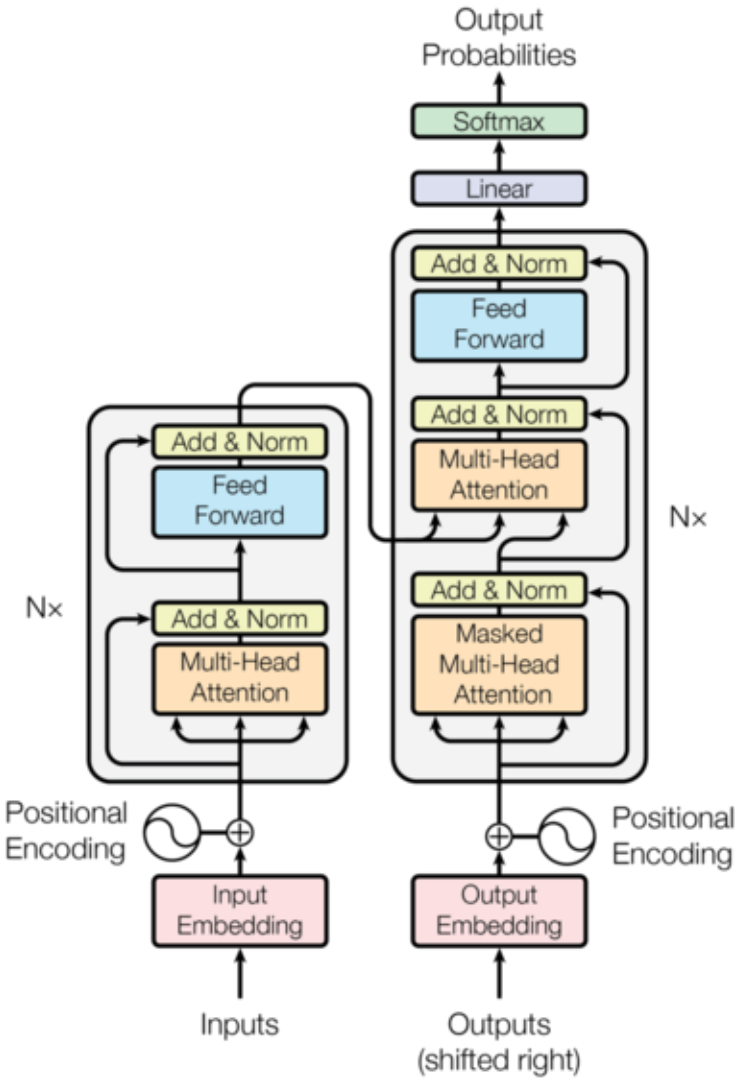
Transformer From Scratch Using Python

<목차>



전체적인 흐름을 보여주는 이미지

목차1번은 간단하게 넘어가고, 2번 Import부터 시작이다.

<2. Import libraries>

```
# importing required libraries import torch.nn as nn
import torch
import torch.nn.functional as F
import math,copy,re
import warnings
import pandas as pd
import numpy as np
import seaborn as sns
import torchtext
import matplotlib.pyplot as plt
warnings.simplefilter("ignore")
print(torch.__version__)
```

```
#torchtext버전 충돌해결방법 : torch, torchtext 호환버전 맞추기#!pip install torch==2.2.0
torchtext==0.17.0
```

이때 나처럼 버전 관리를 안해준사람들은 torchtext에서 버전 오류가 발생하게 된다.

https://pypi.org/project/torchtext/ → 여기서 torchtext와 torch버전 확인하고 맞추면 금방 해결된다.!

<3. Basic components>

Create Word Embeddings

```
class Embedding(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        """
        Args:
            vocab_size: size of vocabulary
            embed_dim: dimension of embeddings
        """
        super(Embedding, self).__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
    def forward(self, x):
        """
        Args:
            x: input vector
        Returns:
            out: embedding vector
        """
        out = self.embed(x)
        return out
```

- **각 단어를 임베딩 벡터로 변환**: 입력 시퀀스의 각 단어를 임베딩 벡터로 변환해야 한다. 임베딩 벡터는 각 단어를 더 의미론적으로 표현하는 방식이다.
- **임베딩 벡터 크기**: 각 임베딩 벡터의 차원이 512이라고 가정한다. 즉, 단어 하나를 512차원의 벡터로 표현하게 된다.
- **임베딩 매트릭스의 크기**: 만약 단어 집합(어휘) 크기가 100이라면, 임베딩 매트릭스는 100개의 단어 각각에 대해 512차원의 벡터를 가지므로 매트릭스의 크기는 **100x512(임베딩 벡터차원)**가 된다. 이 매트릭스는 학습 과정에서 학습되며, 추론(inference) 중에는 각 단어가 해당하는 512차원의 벡터에 매핑된다.
- **출력 차원**: 배치 크기가 32이고, 각 시퀀스의 길이가 10(즉, 10개의 단어로 구성된 시퀀스)이라고 가정하면, 모델의 출력은 **32x10×512** 크기의 텐서가 된다. 즉, 배치의 각 시퀀스에 대해 10개의 단어가 각각 512차원의 임베딩 벡터로 변환된다.

임베딩 벡터는 입력 단어를 고차원 공간에서 의미적으로 더 풍부하게 표현하며, 이를 통해 모델이 더 나은 성능을 발휘할 수 있게 한다.

- 왜 512차원으로 가정하는가?

: 자연어 처리에서 임베딩벡터의 차원을 선택할 때 일반적으로 균형을 고려하기 때문이다. 효율성과 성능을 모두 고려한 경험적 선택이다.

Positional Encoding

```
# register buffer in Pytorch ->
# If you have parameters in your model, which should be saved and restored in the state_dict,
# but not trained by the optimizer, you should register them as buffers.
class PositionalEmbedding(nn.Module):
    def __init__(self,max_seq_len,embed_model_dim):
        """
        Args:
            seq_len: length of input sequence
            embed_model_dim: demension of embedding
        """
        super(PositionalEmbedding, self).__init__()
        self.embed_dim = embed_model_dim

        pe = torch.zeros(max_seq_len,self.embed_dim)
        for pos in range(max_seq_len):
            for i in range(0,self.embed_dim,2):
                pe[pos, i] = math.sin(pos / (10000 ** ((2 * i)/self.embed_dim)))
                pe[pos, i + 1] = math.cos(pos / (10000 ** ((2 * (i + 1))/self.embed_dim)))
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)


    def forward(self, x):
        """
        Args:
            x: input vector
        Returns:
            x: output
        """
```

```
# make embeddings relatively larger
        x = x * math.sqrt(self.embed_dim)
#add constant to embedding
        seq_len = x.size(1)
        x = x + torch.autograd.Variable(self.pe[:,:seq_len], r
equires_grad=False)
        return x
```

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

1. **Positional Encoding 생성**: 모델이 문장의 의미를 이해하려면 각 단어에 대해 두 가지를 알아야 한다:
   - 그 단어가 무엇을 의미하는지(단어의 의미).
   - 그 단어가 문장에서 어느 위치에 있는지(단어의 위치).
2. **"Attention is All You Need" 논문에서 사용된 방식**: 이 논문에서 위치 인코딩을 만들기 위해 다음의 수학적 함수를 사용했다:
   - 홀수 번째 위치에서는 **코사인 함수**가 사용된다.
   - 짝수 번째 위치에서는 **사인 함수**가 사용된다.
3. **용어 설명**:
   - **pos**는 문장 내에서 단어의 순서를 나타낸다.
   - **i**는 임베딩 벡터 차원에서의 위치를 나타낸다.
4. **Positional Embedding**: 위치 인코딩은 임베딩 매트릭스와 비슷한 크기의 매트릭스를 생성한다. 즉, 문장에서 각 단어에 대해 임베딩 차원과 동일한 크기의 위치 벡터를 생성하는 것이다.
   - 예를 들어, 각 단어(토큰)에 대해 1 x 512 크기의 임베딩 벡터가 생성되고, 여기에 1 x 512 크기의 위치 벡터가 더해져서 최종적으로 각 단어에 대해 1 x 512 크기의 벡터를 얻는다.
5. **출력 벡터**: 예를 들어, 배치 크기가 32, 시퀀스 길이가 10, 그리고 임베딩 차원이 512인 경우, 임베딩 벡터의 크기는 **32 x 10 × 512**가 된다. 비슷하게, 위치 인코딩 벡터도 **32 x 10 × 512** 크기로 생성된다. 그런 다음, 두 벡터를 더해서 최종적인 **32 x 10 × 512** 크기의 출력 벡터를 얻는다.

결론적으로, Positional Encoding은 단어의 위치 정보를 모델에 전달하여 단어 간 순서와 문맥을 더 잘 이해할 수 있도록 돕는 역할을 한다.

Self Attention

```
class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim=512, n_heads=8):
        """
        Args:
            embed_dim: dimension of embeding vector output
            n_heads: number of self attention heads
        """
        super(MultiHeadAttention, self).__init__()

        self.embed_dim = embed_dim#512 dim
        self.n_heads = n_heads#8
        self.single_head_dim = int(self.embed_dim / self.n_hea
ds)#512/8 = 64  . each key,query, value will be of 64d#key,que
ry and value matrixes    #64 x 64
        self.query_matrix = nn.Linear(self.single_head_dim , s
elf.single_head_dim ,bias=False)# single key matrix for all 8
keys #512x512
        self.key_matrix = nn.Linear(self.single_head_dim  , se
lf.single_head_dim, bias=False)
        self.value_matrix = nn.Linear(self.single_head_dim ,se
lf.single_head_dim , bias=False)
        self.out = nn.Linear(self.n_heads*self.single_head_dim
,self.embed_dim)

    def forward(self,key,query,value,mask=None):#batch_size x
sequence_length x embedding_dim    # 32 x 10 x 512"""
        Args:
           key : key vector
           query : query vector
           value : value vector
           mask: mask for decoder

        Returns:
           output vector from multihead attention
        """
```

```
        batch_size = key.size(0)
        seq_length = key.size(1)

# query dimension can change in decoder during inference.# so
we cant take general seq_length
        seq_length_query = query.size(1)

# 32x10x512
        key = key.view(batch_size, seq_length, self.n_heads, s
elf.single_head_dim)#batch_size x sequence_length x n_heads x
single_head_dim = (32x10x8x64)
        query = query.view(batch_size, seq_length_query, self.
n_heads, self.single_head_dim)#(32x10x8x64)
        value = value.view(batch_size, seq_length, self.n_head
s, self.single_head_dim)#(32x10x8x64)

        k = self.key_matrix(key)# (32x10x8x64)
        q = self.query_matrix(query)
        v = self.value_matrix(value)

        q = q.transpose(1,2)# (batch_size, n_heads, seq_len, s
ingle_head_dim)    # (32 x 8 x 10 x 64)
        k = k.transpose(1,2)# (batch_size, n_heads, seq_len, s
ingle_head_dim)
        v = v.transpose(1,2)# (batch_size, n_heads, seq_len, s
ingle_head_dim)# computes attention# adjust key for matrix mul
tiplication
        k_adjusted = k.transpose(-1,-2)#(batch_size, n_heads,
single_head_dim, seq_ken)  #(32 x 8 x 64 x 10)
        product = torch.matmul(q, k_adjusted)#(32 x 8 x 10 x 6
4) x (32 x 8 x 64 x 10) = #(32x8x10x10)# fill those positions
of product matrix as (-1e20) where mask positions are 0if mask
is not None:
             product = product.masked_fill(mask == 0, float("-
1e20"))

#divising by square root of key dimension
        product = product / math.sqrt(self.single_head_dim)# /
sqrt(64)#applying softmax
        scores = F.softmax(product, dim=-1)
```

```
#mutiply with value matrix
        scores = torch.matmul(scores, v)##(32x8x 10x 10) x (32
x 8 x 10 x 64) = (32 x 8 x 10 x 64)#concatenated output
        concat = scores.transpose(1,2).contiguous().view(batch
_size, seq_length_query, self.single_head_dim*self.n_heads)#
(32x8x10x64) -> (32x10x8x64)  -> (32,10,512)

        output = self.out(concat)#(32,10,512) -> (32,10,512)re
turn output
```

클래스 설명: MultiHeadAttention

- **역할**: 이 클래스는 입력 임베딩 벡터에 대해 여러 개의 Self-Attention 헤드를 계산하고, 이를 결합해 최종 출력을 생성한다.

생성자 __init__(self, embed_dim=512, n_heads=8)

- **embed_dim (기본값 512)**: 임베딩 벡터의 차원. 즉, 각 단어가 표현되는 벡터의 크기.
- **n_heads (기본값 8)**: Self-Attention의 헤드 개수. 여러 헤드로 나눠 병렬로 처리한다.

내부 변수 및 레이어 정의

- **self.embed_dim**: 임베딩 벡터의 차원. 기본값은 512.
- **self.n_heads**: Self-Attention의 헤드 수. 기본값은 8.
- **self.single_head_dim**: 각 Self-Attention 헤드에 할당된 차원. **512/8 = 64**이므로 각 헤드마다 64차원의 Query, Key, Value 벡터를 사용한다.

Query, Key, Value 행렬 생성

- **self.query_matrix**: Query 벡터를 생성하기 위한 선형 변환 레이어. 입력 차원과 출력 차원이 모두 64로, 각 헤드에 대해 별도의 Query 벡터를 생성한다.
- **self.key_matrix**: Key 벡터를 생성하기 위한 선형 변환 레이어. 입력 및 출력 차원은 동일하게 64.
- **self.value_matrix**: Value 벡터를 생성하기 위한 선형 변환 레이어. 마찬가지로 64 차원의 입력과 출력을 처리한다.

최종 출력 레이어

- **self.out**: 여러 Self-Attention 헤드를 결합한 후, 최종 출력을 생성하는 선형 변환 레이어.
    - 입력 차원은 **8개의 헤드 x 64차원**이므로 512차원이고, 출력도 임베딩 차원인 512차원이다.

이 구조를 통해 입력 벡터를 각 헤드로 분리하여 병렬로 Self-Attention을 계산한 후, 최종적으로 다시 결합하여 512차원의 출력 벡터를 얻는다.

함수 설명: forward(self, key, query, value, mask=None)

Multi-Head attention모듈의 forward메서드를 구현한것으로, 3개의 k,q,v벡터를 기반으로 Self-Attention을 계산하고 출력하는 과정이다.

- **입력 인자**:
  - key, query, value: Key, Query, Value 벡터들로, 차원은 (batch_size, sequence_length, embedding_dim)이다. (예: **32x10×512**)
  - mask: 선택적 인자이며, 디코더에서 특정 토큰을 마스킹하기 위한 마스크
- **출력**: Multi-Head Attention의 최종 출력. 크기는 **(batch_size, seq_length_query, embedding_dim)**
- self.key_matrix, self.query_matrix, self.value_matrix 레이어를 사용해 각각의 Key, Query, Value 벡터를 생성함
- Key 벡터를 전치(transpose)하여 (batch_size, n_heads, single_head_dim, sequence_length) 크기로 변환.
- Query 벡터와 Key 벡터를 곱해 각 쿼리에 대한 주의(attention) 점수를 계산함. 결과 크기는 (batch_size, n_heads, seq_length_query, seq_length). (예: 32x8×10×10)

Self-Attention 단계별 설명!!!

Step 1:

각 단어에 대해 세 개의 벡터(Query, Key, Value)를 생성한다. 각 벡터의 크기는 **1x64**이다. Multi Head Attention를 사용하므로, 8개의 Self-Attention 헤드가 존재한다.

예시: 배치 크기 32, 시퀀스 길이 10, 임베딩 차원이 512일 때, 임베딩 후 출력 크기는 **32x10×512**가 되며, 이를 **32x10×8×64**로 변환한다. 여기서 **8**은 헤드의 수를 의미한다.

Step 2:

Query 벡터와 Key 벡터를 곱해 점수를 계산한다.

예시: Query와 Key 벡터의 크기가 **32x8×10×64**일 때, Key 벡터를 전치(transpose)하여 곱한다. 즉, **(32x8×10×64) x (32×8×64×10) = 32×8×10×10** 크기의 행렬이 된다.

Step 3:

출력 행렬을 Key 벡터 차원의 제곱근으로 나눈 후, Softmax를 적용한다.

예시: 출력 크기 **32x8×10×10**의 벡터를 8(64의 제곱근)로 나눈다.

Step 4:

이 행렬을 Value 벡터와 곱한다. (Value벡터의 등장)

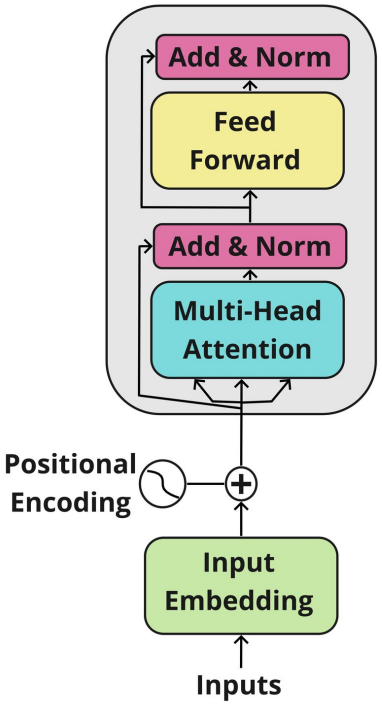
예시: **32x8×10×10** 행렬을 Value 벡터 **32x8×10×64**와 곱해 **32x8×10×64** 크기의 출력 벡터를 얻는다.

Step 5:

출력 벡터를 선형 레이어에 통과시켜 최종 출력을 얻는다.

예시: **32x8×10×64** 벡터를 **32x10×8×64**로 전치한 후 **32x10×512**로 변환하고, 선형 레이어를 통과시켜 최종 출력 **32x10×512**를 생성한다.

<4. Encoder>



인코더에서 데이터가 처리되는 과정

Encoder 단계별 설명!!!

Step 1: Embedding 및 Positional Encoding (전에 수행한 과정)

과정:

- 입력으로 주어진 패딩된 토큰(문장에 해당하는)이 **임베딩 레이어**와 **Positional Encoding 레이어**를 거친다.

코드:

- 입력 크기가 **32x10**(배치 크기=32, 시퀀스 길이=10)인 경우, 임베딩 레이어를 통과하면 크기가 **32x10×512**로 변환된다.

- 여기서 positional encoding 벡터(크기 **32x10×512**)와 더해져, 최종적으로 크기가 **32x10×512**인 출력이 생성된다.
- 이 출력은 **Multihead Attention** 레이어로 전달된다.

Step 2: Multihead Attention(전에 수행한 과정)

과정:

- 위에서 생성된 **32x10×512** 입력이 **Multihead Attention** 레이어를 통과하면서 유용한 표현 행렬을 생성한다.

코드 힌트:

- Multihead Attention의 입력은 **32x10×512**이며, 여기서 Key, Query, Value 벡터가 생성된다. 각 벡터는 **32x10×512**로 처리되고, 최종 출력 역시 **32x10×512**의 크기를 갖는다.

Step 3: Normalization 및 Residual Connection

과정:

- Multihead Attention의 출력(32x10×512)은 그 입력(Embedding 및 Positional Encoding으로부터 생성된 **32x10×512**)과 더해진다.
    - Residual Output=Attention Output+Input (Embedding + Positional Encoding)
- 이 덧셈 결과는 **Layer Normalization**을 거쳐 정규화된다.

코드 힌트:

- Multihead Attention의 출력 **32x10×512**는 임베딩 레이어에서 나온 입력 **32x10×512**와 더해진다.
- 그런 다음, 이 값은 정규화 레이어를 통과하여 최종적으로 정규화된 출력이 생성된다.
- ***잎고 넘어가기**
- **Embedding + Positional Encoding**은 **Multihead Attention**의 입력으로 사용되지만, **Residual Connection**을 통해 다시 한번 이 입력과 출력이 더해진다.
- Multihead Attention은 단어들 사이의 관계(문맥)를 학습하고, Residual Connection을 통해 모델이 원래 입력의 정보를 유지하면서도 새로운 정보를 추가할 수 있게 돕는다.

따라서 **Embedding + Positional Encoding**과 **Multihead Attention**은 각각 독립적으로 처리되는 것이 아니라, Residual Connection을 통해 상호작용하며 모델의 학습에 기여한다.

Step 4: Feed Forward Layer와 Residual Connection

과정:

- 정규화된 출력은 **Feed Forward Layer**로 전달되며, 다시 한번 **Residual Connection**과 **정규화**가 이루어진다.
    - 정규화는 Residual Connection으로 더해진 값을 일정한 분포로 맞추어 주는역할을 한다.
- Feed Forward 레이어는 2개의 선형 레이어를 거치며, 차원 변환이 이루어진다.(정규화 과정)

코드 힌트:

- 정규화된 출력(32x10×512)은 두 개의 선형 레이어를 통과한다:
    1. 첫 번째 레이어: **32x10×512 → 32×10×2048**
    2. 두 번째 레이어: **32x10×2048 → 32×10×512**
- 이 과정을 통해 최종적으로 인코더의 출력이 생성된다.

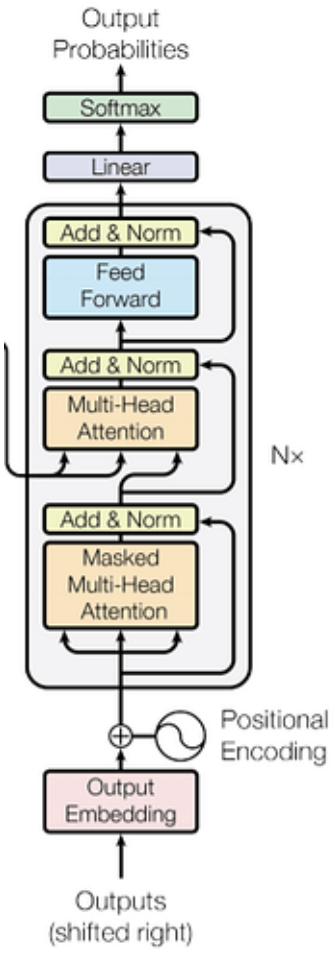
```
class TransformerBlock(nn.Module):
    def __init__(self, embed_dim, expansion_factor=4, n_heads=
8):
        super(TransformerBlock, self).__init__()

        """
        Args:
           embed_dim: dimension of the embedding
           expansion_factor: fator ehich determines output dim
ension of linear layer
           n_heads: number of attention heads

        """
        self.attention = MultiHeadAttention(embed_dim, n_head
s)

        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)

        self.feed_forward = nn.Sequential(
                            nn.Linear(embed_dim, expansion_facto
r*embed_dim),
                            nn.ReLU(),
                            nn.Linear(expansion_factor*embed_di
m, embed_dim)
```

```
        )

        self.dropout1 = nn.Dropout(0.2)
        self.dropout2 = nn.Dropout(0.2)

    def forward(self,key,query,value):

        """
        Args:
           key: key vector
           query: query vector
           value: value vector
           norm2_out: output of transformer block

        """

        attention_out = self.attention(key,query,value)#32x10x
512
        attention_residual_out = attention_out + value#32x10x5
12
        norm1_out = self.dropout1(self.norm1(attention_residua
l_out))#32x10x512

        feed_fwd_out = self.feed_forward(norm1_out)#32x10x512
-> #32x10x2048 -> 32x10x512
        feed_fwd_residual_out = feed_fwd_out + norm1_out#32x10
x512
        norm2_out = self.dropout2(self.norm2(feed_fwd_residual
_out))#32x10x512return norm2_out



class TransformerEncoder(nn.Module):
    """
    Args:
        seq_len : length of input sequence
        embed_dim: dimension of embedding
        num_layers: number of encoder layers
        expansion_factor: factor which determines number of li
near layers in feed forward layer
```

```
        n_heads: number of heads in multihead attention

    Returns:
        out: output of the encoder
    """
    def __init__(self, seq_len, vocab_size, embed_dim, num_lay
ers=2, expansion_factor=4, n_heads=8):
        super(TransformerEncoder, self).__init__()

        self.embedding_layer = Embedding(vocab_size, embed_di
m)
        self.positional_encoder = PositionalEmbedding(seq_len,
embed_dim)

        self.layers = nn.ModuleList([TransformerBlock(embed_di
m, expansion_factor, n_heads) for i in range(num_layers)])

    def forward(self, x):
        embed_out = self.embedding_layer(x)
        out = self.positional_encoder(embed_out)
        for layer in self.layers:
            out = layer(out,out,out)

        return out#32x10x512
```

1.

<5. Decoder>



디코딩 과정 이미지

Decoder 진행과정

Step 1: Embedding 및 Positional Encoding

과정:

- 타겟 시퀀스(예: 번역할 문장)가 먼저 임베딩 레이어와 Positional Encoding 레이어를 통과하여, 각 단어에 해당하는 512차원의 임베딩 벡터를 생성한다.

코드 힌트:

- 시퀀스 길이가 10, 배치 크기가 32, 임베딩 벡터 차원이 512일 경우, 입력 차원은 **32x10**이며, 이를 임베딩 매트릭스를 통과시키면 출력 차원은 **32x10×512**가 된다. 여기에 Positional Encoding 벡터를 더해 최종적으로 **32x10×512** 크기의 벡터가 생성된다.

Step 2: Multihead Attention with Mask

과정:

- 생성된 임베딩 출력이 Multihead Attention 레이어를 통과하면서 Key, Query, Value 벡터가 만들어진다.
- 여기서 **마스크**가 추가되는데, 이는 타겟 시퀀스의 각 단어가 **미래의 단어**를 보지 않도록 하기 위한 것이다. 예를 들어, "I am a student"이라는 문장에서, 단어 "a"는 "student"를 보지 않고 자신의 이전 단어만 보도록 제한된다.

왜 마스크를 사용하는가?

- **Self-Attention**에서 각 단어는 다른 모든 단어를 참조하여 문맥을 파악하는데, 디코더에서 타겟 시퀀스를 처리할 때는 **미래의 단어**를 미리 보는 것을 방지해야 한다. 이는 모델이 단어를 순차적으로 예측하도록 하기 위해서이다.

코드 힌트:

- 마스크는 시퀀스 길이와 동일한 **삼각형 행렬**로 생성된다. 예를 들어, 시퀀스 길이가 5인 경우, 아래와 같은 마스크가 생성된다:

  1 0 0 0 0

  1 1 0 0 0

  1 1 1 0 0

  1 1 1 1 0

  1 1 1 1 1
- 이 마스크를 적용하여 Query와 Key의 곱 결과에서 마스크가 적용된 위치를 **무한대**로 채워 Softmax 계산에서 해당 위치의 값을 0으로 만든다. (코드에서는 매우 작은 값, 예를 들어 **1e20**을 사용한다.)

Step 3: Add & Norm Layer

과정:

- Multihead Attention의 출력과 입력(임베딩 및 Positional Encoding)을 더한 후, 이를 **Layer Normalization**을 통해 정규화한다.
- 이 단계는 인코더에서 본 Add & Norm 과정과 동일하게 작동한다.

Step 4: Encoder-Decoder Multihead Attention

과정:

- 이번에는 **Encoder-Decoder Multihead Attention**이 적용된다. 여기서는 두 가지가 결합된다:
  - **Key**와 **Value** 벡터는 인코더의 출력에서 생성된다.
  - **Query**는 이전 디코더 레이어의 출력에서 생성된다.

- 이렇게 생성된 Query, Key, Value 벡터를 Multihead Attention에 입력하여, 타겟 시퀀스가 인코더 출력과 어떻게 상호작용하는지 학습한다.

코드 힌트:

- 인코더의 출력(예: **32x10×512**)에서 Key와 Value 벡터가 생성되고, 디코더의 이전 출력(예: **32x10×512**)에서 Query 벡터가 생성된다.
- 이를 통해 Multihead Attention을 수행하고, 결과는 Add & Norm 레이어를 통과하여 인코더-디코더 상호작용이 반영된 출력이 생성된다.

Step 5: Feed Forward Layer와 Add & Norm

과정:

- 인코더와 동일한 방식으로, Feed Forward Layer를 통과하고 Add & Norm 레이어를 거친다.
- Feed Forward Layer는 두 개의 선형 레이어로 구성된다:
  - 첫 번째 레이어: **32x10×512 → 32×10×2048**
  - 두 번째 레이어: **32x10×2048 → 32×10×512**

Step 6: Linear Layer와 Softmax

과정:

- 마지막으로, 전체 타겟 어휘(corpus)의 단어 수에 해당하는 크기의 선형 레이어를 통과하여 각 단어의 확률을 구한다.
- **Softmax**를 적용하여 각 단어가 다음 단어로 선택될 확률을 예측한다.

```
class DecoderBlock(nn.Module):
    def __init__(self, embed_dim, expansion_factor=4, n_heads=
8):
        super(DecoderBlock, self).__init__()

        """
        Args:
           embed_dim: dimension of the embedding
           expansion_factor: fator ehich determines output dim
ension of linear layer
           n_heads: number of attention heads

        """
```

```
        self.attention = MultiHeadAttention(embed_dim, n_heads
=8)
        self.norm = nn.LayerNorm(embed_dim)
        self.dropout = nn.Dropout(0.2)
        self.transformer_block = TransformerBlock(embed_dim, e
xpansion_factor, n_heads)


    def forward(self, key, query, x,mask):

        """
        Args:
           key: key vector
           query: query vector
           value: value vector
           mask: mask to be given for multi head attention
        Returns:
           out: output of transformer block

        """

#we need to pass mask mask only to fst attention
        attention = self.attention(x,x,x,mask=mask)#32x10x512
        value = self.dropout(self.norm(attention + x))

        out = self.transformer_block(key, query, value)


        return out


class TransformerDecoder(nn.Module):
    def __init__(self, target_vocab_size, embed_dim, seq_len,
num_layers=2, expansion_factor=4, n_heads=8):
        super(TransformerDecoder, self).__init__()
        """
        Args:
           target_vocab_size: vocabulary size of taget
           embed_dim: dimension of embedding
           seq_len : length of input sequence
```

```
            num_layers: number of encoder layers
            expansion_factor: factor which determines number of
linear layers in feed forward layer
            n_heads: number of heads in multihead attention

        """
        self.word_embedding = nn.Embedding(target_vocab_size,
embed_dim)
        self.position_embedding = PositionalEmbedding(seq_len,
embed_dim)

        self.layers = nn.ModuleList(
            [
                DecoderBlock(embed_dim, expansion_factor=4, n_
heads=8)
                for _ in range(num_layers)
            ]

        )
        self.fc_out = nn.Linear(embed_dim, target_vocab_size)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x, enc_out, mask):

        """
        Args:
            x: input vector from target
            enc_out : output from encoder layer
            trg_mask: mask for decoder self attention
        Returns:
            out: output vector
        """



        x = self.word_embedding(x)#32x10x512
        x = self.position_embedding(x)#32x10x512
        x = self.dropout(x)

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

```
        out = F.softmax(self.fc_out(x))

        return out
```

https://www.kaggle.com/code/pdochannel/swin-transformer-in-pytorch