Transformer

Attention Is All You Need

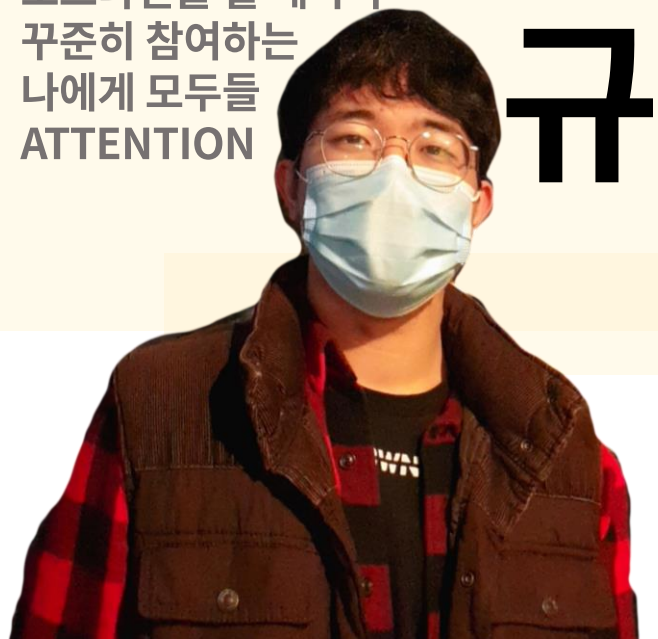
NLP 6조 김채형 백채빈 서경덕 이규민 조유림

코로나 때문에 우리 학생들, 학회를 못가고 있어요

SITUATION 1

오프라인 세션 참여하던 규민

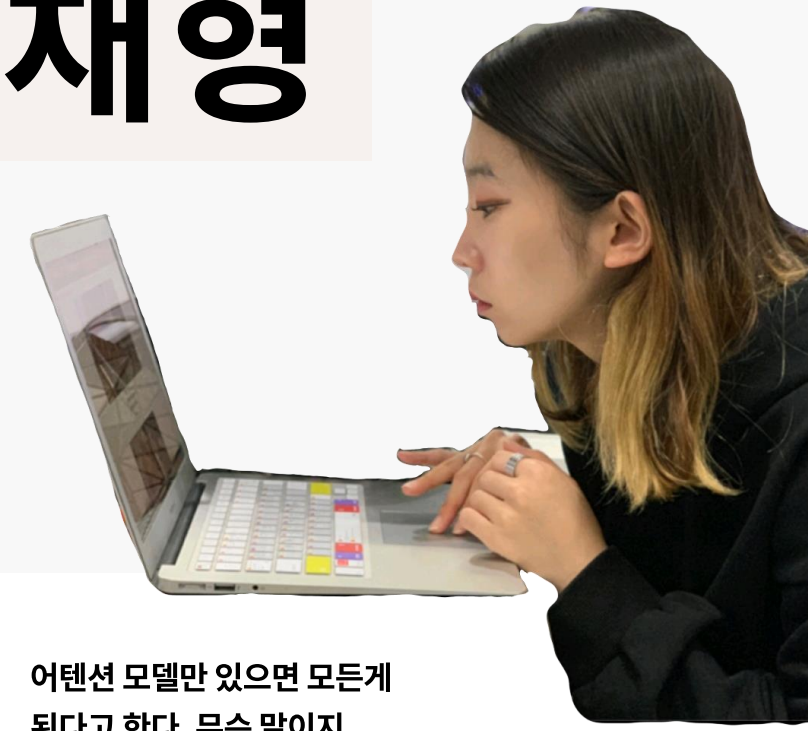
오프라인을 할 때마다
꾸준히 참여하는
나에게 모두들
ATTENTION



SITUATION 2

온라인 세션 참여하는 채형

ATTENTION
IS
ALL
YOU
NEED



어텐션 모델만 있으면 모든게
된다고 한다. 무슨 말이지,
왜 RNN과 CNN은 필요없지.
앞에서 배운 모델은 왜 배운거지.
그런데 어텐션은 도대체 뭐지?

Transformer 코드 구조

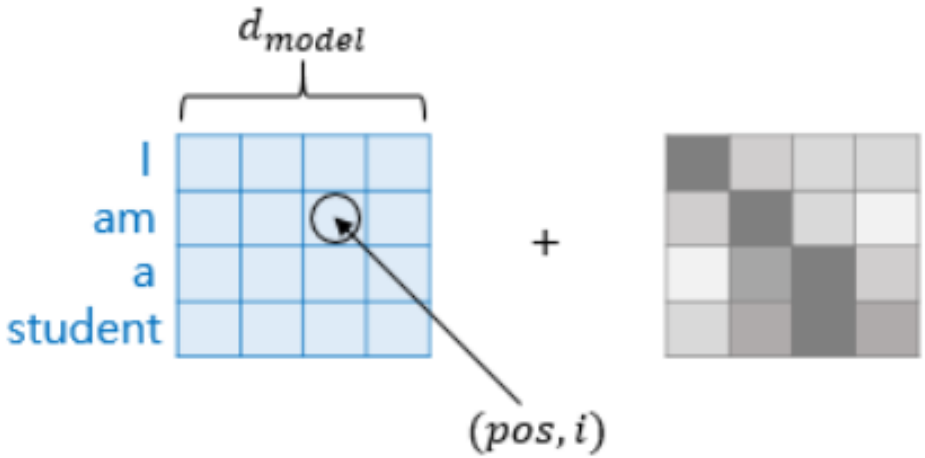

```
CODE
|
├── main                    --> 전체 코드 돌리는 부분
|   |── trainer             --> Train하는 부분
|   |   |── Transformer
|   |
|   └── utils
|       |── load_dataset    --> train,validation,test 데이터 불러오기
|       └── make_iter       --> 데이터를 torchtext dataset으로 변환
|
|
├── Transfomer
|   ├── encoder             --> 인코더
|   |   |── attention       --> attention sublayer
|   |   |── positionwise    --> feedforward network sublayer
|   |   └── ops             --> positional encoding, 마스킹 등
|   |
|   └── decoder             --> 디코더
|       |── attention
|       |── positionwise
|       └── ops
|
├── data                    --> 데이터(train,valid,test)
|   ├── train
|   ├── valid
|   └── test
|
└── config                  --> 하이퍼파라미터 지정
```

인코더 1. 임베딩 행렬 구성 2. positional encoding 3. encoding layer


```
def forward(self, source):
    # source = [batch size, 문장 길이] embedding matrix로 만들기 이전 상태
    source_mask = create_source_mask(source)      # [batch size, 문장 길이, 문장 길이]
    source_pos = create_position_vector(source)   # [batch size, 문장 길이] positional encoding의 인풋값으로 source와 동일한 차원

    # 1. 임베딩 행렬 구성
    source = self.token_embedding(source) * self.embedding_scale
    # [batch size, 문장 길이] 인풋값을 self.token_embedding(source) 통해서 임베딩 테이블 구축
    # self.embedding_scale를 통해 임베딩 벡터 scaling
    # [batch size, 문장 길이] --> [batch size, 문장길이, 512]

    # 2. positional encoding
    source = self.dropout(source + self.pos_embedding(source_pos))
    # 임베딩 행렬에 positional encoding을 더해주어 위치정보를 포함시킴
    # [batch size, 문장길이, 512]

    # 3. encoding layer
    for encoder_layer in self.encoder_layers:
        source = encoder_layer(source, source_mask)
    # 인코더 layer에 인풋값을 넣는다. 인코더는 6개의 layer로 구성되어있기 때문에 반복해서 연산 수행
    # [batch size, 문장길이, 512]
```

인코더 1~2 임베딩 행렬 구성 + positional encoding

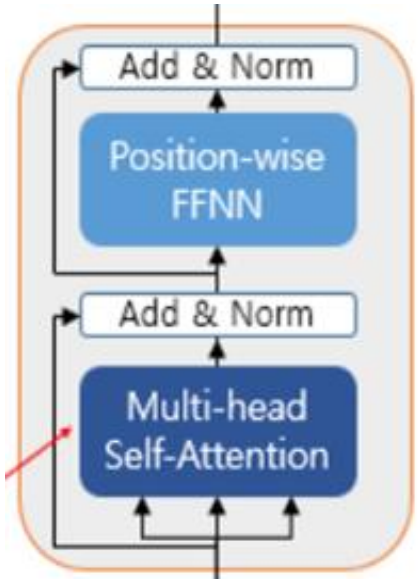

```
# 1. 임베딩 행렬 구성
source = self.token_embedding(source) * self.embedding_scale
# [batch size, 문장 길이] 인풋값을 self.token_embedding(source) 통해서 임베딩 테이블 구축
# self.embedding_scale를 통해 임베딩 벡터 scaling
# [batch size, 문장 길이] --> [batch size, 문장길이, 512]

# 2. positional encoding
source = self.dropout(source + self.pos_embedding(source_pos))
# 임베딩 행렬에 positional encoding을 더해주어 위치정보를 포함시킴
# [batch size, 문장길이, 512]
```



인코더 3. encoding layer


```
# 3. encoding layer
for encoder_layer in self.encoder_layers:
    source = encoder_layer(source, source_mask)
# 인코더 layer에 인풋값을 넣는다. 인코더는 6개의 layer로 구성되어있기 때문에 반복해서 연산 수행
# [batch size, 문장길이, 512]
```

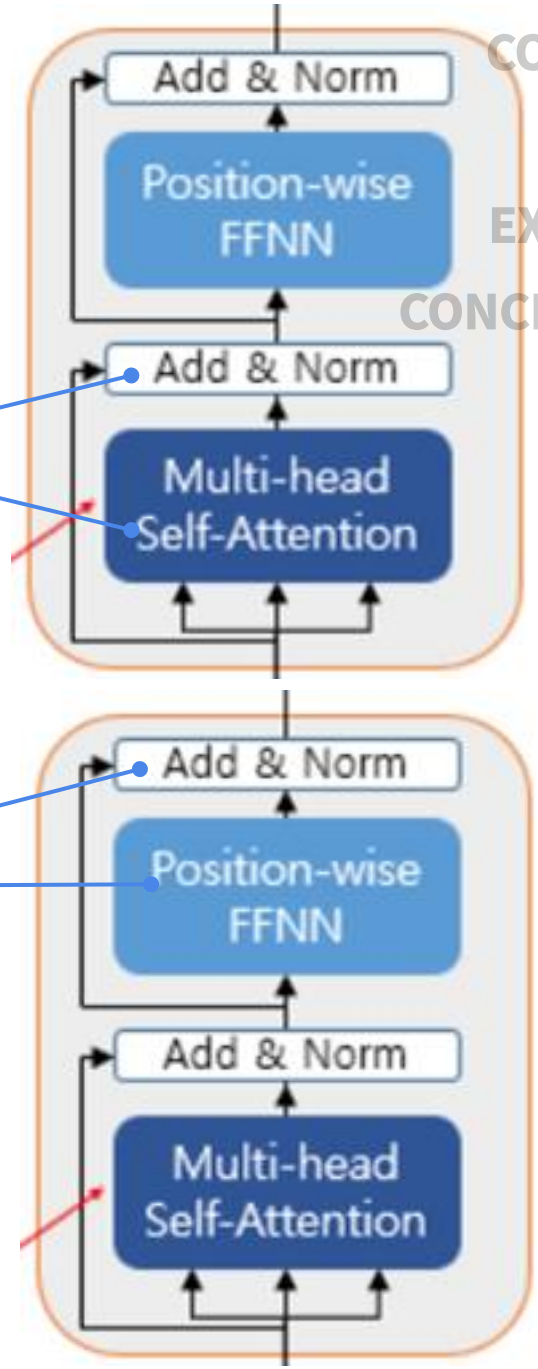
인코더 layer수 : 2
(하이퍼파라미터 변경)




```
class EncoderLayer(nn.Module):
    def __init__(self, params):
        super(EncoderLayer, self).__init__()
        self.layer_norm = nn.LayerNorm(params.hidden_dim, eps=1e-6) #layer normalization 진행하는 부분
        self.self_attention = MultiHeadAttention(params) # MultiheadAttention 진행하는 부분
        self.position_wise_ffn = PositionWiseFeedForward(params) # PositionWiseFeedForward 네트워크 통과
```

Class EncoderLayer

```
def forward(self, source, source_mask):
    # source          = [batch size, 문장길이, 512]
    # source_mask     = [batch size, 문장길이, 문장길이]

    #1. Multi-head attention
    output = source + self.self_attention(source, source, source, source_mask)[0]
    # Multihead attention + residual connection
    # Multi-head Attention에 넣은(F(X)) + 인풋값(x)  = F(x) + x

    normalized_output = self.layer_norm(output)
    # layer normalization
    # 정리하면, normalized_output은 multihead attention, add&norm까지 완료함. FFNN의 인풋값이 된다

    #2. Feed forward
    output = normalized_output + self.position_wise_ffn(normalized_output)
    # FFNN + residual connectoin
    # FFNN의 아웃풋(F(x))과 인풋값 x를 더해준다.

    output = self.layer_norm(output)
    # layer normalization
    # output = [batch size, 문장길이, 512]

    return output
```



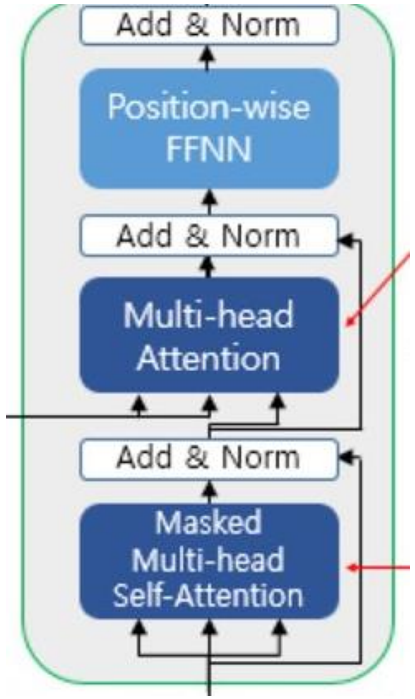
디코더 1~2. 임베딩 행렬 구성 + positional encoding


```
def forward(self, target, source, encoder_output):
    # target              = [batch size, target 문장 길이]
    # source              = [batch size, source 문장 길이]
    # encoder_output      = [batch size, source 문장 길이, 512]

    # 1. 임베딩 행렬 구성
    target_mask, dec_enc_mask = create_target_mask(source, target)
    # target_mask는 zero padding masking과 미래시점 참조 못하도록 하는 masking 포함
    # dec_enc_mask는 zero padding masking
    target_pos = create_position_vector(target)  # [batch size, target 문장 길이] positional encoding의 인풋값으로 targeti
    target = self.token_embedding(target) * self.embedding_scale
    # [batch size, target 문장 길이] 인풋값을 self.token_embedding(target) 통해서 임베딩 테이블 구축
    # self.embedding_scale를 통해 임베딩 벡터 scaling
    # [batch size, target 문장 길이] --> [batch size, target 문장 길이, 512]

    # 2. positional encoding
    target = self.dropout(target + self.pos_embedding(target_pos))
    # 임베딩 행렬에 positional encoding을 더해주어 위치정보를 포함시킴
    # [batch size, target 문장 길이, 512]
```

디코더 3. decoding layer


```
# 3. decoder layer
for decoder_layer in self.decoder_layers:
    target, attention_map = decoder_layer(target, encoder_output, target_mask, dec_enc_mask)
# 디코더 layer에 인풋값(target,encoder_output)을 넣는다.
# [batch size, 문장길이, 512]
```

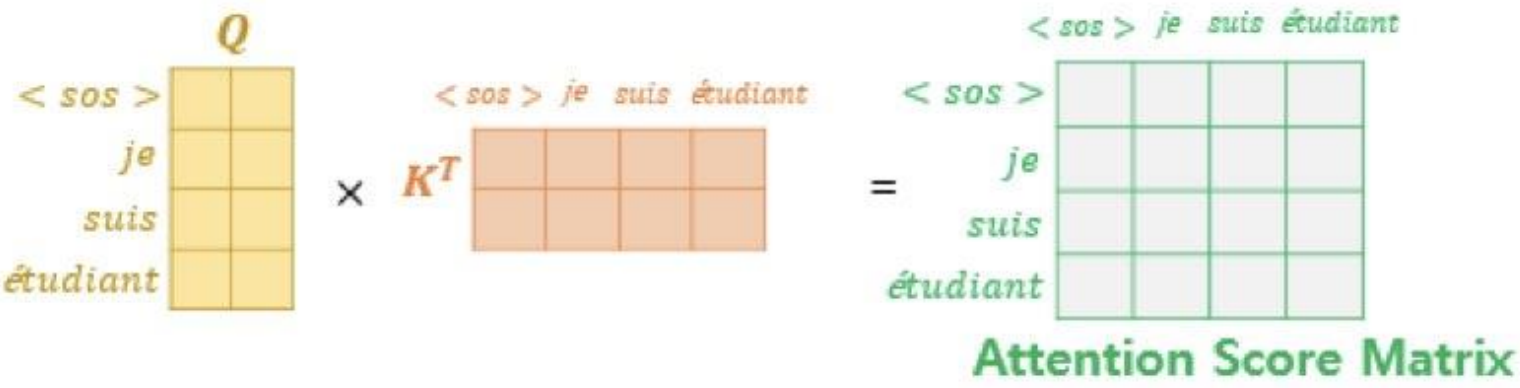
인코더는 2개의 layer로
구성되었기 때문에 반복 연산

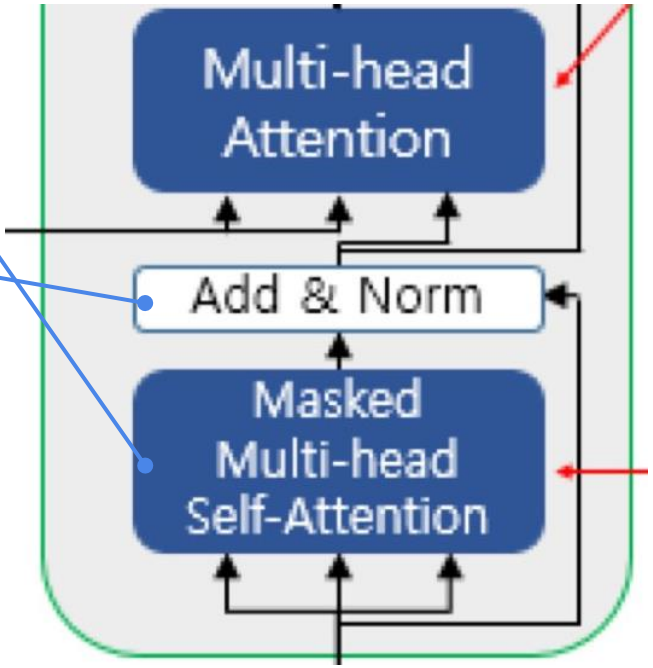



```
class DecoderLayer(nn.Module):

    def __init__(self, params):
        super(DecoderLayer, self).__init__()
        self.layer_norm = nn.LayerNorm(params.hidden_dim, eps=1e-6)
        self.self_attention = MultiHeadAttention(params) # masked multi head attention 부분
        self.encoder_attention = MultiHeadAttention(params) # multi-head attention 부분
        self.position_wise_ffn = PositionWiseFeedForward(params) #FFNN sublayer
```

Class DecoderLayer

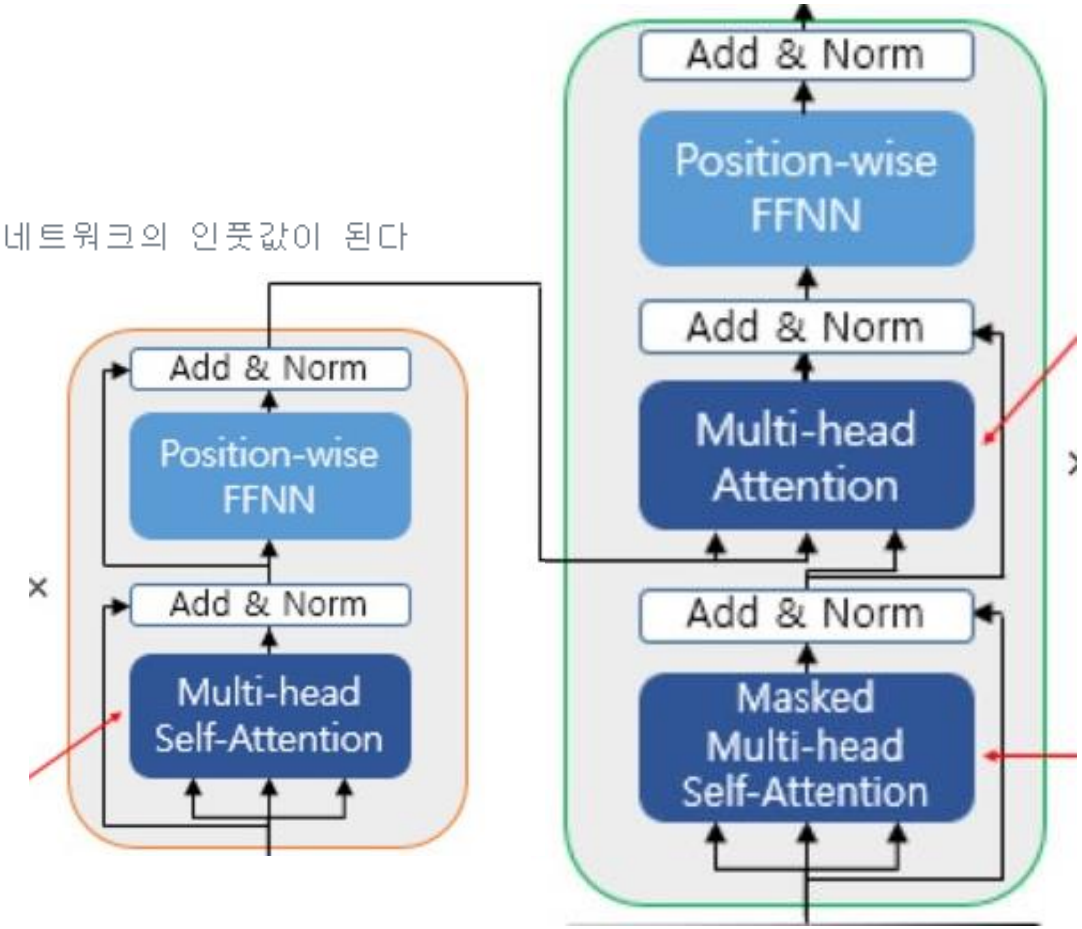

```
def forward(self, target, encoder_output, target_mask, dec_enc_mask):
    # target          = [batch size, target 문장 길이, 512]
    # encoder_output  = [batch size, source 문장 길이, 512]
    # target_mask     = [batch size, target 문장 길이, 512]
    # dec_enc_mask    = [batch size, target 문장 길이, 512]

    # 1. masked multi-head attention
    output = target + self.self_attention(target, target, target, target_mask)[0]
    # masked multi-head attention + residual connection
    # masked multi-head attention 넣은 (F(X)) + 인풋값(x)  = F(x) + x
    norm_output = self.layer_norm(output)
    # layer normalization
    # 정리하면, norm_output은 masked multi-head attention, add&norm까지 완료함.
```





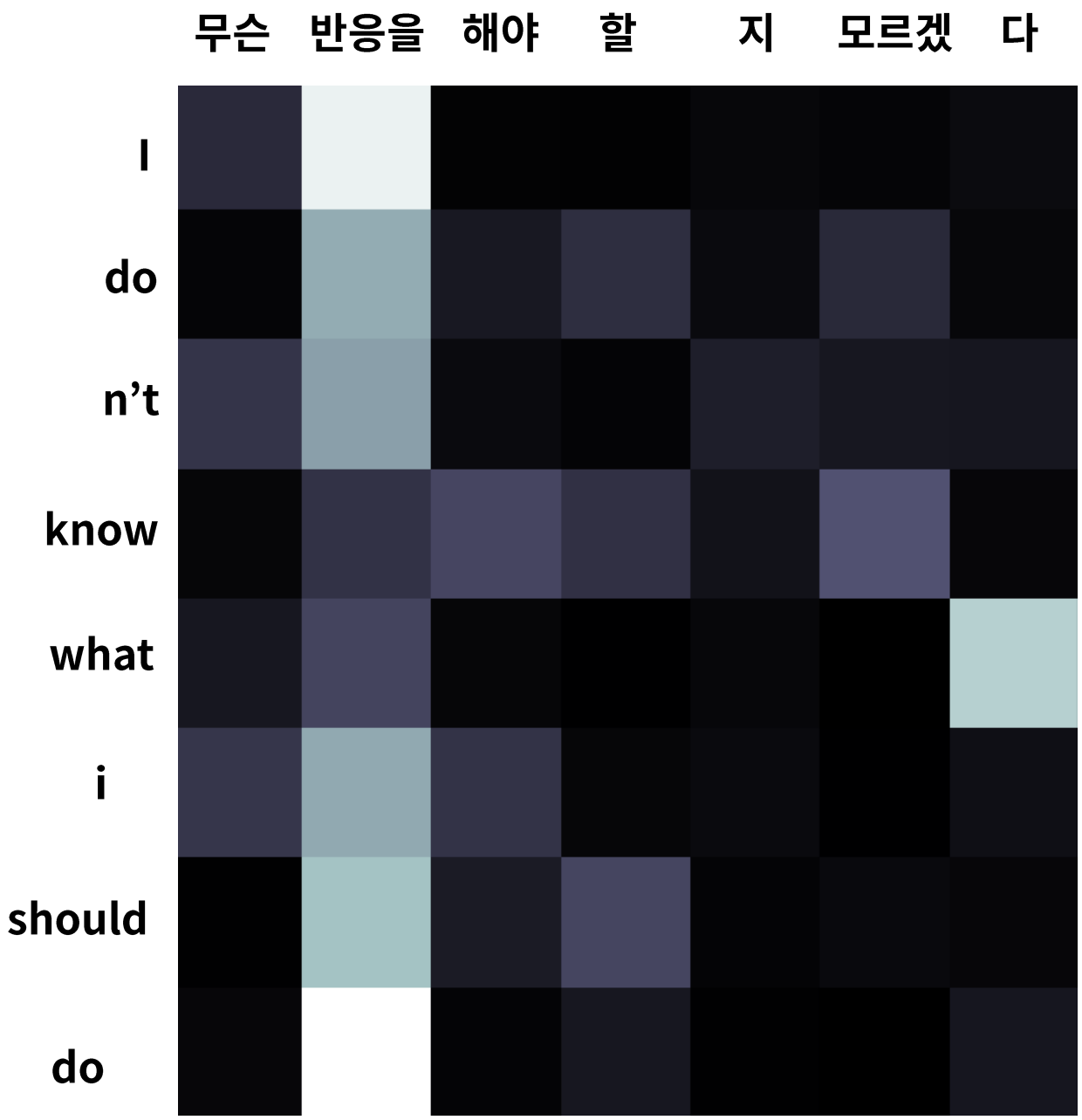
Class DecoderLayer


```
# 2. Multi-head attention
sub_layer, attn_map = self.encoder_attention(norm_output, encoder_output, encoder_output, dec_enc_mask)
# decoder의 두번째 sublayer는 쿼리를 구성하기 위해 decoder 첫번째 sublayer output이 사용되며, key value를 구성하기 위해 인코더 아웃풋이 사용된다.
output = output + sub_layer
# Multi-head attention + residual connection
# F(x) + x
norm_output = self.layer_norm(output)
# layer normalization
# 정리하면, norm_output은 multi-head attention + add$norm까지 완료. FFNN 네트워크의 인풋값이 된다

# 3. Feed forward
output = output + self.position_wise_ffn(norm_output)
# FFNN + residual connectoin
# FFNN의 아웃풋(F(x))과 인풋값 x를 더해준다.
output = self.layer_norm(output)
# layer normalization
# output = [batch size, 문장길이, 512]

return output, attn_map
```



잘 번역된 예시

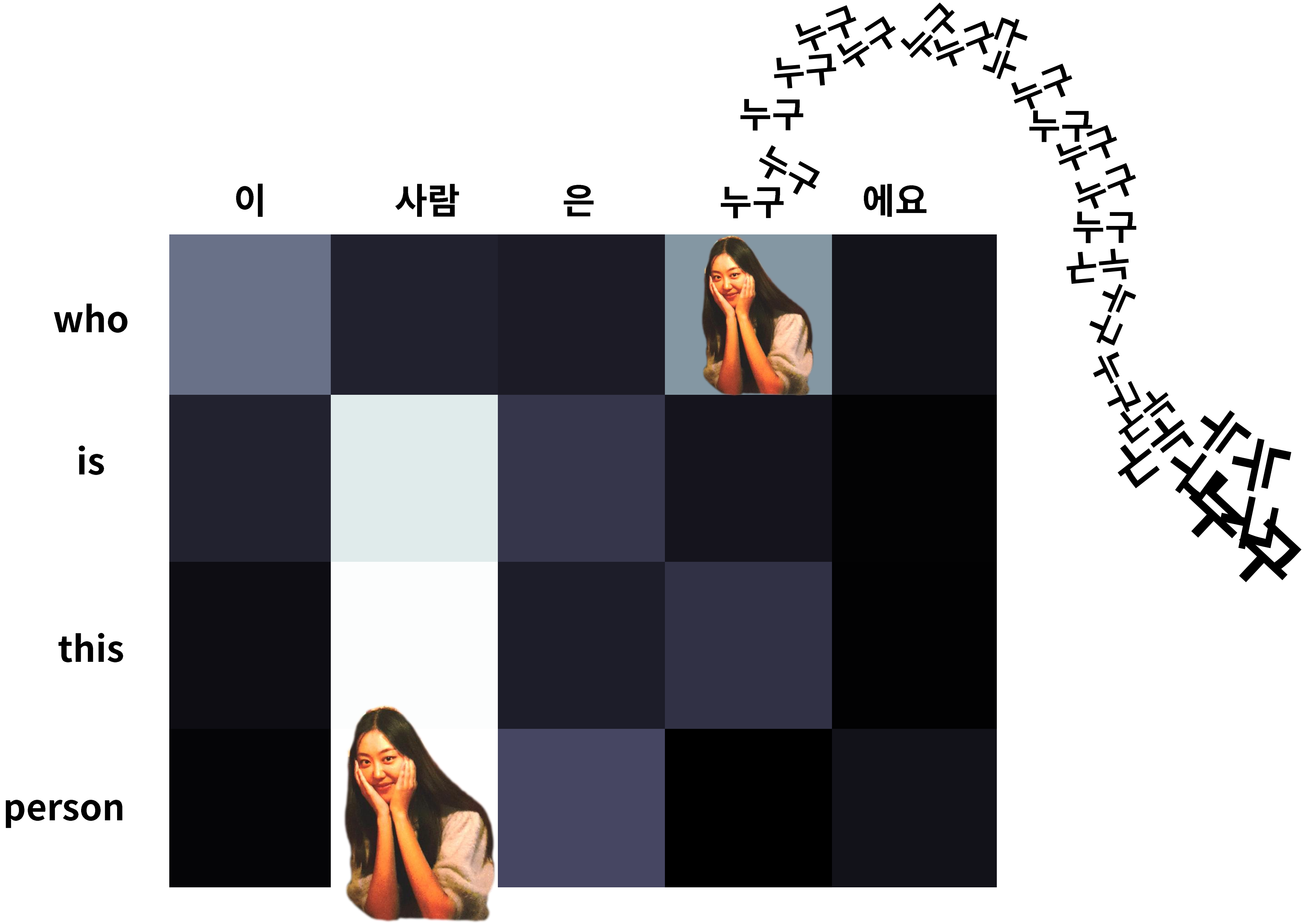
>



(무슨 반응을 해야 할 지 모르겠다)

잘 번역된 예시

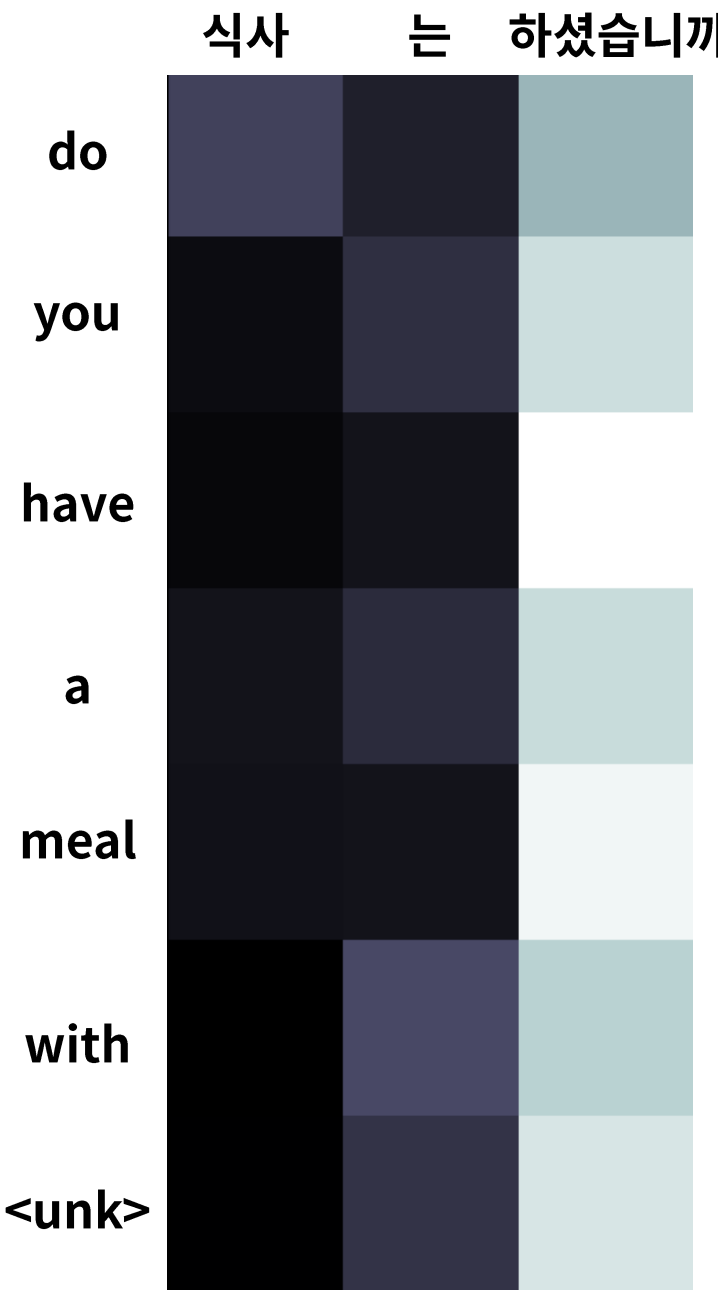
>



정답!

조유림!

잘? 번역된 예시



한국어 영어

식사는 하셨습니까?

sigsaneun hasyeossseubnikka?

Did you have dinner?

Google 번역에서 열기

papago

한국어 감지 영어

식사는 하셨습니까?

10 / 5000

Did you eat?

디드 유 잇

번역 수정 번역 평가

번역하기

잘? 번역된 예시

다음 생애에는 오래오래 당신 옆에 있을게요.

>

I will <unk> you next time.


다음
생에는
오래오래
당신
곁에
있을게요
i
will
<unk>
you
next
time
F
I
N
D

갑자기
분위기
테이큰


I WILL FIND YOU
AND I WILL KILL YOU

잘? 번역된 예시

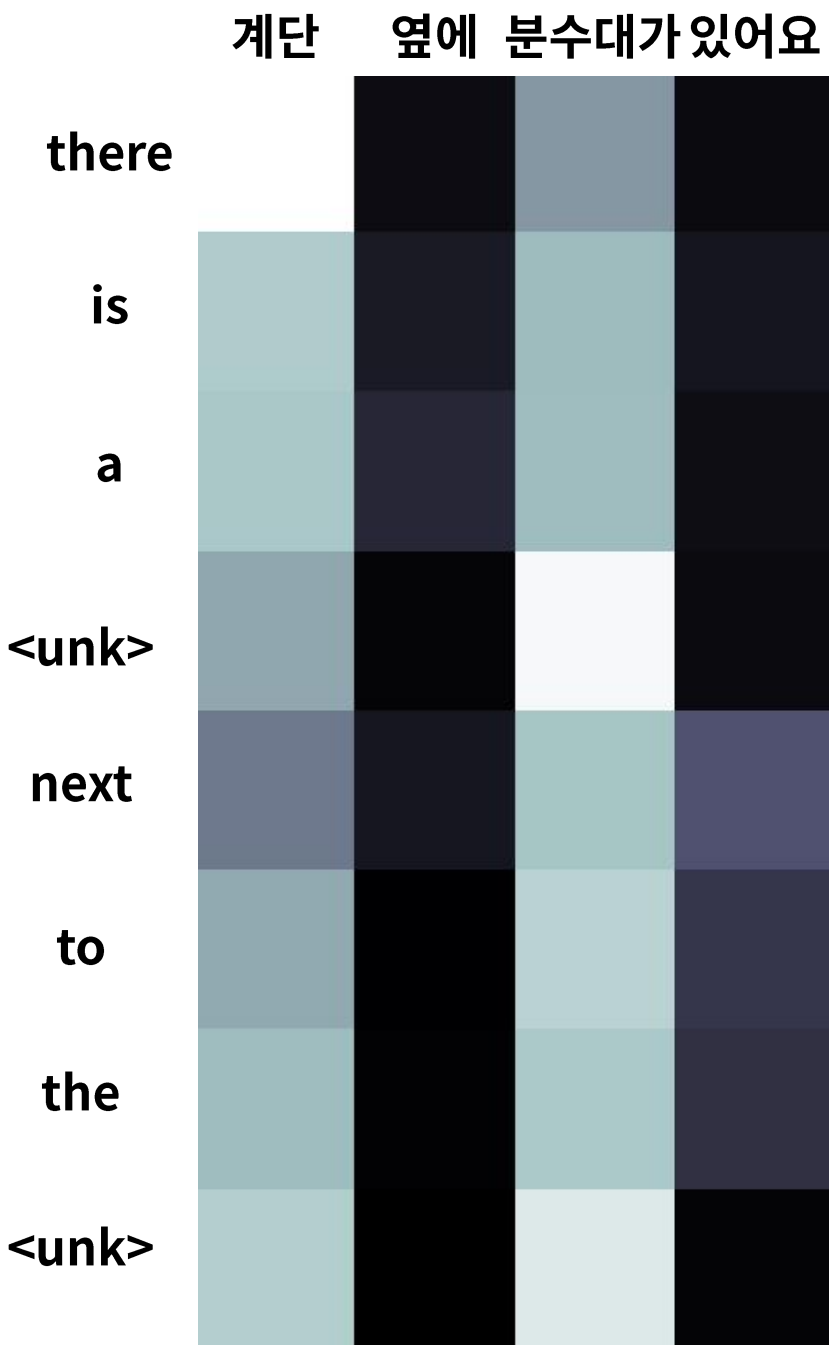
6. 대화를 듣고, 그림에서 대화의 내용과 일치하지 않는 것을 고르시오.

계단 옆에 분수대가 있어요.

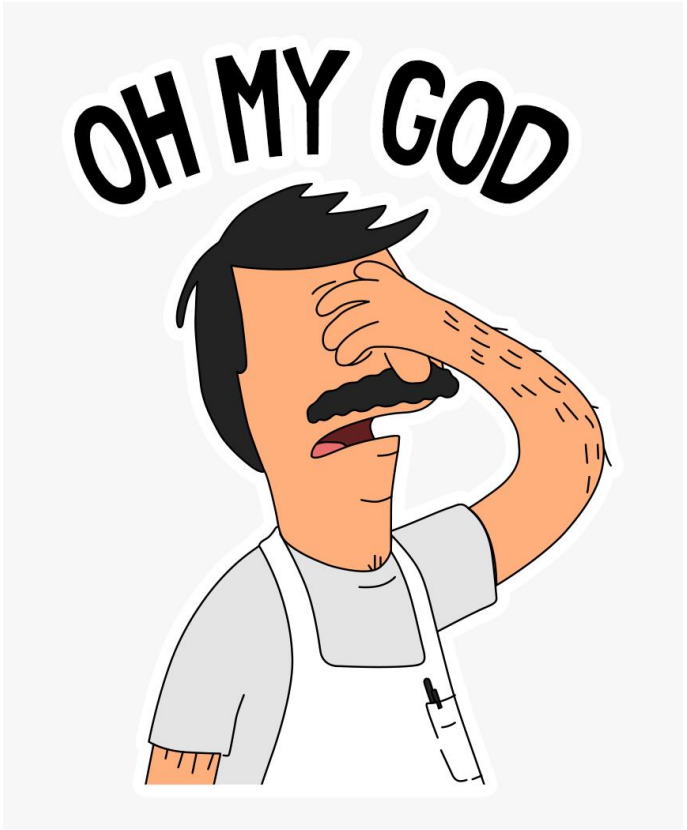
>

6. 대화를 듣고, 그림에서 대화의 내용과 일치하지 않는 것을 고르시오.

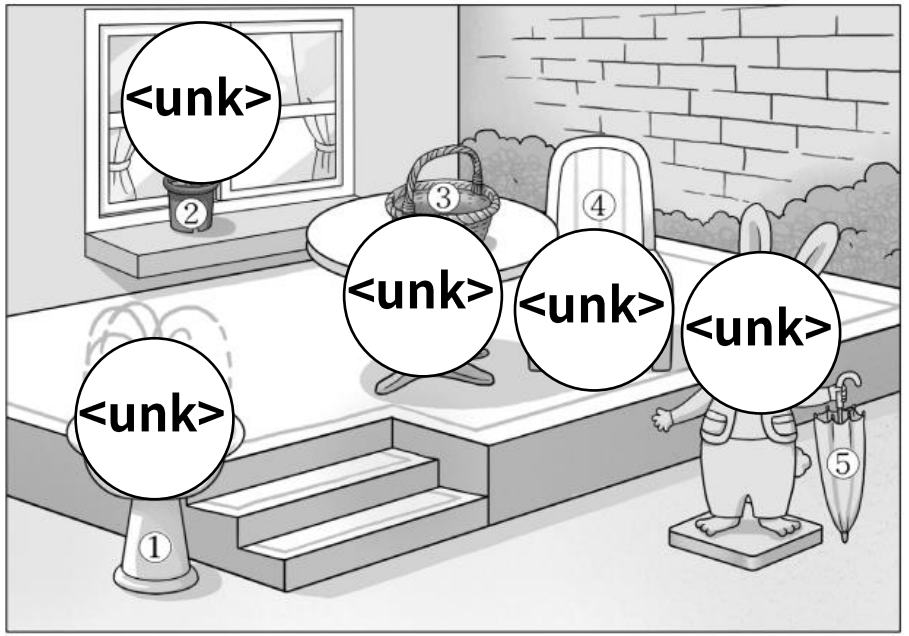
There is a <unk> next to the <unk>.

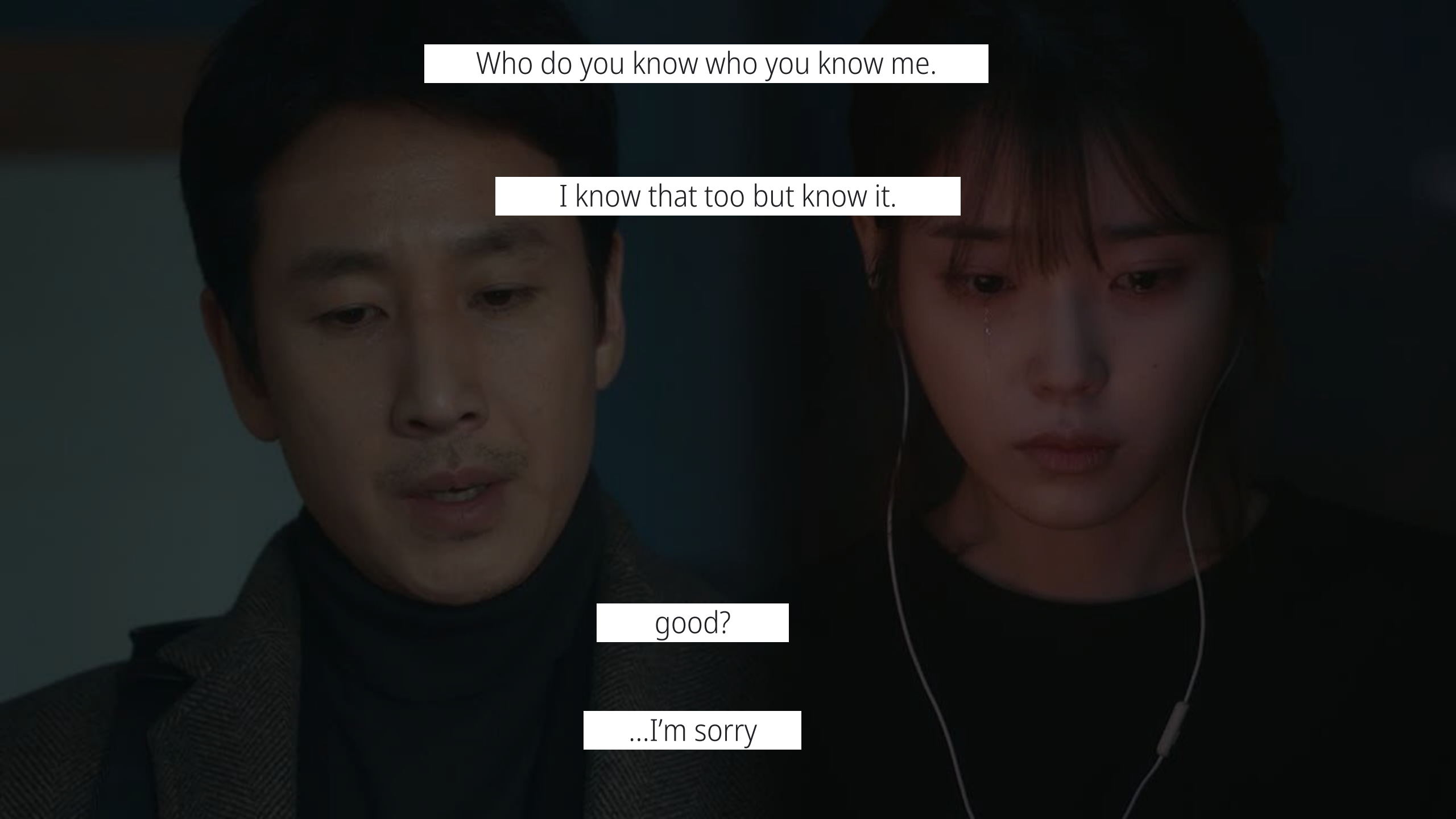


(오늘 수능임)



6. 대화를 듣고, 그림에서 대화의 내용과 일치하지 않는 것을 고르시오.




Who do you know who you know me.
I know that too but know it.
good?
...I'm sorry

Transformer

하이퍼파라미터

```
{
    "model": "transformer",
    "save_model": "model.pt",

    "mode": "train",
    "optim": "Adam",

    "random_seed": 32,
    "clip": 1,

    "batch_size": 128,
    "num_epoch": 100,
    "warm_steps": 2000,

    "hidden_dim": 128,
    "feed_forward_dim": 1024,
    "n_layer": 3,
    "n_head": 8,
    "max_len": 64,
    "dropout": 0.1
}
```

```
The model has 2,390,016 trainable parameters
Epoch: 01 | Epoch Time: 5m 0s
        Train Loss: 3.833 | Val. Loss: 3.447
```

개선점

1. 컴퓨터 성능

기존 하이퍼파라미터로는 학습해야할 파라미터가 4천6백만개 정도

파라미터를 2백4십만개로 낮추고 학습해도 에폭 1번에 5분 * 100 = 500분
(피시방 컴퓨터 기준)

2. 트레인셋이 부족(명사 인지가 약함)

생각보다 성능이 좋지 않습니다

It’s better than I thought about it

다들 발표를 들어주셔서 감사합니다.

Thank you for <unk> the presentation.

감사합니다