KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY
AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE
Narayanaguda, Hyderabad.

Embedded Learning

TRANSFORMERS

Exercise

06-02-2025

By

Asha.M

Assistant professor

CSE(AI&ML)

KMIT

Neural Machine Translation Using Transformer

Sample Sentence: "I love AI"

- **Input Sentence (English)**: "I love AI"
- **Target Sentence (Hindi)**: "मुझे एआई पसंद है"

🚀 Step 1: Tokenization & Word Embeddings

1. **Tokenization**: The English sentence **"I love AI"** is tokenized, and the words are mapped to token IDs. The Hindi translation **"मुझे एआई पसंद है"** is also tokenized.

English Word	Token ID (English)	Hindi Word	Token ID (Hindi)
I	12	मुझे	33
love	25	एआई	67
AI	52	पसंद	45
<EOS>	99	है	98

2. **Word Embeddings**: Each word is transformed into a 4D embedding vector. We assume that each word has a corresponding vector of size 4 (for simplicity).

For example:

- "I" -> $[0.1, 0.5, 0.3, 0.7]$
- "love" -> $[0.2, 0.6, 0.4, 0.8]$
- "AI" -> $[0.3, 0.7, 0.5, 0.9]$
- "मुझे" -> $[0.2, 0.6, 0.5, 0.9]$
- "एआई" -> $[0.3, 0.8, 0.6, 1.0]$
- "पसंद" -> $[0.4, 0.9, 0.7, 1.1]$
- "है" -> $[0.1, 0.5, 0.3, 0.8]$

🚀 Step 2: Positional Encoding

Positional encodings are added to the embeddings to inject information about the position of each word in the sentence.

For simplicity, let's assume positional encodings as follows (normally, it's calculated using sinusoidal functions):

- **Positional Encodings**:
  - "I" (position 1) -> $[0.01, 0.02, 0.03, 0.04]$
  - "love" (position 2) -> $[0.02, 0.03, 0.04, 0.05]$
  - "AI" (position 3) -> $[0.03, 0.04, 0.05, 0.06]$

Adding these positional encodings to the embeddings gives us the final input embeddings:

- Final embeddings after adding positional encoding:
  - "I" = $[0.1 + 0.01, 0.5 + 0.02, 0.3 + 0.03, 0.7 + 0.04] = [0.11, 0.52, 0.33, 0.74]$
  - "love" = $[0.2 + 0.02, 0.6 + 0.03, 0.4 + 0.04, 0.8 + 0.05] = [0.22, 0.63, 0.44, 0.85]$
  - "AI" = $[0.3 + 0.03, 0.7 + 0.04, 0.5 + 0.05, 0.9 + 0.06] = [0.33, 0.74, 0.55, 0.96]$

🚀 Step 3: Self-Attention (Encoder) — Scaled Dot-Product Attention

We now perform the **Scaled Dot-Product Attention** to compute the attention scores. Let's go through it step by step.

1. **Linear Projections** (we use the embeddings from Step 2):
   - We use weight matrices W_Q, W_K, and W_V to project the input embeddings into **Query (Q)**, **Key (K)**, and **Value (V)** matrices.
   - For simplicity, let's assume we have one attention head and $d_k = 4$.

$$Q = X_{\text{input}} \cdot W_Q, \quad K = X_{\text{input}} \cdot W_K, \quad V = X_{\text{input}} \cdot W_V$$

2. **Compute Attention Scores**: For each word, we compute the attention scores using the scaled dot-product between the **Query** and **Key**:

$$\text{Attention}_{\text{score}} = \frac{Q \cdot K^T}{\sqrt{d_k}}$$

For simplicity, let's assume the attention score for word "I" is computed as follows:

$$\text{Attention}_{\text{score\_I}} = \frac{[0.11, 0.52, 0.33, 0.74] \cdot [0.11, 0.52, 0.33, 0.74]^T}{\sqrt{4}} = \text{score}$$

Repeat this for all words, and compute the attention matrix.

3. **Apply Softmax**: Apply **Softmax** to the attention scores to normalize them.
4. **Compute Weighted Sum**: Multiply the normalized attention scores by the **Value (V)** to get the final attention output.

Multi-Head Attention (MHA) in Step 3:

1. **Linear Projections**: We project the input (or embeddings) into multiple different subspaces using learned weight matrices. We do this for **Query (Q)**, **Key (K)**, and **Value (V)**.

   - $Q_i = X_{final} \cdot W_{Qi}$
   - $K_i = X_{final} \cdot W_{Ki}$
   - $V_i = X_{final} \cdot W_{Vi}$

   Here, we have i heads (let's say 8 for example), so we get 8 different **Q**, **K**, and **V** matrices.

2. **Scaled Dot-Product Attention for each head**: For each attention head, we apply the **Scaled Dot-Product Attention** separately. Each head focuses on different aspects of the input sequence.

$$\text{Attention}_i(Q_i, K_i, V_i) = \text{Softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

   Each attention head produces an output.

3. **Concatenate the attention heads**: After computing attention for all heads, we concatenate the outputs of all heads:

$$\text{MHA Output} = \text{Concat}\left(\text{Attention}_1, \text{Attention}_2, \ldots, \text{Attention}_h\right)$$

Here, **h** represents the number of attention heads.

4. **Final Linear Transformation**: After concatenation, we apply a final linear transformation to the concatenated output:

$$\text{MHA Final Output} = \text{MHA Output} \cdot W_O$$

This result is passed to the subsequent layers of the Encoder.

Example with our sentence:

- Suppose we have 4 attention heads. After obtaining **Q**, **K**, and **V** for each head (through linear projections), we perform **Scaled Dot-Product Attention** for each head, then concatenate all the results together, and apply the final linear transformation.

🚀 Step 4: Feed-Forward Neural Network (Encoder)

After the **Multi-Head Attention** output, we pass it through the **Feed-Forward Neural Network (FFN)**:

1. **First Linear Transformation**:

$$\text{FFN1} = \text{ReLU}(W_1 \cdot \text{Attention Output} + b_1)$$

2. **Second Linear Transformation**:

$$\text{FFN2} = W_2 \cdot \text{FFN1} + b_2$$

The output of the **FFN** is then passed to the next encoder block (or used directly if this is the last Encoder block).

🚀 Step 5: Decoder (Masked Multi-Head Attention)

Now we move to the **Decoder**.

1. **Masked Multi-Head Attention**:
    - For the Decoder, we apply **Masked Multi-Head Attention**, where we prevent the attention mechanism from attending to future tokens in the target sequence.
2. **Generating Q, K, V for Decoder**:
    - The **Query (Q)** is generated from the previous Decoder output.
    - The **Key (K)** and **Value (V)** are generated from the previous output of the Decoder, but we mask future tokens.

For example, if we are predicting the first token "**मुझे**", we use "**I**" (the first token of the input) as the **Query (Q)**, and for **Key (K)** and **Value (V)**, we use the embeddings of "**I**", "**love**", and "**AI**" (from the Encoder).

3. **Scaled Dot-Product Attention for Masked MHA**: Similar to Encoder, we calculate attention scores, but **masking** ensures the Decoder can only attend to previous tokens.

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

🚀 Step 6: Encoder-Decoder Attention (Cross-Attention)

The Decoder now attends to the Encoder's output using **Cross-Attention** (Encoder-Decoder Attention):

1. **Generating Q (Decoder) and K, V (Encoder)**:
   - **Q (Decoder)** is generated from the previous Decoder output.
   - **K (Encoder)** and **V (Encoder)** are generated from the Encoder's output.
2. **Attention Calculation**:

$$\text{Cross-Attention}(Q_{\text{decoder}}, K_{\text{encoder}}, V_{\text{encoder}}) = \text{Softmax}\left(\frac{Q_{\text{decoder}} K_{\text{encoder}}^T}{\sqrt{d_k}}\right) V_{\text{encoder}}$$

   This helps the Decoder focus on relevant parts of the Encoder output while generating the next token.

🚀 Step 7: Feed-Forward Neural Network (Decoder)

After the attention mechanism, the output passes through the **Feed-Forward Network (FFN)**:

1. **First Linear Transformation:**

$$\text{FFN1} = \text{ReLU}(W_1 \cdot \text{Decoder Output} + b_1)$$

2. **Second Linear Transformation:**

$$\text{FFN2} = W_2 \cdot \text{FFN1} + b_2$$

🚀 Step 8: Output Generation

Finally, the output of the Decoder is passed through a **Linear Layer** followed by a **Softmax Layer** to generate a probability distribution over the vocabulary for the next token. The word with the highest probability is chosen.

For example, the output for **"I love AI"** in the Decoder will be **"मुझे"**, the first word in the translation.

```
pip install torchtext
```

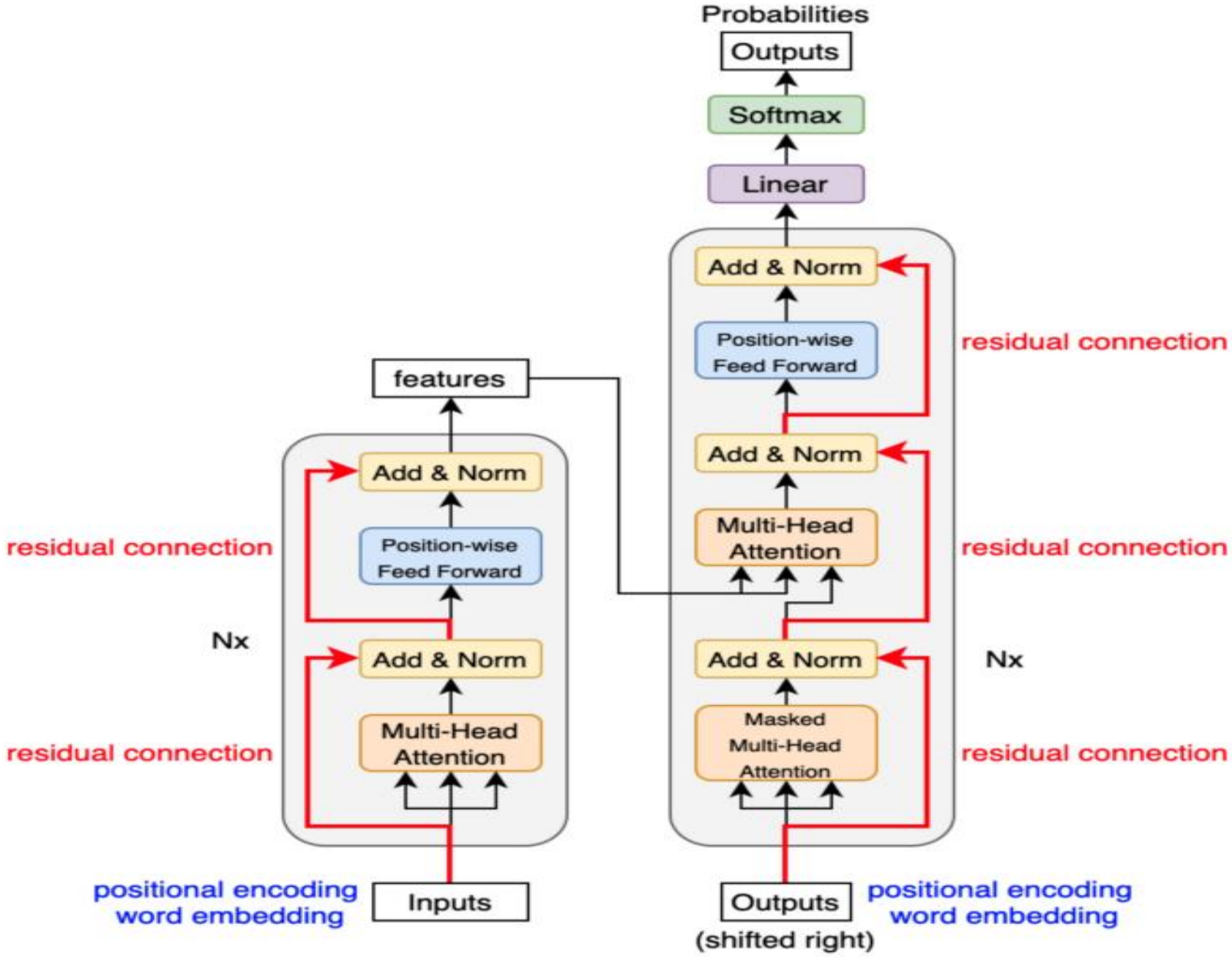
- `torchtext` is a library in the PyTorch ecosystem designed for **natural language processing (NLP)** tasks.
- It provides pre-trained embeddings, datasets, vocabulary processing, and utilities for handling text data.

```
pip install torch==2.0.1 torchtext==0.15.2
```

- This command installs **specific versions** of `torch` (PyTorch) and `torchtext`.
- `torch==2.0.1`: Installs PyTorch version **2.0.1** instead of the latest version.
- `torchtext==0.15.2`: Installs `torchtext` version **0.15.2**, which is compatible with PyTorch 2.0.1.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

- `torch` : The core PyTorch library.
- `torch.nn` : Provides neural network layers.
- `torch.nn.functional` : Provides functions like softmax and relu.


Probabilities
Outputs
Softmax
Linear
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
residual connection
residual connection
residual connection
Nx
Nx
features
Add & Norm
Position-wise Feed Forward
Add & Norm
Multi-Head Attention
residual connection
residual connection
residual connection
positional encoding
word embedding
Inputs
Outputs
(shifted right)
positional encoding
word embedding

Step1: Put together

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

# Step 1: Tokenizer and Vocabulary Creation (Add more words if needed)
sentence_en = "I love AI ."
sentence_fr = "J' adore l'IA ."

word_map_en = {"<pad>": 0, "I": 1, "love": 2, "AI": 3, ".": 4}
word_map_fr = {"<pad>": 0, "J'": 1, "adore": 2, "l'IA": 3, ".": 4}

# Tokenizing sentences
def tokenize(sentence, word_map):
    tokens = [word_map[word] for word in sentence.split()]
    print(f"Tokens for '{sentence}': {tokens}")
    return torch.tensor(tokens)

# Tokenize the input and target sentences
input_tensor = tokenize(sentence_en, word_map_en).unsqueeze(0)  # Shape (1, 4)
target_tensor = tokenize(sentence_fr, word_map_fr).unsqueeze(0)  # Shape (1, 4)
```

```
Tokens for 'I love AI .': [1, 2, 3, 4]
Tokens for 'J' adore l'IA .': [1, 2, 3, 4]
```

```
# Step 1: Tokenizer and Vocabulary Creation (Add more words if needed)
sentence_en = "I love AI ."
sentence_fr = "J' adore l'IA ."

word_map_en = {"<pad>": 0, "I": 1, "love": 2, "AI": 3, ".": 4}
word_map_fr = {"<pad>": 0, "J'": 1, "adore": 2, "l'IA": 3, ".": 4}
```

- `sentence_en` and `sentence_fr` are the English and French sentences you wish to translate.
- `word_map_en` and `word_map_fr` are dictionaries mapping words to integer indices for each vocabulary.
- The key `<pad>` is used to represent padding tokens (not meaningful but necessary to ensure uniform sentence length).

```python
# Tokenizing sentences
def tokenize(sentence, word_map):
    return torch.tensor([word_map[word] for word in sentence.split()])

# Tokenize the input and target sentences
input_tensor = tokenize(sentence_en, word_map_en).unsqueeze(0)  # Shape (1, 6)
target_tensor = tokenize(sentence_fr, word_map_fr).unsqueeze(0)  # Shape (1, 6)
```

- This function takes a sentence and a `word_map` dictionary.
- It splits the sentence into individual words, and then maps each word to its corresponding index using `word_map`.
- The output is a tensor of indices representing the sentence.

2 Positional Encoding

Since Transformers do not have an inherent understanding of word order (unlike RNNs), **positional encodings** are added to the token embeddings to inject sequence information.

- Each position in the sequence gets a unique vector based on sine and cosine functions.
- This helps the model understand **word order and relative positions**.
- The encoded positions are then added to the token embeddings before passing them to the Transformer.

```python
# Step 2: Positional Encoding
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.encoding = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -(torch.log(torch.tensor(10000.0)) / d_model))
        self.encoding[:, 0::2] = torch.sin(position * div_term)
        self.encoding[:, 1::2] = torch.cos(position * div_term)
        self.encoding = self.encoding.unsqueeze(0)

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

- **Positional Encoding** is used because Transformers don't have a notion of sequence order (unlike RNNs/LSTMs). We need to inject positional information into the model.
- `d_model` : The dimension of the model. It determines the size of the vectors representing words.
- `max_len` : Maximum sentence length considered by the model.
- The `encoding` is created using sinusoidal functions (`sin` and `cos`) to represent positional information.
- The `forward()` method adds the positional encoding to the word embeddings to ensure the model is aware of word positions in a sentence.

- **Creates a tensor for positional encodings.**
- `torch.arange(0, max_len)`: Creates positions `[0, 1, 2, ..., max_len-1]`.
- `unsqueeze(1)`: Converts it to a column vector.

- **Computes the denominator for sine and cosine functions.**
- Uses `10000^(2i/d_model)` for scaling.

- **Assigns sine values to even indices and cosine values to odd indices.**
- **Adds batch dimension** (`(1, max_len, d_model)`).
- **Adds positional encoding to input embeddings.**

```
    def forward(self, x):
        pos_enc = self.encoding[:, :x.size(1)]
        print("\nPositional Encoding Values:")
        print(pos_enc)

        final_output = x + pos_enc
        print("\nFinal Input After Adding Positional Encoding:")
        print(final_output)

        return final_output

# Apply positional encoding
pos_encoding_layer = PositionalEncoding(embedding_dim)
encoded_input = pos_encoding_layer(embedded_input)
```

```
Positional Encoding Values:
tensor([[[ 0.0000e+00,  1.0000e+00,  0.0000e+00,  1.0000e+00,  0.0000e+00,
           1.0000e+00,  0.0000e+00,  1.0000e+00],
         [ 8.4147e-01,  5.4030e-01,  9.9833e-02,  9.9500e-01,  9.9998e-03,
           9.9995e-01,  1.0000e-03,  1.0000e+00],
         [ 9.0930e-01, -4.1615e-01,  1.9867e-01,  9.8007e-01,  1.9999e-02,
           9.9980e-01,  2.0000e-03,  1.0000e+00],
         [ 1.4112e-01, -9.8999e-01,  2.9552e-01,  9.5534e-01,  2.9995e-02,
           9.9955e-01,  3.0000e-03,  1.0000e+00]]])

Final Input After Adding Positional Encoding:
tensor([[[ 1.4502,  3.0924, -0.6431,  0.0207,  0.3841,  1.0671, -0.2646,
           1.8030],
         [-0.6485,  0.3933,  0.2490, -0.5742,  0.8068,  3.1474, -0.3838,
           2.2970],
         [ 0.5382, -1.0769,  0.9533,  2.0547, -0.3899,  0.4979, -0.7926,
           0.6992],
         [ 1.4659, -0.3045, -1.0397,  2.7577, -1.9891,  0.4426,  0.1174,
           1.3669]]], grad_fn=<AddBackward0>)
```

3 Multi-Head Attention (MHA)

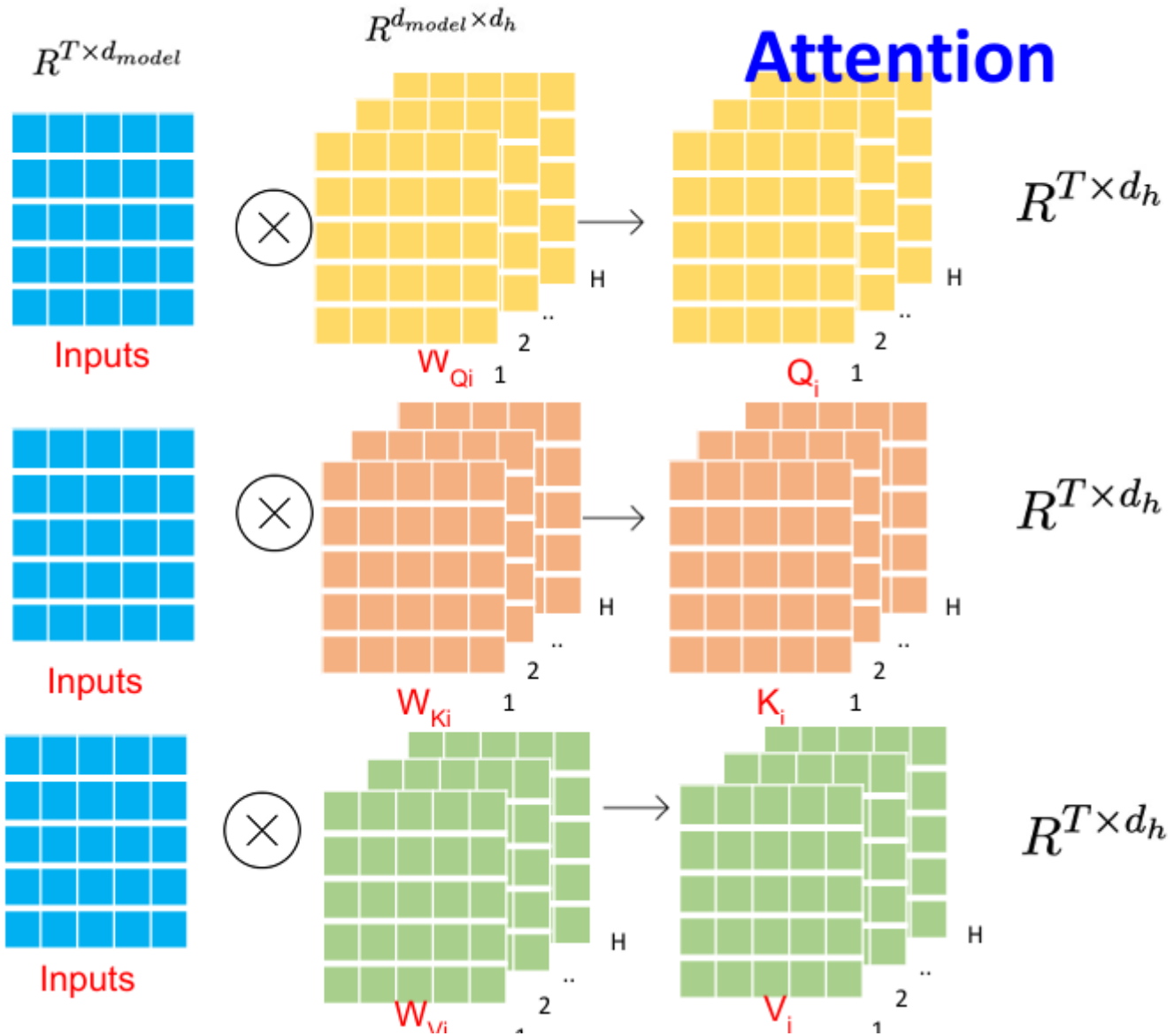
The core mechanism of a Transformer is **Self-Attention**, which allows words to attend to each other in a sentence.

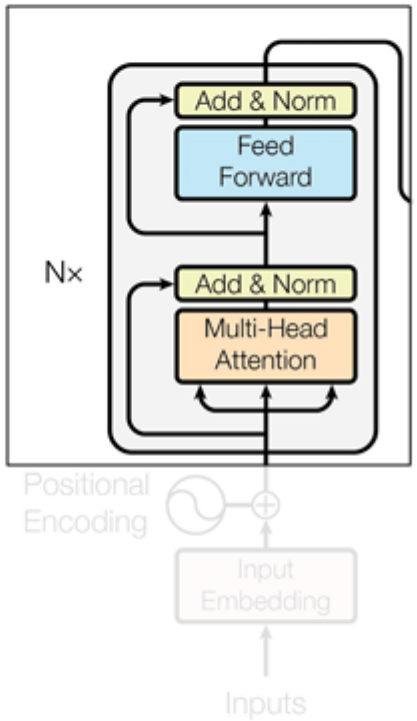
- The input embeddings are transformed into **queries (Q), keys (K), and values (V)** using linear layers.
- Attention scores are computed using **scaled dot-product attention**:

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

- These attention scores determine how much focus a word should have on others in the sentence.
- Since a single attention mechanism may not capture all dependencies, **Multi-Head Attention (MHA)** uses multiple attention heads to learn different contextual relationships.

Multi-Head Attention





```python
# Step 3: Multi-Head Attention
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_model = d_model
        self.d_k = d_model // num_heads
        self.d_v = d_model // num_heads

        self.query = nn.Linear(d_model, d_model)
        self.key = nn.Linear(d_model, d_model)
        self.value = nn.Linear(d_model, d_model)

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

- **Multi-Head Attention** allows the model to focus on different parts of the input sequence at once. Each "head" in the multi-head attention mechanism captures different aspects of the relationships between words in a sentence.
- `d_model` : The size of the model's hidden dimension (i.e., the length of the word vectors).
- `num_heads` : The number of attention heads. More heads allow the model to focus on multiple types of relationships in the input sequence.
- `d_k` and `d_v` : These represent the dimensionality of the query (`q`), key (`k`), and value (`v`) vectors used in attention. They are determined by dividing `d_model` by `num_heads` (hence each head has a smaller focus).
- `query` , `key` , `value` : Linear layers to project the input sequence into query, key, and value vectors.

```
self.query = nn.Linear(d_model, d_model)
self.key = nn.Linear(d_model, d_model)
self.value = nn.Linear(d_model, d_model)
self.fc = nn.Linear(d_model, d_model)
```

- Defines linear layers for query, key, value, and final transformation.

```
print("\nQuery Matrix (Q):")
print(q)
print("\nKey Matrix (K):")
print(k)
print("\nValue Matrix (V):")
print(v)
```

```
Query Matrix (Q):
tensor([[[[-0.2129,  0.6972, -0.4258, -0.6307],
          [ 0.3115,  0.7776, -0.3028, -0.3050],
          [-0.0179,  0.6989, -0.2669, -0.5032],
          [-0.3135,  0.4531, -0.2213, -0.6537]],

         [[ 0.3187,  0.3039,  0.1577, -0.3615],
          [-0.2158, -0.2001,  0.5310, -0.4684],
          [ 0.3159, -0.1083,  0.1931, -0.4048],
          [ 0.0268,  0.1423,  0.3550, -0.1668]]]],
       grad_fn=<TransposeBackward0>)

Key Matrix (K):
tensor([[[[ 0.3806, -0.0871,  0.3698, -0.2368],
          [ 0.5779, -0.1423,  0.1477, -0.2772],
          [ 0.3865, -0.0633,  0.0547, -0.3224],
          [ 0.5260, -0.1161,  0.2348, -0.1759]],

         [[-0.4944,  0.1099,  0.2123,  0.1405],
          [-0.0885,  0.2254, -0.1298,  0.0543],
          [-0.3233,  0.1415,  0.0312,  0.0132],
          [-0.2021,  0.0744,  0.3165,  0.0195]]]],
       grad_fn=<TransposeBackward0>)

Value Matrix (V):
tensor([[[[-0.3256,  0.2563, -0.1962, -0.3466],
          [-0.2972,  0.0439,  0.0220, -0.2928],
          [-0.2679, -0.0642,  0.0275, -0.5175],
          [-0.1620,  0.5927,  0.0147, -0.1755]],

         [[ 0.0671, -0.4492,  0.1692, -0.0067],
          [-0.1241, -0.1383,  0.2545, -0.2605],
          [-0.0519, -0.1548,  0.1850, -0.1169],
          [-0.0812, -0.4221, -0.2275,  0.1615]]]],
       grad_fn=<TransposeBackward0>)
```

```
def forward(self, x, mask=None):
    batch_size, seq_len, _ = x.size()

    q = self.query(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
    k = self.key(x).view(batch_size, seq_len, self.num_heads, self.d_k).transpose(1, 2)
    v = self.value(x).view(batch_size, seq_len, self.num_heads, self.d_v).transpose(1, 2)

    attn_scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.d_k, dtype=torch.float))
    if mask is not None:
        attn_scores = attn_scores.masked_fill(mask == 0, float('-inf'))  # Ensure mask matches input length
    attn_weights = F.softmax(attn_scores, dim=-1)

    attention_output = torch.matmul(attn_weights, v).transpose(1, 2).contiguous().view(batch_size, seq_len, self.d_model)
    output = self.fc(attention_output)

    return output
```

```
# Compute attention scores
attn_scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.d_k, dtype=torch.float))
print("\nRaw Attention Scores (before softmax):")
print(attn_scores)

if mask is not None:
    attn_scores = attn_scores.masked_fill(mask == 0, float('-inf'))  # Ensure mask matches input length

attn_weights = F.softmax(attn_scores, dim=-1)
print("\nAttention Weights (after softmax):")
print(attn_weights)
```

```
Raw Attention Scores (before softmax):
tensor([[[[-7.4902e-02, -5.5138e-02,  2.6818e-02, -9.0990e-02],
          [ 5.5637e-03,  5.4603e-02,  7.6483e-02,  2.8056e-02],
          [-2.3596e-02, -4.8658e-03,  4.8232e-02, -3.2368e-02],
          [-4.2889e-02, -4.8555e-02,  2.4398e-02, -7.7247e-02]],

         [[-7.0743e-02,  9.9435e-05, -2.9940e-02,  5.2107e-04],
          [ 6.5812e-02, -6.0178e-02,  2.5933e-02,  9.3839e-02],
          [-9.1974e-02, -4.9695e-02, -5.8375e-02, -9.3433e-03],
          [ 2.7167e-02, -1.2719e-02,  1.0177e-02,  5.7145e-02]]]],
       grad_fn=<DivBackward0>)

Attention Weights (after softmax):
tensor([[[[0.2432, 0.2481, 0.2693, 0.2394],
          [0.2412, 0.2533, 0.2589, 0.2467],
          [0.2448, 0.2494, 0.2631, 0.2427],
          [0.2481, 0.2467, 0.2654, 0.2397]],

         [[0.2387, 0.2563, 0.2487, 0.2564],
          [0.2583, 0.2278, 0.2482, 0.2657],
          [0.2402, 0.2506, 0.2484, 0.2609],
          [0.2516, 0.2418, 0.2474, 0.2593]]]], grad_fn=<SoftmaxBackward0>)
```

```
# Apply attention to values
attention_output = torch.matmul(attn_weights, v).transpose(1, 2).contiguous().view(batch_size, seq_len, self.d_model)
print("\nAttention Output (before final FC layer):")
print(attention_output)

output = self.fc(attention_output)
print("\nFinal Output (after FC layer):")
print(output)

return output
```

```
Attention Output (before final FC layer):
tensor([[[-0.2638,  0.1978, -0.0314, -0.3383, -0.0495, -0.2894,  0.0933,
          -0.0560],
         [-0.2631,  0.2025, -0.0310, -0.3350, -0.0454, -0.2981,  0.0871,
          -0.0472],
         [-0.2636,  0.2006, -0.0318, -0.3366, -0.0491, -0.2911,  0.0910,
          -0.0538],
         [-0.2640,  0.1995, -0.0325, -0.3376, -0.0470, -0.2942,  0.0909,
          -0.0517]]], grad_fn=<ViewBackward0>)

Final Output (after FC layer):
tensor([[[-0.0445,  0.0602,  0.1060, -0.2020, -0.0383,  0.4067,  0.1886,
          -0.4688],
         [-0.0431,  0.0651,  0.1006, -0.2056, -0.0373,  0.4080,  0.1898,
          -0.4703],
         [-0.0436,  0.0610,  0.1040, -0.2035, -0.0379,  0.4067,  0.1887,
          -0.4693],
         [-0.0434,  0.0631,  0.1042, -0.2033, -0.0377,  0.4079,  0.1884,
          -0.4699]]], grad_fn=<ViewBackward0>)
```

4 Feed-Forward Network (FFN)

After self-attention, the output is passed through a simple **Feed-Forward Neural Network (FFN)** to introduce **non-linearity and further transformations**.

- The FFN consists of:
  - A **linear layer** that expands the dimensionality (`d_model → d_ff`)
  - A **ReLU activation** for non-linearity
  - A **second linear layer** that projects back to `d_model`
- This helps the model **learn complex representations** beyond attention-based dependencies.

```python
# Step 4: Feed-Forward Network
class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff=512):
        super(FeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)

    def forward(self, x):
        return self.fc2(F.relu(self.fc1(x)))
```

- **Defines a feed-forward network.**
- **Applies ReLU activation and transforms back to `d_model`.**

- **Feed-Forward Network** is a simple two-layer fully connected network applied to each position separately.
- `d_ff`: The number of units in the hidden layer of the feed-forward network (typically larger than `d_model`).
- `relu`: The activation function used in the hidden layer.

```
    def forward(self, x):
        print("\nInput to Feed-Forward Network:")
        print(x)

        x_fc1 = self.fc1(x)
        print("\nOutput after first Linear Layer (fc1):")
        print(x_fc1)

        x_relu = F.relu(x_fc1)
        print("\nOutput after ReLU Activation:")
        print(x_relu)

        x_fc2 = self.fc2(x_relu)
        print("\nFinal Output after second Linear Layer (fc2):")
        print(x_fc2)

        return x_fc2
```

```
Input to Feed-Forward Network:
tensor([[[0.6072, 0.3067, 0.9384, 0.5358, 0.6632, 0.6631, 0.2163, 0.2854],
         [0.2046, 0.3522, 0.7372, 0.0804, 0.9467, 0.3911, 0.9607, 0.4374],
         [0.6901, 0.5768, 0.2444, 0.3471, 0.7870, 0.3423, 0.4047, 0.5248],
         [0.8308, 0.1798, 0.5596, 0.0091, 0.4320, 0.3940, 0.2811, 0.8223]]])

Output after first Linear Layer (fc1):
tensor([[[ 0.1442,  0.1719, -0.0513,  ..., -0.5271,  0.5227,  0.3440],
         [ 0.5912,  0.0312,  0.0215,  ..., -0.4863,  0.0659,  0.1485],
         [ 0.3076,  0.0895, -0.0309,  ..., -0.5965,  0.2032,  0.4423],
         [ 0.4466, -0.1877,  0.1568,  ..., -0.6326,  0.4012,  0.1717]]],
       grad_fn=<ViewBackward0>)

Output after ReLU Activation:
tensor([[[0.1442, 0.1719, 0.0000,  ..., 0.0000, 0.5227, 0.3440],
         [0.5912, 0.0312, 0.0215,  ..., 0.0000, 0.0659, 0.1485],
         [0.3076, 0.0895, 0.0000,  ..., 0.0000, 0.2032, 0.4423],
         [0.4466, 0.0000, 0.1568,  ..., 0.0000, 0.4012, 0.1717]]],
       grad_fn=<ReluBackward0>)

Final Output after second Linear Layer (fc2):
tensor([[[ 0.2723,  0.0110,  0.0524,  0.1868, -0.0674,  0.1553, -0.1358,
           0.1721],
         [ 0.3829,  0.0468,  0.2027,  0.2296, -0.0086,  0.0800, -0.1392,
           0.1396],
         [ 0.2904,  0.0516,  0.1415,  0.2412, -0.0546,  0.0780, -0.0862,
           0.1395],
         [ 0.3083,  0.0841,  0.0924,  0.3281, -0.1050,  0.0645, -0.0609,
           0.0782]]], grad_fn=<ViewBackward0>)
```

```python
# Step 5: Encoder Layer
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff=512):
        super(EncoderLayer, self).__init__()
        self.multihead_attn = MultiHeadAttention(d_model, num_heads)
        self.feedforward = FeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, x, mask=None):
        attn_output = self.multihead_attn(x, mask)
        x = self.norm1(x + attn_output)
        ff_output = self.feedforward(x)
        x = self.norm2(x + ff_output)
        return x
```

- An **Encoder Layer** consists of a multi-head attention mechanism and a feed-forward network.
- The input is passed through the multi-head attention, followed by normalization and a feed-forward network.
- **LayerNorm** ensures that the output is normalized before being passed to the next layer.

```python
def forward(self, x, mask=None):
    print("\n=== Input to Encoder Layer ===")
    print(x)

    attn_output = self.multihead_attn(x, mask)
    print("\n=== Output after Multi-Head Attention ===")
    print(attn_output)

    x = self.norm1(x + attn_output)
    print("\n=== After First Layer Normalization (norm1) ===")
    print(x)

    ff_output = self.feedforward(x)
    print("\n=== Output after Feed-Forward Network ===")
    print(ff_output)

    x = self.norm2(x + ff_output)
    print("\n=== After Second Layer Normalization (norm2) ===")
    print(x)

    return x
```

```
=== Input to Encoder Layer ===
tensor([[[0.4497, 0.7260, 0.4404, 0.4178, 0.7647, 0.5061, 0.9833, 0.0241],
         [0.6608, 0.5843, 0.9895, 0.3194, 0.1309, 0.7326, 0.7213, 0.6173],
         [0.8762, 0.6489, 0.9531, 0.7062, 0.2892, 0.3733, 0.9185, 0.5118],
         [0.2711, 0.2895, 0.3719, 0.8460, 0.9464, 0.5677, 0.8753, 0.8132]]])

Query Matrix (Q):
tensor([[[[ 0.5034, -0.2670,  0.2818,  0.0607],
          [ 0.3842, -0.2731,  0.5920,  0.2194],
          [ 0.4148, -0.2922,  0.6939,  0.1690],
          [ 0.4037, -0.3556,  0.4523,  0.0252]],

         [[ 0.0741,  0.3003, -0.6386, -0.6604],
          [-0.2032,  0.6905, -0.4048, -0.5080],
          [ 0.0092,  0.7632, -0.7486, -0.7039],
          [-0.0036,  0.5697, -0.5393, -0.6051]]]],
       grad_fn=<TransposeBackward0>)

Key Matrix (K):
tensor([[[[-0.6169, -0.1033,  1.1295, -0.3342],
          [-0.7209, -0.4332,  1.4594, -0.5364],
          [-0.7757, -0.3507,  1.4784, -0.5839],
          [-0.5807, -0.2013,  1.2884, -0.7128]],

         [[-0.0739, -0.0299,  0.5519,  0.2881],
          [-0.2912, -0.0606,  0.2285,  0.2835],
          [-0.4290, -0.2863,  0.4894,  0.2527],
          [-0.2570, -0.0884,  0.6984,  0.3097]]]],
       grad_fn=<TransposeBackward0>)

Value Matrix (V):
tensor([[[[-0.8372, -0.1793, -0.7788, -0.2899],
          [-0.5660, -0.0224, -0.9742, -0.7210],
          [-0.8816,  0.0088, -1.0154, -0.5776],
          [-0.8158, -0.1582, -0.5429, -0.1548]],

         [[-0.1018, -0.1684,  0.2900, -0.0033],
          [-0.0241,  0.0622,  0.3982,  0.3308],
          [ 0.0994,  0.0052,  0.4233,  0.1018],
          [-0.2425,  0.0617,  0.0180, -0.1023]]]],
       grad_fn=<TransposeBackward0>)
```

```
Raw Attention Scores (before softmax):
tensor([[[[ 0.0075,  0.0657,  0.0421,  0.0406],
          [ 0.1933,  0.2938,  0.2724,  0.2191],
          [ 0.2508,  0.3748,  0.3539,  0.2958],
          [ 0.1451,  0.2548,  0.2327,  0.2010]],

         [[-0.2786, -0.1864, -0.2986, -0.3481],
          [-0.1877, -0.1096, -0.2185, -0.2244],
          [-0.3197, -0.2098, -0.3833, -0.4053],
          [-0.2443, -0.1641, -0.2892, -0.3067]]]], grad_fn=<DivBackward0>)

Attention Weights (after softmax):
tensor([[[[0.2422, 0.2567, 0.2507, 0.2503],
          [0.2373, 0.2624, 0.2568, 0.2435],
          [0.2333, 0.2641, 0.2586, 0.2440],
          [0.2345, 0.2616, 0.2559, 0.2479]],

         [[0.2494, 0.2735, 0.2445, 0.2327],
          [0.2491, 0.2693, 0.2415, 0.2401],
          [0.2517, 0.2810, 0.2362, 0.2311],
          [0.2513, 0.2723, 0.2403, 0.2361]]]], grad_fn=<SoftmaxBackward0>)
```

```
Attention Output (before final FC layer):
tensor([[[-0.7734, -0.0866, -0.8292, -0.4389, -0.0641, -0.0094,  0.2889,
           0.0907],
         [-0.7722, -0.0847, -0.8334, -0.4440, -0.0661, -0.0091,  0.2860,
           0.0883],
         [-0.7719, -0.0841, -0.8340, -0.4452, -0.0650, -0.0094,  0.2890,
           0.0925],
         [-0.7723, -0.0849, -0.8320, -0.4429, -0.0655, -0.0096,  0.2873,
           0.0895]]], grad_fn=<ViewBackward0>)

Final Output (after FC layer):
tensor([[[ 0.0310,  0.3955,  0.5592,  0.1114, -0.1290,  0.2619,  0.0223,
           0.3381],
         [ 0.0321,  0.3977,  0.5593,  0.1092, -0.1300,  0.2599,  0.0210,
           0.3418],
         [ 0.0326,  0.3976,  0.5619,  0.1087, -0.1302,  0.2600,  0.0211,
           0.3410],
         [ 0.0320,  0.3971,  0.5598,  0.1097, -0.1299,  0.2605,  0.0214,
           0.3406]]], grad_fn=<ViewBackward0>)

=== Output after Multi-Head Attention ===
tensor([[[ 0.0310,  0.3955,  0.5592,  0.1114, -0.1290,  0.2619,  0.0223,
           0.3381],
         [ 0.0321,  0.3977,  0.5593,  0.1092, -0.1300,  0.2599,  0.0210,
           0.3418],
         [ 0.0326,  0.3976,  0.5619,  0.1087, -0.1302,  0.2600,  0.0211,
           0.3410],
         [ 0.0320,  0.3971,  0.5598,  0.1097, -0.1299,  0.2605,  0.0214,
           0.3406]]], grad_fn=<ViewBackward0>)

=== After First Layer Normalization (norm1) ===
tensor([[[-0.9807,  1.4639,  0.9987, -0.7962, -0.3895,  0.1150,  1.0218,
          -1.4330],
         [-0.2362,  0.4436,  1.7760, -0.8578, -1.8632,  0.4681, -0.1200,
           0.3895],
         [ 0.1403,  0.5267,  1.8401, -0.1228, -1.9622, -0.6320,  0.2267,
          -0.0168],
         [-2.2277, -0.5797,  0.4734,  0.5764, -0.0218,  0.0289,  0.3230,
           1.4275]]], grad_fn=<NativeLayerNormBackward0>)
```

```
Input to Feed-Forward Network:
tensor([[[-0.9807,  1.4639,  0.9987, -0.7962, -0.3895,  0.1150,  1.0218,
          -1.4330],
         [-0.2362,  0.4436,  1.7760, -0.8578, -1.8632,  0.4681, -0.1200,
           0.3895],
         [ 0.1403,  0.5267,  1.8401, -0.1228, -1.9622, -0.6320,  0.2267,
          -0.0168],
         [-2.2277, -0.5797,  0.4734,  0.5764, -0.0218,  0.0289,  0.3230,
           1.4275]]], grad_fn=<NativeLayerNormBackward0>)

Output after first Linear Layer (fc1):
tensor([[[ 0.0379, -0.1548,  0.9839,  ...,  1.1206, -0.5564,  0.3780],
         [ 0.5328, -0.5287,  0.7750,  ..., -0.4577, -0.1770,  0.4337],
         [ 0.8690, -0.6263,  1.0986,  ...,  0.0652, -0.4273,  0.4682],
         [-0.4408, -0.7520, -0.7925,  ..., -0.3779, -0.4454, -0.0482]]],
       grad_fn=<ViewBackward0>)

Output after ReLU Activation:
tensor([[[0.0379, 0.0000, 0.9839,  ..., 1.1206, 0.0000, 0.3780],
         [0.5328, 0.0000, 0.7750,  ..., 0.0000, 0.0000, 0.4337],
         [0.8690, 0.0000, 1.0986,  ..., 0.0652, 0.0000, 0.4682],
         [0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.0000, 0.0000]]],
       grad_fn=<ReluBackward0>)

Final Output after second Linear Layer (fc2):
tensor([[[ 0.3366,  0.3459, -0.2057,  0.0539, -0.5932,  0.0390, -0.1564,
          -0.3041],
         [-0.1223,  0.2358,  0.0662,  0.0273, -0.2108,  0.2893, -0.3187,
          -0.4297],
         [-0.2161,  0.0633,  0.0308,  0.2714, -0.2811,  0.1356, -0.1956,
          -0.6492],
         [ 0.1906,  0.4363,  0.1223, -0.4548, -0.1615,  0.1959, -0.2052,
          -0.3630]]], grad_fn=<ViewBackward0>)
```

```
=== Output after Feed-Forward Network ===
tensor([[[ 0.3366,  0.3459, -0.2057,  0.0539, -0.5932,  0.0390, -0.1564,
           -0.3041],
         [-0.1223,  0.2358,  0.0662,  0.0273, -0.2108,  0.2893, -0.3187,
           -0.4297],
         [-0.2161,  0.0633,  0.0308,  0.2714, -0.2811,  0.1356, -0.1956,
           -0.6492],
         [ 0.1906,  0.4363,  0.1223, -0.4548, -0.1615,  0.1959, -0.2052,
           -0.3630]]], grad_fn=<ViewBackward0>)

=== After Second Layer Normalization (norm2) ===
tensor([[[-0.5329,  1.7076,  0.7793, -0.6225, -0.8420,  0.1958,  0.8454,
           -1.5308],
         [-0.2728,  0.6689,  1.7241, -0.7011, -1.8295,  0.7397, -0.3455,
            0.0161],
         [ 0.0269,  0.6375,  1.8123,  0.2327, -1.9610, -0.3589,  0.1249,
           -0.5144],
         [-2.3646, -0.1338,  0.7371,  0.1785, -0.1807,  0.3001,  0.1741,
            1.2893]]], grad_fn=<NativeLayerNormBackward0>)
```

```python
# Step 6: Decoder Layer
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff=512):
        super(DecoderLayer, self).__init__()
        self.multihead_attn1 = MultiHeadAttention(d_model, num_heads)
        self.multihead_attn2 = MultiHeadAttention(d_model, num_heads)
        self.feedforward = FeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.norm3 = nn.LayerNorm(d_model)

    def forward(self, x, encoder_output, tgt_mask=None, src_mask=None):
        attn_output1 = self.multihead_attn1(x, tgt_mask)
        x = self.norm1(x + attn_output1)
        attn_output2 = self.multihead_attn2(x, src_mask)
        x = self.norm2(x + attn_output2)
        ff_output = self.feedforward(x)
        x = self.norm3(x + ff_output)
        return x
```

- A **Decoder Layer** is similar to an Encoder Layer, but it attends to both the target sequence (`tgt_mask`) and the encoder output (`src_mask`).
- Two multi-head attention layers are applied: one for attending to the target sequence and another for attending to the encoder output.

```python
# Step 7: Transformer Model
class Transformer(nn.Module):
    def __init__(self, vocab_size, d_model, num_heads, num_encoder_layers, num_decoder_layers, max_len=5000):
        super(Transformer, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model)
        self.pos_encoder = PositionalEncoding(d_model, max_len)
        self.encoder_layers = nn.ModuleList([EncoderLayer(d_model, num_heads) for _ in range(num_encoder_layers)])
        self.decoder_layers = nn.ModuleList([DecoderLayer(d_model, num_heads) for _ in range(num_decoder_layers)])
        self.fc_out = nn.Linear(d_model, vocab_size)

    def forward(self, src, tgt, tgt_mask=None):
        src = self.pos_encoder(self.embedding(src))
        tgt = self.pos_encoder(self.embedding(tgt))

        encoder_output = src
        for layer in self.encoder_layers:
            encoder_output = layer(encoder_output)

        decoder_output = tgt
        for layer in self.decoder_layers:
            decoder_output = layer(decoder_output, encoder_output, tgt_mask=tgt_mask)

        output = self.fc_out(decoder_output)
        return output
```

- **Transformer Model** is the overall model that uses multiple encoder and decoder layers.
- `embedding`: An embedding layer converts word indices to word vectors of size `d_model`.
- `pos_encoder`: Adds positional encodings to the word embeddings.
- `encoder_layers` and `decoder_layers`: The model stacks multiple encoder and decoder layers.
- `fc_out`: Final linear layer to predict the output vocabulary.

```python
# Step 8: Prediction Module
def translate(input_sentence, word_map_en, word_map_fr, transformer):
    # Tokenize input sentence
    input_tensor = tokenize(input_sentence, word_map_en).unsqueeze(0)  # Shape (1, seq_len)

    # Generate a mask for the target sentence
    tgt_mask = torch.tril(torch.ones((input_tensor.size(1), input_tensor.size(1)))).unsqueeze(0).unsqueeze(0)  # Lower triangular mask

    # Initialize a tensor for the target sentence
    target_tensor = torch.zeros((1, input_tensor.size(1)), dtype=torch.long)

    # Predict the output sentence (translation)
    output = transformer(input_tensor, target_tensor, tgt_mask)

    # Apply Softmax to get probabilities
    softmax_output = F.softmax(output, dim=-1)

    # Get predicted token indices (Argmax)
    predicted_tokens = torch.argmax(softmax_output, dim=-1)

    # Convert predicted tokens back to words
    reverse_word_map_fr = {v: k for k, v in word_map_fr.items()}
    translated_sentence = [reverse_word_map_fr[token.item()] for token in predicted_tokens[0] if token != 0]

    return " ".join(translated_sentence)
```

```
# Step 9: Initialize Model
vocab_size_en = len(word_map_en)
vocab_size_fr = len(word_map_fr)
d_model = 128
num_heads = 8
num_encoder_layers = 2
num_decoder_layers = 2

transformer = Transformer(vocab_size_en, d_model, num_heads, num_encoder_layers, num_decoder_layers)
```

- `vocab_size_en` and `vocab_size_fr`: The size of the English and French vocabularies.
- `d_model`: The dimensionality of the embeddings and model (128).
- `num_heads`: Number of attention heads (8).
- `num_encoder_layers` and `num_decoder_layers`: Number of encoder and decoder layers (2).
- `transformer`: The instantiated Transformer model.