KOREATECH 한국기술교육대학교

8 주차 과제

학번	2020136087
이름	윤아현
수강 과목	자연어처리
담당 교수	임상훈 교수님
제출일	2024.05.04

문제 및 해결 방법 (구현 코드)

1. Attention Heatmap 출력 및 분석 (50 점)

* Transformer 모델의 Attention distribution 을 Heatmap 형식으로 출력하시오.
    * Attention Heatmap 을 출력한 후 비교, 분석 하시오
        * 학습전, 학습후(최소 10epoch 이상)의 차이
        * Head 별 분석
        * Layer 별 분석

****GRADING****

* Heatmap 출력 (+20 점)
* Heatmap 비교 분석 (+30 점)

Code

```python
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
        assert d_model % self.num_heads == 0
        self.depth = d_model // self.num_heads

        self.Wq = nn.Linear(d_model, d_model)
        self.Wk = nn.Linear(d_model, d_model)
        self.Wv = nn.Linear(d_model, d_model)

        self.dense = nn.Linear(d_model, d_model)

    def split_heads(self, x, batch_size):
        x = x.view(batch_size, -1, self.num_heads, self.depth)
        return x.permute(0, 2, 1, 3)

    # Decoder에서 mask 기법 사용하기 위해서 multi-head에서 attention mask를 입력으로 받는다.
    def forward(self, q, k, v, mask):
        batch_size = q.size(0)

        q = self.Wq(q)
        k = self.Wk(k)
        v = self.Wv(v)

        # heads 개수로 쪼개준다. → 128차원 vector로 만들어, 8개(head 개수)로 쪼개준다.
        q = self.split_heads(q, batch_size)
        k = self.split_heads(k, batch_size)
        v = self.split_heads(v, batch_size)


        attn = torch.matmul(q, k.permute(0, 1, 3, 2)) / math.sqrt(self.depth)
        attn = attn.masked_fill(mask.unsqueeze(1) == 0, -1e9)
        attn = torch.nn.functional.softmax(attn, dim=-1)

        out = torch.matmul(attn, v)

        out = out.permute(0, 2, 1, 3).contiguous()
        out = out.view(batch_size, -1, self.d_model)

        return self.dense(out), attn
```

```python
# Decoder Layer : masked Multi-head attention - Multi-head Attention + Feed-Forward
class TrasnformerDecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, dff, dropout_rate):
        super(TrasnformerDecoderLayer, self).__init__()
        # multi-head Attention 2개 정의 (masked. origin)
        self.mha1 = MultiHeadAttention(d_model, num_heads)
        self.mha2 = MultiHeadAttention(d_model, num_heads)
        # feed-forward 층
        self.ffn = nn.Sequential(
            nn.Linear(d_model, dff),
            nn.ReLU(),
            nn.Linear(dff, d_model)
        )
        # add & norm layer 3개
        self.layernorm1 = nn.LayerNorm(d_model)
        self.layernorm2 = nn.LayerNorm(d_model)
        self.layernorm3 = nn.LayerNorm(d_model)
        self.dropout1 = nn.Dropout(dropout_rate)
        self.dropout2 = nn.Dropout(dropout_rate)
        self.dropout3 = nn.Dropout(dropout_rate)

    # decoder의 입력(target text) + encoder에서 나오는 최종 output + mask + encoder+decoder에서 쓰이는 mask
    def forward(self, x, enc_output, look_ahead_mask, padding_mask):
        attn1, attn_weights1  = self.mha1(x, x, x, look_ahead_mask)
        attn1 = self.dropout1(attn1)
        out1 = self.layernorm1(x + attn1)

        attn2, attn_weights2 = self.mha2(out1, enc_output, enc_output, padding_mask)
        attn2 = self.dropout2(attn2)
        out2 = self.layernorm2(out1 + attn2)

      # feed forward, 및 add & norm 과정 수행
        ffn_output = self.ffn(out2)
        ffn_output = self.dropout3(ffn_output)
        out3 = self.layernorm3(out2 + ffn_output)

        return out3, attn_weights1, attn_weights2
```

```python

# Decoder: pad mask, look_ahead_mask 둘 다 사용
# [PAD] 토큰으로 [PAD] 토큰도 같이 학습되는 것을 금지한다.
class TransformerDecoder(nn.Module):
    def __init__(self, num_layers, d_model, num_heads, dff, dropout_rate):
        super(TransformerDecoder, self).__init__()
        self.dec_layers = nn.ModuleList([TrasnformerDecoderLayer(d_model, num_heads, dff, dropout_rate) for _ in range(num_layers)])
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, x, enc_output, look_ahead_mask, padding_mask):
        attention_weights = {}

        for i, layer in enumerate(self.dec_layers):
            x, weights1, weights2 = layer(x, enc_output, look_ahead_mask, padding_mask)
            attention_weights[f'decoder_layer{i+1}_block1'] = weights1
            attention_weights[f'decoder_layer{i+1}_block2'] = weights2

        return x, attention_weights
```

```python
class Transformer(nn.Module):
    def __init__(self, num_layers, d_model, num_heads, dff, dropout_rate, en_vocab_size, fr_vocab_size):
        super(Transformer, self).__init__()
        self.output_dim = fr_vocab_size
        # 1번. Embedding 정의 (position Embedding은 수행하지 않았음)
        self.en_Embedding = nn.Embedding(en_vocab_size, d_model) # encoder embedding Layer
        self.fr_Embedding = nn.Embedding(fr_vocab_size, d_model) # decoder embedding Layer

        # 사실 position Embedding도 해야함

        # Transformer 정의
        self.encoder = TransformerEncoder(num_layers, d_model, num_heads, dff, dropout_rate) # Transformer-encoder
        self.decoder = TransformerDecoder(num_layers, d_model, num_heads, dff, dropout_rate) # Transformer-decoder
        self.final_layer = nn.Linear(d_model, fr_vocab_size)

    # embedding layer -> 입력을 통해 output 출력
    # enc_padding_mask를 통해 [pad] 토큰들을 mask한다. (attention matrix 계산에 사용 X)
     def encode(self, enc_input, enc_padding_mask):
        return self.encoder(self.en_Embedding(enc_input), enc_padding_mask)

  # embedding layer -> encoder output, decoder에 해당하는 입력 전달
  # [pad] 토큰 mask 및 현 위치에서 미래 정보 mask 수행
    def decode(self, dec_input, enc_output, look_ahead_mask, dec_padding_mask):
        return self.decoder(self.fr_Embedding(dec_input), enc_output, look_ahead_mask, dec_padding_mask)

    def forward(self, enc_input, dec_input, enc_padding_mask, look_ahead_mask, dec_padding_mask):
        enc_output = self.encode(enc_input, enc_padding_mask)
        dec_output, attention_weights = self.decode(dec_input, enc_output, look_ahead_mask, dec_padding_mask)
        final_output = self.final_layer(dec_output) # decoder의 마지막 output에서 최종 output 출력
        return final_output, attention_weights
```

[Transformer PL 함수]

```
    # padding mask : encoder, decoder
    # casual mask : decoder 적용
    def forward(self, src, trg):
        enc_padding_mask = self.make_pad_mask(src, src)
        dec_padding_mask = self.make_pad_mask(trg, src)
        look_ahead_mask = self.make_causal_mask(trg)

        # Head별 저장
        outputs, attention_weights = self.model(src, trg, enc_padding_mask, look_ahead_mask, dec_padding_mask)

        return outputs, attention_weights
```

```
    def decode(self, src):
      # encoder input에 대한 padding mask 적용
        enc_output = self.model.encode(src, self.make_pad_mask(src, src))
        trg_len = 30
        outputs = [2] # [sos] token
        # decoder 초기 입력 설정 (모든 배치에 대해 시작 토큰을 설정한다.)
        input = torch.LongTensor([[2] for _ in range(src.size(0))]).to(src.device)
        for t in range(1, trg_len):
          # 미래 token이 안 보이도록 설정
            look_ahead_mask = self.make_causal_mask(input)
          # decoder, encoder 사이 padding mask 생성
            dec_padding_mask = self.make_pad_mask(input, src)
            output, attention_weights = self.model.decode(input, enc_output, look_ahead_mask, dec_padding_mask)
            output = self.model.final_layer(output)
            output = output[:,-1,:]
            top1 = output.argmax(1)
            outputs.append(top1.item())
            if top1.item() == 3:
                break
            input = torch.cat([input, top1.unsqueeze(1)], dim=1)
        return outputs, attention_weights
```

```
import matplotlib.pyplot as plt
import seaborn as sns

def plot_attention_heatmaps(attn_weights, input, model_output):
    num_heads, source_length, target_length = attn_weights.shape[1:]

    fig, axes = plt.subplots(2, 4, figsize=(16, 8))

    for head in range(num_heads):
        ax = axes[head // 4, head % 4]  # 2행 4열로 인덱싱

        # 히트맵 그리기
        sns.heatmap(attn_weights[0, head].detach().numpy(), cmap='viridis', ax=ax, cbar=False)

        ax.set_xlabel('Input tokens')
        ax.set_ylabel('Output tokens')

        ax.set_title(f'Layer 1, Head {head+1}')

    plt.tight_layout()
    plt.show()
```

```
test_data = test_dataset.__getitem__(234)
a, attn_weights = model_pl.decode(test_data["src"].unsqueeze(0))
print("attn_weights keys:", attn_weights.keys())

input = " ".join([list(en_vocab.keys())[list(en_vocab.values()).index(i)] for i in test_data["src"]])
target = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in test_data["trg"]])
model_output = " ".join([list(fr_vocab.keys())[list(fr_vocab.values()).index(i)] for i in a])

print("Input (English):", input)
print("Target (French):", target)
print("Model Output (French):", model_output)
```

※ 수정한 코드만 첨부하였음

Outputs

Epoch 1:

```
attn_weights keys: dict_keys(['decoder_layer1_block1', 'decoder_layer1_block2', 'decoder_layer2_block1', 'decoder_layer2_block2'])
Input (English): i thought that you d want to see this movie . [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [P
Target (French): [SOS] je pensais que tu voudrais voir ce film . [EOS] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
Model Output (French): [SOS] je veux que vous voulez que ca vous voulez . [EOS]
```

Epoch 10:

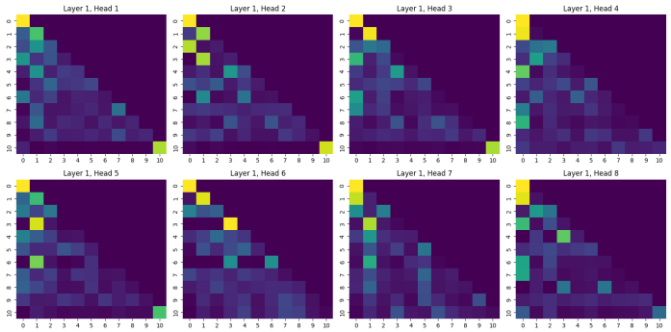
```
attn_weights keys: dict_keys(['decoder_layer1_block1', 'decoder_layer1_block2', 'decoder_layer2_block1', 'decoder_layer2_block2'])
Input (English): i thought that you d want to see this movie . [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD
Target (French): [SOS] je pensais que tu voudrais voir ce film . [EOS] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [P
Model Output (French): [SOS] je pensais que vous voudriez voir ce film . [EOS]
```

Epoch 1 VS Epoch 10 Layer 및 Head 별 분석:

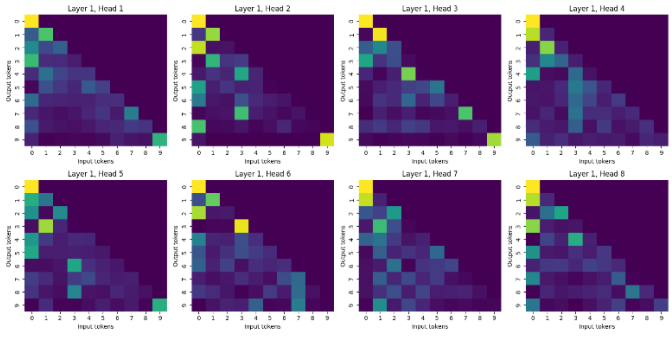
위에서 나온 Input 과 Model Output 을 통해 히트맵을 비교하였다.

Decoder Layer1 Block1 (Decoder 의 Multi-head Self-Attention, 첫 번째 Layer)

Epoch 1:

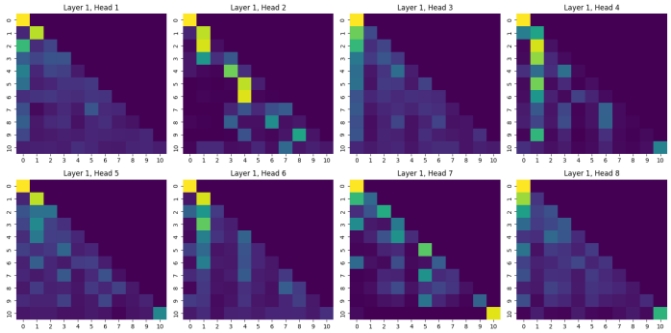


Epoch 10:

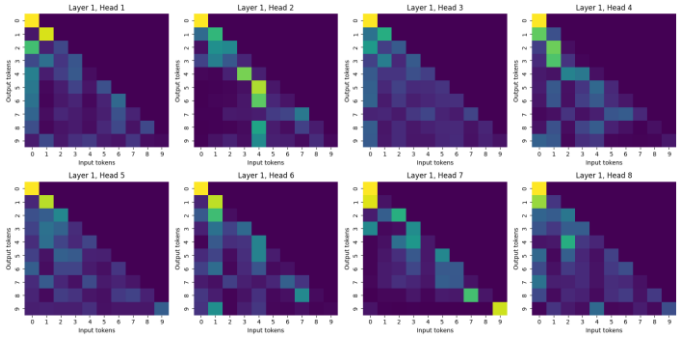


Decoder Layer2 Block1 (Decoder 의 Multi-head Self-Attention, 두 번째 Layer)

Epoch 1:



Epoch 10:



분석 결과

1 번. Head 별 분석

각 Head 들은 같은 heatmap 의 분포를 보이지 않고, 서로 다른 특성들을 반영한 heatmap 분포를 보인다. 몇 개의 Head 1 은 전반적으로 가중치가 행의 초반 부분에 집중되어 있지만, 갈수록 중앙 부분 혹은 고른 분포를 보이는 경향이 보였다.

하지만, Layer 1 의 Head 7 은 가중치가 행의 후반 부분에 집중되는 특이한 분포를 보였다.

2 번. Layer 별 분석

Layer1 과 Layer2 를 비교해보았을 때 Layer1 의 대체로 모든 단어에 대해 가중치의 다양한 구역에서 가중치가 높게 나타난다면, Layer2 는 비교적 가중치가 한 부분에 명확하게 나타낸다.

이를 통해, Layer1 은 서로 다른 부분을 비등비등하게 다양하게 해석을 하고 있고 Layer2 는 조금 더 한 부분에 집중하여 세밀한 학습을 수행하는 것처럼 관찰되었다.

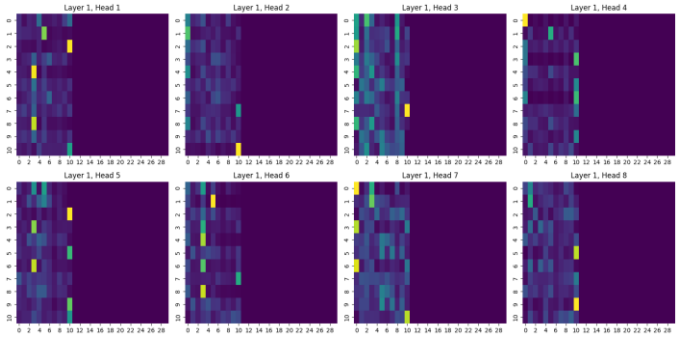
3 번. 학습 수에 따른 분석

Epoch 를 1 번, 10 번 진행되었을 때를 분석해보면, 각 Head 별로 가중치 분포가 조금 더 분명하게 보였다. 초기에 집중되거나 상관관계가 각 단어별로 비슷하게 나타나는 경향이 줄었고 한 부분에 상관관계가 높게 나타나는 비율이 높아졌다.

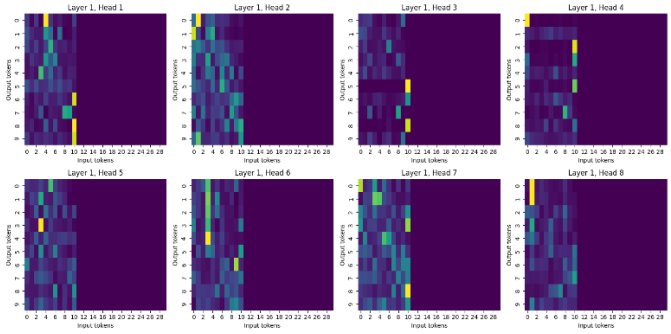
학습을 많이 수행할수록, 각 단어가 다른 단어와 얼마만큼의 연관이 있는지가 더 선명하게 보이는 것을 관찰할 수 있었다.

Decoder Layer1 Block2 (Decoder 의 Multi-head Attention, 첫 번째 Layer)

Epoch 1:

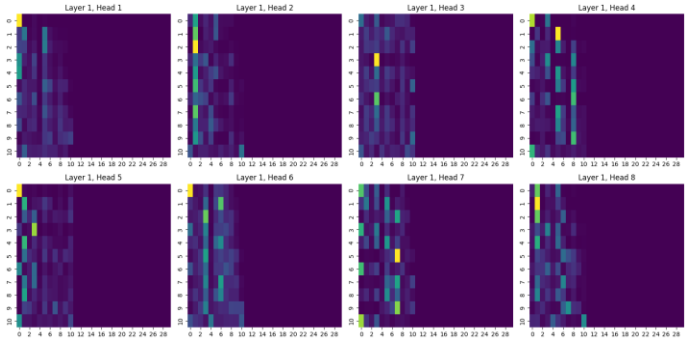


Epoch 10:

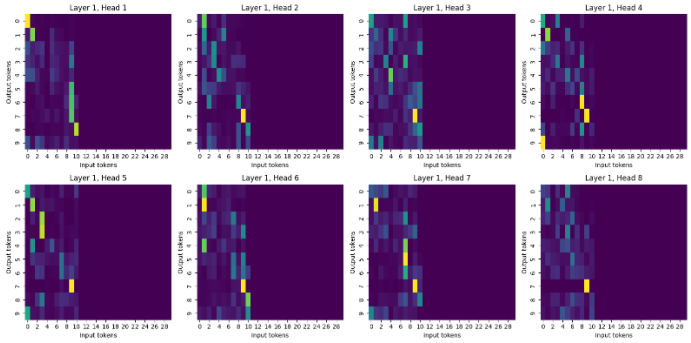


Decoder Layer2 Block2 (Decoder 의 Multi-head Attention, 두 번째 Layer)

Epoch 1:



Epoch 10:



분석 결과

1 번. Head 별 분석

Decoder 의 Multi-Head Self-Attention 과 비슷하게 다양한 어텐션 분포를 보인다. 각 Head 별 공통점 없이 서로 다른 특성들에 집중하고 있다.

어떤 부분들은 매트릭스의 초기 부분(예시. Head 1)에 상관관계가 집중되어 있거나, 하단 부분(예시. Head 7), 특정 지점, 전반적으로 균일한 분포를 가지는 것을 관찰할 수 있다.

2 번. Layer 별 분석

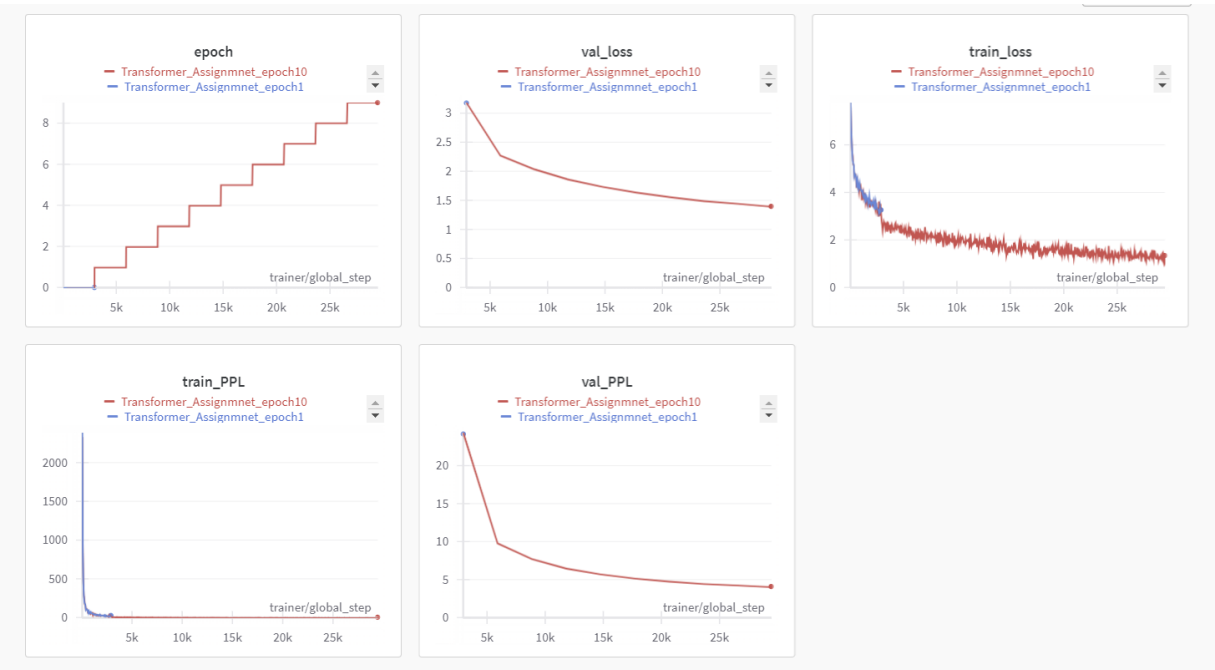
Layer1 은 전반적으로 몇몇 특정 입력 값에 집중되는 것 없이 균일한 분포를 보인다. Layer2 는 다른 조금 더 특정 입력 값에 선명하게 집중되는 것을 관찰할 수 있다.

3 번. 학습 수에 따른 분석

동일하게 학습 수가 많아질수록, 특정 부분에서 더 선명하게 가중치 분포가 나타나는 것을 볼 수 있다. 이는 특정 문맥이나 의미적 관계를 더욱 잘 파악하고 있다는 것이다.

Model Output(번역 결과)를 비교했을 때도, 학습을 수행할 수록 더 많은 단어들을 정확하게 예측한다는 것을 확인할 수 있다.

WANDB Loss 값 출력



2. Transformer 의 장점, 단점 (50 점)

* RNN 계열 모델, CNN 계열 모델에 대비 Transformer 의 장단점을 서술하시오.
* 지금까지한 실습, 과제를 기반해 근거를 제시하시오.

RNN Model 과 Transformer Model 을 비교해보면,
RNN Model 순차적 학습 Model 로써, 병렬 처리가 불가능하기 때문에 학습하는 데 시간이 오래 걸렸다. 이전 실습과 비교해보았을 때 Seq2Seq 의 Epoch 를 한 번 돌릴 때 10 분 이상 걸렸는데 Transformer Model 을 사용하니 Epoch 를 한 번 돌릴 때 1 분 미만으로 학습이 수행된다. Transformer 는 Attention 기법을 사용하여 입력 시퀀스의 모든 값들을 동시 처리할 수 있으므로 훨씬 빠른 추론이 가능하다.

CNN Model 과 Transformer Model 을 비교해보면,
CNN Model 은 병렬처리와 특정 지역에 초점을 맞추어 작동한다. 병렬처리가 가능하여 속도면에서는 빠르지만, Kernel 을 통해 특정 지역에만 의존하다 보니 긴 시퀀스에 대해서는 전반적인 문맥 파악이 어렵다. (멀리 떨어진 정보들에 대한 연관성을 파악하기엔 한계가 있다.)
Transformer 는 Self-Attention 을 사용하여 문장에서 하나의 입력 값이 다른 입력 값과 얼마나 많은 연관성을 가지는 지 알 수 있기에 문맥을 파악하는 데 더 좋은 성능을 보인다. (예를 들어, it 과 같이 문맥을 통해서만 파악할 수 있는 단어들에 대한 파악이 더욱 쉽다.)

Transformer Model 의 단점에 대해서 추측해보았을 때,
Model 의 구조만 보아도 많은 Layer(Self-Attention Layer, General-Attention Layer, Feed-Forward Layer, Add&Norm Layer)들이 사용되기 때문에 높은 메모리가 필요하다. 수많은 파라미터들이 존재하고, 시퀀스가 길수록 파라미터 수도 비례해서 늘어난다. 이를 통해 많은 GPU 를 사용하게 되며 Transformer Model 자체를 처음부터 직접 설계하는 것에는 한계가 있기 때문에 Pre-Trained Model 에 의존해야 된다는 단점이 있다.
또한, 현재는 max sequence 를 30 으로 설정하여 30 개의 단어까지만 처리를 수행하다보니, 이보다 더 긴 문장이 들어왔을 때 주요 정보들이 삭제되어 맥락을 제대로 파악할 수 없어 Model 의 성능이 급격하게 저하될 수 있다는 단점이 있다. Sequence 가 30 보다 훨씬 더 짧은 문장들은 [PAD] 토큰을 사용하여 단어를 추가하기에 모델의 학습 효율성을 저하시킬 수도 있다.