> None:
        """Initialize PositionalEncoding module."""
        super().__init__()

        # Create positional encoding matrix
        pe = torch.zeros(max_len, d_model)
        # Position indices [0, 1, 2, ..., max_len-1]
```

```
        position = torch.arange(0, max_len, dtype=torch.float).
↪unsqueeze(1)
        # Division term for frequency calculation
        div_term = torch.exp(torch.arange(0, d_model, 2).float()
            * (-math.log(10000.0) / d_model))
        # Apply sine to even indices
        pe[:, 0::2] = torch.sin(position * div_term)
        # Apply cosine to odd indices
        pe[:, 1::2] = torch.cos(position * div_term)

        # Add batch dimension: (1, max_len, d_model)
        pe = pe.unsqueeze(0)
        # Register as buffer (not a parameter)
        self.register_buffer('pe', pe)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Add positional encoding to input tensor.

        :param x: Input tensor of shape (B, T, C).
        :returns: Output tensor with positional encoding added.
        """
        # Get the feature dimension of input
        input_feature_dim = x.size(-1)
        pe_feature_dim = self.pe.size(-1)

        # Check if dimensions match:
        if input_feature_dim != pe_feature_dim:
            raise ValueError(
                f"Feature dimension mismatch: input has␣
↪{input_feature_dim} "
                f"features, but positional encoding has {pe_feature_dim} "
                f"features. Make sure your input projection matches "
                "the `d_model` used in PositionalEncoding."
            )

        # Add positional encoding to input
        # pe is sliced to match sequence length T.
        return x + self.pe[:, :x.size(1), :]
```

Autre exemple avec une Phrase Complète

Séquence : "JE VAIS BIEN" (3 signes sur 9 frames)

```
Frames 0-2: Signe "JE"     → Encodages [PE0, PE1, PE2]
Frames 3-5: Signe "VAIS"   → Encodages [PE3, PE4, PE5]
```

`Frames 6-8: Signe "BIEN"  → Encodages [PE6, PE7, PE8]`

Le Transformer apprend : - Les patterns `[PE0, PE1, PE2]` correspondent souvent au mot “JE” - `[PE3, PE4, PE5]` correspondent à “VAIS”
- `[PE6, PE7, PE8]` correspondent à “BIEN” - La séquence complète `[PE0...PE8]` signifie “JE VAIS BIEN”

Voici une autre analogie dans la musique

Imaginer que chaque frame de votre vidéo est un musicien dans un orchestre :

- Sans encodage positionnel, tous les musiciens joueraient en même temps, ce qui conduirait à une vraie **cacophonie** !

- Avec encodage positionnel, chacun joue au bon moment ce qui donnerait une **belle symphonie** !

1.3 Mécanisme d’Auto-Attention

Imaginer que vous regarder un interprète en langue des signes traduire “JE VAIS À LA MAISON”. Votre cerveau ne regarde pas toutes les parties du corps en même temps avec la même intensité. Au lieu de cela :

- Quand il signe “JE”, vous **focalisez** sur la poitrine.
- Pour “MAISON”, vos yeux **se déplacent** vers la forme des mains qui dessinent un toit.
- Pendant “ALLER”, vous **suivez** le mouvement des bras.

C’est exactement ce que fait le mécanisme d’**attention multi-têtes** ! Il permet à ton modèle de “regarder” différentes parties de la séquence vidéo avec différentes “intensités” selon ce qu’il veut traduire.

```python
class MultiHeadAttention(nn.Module):
    """
    Multi-head auto and cross-attention mechanism.

    This module computes multi-head attention where the query comes
    from one sequence and the keys/values come from another sequence
    (encoder outputs).

    :param d_model: The dimension of the input and output features.
    :param num_heads: The number of attention heads.
    :param dropout: Dropout rate for attention weights, defaults to 0.1.
    """
    def __init__(
        self,
        d_model: int,
        num_heads: int,
```

```
        dropout: float=0.1
    ) -> None:
        """Initialize MultiHeadCrossAttention module."""
        super().__init__()

        # Validate that d_model is divisible by num_heads
        if d_model % num_heads != 0:
            raise ValueError("d_model must be divisible by num_heads.")

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads  # Dimension of each head

        # Linear projections for Query, Key, Value
        self.w_q = nn.Linear(d_model, d_model, bias=False)  # Query from␣
↪decoder
        self.w_k = nn.Linear(d_model, d_model, bias=False)  # Key from␣
↪encoder
        self.w_v = nn.Linear(d_model, d_model, bias=False)  # Value from␣
↪encoder
        self.w_o = nn.Linear(d_model, d_model)  # Output projection

        self.dropout = nn.Dropout(dropout)
        self.scale = math.sqrt(self.d_k)  # Scaling factor for stability

    def forward(
        self,
        query: torch.Tensor,
        key: torch.Tensor,
        value: torch.Tensor,
        mask: torch.Tensor = None
    ) -> torch.Tensor:
        """
        Forward pass of multi-head cross-attention.

        :param query: Query tensor from decoder of shape (B, T_dec, C).
        :param key: Key tensor from encoder of shape (B, T_enc, C).
        :param value: Value tensor from encoder of shape (B, T_enc, C).
        :param mask: Optional mask tensor for attention, defaults to None.
        :returns: Output tensor of shape (B, T_dec, C).
        """
        batch_size, t_dec, d_model = query.shape
        _, t_enc, _ = key.shape
```

```
        # Linear projections and reshape for multi-head attention
        # Query from decoder: (B, num_heads, T_dec, d_k)
        Q = self.w_q(query) \
            .view(batch_size, t_dec, self.num_heads, self.d_k) \
            .transpose(1, 2)
        # Key from encoder: (B, num_heads, T_enc, d_k)
        K = self.w_k(key) \
            .view(batch_size, t_enc, self.num_heads, self.d_k) \
            .transpose(1, 2)
        # Value from encoder: (B, num_heads, T_enc, d_k)
        V = self.w_v(value) \
            .view(batch_size, t_enc, self.num_heads, self.d_k) \
            .transpose(1, 2)

        # Compute attention scores: (B, num_heads, T_dec, T_enc)
        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
        # Apply mask if provided (for padding or other masks)
        if mask is not None:
            # Mask should have shape (B, T_dec, T_enc) or broadcastable␣
↪shape
            scores = scores.masked_fill(mask == 0, -1e9)
        # Apply softmax to get attention weights
        attn_weights = F.softmax(scores, dim=-1)
        attn_weights = self.dropout(attn_weights)
        # Apply attention to values: (B, num_heads, T_dec, d_k)
        attn_output = torch.matmul(attn_weights, V)
        # Reshape back to (B, T_dec, C)
        attn_output = attn_output \
            .transpose(1, 2) \
            .contiguous() \
            .view(batch_size, t_dec, d_model)
        # Final linear projection
        output = self.w_o(attn_output)
        return output
```

imaginer que vous avez **8 experts** (num_heads=8) qui analysent la même séquence vidéo, mais chacun se spécialise sur un aspect différent :

- **Expert 1** : Se concentre sur la **forme des mains**
- **Expert 2** : Analyse les **expressions faciales**

- **Expert 3** : Étudie la **vitesse des mouvements**
- **Expert 4** : Observe la **direction des gestes**
- ... et ainsi de suite jusqu'à l'expert 8

Chaque expert regarde la même vidéo, mais **se pose des questions différentes** et **tire**

des conclusions différentes.

Initialisation : Préparation des Experts

```
self.d_k = d_model // num_heads  # Dimension de chaque tête
```

Exemple : Si `d_model=512` et `num_heads=8`, alors `d_k=64`.
Chaque expert reçoit 64 dimensions sur lesquelles se spécialiser, comme si on donnait à chaque traducteur un domaine spécifique à analyser.

Concerant, les Quatre Projections Linéaires :

1. `w_q` - Les Questions de l'Expert

```
self.w_q = nn.Linear(d_model, d_model, bias=False)
```

Rôle : Transforme ce que le décodeur "veut savoir" en questions spécifiques.

Exemple : Si le décodeur génère le mot "MAISON", `w_q` pose la question : "Quelles parties de la vidéo montrent des gestes liés à une maison ?"

2. `w_k` - Les Clés de Référence

```
self.w_k = nn.Linear(d_model, d_model, bias=False)
```

Rôle : Transforme chaque frame de l'encodeur en "étiquette" qui décrit son contenu.

Exemple : Une frame où les mains forment un toit aura une clé qui dit "je contient un geste de construction".

3. `w_v` - Les Valeurs Informatives

```
self.w_v = nn.Linear(d_model, d_model, bias=False)
```

Rôle : Transforme chaque frame en information détaillée utilisable.

Exemple : La même frame avec les mains en toit contient la valeur "position mains: toit, orientation: vers le haut, mouvement: statique".

4. `w_o` - La Synthèse des Experts

```
self.w_o = nn.Linear(d_model, d_model)
```

Rôle : Combine les conclusions de tous les experts en une réponse cohérente.

1.3.1 Le Processus de l'Attention en Action**

1. **Étape 1 : Préparation des Experts**

```
Q = self.w_q(query).view(batch_size, t_dec, self.num_heads, self.d_k).transpose(1, 2)
```

Transformation concrète :

```
Avant: (4, 10, 512)  # 4 vidéos, 10 mots à traduire, 512 dimensions
Après: (4, 8, 10, 64)  # 4 vidéos, 8 experts, 10 mots, 64 dimensions par expert
```

Chaque expert reçoit sa propre version des questions !

2. **Étape 2 : Calcul des Affinités**, Il s'agit de l'auto-attention. L'auto-attention permet à chaque frame de la séquence d'interagir avec toutes les autres frames, capturant ainsi :

- Les **dépendances à longue portée** entre gestes éloignés dans le temps
- Les **relations contextuelles** entre différents éléments du signe
- La **cohérence temporelle** de l'ensemble du mouvement

Équation Clé

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

Où : - Q (Queries) : Ce que chaque position "cherche" - K (Keys) : Ce que chaque position "offre"
- V (Values) : L'information réelle à agréger

```
scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
```

Ce qui se passe vraiment : Chaque expert compare ses questions avec toutes les étiquettes des frames vidéo.

Exemple : - Expert "forme des mains" demande : "Quelles frames ont des mains en forme de toit ?" - Les frames où les mains forment un toit obtiennent un **score élevé** - Les frames avec d'autres formes obtiennent un **score bas**

3. **Étape 3 : Application du Masque et Softmax**

```
if mask is not None:
    scores = scores.masked_fill(mask == 0, -1e9)
attn_weights = F.softmax(scores, dim=-1)
```

Le masque : Comme dire à certains experts "ne regarde pas ces frames-là, elles ne sont pas pertinentes".

Softmax : Convertit les scores en **pourcentages d'attention**.
Si une frame a un score de 0.8, cela signifie "80% de ton attention devrait être sur cette frame".

4. **Étape 4 : Combinaison Pondérée**

```
attn_output = torch.matmul(attn_weights, V)
```

Action : Chaque expert prend les informations détaillées (V) et les combine selon ses pourcentages d'attention.

Exemple : - Expert "forme des mains" : 80% d'info frame15 + 20% d'info frame16 - Expert "expression faciale" : 60% frame14 + 40% frame15

5. **Étape 5 : Réunion des Experts**

```
attn_output = attn_output.transpose(1, 2).contiguous().view(batch_size, t_dec, d_model)
output = self.w_o(attn_output)
```

On rassemble les conclusions des 8 experts et on les combine intelligemment.

```
Expert1: "C'est le geste MAISON à 85% de confiance"
Expert2: "L'expression confirme un lieu à 70%"
Expert3: "Le mouvement suggère une destination à 90%"
→ Synthèse: "Traduire par 'MAISON' avec haute confiance"
```

1.4 Le Cerveau qui Réfléchit

Imaginer que votre modèle vient de comprendre, grâce à l'attention, qu'une séquence de gestes signifie "JE T'AIME". Mais cette compréhension est encore "superficielle". C'est comme reconnaître des lettres sans comprendre le sens profond des mots.

Le **réseau feed-forward position-wise** agit comme le **cerveau qui réfléchit en profondeur**. Après que l'attention a "regardé" les bonnes parties de la vidéo, ce module prend ces observations et les **transforme en compréhension sémantique riche**.

Analogie Concrete : L'Usine de Transformation des Idées

Imagine que chaque position dans ta séquence (chaque "instant de compréhension") passe par une petite usine de traitement :

```
[Idée brute] --> [Machine à enrichir] --> [Idée enrichie] --> [Machine à synthétiser] --
```

Chaque usine est identique et travaille indépendamment sur son propre "lot" d'information, mais toutes suivent le même processus de raffinement.

```python
class PositionwiseFeedForward(nn.Module):
    """
    Position-wise feed-forward network.

    This module applies the same feed-forward network to each position
    separately and identically. It consists of two linear transformations
    with a non-linearity in between.

    :param d_model: The dimension of input and output features.
    :param d_ff: The dimension of the hidden layer in feed-forward␣
↪network.
    :param dropout: Dropout rate, defaults to 0.1.
    """
    def __init__(self, d_model: int, d_ff: int, dropout: float = 0.1) ->␣
↪None:
        """Initialize PositionwiseFeedForward module."""
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
```

```
        self.linear2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = nn.GELU()
        #: GELU activation often works better than ReLU;

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        """
        Forward pass of position-wise feed-forward network.

        :param x: Input tensor of shape (B, T, C).
        :returns: Output tensor of same shape as input (B, T, C).
        """
        # First linear transformation + activation + dropout
        hidden = self.activation(self.linear1(x))
        hidden = self.dropout(hidden)

        # Second linear transformation
        output = self.linear2(hidden)
        return output
```

1. **Étape 1 : Expansion et Enrichissement**

`self.linear1 = nn.Linear(d_model, d_ff)`

Ce qui se passe : On prend une compréhension de dimension `d_model` (ex: 512) et on l'**étend** vers une dimension beaucoup plus grande `d_ff` (ex: 2048).

Analogie : - **Entrée** : "C'est le geste MAISON" (512 dimensions de compréhension) - **Sortie linear1** : "MAISON = bâtiment + toit + murs + fenêtres + porte + habitat + famille + sécurité + ..." (2048 dimensions détaillées)

2. **Étape 2 : Activation GELU - La Pensée Non-Linéaire**

`self.activation = nn.GELU()`

GELU vs ReLU : - **ReLU** : "Si c'est négatif, ignore; si c'est positif, garde" (binaire) - **GELU** : "Si c'est négatif, atténue progressivement; si c'est positif, garde mais de façon nuancée" (graduel)

Formule GELU : $GELU(x) = x \cdot \Phi(x)$ où $\Phi(x)$ est la fonction de répartition gaussienne

Exemple concret : - Une feature dit "mouvement des mains = 0.8" (forte confiance) - GELU garde cette information presque intacte - Une autre feature dit "expression faciale = -0.1" (légère contradiction)
- GELU l'atténue légèrement plutôt que de la supprimer complètement

3. **Étape 3 : Régularisation**

`self.dropout = nn.Dropout(dropout)`

Rôle : Pendant l'entraînement, désactive aléatoirement 10% des neurones pour éviter la "paresse intellectuelle".

Effet : Force le réseau à développer des **représentations redondantes et robustes**.

4. **Étape 4 : Compression et Synthèse**

```
self.linear2 = nn.Linear(d_ff, d_model)
```

Ce qui se passe : On repasse de la dimension étendue (2048) à la dimension originale (512), mais maintenant l'information est **enrichie et transformée**.

Analogie : - **Entrée linear2** : "MAISON = bâtiment + toit + murs + fenêtres + porte + habitat + famille + sécurité + ..." - **Sortie linear2** : "Concept MAISON enrichi" (même dimension 512, mais signification plus profonde)

1.4.1 Processus Complet sur un Exemple Réel

Scénario : Le modèle analyse le signe "AIMER" en langue des signes

Entrée au module (après l'attention) :

```
[Position 22]: "Mains sur cœur + regard direct + sourire léger" (512D)
```

Étape par étape :

1. **Linear1 - Expansion** :

   ```
   Input: 512 dimensions → Output: 2048 dimensions
   "Mains sur cœur" → "cœur + affection + amour + émotion + sentiment + tendresse + at
   ```

2. **GELU - Raffinement non-linéaire** :

   ```
   Renforce: "affection", "amour", "tendresse"
   Atténue: "mouvement brusque" (peu pertinent)
   Conserve: "regard direct" (modérément important)
   ```

3. **Dropout - Régularisation** :

   ```
   Désactive aléatoirement: "attachement", "sentiment" (forçant d'autres features à co
   ```

4. **Linear2 - Synthèse** :

   ```
   Input: 2048 dimensions → Output: 512 dimensions
   "affection + amour + tendresse + émotion + ..." → "Concept AIMER enrichi"
   ```

Résultat : La même position a maintenant une **compréhension bien plus riche** du geste "AIMER".

Pourquoi "Position-wise" ?

Clé importante : Le même réseau feed-forward est appliqué **indépendamment à chaque position** dans la séquence.

Exemple avec une séquence de 3 frames :

```
Frame 15: [geste JE]     → FFN indépendant → [JE enrichi]
Frame 16: [geste T']     → FFN indépendant → [T' enrichi]
Frame 17: [geste AIME]   → FFN indépendant → [AIME enrichi]
```

Chaque frame passe par sa **propre copie** du même réseau feed-forward. C'est comme si chaque instant avait son propre petit cerveau qui réfléchit, mais tous ces petits cerveaux ont été entraînés ensemble.

Le Rôle dans l'Architecture Globale

Dans une couche complète de Transformer : 1. **MultiHeadAttention** : "Regarde quelles parties sont importantes" 2. **PositionwiseFeedForward** : "Réfléchis profondément à ce que tu as vu"

Analogie : - **Attention** : Comme discuter avec des collègues pour comprendre un problème - **FeedForward** : Comme rentrer chez soi et réfléchir seul en profondeur

Pourquoi $d_{ff} = 4 \times d_{model}$?

La règle empirique `d_ff = 4 × d_model` (2048 pour d_model=512) vient de l'observation que :

- **Trop petit** : Le réseau ne peut pas capturer des transformations complexes
- **Trop grand** : Risque de surapprentissage et calculs inutiles
- **4×** : Bon compromis expressivité/efficacité

Exemple avec des Dimensions Réelles

Configuration typique :

```
d_model = 512    # Dimension des embeddings
d_ff = 2048      # Dimension cachée (4 × d_model)
```

Flux de données :

```
Input: (4, 100, 512)    # 4 vidéos, 100 frames, 512 features
Linear1: (4, 100, 512) → (4, 100, 2048)   # Expansion
GELU: (4, 100, 2048) → (4, 100, 2048)     # Activation
Dropout: (4, 100, 2048) → (4, 100, 2048)  # Régularisation
Linear2: (4, 100, 2048) → (4, 100, 512)   # Compression
```

1.4.2 Impact sur la Traduction de Langue des Signes

Sans PositionwiseFeedForward : - Le modèle reconnaîtrait les gestes mais sans comprendre leur **signification contextuelle** - "MAISON" serait juste un geste de toit, pas le concept de domicile

Avec PositionwiseFeedForward : - "MAISON" devient associé à "habitat", "famille", "sécurité", "retour" - "AIMER" devient associé à "affection", "émotion", "relation", "attachement"

Ce module est essentiel pour transformer la **reconnaissance de patterns** en **compréhension sémantique**, permettant une traduction fidèle et naturelle.

Question : Pourquoi penses-tu qu'il est important que chaque position soit traitée indépendamment, plutôt que d'avoir un réseau qui regarde toute la séquence en même temps ?

1.5 La couche d'encodage

La couche d'encodage est l'unité fondamentale de compréhension pour la langue des signes. Elle est composée des autres modules. Voici son implémnentation :

```
class EncoderLayer(nn.Module):
    """
    Single encoder layer of the Transformer.

    This layer contains multi-head self-attention and position-wise
    feed-forward network with residual connections and layer␣
↪normalization.

    :param d_model: The dimension of input and output features.
    :param num_heads: The number of attention heads.
    :param d_ff: The dimension of hidden layer in feed-forward network.
    :param dropout: Dropout rate, defaults to 0.1.
    """
    def __init__(
        self,
        d_model: int,
        num_heads: int,
        d_ff: int,
        dropout: float = 0.1
    ) -> None:
        """Initialize EncoderLayer module."""
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads, dropout)
        self.feed_forward = PositionwiseFeedForward(d_model, d_ff,␣
↪dropout)

        # Layer normalization for residual connections
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x: torch.Tensor,
                mask: torch.Tensor = None) -> torch.Tensor:
        """
```

```
        Forward pass of encoder layer.

        :param x: Input tensor of shape (B, T, C).
        :type x: torch.Tensor
        :param mask: Optional mask tensor for attention, defaults to None.
        :type mask: torch.Tensor, optional
        :returns: Output tensor of same shape as input (B, T, C).
        :rtype: torch.Tensor
        """
        # Self-attention with residual connection and layer norm
        attn_output = self.self_attn(x, x, x, mask)
        x = x + self.dropout(attn_output)  # Residual connection
        x = self.norm1(x)  # Layer normalization after residual

        # Feed-forward with residual connection and layer norm
        ff_output = self.feed_forward(x)
        x = x + self.dropout(ff_output)  # Residual connection
        x = self.norm2(x)  # Layer normalization after residual

        return x
```

Quand tu calcules les gradients pendant la rétropropagation, chaque couche successive **atténue exponentiellement** le signal de correction.

1.5.1 Le Calcul des Gradients Sans Résiduel

Prenons 3 couches simples :

$$\text{Sortie} = f_3(f_2(f_1(x)))$$

Règle de la chaîne pour la rétropropagation :

$$\frac{\partial \text{Sortie}}{\partial x} = \frac{\partial f_3}{\partial f_2} \times \frac{\partial f_2}{\partial f_1} \times \frac{\partial f_1}{\partial x}$$

Le problème : Si chaque terme Jacobien a des valeurs propres < 1, le produit devient rapidement proche de 0.

Exemple numérique : - Si chaque couche atténue le gradient de 50% : $0.5 \times 0.5 \times 0.5 = 0.125$ - Après 6 couches : $0.5^6 = 0.015625 \rightarrow$ **98.4% du signal est perdu** !

1.5.2 La Solution Résiduelle : Les Autoroutes de Gradients

`x = x + self.dropout(attn_output)`

Transforme :

$$\text{Sortie} = f(x)$$

En :

$$\text{Sortie} = x + f(x)$$

1.5.3 Impact sur le Calcul des Gradients

Dérivée de la connexion résiduelle :

$$\frac{\partial \text{Sortie}}{\partial x} = \frac{\partial (x + f(x))}{\partial x} = 1 + \frac{\partial f(x)}{\partial x}$$

Conséquence cruciale : Même si $\frac{\partial f(x)}{\partial x} \approx 0$, tu as toujours ce $+\mathbf{1}$ qui garantit que :

$$\left\| \frac{\partial \text{Sortie}}{\partial x} \right\| \geq 1$$

1.5.4 Exemple Concret d'Optimisation

Scénario : Tu entraînes ton modèle sur le signe "MAISON"

Sans résiduel (après 6 couches) :

$$\nabla_{W_1} \mathcal{L} \approx 0.0001 \quad // \text{ Trop petit}$$

$$W_1 \leftarrow W_1 - \eta \times 0.0001 \quad // \text{ Mise à jour négligeable}$$

Avec résiduel (même après 6 couches) :

$$\nabla_{W_1} \mathcal{L} \approx 1.0 + 0.0001 \approx 1.0001 \quad // \text{ Signal fort}$$

$$W_1 \leftarrow W_1 - \eta \times 1.0001 \quad // \text{ Mise à jour significative}$$

1.5.5 Analyse Mathématique Approfondie

Pour une couche résiduelle :

$$y = x + f(x, W)$$

Gradient par rapport aux poids :

$$\frac{\partial \mathcal{L}}{\partial W} = \frac{\partial \mathcal{L}}{\partial y} \times \frac{\partial y}{\partial W} = \frac{\partial \mathcal{L}}{\partial y} \times \frac{\partial f(x, W)}{\partial W}$$

Gradient par rapport à l'entrée :

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \times \frac{\partial y}{\partial x} = \frac{\partial \mathcal{L}}{\partial y} \times \left(1 + \frac{\partial f(x, W)}{\partial x}\right)$$

La magie opère : Même si $\frac{\partial f(x,W)}{\partial x} \to 0$, le terme 1 préserve le gradient.

Sans Connexions Résiduelles, nous avons l'effet multiplicatif dévastateur :

```
Entrée → [C1] → [C2] → [C3] → ... → [CN] → Sortie
```

$$\|\nabla_{W_{\text{couche1}}}\mathcal{L}\| \propto \prod_{i=1}^{N} \|J_i\| \quad \text{avec} \quad \|J_i\| < 1$$

- $\|\nabla_{W_{\text{couche1}}}\mathcal{L}\|$: **Norme du gradient** pour les poids de la première couche.
- $\mathcal{L}$: La fonction de perte (loss) qu'on cherche à minimiser.
- J_i : **Matrice Jacobienne** de la couche i, qui capture comment la sortie change par rapport à l'entrée.

Pour atteindre la couche 1, le gradient doit traverser toutes les couches intermédiaires :

$$Gradient\_Couche1 = Gradient\_Sortie \times J_N \times J_{N-1} \times \ldots \times J_2 \times J_1$$

Chaque J_i est une matrice qui représente la "sensibilité" de la couche i. Si $\|J_i\| < 1$, cela signifie que la couche **atténue** le signal.

Avec Connexions Résiduelles, nous avons l'effet additif salutaire.

$$\|\nabla_{W_{\text{couche1}}}\mathcal{L}\| \propto \sum_{i=1}^{N} \|J_i\|$$

La propagation en arrière peut "sauter" des couches avec les connexions résiduelles. Ce qui donne la possibilité au gradient d'atteigne la première couche :

```
Entrée → [C1] →→→ [C2] →→→ [C3] →→→ ... →→→ [CN] → Sortie
```

1. **Chemin direct** : Sortie → Couche1 (gradient 1.0)
2. **Chemin à travers une couche** : Sortie → Couche2 → Couche1 (gradient $\|J_2\|$)
3. **Chemin à travers deux couches** : Sortie → Couche3 → Couche2 → Couche1 (gradient $\|J_3\| \times \|J_2\|$)

La somme capture l'idée que tous ces chemins contribuent au gradient final.

Un seul chemin : Gradient doit traverser toutes les couches

Sans connexions résiduelles, après plusieurs 6 couches par exemple, le gradient couche 1 a environ 12% de la force originale. Le problème est que la reconnaissance des formes basiques des mains n'évolue presque pas. Ce qui génère une mauvaise apprentissage des patterns élémentaires.

Avec les connexions résiduels (même après 6 couches), le gradient de la couche 1 a environs 400% de la force originale. Ce qui permet à toutes les couches d'apprendre simultanément, efficacement et arriver à reconnaitre des patterns simples et complexes.

Nombre de Couches	Sans Résiduels	Avec Résiduels
3 couches	34%	210%
6 couches	12%	420%
12 couches	1.4%	840%
24 couches	0.02%	1680%

Conclusion : Plus le réseau est profond, plus les résiduels deviennent essentiels pour maintenir des gradients utilisables.

C'est pourquoi les Transformers modernes peuvent avoir des dizaines de couches tout en restant entraînables efficacement !

Dans le Contexte de notre Encodeur, pendant l'optimisation : - **Self-attention** : Apprend à pondérer correctement les frames - **Feed-forward** : Apprend les transformations non-linéaires complexes
- **Grâce aux résiduels** : Les gradients circulent librement à travers toutes les couches

Résultat : Même la première couche reçoit un signal d'erreur fort, permettant un apprentissage coordonné de tout le réseau. Grâce aux chemins directs, on obtient un comportement **additif** plutôt que multiplicatif.

2 Décodeur de la langue des signes en naturelle

Cette partie présente une analyse approfondie de l'architecture du décodeur Transformer spécialisé dans la traduction de la langue des signes. L'intégration du tokenizer DeepSeek 3.1 permet une modélisation linguistique avancée des séquences textuelles générées, offrant ainsi une traduction contextuellement riche et grammaticalement correcte des séquences gestuelles vers le langage naturel.

La traduction automatique de la langue des signes représente un défi complexe nécessitant une modélisation séquentielle sophistiquée. Le décodeur Transformer décrit dans ce document opère la transformation des représentations encodées des séquences gestuelles en séquences textuelles en langue naturelle. L'incorporation du tokenizer DeepSeek 3.1 au niveau de l'entrée du décodeur permet de bénéficier d'un vocabulaire étendu de 128 000 tokens, capturant les subtilités morphologiques et sémantiques du français contemporain.

L'architecture proposée suit le paradigme encodeur-décodeur traditionnel des Transformers, mais avec des adaptations spécifiques au domaine de la langue des signes. Le décodeur prend en entrée les séquences tokenisées par DeepSeek 3.1 et les représentations contextuelles produites par l'encodeur, générant ainsi des séquences textuelles de manière auto-régressive.

```
class TransformerDecoder(nn.Module):
    """
    Complete Transformer Decoder.
```

```
    This module implements the full decoder stack of the Transformer␣
↪model.
    It takes the encoder outputs and generates output sequences
    autoregressively.

    :param output_dim: Dimension of output features (vocabulary size
      or feature dimension).
    :param d_model: Dimension of the model (embedding size).
    :param num_heads: Number of attention heads.
    :param num_layers: Number of decoder layers.
    :param d_ff: Dimension of feed-forward network hidden layer.
    :param dropout: Dropout rate, defaults to 0.1.
    :param max_seq_len:
      Maximum sequence length for positional encoding, defaults to 5000.
    """
    def __init__(
        self,
        output_dim: int,
        d_model: int,
        num_heads: int,
        num_layers: int,
        d_ff: int,
        dropout: float = 0.1,
        max_seq_len: int = 4096,
    ) -> None:
        """Initialize TransformerDecoder module."""
        super().__init__()

        # Output projection (for token embeddings or output features):
        self.embeddings = nn.Embedding(output_dim, d_model, padding_idx=0)

        # Positional encoding:
        self.pos_encoding = PositionalEncoding(d_model, max_seq_len)

        # Dropout for input embeddings:
        self.dropout = nn.Dropout(dropout)

        # Stack of decoder layers:
        self.layers = nn.ModuleList([
            DecoderLayer(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)
        ])

        # Final layer norm
```

```
        # self.norm = nn.LayerNorm(d_model)

        # Final output projection (to vocabulary or target feature␣
↪dimension):
        self.fc = nn.Linear(d_model, output_dim)

    def forward(
        self,
        x: torch.Tensor,
        encoder_output: torch.Tensor,
        self_attn_mask: torch.Tensor = None,
        cross_attn_mask: torch.Tensor = None
    ) -> torch.Tensor:
        """
        Forward pass of Transformer decoder.

        :param x: Input tensor of shape (B, T_dec, output_dim)
          - target sequence.
        :param encoder_output: Encoder output of shape (B, T_enc,␣
↪d_model).
        :param self_attn_mask: Mask for decoder self-attention to prevent
          looking at future tokens, defaults to None.
        :param cross_attn_mask: Mask for cross-attention, defaults to␣
↪None.
        :returns: Output tensor of shape (B, T_dec, output_dim).

        :raises ValueError: If input tensor does not have 3 dimensions.
        """
        # Validate input shape:
        if len(x.shape) != 2:
            raise ValueError(
                "Input must have 3 dimensions (B, T_dec, output_dim),"
                f"got \"{len(x.shape)}\"."
            )

        # Project input to model dimension
        x = self.embeddings(x)  # (B, T_dec, d_model)
        print(x.shape)

        # Add positional encoding and apply dropout
        x = self.pos_encoding(x)
        x = self.dropout(x)

        # Pass through each decoder layer
```

```
        for layer in self.layers:
            x = layer(x, encoder_output, self_attn_mask, cross_attn_mask)

        # Apply final layer norm
        # x = self.norm(x)

        # Project back to output dimension:
        output = self.fc(x)
        return output
```

1. **Initialisation et Configuration du Module**

```
def __init__(
    self,
    output_dim: int,
    d_model: int,
    num_heads: int,
    num_layers: int,
    d_ff: int,
    dropout: float = 0.1,
    max_seq_len: int = 4096,
) -> None:
```

L'initialisation du décodeur commence par la définition des paramètres fondamentaux qui gouvernent sa capacité représentationnelle. Le paramètre `output_dim` correspond exactement à la taille du vocabulaire DeepSeek 3.1, établissant une correspondance bijective entre l'espace de représentation interne et l'espace lexical. Cette dimension typiquement fixée à 128 000 tokens permet de couvrir l'ensemble des unités linguistiques du français, des morphèmes aux expressions idiomatiques.

La dimension du modèle `d_model` fixée à 512 offre un compromis optimal entre expressivité représentationnelle et efficacité computationnelle. Cette valeur, devenue standard dans les architectures Transformer modernes, fournit un espace suffisamment vaste pour encoder les relations sémantiques complexes tout en maintenant une complexité calculatoire raisonnable.

2. **Système d'Embedding et Encodage Positionnel**

```
self.embeddings = nn.Embedding(output_dim, d_model, padding_idx=0)
self.pos_encoding = PositionalEncoding(d_model, max_seq_len)
```

Le système d'embedding transforme les indices token discrets en vecteurs denses de dimension 512 (même dimension que celle de l'encodeur). Chaque token du vocabulaire DeepSeek se voit attribuer une signature vectorielle unique qui capture ses propriétés sémantiques, syntaxiques et distributionnelles. L'utilisation de `padding_idx=0` permet de traiter efficacement les séquences de longueurs variables en ignorant les tokens de padding pendant les calculs d'embedding.

L'encodage positionnel sinusoïdal injecte des informations temporelles essentielles à la mod-

élisation des dépendances séquentielles. Étant donné que le décodeur fonctionne de manière auto-régressive, la préservation de l'ordre temporel est cruciale pour maintenir la cohérence grammaticale des phrases générées. Les fonctions d'encodage positionnel sont définies par :

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

où pos représente la position dans la séquence et i l'indice de dimension. Cette formulation garantit que chaque position reçoit un encodage unique tout en permettant au modèle d'extrapoler à des séquences plus longues que celles rencontrées pendant l'entraînement.

3. **Architecture en Couches et Projection Finale**

```
self.layers = nn.ModuleList([
    DecoderLayer(d_model, num_heads, d_ff, dropout)
    for _ in range(num_layers)
])
self.fc = nn.Linear(d_model, output_dim)
```

L'architecture comprend un empilement de `num_layers` couches de décodeur, généralement fixé à 6 pour les applications de traduction. Chaque couche de décodeur contient trois sous-composants principaux : l'auto-attention masquée, l'attention croisée et le réseau feed-forward positionnel. Cette organisation permet une transformation progressive des représentations, où chaque couche ajoute un niveau supplémentaire de compréhension contextuelle.

La projection linéaire finale via `self.fc` opère la transformation cruciale des représentations internes riches en logits correspondant aux scores de chaque token du vocabulaire DeepSeek. Cette transformation s'effectue selon l'équation :

$$\text{logits} = Wx + b$$

où $W \in \mathbb{R}^{d_{model} \times output\_dim}$ et $b \in \mathbb{R}^{output\_dim}$ sont les paramètres apprenables de la projection.

4. **Transformation des Entrées Tokenisées**

```
x = self.embeddings(x)  # (B, T_dec, d_model)
x = self.pos_encoding(x)
x = self.dropout(x)
```

La passe forward commence par la transformation des séquences tokenisées par DeepSeek 3.1 en représentations vectorielles denses. Cette étape convertit les identifiants numériques discrets en vecteurs continus riches en information sémantique. Par exemple, le token représentant "remerciement" se voit attribuer des caractéristiques vectorielles qui reflètent sa nature poli, sa fréquence d'usage et ses relations avec d'autres tokens apparentés.

L'application successive de l'encodage positionnel et du dropout prévient le surapprentissage tout en préservant l'information temporelle essentielle. Le dropout, avec un taux typique de 0.1, introduit une régularisation stochastique qui force le modèle à développer des représentations redondantes et robustes.

5. **Propagation à Travers les Couches de Décodeur**

```
for layer in self.layers:
    x = layer(x, encoder_output, self_attn_mask, cross_attn_mask)
```

Chaque couche de décodeur opère une transformation séquentielle des représentations via trois mécanismes interdépendants. Le mécanisme d'auto-attention masquée permet au modèle de prendre en compte le contexte gauche uniquement, préservant ainsi la nature autorégressive de la génération. Formellement, l'auto-attention est calculée comme :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

où M représente le masque triangulaire supérieur empêchant l'accès aux tokens futurs, et $d_k = d_{model}/num\_heads$ est la dimension par tête d'attention.

L'attention croisée constitue le pont essentiel entre les représentations gestuelles et linguistiques. Ce mécanisme permet au décodeur d'aligner chaque token généré avec les parties pertinentes de la séquence vidéo encodée. Mathématiquement, l'attention croisée s'exprime comme :

$$\text{CrossAttention}(Q, K_{enc}, V_{enc}) = \text{softmax}\left(\frac{QK_{enc}^T}{\sqrt{d_k}}\right)V_{enc}$$

où Q provient du décodeur et K_{enc}, V_{enc} proviennent de la sortie de l'encodeur.

6. **Projection Finale et Génération de Texte**

```
output = self.fc(x)
```

La projection finale transforme les représentations contextuelles de dimension 512 en logits de dimension 128 000, correspondant aux scores de chaque token du vocabulaire DeepSeek. Ces logits sont ensuite transformés en probabilités via une fonction softmax, permettant la sélection du token suivant selon différentes stratégies de décodage.

Dans le contexte de la traduction de la langue des signes, cette étape doit arbitrer entre plusieurs contraintes : la fidélité au message gestuel original, la correction grammaticale, la fluidité stylistique et la cohérence sémantique. La richesse du vocabulaire DeepSeek permet au modèle de sélectionner des formulations précises qui capturent les nuances du message gestuel.

2.1 Architecture de la Couche de Décodeur Transformer pour la Traduction de la Langue des Signes

Ce document présente une analyse technique approfondie de la couche de décodeur Transformer, composant fondamental dans le système de traduction de la langue des signes vers le langage naturel. La couche de décodeur intègre trois mécanismes d'attention distincts permettant une génération textuelle contextuellement informée et grammaticalement cohérente à partir de séquences gestuelles encodées.

La couche de décodeur (DecoderLayer) représente l'unité de traitement élémentaire dans l'architecture de décodeur Transformer pour la traduction de la langue des signes. Chaque couche opère une transformation séquentielle des représentations linguistiques en intégrant progressivement les informations contextuelles provenant de l'encodeur. La conception modulaire de cette couche permet un empilement multiple, créant ainsi une hiérarchie de traitement où chaque niveau affine la compréhension et la génération textuelle.

L'utilisation du tokenizer DeepSeek 3.1 apporte des avantages significatifs pour la traduction de la langue des signes. La granularité lexicale fine permettra de capturer des constructions linguistiques complexes particulièrement importantes pour restituer la richesse expressive des signes. Par exemple, un geste emphatique pourra être traduit par des adverbes d'intensité spécifiques, tandis qu'un signe dénotant une action répétitive pourra bénéficier de préfixes appropriés.

La taille étendue du vocabulaire réduit considérablement le taux de tokens inconnus, améliorant ainsi la fluidité des traductions générées. De plus, la conception linguistiquement informée du tokenizer facilite l'apprentissage de patterns grammaticaux complexes essentiels à la génération de texte naturel et idiomatique.

L'architecture de la couche de décodeur se distingue par l'intégration de trois sous-systèmes spécialisés : l'auto-attention masquée pour la cohérence linguistique, l'attention croisée pour l'alignement geste-texte, et le réseau feed-forward pour l'enrichissement représentationnel. Cette organisation triadique assure une transformation progressive et contrôlée des représentations internes.

```
class DecoderLayer(nn.Module):
    """
    Single decoder layer of the Transformer.

    This layer contains:
    - Masked multi-head self-attention (to prevent looking at future␣
 ↪tokens)
    - Multi-head cross-attention (attending to encoder outputs)
    - Position-wise feed-forward network
    All with residual connections and layer normalization.

    :param d_model: The dimension of input and output features.
```

```
    :param num_heads: The number of attention heads.
    :param d_ff: The dimension of hidden layer in feed-forward network.
    :param dropout: Dropout rate, defaults to 0.1.
    """
    def __init__(
        self,
        d_model: int,
        num_heads: int,
        d_ff: int,
        dropout: float = 0.1
    ) -> None:
        """Initialize DecoderLayer module."""
        super().__init__()

        # Masked self-attention (prevents attending to future positions):
        self.self_attn = MultiHeadAttention(d_model, num_heads, dropout)
        # Cross-attention (attends to encoder outputs):
        self.cross_attn = MultiHeadAttention(d_model, num_heads, dropout)

        # Feed-forward network
        self.feed_forward = PositionwiseFeedForward(d_model, d_ff,␣
↪dropout)

        # Layer normalization for residual connections:
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)

        self.dropout = nn.Dropout(dropout)

    def forward(
        self,
        x: torch.Tensor,
        encoder_output: torch.Tensor,
        self_attn_mask: torch.Tensor = None,
        cross_attn_mask: torch.Tensor = None
    ) -> torch.Tensor:
        """
        Forward pass of decoder layer.

        :param x: Input tensor from previous decoder layer of shape
          (B, T_dec, C).
        :param encoder_output: Output from encoder of shape (B, T_enc, C).
        :param self_attn_mask: Mask for self-attention to prevent looking
```

```
          ahead, defaults to None.
        :param cross_attn_mask: Mask for cross-attention, defaults to␣
↪None.
        :returns: Output tensor of same shape as input (B, T_dec, C).
        """
        # Masked self-attention with residual connection and layer norm:
        self_attn_output = self.self_attn(x, x, x, self_attn_mask)
        x = x + self.dropout(self_attn_output)  # Residual connection;
        x = self.norm1(x)  # Layer normalization;

        # Cross-attention with residual connection and layer norm
        # Query from decoder, Key/Value from encoder
        cross_attn_output = self.cross_attn(
            query=x, key=encoder_output, value=encoder_output,
            mask=cross_attn_mask
        )
        x = x + self.dropout(cross_attn_output)  # Residual connection;
        x = self.norm2(x)  # Layer normalization;

        # Feed-forward with residual connection and layer norm:
        ff_output = self.feed_forward(x)
        x = x + self.dropout(ff_output)  # Residual connection;
        x = self.norm3(x)  # Layer normalization;
        return x
```

1. **Initialisation et Configuration des Sous-Modules**

```
def __init__(
    self,
    d_model: int,
    num_heads: int,
    d_ff: int,
    dropout: float = 0.1
) -> None:
```

L'initialisation de la couche de décodeur définit les composants fondamentaux qui orchestrent la transformation des représentations. La dimension du modèle `d_model` fixée à 512 constitue l'espace de représentation unifié à travers tous les sous-modules, garantissant la compatibilité dimensionnelle et facilitant les connexions résiduelles.

Le paramètre `num_heads` détermine le degré de spécialisation attentionnelle, permettant au modèle de capturer simultanément différents aspects des relations contextuelles. Typiquement fixé à 8, ce paramètre divise l'espace de représentation en sous-espaces spécialisés où chaque tête d'attention se concentre sur des types spécifiques de dépendances linguistiques et contextuelles.

2. **Architecture des Mécanismes d'Attention**

```
self.self_attn = MultiHeadAttention(d_model, num_heads, dropout)
self.cross_attn = MultiHeadAttention(d_model, num_heads, dropout)
```

L'auto-attention masquée constitue le premier mécanisme de traitement, permettant à chaque position de la séquence de texte en cours de génération d'interagir avec les positions précédentes. Ce mécanisme préserve la propriété auto-régressive essentielle à la génération séquentielle en empêchant l'accès aux tokens futurs. Mathématiquement, l'auto-attention masquée s'exprime comme :

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

où M est une matrice masque avec $M_{ij} = -\infty$ si $i < j$ (empêchant le regard vers le futur) et 0 sinon. Cette formulation garantit que la génération de chaque token ne dépend que des tokens précédemment générés.

L'attention croisée établit le pont crucial entre les domaines gestuel et linguistique. Ce mécanisme permet au décodeur de "consulter" les représentations encodées de la séquence vidéo pour chaque étape de génération textuelle. L'attention croisée est définie par :

$$\text{CrossAttention}(Q, K_{enc}, V_{enc}) = \text{softmax}\left(\frac{QK_{enc}^T}{\sqrt{d_k}}\right)V_{enc}$$

où Q provient des représentations courantes du décodeur, et K_{enc}, V_{enc} proviennent de la sortie de l'encodeur. Ce mécanisme permet un alignement dynamique entre les unités textuelles générées et les segments gestuels pertinents.

3. **Réseau Feed-Forward et Normalisation**

```
self.feed_forward = PositionwiseFeedForward(d_model, d_ff, dropout)
self.norm1 = nn.LayerNorm(d_model)
self.norm2 = nn.LayerNorm(d_model)
self.norm3 = nn.LayerNorm(d_model)
```

Le réseau feed-forward positionnel opère une transformation non-linéaire indépendante sur chaque position de la séquence. Avec une dimension cachée `d_ff` typiquement fixée à 2048 (4 × d_model), ce réseau introduit une capacité computationnelle additionnelle essentielle pour capturer des transformations complexes. L'architecture de ce sous-module suit la formulation :

$$\text{FFN}(x) = W_2 \cdot \text{GELU}(W_1 x + b_1) + b_2$$

où $W_1 \in \mathbb{R}^{d_{model} \times d_{ff}}$, $W_2 \in \mathbb{R}^{d_{ff} \times d_{model}}$, et GELU représente la fonction d'activation Gaussian Error Linear Unit.

Les couches de normalisation (LayerNorm) stabilisent l'apprentissage en normalisant les activations à travers la dimension des features. Cette normalisation s'exprime comme :

$$\text{LayerNorm}(x) = \frac{x - \mu}{\sigma} \cdot \gamma + \beta$$

où μ et σ sont la moyenne et l'écart-type calculés sur la dernière dimension, et γ, β sont des paramètres apprenables.

4. **Processus Séquentiel de Transformation**

```
self_attn_output = self.self_attn(x, x, x, self_attn_mask)
x = x + self.dropout(self_attn_output)
x = self.norm1(x)
```

La première étape de transformation applique l'auto-attention masquée aux représentations courantes du décodeur. Dans le contexte de la traduction de la langue des signes, ce mécanisme permet d'établir la cohérence linguistique de la séquence textuelle en cours de génération. Par exemple, lors de la génération du verbe dans une phrase, l'auto-attention masquée permet de maintenir l'accord avec le sujet déjà généré.

La connexion résiduelle ($x + \text{dropout}(self\_attn\_output)$) préserve l'information originale tout en permettant au mécanisme d'attention d'apporter des améliorations incrémentales. Cette approche atténue le problème des gradients disparaissants et facilite l'apprentissage de transformations profondes.

5. **Attention Croisée et Alignement Geste-texte**

```
cross_attn_output = self.cross_attn(
    query=x, key=encoder_output, value=encoder_output,
    mask=cross_attn_mask
)
x = x + self.dropout(cross_attn_output)
x = self.norm2(x)
```

L'attention croisée représente l'étape cruciale d'intégration des informations gestuelles dans le processus de génération textuelle. Ce mécanisme permet au décodeur d'aligner dynamiquement chaque unité textuelle générée avec les segments pertinents de la séquence vidéo encodée.

Dans le contexte pratique de la traduction, lorsque le décodeur génère un mot comme "maison", l'attention croisée lui permet de se concentrer sur les frames vidéo où les mains forment le geste correspondant au toit caractéristique. De même, pour générer des adverbes d'intensité comme "vraiment" ou "beaucoup", le mécanisme peut s'aligner sur les frames montrant une amplitude gestuelle accentuée.

Le masque d'attention croisée (`cross_attn_mask`) permet d'ignorer les positions de padding dans la séquence encodée, optimisant ainsi le calcul attentionnel sur les seules régions informationnelles.

6. **Raffinement Représentationnel par Feed-Forward**

```
ff_output = self.feed_forward(x)
x = x + self.dropout(ff_output)
x = self.norm3(x)
```

La dernière étape applique le réseau feed-forward positionnel pour enrichir les représentations avec des transformations non-linéaires complexes. Ce sous-module opère indépendamment sur chaque position, permettant un raffinement local des représentations tout en maintenant l'information contextuelle capturée par les mécanismes d'attention précédents.

Dans le contexte de la génération textuelle pour la traduction de la langue des signes, cette étape permet de transformer les représentations alignées geste-texte en représentations linguistiquement riches, préparant ainsi le terrain pour la génération du token suivant. Par exemple, après l'alignement avec les gestes d'affection, le réseau feed-forward peut enrichir la représentation pour favoriser la génération de termes affectueux spécifiques.

7. **Dynamique d'Apprentissage et Régularisation**

L'utilisation systématique du dropout sur chaque sortie de sous-module introduit une régularisation stochastique essentielle pour prévenir le surapprentissage. Le taux de dropout typique de 0.1 désactive aléatoirement 10% des unités pendant l'entraînement, forçant le modèle à développer des représentations redondantes et robustes.

Les connexions résiduelles successives créent des chemins de gradient directs à travers les multiples transformations, atténuant le problème des gradients disparaissants et permettant un apprentissage efficace même avec des réseaux profonds. Cette architecture facilite la circulation de l'information et des gradients à travers les multiples couches de traitement.

2.1.1 Applications à la Traduction de la Langue des Signes

La conception de la couche de décodeur est particulièrement adaptée aux défis spécifiques de la traduction de la langue des signes. L'attention croisée permet de gérer l'asynchronisme naturel entre les séquences gestuelles et textuelles, tandis que l'auto-attention masquée assure la cohérence grammaticale des phrases générées.

La capacité à capturer des relations complexes à longue distance est essentielle pour traduire les structures grammaticales spécifiques à la langue des signes, où l'information est souvent distribuée temporellement. L'architecture multi-têtes permet de modéliser simultanément différents aspects des relations contextuelles, depuis les accords grammaticaux jusqu'aux relations sémantiques profondes.

3 Configuration Typique du modèle pour la Langue des Signes

Paramètre	Valeur Recommandée	Justification
`input_dim`	126	42 points MediaPipe $ $\times 3$ $coordonnes(x, y, z)$
`d_model`	512	Bon compromis expressivité/calcul
`num_heads`	8	Divise 512. Capture divers aspects des gestes
`num_layers`	6	Suffisant pour modéliser la complexité temporelle
`d_ff`	2048	$4 \times d_{model}$, standard pour les Transformers
`max_seq_len`	250	Supporte 5 secondes à raisons de 50 fps

Avantages pour la Traduction de Langue des Signes :

1. **Capture des Dépendances Temporelles** : Comprend les relations entre gestes séparés dans le temps
2. **Traitement Parallèle** : Analyse toute la séquence simultanément (vs RNN séquentiels)
3. **Robustesse aux Variations** : S'adapte aux différences de vitesse et de style entre signeurs
4. **Contextualisation Rich** : Chaque geste est interprété dans le contexte de l'ensemble du message

Cette architecture permet de transformer une séquence vidéo brute en une représentation vectorielle structurée qui capture toute la richesse linguistique et temporelle de la langue des signes, préparant ainsi le terrain pour la phase de décodage vers la langue naturelle.

4 Testons l'inférence

```python
@dataclass
class EncoderConfig:
    batch_size: int = 4  # B: Number of sequences in batch;
    max_seq_len: int = 250  # T: Number of frames in sequence;
    input_dim: int = 2048  # C: Size of landmark feature vector from each␣
 ↪image;
    d_model: int = 512  # Model dimension (embedding size);
    num_heads: int = 8  # Number of attention heads;
    num_layers: int = 6  # Number of encoder layers;
    d_ff: int = 2048  # Hidden dimension in feed-forward network;
    dropout: int = 0.1  # Dropout rate;
```

```python
@dataclass
class DecoderConfig:
    """
    Configuration dataclass for Transformer decoder.

    This class holds all the hyperparameters needed to configure
    the Transformer decoder.

    :param batch_size: Number of sequences in batch, defaults to 8.
    :param max_seq_len: Maximum sequence length for decoder, defaults to␣
 ↪100.
    :param output_dim: Dimension of output features, defaults to 256.
    :param d_model: Model dimension (embedding size), defaults to 512.
    :param num_heads: Number of attention heads, defaults to 8.
    :param num_layers: Number of decoder layers, defaults to 6.
    :param d_ff: Hidden dimension in feed-forward network, defaults to␣
 ↪2048.
    :param dropout: Dropout rate, defaults to 0.1.
    """
    batch_size: int = 1
    max_seq_len: int = 1024
    output_dim: int = 130_000
    d_model: int = 512
    num_heads: int = 8
    num_layers: int = 6
    d_ff: int = 2048
    dropout: float = 0.1
```

La fonction suivante génère un masque d'attention triangulaire inférieur essentiel au fonctionnement auto-régressif du décodeur. Ce masque empêche chaque position de la séquence de "regarder" vers les positions futures, préservant ainsi la propriété fondamentale de génération séquentielle où chaque token ne peut dépendre que des tokens précédemment générés. En pratique, cette matrice binaire de forme (seq_len, seq_len) contient des 1 dans sa partie triangulaire inférieure (incluant la diagonale) et des 0 dans sa partie triangulaire supérieure, forçant mécaniquement le modèle à respecter l'ordre temporel lors du calcul des scores d'attention. Cette contrainte est cruciale pendant l'entraînement teacher forcing où la séquence cible complète est disponible, mais où le modèle doit apprendre à générer de manière séquentielle comme il le fera pendant l'inférence.

```python
def create_look_ahead_mask(seq_len: int) -> torch.Tensor:
    """
    Create look-ahead mask for decoder self-attention.

    This mask prevents the decoder from attending to future positions
```

```
    during training when the entire target sequence is available.

    :param seq_len: Length of the target sequence.
    :returns: Look-ahead mask tensor of shape (seq_len, seq_len).
    """
    # Create lower triangular matrix (ones in lower triangle, zeros in␣
↪upper):
    mask = torch.tril(torch.ones(seq_len, seq_len))
    return mask
```

La fonction suivante fournit une analyse détaillée de l'architecture et des caractéristiques computationnelles d'un modèle PyTorch. En utilisant la bibliothèque `torchsummary`, elle génère un rapport complet incluant la structure hiérarchique des couches, le nombre de paramètres entraînables, la consommation mémoire et la forme des tenseurs à chaque étape de traitement. Le paramètre `depth` permet de contrôler le niveau de détail de l'analyse, tandis que la mesure du temps d'inférence offre une estimation des performances en millisecondes. Cette fonction est particulièrement utile pour debuguer l'architecture, optimiser l'utilisation mémoire et comparer différentes configurations de modèles dans le contexte de la traduction de langue des signes où l'efficacité computationnelle est cruciale pour le traitement en temps réel des séquences vidéo.

```
def print_model_summary(
    model: nn.Module,
    input_data: tuple,
    depth: int=8,
    device=None
):
    """
    This function to make summary for the model instance received
    by arguments.
    """
    start = tm.time()
    state = summary(
        model=model, input_data=input_data, device=device, depth=depth,␣
↪verbose=1
    )
    end = tm.time()
    inference_time = (end - start) * 1000
    return state, inference_time
```

```
# Hyperparameters
model_config = EncoderConfig()
# Create encoder
encoder = TransformerEncoder(
```

```
    input_dim=model_config.input_dim,
    d_model=model_config.d_model,
    num_heads=model_config.num_heads,
    num_layers=model_config.num_layers,
    d_ff=model_config.d_ff,
    dropout=model_config.dropout
)

# Test with random input (B, T, C):
input_shape = (
    model_config.batch_size, model_config.max_seq_len,
    model_config.input_dim
)
x = torch.randn(input_shape)
start = tm.time()
output = encoder(x)
end = tm.time()
inference_time = (end - start) * 1000
LOGGER.info("Inference time: " + str(inference_time) + " ms.")
print_model_summary(encoder, (x,))

# Print shapes and model information
LOGGER.info("Transformer Encoder Example")
LOGGER.info("=" * 80)
LOGGER.info(f"Input shape: {x.shape}")
LOGGER.info(f"Output shape: {output.shape}")
LOGGER.info(
    "Number of parameters: "
    f"{sum(p.numel() for p in encoder.parameters()):,}"
)
LOGGER.info(f"Input projection: {encoder.input_projection}")
LOGGER.info(f"Number of encoder layers: {len(encoder.layers)}")
LOGGER.info(f"Positional encoding: {encoder.pos_encoding}")
```

```
2025-10-29 23:37:26,944 - - INFO - Inference time: 6316.2291049957275 ms.
2025-10-29 23:37:27,884 - - INFO - Transformer Encoder Example
2025-10-29 23:37:27,886 - - INFO -
================================================================================
2025-10-29 23:37:27,888 - - INFO - Input shape: torch.Size([4, 250, 2048])
2025-10-29 23:37:27,889 - - INFO - Output shape: torch.Size([4, 250, 512])
2025-10-29 23:37:27,894 - - INFO - Number of parameters: 19,955,200
2025-10-29 23:37:27,899 - - INFO - Input projection:␣
 ↪Linear(in_features=2048,
out_features=512, bias=True)
2025-10-29 23:37:27,901 - - INFO - Number of encoder layers: 6
```

```
2025-10-29 23:37:27,903 - - INFO - Positional encoding: PositionalEncoding()
==========================================================================================
===============
Layer (type:depth-idx)                             Output Shape              ␣
 ↪Param #
==========================================================================================
===============
TransformerEncoder                                 [4, 250, 512]              --
Linear: 1-1                                       [4, 250, 512]
1,049,088
PositionalEncoding: 1-2                           [4, 250, 512]              --
Dropout: 1-3                                      [4, 250, 512]              --
ModuleList: 1-4                                   --                         --
    EncoderLayer: 2-1                            [4, 250, 512]              --
        MultiHeadAttention: 3-1                  [4, 250, 512]              --
            Linear: 4-1                          [4, 250, 512]              262,144
            Linear: 4-2                          [4, 250, 512]              262,144
            Linear: 4-3                          [4, 250, 512]              262,144
            Dropout: 4-4                         [4, 8, 250, 250]           --
            Linear: 4-5                          [4, 250, 512]              262,656
        Dropout: 3-2                             [4, 250, 512]              --
        LayerNorm: 3-3                           [4, 250, 512]              1,024
        PositionwiseFeedForward: 3-4             [4, 250, 512]              --
            Linear: 4-6                          [4, 250, 2048]
1,050,624
            GELU: 4-7                            [4, 250, 2048]             --
            Dropout: 4-8                         [4, 250, 2048]             --
            Linear: 4-9                          [4, 250, 512]
1,049,088
        Dropout: 3-5                             [4, 250, 512]              --
        LayerNorm: 3-6                           [4, 250, 512]              1,024
    EncoderLayer: 2-2                            [4, 250, 512]              --
        MultiHeadAttention: 3-7                  [4, 250, 512]              --
            Linear: 4-10                         [4, 250, 512]              262,144
            Linear: 4-11                         [4, 250, 512]              262,144
            Linear: 4-12                         [4, 250, 512]              262,144
            Dropout: 4-13                        [4, 8, 250, 250]           --
            Linear: 4-14                         [4, 250, 512]              262,656
        Dropout: 3-8                             [4, 250, 512]              --
        LayerNorm: 3-9                           [4, 250, 512]              1,024
        PositionwiseFeedForward: 3-10            [4, 250, 512]              --
            Linear: 4-15                         [4, 250, 2048]
1,050,624
            GELU: 4-16                           [4, 250, 2048]             --
```

```
            Dropout: 4-17                  [4, 250, 2048]            --
            Linear: 4-18                   [4, 250, 512]
1,049,088
        Dropout: 3-11                       [4, 250, 512]             --
        LayerNorm: 3-12                     [4, 250, 512]             1,024
    EncoderLayer: 2-3                       [4, 250, 512]             --
        MultiHeadAttention: 3-13            [4, 250, 512]             --
            Linear: 4-19                   [4, 250, 512]             262,144
            Linear: 4-20                   [4, 250, 512]             262,144
            Linear: 4-21                   [4, 250, 512]             262,144
            Dropout: 4-22                  [4, 8, 250, 250]          --
            Linear: 4-23                   [4, 250, 512]             262,656
        Dropout: 3-14                       [4, 250, 512]             --
        LayerNorm: 3-15                     [4, 250, 512]             1,024
        PositionwiseFeedForward: 3-16       [4, 250, 512]             --
            Linear: 4-24                   [4, 250, 2048]
1,050,624
            GELU: 4-25                     [4, 250, 2048]            --
            Dropout: 4-26                  [4, 250, 2048]            --
            Linear: 4-27                   [4, 250, 512]
1,049,088
        Dropout: 3-17                       [4, 250, 512]             --
        LayerNorm: 3-18                     [4, 250, 512]             1,024
    EncoderLayer: 2-4                       [4, 250, 512]             --
        MultiHeadAttention: 3-19            [4, 250, 512]             --
            Linear: 4-28                   [4, 250, 512]             262,144
            Linear: 4-29                   [4, 250, 512]             262,144
            Linear: 4-30                   [4, 250, 512]             262,144
            Dropout: 4-31                  [4, 8, 250, 250]          --
            Linear: 4-32                   [4, 250, 512]             262,656
        Dropout: 3-20                       [4, 250, 512]             --
        LayerNorm: 3-21                     [4, 250, 512]             1,024
        PositionwiseFeedForward: 3-22       [4, 250, 512]             --
            Linear: 4-33                   [4, 250, 2048]
1,050,624
            GELU: 4-34                     [4, 250, 2048]            --
            Dropout: 4-35                  [4, 250, 2048]            --
            Linear: 4-36                   [4, 250, 512]
1,049,088
        Dropout: 3-23                       [4, 250, 512]             --
        LayerNorm: 3-24                     [4, 250, 512]             1,024
    EncoderLayer: 2-5                       [4, 250, 512]             --
        MultiHeadAttention: 3-25            [4, 250, 512]             --
            Linear: 4-37                   [4, 250, 512]             262,144
            Linear: 4-38                   [4, 250, 512]             262,144
```

```
            Linear: 4-39                  [4, 250, 512]             262,144
            Dropout: 4-40                 [4, 8, 250, 250]          --
            Linear: 4-41                  [4, 250, 512]             262,656
        Dropout: 3-26                      [4, 250, 512]             --
        LayerNorm: 3-27                    [4, 250, 512]             1,024
        PositionwiseFeedForward: 3-28      [4, 250, 512]             --
            Linear: 4-42                  [4, 250, 2048]
1,050,624
            GELU: 4-43                    [4, 250, 2048]            --
            Dropout: 4-44                 [4, 250, 2048]            --
            Linear: 4-45                  [4, 250, 512]
1,049,088
        Dropout: 3-29                      [4, 250, 512]             --
        LayerNorm: 3-30                    [4, 250, 512]             1,024
    EncoderLayer: 2-6                       [4, 250, 512]             --
        MultiHeadAttention: 3-31           [4, 250, 512]             --
            Linear: 4-46                  [4, 250, 512]             262,144
            Linear: 4-47                  [4, 250, 512]             262,144
            Linear: 4-48                  [4, 250, 512]             262,144
            Dropout: 4-49                 [4, 8, 250, 250]          --
            Linear: 4-50                  [4, 250, 512]             262,656
        Dropout: 3-32                      [4, 250, 512]             --
        LayerNorm: 3-33                    [4, 250, 512]             1,024
        PositionwiseFeedForward: 3-34      [4, 250, 512]             --
            Linear: 4-51                  [4, 250, 2048]
1,050,624
            GELU: 4-52                    [4, 250, 2048]            --
            Dropout: 4-53                 [4, 250, 2048]            --
            Linear: 4-54                  [4, 250, 512]
1,049,088
        Dropout: 3-35                      [4, 250, 512]             --
        LayerNorm: 3-36                    [4, 250, 512]             1,024
LayerNorm: 1-5                               [4, 250, 512]               1,024
==========================================================================
==============
Total params: 19,955,200
Trainable params: 19,955,200
Non-trainable params: 0
Total mult-adds (M): 79.82
==========================================================================
==============
Input size (MB): 8.19
Forward/backward pass size (MB): 278.53
Params size (MB): 79.82
Estimated Total Size (MB): 366.54
```

==========

```
# Configuration:
model_config = DecoderConfig()
# Create decoder instance:
decoder = TransformerDecoder(
    output_dim=model_config.output_dim,
    d_model=model_config.d_model,
    num_heads=model_config.num_heads,
    num_layers=model_config.num_layers,
    d_ff=model_config.d_ff,
    dropout=model_config.dropout
)
decoder.eval()

# Create sample tensors
# Target sequence (what we want to generate)
target_seq_shape = (model_config.batch_size, model_config.max_seq_len)
target_seq = torch.randint(0, model_config.output_dim - 1,␣
 ↪target_seq_shape)

# Encoder output (from the encoder module)
encoder_output_shape = (
    model_config.batch_size, model_config.max_seq_len, model_config.
 ↪d_model
)
encoder_output = torch.randn(encoder_output_shape)

# Create look-ahead mask for decoder self-attention
look_ahead_mask = create_look_ahead_mask(model_config.max_seq_len)
look_ahead_mask = look_ahead_mask \
    .unsqueeze(0) \
    .unsqueeze(0)  # Add batch and head dimensions;
LOGGER.info("look_ahead_mask shape: " + str(look_ahead_mask.shape))
LOGGER.info("look_ahead_mask:\n" + repr(look_ahead_mask))

# Forward pass through decoder:
output = decoder(
    x=target_seq, encoder_output=encoder_output,
    self_attn_mask=look_ahead_mask
)
_, inference_time = print_model_summary(
    decoder, (target_seq, encoder_output)
```

```
)
LOGGER.info("Inference time: " + str(inference_time) + " ms.")

# Print shapes and model information
LOGGER.info("Transformer Decoder Example")
LOGGER.info("=" * 80)
LOGGER.info(f"Target sequence shape: {target_seq.shape}")
LOGGER.info(f"Encoder output shape: {encoder_output.shape}")
LOGGER.info(f"Decoder output shape: {output.shape}")
LOGGER.info(f"Look-ahead mask shape: {look_ahead_mask.shape}")
LOGGER.info(
    f"Number of parameters: "
    f"{sum(p.numel() for p in decoder.parameters()):,}"
)
LOGGER.info(f"Number of decoder layers: {len(decoder.layers)}")
```

```
2025-10-29 23:37:35,886 - - INFO - look_ahead_mask shape: torch.Size([1, 1,
1024, 1024])
2025-10-29 23:37:35,936 - - INFO - look_ahead_mask:
tensor([[[[1., 0., 0.,  ..., 0., 0., 0.],
          [1., 1., 0.,  ..., 0., 0., 0.],
          [1., 1., 1.,  ..., 0., 0., 0.],
          ...,
          [1., 1., 1.,  ..., 1., 0., 0.],
          [1., 1., 1.,  ..., 1., 1., 0.],
          [1., 1., 1.,  ..., 1., 1., 1.]]]])

torch.Size([1, 1024, 512])
torch.Size([1, 1024, 512])

2025-10-29 23:37:47,766 - - INFO - Inference time: 3956.8819999694824 ms.
2025-10-29 23:37:47,768 - - INFO - Transformer Decoder Example
2025-10-29 23:37:47,769 - - INFO -
================================================================================
2025-10-29 23:37:47,771 - - INFO - Target sequence shape: torch.Size([1,␣
 ↪1024])
2025-10-29 23:37:47,772 - - INFO - Encoder output shape: torch.Size([1,␣
 ↪1024,
512])
2025-10-29 23:37:47,775 - - INFO - Decoder output shape: torch.Size([1,␣
 ↪1024,
130000])
2025-10-29 23:37:47,776 - - INFO - Look-ahead mask shape: torch.Size([1, 1,
1024, 1024])
2025-10-29 23:37:47,779 - - INFO - Number of parameters: 158,455,760
```

```
2025-10-29 23:37:47,784 - - INFO - Number of decoder layers: 6

==========================================================================================
Layer (type:depth-idx)                        Output Shape              ␣
 ↪Param #
==========================================================================================
TransformerDecoder                            [1, 1024, 130000]         --
Embedding: 1-1                                [1, 1024, 512]
66,560,000
PositionalEncoding: 1-2                       [1, 1024, 512]            --
Dropout: 1-3                                  [1, 1024, 512]            --
ModuleList: 1-4                               --                        --
    DecoderLayer: 2-1                         [1, 1024, 512]            --
        MultiHeadAttention: 3-1               [1, 1024, 512]            --
            Linear: 4-1                       [1, 1024, 512]            262,144
            Linear: 4-2                       [1, 1024, 512]            262,144
            Linear: 4-3                       [1, 1024, 512]            262,144
            Dropout: 4-4                      [1, 8, 1024, 1024]        --
            Linear: 4-5                       [1, 1024, 512]            262,656
        Dropout: 3-2                          [1, 1024, 512]            --
        LayerNorm: 3-3                        [1, 1024, 512]            1,024
        MultiHeadAttention: 3-4               [1, 1024, 512]            --
            Linear: 4-6                       [1, 1024, 512]            262,144
            Linear: 4-7                       [1, 1024, 512]            262,144
            Linear: 4-8                       [1, 1024, 512]            262,144
            Dropout: 4-9                      [1, 8, 1024, 1024]        --
            Linear: 4-10                      [1, 1024, 512]            262,656
        Dropout: 3-5                          [1, 1024, 512]            --
        LayerNorm: 3-6                        [1, 1024, 512]            1,024
        PositionwiseFeedForward: 3-7          [1, 1024, 512]            --
            Linear: 4-11                      [1, 1024, 2048]
1,050,624
            GELU: 4-12                        [1, 1024, 2048]           --
            Dropout: 4-13                     [1, 1024, 2048]           --
            Linear: 4-14                      [1, 1024, 512]
1,049,088
        Dropout: 3-8                          [1, 1024, 512]            --
        LayerNorm: 3-9                        [1, 1024, 512]            1,024
    DecoderLayer: 2-2                         [1, 1024, 512]            --
        MultiHeadAttention: 3-10              [1, 1024, 512]            --
            Linear: 4-15                      [1, 1024, 512]            262,144
            Linear: 4-16                      [1, 1024, 512]            262,144
            Linear: 4-17                      [1, 1024, 512]            262,144
```

```
            Dropout: 4-18                [1, 8, 1024, 1024]        --
            Linear: 4-19                 [1, 1024, 512]            262,656
        Dropout: 3-11                     [1, 1024, 512]            --
        LayerNorm: 3-12                   [1, 1024, 512]            1,024
        MultiHeadAttention: 3-13          [1, 1024, 512]            --
            Linear: 4-20                 [1, 1024, 512]            262,144
            Linear: 4-21                 [1, 1024, 512]            262,144
            Linear: 4-22                 [1, 1024, 512]            262,144
            Dropout: 4-23                [1, 8, 1024, 1024]        --
            Linear: 4-24                 [1, 1024, 512]            262,656
        Dropout: 3-14                     [1, 1024, 512]            --
        LayerNorm: 3-15                   [1, 1024, 512]            1,024
        PositionwiseFeedForward: 3-16     [1, 1024, 512]            --
            Linear: 4-25                 [1, 1024, 2048]
1,050,624
            GELU: 4-26                   [1, 1024, 2048]           --
            Dropout: 4-27                [1, 1024, 2048]           --
            Linear: 4-28                 [1, 1024, 512]
1,049,088
        Dropout: 3-17                     [1, 1024, 512]            --
        LayerNorm: 3-18                   [1, 1024, 512]            1,024
    DecoderLayer: 2-3                      [1, 1024, 512]            --
        MultiHeadAttention: 3-19          [1, 1024, 512]            --
            Linear: 4-29                 [1, 1024, 512]            262,144
            Linear: 4-30                 [1, 1024, 512]            262,144
            Linear: 4-31                 [1, 1024, 512]            262,144
            Dropout: 4-32                [1, 8, 1024, 1024]        --
            Linear: 4-33                 [1, 1024, 512]            262,656
        Dropout: 3-20                     [1, 1024, 512]            --
        LayerNorm: 3-21                   [1, 1024, 512]            1,024
        MultiHeadAttention: 3-22          [1, 1024, 512]            --
            Linear: 4-34                 [1, 1024, 512]            262,144
            Linear: 4-35                 [1, 1024, 512]            262,144
            Linear: 4-36                 [1, 1024, 512]            262,144
            Dropout: 4-37                [1, 8, 1024, 1024]        --
            Linear: 4-38                 [1, 1024, 512]            262,656
        Dropout: 3-23                     [1, 1024, 512]            --
        LayerNorm: 3-24                   [1, 1024, 512]            1,024
        PositionwiseFeedForward: 3-25     [1, 1024, 512]            --
            Linear: 4-39                 [1, 1024, 2048]
1,050,624
            GELU: 4-40                   [1, 1024, 2048]           --
            Dropout: 4-41                [1, 1024, 2048]           --
            Linear: 4-42                 [1, 1024, 512]
1,049,088
```

```
        Dropout: 3-26                  [1, 1024, 512]            --
        LayerNorm: 3-27                [1, 1024, 512]            1,024
    DecoderLayer: 2-4                  [1, 1024, 512]            --
        MultiHeadAttention: 3-28       [1, 1024, 512]            --
            Linear: 4-43               [1, 1024, 512]            262,144
            Linear: 4-44               [1, 1024, 512]            262,144
            Linear: 4-45               [1, 1024, 512]            262,144
            Dropout: 4-46              [1, 8, 1024, 1024]        --
            Linear: 4-47               [1, 1024, 512]            262,656
        Dropout: 3-29                  [1, 1024, 512]            --
        LayerNorm: 3-30                [1, 1024, 512]            1,024
        MultiHeadAttention: 3-31       [1, 1024, 512]            --
            Linear: 4-48               [1, 1024, 512]            262,144
            Linear: 4-49               [1, 1024, 512]            262,144
            Linear: 4-50               [1, 1024, 512]            262,144
            Dropout: 4-51              [1, 8, 1024, 1024]        --
            Linear: 4-52               [1, 1024, 512]            262,656
        Dropout: 3-32                  [1, 1024, 512]            --
        LayerNorm: 3-33                [1, 1024, 512]            1,024
        PositionwiseFeedForward: 3-34  [1, 1024, 512]            --
            Linear: 4-53               [1, 1024, 2048]
1,050,624
            GELU: 4-54                 [1, 1024, 2048]           --
            Dropout: 4-55              [1, 1024, 2048]           --
            Linear: 4-56               [1, 1024, 512]
1,049,088
        Dropout: 3-35                  [1, 1024, 512]            --
        LayerNorm: 3-36                [1, 1024, 512]            1,024
    DecoderLayer: 2-5                  [1, 1024, 512]            --
        MultiHeadAttention: 3-37       [1, 1024, 512]            --
            Linear: 4-57               [1, 1024, 512]            262,144
            Linear: 4-58               [1, 1024, 512]            262,144
            Linear: 4-59               [1, 1024, 512]            262,144
            Dropout: 4-60              [1, 8, 1024, 1024]        --
            Linear: 4-61               [1, 1024, 512]            262,656
        Dropout: 3-38                  [1, 1024, 512]            --
        LayerNorm: 3-39                [1, 1024, 512]            1,024
        MultiHeadAttention: 3-40       [1, 1024, 512]            --
            Linear: 4-62               [1, 1024, 512]            262,144
            Linear: 4-63               [1, 1024, 512]            262,144
            Linear: 4-64               [1, 1024, 512]            262,144
            Dropout: 4-65              [1, 8, 1024, 1024]        --
            Linear: 4-66               [1, 1024, 512]            262,656
        Dropout: 3-41                  [1, 1024, 512]            --
        LayerNorm: 3-42                [1, 1024, 512]            1,024
```

```
        PositionwiseFeedForward: 3-43     [1, 1024, 512]            --
            Linear: 4-67                  [1, 1024, 2048]
1,050,624
            GELU: 4-68                    [1, 1024, 2048]           --
            Dropout: 4-69                 [1, 1024, 2048]           --
            Linear: 4-70                  [1, 1024, 512]
1,049,088
        Dropout: 3-44                     [1, 1024, 512]            --
        LayerNorm: 3-45                   [1, 1024, 512]            1,024
    DecoderLayer: 2-6                      [1, 1024, 512]            --
        MultiHeadAttention: 3-46          [1, 1024, 512]            --
            Linear: 4-71                  [1, 1024, 512]            262,144
            Linear: 4-72                  [1, 1024, 512]            262,144
            Linear: 4-73                  [1, 1024, 512]            262,144
            Dropout: 4-74                 [1, 8, 1024, 1024]        --
            Linear: 4-75                  [1, 1024, 512]            262,656
        Dropout: 3-47                     [1, 1024, 512]            --
        LayerNorm: 3-48                   [1, 1024, 512]            1,024
        MultiHeadAttention: 3-49          [1, 1024, 512]            --
            Linear: 4-76                  [1, 1024, 512]            262,144
            Linear: 4-77                  [1, 1024, 512]            262,144
            Linear: 4-78                  [1, 1024, 512]            262,144
            Dropout: 4-79                 [1, 8, 1024, 1024]        --
            Linear: 4-80                  [1, 1024, 512]            262,656
        Dropout: 3-50                     [1, 1024, 512]            --
        LayerNorm: 3-51                   [1, 1024, 512]            1,024
        PositionwiseFeedForward: 3-52     [1, 1024, 512]            --
            Linear: 4-81                  [1, 1024, 2048]
1,050,624
            GELU: 4-82                    [1, 1024, 2048]           --
            Dropout: 4-83                 [1, 1024, 2048]           --
            Linear: 4-84                  [1, 1024, 512]
1,049,088
        Dropout: 3-53                     [1, 1024, 512]            --
        LayerNorm: 3-54                   [1, 1024, 512]            1,024
Linear: 1-5                                [1, 1024, 130000]
66,690,000
==========================================================================
==============
Total params: 158,455,760
Trainable params: 158,455,760
Non-trainable params: 0
Total mult-adds (M): 158.46
==========================================================================
==============
```

```
Input size (MB): 2.11
Forward/backward pass size (MB): 1471.81
Params size (MB): 633.82
Estimated Total Size (MB): 2107.74
==========================================================================
===============
```