Transformer Notes

Yue Wu

February 2026

Contents

1 Input Embedding implementation 3

2 Positional Encoding 4

3 Layer Norm 8

4 Multi-head Attention 10

5 Position-wise Feed-Forward Network 14

6 Residual Connection with Pre-Layer Normalization 16

7 Encoder Block 17

8 Transformer Encoder 20

9 Decoder Block 22

10 Transformer Decoder 23

11 Projection Layer 25

12 Full Transformer Architecture 25

13 build_transformer 29

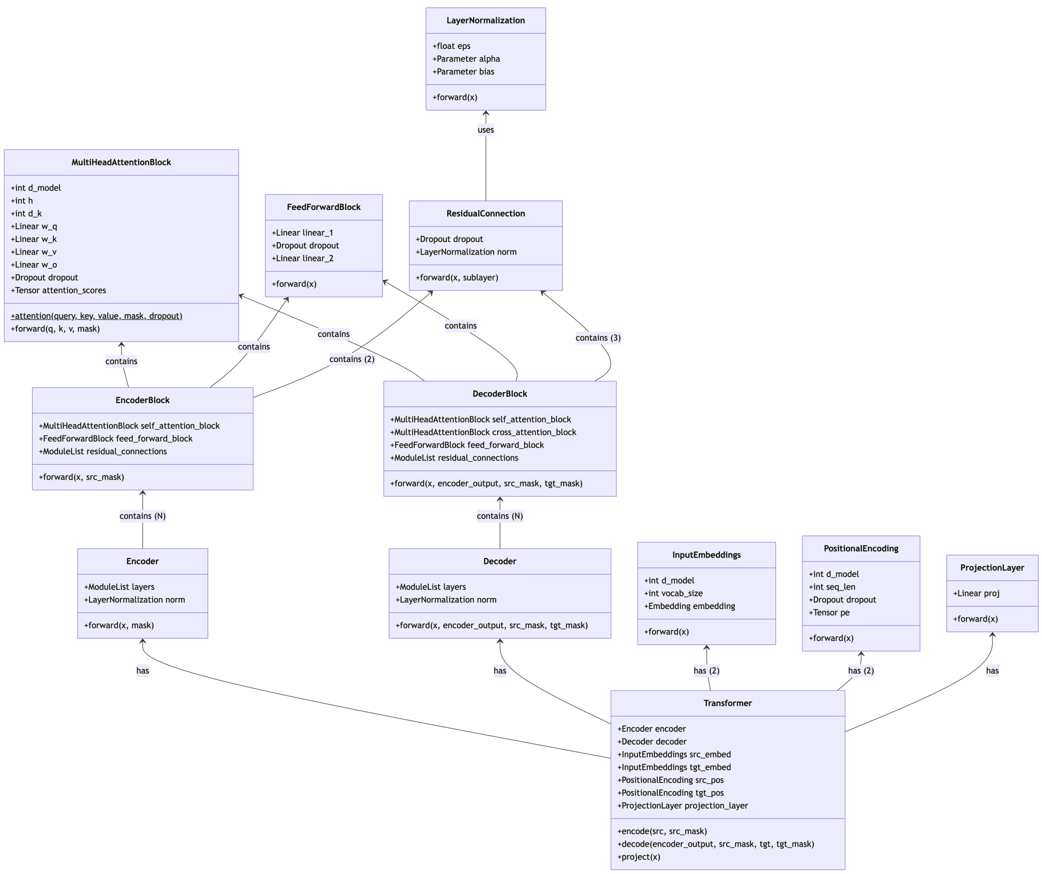

Figure 1: UML diagram

1 Input Embedding implementation

```
import torch
import torch.nn as nn
import math

class InputEmbeddings(nn.Module):
    def __init__(self, d_model:int, vocabu_size:int):
        super().__init__()
        self.d_model = d_model
        self.vocabu_size = vocabu_size
        self.embedding = nn.Embedding(vocabu_size, d_model)
    def forward(self, x):
        return self.embedding(x) * math.sqrt(self.d_model)
```

Let

$$E \in \mathbb{R}^{V\times d} \quad \text{(embedding matrix, } V = \text{vocabulary size, } d = \text{model dim)}$$

and let the input token indices be

$$X \in \{0, \dots, V-1\}^{B\times S}$$

with B = batch size and S = sequence length.

We can represent the lookup at each position as a one-hot vector:

$$\mathbf{1}(X_{b,s}) \in \{0,1\}^V \subset \mathbb{R}^V, \quad \text{where } \mathbf{1}(X_{b,s})_i = \begin{cases} 1 & \text{if } i = X_{b,s}, \\ 0 & \text{otherwise.} \end{cases}$$

Then the embedded tensor (before scaling) can be written as:

$$\tilde{Y}_{b,s,:} = \mathbf{1}(X_{b,s})^\top E \in \mathbb{R}^d.$$

Stacking over batch and sequence gives:

$$\tilde{Y} \in \mathbb{R}^{B\times S\times d}.$$

Finally the module multiplies by $\sqrt{d}$ (to scale the embeddings):

$$Y = \tilde{Y} \cdot \sqrt{d} \in \mathbb{R}^{B\times S\times d}.$$

Equivalently, if we expand the one-hot vectors into a 3-D tensor $O \in \{0,1\}^{B\times S\times V}$, we can write

$$\tilde{Y}_{b,s,:} = O_{b,s,:}^\top E,$$

2 Positional Encoding

The sinusoidal positional encoding is defined as

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

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

Instead of explicitly dividing by $10000^{\frac{2i}{d_{\mathrm{model}}}}$, the implementation precomputes:

$$\frac{1}{10000^{\frac{2i}{d_{\mathrm{model}}}}}$$

so that the computation can be written as a multiplication:

$$\mathrm{position} \times \mathrm{div\_term}.$$

This is numerically stable and computationally efficient.

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, seq_len: int, dropout: float) -> None:
        super().__init__()
        self.d_model = d_model
        self.seq_len = seq_len
        self.dropout = nn.Dropout(dropout)

        # initialize matrix of shape (seq_len, d_model)
        pe = torch.zeros(seq_len, d_model)

        # create position indices
        position = torch.arange(0, seq_len, dtype=torch.float).unsqueeze(1)

        # numerically stable dividing term
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-math.log(10000.0) / d_model)
        )

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        # add batch dimension
        pe = pe.unsqueeze(0)
```

```python
        # register buffer (not updated during backprop)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1), :]
        return self.dropout(x)
```

Step 1: Index Vector The expression

`torch.arange(0, d_model, 2)`

produces the vector

$$[0, 2, 4, 6, \ldots, d_{\text{model}} - 2].$$

This can be written as

$$2i, \quad i = 0, 1, 2, \ldots, \frac{d_{\text{model}}}{2} - 1.$$

Step 2: Multiply by Constant Each element is multiplied by

$$-\frac{\log(10000)}{d_{\text{model}}}.$$

Thus the resulting expression becomes

$$2i \cdot \left(-\frac{\log(10000)}{d_{\text{model}}}\right).$$

Step 3: Apply the Exponential The exponential function is then applied:

$$\text{div\_term}_i = \exp\left(2i \cdot \left(-\frac{\log(10000)}{d_{\text{model}}}\right)\right).$$

Step 4: Simplification Using the identity

$$e^{a \log b} = b^a,$$

we obtain

$$\text{div\_term}_i = 10000^{-\frac{2i}{d_{\text{model}}}} = \frac{1}{10000^{\frac{2i}{d_{\text{model}}}}}.$$

Final Result Therefore,

$$\boxed{\text{div\_term}_i = \frac{1}{10000^{\frac{2i}{d_{\text{model}}}}}}$$

Interpretation The resulting values correspond to geometrically increasing wavelengths:

- Small values of i produce high-frequency oscillations.
- Large values of i produce low-frequency oscillations.
- Each embedding dimension encodes position at a different scale.

Below we explain each line of the implementation in detail.

Initialization

- `super().`$_{init_{()}}$

  Initializes the parent `nn.Module` class so that the module properly registers parameters and buffers.

- `self.d_model = d_model`

  Stores the embedding dimension.

- `self.seq_len = seq_len`

  Stores the maximum sequence length for which positional encodings will be created.

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

  Creates a dropout layer that will be applied after adding positional encodings.

Creating the Positional Encoding Matrix

- `pe = torch.zeros(seq_len, d_model)`

  Creates a tensor of zeros with shape

  $$(seq\_len, d\_model).$$

  Each row will correspond to one position in the sequence.

- `position = torch.arange(0, seq_len).unsqueeze(1)`

  Creates a column vector of shape

  $$(seq\_len, 1),$$

  containing the position indices:

  $$0, 1, 2, \ldots, seq\_len - 1.$$

- ```
  div_term = torch.exp(
      torch.arange(0, d_model, 2).float() *
      (-math.log(10000.0) / d_model)
  )
  ```

  Creates a vector containing the scaling factors for each pair of dimensions.

  - `torch.arange(0, d_model, 2)` selects even indices.
  - The exponential term ensures geometrically increasing wavelengths.
  - The division by d_{model} stabilizes the scaling.

  The resulting shape is:

  $$(d_{\text{model}}/2).$$

Applying Sine and Cosine

- `pe[:, 0::2] = torch.sin(position * div_term)`

  Applies the sine function to all even dimensions.

- `pe[:, 1::2] = torch.cos(position * div_term)`

  Applies the cosine function to all odd dimensions.

After these operations,

$$pe \in \mathbb{R}^{seq\_len \times d_{\text{model}}}.$$

Adding Batch Dimension

- `pe = pe.unsqueeze(0)`

  Adds a batch dimension at the front.

  New shape:

  $$(1, seq\_len, d_{\text{model}}).$$

  This allows broadcasting across batches.

Registering as Buffer

- `self.register_buffer('pe', pe)`

  Registers `pe` as a buffer.

  A buffer:

  - is saved with the model,
  - moves to GPU with the model,
  - is not updated during backpropagation.
  ```

Forward Pass

- `x = x + self.pe[:, :x.size(1), :]`

  Adds positional encodings to the input embeddings.

  - `x` has shape (B, S, d_{model}).
  - `self.pe` has shape $(1, seq\_len, d_{\text{model}})$.
  - Broadcasting expands the first dimension automatically.

- `return self.dropout(x)`

  Applies dropout and returns the result.

3 Layer Norm

```
class LayerNormalization(nn.Module):
    def __init__(self, parameters_shape, eps=1e-5):
        super().__init__()
        self.parameters_shape = parameters_shape
        self.eps = eps
        self.gamma = nn.Parameter(torch.ones(parameters_shape))
        self.beta = nn.Parameter(torch.zeros(parameters_shape))

    def forward(self, x):
        dims = [-(i+1) for i in range(len(self.parameters_shape))]

        mean = x.mean(dim=dims, keepdim=True)
        print(f"Mean \n ({mean.size()}) : \n{mean}")

        var = ((x - mean) ** 2).mean(dim=dims, keepdim=True)
        std = (var + self.eps).sqrt()
        print(f"Standard Deviation \n ({std.size()}) : \n{std}")

        y = (x - mean) / std
        print(f"y \n ({y.size()})= \n {y}")

        out = self.gamma * y + self.beta
        return out
```

Layer Normalization normalizes across the feature dimension for each individual sample. Let

$$x \in \mathbb{R}^{B \times S \times d_{\text{model}}},$$

where:

- B is the batch size,
- S is the sequence length,
- d_{model} is the embedding dimension.

Layer Normalization is applied independently at each position (b, s) across the feature dimension.

Step 1: Compute Mean

For each token vector $x_{b,s,:} \in \mathbb{R}^{d_{\text{model}}}$:

$$\mu_{b,s} = \frac{1}{d_{\text{model}}} \sum_{j=1}^{d_{\text{model}}} x_{b,s,j}.$$

Step 2: Compute Variance

$$\sigma_{b,s}^2 = \frac{1}{d_{\text{model}}} \sum_{j=1}^{d_{\text{model}}} \left(x_{b,s,j} - \mu_{b,s}\right)^2.$$

Step 3: Normalize

$$\hat{x}_{b,s,j} = \frac{x_{b,s,j} - \mu_{b,s}}{\sqrt{\sigma_{b,s}^2 + \varepsilon}}.$$

Step 4: Scale and Shift

$$\boxed{y_{b,s,j} = \gamma_j \hat{x}_{b,s,j} + \beta_j}$$

Thus, the compact form of LayerNorm is:

$$\boxed{\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} + \beta}$$

where:

- $\gamma \in \mathbb{R}^{d_{\text{model}}}$ is a learnable scaling parameter,
- $\beta \in \mathbb{R}^{d_{\text{model}}}$ is a learnable shift parameter,
- ε is a small constant for numerical stability.

1. The ε Term

$$\varepsilon > 0$$

is added inside the square root to prevent division by zero and ensure numerical stability when the variance is very small.

2. The Scale Parameter γ

After normalization:

$$\mathbb{E}[\hat{x}] = 0, \quad \mathrm{Var}[\hat{x}] = 1.$$

The parameter γ allows the model to rescale features, restoring flexibility and allowing different dimensions to have different learned magnitudes.

3. The Shift Parameter β

Normalization forces zero mean. The parameter β allows the model to shift the features to a non-zero mean if beneficial.

4 Multi-head Attention

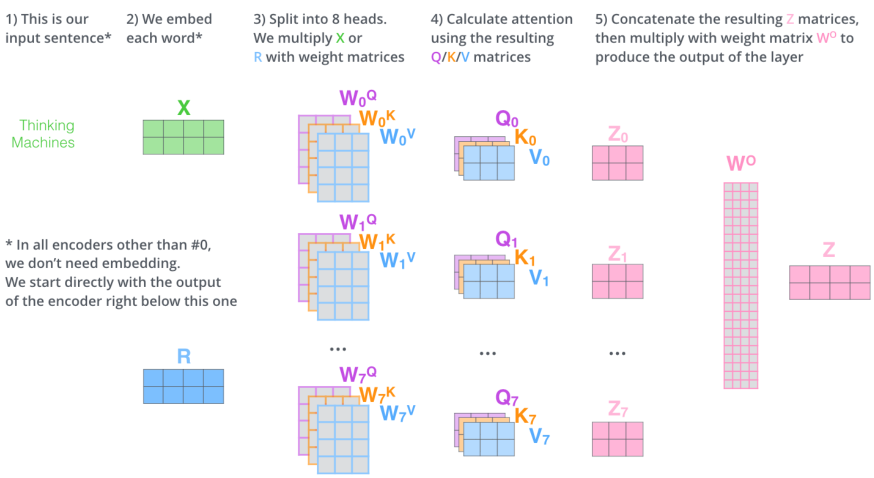

Figure 2: Multihead Self Attention

```
class MultiheadAttention(nn.Module):

    def __init__(self, d_model, n_heads, dropout) -> None:
        super().__init__()
```

```python
        self.d_model = d_model
        self.n_heads = n_heads

        assert d_model % n_heads == 0, \
            "Model dimension is not divisible by number of heads"

        self.d_k = d_model // n_heads

        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_o = nn.Linear(d_model, d_model)

        self.dropout = nn.Dropout(dropout)

    @staticmethod
    def attention(query, key, value, mask, dropout: nn.Dropout):
        d_k = query.shape[-1]

        # Attention scores
        attention_scores = (query @ key.transpose(-2, -1)) \
                           / math.sqrt(d_k)

        if mask is not None:
            attention_scores.masked_fill_(mask == 0, -1e11)

        attention_scores = attention_scores.softmax(dim=-1)

        if dropout is not None:
            attention_scores = dropout(attention_scores)

        qkv = attention_scores @ value

        return qkv, attention_scores

    def forward(self, q, k, v, mask):

        # Linear projections
        query = self.w_q(q)
        key   = self.w_k(k)
        value = self.w_v(v)

        # Reshape for multi-head attention
```

```python
# (B, S, d_model) ->
# (B, S, n_heads, d_k) ->
# (B, n_heads, S, d_k)

query = query.view(query.shape[0], query.shape[1],
                   self.n_heads, self.d_k).transpose(1, 2)

key = key.view(key.shape[0], key.shape[1],
               self.n_heads, self.d_k).transpose(1, 2)

value = value.view(value.shape[0], value.shape[1],
                   self.n_heads, self.d_k).transpose(1, 2)

x, self.attention_scores = self.attention(
    query, key, value, mask, self.dropout
)

# Combine heads
# (B, n_heads, S, d_k) ->
# (B, S, n_heads, d_k) ->
# (B, S, d_model)

x = x.transpose(1, 2).contiguous().view(
    x.shape[0], -1, self.n_heads * self.d_k
)

return self.w_o(x)
```

After the linear projections, the tensors `query`, `key`, and `value` have shape

$$(B, S, d_{\text{model}})$$

where

- B is the batch size,
- S is the sequence length,
- d_{model} is the embedding dimension.

Goal

Multi-head attention splits the embedding dimension into multiple heads:

$$d_{\text{model}} = n_{\text{heads}} \cdot d_k.$$

Each head operates on a subspace of dimension d_k.

Step 1: Reshape

The instruction

```
query.view(B, S, n_heads, d_k)
```

reshapes the tensor from

$$(B, S, d_{\text{model}})$$

to

$$(B, S, n_{\text{heads}}, d_k).$$

This operation does not change the data values. It only reorganizes the embedding dimension into n_{heads} smaller vectors of size d_k.

Step 2: Transpose

The instruction

```
.transpose(1, 2)
```

swaps dimensions 1 and 2, transforming

$$(B, S, n_{\text{heads}}, d_k)$$

into

$$(B, n_{\text{heads}}, S, d_k)$$

Why This Is Necessary

This rearrangement allows attention to be computed independently for each head.

The attention score computation becomes

$$QK^\top$$

with shapes:

$$(B, n_{\text{heads}}, S, d_k) \quad \times \quad (B, n_{\text{heads}}, d_k, S)$$

which produces

$$(B, n_{\text{heads}}, S, S).$$

Thus:

- Each head attends independently.
- Matrix multiplication is efficiently batched.
- Different heads can learn different attention patterns.

5 Position-wise Feed-Forward Network

Implementation

```
class FeedForwardBlock(nn.Module):

    def __init__(self, d_model: int, d_ff: int, dropout: float) -> None:
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        return self.linear2(
            self.dropout(
                torch.relu(
                    self.linear1(x)
                )
            )
        )
```

Mathematical Formulation

Let

$$x \in \mathbb{R}^{B \times S \times d_{\text{model}}}.$$

The feed-forward network is applied independently at each position:

$$\text{FFN}(x) = W_2\,\sigma(W_1 x + b_1) + b_2,$$

where:

- $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{ff}}$,
- $W_2 \in \mathbb{R}^{d_{ff} \times d_{\text{model}}}$,
- $\sigma(\cdot)$ is the ReLU activation function.

Dropout is applied after the activation.

Difference Between `linear1` and `linear2`

1. First Linear Layer (`linear1`)

$$\texttt{linear1}: \quad \mathbb{R}^{d_{\text{model}}} \to \mathbb{R}^{d_{ff}}$$

This layer:

- Expands the dimensionality.
- Projects each token embedding into a higher-dimensional space.
- Increases model capacity.

Typically:

$$d_{ff} = 4 \times d_{\text{model}}.$$

This is sometimes called the **expansion layer**.

2. Second Linear Layer (`linear2`)

$$\texttt{linear2}: \quad \mathbb{R}^{d_{ff}} \to \mathbb{R}^{d_{\text{model}}}$$

This layer:

- Reduces dimensionality back to d_{model}.
- Projects back to the original embedding space.
- Ensures compatibility with residual connections.

This is sometimes called the **projection layer**.

Intuition

The feed-forward block performs:

$$\textbf{Expand} \to \textbf{Non-linearity} \to \textbf{Compress}.$$

It allows the Transformer to:

- Learn complex feature transformations.
- Increase expressiveness without increasing sequence interaction.
- Process each token independently (position-wise).

Unlike attention, the feed-forward network does not mix information across sequence positions.

6 Residual Connection with Pre-Layer Normalization

Implementation

```
class ResidualConnection(nn.Module):
    def __init__(self, features, dropout) -> None:
        super().__init__()
        self.features = features
        self.dropout = nn.Dropout(dropout)
        self.norm = LayerNormalization(features)

    def forward(self, x, sublayer):
        return x + self.dropout(sublayer(self.norm(x)))
```

Mathematical Formulation

Let

$$x \in \mathbb{R}^{B \times S \times d_{\text{model}}}.$$

This block computes:

$$\boxed{\text{Output}(x) = x + \text{Dropout}\big(\text{Sublayer}(\text{LayerNorm}(x))\big)}$$

Step-by-Step Explanation

1. Layer Normalization (Pre-Norm)

$$\tilde{x} = \text{LayerNorm}(x)$$

The input is first normalized across the feature dimension. This stabilizes training and improves gradient flow.

2. Sublayer Transformation

$$z = \text{Sublayer}(\tilde{x})$$

The sublayer may be:

- Multi-Head Attention, or
- Feed-Forward Network.

3. Dropout

$$z' = \text{Dropout}(z)$$

Dropout is applied for regularization.

4. Residual Addition

$$y = x + z'$$

The original input is added back to the transformed output.

Why Use Residual Connections?

Residual connections:

- Improve gradient flow in deep networks.
- Prevent vanishing gradients.
- Allow the model to learn identity mappings easily.
- Make very deep Transformers trainable.

Pre-Norm vs Post-Norm

This implementation uses **Pre-Norm**:

$$x + \text{Sublayer}(\text{LayerNorm}(x))$$

The original Transformer paper used **Post-Norm**:

$$\text{LayerNorm}(x + \text{Sublayer}(x)).$$

Modern Transformers typically use Pre-Norm because it leads to more stable training.

7 Encoder Block

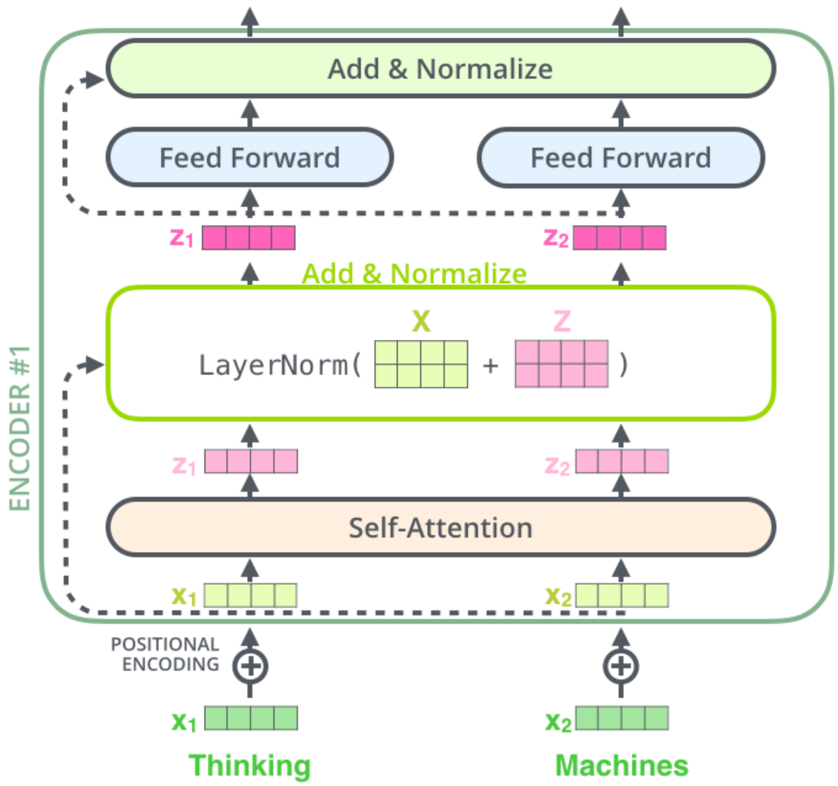

Figure 3: Encoder Block

Implementation

```python
class EncoderBlock(nn.Module):
    def __init__(self, self_attention_block: MultiheadAttention,
                 feed_forward_block: FeedForwardBlock,
                 features: int,
                 dropout: float) -> None:
        super().__init__()
        self.self_attention_block = self_attention_block
        self.feed_forward_block = feed_forward_block

        # Two residual connections:
        # one for attention, one for feed-forward
        self.residual_connections = nn.ModuleList(
            [ResidualConnection(features, dropout) for _ in range(2)]
        )

    def forward(self, x, src_mask):

        # Self-attention + residual
        x = self.residual_connections[0](
            x,
            lambda x: self.self_attention_block(x, x, x, src_mask)
        )

        # Feed-forward + residual
        x = self.residual_connections[1](
            x,
            self.feed_forward_block
        )

        return x
```

Mathematical Formulation

Let

$$x \in \mathbb{R}^{B \times S \times d_{\text{model}}}.$$

An encoder block consists of two sublayers:

1. Multi-head self-attention
2. Position-wise feed-forward network

Step 1: Self-Attention with Residual (Pre-Norm)

$$x^{(1)} = x + \text{Dropout}\left(\text{SelfAttention}(\text{LayerNorm}(x))\right)$$

Since this is self-attention:

$$Q = K = V = x.$$

Step 2: Feed-Forward with Residual (Pre-Norm)

$$x^{(2)} = x^{(1)} + \text{Dropout}\left(\text{FFN}(\text{LayerNorm}(x^{(1)}))\right)$$

Final Output

$$\boxed{\text{EncoderBlock}(x) = x^{(2)}}$$

Key Properties

- Shape is preserved:
$$(B, S, d_{\text{model}}) \to (B, S, d_{\text{model}})$$
- Attention mixes information across sequence positions.
- The feed-forward network processes each position independently.
- Residual connections improve gradient flow and stabilize deep training.

8 Transformer Encoder

Implementation

```
class Encoder(nn.Module):
    def __init__(self, features: int, layers: nn.ModuleList):
        super().__init__()
        self.layers = layers
        self.features = features
        self.norm = LayerNormalization(features)

    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

Mathematical Formulation

Let

$$x^{(0)} \in \mathbb{R}^{B \times S \times d_{\text{model}}}$$

be the embedded input sequence.

Assume the encoder contains N stacked encoder blocks:

$$\{\text{EncoderBlock}_1, \ldots, \text{EncoderBlock}_N\}.$$

The encoder applies them sequentially:

$$x^{(1)} = \text{EncoderBlock}_1(x^{(0)}, \text{mask})$$

$$x^{(2)} = \text{EncoderBlock}_2(x^{(1)}, \text{mask})$$

$$\vdots$$

$$x^{(N)} = \text{EncoderBlock}_N(x^{(N-1)}, \text{mask})$$

After the final block, a Layer Normalization is applied:

$$\boxed{\text{Encoder}(x^{(0)}) = \text{LayerNorm}(x^{(N)})}$$

Explanation

1. Stacked Structure The loop

```
for layer in self.layers:
    x = layer(x, mask)
```

applies N encoder blocks sequentially. Each block refines the representation of the sequence.

2. Shape Preservation Throughout the encoder:

$$(B, S, d_{\text{model}}) \quad \longrightarrow \quad (B, S, d_{\text{model}})$$

The dimensionality remains constant. Only the representation is transformed.

3. Final Layer Normalization The final normalization:

$$\text{LayerNorm}(x^{(N)})$$

stabilizes the output before it is passed to the decoder or to subsequent components of the Transformer.

9 Decoder Block

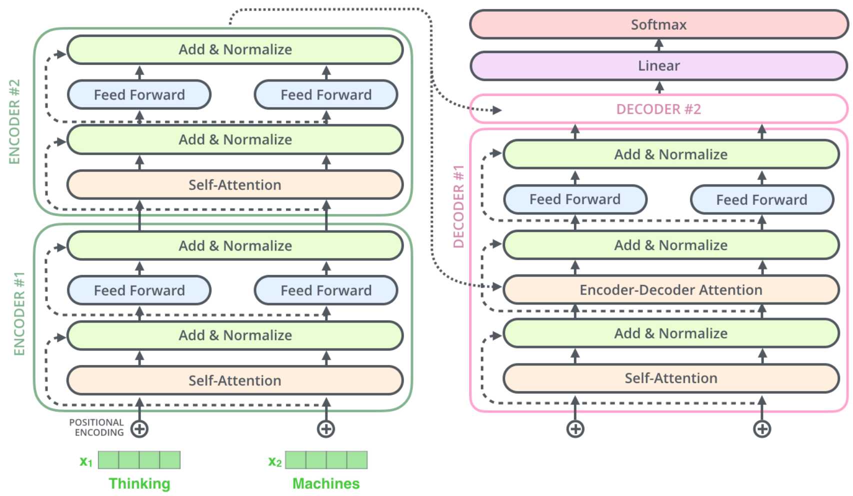

Figure 4: Encoder-Decoder

Implementation

```
class DecoderBlock(nn.Module):

    def __init__(self,
                 features: int,
                 self_attention_block: MultiheadAttention,
                 cross_attention_block: MultiheadAttention,
                 feed_forward_block: FeedForwardBlock,
                 dropout: float) -> None:

        super().__init__()
        self.self_attention_block = self_attention_block
        self.cross_attention_block = cross_attention_block
        self.feed_forward_block = feed_forward_block

        # Three residual connections:
        # 1) masked self-attention
        # 2) cross-attention
        # 3) feed-forward
        self.residue_connections = nn.ModuleList(
            [ResidualConnection(features, dropout) for _ in range(3)]
        )
```

```python
    def forward(self, x, encoder_output, src_mask, tgt_mask):

        # Masked self-attention
        x = self.residue_connections[0](
            x,
            lambda x: self.self_attention_block(x, x, x, tgt_mask)
        )

        # Cross-attention (encoder-decoder attention)
        x = self.residue_connections[1](
            x,
            lambda x: self.cross_attention_block(
                x, encoder_output, encoder_output, src_mask
            )
        )

        # Feed-forward
        x = self.residue_connections[2](
            x,
            self.feed_forward_block
        )

        return x
```

10 Transformer Decoder

Implementation

```python
class Decoder(nn.Module):
    def __init__(self, layers: nn.ModuleList, features: int):
        super().__init__()
        self.layers = layers
        self.norm = LayerNormalization(features)

    def forward(self, x, encoder_output, tgt_mask, src_mask):
        for layer in self.layers:
            x = layer(x, encoder_output, src_mask, tgt_mask)
        return self.norm(x)
```

Explanation of the Decoder

The Decoder is composed of N stacked DecoderBlocks followed by a final Layer Normalization.

Let

$$x^{(0)} \in \mathbb{R}^{B\times S_{\text{tgt}}\times d_{\text{model}}}$$

be the target input embeddings, and let

$$H^{enc} \in \mathbb{R}^{B\times S_{\text{src}}\times d_{\text{model}}}$$

be the encoder output.

Stacked Decoder Blocks

The loop

```
for layer in self.layers:
    x = layer(x, encoder_output, src_mask, tgt_mask)
```

applies each DecoderBlock sequentially:

$$x^{(1)} = \text{DecoderBlock}_1(x^{(0)}, H^{enc})$$

$$x^{(2)} = \text{DecoderBlock}_2(x^{(1)}, H^{enc})$$

$$\vdots$$

$$x^{(N)} = \text{DecoderBlock}_N(x^{(N-1)}, H^{enc})$$

Each DecoderBlock contains:

- Masked self-attention (prevents attending to future tokens),
- Cross-attention (queries the encoder output),
- Feed-forward network,
- Residual connections and Layer Normalization.

Final Layer Normalization

After the final block, a LayerNorm is applied:

$$\boxed{\text{Decoder}(x^{(0)}) = \text{LayerNorm}(x^{(N)})}$$

This normalization stabilizes the final representation before it is passed to the output projection layer.

Shape Preservation

Throughout the decoder, the tensor shape remains constant:

$$(B, S_{\text{tgt}}, d_{\text{model}}) \longrightarrow (B, S_{\text{tgt}}, d_{\text{model}})$$

Only the internal representation is refined at each layer.

Role of Masks

- The **target mask** ensures autoregressive behavior by preventing attention to future tokens.
- The **source mask** ensures padding tokens in the encoder output are ignored during cross-attention.

11 Projection Layer

Implementation

```
class ProjectionLayer(nn.Module):

    def __init__(self, d_model, vocab_size):
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        self.proj = nn.Linear(self.d_model, self.vocab_size)

    def forward(self, x):
        # (batch, seq_len, d_model) --> (batch, seq_len, vocab_size)
        return self.proj(x)
```

12 Full Transformer Architecture

Implementation

```
class Transformer(nn.Module):

    def __init__(self,
                 encoder: Encoder,
                 decoder: Decoder,
                 src_embed: InputEmbeddings,
                 tgt_embed: InputEmbeddings,
                 src_pos: PositionalEncoding,
```

```python
                 tgt_pos: PositionalEncoding,
                 projection_layer: ProjectionLayer) -> None:

        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.src_pos = src_pos
        self.tgt_pos = tgt_pos
        self.projection_layer = projection_layer

    def encode(self, src, src_mask):
        src = self.src_embed(src)
        src = self.src_pos(src)
        return self.encoder(src, src_mask)

    def decode(self, tgt, encoder_output, src_mask, tgt_mask):
        tgt = self.tgt_embed(tgt)
        tgt = self.tgt_pos(tgt)
        return self.decoder(tgt, encoder_output, src_mask, tgt_mask)

    def project(self, x):
        return self.projection_layer(x)
```

The Transformer model is composed of three main components:

1. Source embedding and encoder
2. Target embedding and decoder
3. Final linear projection to vocabulary space

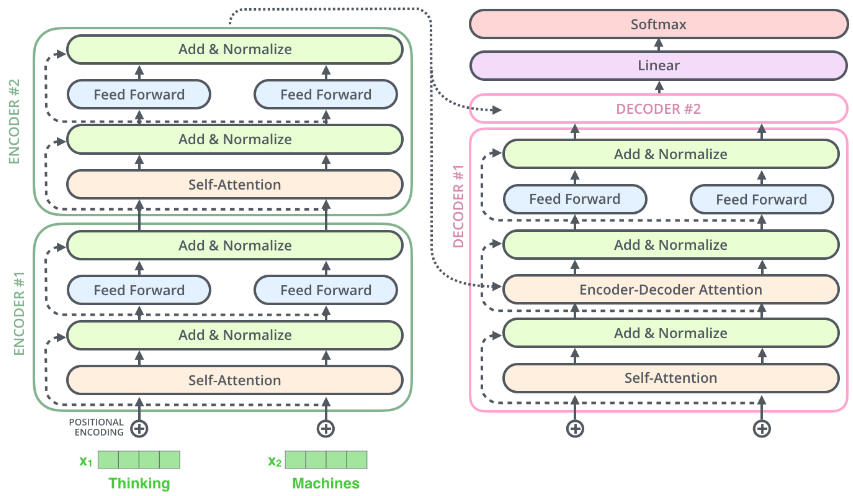

Figure 5: Encoder-Decoder

1. Source Encoding

Let the source input be

$$\text{src} \in \mathbb{R}^{B \times S_{\text{src}}},$$

where:

- B is the batch size,
- S_{src} is the source sequence length.

Embedding Each token index is mapped to a vector of dimension d_{model}:

$$E_{\text{src}} = \text{Embed}_{\text{src}}(\text{src}) \in \mathbb{R}^{B \times S_{\text{src}} \times d_{\text{model}}}.$$

Positional Encoding Positional information is added:

$$\tilde{E}_{\text{src}} = E_{\text{src}} + PE_{\text{src}}.$$

Encoder The encoder transforms the representation:

$$H^{enc} = \text{Encoder}(\tilde{E}_{\text{src}}) \in \mathbb{R}^{B \times S_{\text{src}} \times d_{\text{model}}}.$$

The encoder output contains contextualized representations of the source sequence.

2. Target Decoding

Let the target input be

$$\text{tgt} \in \mathbb{R}^{B \times S_{\text{tgt}}}.$$

Target Embedding

$$E_{\text{tgt}} = \text{Embed}_{\text{tgt}}(\text{tgt}) \in \mathbb{R}^{B \times S_{\text{tgt}} \times d_{\text{model}}}.$$

Add Positional Encoding

$$\tilde{E}_{\text{tgt}} = E_{\text{tgt}} + PE_{\text{tgt}}.$$

Decoder The decoder uses both the target representation and the encoder output:

$$H^{dec} = \text{Decoder}(\tilde{E}_{\text{tgt}}, H^{enc}).$$

The decoder applies:

- Masked self-attention,
- Cross-attention with encoder outputs,
- Feed-forward transformations.

The resulting tensor has shape

$$H^{dec} \in \mathbb{R}^{B \times S_{\text{tgt}} \times d_{\text{model}}}.$$

3. Projection to Vocabulary

The final step maps the decoder output to vocabulary logits:

$$\boxed{\text{logits} = H^{dec} W + b}$$

where

$$W \in \mathbb{R}^{d_{\text{model}} \times V}, \quad b \in \mathbb{R}^{V},$$

and V is the vocabulary size.
The final output shape is

$$(B, S_{\text{tgt}}, V).$$

Applying softmax gives probabilities over the vocabulary:

$$\text{Softmax}(\text{logits}).$$

Overall Transformer Function

The entire Transformer can be written compactly as:

$$\boxed{\text{Transformer}(\text{src}, \text{tgt}) = \text{Projection}\Big(\text{Decoder}(\text{Encoder}(\text{src}), \text{tgt})\Big)}$$

Key Properties

- The encoder processes the source sequence.
- The decoder generates the target sequence autoregressively.
- Attention enables global interaction across tokens.
- Dimensionality d_{model} remains constant throughout the model.

13 build_transformer

Implementation

```python
def build_transformer(src_vocab_size: int,
                      tgt_vocab_size: int,
                      src_seq_len: int,
                      tgt_seq_len: int,
                      d_model: int = 512,
                      N: int = 6,
                      h: int = 8,
                      dropout: float = 0.1,
                      d_ff: int = 2048):

    # Create the embedding layers
    src_embed = InputEmbeddings(d_model, src_vocab_size)
    tgt_embed = InputEmbeddings(d_model, tgt_vocab_size)

    # Create the positional encoding layers
    src_pos = PositionalEncoding(d_model, src_seq_len, dropout)
    tgt_pos = PositionalEncoding(d_model, tgt_seq_len, dropout)

    # Create the encoder blocks
    encoder_blocks = []
    for _ in range(N):
        encoder_self_attention_block = MultiheadAttention(d_model, h, dropout)
        feed_forward_block = FeedForwardBlock(d_model, d_ff, dropout)
        encoder_block = EncoderBlock(encoder_self_attention_block,
```

```python
                                       feed_forward_block,
                                       d_model,
                                       dropout)
        encoder_blocks.append(encoder_block)

    # Create the decoder blocks
    decoder_blocks = []
    for _ in range(N):
        decoder_self_attention_block = MultiheadAttention(d_model, h, dropout)
        decoder_cross_attention_block = MultiheadAttention(d_model, h, dropout)
        feed_forward_block = FeedForwardBlock(d_model, d_ff, dropout)
        decoder_block = DecoderBlock(d_model,
                                     decoder_self_attention_block,
                                     decoder_cross_attention_block,
                                     feed_forward_block,
                                     dropout)
        decoder_blocks.append(decoder_block)

    # Create the encoder and decoder
    encoder = Encoder(d_model, nn.ModuleList(encoder_blocks))
    decoder = Decoder(nn.ModuleList(decoder_blocks), d_model)

    # Create the projection layer
    projection_layer = ProjectionLayer(d_model, tgt_vocab_size)

    # Create the transformer
    transformer = Transformer(encoder, decoder,
                              src_embed, tgt_embed,
                              src_pos, tgt_pos,
                              projection_layer)

    # Initialize the parameters (Xavier / Glorot for matrices)
    for p in transformer.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return transformer
```

What this function constructs

- **Input embeddings:** $\mathrm{Embed}_{\mathrm{src}}, \mathrm{Embed}_{\mathrm{tgt}}$ of shape mapping token indices $\to \mathbb{R}^{d_{\mathrm{model}}}$.
- **Positional encodings:** fixed sinusoidal positional layers for source/target with maximum lengths `src_seq_len`, `tgt_seq_len`.

- **Encoder stack:** N copies of EncoderBlock. Each block contains self-attention and position-wise feed-forward sublayers (with residual + LayerNorm).
- **Decoder stack:** N copies of DecoderBlock. Each block contains masked self-attention, cross-attention and feed-forward (with residual + LayerNorm).
- **Projection:** linear layer $W \in \mathbb{R}^{d_{\text{model}} \times V_{\text{tgt}}}$ mapping decoder outputs to vocabulary logits.
- **Parameter initialization:** Xavier (Glorot) uniform initialization is applied to all parameters with dim > 1 (i.e., weight matrices).

Shapes / notation

- Input tokens: src $\in \mathbb{Z}^{B \times S_{\text{src}}}$, tgt $\in \mathbb{Z}^{B \times S_{\text{tgt}}}$.
- After embedding: $\in \mathbb{R}^{B \times S \times d_{\text{model}}}$.
- Encoder / decoder keep the hidden shape (B, S, d_{model}).
- Final logits: $(B, S_{\text{tgt}}, V_{\text{tgt}})$.