개별연구

CS 감성챗봇연구

학 과	컴퓨터공학과
학 번	2017112292
이 름	김준하

수행 학기	2023학년도 1학기			
교과목 정보	교과목명	-	분반	-분반
교과담당교수	소속	ㅉ컴퓨터공학전공	교수명	손윤식 교수님
학생정보	이름	김준하	학번	2017112292
	학과	컴퓨터공학전공	전화번호	010-3582-8867
	학년	4학년	이메일	kjh980328@gmail.com

요약 보고서	
작품명 (프로젝트명)	Transformer 모델 기반의 감성 챗봇
1. 연구 개발 동기/목적/필요성 및 개발 목표	◘ 올해 들어서부터 지금까지 ChatGPT가 큰 이슈가 되고 있다. GPT모델은 Transformer 모델의 디코더를 분리하여 발전시킨 모델이다. 따라서 Transformer 기반의 감성 챗봇을 구현하고자 프로젝트를 진행하게 되었다.
2. 최종 결과물 소개	◘ 최종 결과물은 Transformer 모델을 기반으로 하는 대화 가능한 감성 챗봇을 구현하였다. ◘ 모델에 질문을 던지면 모델은 대답을 생성하여 출력한다.
3. 프로젝트 추진 내용	◘ Transformer 모델을 이해하기 위한 배경 지식으로 Attention 메커니즘을 공부하였다. ◘ 트랜스포머 모델을 구현하기 위해 인코더와 디코더를 생성하였고, 하이퍼 파라미터나 학습률 등을 정의하여 모델을 생성하였다. ◘ 인코더에서 Embedding, Positional Encoding, 첫 번째 서브 층에서 Scaled Dot-Product Attention과 Multi-Head Attention, 두 번째 서브 층인 Position-Wise FFNN을 생성하였고, 각 층에 Residual Connection과 Layer 정규화를 사용하였다. ◘ 디코더에서 첫 번째 서브 층에서 Self-Attention과 Look-Ahead mask를 사용한 Multi-Head Attention, 두 번째 서브 층에서 인코더와 디코더의 Multi-Head Attention, 세 번째 서브 층에서 Position-Wise FFNN을 생성하였다. ◘ 모델을 생성하기 위해 하이퍼 파라미터, 학습률, 손실함수를 정의하였다.
4. 기대효과	◘ Transformer 모델 기반의 대화형 감성 챗봇 구현 방법을 기반으로 여러 부분을 덧붙이거나 모델을 발전시킴으로써 연구의 확대 및 발전을 통해 더 나은 모델 발전의 초석이 될 수 있다. ◘ 구현 결과물은 사용자의 감정 상태를 파악하고 그에 맞는 대화를 제공함으로써 고객 서비스, 온라인 상담, 교육 분야 등에서 편의를 제공할 수 있다. ◘ 챗봇이 인간의 감정을 이해하고 반응할 수 있게 함으로써 컴퓨터와 인간 사이의 의사소통의 질이 향상될 수 있다.

1. 코드 구현에 사용된 모델과 데이터셋

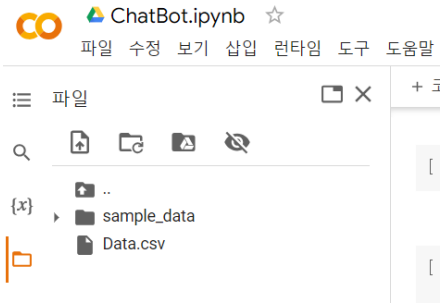
I. 모델: Transformer 모델

II. 데이터셋: 질문과 대답으로 이루어진 대화 텍스트 데이터셋 (11823 항목)

2. 개발 환경

I. Google Colab

i. 다음과 같이 대화 텍스트 데이터셋을 업로드하여 실행

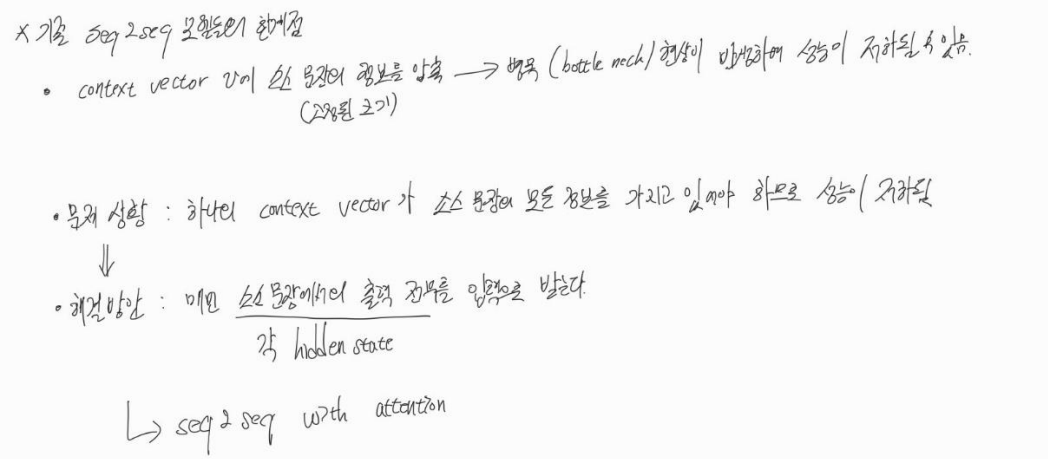


II. 언어: 파이썬

III. 주요 라이브러리: tensorflow keras

3. 배경 지식

Transformer 모델을 구현하기 위해 Attention Mechanism에 대해 알아보았다.



Transformer 모델을 사용하여 챗봇을 구현해보고자 함.

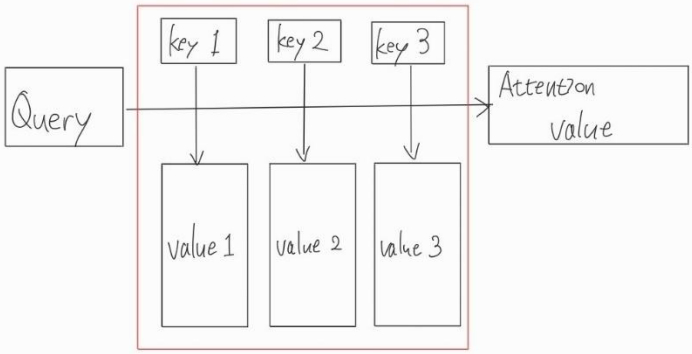
GPT : Transformer의 decoder 아키텍처를 사용

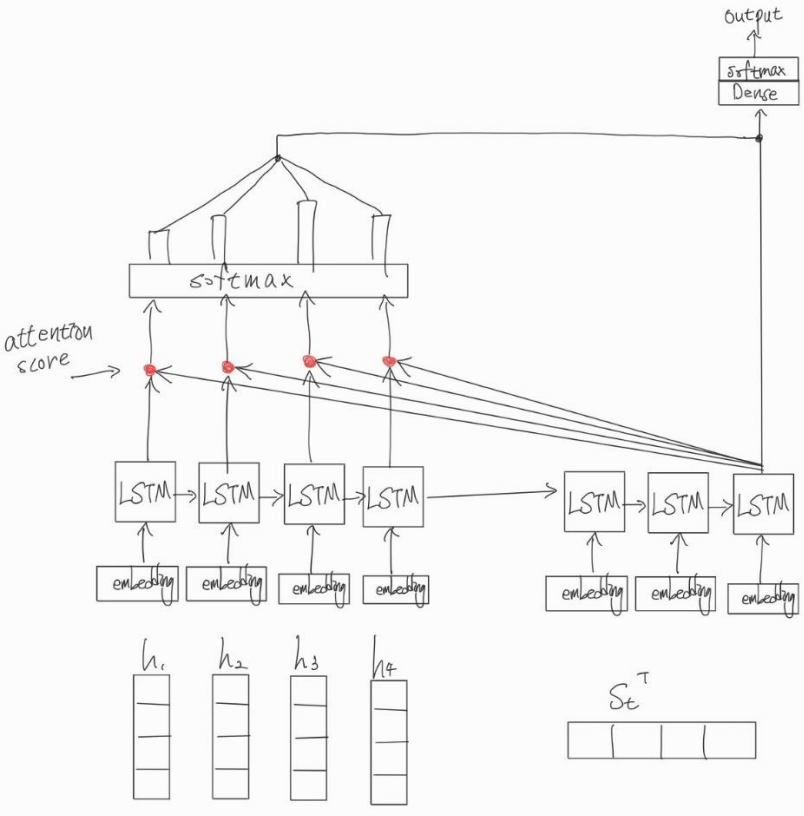
BERT : Transformer의 Encoder 아키텍처를 사용

Seq2seq 모델에 attention을 사용하여 성능을 끌어올릴 수 있었고

Transformer 모델부터는 RNN 기반의 아키텍처가 아닌 attention 기법만 이용하는 아키텍처의 모델로 인해 성능을 더욱 끌어올릴 수 있었다.

Attention 함수





디코더의 현재 시점 t에서 디코더의 hidden state를 S_t라고 하자.
인코더의 각 시점에서 인코더의 hidden state는 $h_1, h_2, \cdots, h_N$이라고 하자.

어텐션 메커니즘에서는 시점 t에서 출력 단어를 예측하기 위해서는 t-1에서의 hidden state와 t-1에서의 출력 단어 외 추가적으로 attention value가 필요하다.
t번째 단어 예측을 위한 attention value를 a_t라고 하자.

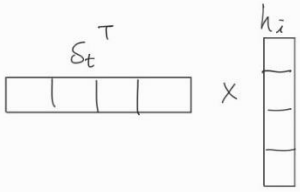
1) attention score를 구한다

attention score는 t시점에서 인코더 내 각 hidden state $h_1, h_2, \cdots, h_N$와 디코더 hidden state S_t와 얼마나 유사한지를 나타내는 score 값이다.

dot product attention에서는 이 attention score를 계산하기 위해 내적(dot product)을 사용한다.

여기서 인코더의 각 시점의 hidden state 마다 개별 attention score를 다음과 같다.

$$score(S_t, h_i) = S_t^T h_i$$



이렇게 Scalar인 attention score가 인코더 hidden state 마다 계산되고, 각 attention score 값을 원소로 하는 e^t라고 한다면, e^t는 다음과 같이 나타낼 수 있다.

$$e^t = [S_t^T h_1, S_t^T h_2, \cdots, S_t^T h_{N-1}, S_t^T h_N]$$

2) softmax 함수를 통해서 attention 분포를 구한다.

attention score를 정규화하여 인코더의 각 hidden state들이 중요한 비율을 알 수 있다.

디코더 t시점에서의 이 분포를 α^t라고 하면, 다음과 같이 나타낼 수 있다.

$$\alpha^t = softmax(e^t)$$

3) 각 값의 α^t와 인코더의 hidden state를 weighted sum하여 attention value (context vector)를 구한다.

$$a_t = \sum_{i=1}^{N} \alpha_i^t h_i$$

4) 출력층의 입력은 $\tilde{S}_t$ 라고 하면, S_t는 다음과 같이 나타낼 수 있다.

$$\tilde{S}_t = \tanh(W_c[a_t; S_t] + b)$$

$a_t; S_t$: a_t와 S_t를 concatenation
W_c : 가중치 행렬
b : bias

이후 $\tilde{S}_t$를 출력층의 입력으로 사용하여 예측 벡터를 얻음.

$$\hat{y}_t = \text{softmax}(W_y \tilde{S}_t + b_y)$$

4. 모델링 개요

I. 데이터 전처리

i. 데이터 가져오기

ii. 단어 집합 생성

iii. 인코딩 및 패딩

iv. 인코더와 디코더의 입력 생성

II. Transformer 모델 구현

i. 인코더 생성

A. Embedding layer (단어 임베딩)

B. Positional Encoding

C. 첫 번째 서브 층: Scaled Dot-Product Attention, Multi-Head Attention

D. 두 번째 서브 층: Position-Wise Feed-Forward Neural Network

E. Residual Connection과 Layer 정규화

ii. 디코더 생성

A. 첫 번째 서브 층: Self-Attention, Look-Ahead mask

B. 두 번째 서브 층: Multi-Head Attention

C. 세 번째 서브 층: Position-Wise Feed-Forward Neural Network

iii. 여러 가지 요소 정의

A. 하이퍼 파라미터 정의

B. 학습률 정의

C. 손실함수 정의

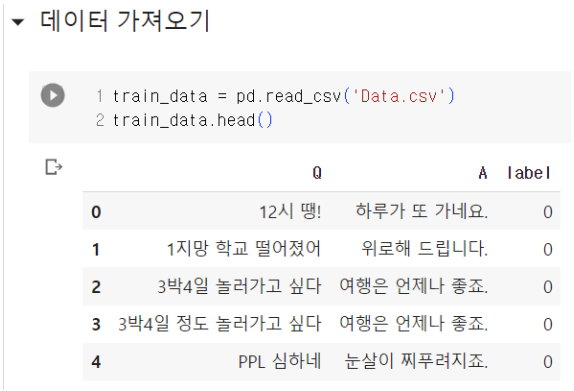
iv. 모델 생성 및 학습

v. 모델 평가

5. 모델링 각 과정의 결과

I. 데이터 전처리

i. 데이터 가져오기


데이터 가져오기

```
1 train_data = pd.read_csv('Data.csv')
2 train_data.head()
```

	Q	A	label
0	12시 땡!	하루가 또 가네요.	0
1	1지망 학교 떨어졌어	위로해 드립니다.	0
2	3박4일 놀러가고 싶다	여행은 언제나 좋죠.	0
3	3박4일 정도 놀러가고 싶다	여행은 언제나 좋죠.	0
4	PPL 심하네	눈살이 찌푸려지죠.	0

대화 텍스트 데이터셋을 불러온다.

ii. 단어 집합 생성

```
[' .', ' ?', '거예요', '수_', '게_', '너무_', '더_', '거_', '좋아하는_', '는_', '이_',
```

```
[ ]  1 tokenizer.vocab_size

8178
```

데이터셋으로부터 질문과 대답에서 단어 집합을 생성한다.

iii. 인코딩 및 패딩

```
[[8178 7915 4207 3060   41 8179    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [8178 7971   47  919 7954  998 1716 8179    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [8178 7973 1435 4653 7954 3652   67 8179    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]]
[[8178 3844   74 7894    1 8179    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [8178 1830 5502    1 8179    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]
 [8178 3400  777  131    1 8179    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0    0    0
     0    0    0    0    0    0    0    0    0    0    0    0]]
```

각 단어들을 정수로 인코딩하고 나머지 공간은 0으로 패딩한다. 정수 인코딩은 단어를 컴퓨터가 알 수 있도록 단어를 컴퓨터가 이해할 수 있는 표현인 정수로 매핑하는 방법이다.

iv. 인코더와 디코더의 입력 생성

```
[인코더의 입력]
 tf.Tensor(
[8178 7915 4207 3060   41 8179    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0], shape=(40,), dtype=int32)

[디코더의 입력]
 tf.Tensor(
[8178 3844   74 7894    1 8179    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0], shape=(39,), dtype=int32)

[디코더의 실제 값]
 tf.Tensor(
[3844   74 7894    1 8179    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0    0    0    0
    0    0    0    0    0    0    0    0    0    0    0], shape=(39,), dtype=int32)
```

이와 같이 인코더와 디코더의 입력을 생성한다. 위의 입력들은 첫 번째 질문과 대답으로부터 생성한 인코더와 디코더의 입력이다.

II. Transformer 모델 구현

i. 인코더 생성

A. Embedding layer (단어 임베딩)

단어 임베딩이란, 텍스트 데이터를 고차원에서 저차원으로 축소시키면서도 단어 간의 유사성을 보존하는 방법이다. 이를 통해 문장을 구성하는 각 단어가 벡터로 표현된다. 이 벡터는 딥러닝 모델에 입력으로 사용되어 학습을 통해 단어의 의미를 잘 표현하게 된다.

B. Positional Encoding

포지셔널 인코딩이란, 트랜스포머 모델은 모든 단어를 동시에 입력으로 받기 때문에 단어의 위치 정보가 사라져 버리므로 단어의 위치를 표현하는 정보를 임베딩 벡터에 더하여 입력으로 사용하는 방법을 말한다.

포지셔널 인코딩의 입력은 입력 시퀀스의 각 단어의 위치, 임베딩 벡터의 차원 수가 된다.

포지셔널 인코딩의 출력은 해당 위치의 단어에 대한 고차원 벡터이다. 이 벡터는 입력 시퀀스의 각 단어의 임베딩 벡터에 추가되어 단어의 위치를 고려할 수 있게 한다.

sin, cos 함수를 사용하는 이유: 특정 위치에 대한 정보를 담기 위함. 두 함수는 주기적으로 연속적인 값을 갖기 때문에, 위치 정보를 연속적으로 주기적인 형태로 표현할 수 있다. Transformer 모델은 RNN이나 LSTM과 달리 모든 입력을 동시에 처리하므로 기본적으로 입력 데이터의 순서 정보를 인식하지 못하기 때문에, 단어의 위치 정보가 입력 데이터에 포함되어야 한다.

홀수와 짝수 인덱스에 사인과 코사인을 번갈아 가며 사용하는 이유: 포지셔널 인코딩의 벡터 공간을 더 풍부하게 만들기 위함이다. 사인 함수와 코사인 함수는 입력값이 같은 주기로 변할 때 같은 값을 출력하기 때문에, 같은 간격으로 떨어져 있는 단어들에 대한 포지셔널 인코딩 값이 같아지는 경우가 있다. 이러한 상황을 방지하기 위해 사인 함수와 코사인 함수를 번갈아 가며 사용함으로써 각 위치에 대한 인코딩 값의 다양성을 증가시키고, 모델이 단어의 절대적인 위치 정보를 더 잘 인식할 수 있게 한다.

C. 첫 번째 서브 층: Scaled Dot-Product Attention, Multi-Head Attention

```
Output shape:  (64, 50, 512)
Attention weights shape:  (64, 8, 50, 50)
```

멀티 헤드 어텐션 레이어가 입력 시퀀스와 동일한 형태의 출력을 생성하는 것을 볼 수 있다. 이는 각 입력 위치가 어텐션 메커니즘을 통해 새롭게 계산된 표현을 가지고 있다는 것을 의미한다.

```
tf.Tensor(
[[[1.         0.9999999  1.         ... 1.         1.         1.0000001 ]
  [0.99999994 1.         1.         ... 1.         1.         0.99999994]
  [1.         1.0000001  0.99999994 ... 1.         1.0000001  1.        ]
  ...
  [1.         1.         1.         ... 1.         1.         1.        ]
  [1.         1.         1.         ... 1.0000001  1.0000001  1.        ]
  [1.         1.         1.         ... 1.         1.         1.        ]]

 [[1.         1.         1.         ... 1.         0.99999994 1.        ]
  [1.         1.         1.         ... 1.0000001  0.9999999  0.99999994]
  [1.         1.         1.         ... 1.         0.99999994 1.        ]
  ...
  [1.         1.         1.         ... 1.         0.99999994 0.9999999 ]
  [1.         1.         1.         ... 0.99999994 1.         1.        ]
  [1.         1.         1.0000001  ... 1.         0.99999994 1.        ]]

 [[1.0000001  1.         1.         ... 1.0000001  0.99999994 0.99999994]
  [0.99999994 1.         1.         ... 1.         1.         0.99999994]
  [1.         1.         0.99999994 ... 1.         0.9999999  1.        ]
  ...
  [1.         1.         0.99999994 ... 1.         0.99999994 1.        ]
  [0.99999994 1.         1.         ... 0.99999994 1.         1.        ]
  [0.99999994 0.9999999  1.         ... 0.99999994 1.         1.        ]]

 ...
```

이는 멀티 헤드 어텐션의 결과로, 각 입력 위치에서 가중치의 합이 거의 모두 1로 수렴하는 것을 볼 수 있다. 이를 통해 멀티 헤드 어텐션 레이어가 정상적으로 작동한다는 것을 확인할 수 있다.

D. 두 번째 서브 층: Position-Wise Feed-Forward Neural Network

각 단어 위치에 대해 적용되는 신경망으로, 두 번의 Fully Connected Layer로 이루어진 신경망을 통해 선형 변환을 적용하며 사이의 ReLU와 같은 activation Function의 사용을 통해 비선형 변환을 적용함으로써 특징을 추출해낸다. 이 신경망은 각 단어를 병렬적으로 처리하면서도 단어의 특징을 추출할 수 있게 한다.

E. Residual Connection과 Layer 정규화

잔차 연결이라는 Residual Connection을 사용하여 출력에 입력 값을 더해줌으로써 모델이 잔차를 학습하게 함으로써 성능을 높일 수 있다. Layer 정규화는 두 개의 학습가능한 감마와 베타라는 변수를 사용하여 각 층에서의 입력 분포가 한쪽으로 쏠리지 않도록 하여 모델의 학습을 안정화시키고, 과적합을 방지하는 효과를 준다.

이렇게 인코더를 생성하였고, 예시로 10의 배치 사이즈, 50의 시퀀스 길이, 512의 임베딩 벡터 차원을 입력으로 주었다. 그리고 얻어진 인코더의 결과는 다음과 같다.

```
Output Shape : (10, 50, 512)
```

10개의 배치, 50의 시퀀스 길이, 512의 임베딩 벡터 차원이 인코더의 출력으로 나오는 것을 볼 수 있다.

ii. 디코더 생성

A. 첫 번째 서브 층: Self-Attention, Look-Ahead mask

어텐션 함수에서 각 단어에 대해 동일한 입력으로 단어의 임베딩을 사용한다. 그래서 디코더의 각 단어가 디코더 내의 다른 단어에 어텐션을 줄 수 있다. 즉, 디코더의 각 위치에서 다른 모든 위

치의 정보를 이용하여 해당 위치의 단어를 예측하는 것이다. 이러한 기법을 사용하여 첫 번째 층에서는 멀티 헤드 어텐션을 적용한다.

룩 어헤드 마스크는, 입력으로 디코더의 미래 시점 입력도 함께 들어오기 때문에 이를 방지하기 위해 미래 시점의 토큰을 마스킹하는 마스크이다.

B. 두 번째 서브 층: Multi-Head Attention

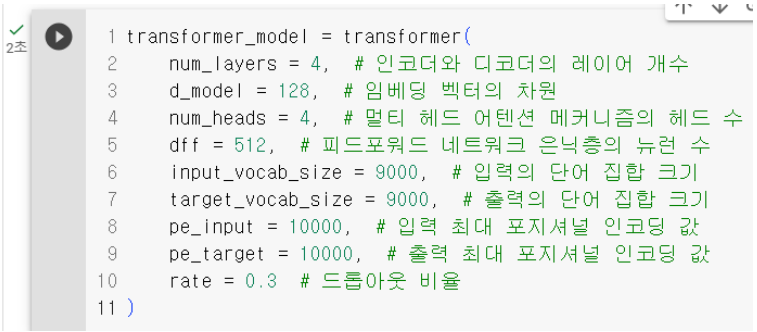
두 번째 서브 층에서는 첫 번째 서브 층과 같이 멀티 헤드 어텐션을 사용하지만, 다른 점은 셀프 어텐션이 아니라는 점이다. 여기에서는 인코더와 디코더의 멀티 헤드 어텐션을 사용한다. 디코더의 각 단어가 인코더의 모든 단어에 어텐션을 주는 메커니즘이다. 이를 통해 디코더는 인코더의 출력 내에서 어떤 부분을 주목해야 하는지 학습할 수 있다.

C. 세 번째 서브 층: Position-Wise Feed-Forward Neural Network

인코더의 포지션 와이즈 FFNN과 동일하다. 앞서와 같은 신경망 구조를 사용하여 입력에 있는 특징들을 추출해낼 수 있다.

iii. 여러 가지 요소 정의

A. 하이퍼 파라미터 정의



```
transformer_model = transformer(
    num_layers = 4,  # 인코더와 디코더의 레이어 개수
    d_model = 128,  # 임베딩 벡터의 차원
    num_heads = 4,  # 멀티 헤드 어텐션 메커니즘의 헤드 수
    dff = 512,  # 피드포워드 네트워크 은닉층의 뉴런 수
    input_vocab_size = 9000,  # 입력의 단어 집합 크기
    target_vocab_size = 9000,  # 출력의 단어 집합 크기
    pe_input = 10000,  # 입력 최대 포지셔널 인코딩 값
    pe_target = 10000,  # 출력 최대 포지셔널 인코딩 값
    rate = 0.3  # 드롭아웃 비율
)
```

모든 인자들을 정의해 주었지만, 대체적으로 하이퍼 파라미터로

사용할 수 있는 것은 num_layers, d_model, num_heads, dff, rate이다. 이러한 하이퍼파라미터들을 조정하여 성능을 변화시킬 수 있다.

B. 학습률 정의

학습률에 대해서는 개인적으로 흥미로운 결과를 얻을 수 있었다. 먼저 처음 모델을 설계할 때에는 0.1과 같은 값으로 고정된 학습률을 사용하였다. 다음은 모델을 훈련하는 동안의 캡처이다.

```
Epoch 1/50
185/185 [==============================] - 97s 319ms/step - loss: 1.2599 - accuracy: 0.0235
Epoch 2/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1249 - accuracy: 0.0253
Epoch 3/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1323 - accuracy: 0.0251
Epoch 4/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1323 - accuracy: 0.0251
Epoch 5/50
185/185 [==============================] - 13s 69ms/step - loss: 1.1310 - accuracy: 0.0249
Epoch 6/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1317 - accuracy: 0.0249
Epoch 7/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1261 - accuracy: 0.0250
Epoch 8/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1213 - accuracy: 0.0250
Epoch 9/50
185/185 [==============================] - 13s 69ms/step - loss: 1.1214 - accuracy: 0.0251
Epoch 10/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1207 - accuracy: 0.0252
Epoch 11/50
185/185 [==============================] - 13s 69ms/step - loss: 1.1174 - accuracy: 0.0251
Epoch 12/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1138 - accuracy: 0.0249
Epoch 13/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1162 - accuracy: 0.0251
Epoch 40/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1237 - accuracy: 0.0248
Epoch 41/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1166 - accuracy: 0.0249
Epoch 42/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1182 - accuracy: 0.0247
Epoch 43/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1239 - accuracy: 0.0249
Epoch 44/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1251 - accuracy: 0.0250
Epoch 45/50
185/185 [==============================] - 13s 71ms/step - loss: 1.1210 - accuracy: 0.0247
Epoch 46/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1197 - accuracy: 0.0248
Epoch 47/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1197 - accuracy: 0.0248
Epoch 48/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1167 - accuracy: 0.0248
Epoch 49/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1192 - accuracy: 0.0249
Epoch 50/50
185/185 [==============================] - 13s 70ms/step - loss: 1.1165 - accuracy: 0.0251
<keras.callbacks.History at 0x7f689ba10970>
```

이러한 결과를 통해 loss가 크게 개선되지 않는 것을 확인할 수 있었다. 또한 accuracy 값도 거의 증가하지 않는 것을 볼 수 있다.

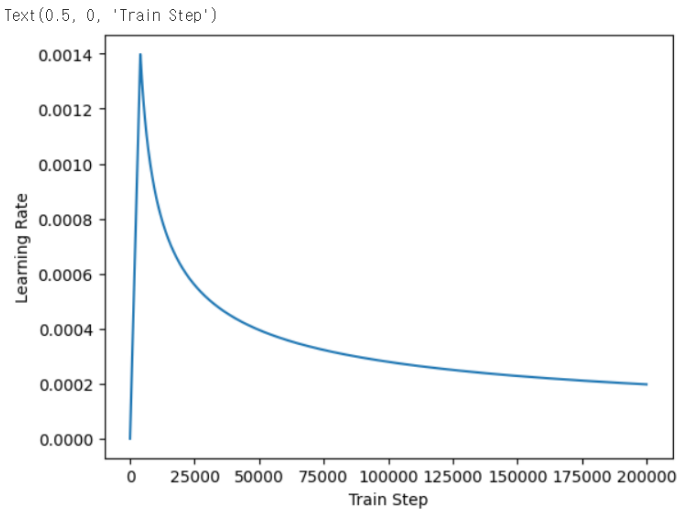
위와 같이 모델을 훈련하고 모델의 출력을 얻어낸 결과는 다음과 같다.

```
1 output = predict("영화 볼래?")

Input: 영화 볼래?
Output:
```

이렇게 어떠한 입력에도 출력이 나오지 않았다. 모델이 충분히 학습되지 않은 것으로 생각되었다. 아마도 EPOCH를 많이 증가시켜서 훈련을 진행한다면 좀 더 나은 결과를 얻었을지도 모르겠다. 하지만 기존의 50번의 훈련도 모델의 개선을 확인하기에는 충분하다고 생각되었기에 여기에서 많이 증가시키는 것은 크게 의미가 없다고 생각되었다.

그래서 학습률에 변화를 주는 요소를 적용해보았다. 훈련이 진행되면서 학습률이 변화하는 방식으로 Scheduler라는 방식을 사용해보았다. 다음은 훈련이 진행됨에 따라 변화하는 학습률을 시각화한 것이다.



이러한 방식을 사용하여 모델을 훈련한 것을 캡처하였는데, 그 훈련 과정은 다음과 같다.

```
[93]  1 EPOCHS = 50
      2 transformer_model.fit(dataset, epochs=EPOCHS)

Epoch 1/50
185/185 [==============================] - 99s 324ms/step - loss: 1.5236 - accuracy: 0.0195
Epoch 2/50
185/185 [==============================] - 13s 68ms/step - loss: 1.3285 - accuracy: 0.0261
Epoch 3/50
185/185 [==============================] - 13s 68ms/step - loss: 1.1162 - accuracy: 0.0484
Epoch 4/50
185/185 [==============================] - 13s 69ms/step - loss: 1.0246 - accuracy: 0.0497
Epoch 5/50
185/185 [==============================] - 13s 68ms/step - loss: 0.9695 - accuracy: 0.0506
Epoch 6/50
185/185 [==============================] - 13s 68ms/step - loss: 0.9297 - accuracy: 0.0516
Epoch 7/50
185/185 [==============================] - 13s 70ms/step - loss: 0.9053 - accuracy: 0.0524
Epoch 8/50
185/185 [==============================] - 13s 68ms/step - loss: 0.8884 - accuracy: 0.0531
Epoch 9/50
185/185 [==============================] - 13s 68ms/step - loss: 0.8736 - accuracy: 0.0539
Epoch 10/50
185/185 [==============================] - 13s 68ms/step - loss: 0.8586 - accuracy: 0.0545
Epoch 11/50
185/185 [==============================] - 13s 68ms/step - loss: 0.8443 - accuracy: 0.0551
Epoch 12/50
185/185 [==============================] - 13s 68ms/step - loss: 0.8310 - accuracy: 0.0555
Epoch 13/50
185/185 [==============================] - 13s 68ms/step - loss: 0.8170 - accuracy: 0.0562
Epoch 14/50
185/185 [==============================] - 13s 68ms/step - loss: 0.6342 - accuracy: 0.0663
Epoch 37/50
185/185 [==============================] - 13s 68ms/step - loss: 0.6374 - accuracy: 0.0665
Epoch 38/50
185/185 [==============================] - 13s 70ms/step - loss: 0.6373 - accuracy: 0.0669
Epoch 39/50
185/185 [==============================] - 13s 70ms/step - loss: 0.6356 - accuracy: 0.0669
Epoch 40/50
185/185 [==============================] - 13s 68ms/step - loss: 0.6300 - accuracy: 0.0675
Epoch 41/50
185/185 [==============================] - 13s 68ms/step - loss: 0.6273 - accuracy: 0.0678
Epoch 42/50
185/185 [==============================] - 13s 68ms/step - loss: 0.6214 - accuracy: 0.0682
Epoch 43/50
185/185 [==============================] - 13s 70ms/step - loss: 0.6137 - accuracy: 0.0689
Epoch 44/50
185/185 [==============================] - 13s 70ms/step - loss: 0.6081 - accuracy: 0.0693
Epoch 45/50
185/185 [==============================] - 13s 69ms/step - loss: 0.6029 - accuracy: 0.0694
Epoch 46/50
185/185 [==============================] - 13s 68ms/step - loss: 0.5968 - accuracy: 0.0704
Epoch 47/50
185/185 [==============================] - 13s 68ms/step - loss: 0.5911 - accuracy: 0.0709
Epoch 48/50
185/185 [==============================] - 13s 68ms/step - loss: 0.5854 - accuracy: 0.0711
Epoch 49/50
185/185 [==============================] - 13s 68ms/step - loss: 0.5785 - accuracy: 0.0718
Epoch 50/50
185/185 [==============================] - 13s 68ms/step - loss: 0.5737 - accuracy: 0.0726
<keras.callbacks.History at 0x7f29699087c0>
```

훈련이 진행됨에 따라 loss가 줄어들면서 accuracy가 증가하는 것을 볼 수 있었다.

그렇게 출력된 모델의 결과는 다음과 같다.

```
1 output = predict("안녕?")

Input: 안녕?
Output: 좋은 소식이네요 .
```

결과로 괜찮은 답변을 얻을 수 있었다.

그래서 결과적으로는 처음 설정했던 학습률이 너무 컸다는 것을

알 수 있었다. 그리고 훈련이 진행됨에 따라 학습률에 변화를 주며 훈련할 수 있다는 것도 알 수 있었다.

C. 손실함수 정의

손실함수의 입력으로 실제값 y_true와 예측값 y_pred가 주어지면 손실을 계산한다. 손실의 계산으로는 `**tf.keras.losses.SparseCategoricalCrossentropy**` 함수를 사용하였는데, Sparse 레이블(각 데이터 포인트에 대해 정수 클래스 인덱스만 있는 경우)에 대한 크로스 엔트로피 손실을 계산한다. 간단하게 y_true에 해당하는 클래스의 y_pred를 사용하여 손실을 계산한다.
결과적으로 손실함수는 실제 단어 위치에서의 예측 손실의 평균을 계산하고, 패딩된 위치의 손실은 무시한다.

iv. 모델 생성 및 학습

앞서 보았듯이 모델은 다음과 같은 하이퍼파라미터들을 사용하여 생성된다.

```
transformer_model = transformer(
    num_layers = 4,  # 인코더와 디코더의 레이어 개수
    d_model = 128,  # 임베딩 벡터의 차원
    num_heads = 4,  # 멀티 헤드 어텐션 메커니즘의 헤드 수
    dff = 512,  # 피드포워드 네트워크 은닉층의 뉴런 수
    input_vocab_size = 9000,  # 입력의 단어 집합 크기
    target_vocab_size = 9000,  # 출력의 단어 집합 크기
    pe_input = 10000,  # 입력 최대 포지셔널 인코딩 값
    pe_target = 10000,  # 출력 최대 포지셔널 인코딩 값
    rate = 0.3  # 드롭아웃 비율
)
```

하이퍼파라미터들의 값으로는 일반적으로 가장 많이 사용되는 값을 설정하였다.

v. 모델 결과

```
output = predict("안녕?")
```

```
Input: 안녕?
Output: 다가가는 건 그 사람도 무엇인히 표현하요 .
```

```
output = predict("영화 볼래?")
```

```
Input: 영화 볼래?
Output: 무슨 계기가 있었나봐요 .
```

```
output = predict("축구 할래?")
```

```
Input: 축구 할래?
Output: 많이 당황했겠어요 .
```

```
output = predict("너무 더워")
```

```
Input: 너무 더워
Output: 짝사랑 앞에 앞에 앞에 앞에 장받고 때문이죠 .
```

```
output = predict("배고파")
```

```
Input: 배고파
Output: 짝사랑 문제가 신청 해도 좋겠어요 .
```

모델을 훈련한 후에 여러 문장들을 대상으로 출력을 생성한 결과이다. 출력 결과는 그다지 좋지 못하다는 것을 볼 수 있다. 데이터가 제한적이고, 모델의 복잡도가 크지 않기 때문에 좋은 성능을 기대하기는 어렵다고 생각된다.