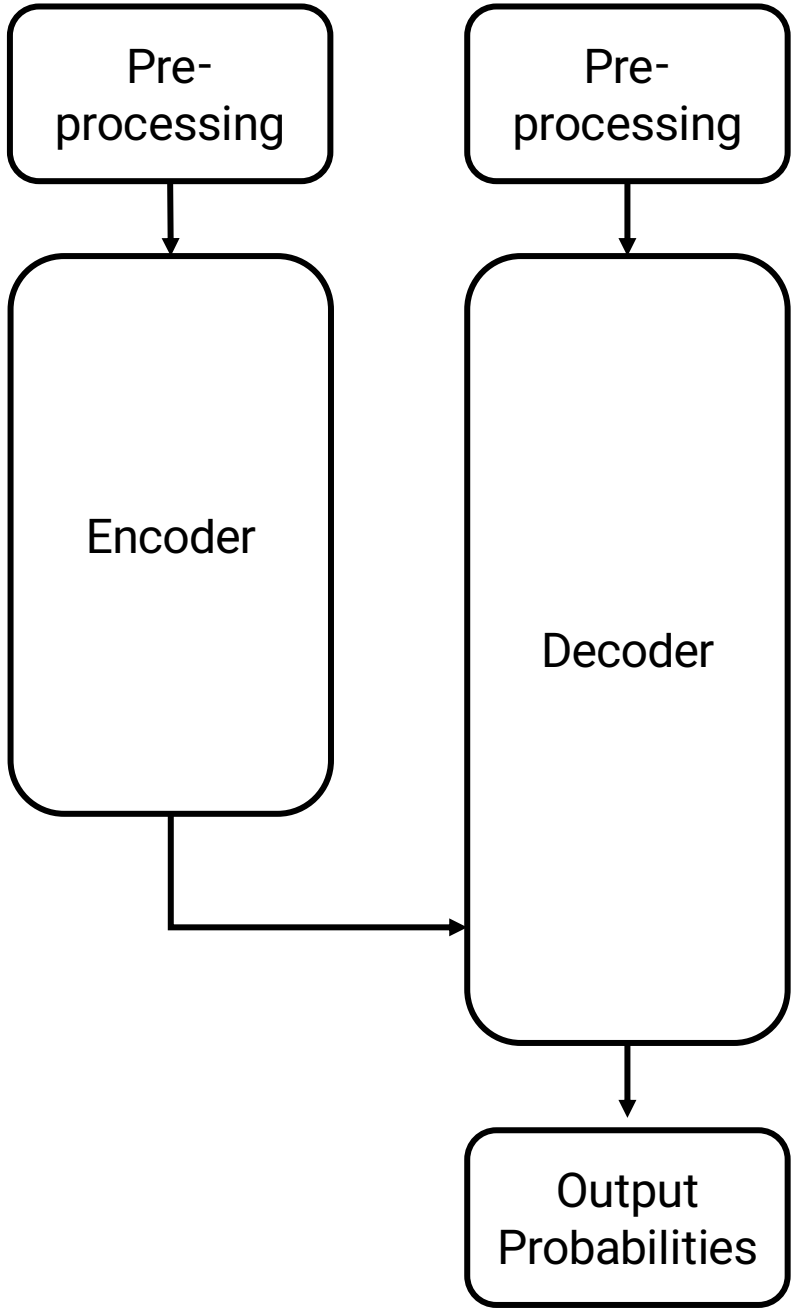
Transformer

Preview

Transformer는 입력된 정보를 동시에 펼쳐놓고
서로 간의 연관성을 확인하여 문맥을 재구성하는 구조입니다.

Encoder 에서는 입력 단어를 문맥을 포함하는 vector 데이터로 변환하고

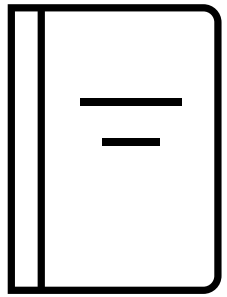
Decoder 에서는 Encoder의 출력을 근거로 단어의 확률을 예측합니다.



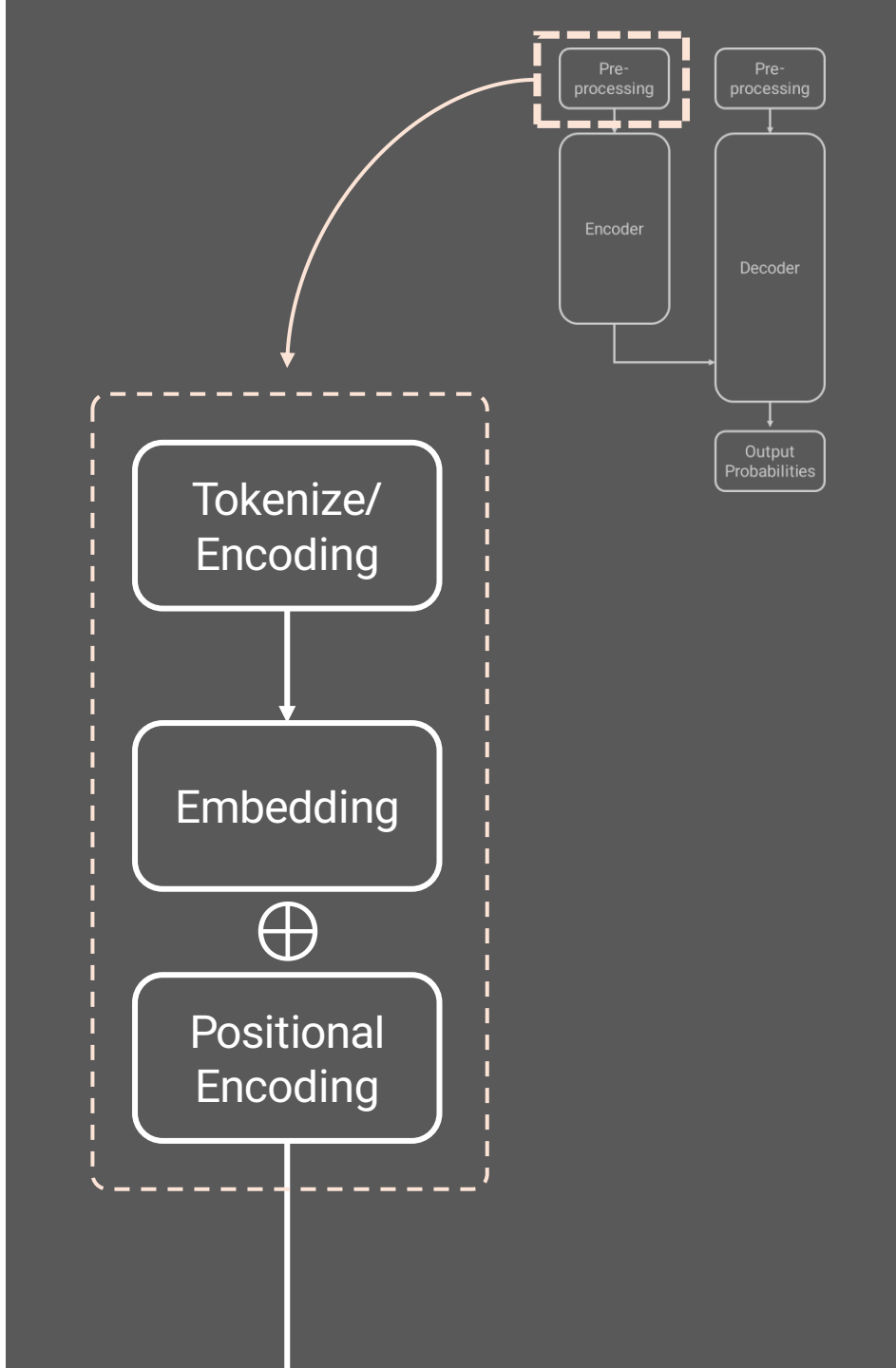
1. Preprocessing

전처리는 다음의 과정을 거칩니다.
문장이 입력으로 들어오면 해당 문장을 의미정보와 위치정보를 가진 Vector로 변환하는 과정입니다

해당 과정의 이전에, 단어 정보를 미리 저장해둔 단어장이 필요합니다.
이를 Vocab 이라고 합니다.

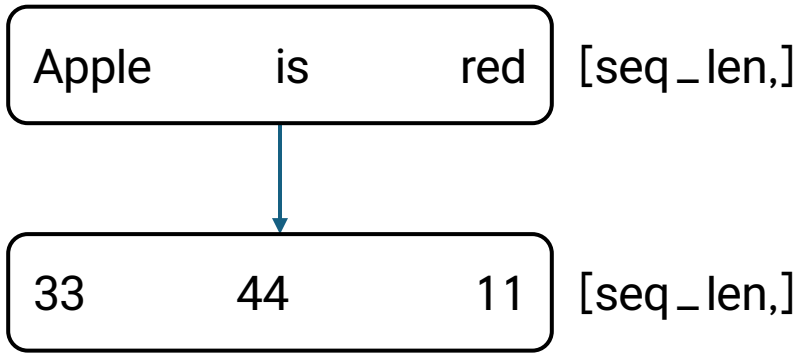


Vocab: 해당 논문에선 37,000의 크기를 가집니다

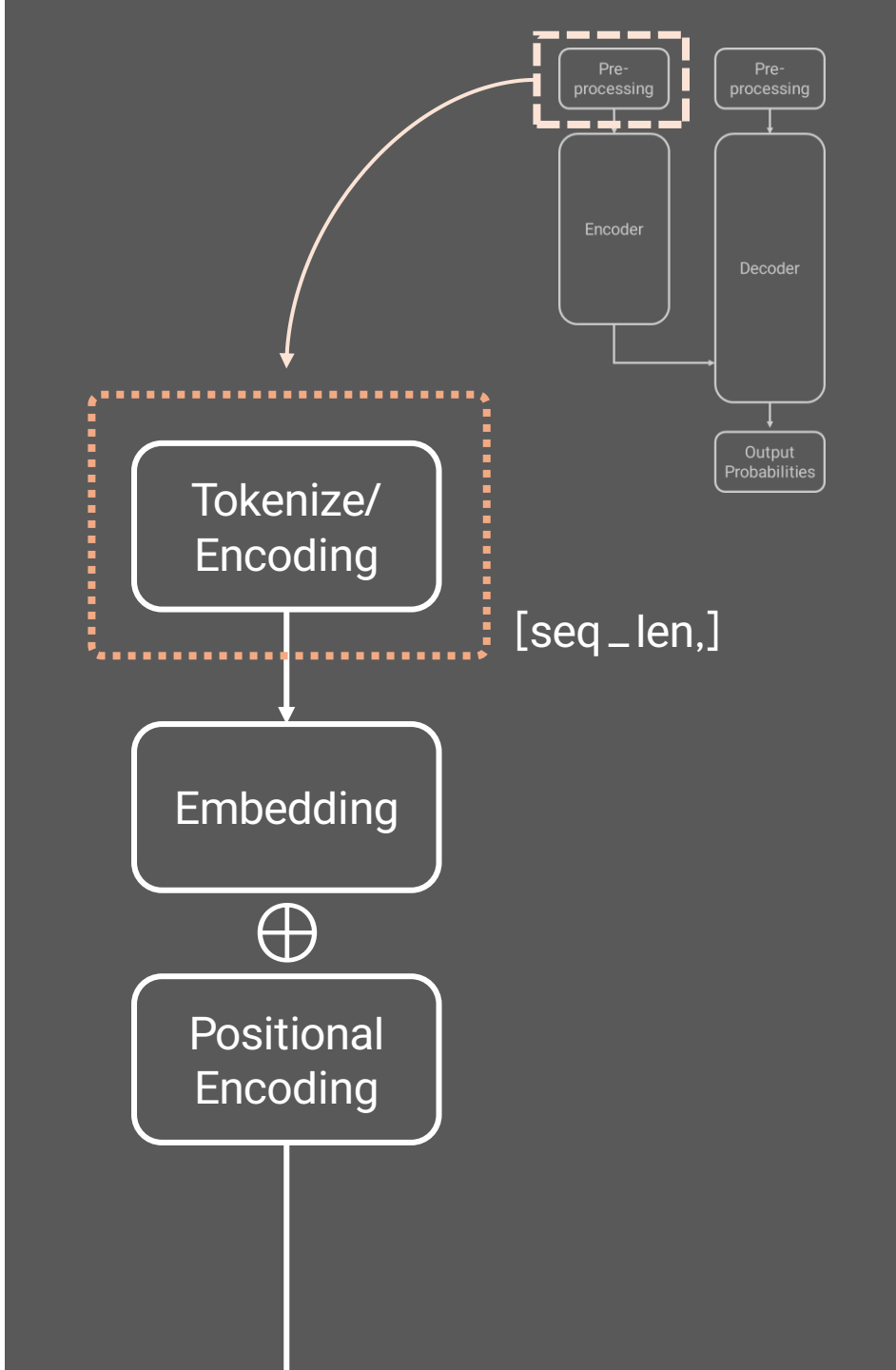


1-1. Tokenize

Tokenize에서는 기계가 이해할 수 있는 단위로 문장을 쪼개고,
Encoding 을 통해 고유한 Index를 부여합니다.



언어가 숫자로 Encoding 되었지만, 아직은 의미를 갖는 것은 아닙니다



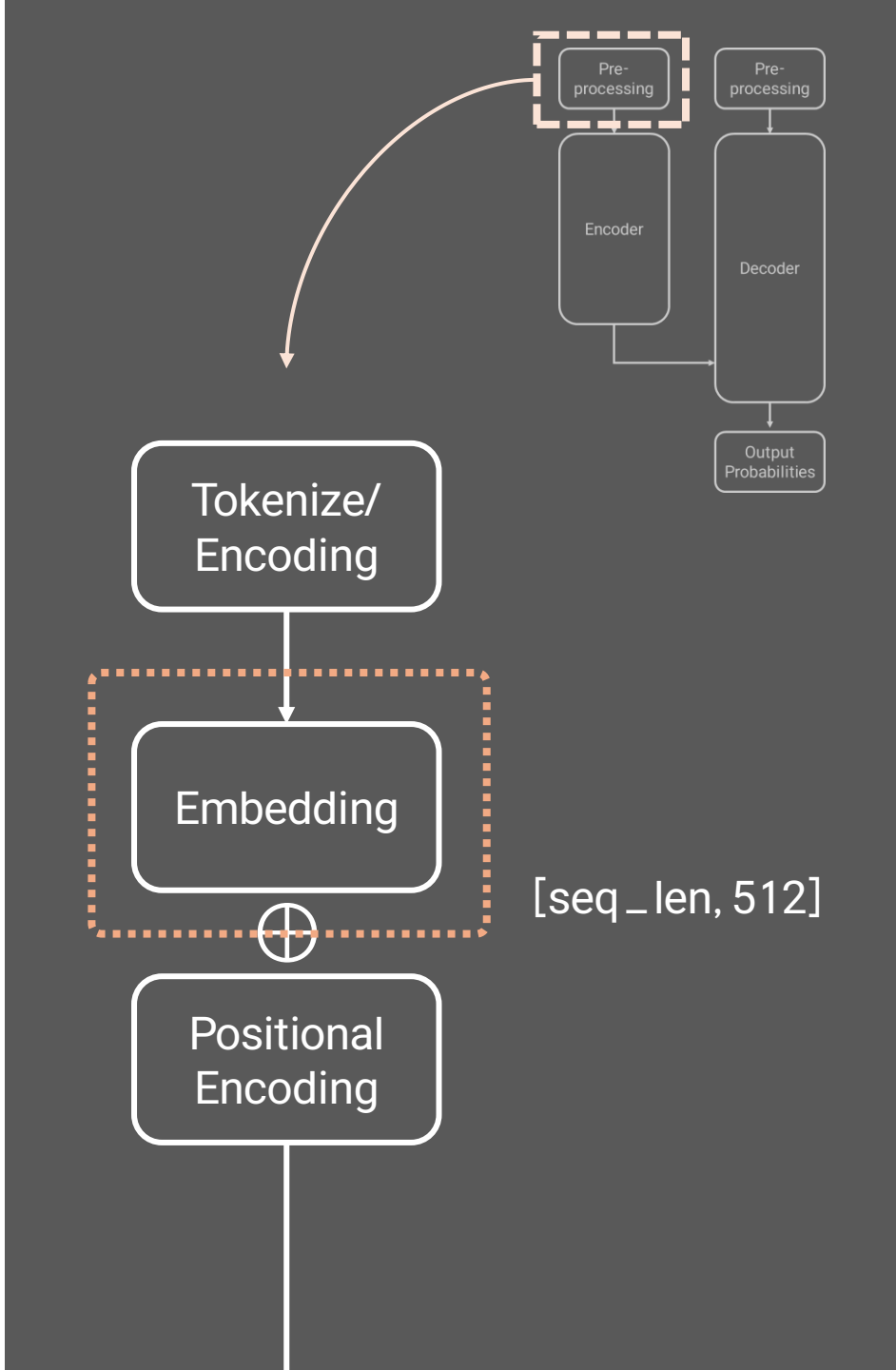
1-2. Embedding

Embedding은 의미정보를 담은 Vector로 변환합니다.
원래대로면 최초의 Linear layer로,
입력된 정보와 Embedding matrix간의
행렬연산을 의미합니다.

$$W_{embedding} \times X_{input} = X_{embedding}$$

Embedding matrix는 [vocab_size, model_dim] 으로 사전에 정의된 가중치행렬입니다. 현재는 초기화된 값이며, 학습을 하면서 점점 의미를 담도록 변합니다

X_input 은 자신의 index만 1인 one-hot vector입니다.
그래서 사실상 해당 인덱스의 열을 가져오는(look-up)
과정만을 거칩니다.



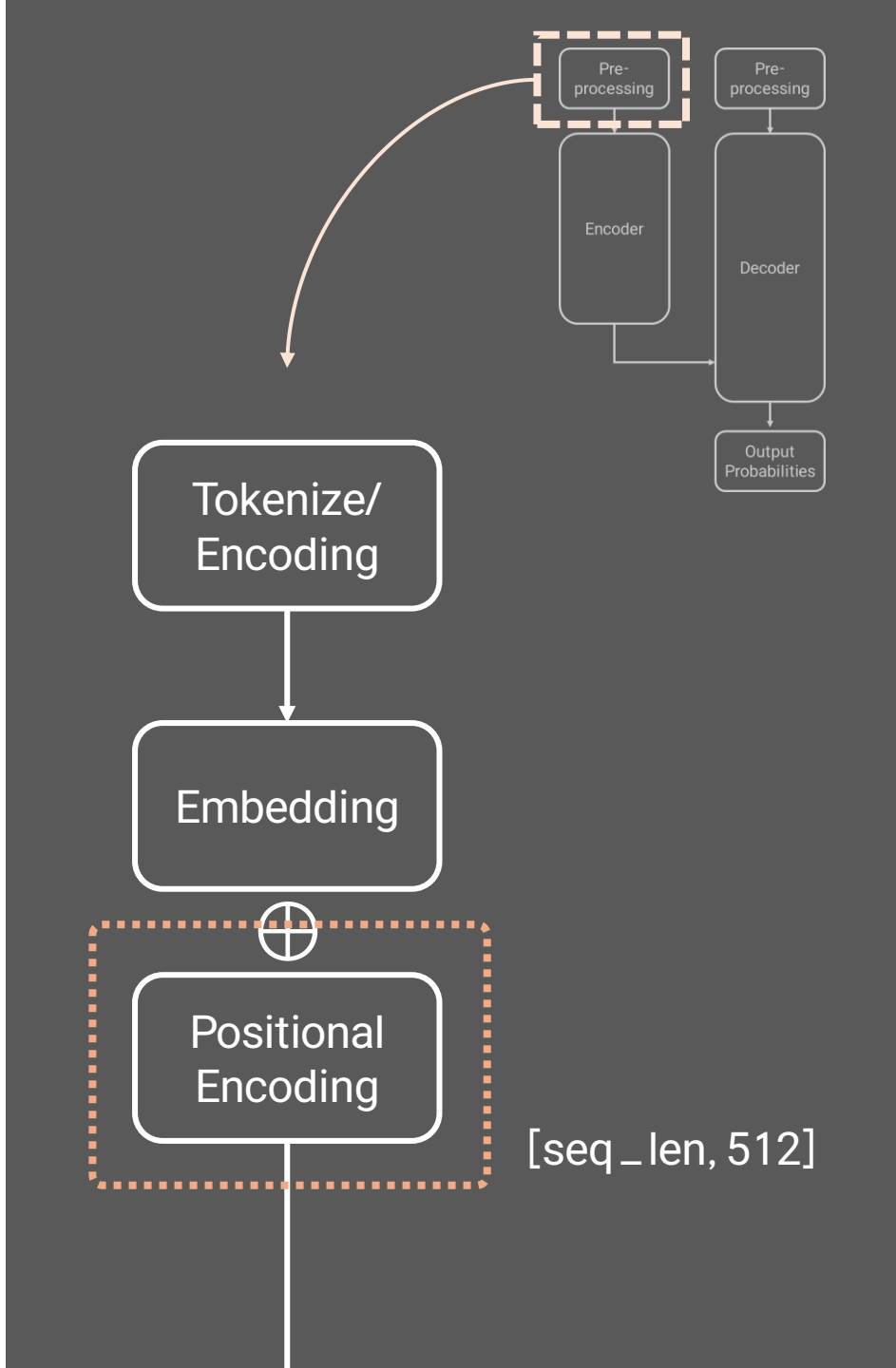
1-3. Positional Encoding

Vector의 위치정보를 입력합니다.
Sin/cos 공식을 이용하는데, 이렇게 하면 모델이 행렬연산을 통해서 단어들의 상대적인 위치를 구해낼 수 있기 때문입니다.

$$PE \oplus X_{embed} = X_{enc}$$

Concat 하지 않고 element-wise sum 을 하는 이유는, 연산의 효율성, Parameter를 늘리지 않기 위해서입니다.
데이터의 차원이 충분히 큰 관계로 현재 위치정보와 의미정보는 다른 부공간에 저장됩니다.

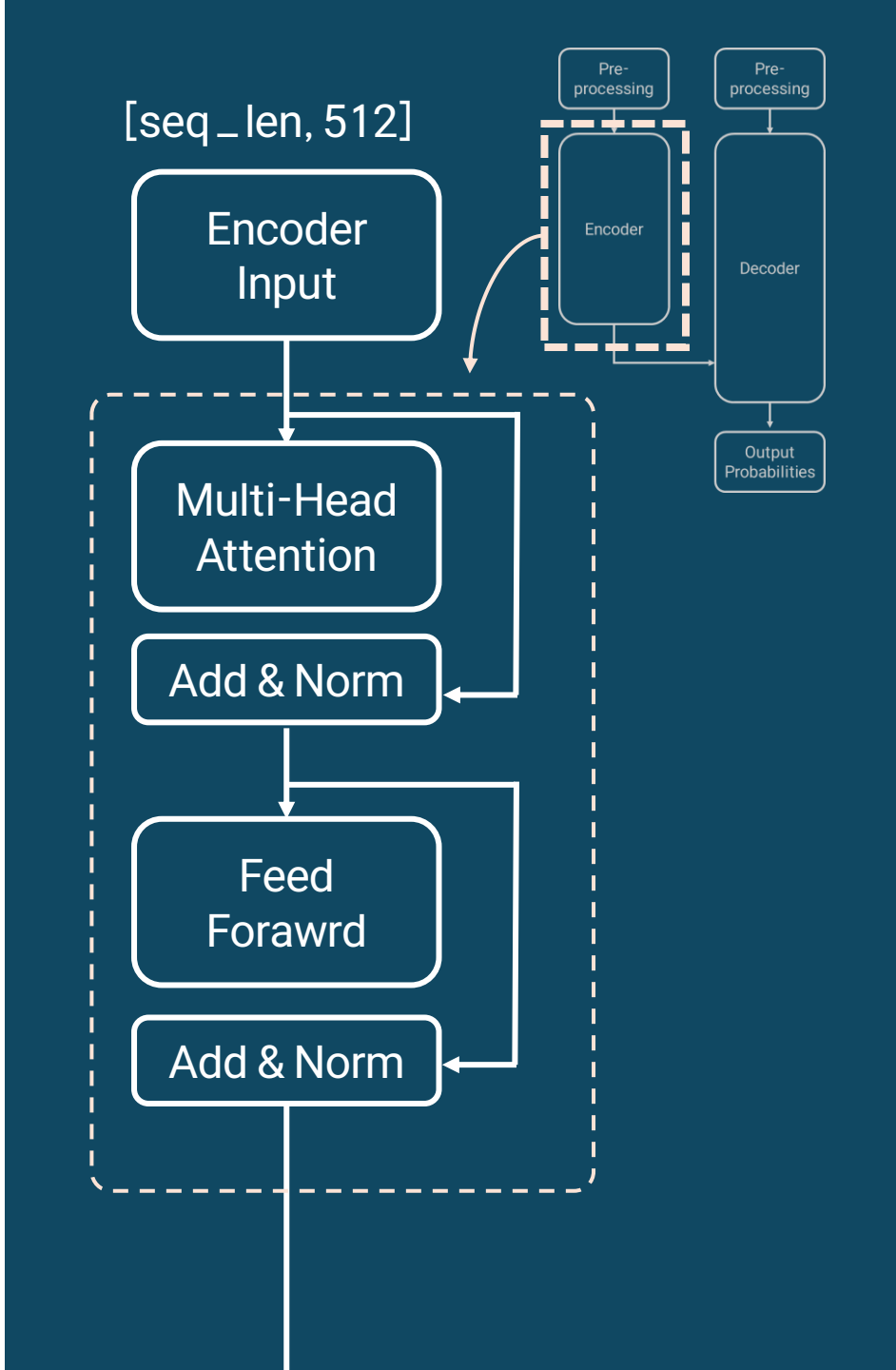
이렇게 해서 위치정보, 의미정보를 모두 포함하는 입력 Vector로의 변환이 완료되었습니다



2. Encoder

Encoder의 핵심은 문맥화 입니다.
입력된 독립적인 단어 Vector들에 서로 간의 연관성을 파악하여
해당 정보를 가진 Vector로 변환합니다.
나중에 Decoder 가 이 결과물을 참조합니다

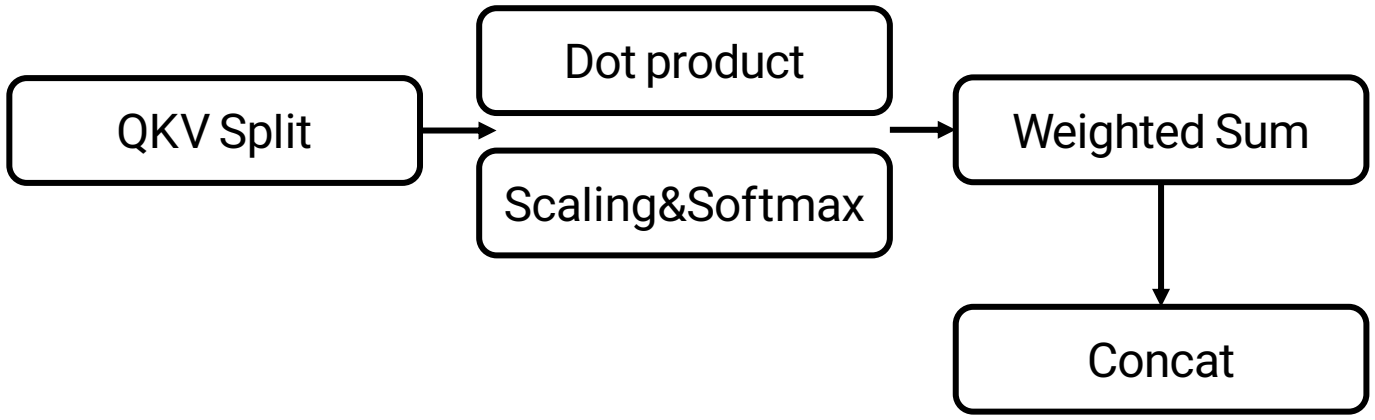
각 단계에서의 Input 이 Output에 더해지는
Add & Norm 의 과정이 SubLayer 마다 있습니다.

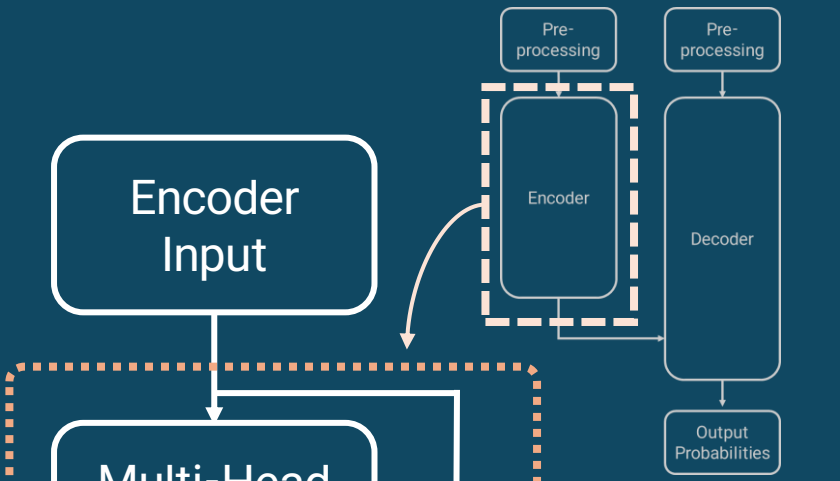


2-1. Multi-Head Self Attention

Self:문장 내 단어들 간의 연관성을 구하기 위한 단계입니다.
Multi-Head: 이때 각자의 분류에 특화된 Heads 가 있으며,
한번에 처리하는 대신 각자의 부분공간으로 쪼개 병렬 처리합니다.

이하의 과정을 거칩니다.





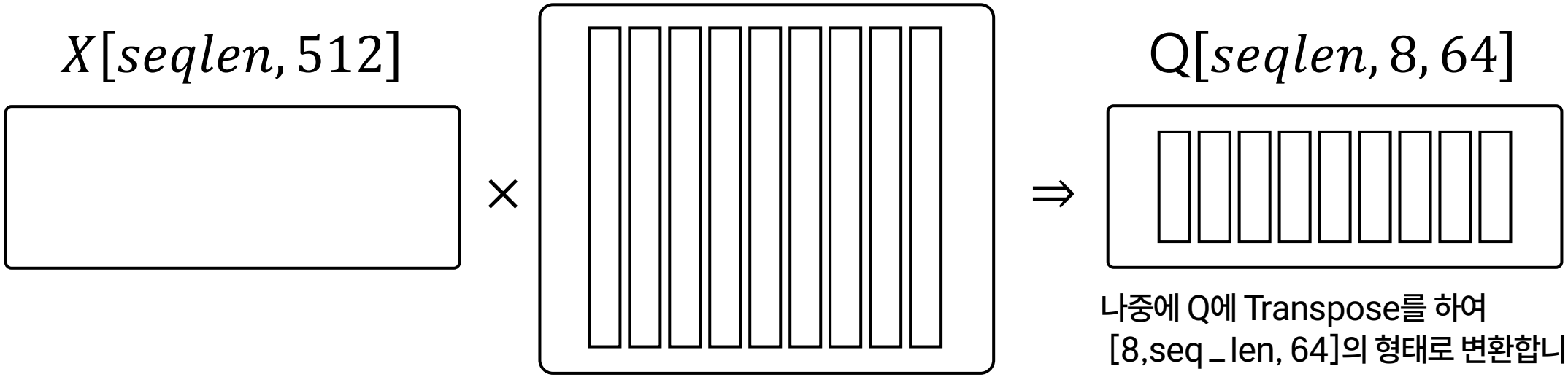
[8, seq _ len, 64]

1) QKV Split

행렬연산 자체에서는 2차원 행렬입니다.
연산이 끝나고 마지막에 이를 3차원 텐서로 변환합니다.

$W_Q[512, 512]$

$X[seqlen, 512]$



$\times$

$\Rightarrow$

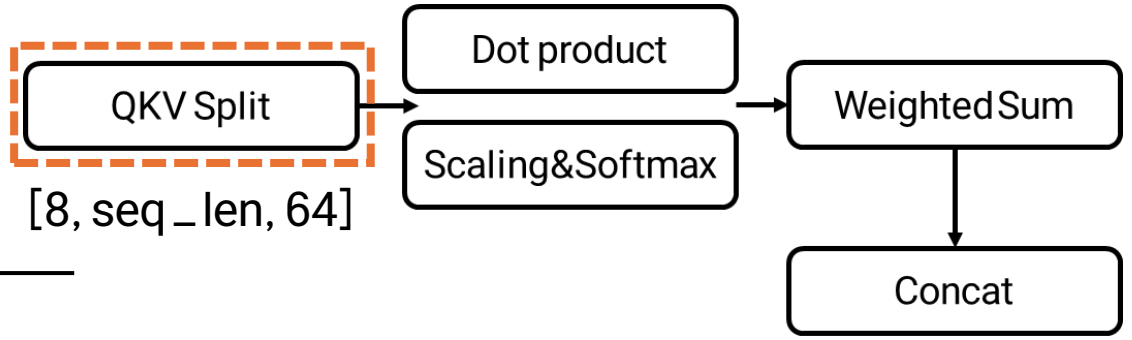
$Q[seqlen, 8, 64]$

나중에 Q에 Transpose를 하여
[8,seq_len, 64]의 형태로 변환합니다.

Q를 만들기 위한 가중치행렬 W_q,
이는 각기 다른 의미분석에 특화된 W_1 에서 W_8으로 나뉜 형태입니다.

개념적으로는 512차원을 64차원으로 줄여주는 작은 행렬이 8개 존재하지만,
실제 연산 효율을 위해 이들을 병렬적으로 처리하기 위해서 3차원 텐서 로 관리합니다.
이는 실제 연산을 할 때 8개의 부공간을 만들어서 해당 공간에서 연산하는 것과 같습니다.

1) QKV Split

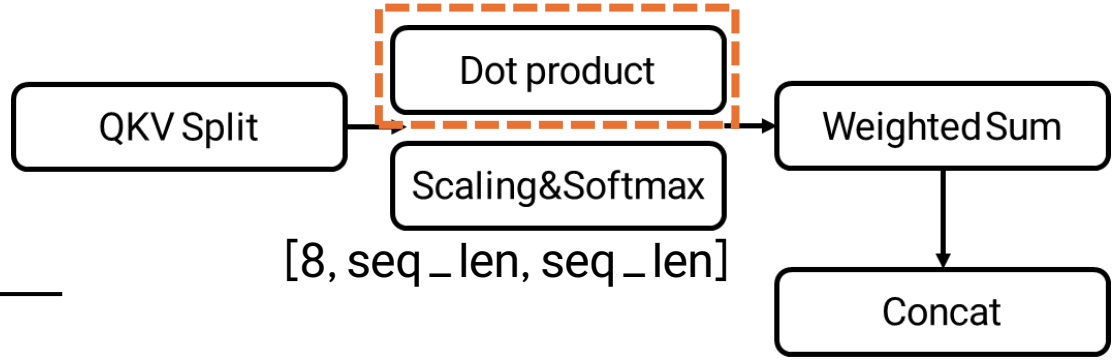


$$Q = X W_Q, \quad K = X W_K, \quad V = X W_V$$

Attention 연산을 위한 분할이 모두 끝났습니다.
여기서 가중치행렬 W_q, W_k, W_v는 각각 다음의 역할을 합니다.
학습을 반복하면서, 해당 방향으로 가중치를 조절합니다.

- W_Q: 탐색을 위한 Vector, K와의 내적이 최대화하는 방향으로 회전합니다.
- W_K: 검색되기 위한 Vector, Q와의 내적이 최대화하는 방향으로 회전합니다.
- W_V: 실제 value를 담은 Vector, 본연의 의미를 잘 보존하도록 변환됩니다.

2) Dot product

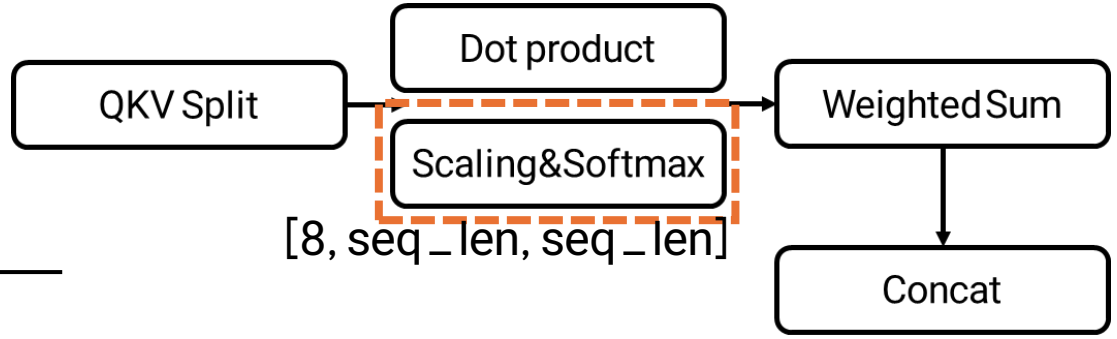


$$Q \times K^T$$

다음으로 Q와 K의 내적을 통해 두 벡터 사이의 각도를 계산합니다.
이는 내가 찾는 정보 Q가 K 속에 얼마나 포함되어있는지를 묻고,
단어들 사이의 연관성을 수치화 하는 과정입니다.

연산 과정을 거쳐서 단어들 간의 관계를 나타내는
Attention score 이 생성됩니다.

3) Scaling & Softmax

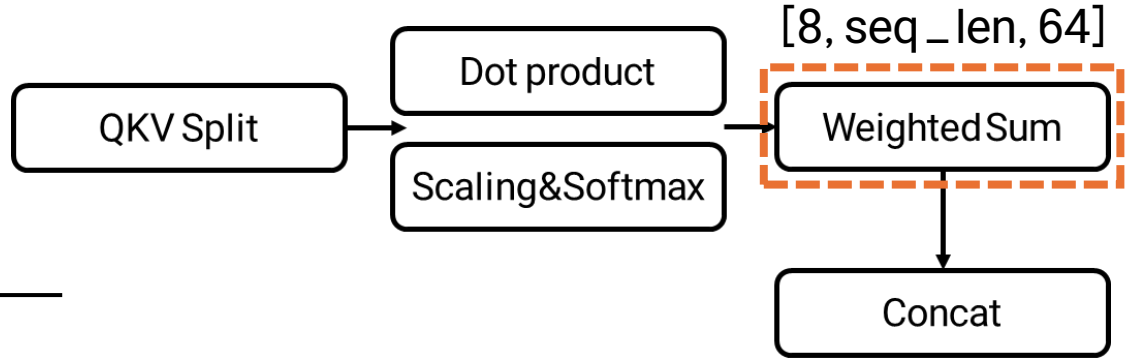


$$\left(\left(Q \times K^T \right) \frac{1}{\sqrt{dim}} \right) softmax = \underset{[8,\ seq\_len,\ seq\_len]}{\text{Attention weight}}$$

연관성의 비중을 나타내는Vector로 변환합니다.
여러 단어의 정보를 합칠 때 어떤 단어와 깊게 연관되어 있는지 그 반영 비율을 결정하는 분포입니다.

다만, 이전의 결과를 그대로 사용하면 Softmax의 특성상 값이 극단적으로 변할 수 있습니다.
따라서 표준편차로 나누는 scaling 을 실행합니다.

4) Weighted Sum

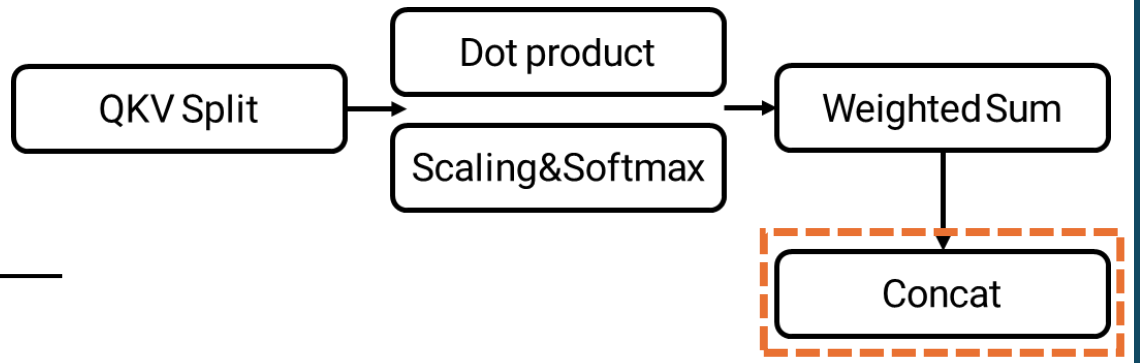


$$\text{Attenion\_weight} \times V$$

[8, seq_len, seq_len] * [8, seq_len, 64]

앞서 구한 가중치를 이용해 V의 weighted sum 을 구합니다.
중요한 단어 정보는 많이, 안 중요한 단어 정보는 적게 가져와서
문맥 정보가 반영된 Context Vector로 업데이트합니다.

5) Concat



$$Concat(H_1, \ldots, H_8) = \text{Output}$$

[8, seq _len, 64] => [seq _len,512]

$$\text{Output} \times W_O$$

[seq _len, 512]* [512, 512]

분할된 정보들을 하나로 Concat(reshape) 하고, 가중치 행렬 W_o를 곱해서 최종차원으로 변환합니다.
8개의 헤드로 파악한 문맥 정보를 하나의 결과물로 취합하고 융합하는 단계입니다.

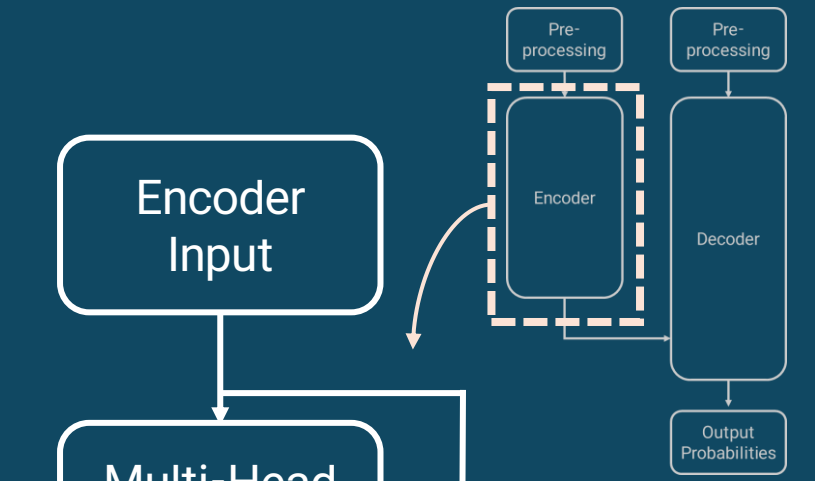
2-2. Add & Norm

Sublayer 사이에는 Residual Connection 과 Layer Normalization을 시행합니다.

입력으로 들어왔던 X를 그대로 Attention 결과에 더해서 backpropagation 때의 Vanishing Gradient 문제를 방지합니다.

문장 데이터는 길이가 들쭉날쭉하기 때문에 Batch Norm 문제는 통계가 왜곡될 위험이 있습니다.
따라서 layer의 분포를 일정하게 맞춰 학습 안정성을 높입니다.

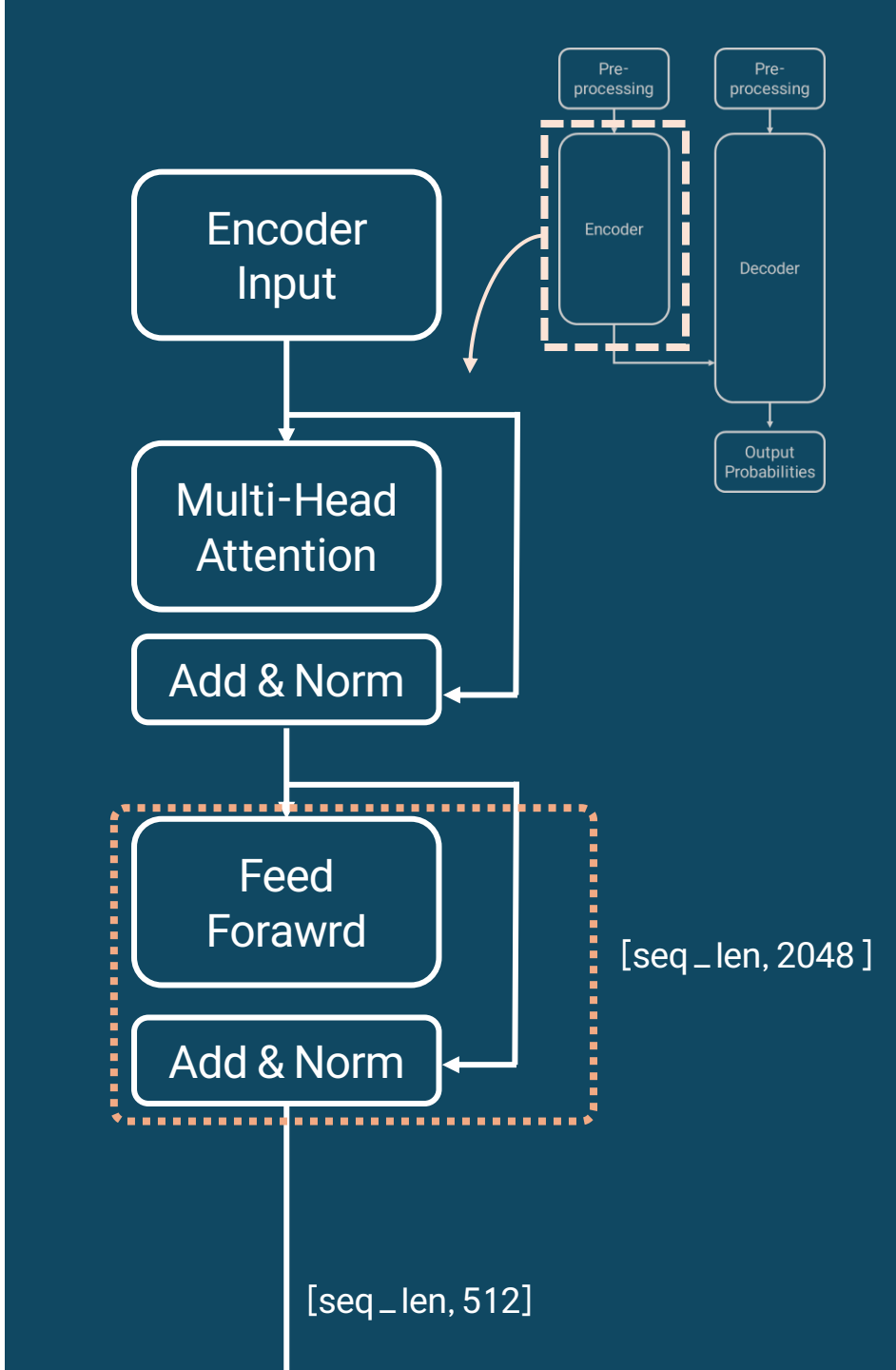
$$\text{Output} = \text{LayerNorm}(X + \text{SubLayer}(X))$$



2-3. FeedForward

Expansion 을 통해 차원을 확장하여 의미를
더 세부적으로 구분합니다.
중간에 Activation 을 거치면서 음수(불필요한 정보)를
지워버립니다.
Compression 을 통해서 문맥에 맞는 핵심 의미만 정제하여 원래의 크기로 내보냅니다.

이후 해당 과정을 반복하면서, 문맥에 대한 표층적인 이해에서 나아가서 깊은 이해를 할 수 있습니다.

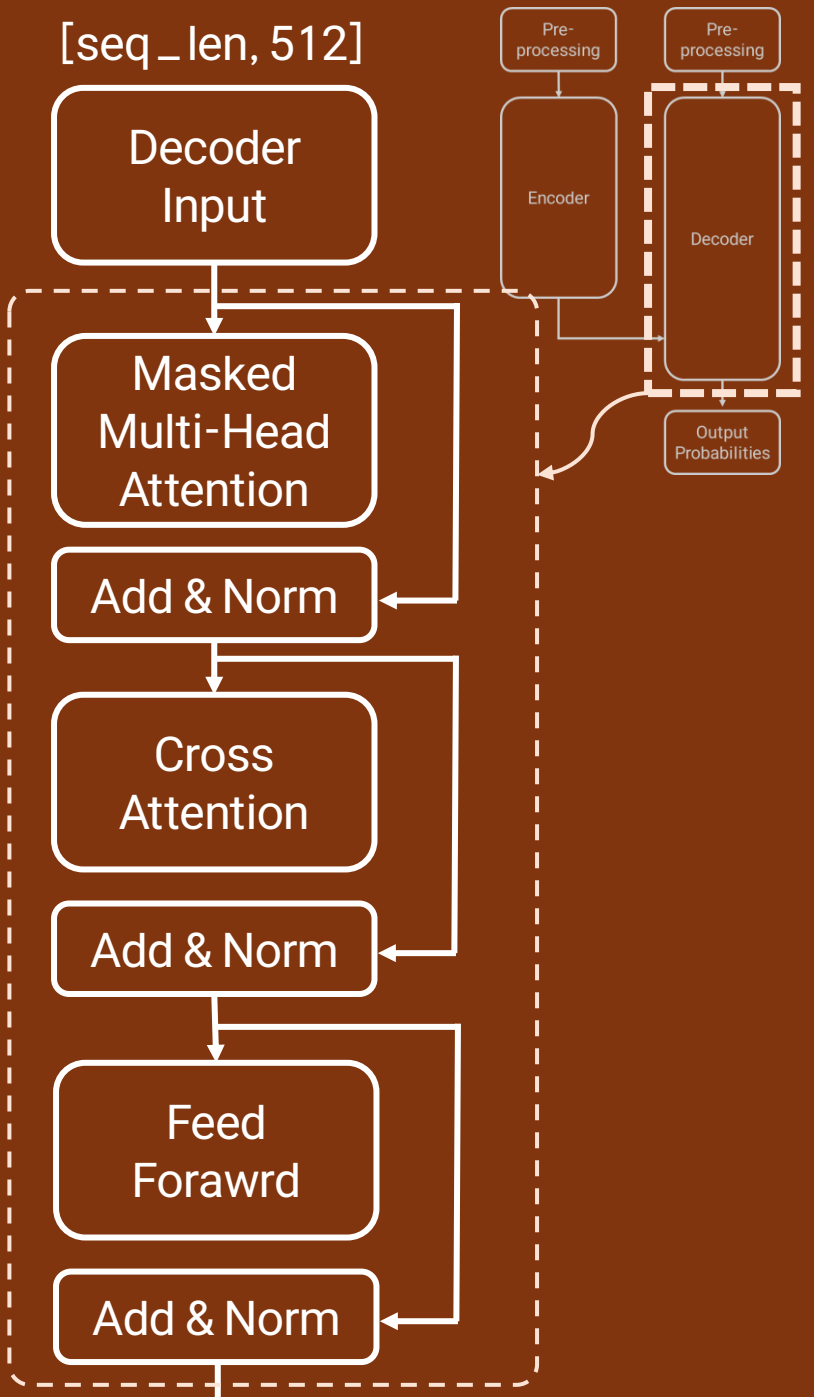


3. Decoder

Decoder는 인코더가 압축해둔 원본을 바탕으로
번역문을 한 단어씩 생성하는 역할을 합니다.

한 번에 문장을 만드는 것이 아니고 이전 단어를 보고 다음 단어를
하나씩 예측하는 Auto Regressive 단계입니다.

Encoder input 의 번역 결과가 들어가게 됩니다.
모델에게 정답을 미리 다 보여줄 수 없으므로 따라서 문장 맨 앞에
<BOS>(Start of Sentence)라는 시작 토큰을 붙여서, 전체
문장을 오른쪽으로 한 칸 밉니다.

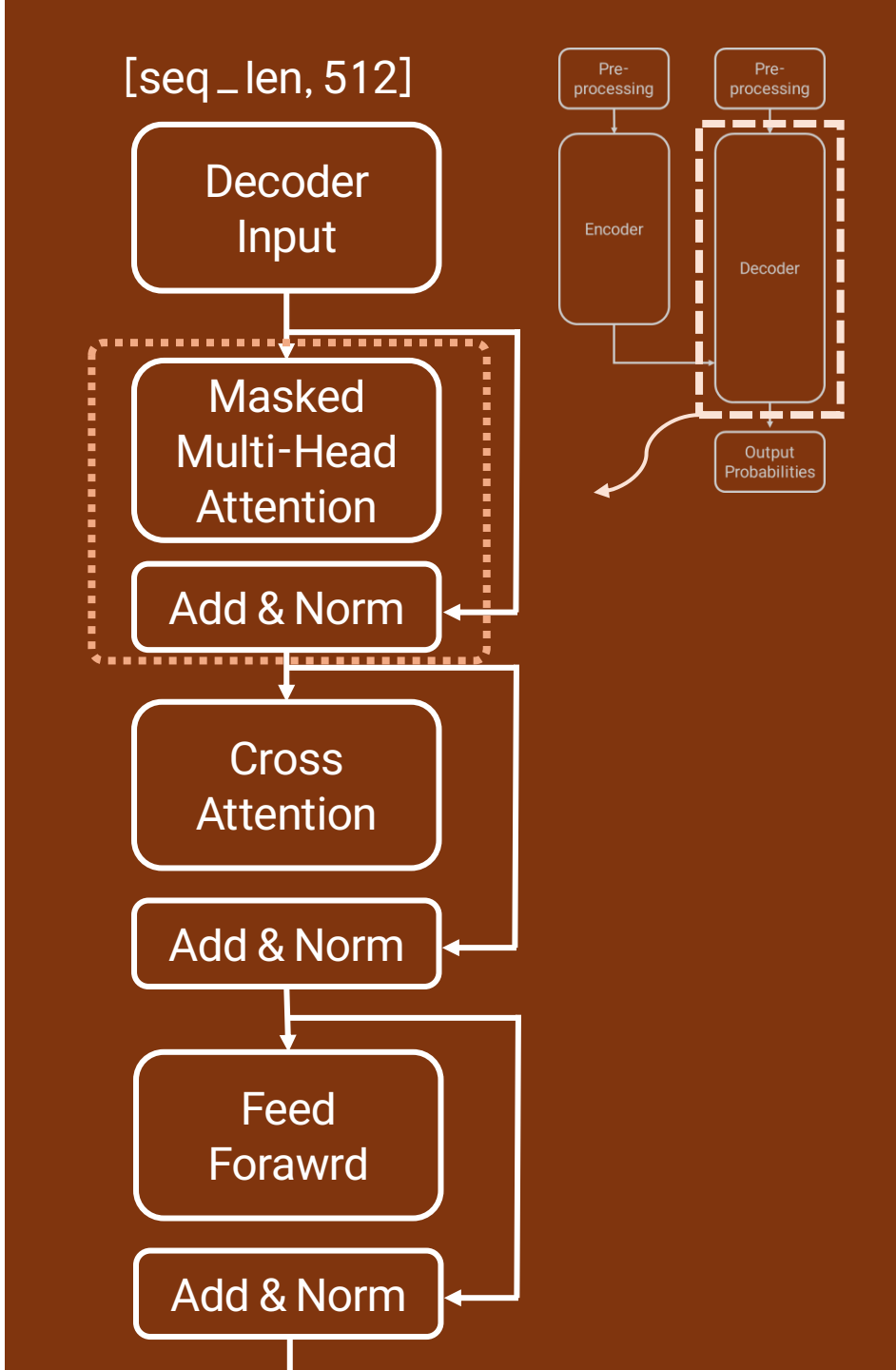


3-1. Masked-Multi-Head self Attention

자기 자신의 문맥을 파악하되, 미래의 정답을 보고 베끼는 행위를 방지하는 Masking이 이뤄집니다.

Q, K, V를 Split 하고 Q와 K의 dot product를 통해 attention score를 구하고 scaling 을 합니다.

$$\left((Q \times K^T) \frac{1}{\sqrt{dim}} \right)$$



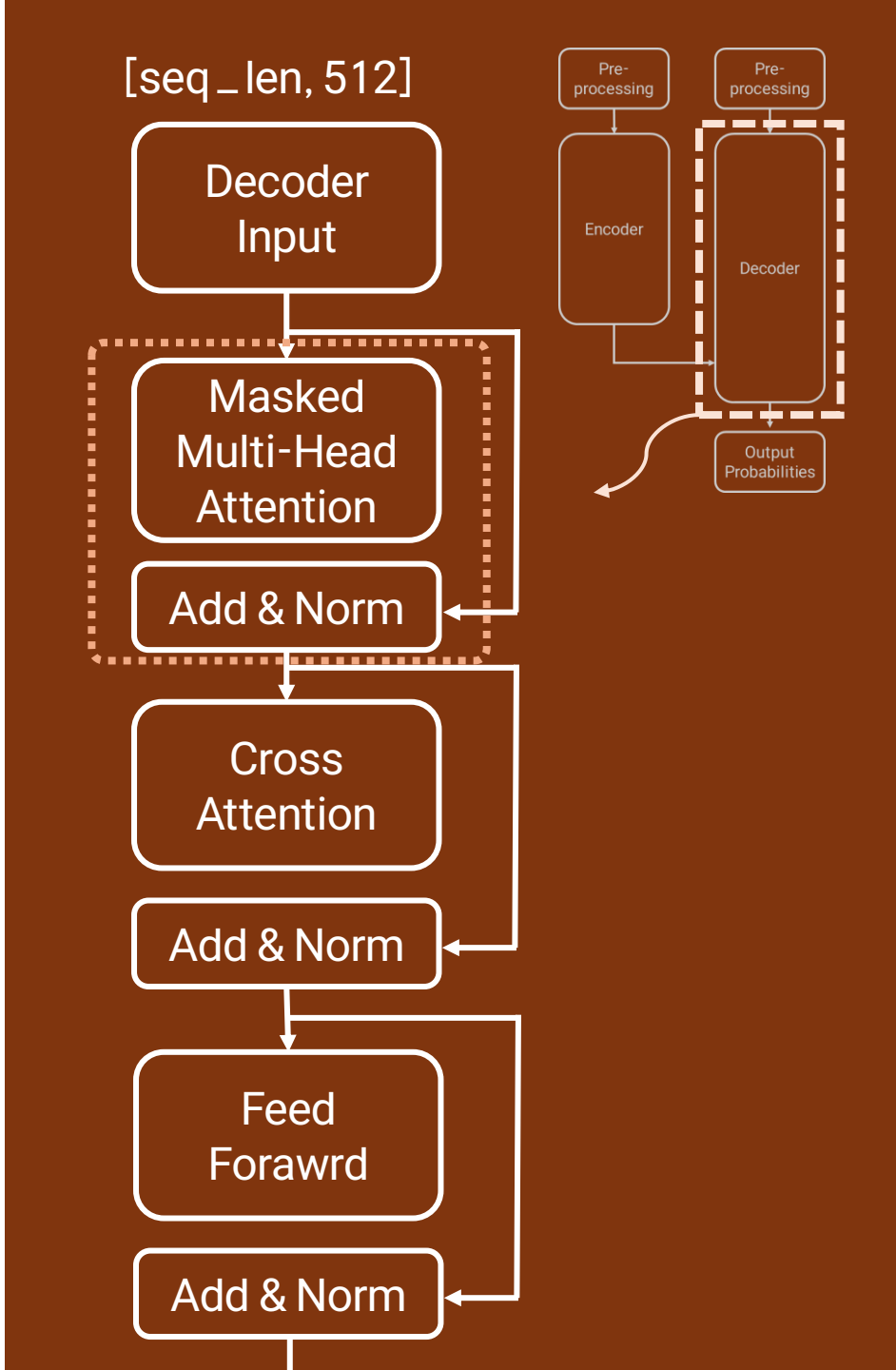
3-1. Masked-Multi-Head self Attention

Masking 에 앞서 Causal Mask를 준비합니다.
M의 대각성분 아래쪽은 0, 위쪽은 $-\infty$ 인 행렬입니다.

[seq_len, 512] : softmax의 공식상 e^0을 하면 1이 되어버림

이는 과거의 단어만 참고하여 문맥을 파악하도록 하기 위해,
순서 상 미래의 단어들을 가려버리는 역할을 합니다.
이후에 입력된 Vector와 덧셈연산을 합니다.

$$\left(\left(\left(Q \times K^T\right)\frac{1}{\sqrt{dim}}\right) + \mathrm{M}\right) softmax$$

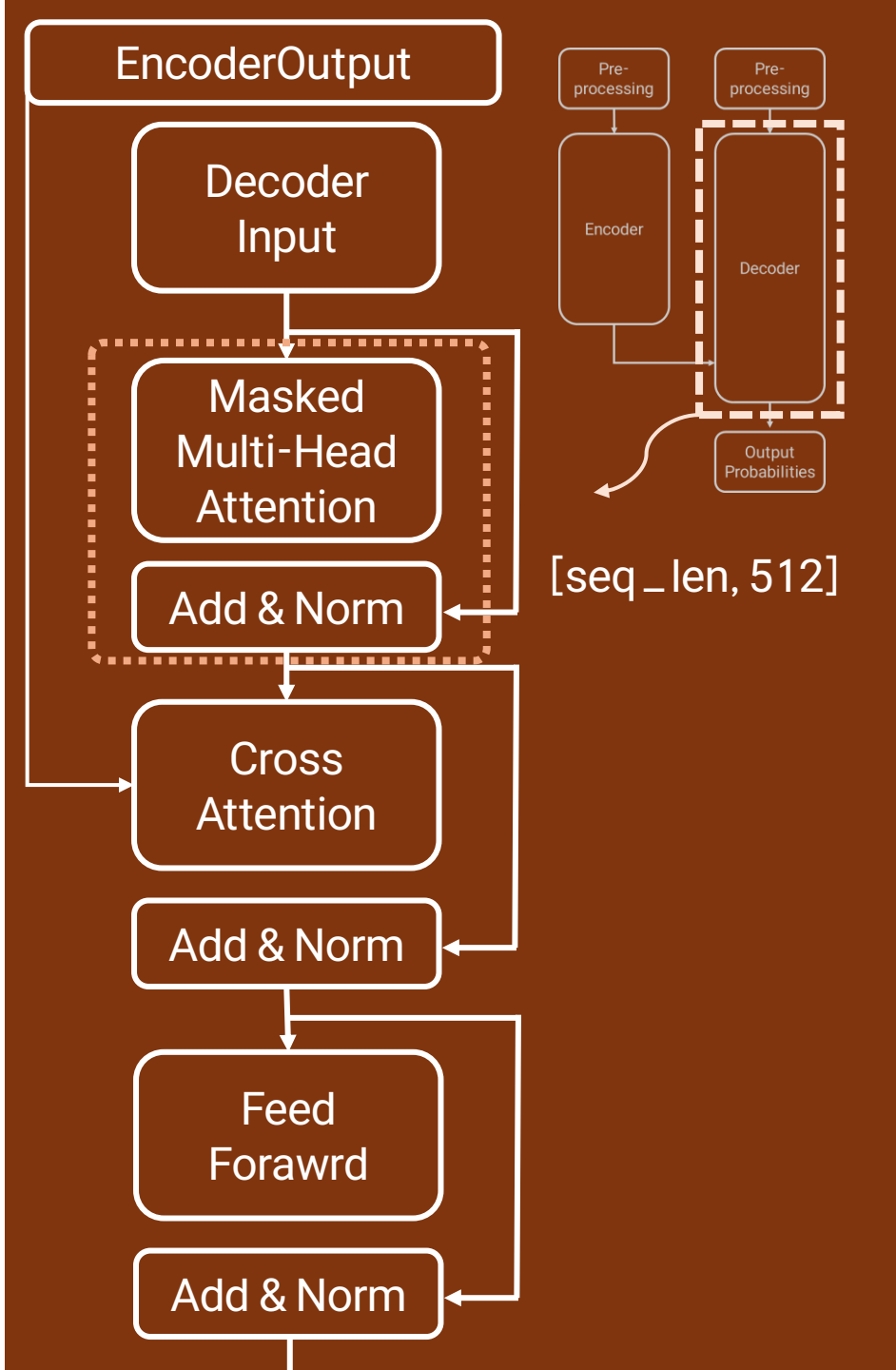


3-2. Masked-Multi-Head self Attention

준비된 attention weight 에 V를 곱해
다음에 올 단어를 찾기 위한 단서를 담고 있는
context vector를 구성합니다.
디코더는 이 정보를 단서로 다음 토큰이 무엇일지 추론하게 됩니다.

이후 concat => W_o를 곱해 정보를 혼합하고 차원을 정리합니다.

decoder input과 attention 의 결과를
Add & norm 하여 다음 단계로 넘깁니다.



3-3. Cross Attention

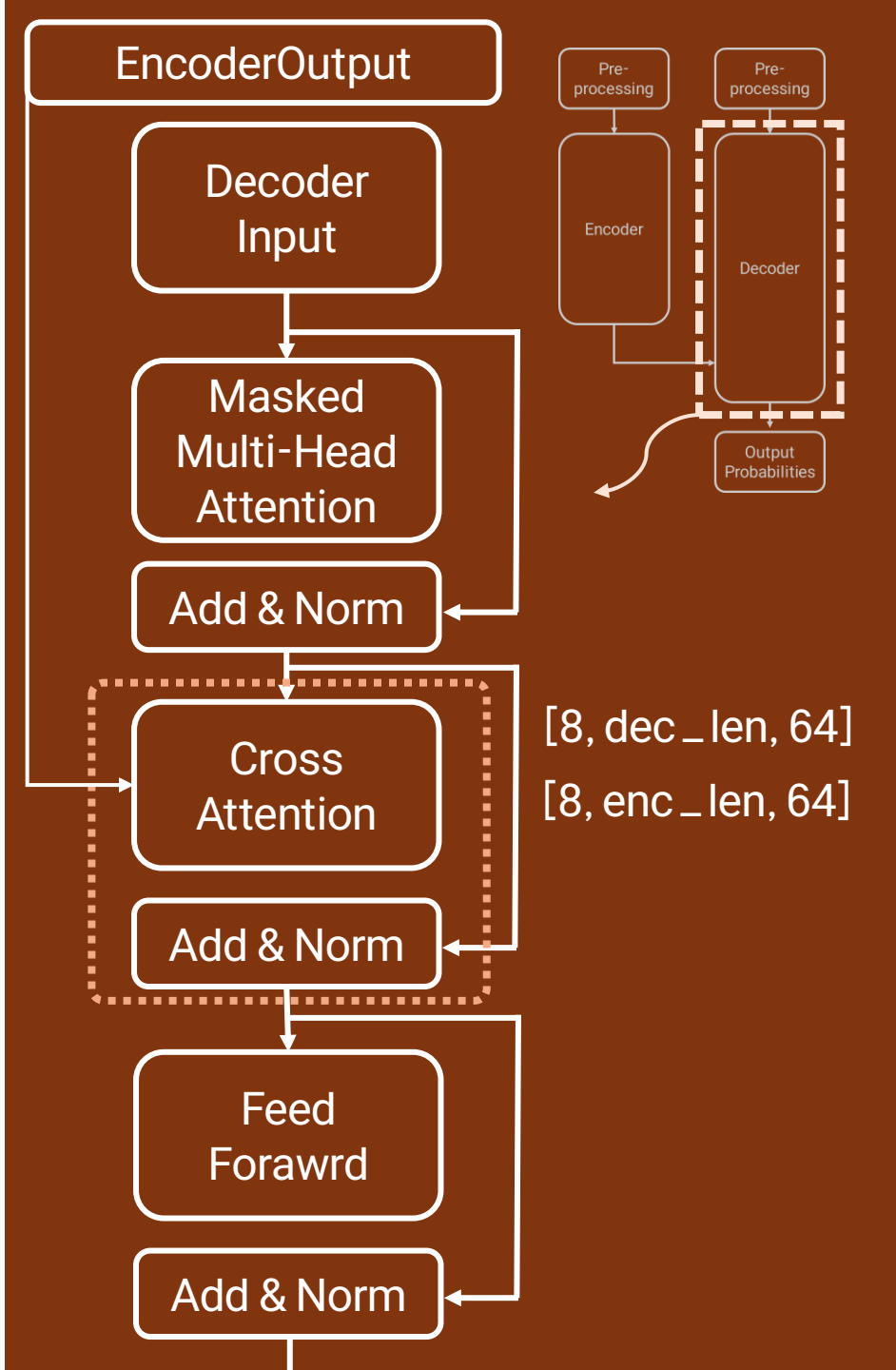
Decoder가 encoder가 보낸 원문 데이터를 보고, 원문에서 단서를 찾아오는 과정입니다. Q는 decoder 에 있지만,
K와 V는 원문에서 가져와 다음 단어를 생성할 준비를 마칩니다.

- Q: 번역을 수행 중인 decoder
- K & V: 분석이 끝난 Encoder output

$$X_{dec} \times W_Q \quad [dec\_len, 512]* [512, 512]$$

$$X_{enc} \times W_{K,V} \quad [enc\_len, 512]* [512, 512]$$

이후 Head 만큼 split을 해줍니다.



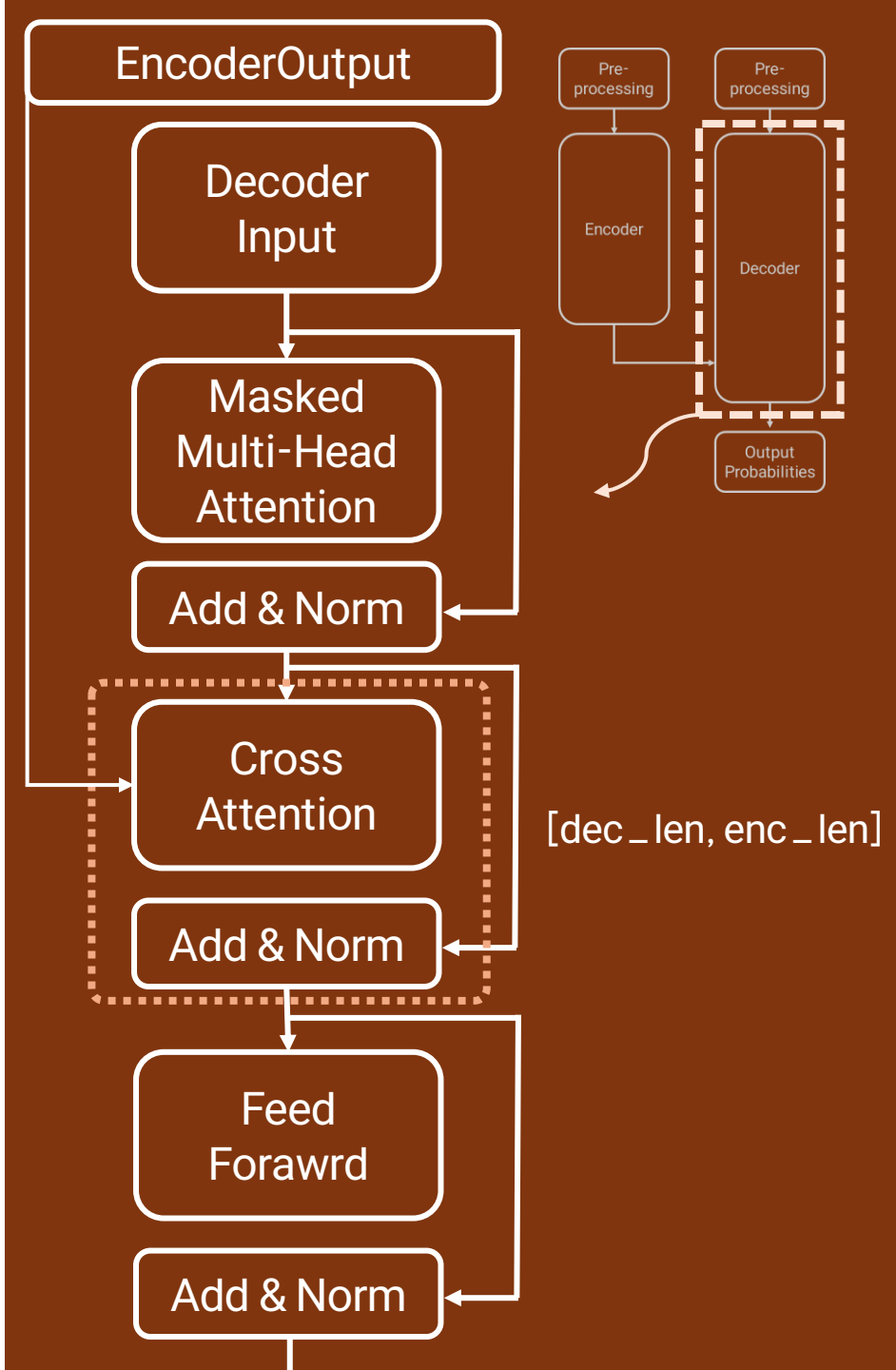
3-3. Cross Attention

Decoder의 Q와 Encoder K의 Dot product를 실행해 현재 보는 Q가 원문의 어느 부분과 연관되는지 상관관계를 파악하고, Decoder에서 생성하려는 단어마다 Encoder의 모든 시점에 대한 유사도점수를 파악합니다.

$$Q \times K^T = Attention\ score$$

[dec_len, 512]* [512, enc_len]

이때 나오는 Attention score는 [dec_len, enc_len]의 형태입니다.



3-3. Cross Attention

Decoder 가 가진 현재의 문맥과 Encoder가 제공하는 원문의 정보가 융합되어, 새로운 Context vector를 생성합니다.
문장 내의 단어 맥락이 아니라, 다음에 생성할 단어에 원문의 깊은 맥락까지 입힌 상태입니다.

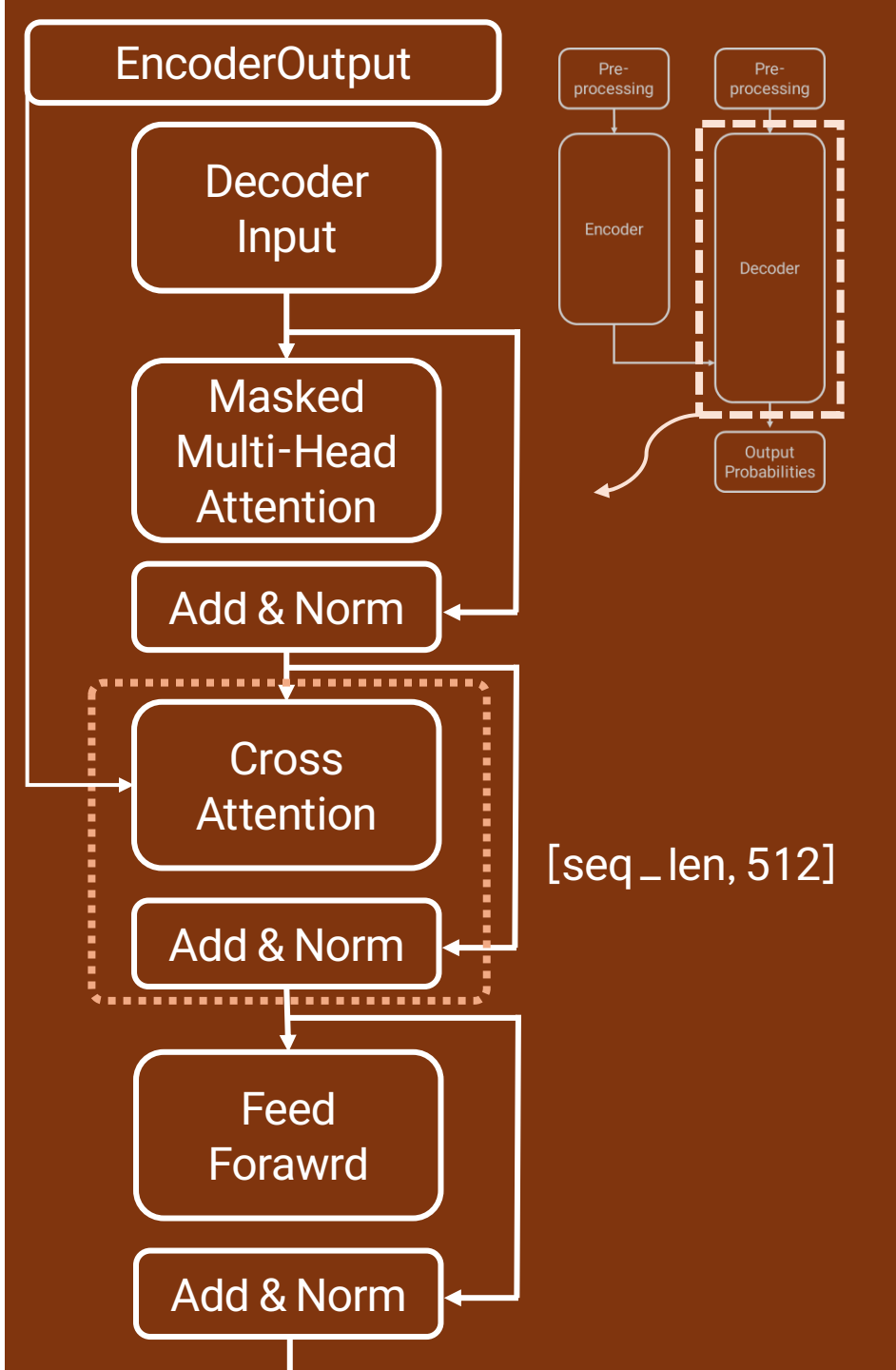
$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \times V = Z$$

[dec_len, enc_len] * [enc_len, 512]

[seq_len, 512]

concat => W_o를 곱해정보를 혼합하고 차원을 정리합니다.

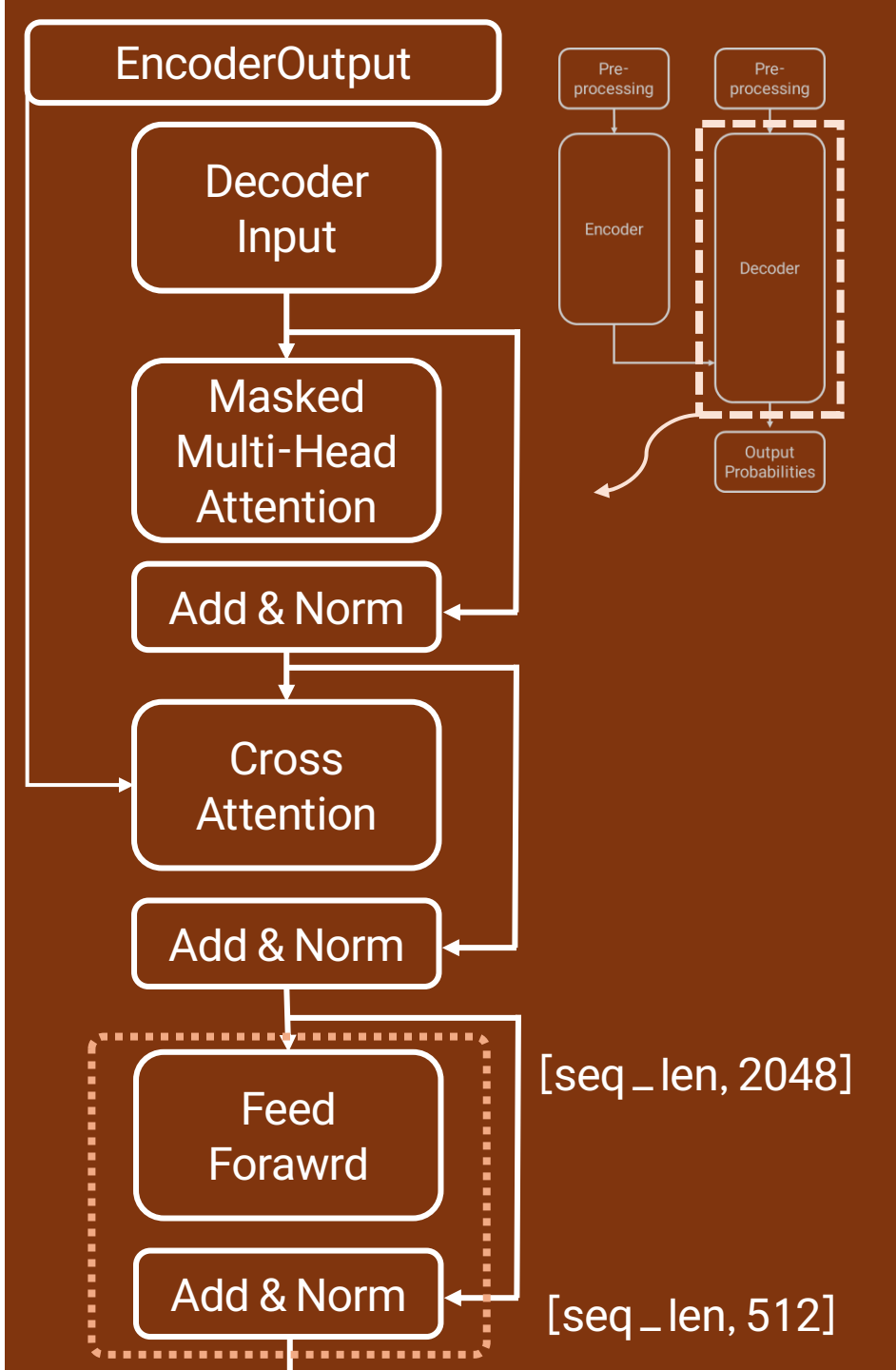
decoder input과 attention 의 결과를
Add & norm 하여 다음 단계로 넘깁니다.



3-4. Feed Forward

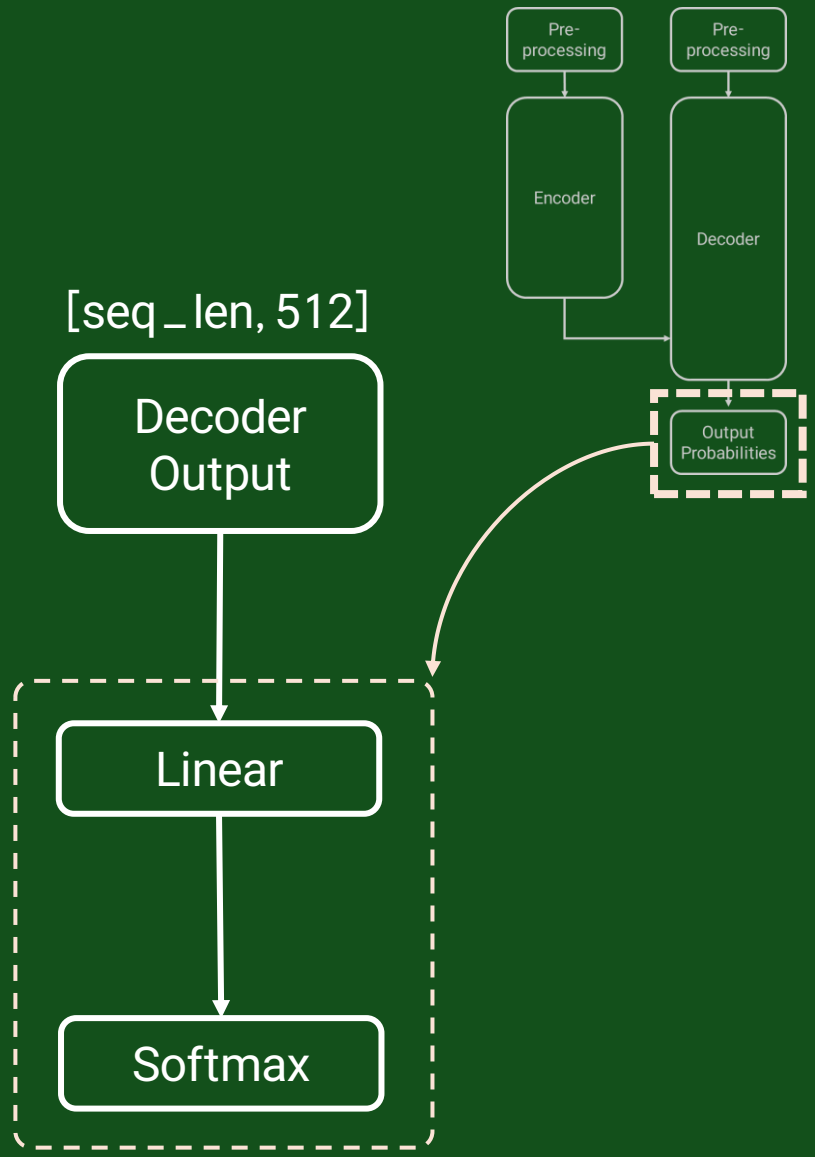
[seq _ len,512] 크기의 vector를 2048차원으로 expansion 하여 특징을 더욱 세밀하게 분해합니다.

ReLU를 통해 불필요한 정보(음수)를 없애며 비선형성을 추가하고, 다시 원래의 차원으로 압축하는 Compression 을 거쳐서 다음 단어 예측에 최적화된 정보만 남깁니다.



4. OutputProbabilities

Decoder Output을 실제 Vocab 의 단어로 match 하는 과정이 필요합니다.
즉 Vector 를 word로 바꾸는 과정입니다.

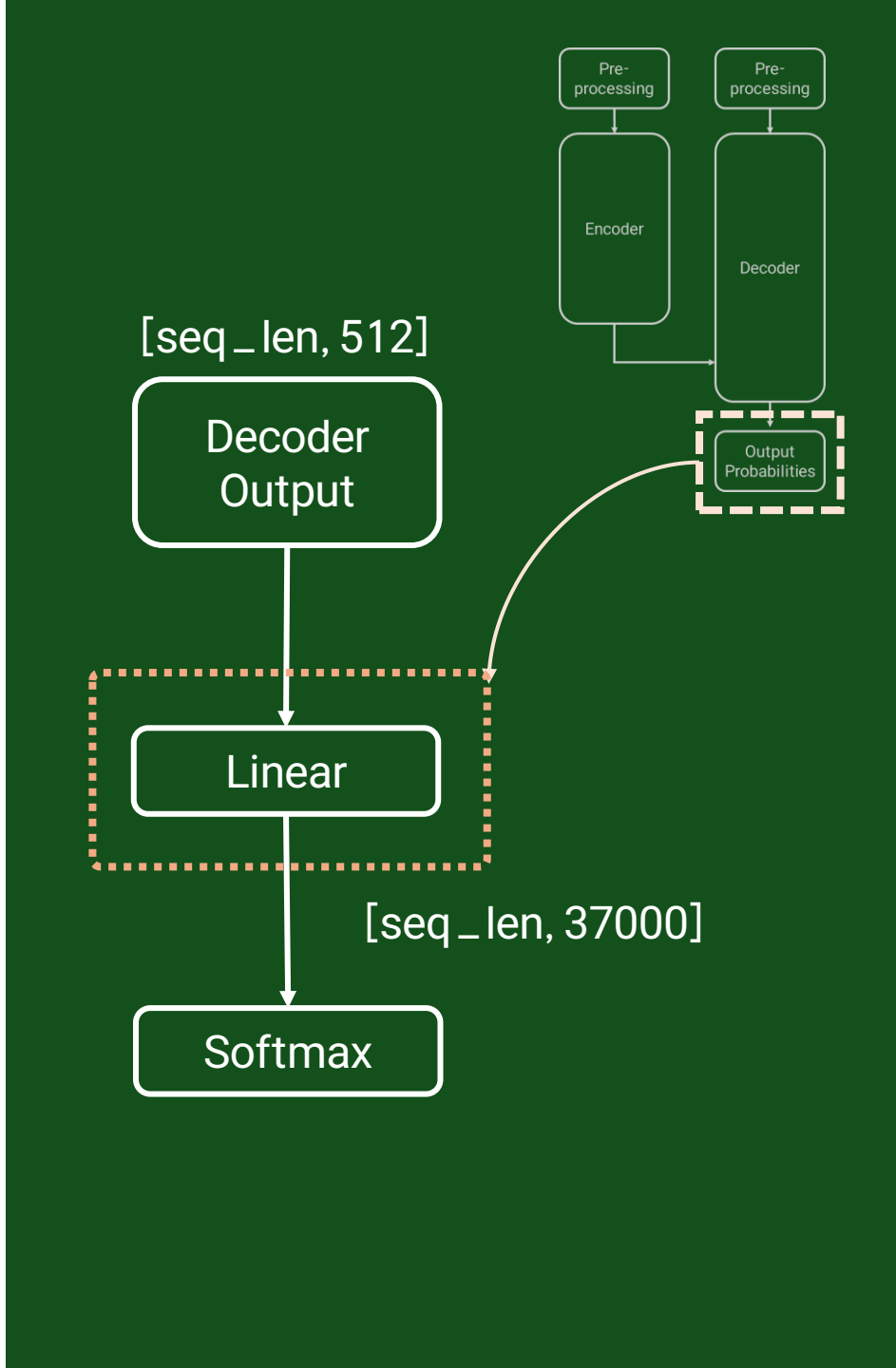


4-1. Linear

$$X \times W_{vocab} = Logits$$

[seq_len, 512] * [512, 37000]

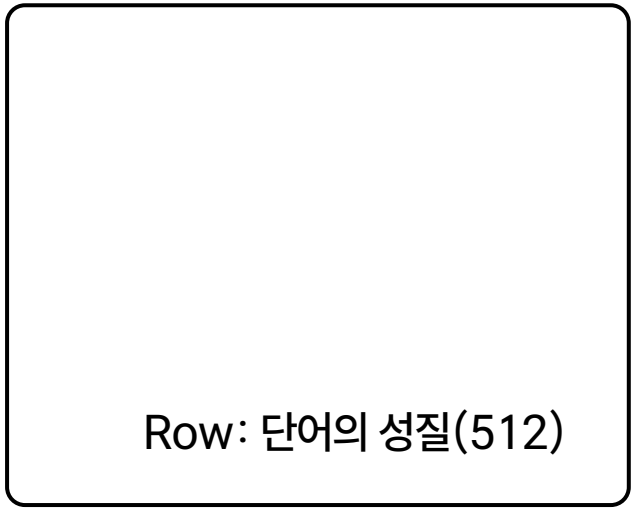
Expansion: 512개의 특징(dim)을 가진 vector를 vocab size 만큼 확장합니다. 결과물의 vector의 각 인덱스는 사전에 있는 특정 단어에 대응되며, 그 값은 해당 단어가 정답일 가능성을 나타내는 Logits 가 됩니다.



4-1. Linear

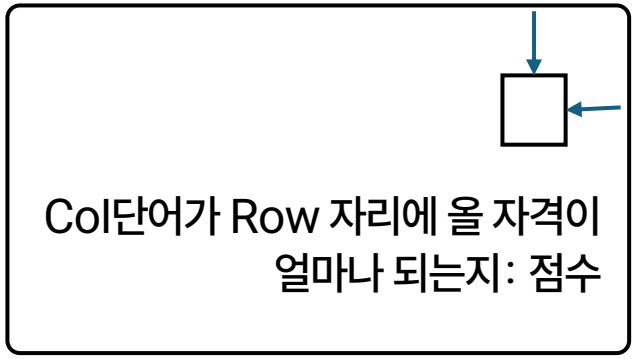
W_{vocab} [512, 37000]

Col: 사전에 있는 단어의 목록



$Logits$ [seq_len, 37000]

Col: 사전에 있는 단어의 목록



Row: 문장의 순서
(row I 는 i+1을 예측한 결과

모델이 찾아낸 512개 특징을 37000개의 단어와 일일이 대조하여 정답이 될 자격을 점수로 나타냅니다.

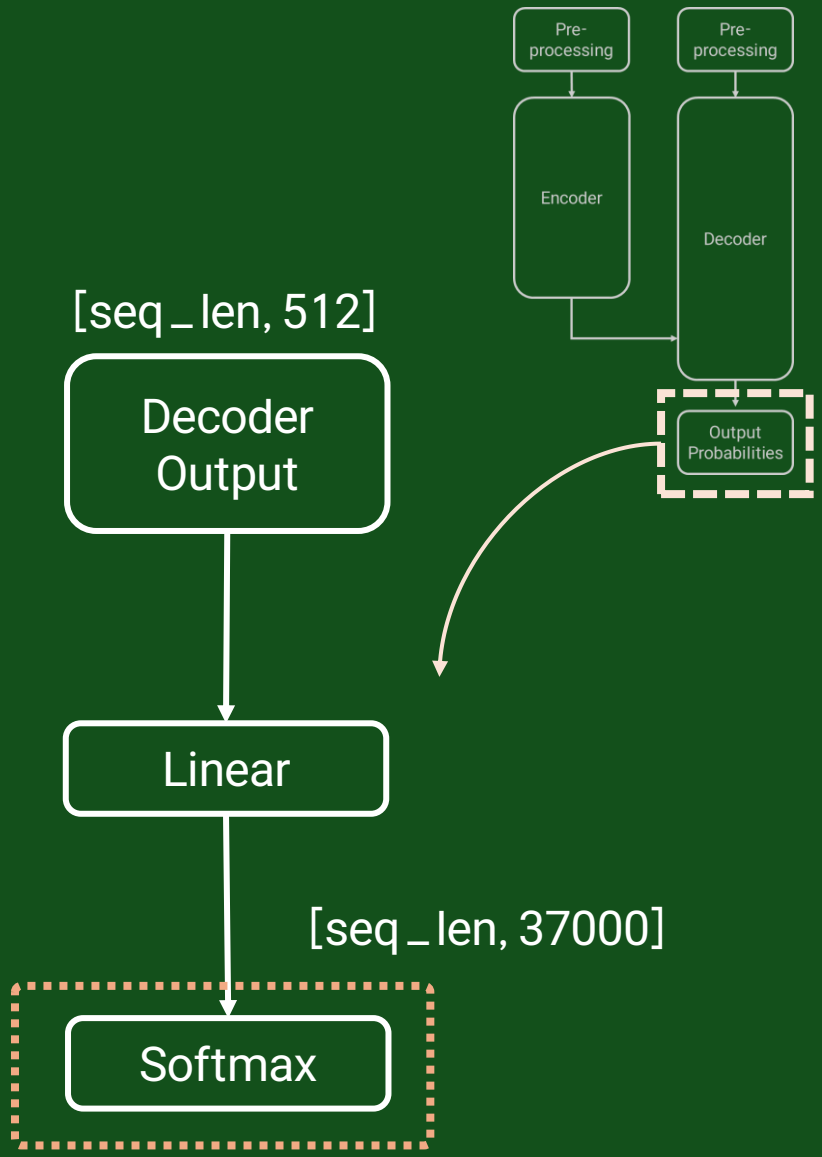
4-2. Softmax

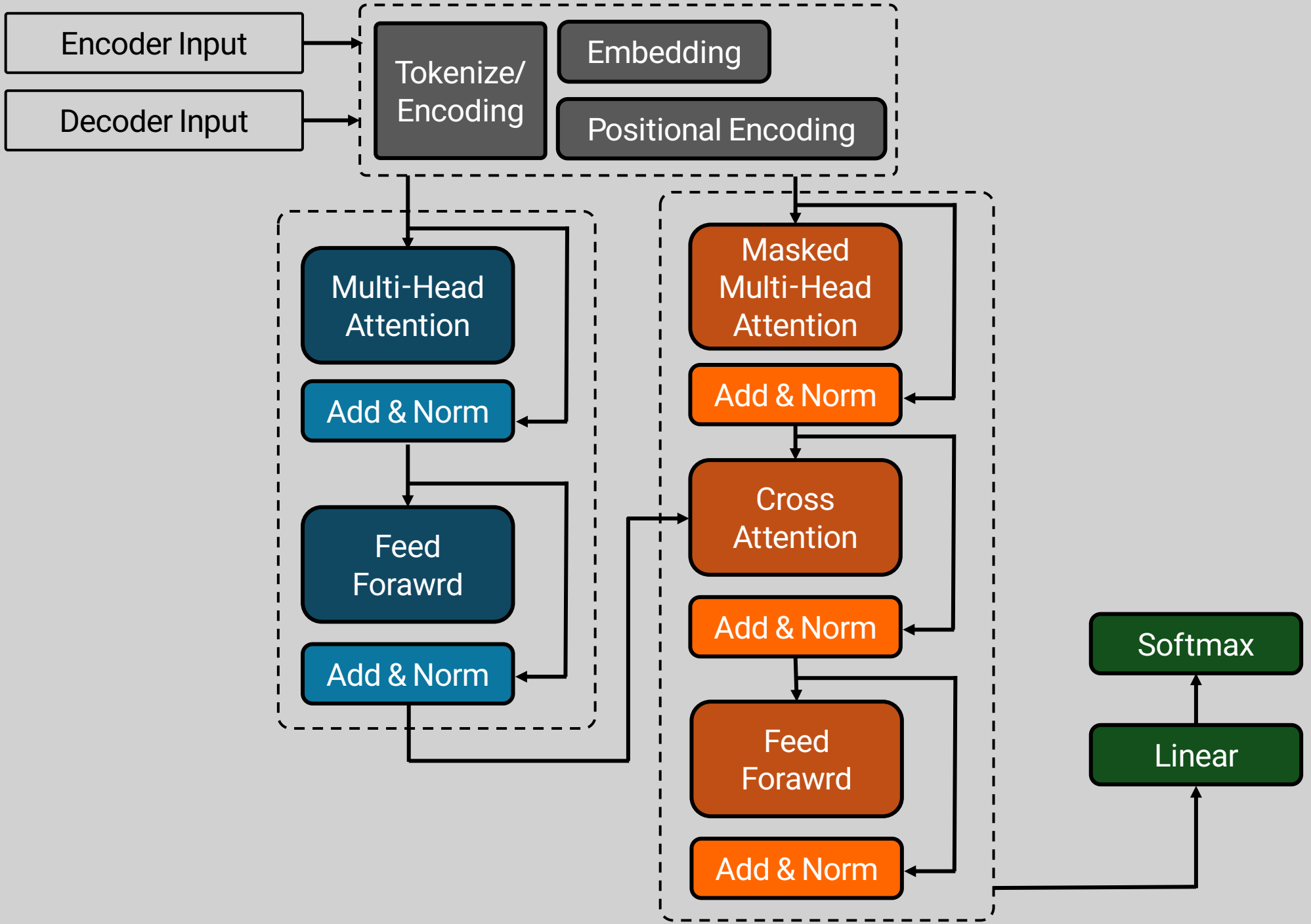
$$(Logits)softmax = Probabilities$$

[seq_len, 37000]

Logits 에 저장된 점수를 확률로 변환하여
다음에 어떤 단어가 올지 예측할 수 있도록 합니다.

이후 모델은 Row(순서)를 보고 어떤 단어가 와야하는지 확률을
확인하여 출력합니다.




Encoder Input
Decoder Input
Tokenize/
Encoding
Embedding
Positional Encoding
Multi-Head
Attention
Add & Norm
Feed
Forawrd
Add & Norm
Masked
Multi-Head
Attention
Add & Norm
Cross
Attention
Add & Norm
Feed
Forawrd
Add & Norm
Softmax
Linear