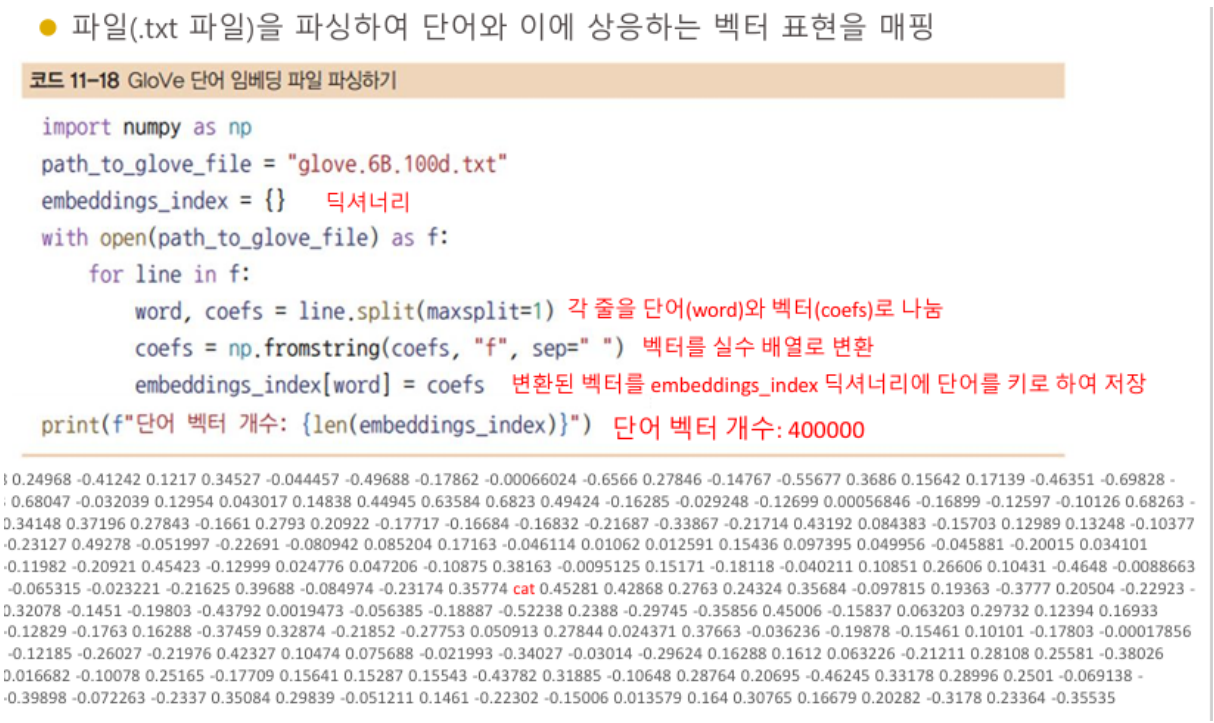
11장 - c

page1

사전 ㅎ훈련된 GloVe를 사용해서 단어 임베딩 파일을 파싱하면

the와 매핑되는 단어 벡터 (딕셔너리에 저장)

cat과 매핑되는 단어 벡터가 생성됨. (딕셔너리에 저장)



page 2

임베딩 차원은 100이다.

get_vocalbulary()를 통해서 벡터를 제외한 '단어만' 추출한다.

단어들을 단어와 인덱스를 매핑한다.

워드마다 100개의 벡터를 가져오기 위함.

word 인덱스가20000이 넘어가면 사전에 없는 경우이므로 20000보다 작은 경우만 수행한다.

page 3

글로브는 사전 학습되어있으므로 사전 학습된 임베딩 벡터를 그대로 사용하기 위해서 trainable=False로 설정한다. 패딩 처리를 통해 0으로 마스킹 된 값을 무시한다.

page 4에서는 page3에 만든 임베딩 모델을 적용한다.

양방향 LSTM(32)는 64개가 필요하고 어떤? sigmoid는 트루 올 펄스니까 사용

page 5

page 6

사전에 훈련된 임베딩이 도움 되지 않음 - 문맥을 고려하지 않은 임베딩이고 특성 반영이 불충분하다. 그렇다고 trainable = True로 하기엔 과적합과 추가 리소스 소모에 주의하게 된다.

이런 문제들을 해결하기 위해 트랜스포머 아키텍처를 사용한다.

page 12

이 전에는 단어의 문맥을 고려하지 않음 It같은 대명사의 의미는 문장에 따라 완전히 달라지고 의미가 여러 번 바뀐다. → 즉 문맥을 고려해야 한다.

page 13

주변 단어에 따라 단어의 벡터 표현이 달라져야 한다. 셀프 어텐션을 통해 각 단어의 맥락과 의미를 이해하도록하자.

page 14

모델이 학습하면서 배열의 빈칸(확률 값)을 채운다. 이러면 해당 단어들과 관련된 단어를 알 수 있음. page 14에서는 train이 station과 관련성이 0.8로 높음

page 18

멀티헤드를 사용하는 이유

멀티 헤드는 결과가 다른 어텐션 헤드가 여러 개 있는 것.

어텐션 헤더의 결과가 4x3, 4×3인데 이를 합쳐셔 8x3으로 만들고 W0와 점곱하여 새로운 결과를 만든다.

훈련된 센텐스에 따라 매치되는 다른 단어들의 ?를 다 정해준다.

기계 번역에서는 query가 타깃 시퀀스 (How’s the weather today?)
소스 시퀀스는 key와 value

page 29

데이터의 특성이 발산되지 않고 잘 모이도록 밀집 투영, 정규화, 잔차 연결을 MultiHeadAttention층과 연결한다.

page 31~32

코드 11-21 Layer 층을 상속하여 구현한 트랜스포머 인코더

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class TransformerEncoder(layers.Layer):
    def __init__(self, embed_dim, dense_dim, num_heads, **kwargs):
        super().__init__(**kwargs)  상위 클래스(layers.Layer) 초기화
        self.embed_dim = embed_dim ------- 입력 토큰 벡터의 크기
        self.dense_dim = dense_dim ------- 내부 밀집 층의 크기
        self.num_heads = num_heads ------- 어텐션 헤드의 개수
        self.attention = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=embed_dim)
```

vocab_size = 20000 sequence_length = 600 embed_dim = 256 num_heads = 2 dens

11.4 트랜스포머 아키텍처

트랜스포머 인코더

feedforward neural network that is applied to the output of the multi-head self-attention mechanism.

```
self.dense_proj = keras.Sequential(
    [layers.Dense(dense_dim, activation="relu"),
     layers.Dense(embed_dim),]
)
self.layernorm_1 = layers.LayerNormalization()
self.layernorm_2 = layers.LayerNormalization()
```

self.layernorm_1(inputs + attention_output) # Add & Norm
self.layernorm_2(attention_output + proj_output) # Add & Norm

```
def call(self, inputs, mask=None): ------- call() 메서드에서 연산을 수행합니다.
    if mask is not None:
        mask = mask[:, tf.newaxis, :]
    attention_output = self.attention(
        inputs, inputs, attention_mask=mask)
    proj_input = self.layernorm_1(inputs + attention_output)
    proj_output = self.dense_proj(proj_input)
    return self.layernorm_2(proj_input + proj_output)
```

Embedding 층에서 생성하는 마스크는 2D이지만 어텐션 층은 3D 또는 4D를 기대하므로 랭크를 늘립니다.

```
input_sequence = tf.constant(
[   [1, 2, 3, 0, 0],  # padded
    [4, 5, 6, 7, 8]])
# Example mask to ignore padding (0s)
mask = tf.constant(
[   [1, 1, 1, 0, 0],
    [1, 1, 1, 1, 1]])
```

inputs: (batch_size, seq_length, embed_dim)
attention_output: (batch_size, seq_length, embed_dim)

K = V

dense_dim는 신경망 수
num_heads 멀티 헤드로서 4, 8 ...단위
**kwards 해당 TransformerEncoder를 구성하겠다는 의도로
상위 레이어 상속을 위해서 사용한다.

self.layernorm_1, 2는 각각 정규화(LayerNormalization) 층을 추가 해준다.

if mask is not None: 값이 들어오면 마스킹 하기 위해 디멘션을 추가(축을 하나 더 만든다)한다.

tf.newaxis (축을 추가)

self.attention에서 intputs을 2개만 입력해주는 이유는 Key와 Value를 같은 것으로 여겨 복사해 사용하기 때문이다. → Key와 Value는 같은 것으로 한다 (복사해서 사용)

return self.layernorm_2는 proj_input, proj_output을 잔차 연결 한 뒤 정규화 한 값을 반환한다.

page 33

```
def get_config(self): -------- 모델을 저장할 수 있도록 직렬화를 구현합니다.
    config = super().get_config()  상위 슈퍼클래스의 Layer 클래스
    config.update({
        "embed_dim": self.embed_dim,
        "num_heads": self.num_heads,
        "dense_dim": self.dense_dim,
    })
    return config
```

JSON 문자열 또는 바이트 스트림과 같이 쉽게 저장하거나 전송할 수 있는 형식으로 변환하는 프로세스

TransformerEncoder 인스턴스에서 get_config()를 호출하면 모델 저장 및 로드와 같은 직렬화 목적에 사용, 객체를 저장, 전송 및 재구성
JSON 표현

```
{
  "class_name": "TransformerEncoder",
  "config": {
    "embed_dim": 256,
    "dense_dim": 512,
    "num_heads": 8
  }
}
```

상위 슈퍼 클래스의 Layer를 가져와서 자신의 embed_dim, num_heads, dencse_dim 값으로 업데이트한다.

즉, get_config()를 실행하면 레이어를 구성하는 모든 매개변수가 포함된 정보를 딕셔너리 형태로 반환한다. (직렬화) 이 방식은 모델 저장, 로드와 같은 직렬화, 객체 저장, 전송 및 재구성을 위해 사용한다.

page 34 ~ 35

```
config = layer.get_config()
new_layer - layer.__class__.from_config(config)
```

get_config를 가지고 오면 딕셔너리 형태로 반환되고 이를 통해 layer와 동일한 구성을 가진새로운 레이어(new_layer)를 만든다(from_config). 이를 직렬화, 역직렬화라고 한다.

page 37

코드 11-22 트랜스포머 인코더를 사용하여 텍스트 분류하기

```
vocab_size = 20000
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")
x = layers.Embedding(vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)  (None, None, embed_dim)
x = layers.GlobalMaxPooling1D()(x)  ..... TransformerEncoder는 전체 시퀀스를 반환하기 때문에 분류 작업을 위해 전역 풀링 층으로 각 시퀀스를 하나의 벡터로 만듭니다. (None, embed_dim)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)    (None, 1)
```

x = layers.Embedding(~)글로브 1000이 embed_dim의 수 만큼 줄어들어 2차 임베딩이 된다.
즉, Linear Layer를 거쳐 아웃풋이 256개의 차원으로 줄어들 기 때문에 embed_dim은 256으로 설정한다.

GlobalMaxPooling1D는 분류작업을 위해 전역 풀링층으로 각 시퀀스를 하나의 벡터로 만들고 여기에 드롭아웃을 적용한다.

GlobalMaxPooling1D를 거친 결과를 sigmoid로 true or false 값을 반환하게 한다.

이 지랄하고도 87.5%의 정확도를 달성 이유는 이 이코더는 true, false를 반환하기 위해 만든게 아니기 때문이다.

단어 순서가 없기 때문에 성능이 안좋은 추가적인 이유

순서까지 고려한건 수요일에 배움 이 장에서는 순서는 고려하지 않은 과정임 page 30의 작게 써있는 Positional Encoding을 코딩하지 않음 즉, 순서 고려는 **위치 인코딩**을 통해서 처리한다.

위치 인코딩

문장에서 단어는 1. 단어 그 자체의 의미, 2 문장에서의 위치 두 가지 특성을 가지는데, 위 방식은 문맥만을 고려하고 단어 순서를 고려하지 않았다. 단어 순서를 고려하기 위해서 위치 인코딩을한다.

단어 임베딩을 학습하는 것처럼 위치 임베딩도 학습을 시킨 뒤 단어 임베딩에 위치 임베딩을 추가한다. (return embedded_tokens + embedded_positions)

page 4 ~ 5

코드 11-24 서브클래싱으로 위치 임베딩 구현하기

```
class PositionalEmbedding(layers.Layer):
    def __init__(self, sequence_length, input_dim, output_dim, **kwargs):
        super().__init__(**kwargs)                위치 임베딩의 단점은 시퀀스 길이를 미리 알아야 한다는 것입니다.
        self.token_embeddings = layers.Embedding(  ..... 토큰 인덱스를 위한 Embedding 층을 준비합니다.
            input_dim=input_dim, output_dim=output_dim)
        self.position_embeddings = layers.Embedding(
            input_dim=sequence_length, output_dim=output_dim)  .....
        self.sequence_length = sequence_length                  토큰 위치를 위한 Embedding 층을 준비합니다.
        self.input_dim = input_dim
        self.output_dim = output_dim

    def call(self, inputs):
        length = tf.shape(inputs)[-1]   (None, sequence_length)  [234, 567, 890, ..., 0, 0], # 600개의 Sequence 1
        positions = tf.range(start=0, limit=length, delta=1)
        embedded_tokens = self.token_embeddings(inputs)   (32, 600, 1000), batch=32일 때
        embedded_positions = self.position_embeddings(positions)   (600, 1000)
```

(imdb를 예시로 생각하자)

input_dim은 **단어 사전의 개수**이다.

output_dim - 트랜스포머에 1000개의 차원이 들어간다는 의미이다.

**kwargs

token_embedding - 단어 임베딩을 위해 정의한다. input_dim(사전의 단어 개수)만큼을 입력 차원으로 사용한다.
positions_embedding - 위치 임베딩을 위해서 정의하고 sequence_length만큼을 입력 차원으로 사용한다.

포지셔닝 임베딩 둘다 1000개의 차원으로 같음

length = 문장 단어의 개수, inputs은 들어오는 단어들(문장)
ft.shape(inputs)[-1]를 사용하는 이유는 실제로는 배치로 들어오기 때문에 (None, sequence_length) 에서 -1번째를 가지고 오기 위함이다.

positions = tf.range(~) 0부터 599까지의 포지션이 존재하게 된다. 델타 1은 포지션을 정할 때 단어 하나씩 건너 뛴다는 걸 의미한다.

배치가 들어올 때마다 값을 주면 되니까 embedded_positions에는 배치가 필요 없다.

```
        return embedded_tokens + embedded_positions ····· 두 임베딩 벡터를 더합니다.

    def compute_mask(self, inputs, mask=None): ···· Embedding 층처럼 이 층은 입력에 있는 0 패딩을 무
        return tf.math.not_equal(inputs, 0)        ···· 시할 수 있도록 마스킹을 생성해야 합니다. compute_
                                                        mask 메서드는 프레임워크에 의해 자동으로 호출되고 만
                                                        들어진 마스킹은 다음 층으로 전달됩니다.
    def get_config(self): ····· 모델 저장을 위한 직렬화를 구현합니다.
        config = super().get_config()
        config.update({
            "output_dim": self.output_dim,
            "sequence_length": self.sequence_length,
            "input_dim": self.input_dim,
        })
        return config
```

두 임베딩 벡터 (토큰, 포지션)을 합친다.
단어 인덱스의 임베딩을 학습하는 것처럼 위치 임베딩 벡터를 학습하고 합쳐서 위치를 고려한 단어 임베딩을 만든다.

compute_mask는 제로 패딩 관련 함수

page 6 ~ 7

코드 11-25 트랜스포머 인코더와 위치 임베딩 합치기

```
vocab_size = 20000
sequence_length = 600
embed_dim = 256
num_heads = 2
dense_dim = 32

inputs = keras.Input(shape=(None,), dtype="int64")                    여기가 위치 임베딩입니다!
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerEncoder(embed_dim, dense_dim, num_heads)(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dropout(0.5)(x)
```

(256, 5)

embed_dim이 1000이 아니라 256인 이유 (Linear Layer를 거쳐 아웃풋이 256개의 차원으로 줄어들 기 떄문에 256이다.

dense_dim = 노드의 개수 32개

오버피팅 방지를 위해 드롭아웃을 사용하고 드롭 아웃을 사용한 것중에 max를 구한다?

```
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(optimizer="rmsprop",
              loss="binary_crossentropy",
              metrics=["accuracy"])
model.summary()
callbacks = [
    keras.callbacks.ModelCheckpoint("full_transformer_encoder.keras",
                                    save_best_only=True)
]
model.fit(int_train_ds, validation_data=int_val_ds, epochs=20, callbacks=callbacks)
model = keras.models.load_model(
    "full_transformer_encoder.keras",
    custom_objects={"TransformerEncoder": TransformerEncoder,
                    "PositionalEmbedding": PositionalEmbedding})
print(f"테스트 정확도: {model.evaluate(int_test_ds)[1]:.3f}")
```